
PetaPerl documentation

Experimental preview — draft documentation 0.6.0

PetaMem, s.r.o.

Jul 12, 2026

USER GUIDE

1	Introduction	3
1.1	Goals	3
1.2	Architecture Overview	4
1.3	Quick Start	5
1.4	Compatibility	5
1.5	Platform Support	5
2	Getting Started	7
2.1	Installation	7
2.2	First Script	7
2.3	Command Line Options	7
2.4	Notes	7
3	CLI options	9
3.1	Usage	9
3.2	Perl-compatible options	10
3.3	Pperl-specific options	10
3.4	Runtime options	10
3.5	Examples	11
3.6	Environment variables	11
4	How-To	13
4.1	PPerl Architecture	14
4.1.1	JIT Compilation	14
	What Gets JIT-Compiled	14
	What Is Not JIT-Compiled	15
	Performance	15
	CLI Control	16
	Architecture	16
4.1.2	Auto-FFI (Peta::FFI)	17
	Architecture	17
	Layer 0: Raw FFI	17
	Layer 1: Pre-baked Bindings	18
	When to Use FFI vs Native Modules	20
4.1.3	Native Modules	20
	How It Works	20
	Available Native Modules	20
	Performance	22
	Adding Native Modules	22

4.1.4	Differences from Perl 5	22
	Intentional Differences	22
	Unimplemented Features	23
	Known Limitations	24
4.2	Concurrent Execution	25
4.2.1	Parallel Execution	26
	How It Works	26
	What Gets Parallelized	26
	CLI Control	26
	Performance	27
	Limitations	27
	How Reduction Detection Works	28
4.2.2	Threads	28
	ppperl status - read this first	29
	How this chapter is organised	29
	Reference cross-links	42
4.3	Regular expressions	42
4.3.1	Who this is for	42
4.3.2	How this guide is organised	42
	Basics	42
	Character classes	49
	Anchors and assertions	55
	Quantifiers	63
	Groups and captures	70
	Alternation	81
	Modifiers	86
	Substitution	94
	Unicode	100
	Performance	108
	Cross-engine comparison	123
4.3.3	A first round-trip	132
4.3.4	Conventions in the examples	132
4.3.5	Related reference pages	132
4.4	Performance	133
4.4.1	Who this is for	133
4.4.2	The one rule that survives everything	133
4.4.3	How this guide is organised	133
	Measuring	133
	Idioms	137
	Sorting	140
	Writing fast pperl	143
4.4.4	What this guide does not cover	147
4.4.5	A note on numbers	147
4.5	Object-oriented programming	147
4.5.1	Which chapter you want	148
4.5.2	When to use OO at all	148
4.5.3	A note on scope	149
	Modern classes	149
	Classical OO	155
	Inheritance and method resolution	161
	Roles and delegation	166

	Migrating from classical to modern	170
4.6	Debugging Perl programs	175
4.6.1	Which chapter do I want?	176
4.6.2	The three layers of this guide	176
4.6.3	Known gaps	176
4.6.4	See also	177
	Preflight: catching bugs before runtime	177
	Print debugging, warn, and the Carp family	180
	Exceptions, die, and typed error objects	184
	The interactive debugger	189
	Breakpoints, actions, and watches	194
	Inspecting state	199
	Tracing: watching execution flow	203
	IDE and DAP integration	207
4.7	The Definitive Perl One-Liner Guide	211
4.7.1	Which chapter do I want?	212
4.7.2	The three layers of this guide	212
4.7.3	Two examples before you start	212
	Sum the numbers in the last column of a whitespace-separated file	212
	Extract every unique IPv4 address from a log, newest first	212
4.7.4	Read in this order on a first pass	213
4.7.5	Bibliography	213
	The switches	213
	Progressive recipes	219
	Regex-driven one-liners	225
	Numeric and date one-liners	230
	From one-liner to shell alias	237
	Traps and surprises	242
4.8	Tying variables	248
4.8.1	Which chapter you want	249
4.8.2	How each chapter is built	249
4.8.3	What ties cost	249
4.8.4	Differences from upstream	249
	Tied scalars	249
	Tied hashes	252
	Tied arrays	255
	Tied filehandles	259
4.9	Coming from another language	262
4.9.1	Which guide you want	262
4.9.2	The one concept to read first	263
4.9.3	A note on the runtime	263
	Python to Perl	263
	PHP to Perl	276
	Ruby to Perl	291
	Lua to Perl	305
	Sed to Perl	320
	Awk to Perl	327
	Shell to Perl	336
	Tcl to Perl	347
4.10	The Definitive Guide to Graphics with Perl	358
4.10.1	Which chapter do I want?	358

4.10.2	The three arcs of this guide	358
4.10.3	A note on the runtime	359
4.10.4	Known gaps	359
4.10.5	See also	359
	Foundations: pixels, paths, and color	359
	Drawing primitives from scratch	362
	Raster drawing with Imager	367
	GD and ImageMagick when you need them	370
	Vector graphics: Cairo and SVG	373
	Image metadata and applied 2D charting	375
	2D on a screen: GUI canvases	378
	Reaching further: Peta::FFI for graphics	381
	A native Cairo binding, shipped	383
	Capstone: a drawing application	387
	Capstone: a 2D game	390
	Known gaps and what's next	394
4.11	pack and unpack - a tutorial	396
4.11.1	When you need pack and unpack	396
4.11.2	How this tutorial is organised	396
	Bytes and widths	396
	Endianness	399
	Strings, bits, and nybbles	401
	Grouping and counts	405
	Positioning and padding	408
	Network protocols - a DNS query	411
	File formats - a GIF header	414
4.11.3	A first round-trip	419
4.11.4	Conventions in the examples	419
4.12	Opening files	419
4.12.1	What this chapter does not cover	420
4.12.2	Where to go next	420
	Reading, writing, appending	420
	Encoding and layers	422
	Pipes	424
	In-memory handles	427
	Handling errors	429
4.12.3	Reference cross-links	432
4.13	References - a tutorial	432
4.13.1	Who this is for	432
4.13.2	How this tutorial is organised	432
	Basics - taking references and using them	432
	Arrays of arrays	435
	Hashes and mixed structures	439
	Anonymous references - [...] and {...}	443
	Subroutine references	445
	Weak references	448
4.13.3	Related reference pages	451
4.14	Unicode - a tutorial	451
4.14.1	The one idea	452
4.14.2	The pipeline	452
4.14.3	Chapters	452

	Strings and encodings	452
	I/O	454
	Regex and properties	457
	Pitfalls	459
	Recipes	462
4.14.4	Conventions	465
4.15	Boolean Logic for Perl Programmers - a tutorial	466
4.15.1	Who this is for	466
4.15.2	Scope	466
4.15.3	How this tutorial is organised	466
	Truthiness	466
	Operators	468
	Truth tables	471
	De Morgan's laws	474
	Functional completeness	477
	Applications	479
4.15.4	What you will be able to do after reading	483
4.16	Interprocess communication - a tutorial	484
4.16.1	Choosing a channel	484
4.16.2	How this tutorial is organised	484
	Subprocesses and co-processes	484
	FIFOs	487
	TCP sockets	489
	UNIX-domain and UDP sockets	493
	System V IPC	495
4.16.3	What this tutorial does not cover	498
5	Reference	499
5.1	Perl language	499
5.1.1	Special variables	499
	Choose a family	500
	Variable namespaces	537
	The English module	538
	Localising special variables	538
	See also	538
5.1.2	Built-in Functions	539
	A	539
	B	539
	C	539
	D	540
	E	540
	F	541
	G	541
	H	542
	I	542
	J	542
	K	543
	L	543
	M	543
	N	544
	O	544

P	544
Q	544
R	545
S	545
T	547
U	547
V	548
W	548
Y	548
__ (double-underscored)	548
5.1.3 Operators	1288
Choose a family	1289
Cross-cutting concepts	1327
See also	1327
5.1.4 Data types	1328
Choose a topic	1328
Cross-cutting concepts	1364
See also	1365
5.1.5 Subroutines	1365
Choose a topic	1365
Defining a sub vs. calling a sub	1401
The classic vs. modern split	1402
See also	1402
5.1.6 Locale handling	1403
What a locale is	1403
The use <code>locale</code> pragma	1403
Locale categories	1404
Setting the locale	1404
Locale and Unicode	1405
Locale leaks and safety	1405
Parallel execution and locale	1405
Environment variables	1405
Differences from upstream	1406
See also	1406
5.1.7 Diagnostic messages	1406
Reading an entry	1407
Enabling and silencing warnings	1407
A note on portability	1407
The diagnostics	1408
5.1.8 Security and taint mode	1421
Taint mode	1421
What gets tainted	1422
Laundering and detecting tainted data	1423
Cleaning up <code>%ENV</code> and <code>\$ENV{PATH}</code>	1423
Taint mode and <code>@INC</code>	1424
Set-id scripts and the shebang race	1424
Algorithmic complexity attacks	1425
Differences from upstream	1425
See also	1425
5.2 Modules	1426
5.2.1 B	1427

Functions	1427
5.2.2 Clone	1441
Synopsis	1441
Functions	1442
5.2.3 Config	1443
Functions	1444
5.2.4 Cwd	1455
Functions	1455
5.2.5 Data	1462
Data::Dumper	1462
5.2.6 Devel	1465
Devel::Peek	1465
5.2.7 Digest	1469
Digest::MD5	1470
Digest::SHA	1490
5.2.8 Encode	1519
Functions	1520
5.2.9 Errno	1531
Synopsis	1532
Functions	1532
5.2.10 Fcntl	1533
Open flags (sysopen)	1533
File-descriptor control (fcntl)	1533
Lock types (flock)	1533
Seek whence	1533
File mode bits (stat)	1533
Mode test helpers	1534
Performance	1534
Functions	1534
5.2.11 File	1537
File::Glob	1538
File::Map	1542
File::Temp	1552
5.2.12 Filter	1575
Filter::Util	1576
5.2.13 Hash	1584
Hash::Util	1584
5.2.14 I18N	1596
I18N::Langinfo	1596
5.2.15 IO	1597
Functions	1597
5.2.16 List	1599
List::Util	1599
5.2.17 MIME	1629
MIME::Base64	1629
MIME::QuotedPrint	1637
5.2.18 Math	1639
Math::GMP	1639
5.2.19 PDL	1654
Option B Architecture	1654
Modules	1654

Functions	1655
5.2.20 POSIX	1687
Functions	1687
5.2.21 PadWalker	1725
Functions	1726
5.2.22 PerlIO	1733
PerlIO::encoding	1733
PerlIO::mmap	1734
PerlIO::via	1734
5.2.23 Peta	1735
Peta::FFI	1735
5.2.24 SDBM_File	1765
Synopsis	1765
Persistence	1765
Size limits	1765
Security	1765
Functions	1766
5.2.25 Scalar	1768
Scalar::Util	1768
5.2.26 Socket	1768
What lives here	1769
Functions	1769
5.2.27 Storable	1785
Functions	1785
5.2.28 Sys	1802
Sys::Hostname	1802
5.2.29 Term	1802
Term::ReadLine	1802
5.2.30 Time	1815
Time::HiRes	1815
5.2.31 attributes	1816
Functions	1817
5.2.32 mro	1823
Synopsis	1823
Functions	1823
5.2.33 re	1829
Synopsis	1829
What you can do with it	1829
Functions	1830
5.2.34 version	1837
Synopsis	1837
Operators	1837
Functions	1838

6 Errata	1851
6.1 Identifier length cap (Identifier too long)	1851
6.1.1 Upstream wording	1851
6.1.2 Correction	1851
Upstream bleed has already raised the cap	1852
6.1.3 pperl behavior	1852
6.1.4 Tests	1853

A	Disclaimer & bug reports	1855
A.1	PetaPerl is not upstream Perl	1855
A.2	Bug reports	1855
A.3	Intent of compatibility	1856
A.4	Upstream’s contributions	1856
A.5	License	1856
B	License	1857
B.1	PetaMem Research Preview License	1857
B.1.1	1. Definitions	1857
B.1.2	2. Grant	1857
B.1.3	3. Attribution and Non-Appropriation	1857
B.1.4	4. Reverse Engineering	1857
B.1.5	5. Research Preview Status	1857
B.1.6	6. No Warranty	1858
B.1.7	7. Limitation of Liability	1858
B.1.8	8. Termination	1858
B.1.9	9. Reservation of Rights	1858
B.1.10	10. Governing Law and Venue	1858
C	Glossary	1859
	Index	1861

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

This is the **ppperl 0.6.0** documentation, built 2026-07-12. (translated $\approx$ words, %)

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

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

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

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

genindex

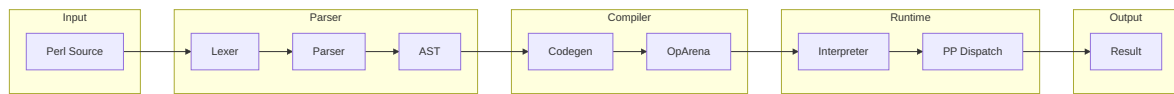
INTRODUCTION

ppperl is a next-generation Perl 5 platform written in Rust, targeting Perl 5.42+ compatibility.

1.1 Goals

- **Auto-Parallelization** - Automatic parallel map, grep, for, while loops via Rayon work-stealing
- **JIT Compilation** - Native code generation via Cranelift for hot loops (up to 76x faster than perl5)
- **Pure Perl Viability** - Fast enough that XS becomes optional

1.2 Architecture Overview



1.3 Quick Start

```
# Run a script
ppperl script.pl

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

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

1.4 Compatibility

ppperl aims for high compatibility with Perl 5.42+.

1.5 Platform Support

- **Linux only** (any architecture)
- No Windows, macOS, or other platforms

GETTING STARTED

2.1 Installation

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

2.2 First Script

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

2.3 Command Line Options

Option	Description
-e 'code'	Execute code (modern features always on)
-c	Check syntax only
-w	Enable warnings
-v	Show version
--no-jit	Disable JIT compilation
--no-parallel	Disable auto-parallelization
--cache	Enable bytecode caching

See *CLI options* for the full option list.

2.4 Notes

- Modern features (say, state, //, signatures) are always enabled - no use feature needed
- XS modules are not supported; native Rust reimplementations are provided for common modules
- -I is not supported; use PERL5LIB environment variable to add include paths

CLI OPTIONS

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

Note

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

PetaPerl - Next-generation Perl 5 runtime

3.1 Usage

```
ppperl [OPTIONS] [--] [SCRIPT [ARGS...]]  
ppperl -e 'code'
```

3.2 Perl-compatible options

Flag	Description
-e 'code'	Execute one-liner (modern features always enabled)
-Idirectory	Prepend directory to @INC
-c	Check syntax only, don't execute
-w	Enable warnings
-n	Loop over input lines (no auto-print), like while(<>)
-p	Loop over input lines and auto-print, like a sed filter
-a	Autosplit each line into @F (with -n or -p)
-F pattern	Set the -a split pattern (implies -a)
-l[octal]	Auto-chomp input, set output record separator
-0[octal]	Set input record separator (-0 = NUL, -00 = paragraph)
-i[ext]	Edit files in place (optional backup extension)
-T	Taint mode (forced)
-t	Taint warnings
-v	Show version information
-V	Show detailed configuration
--interactive	Launch interactive REPL (requires 'shell' feature)
-h, -?, --help	Show this help message

3.3 Pperl-specific options

Flag	Description
--from-json, -j	Read op tree from stdin (B::PetaPerl JSON format)
--optree, -o	Read op tree from stdin (B::Concise format)
--stats, -s	Show performance statistics (time, memory, ops)
--trace	Enable execution tracing (note: short -t is perl5 taint-warn)
--timeout=SECS	Set execution timeout in seconds (default: none)
--dump-optree	Dump canonical op tree (don't execute)
--compare-bytecode	Compare op trees: perl5 backend vs native parser
--cache	Enable bytecode caching (~/.ppperl/cache/)
--flush	Clear all bytecode caches and exit

3.4 Runtime options

Flag	Description
--no-jit	Disable JIT compilation (on by default)
--no-parallel	Disable auto-parallelization (on by default)
--threads=N	Number of threads (default: all CPUs)
--parallel-threshold=N	Minimum iterations to parallelize (default: 100)
--jit-stats	Print JIT summary to stderr at exit (see also PPERL_JIT_LOG)

3.5 Examples

```
ppperl script.pl
```

Execute a Perl script

```
ppperl -e 'say "Hello!"'
```

Execute a one-liner (say, state, etc. always available)

```
ppperl -c script.pl
```

Check script syntax without executing

```
ppperl script.pl arg1 arg2
```

Pass arguments to script (@ARGV)

For more information, see: <https://gl.petatech.eu/petatech/peta-perl>

3.6 Environment variables

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

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

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

HOW-TO

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

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

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

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

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

Regular Expressions - Perl regex from first principles through performance tuning.

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

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

Debugging - finding and fixing bugs: the `-d` debugger, inspecting live state, production-path tools.

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

Tying Variables - making a variable dispatch through a class: tied scalars, arrays, hashes, and filehandles, with a complete worked class for each.

Coming from another language - Perl by analogy for developers fluent in another language: each construct you already know mapped to its Perl counterpart, with the gotcha the similarity hides. Python first; PHP, Ruby, Lua planned.

Graphics - two-dimensional graphics in Perl, from the pixel up: the drawing algorithms, the CPAN toolkits (Imager, Cairo, Prima), reaching C libraries through pperl's native FFI, and two runnable capstones (a drawing app and a 2D game).

Tutorials ***Pack & Unpack*** - binary data layout: directives, endianness, network protocols, file formats.

Working with I/O Streams - the full open story: modes, encoding layers, pipes, in-memory handles, error handling.

Perl References - scalars, arrays, hashes, code, anonymous and weak references, nested structures.

Unicode in Perl - text vs bytes, source encoding, PerlIO layers, Unicode regex, common pitfalls.

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

Interprocess Communication - runnable IPC cookbook: subprocesses, FIFOs, TCP and UDP sockets, socket pairs, System V semaphores and shared memory.

4.1 PPerl Architecture

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

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

4.1.1 JIT Compilation

PetaPerl includes a JIT (Just-In-Time) compiler based on [Cranelfit](#), the same backend used by Wasmtime. The JIT compiles hot loops into native machine code, delivering performance that can exceed perl5 by orders of magnitude.

What Gets JIT-Compiled

The JIT targets **loops** - `for` and `while` - that contain arithmetic and comparison operations. Specifically:

Integer For-Loops

```
my $sum = 0;
for my $i (1..1_000_000) {
    $sum += $i;
}
```

The JIT detects the integer accumulator pattern and compiles the loop body to native code. The loop variable (`$i`) and accumulators (`$sum`) are held in CPU registers.

While-Loops with Float Arithmetic

```
while ($y < $max_y) {
    my $cx = $x * $scale;
    my $cy = $y * $scale;
```

(continues on next page)

(continued from previous page)

```
# ... computation ...
$y += $step;
}
```

Floating-point arithmetic (+, -, *, /), comparisons (<, >, <=, >=), and control flow (if/last) within while-loops are compiled to native SSE/AVX instructions.

Nested Loops

The JIT handles nested loops up to 5 levels deep. Each nesting level generates its own set of Cranelift basic blocks with proper phi-node connections for variables that flow between levels.

String Operations (via Extern Calls)

The JIT supports .= (concat-assign) and \$x = "" (clear) operations on string variables through extern function calls from JIT-compiled code back into the Rust runtime.

What Is Not JIT-Compiled

- **Subroutine calls** - function call overhead dominates, JIT benefit is marginal
- **Regex operations** - the regex engine has its own optimization path
- **I/O operations** - I/O-bound code doesn't benefit from JIT
- **Complex data structure access** - hash/array operations with dynamic keys
- **String-heavy computation** - string building is handled by the PvBuf optimization instead

Code that isn't JIT-compiled still runs on the interpreter, which has its own fast paths for common operations.

Performance

Ackermann Function

The interpreter fast path (not JIT) handles recursive integer arithmetic:

Runtime	Time	Speedup
perl5	630ms	1.0x
ppperl (interpreter)	14ms	45x faster

Mandelbrot Set (1000x1000)

JIT compilation of nested while-loops with float arithmetic:

Runtime	Time	Speedup
perl5	12,514ms	1.0x
ppperl (JIT only)	163ms	76x faster
ppperl (JIT + parallel)	29ms	431x faster

How It Achieves This

1. **Register allocation:** Loop variables in CPU registers instead of interpreter stack
2. **Type specialization:** Variables proven to be integer or float use native instructions directly
3. **Branch elimination:** Constant conditions removed at compile time
4. **No dispatch overhead:** Native code replaces the interpreter dispatch loop entirely within JIT'd regions

CLI Control

```
# Default: JIT enabled
ppperl script.pl

# Disable JIT (interpreter only)
ppperl --no-jit script.pl
```

The test harness runs with `--no-jit` by default to test interpreter correctness. JIT-specific tests in `t/62-jit/` override this.

Architecture

Compilation Pipeline

```
Loop detected → Analyze variables and types → Build JIT IR
→ Compile via Cranelift → Cache compiled function → Execute native
↳ code
```

Caching

Compiled functions are cached by the enterloop op's ID in the op arena. A `Cached-WhileLoop` stores:

- The compiled native function pointer
- Variable type information (float vs integer vs string)
- Constant string pool (for string operations)
- Metadata for parallel dispatch eligibility

Subsequent iterations of the same loop reuse the cached compilation.

JIT IR

The JIT uses its own intermediate representation (JitIr) that maps Perl operations to Cranelift operations:

- `JitIr::WhileLoop { condition_ir, body_ir }` - recursive for nesting
- `JitIr::FloatVar / JitIr::IntVar` - typed variable access
- `JitIr::BinOp` - arithmetic and comparison
- `JitIr::ExitIfFalse / JitIr::ExitIfTrue` - loop exit and last

Variable Types

The JIT tracks two variable types:

- **`JitType::F64`** - floating-point values held in an f64 buffer
- **`JitType::Ptr`** - string values accessed through extern calls to the Rust runtime

Variables are typed based on their usage pattern in the loop body. Mixed-type variables fall back to the interpreter.

4.1.2 Auto-FFI (Peta::FFI)

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

Architecture

The FFI system has three layers:

Layer	Module	Purpose
0	<code>Peta::FFI</code>	Raw FFI: <code>dlopen</code> + call with type signatures
1	<code>Peta::FFI::Libc</code> , <code>::UUID</code> , <code>::GMP</code>	Pre-baked bindings for specific libraries
2	<code>Peta::FFI::scan()</code>	Discovery: enumerate available system libraries

Layer 0 uses **`libffi`** for dispatch - any C function signature works, no code generation required.

Layer 0: Raw FFI

`dlopen / call / dlclose`

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

my $lib = dlopen("libm.so.6");
my $result = call($lib, "sqrt", "(d)d", 2.0); # √2 = 1.4142...
```

(continues on next page)

(continued from previous page)

```
print "sqrt(2) = $result\n";
dlclose($lib);
```

Type Signature Format

The signature string "(arg_types)return_type" uses single-character type codes:

Code	C Type	Perl Mapping
v	void	(no value)
i	int	IV
l	long	IV
L	unsigned long / size_t	UV
d	double	NV
f	float	NV
p	const char*	String (input)
P	void* (mutable buffer)	String (output)

Examples:

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

Library Discovery

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

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

for my $soname (sort keys %$libs) {
    print "$soname => $libs->{$soname}\n";
}
```

scan() calls ldconfig -p to enumerate all shared libraries available on the system.

Layer 1: Pre-baked Bindings

Layer 1 modules provide Perl-native APIs for specific C libraries, with proper argument validation, return value conversion, and error handling. No type signatures needed - just call functions.

Peta::FFI::Libc

Direct bindings to libc functions, grouped by category:

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

Strings: strlen, strerror

Environment: getenv, setenv, unsetenv

Math: abs, labs

System: sleep, usleep, gethostname, uname

File operations: access, unlink, rmdir, mkdir, chmod, chown

```

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

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

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

```

Peta::FFI::UUID

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

```

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

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

```

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

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

Peta::FFI::GMP (Math::GMP)

Arbitrary-precision integer arithmetic via libgmp. Integrates with the Math::GMP class name for compatibility with CPAN's Math::GMP:

```

use Math::GMP;

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

```

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

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

When to Use FFI vs Native Modules

Scenario	Recommendation
Common CPAN module (List::Util, etc.)	Native module (already built in)
System library not yet wrapped	Layer 0 raw FFI
Frequent calls to same library	Request Layer 1 pre-baked binding
One-off experiment	Layer 0 raw FFI

Layer 0 FFI has per-call overhead from argument marshalling and libffi dispatch. Layer 1 pre-baked bindings call Rust/C directly and are faster. Native modules are fastest - same speed as built-in operators.

4.1.3 Native Modules

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

How It Works

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

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

Available Native Modules

Core Utilities

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

Digest / Crypto

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

File / System

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

I/O

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

Data / Encoding

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

Numeric / Math

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

Build / Config

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

Introspection

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

Performance

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

Benchmarks (vs perl5 with XS):

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

Adding Native Modules

Native module documentation is extracted from the module's Rust source (`//! module docs` and `/// function docs`) by `docs/sphinx/bin/02-reference` during the docs build.

4.1.4 Differences from Perl 5

PetaPerl targets near-complete compatibility with Perl 5.42, but intentionally diverges in several areas. This document catalogs both intentional differences and current limitations.

Intentional Differences

Always-On Modern Features

PetaPerl enables all modern Perl features by default. No `use feature` or `version` declaration is needed:

- `say` - print with newline
- `state` - persistent lexical variables
- `//` - defined-or operator
- `fc` - Unicode case folding
- Subroutine signatures

use `v5.XX` declarations are parsed but have no effect. PetaPerl always behaves like 5.42.

No XS Support

PetaPerl does not load XS (C extension) modules. Instead:

- **Native implementations:** Performance-critical modules (`List::Util`, `Scalar::Util`, `Digest::MD5`, etc.) are reimplemented in Rust as native functions
- **Pure Perl fallback:** Modules with Pure Perl implementations work automatically
- **JIT compensation:** The JIT compiler closes the performance gap that historically justified XS

This is a strategic decision: XS ties Perl to a C calling convention and prevents JIT optimization. PetaPerl's JIT + auto-parallelization makes Pure Perl competitive with XS.

Linux Only

PetaPerl runs on Linux exclusively (any architecture supported by Cranelift: x86-64, AArch64, RISC-V, s390x). No Windows, macOS, BSD, or other platforms.

`$^O` always returns `"linux"`.

Version Reporting

`$]` reports `5.042000` and `$^V` reports `v5.42.0`. Code that checks these values will see PetaPerl as a perl 5.42 interpreter.

Unimplemented Features

`tie/tied/untie`

The `tie` mechanism is parsed and the functions exist, but they are stubs that return `undef`. Tied variables do not dispatch to the tie class methods.

Modules that depend on `tie` (e.g., `Tie::File`, `Tie::Hash::NamedCapture` internals) will not function correctly.

Formats

`format/write/formline` are parsed but the format output system is not implemented. `write` and `formline` are stubs. Use `printf/sprintf` instead.

Lvalue Subroutines

The `:lvalue` attribute is parsed but not enforced. Subroutines declared with `:lvalue` behave like normal subroutines.

Internals::SvREADONLY

Not implemented. This affects modules that rely on marking scalars read-only (charnames, unicode).

Some Pragmas

Pragma	Status
<code>strict</code>	Parsed, partially enforced
<code>warnings</code>	Parsed, partial warning categories
<code>utf8</code>	Parsed, strings are always UTF-8 internally
<code>constant</code>	Fully working
<code>base/parent</code>	Fully working
<code>Carp</code>	Fully working
<code>overload</code>	Partially working
<code>re</code>	Partially working

Command-Line Flags

These perl5 flags are not supported:

Flag	Description
<code>-I</code>	Add include path (use <code>PERL5LIB</code> instead)
<code>-M / -m</code>	Load module from command line
<code>-n / -p</code>	Implicit input loop
<code>-l</code>	Automatic line-ending processing
<code>-a / -F</code>	Autosplit mode
<code>-i</code>	In-place edit
<code>-x</code>	Extract script from message
<code>-T</code>	Taint mode

Known Limitations

Regex Engine

PetaPerl's regex engine passes 99.3% of perl5's `re_tests` (1959/1972). Known gaps:

- Self-referential captures (e.g., `(a\1)`) - 3 tests
- `local` in `(?{code})` blocks - 2 tests
- Multi-character Unicode case folding - 2 tests
- Branch reset group backreferences - 5 tests

- String interpolation edge cases - 1 test

Module Compatibility

Module	Issue
Test2::API	<code>use Module(\lexical)</code> loses compile-time lexical side effects
Module::CoreList	Timeout (>20s) due to enormous hash literals
charnames/unicode	Depends on <code>Internals::SvREADONLY</code>
Pod::Simple::RTF	Flip-flop operator edge case

Flip-Flop Operator

The `..` and `...` operators in scalar context (flip-flop) have edge cases where pperl's behavior differs from perl5. Most common uses work, but complex chaining may trigger errors.

Filehandle Heuristics

`print $x, "\n"` interprets `$x` as a potential filehandle (matching perl5 behavior). For variables that hold filehandle references, use the block form: `print {$fh} "data\n"`.

Compile-Time Module Effects

`use Module(args)` runs at compile time in perl5. Modules that produce lexical side effects at compile time (e.g., `use Instance(\INST)`) may not work because pperl's runtime doesn't replay those effects. Package-variable side effects work correctly.

4.2 Concurrent Execution

Doing work in parallel. pperl ships two distinct concurrency stories and they answer different questions:

- **Auto-parallelization** - pperl's runtime detects parallelisable patterns in ordinary Perl and spreads them across cores without any user-level threading. This is the preferred answer for CPU-bound loops over large data.
- **Classical ithreads (partial support)** - Perl 5's `threads` and `threads::shared` model. pperl's support here is deliberately limited; many programs that historically used ithreads are better rewritten against the auto-parallel path.

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

4.2.1 Parallel Execution

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

How It Works

When PetaPerl encounters a parallelizable loop, it:

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

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

What Gets Parallelized

JIT'd While-Loops

When the JIT compiles a while-loop and the analysis detects:

- A counter variable with known bounds
- Reduction variables (accumulate-only pattern)
- No I/O or side effects in the loop body

Then the loop body is compiled once and executed in parallel across threads.

Built-in Functions

`map` and `grep` with pure callbacks can execute in parallel:

```
my @results = map { expensive_computation($_) } @large_array;  
my @filtered = grep { complex_test($_) } @large_array;
```

Parallelization requires:

- No side effects in the callback
- No shared mutable state
- Collection size above the parallelization threshold

CLI Control

```
# Default: parallelization enabled  
ppperl script.pl  
  
# Disable parallelization  
ppperl --no-parallel script.pl
```

(continues on next page)

(continued from previous page)

```
# Explicitly enable (default)
ppperl --parallel script.pl

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

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

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

Performance

Mandelbrot Set (1000x1000)

Mode	Time	vs perl5
perl5	12,514ms	1.0x
ppperl interpreter	~3,500ms	3.6x faster
ppperl JIT	163ms	76x faster
ppperl JIT + parallel (8 threads)	29ms	431x faster

Scaling

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

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

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

Limitations

String Operations

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

Side-Effect Detection

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

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

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

Determinism

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

Output order is preserved for map and grep - the results array maintains the same element ordering as sequential execution.

How Reduction Detection Works

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

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

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

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

4.2.2 Threads

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

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

ppperl status - read this first

ppperl **does not implement ithreads**. The `threads` and `threads::shared` modules are not available. At runtime:

- `use threads` fails at compile time.
- `use threads::shared` fails at compile time.
- The `lock` built-in is a silent no-op, exactly as upstream Perl behaves when built without thread support. Code that sprinkles `lock` defensively for single-threaded compatibility runs unchanged.

That decision is deliberate. `ithreads` impose a per-interpreter data copy, constrain every core module's internals, and historically bred a long tail of subtle bugs around shared aggregates, signal delivery, and library thread-safety. pppperl invests instead in **runtime-level parallelism for pure Perl code** - JIT-compiled loop bodies dispatched across a Rayon work-stealing pool, with automatic reduction analysis and zero user-visible thread objects. See [Parallel Execution](#) for how that works and when it fires.

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

How this chapter is organised

ithreads basics

This chapter describes the perl5 `ithreads` model as it exists in upstream Perl 5.42. It is the vocabulary you need to read existing threaded Perl code. pppperl does not run this code - see the landing page for the reason and [Alternatives](#) for the pppperl-native equivalents.

What is a thread

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

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

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

Checking for thread support

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

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

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

Creating a thread

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

```
use threads;

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

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

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

Passing arguments:

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

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

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

Joining a thread

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

```
use threads;

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

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

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

spawning a thread whose return will be consumed as a list.

If several threads are outstanding, join them in turn:

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

Detaching a thread

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

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

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

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

Process and thread termination

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

The canonical hang-up:

```
use threads;

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

sub thrsb {
    my ($label) = @_;
    sleep 1;
    print "thread $label\n";
}

# Falls off the end. Perl warns "Perl exited with active threads:
# 2 running and unjoined" and aborts both threads.
```

Fix by joining before the main thread exits:

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

Three structural patterns

Most threaded programs fall into one of three shapes.

Boss/worker

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

Work crew

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

Pipeline

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

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

Thread identity

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

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

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

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

yield - usually a no-op

```
threads->yield;
```

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

ppperl status

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

Thread-safety of modules

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

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

See also

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

Shared data

Under upstream Perl 5.42, sharing data between ithreads is explicit: variables are private by default, and become visible across threads only when the `:shared` attribute is applied or a value is passed through a thread-safe primitive. This chapter walks through the full upstream shared-data surface - `threads::shared`, [lock](#), semaphores, queues, condition variables - and flags pperl's status at each section.

ppperl status at a glance

None of the sharing primitives on this page do anything at runtime under pperl:

- `use threads::shared` fails at compile time - the module is not shipped.
- `:shared` is parsed but has no underlying interpreter to share to.
- [lock](#) is a silent no-op, as documented on its reference page.
- `Thread::Queue` and `Thread::Semaphore` are not available.

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

Shared and unshared variables

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

```
use threads;

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

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

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

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

For an aggregate, every element becomes shared automatically:

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

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

The lock built-in

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

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

my $total :shared = 0;

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

Properties worth remembering:

- **Advisory only.** `lock` blocks other callers of `lock` on the same datum. A thread that reads or writes `$total` without calling `lock` is not serialised.
- **Block-scoped.** Release happens at block exit, not at the end of the `lock` statement. There is no `unlock`.

- **Recursive per thread.** The same thread may re-acquire a lock it already holds; other threads still block.
- **Works on aggregates.** `lock @queue`, `lock %h`, `lock &sub` are all valid.

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

Races

The canonical race:

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

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

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

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

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

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

Deadlocks

Two threads, two locks, acquired in opposite order:

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

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

The standard defences:

- **Fixed lock order.** Every thread that needs multiple locks acquires them in the same global order. Always `$x` before `$y`.

- **Short critical sections.** Hold a lock only for the work that must be serialised, not for anything else.
- **Fewer locks.** Protect the whole structure with one lock rather than one per field.

Queues - the preferred communication channel

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

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

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

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

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

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

Under pperl the equivalent shape is a straight Perl array combined with the parallel `map` or `grep` built-ins - see [Alternatives](#).

Semaphores

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

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

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

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

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

Condition variables

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

Process-scope side effects

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

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

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

Differences from upstream

ppperl does not ship `threads::shared`, `Thread::Queue`, or `Thread::Semaphore`. The `:shared` attribute, `cond_wait`, and `cond_signal` are not recognised. `lock` compiles but is a no-op; this matches upstream’s behaviour under a build without thread support.

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

See also

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

Alternatives to ithreads

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

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

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

Decision table

Your goal	Upstream tool	ppperl-native answer
Parallelise a compute-heavy loop	threads + work-crew pattern	Auto-parallelisation (JIT + Rayon)
Parallelise map/grep over large data	Work-crew thread pool	Parallel map / grep
Run I/O-bound tasks concurrently	threads + Thread::Queue	<code>fork</code> with pipes, or event-driven I/O
Isolate a subtask with its own state	Detached thread	<code>fork</code>
Produce/consume pipeline	threads + Thread::Queue chain	<code>fork</code> + pipes, or sequential iterator
Synchronised counter	:shared + <code>lock</code>	Sequential accumulator + parallel reduction

Auto-parallelisation for compute loops

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

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

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

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

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

Before:

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

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

After:

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

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

Parallel map and grep

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

```
my @results = map { expensive($_) } @input;
my @filtered = grep { test($_) } @input;
```

When the callback has no detectable side effects and the input size exceeds `--parallel-threshold`, pperl dispatches the callback in parallel. Result order is preserved.

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

fork - process-level concurrency

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

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

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

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

Forking a work crew

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

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

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

Pipeline via fork and pipes

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

```
open my $producer, '|-', 'find', '/data', '-type', 'f'
    or die "fork: $!";
while (my $path = <$producer>) {
    chomp $path;
    # filter / process / forward
}
close $producer;
```

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

Isolated background work

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

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

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

Choosing between auto-parallelisation and fork

- **Auto-parallelisation** wins for compute-bound loops over in-memory data: no process startup cost, no serialisation of results, JIT-compiled body.
- **fork** wins for I/O, subprocess work, and anything where the task should not inherit the parent's global state changes.
- **Neither** is the right answer for low-cost task dispatch in tight loops - the usual culprit there is a sequential loop that does not actually benefit from concurrency. Measure before adding either layer.

See also

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

Reference cross-links

- [lock](#) - the only ithreads primitive with a reference page. Its *Differences from upstream* section is the canonical statement of pperl's divergence.
- [fork](#) - process-level concurrency, fully supported under pperl.
- [wait](#) - reap a child process.
- [Parallel Execution](#) - pperl's auto-parallelisation.
- [Reference](#) - threading support is a runtime-level topic.

4.3 Regular expressions

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

This guide is the definitive pperl reference for regex. Each chapter covers one topic in three layers: an overview that orients you, a precise reference that defines what the construct does, and the gory details - corner cases, performance pathologies, and the cross-engine differences that surprise a Perl-trained reader. The same chapter serves the reader who needs a one-paragraph answer and the reader who wants to know exactly when a pattern will misbehave.

4.3.1 Who this is for

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

4.3.2 How this guide is organised

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

Basics

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

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

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

The match operator

The long form is `m//`:

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

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

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

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

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

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

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

Other regex operators

`m//` is one of four:

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

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

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

Pattern reuse with `qr//`

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

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

(continues on next page)

(continued from previous page)

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

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

```
my $name    = qr/[A-Z][a-z]+/;  
my $number  = qr/\d+/;  
my $entry   = qr/$name\s+$number/x;  
  
"Alice 42" =~ /^$entry$/;    # matches
```

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

Binding: `=~` and `!~`

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

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

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

Matching against `$_`

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

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

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

Case sensitivity and the default anchor

Matches are case-sensitive and unanchored:

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

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

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

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

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

Metacharacters

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

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

Two more are special only in specific contexts:

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

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

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

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

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

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

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

Strict mode: use `re 'strict'`

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

```
use re 'strict';

/abc{,1/;           # error: unescaped '{' in non-quantifier context
```

(continues on next page)

(continued from previous page)

```
/(?-p)/;      # error: useless negation of always-on flag
/[a-]/;       # error: dash at end of class
```

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

Escape sequences

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

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

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

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

Variables in patterns

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

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

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

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

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

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

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

How a regex is read

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

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

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

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

Earliest match wins - the bump-along

Friedl puts it precisely:

The match that begins earliest wins.

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

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

Say what you mean

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

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

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

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

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

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

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

Substitution at a glance

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

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

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

Where to go next

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

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

See also

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

Character classes

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

```

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

```

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

Ranges

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

```

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

```

Ranges can be combined with individual characters:

```

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

```

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

```

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

```

Negation

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

```

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

```

A caret elsewhere is literal:

```

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

```

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

Special characters inside a class

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

```

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

```

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

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

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

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

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

Shorthand classes

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

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

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

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

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

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

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

The period

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

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

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

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

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

Composing classes

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

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

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

POSIX classes

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

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

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

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

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

POSIX also defines two related constructs that are rarely implemented:

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

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

Unicode properties

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

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

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

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

Extended bracketed classes - (? [])

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

Negated class beats .+?

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

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

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

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

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

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

Worked example: an IP-address regex

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

1. **Naive.** “Four dot-separated digit groups.”

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

Matches and then ? happily - every group is optional, the pattern is satisfied by four dots and nothing else.

2. Require digits. Anchor the pattern.

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

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

3. Bound the digit count, badly.

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

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

4. Allow 1 to 3 digits.

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

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

5. Range-correct.

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

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

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

Cross-engine: shorthand classes

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

Shorthand	Perl 5.42 (default)	PCRE	Emacs	POSIX BRE / ERE	RE2 / Go (default)
\d	ASCII (or Unicode under /u)	ASCII	NO (use [0-9] or [[:digit:]])	NO	ASCII; Unicode under (?u)
\w	ASCII or Unicode	ASCII	yes (syntax-table-driver)	NO	ASCII; Unicode under (?u)
\s	ASCII or Unicode	ASCII	yes	NO	ASCII; Unicode under (?u)
\b word boundary	yes	yes	yes (\</\> traditional)	NO	yes
\h, \v	yes	yes	NO	NO	NO

Two things to internalise:

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

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

A useful habit

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

See also

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

Anchors and assertions

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

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

Start and end of string

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

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

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

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

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

Absolute versus line-relative anchors

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

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

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

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

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

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

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

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

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

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

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

Word boundaries

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

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

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

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

\b outside [...] is a word boundary. \b *inside* [...] is the *backspace* character, \x08. The dual meaning is the single most common trap in the regex syntax. Always read \b's context before reading what it does.

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

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

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

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

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

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

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

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

Unicode word-boundary variants

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

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

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

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

\G - the last-match position

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

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

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

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

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

```
my $s = "06192054410-44272-13901-44106-22134";
while ($s =~ /44(\d{5})/g) {
    print "got 44$1\n";
}
```

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

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

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

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

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

Lookahead

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

```
my $x = "I catch the housecat 'Tom-cat' with catnip";
$x =~ /cat(=?\s)/;    # matches 'cat' in 'housecat' - space follows
$x =~ /cat(?! \s)/;   # matches 'cat' in 'catch' - no space follows
```

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

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

Lookahead worked example: digits not followed by a period

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

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

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

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

Now apply it to `4423.45`:

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

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

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

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

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

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

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

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

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

To force “first word that doesn’t begin with `cat`”, anchor the lookahead to a word boundary:

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

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

Lookbehind

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

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

Negative lookbehind is `(?<!...)`:

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

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

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

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

\K - keep-left

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

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

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

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

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

Long-form lookahead aliases

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

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

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

Combining lookahead with split

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

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

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

Two zero-width assertions juxtaposed are AND-ed

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

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

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

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

Cross-engine: anchors and lookahead

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

Anchors and newline handling

Concern	Perl 5.42	PCRE2	Emacs	POSIX BRE / ERE	RE2 / Go
<code>^</code> start-of-string	yes	yes	start-of-buffer	yes	yes
<code>^</code> after <code>\\n</code> under multi-line	/m	(?m)	always (line-oriented)	not specified	(?m)
<code>\$</code> end-of-string	yes	yes	end-of-buffer	yes	yes
<code>\$</code> before final <code>\\n</code>	yes	yes	no	varies	yes
<code>\\A</code> / <code>\\z</code>	yes	yes	<code>\\`</code> / <code>\\'</code>	no	yes
<code>\\Z</code>	yes	yes	no	no	no (use <code>\\z</code>)
<code>\\G</code>	yes	yes	no	no	no

Lookaround and atomic constructs

Feature	Perl 5.42	PCRE2	Emacs	POSIX	RE2 / Go
Lookahead <code>(?=...)</code> / <code>(?!...)</code>	yes	yes	NO	NO	NO
Fixed-width lookbehind	yes	yes	NO	NO	NO
Variable-length lookbehind	yes (exp.)	yes	NO	NO	NO
<code>\\K</code>	yes	yes	NO	NO	NO
Atomic group <code>(?>...)</code>	yes	yes	NO	NO	NO

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

Summary

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

See also

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

Quantifiers

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

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

The basic quantifiers

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

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

The upper bound `n` and `m` cannot exceed an engine-built constant (typically 65534 - the actual limit appears in the error if you exceed it). For “any sane number” this is plenty; patterns hitting the limit are usually generated, not handwritten.

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

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

The quantifier applies to the atom immediately before it:

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

Group or class, then quantify.

Greedy by default

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

Greedy gives back only what it must

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

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

Greediness vs. matching

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

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

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

Principles of the match

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

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

Examples:

```

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

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

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

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

```

Non-greedy quantifiers

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

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

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

```

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

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

```

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

```

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

```

(continues on next page)

(continued from previous page)

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

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

When non-greedy loses to a negated class

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

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

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

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

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

Possessive quantifiers

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

A common idiom for matching quoted strings without backtracking catastrophe:

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

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

Possessive equals atomic-group sugar

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

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

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

Illegal possessive-and-non-greedy combinations

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

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

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

Choosing the right quantifier

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

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

Zero-width quantification

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

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

Worked examples from perlre’s backtracking section

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

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

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

Reading each:

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

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

The lessons:

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

Cross-engine notes

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

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

See also

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

Groups and captures

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

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

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

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

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

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

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

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

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

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

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

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

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

Captured the empty string vs. did not match at all

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

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

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

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

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

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

Failed matches do not reset capture variables

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

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

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

Non-capturing groups: `(?:...)`

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

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

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

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

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

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

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

Named captures

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

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

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

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

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

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

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

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

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

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

Backreferences

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

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

Examples:

```
# Match a three-letter word followed by a space and the same word.
"the the other day" =~ /\b(\w{3})\s\1\b/;    # $1 eq 'the'

# Match a four-letter, three-letter, two-letter, or one-letter
# run followed by itself.
/^(\\w{1,4})\\1$/;    # 'beriberi', 'booboo', 'coco', 'mama', 'papa'
```

`\g{...}` versus `\1` for disambiguation

Use `\g{...}` (or `\k<...>`) when digits or octal-looking characters follow the reference, to avoid ambiguity:

```
/(\\d)abc\\g{1}23/;    # the '1' refers to group 1, '23' is literal
/(\\d)abc\\123/;    # '\\123' is octal 0x53 ('S'), not group 1
```

Perl's rule for the bare `\N` form: `\1` through `\9` always mean backreferences. `\10`, `\11`, ... mean a backreference *only if* that many capture groups have opened earlier in the pattern; otherwise they are octal character literals. This is why `\g{...}` is the safer form when the pattern is built by concatenation.

If you use the brace form, optional surrounding spaces are permitted - `\g{ -1 }` and `\k{ name }` are valid.

Relative backreferences

`\g-1`, `\g{-1}` refer to the *immediately previous* capturing group; `\g{-2}` refers to the one before that; and so on. The distance is counted by *opened* parens, including unclosed ones. This matters when a pattern fragment is interpolated inside another:

```
my $pair = '([a-z])(\\d)\\g{-1}\\g{-2}';    # a11a, g22g, x33x, ...

# Embed it: outer group shifts numbering by 1, but relative
# backreferences still work:
"code=e99e" =~ /^(\\w+)= $pair$/;    # matches
```

Without relative backreferences, this would require knowing how many groups precede `$pair` at every interpolation site.

Named and relative references make long patterns robust against copy-paste.

Atomic groups: `(?>...)`

`(?>...)` is a non-capturing group whose contents, once matched, are *committed*. The engine cannot backtrack into the group on failure - only past it as a whole.

```
"aaab" =~ /a*ab/;    # matches: a* gives back one 'a'
"aaab" =~ /(??>a*)ab/;    # does not match: a* takes all, refuses to
↪give
```

The full treatment lives in the [performance](#) chapter. The construct is documented here because its syntax is a parenthesised group; structurally it belongs in the captures chapter alongside `(?:...)`.

Possessive quantifiers `(*+, ++, ?+, {n,m}+)` are exact syntactic sugar for `(?>...)` around the quantified atom. The following are equivalent:

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

The long-form spelling `(*atomic:...)` is also accepted.

Recursive subpatterns

Perl regexps can refer back to a capture group as if it were a subroutine call. The construct re-runs the captured pattern at the current position. This makes genuinely recursive structures matchable - balanced parentheses, S-expressions, nested brackets - without falling out of the regex DSL.

Form	Recurses to
<code>(?R)</code>	the entire pattern
<code>(?0)</code>	same as <code>(?R)</code>
<code>(?1)</code>	group 1
<code>(?2)</code>	group 2 (and so on)
<code>(?-1)</code>	most-recently-opened group (relative)
<code>(?+1)</code>	next group to be opened (forward relative)
<code>(?&NAME)</code>	named group
<code>(?P>NAME)</code>	same as <code>(?&NAME)</code> (Python-compatible)

Note: relative recursion counts unclosed groups, unlike relative backreferences. `(?-1)` always means the most recently opened group whether or not it has closed.

A balanced-paren matcher:

```
my $bal = qr/
    (? (DEFINE)
        (?<paren>
            \(                # opening paren
            (?
                [^() ]++      # non-paren run, possessive
                | (?&paren)    # or a balanced sub-group,
↪ recursively
            ) *+
            \)                # closing paren
        )
    )
```

(continues on next page)

(continued from previous page)

```

    (?&paren)
/x;

"((a)(b(c)))" =~ /^$bal$/;    # matches

```

The `(?(DEFINE) ...)` block declares a named subpattern without matching anything itself; the `(?&paren)` after it invokes that subpattern, and the recursive `(?R)` inside the body is what gives the unbounded depth. `(?R)` would not work here - it recurses the *whole* enclosing pattern, including the `^` and `$` anchors that get added at the call site, which forces every level of recursion to demand start- and end-of-string.

A more nuanced recursive pattern: match a function `foo(...)` where the argument may itself contain balanced parens.

```

my $re = qr/(
    foo                # group 1: full function call
    (                  # group 2: parens with content
        \ (            # group 3: contents of parens
            (?
                (?> [^()]+ ) # non-paren without_
                ↪backtracking
                |
                (?2)         # recurse to group 2
            ) *
        )
    )
)/x;

'foo(bar(baz)+baz(bop))' =~ /$re/ and
    print "1: $1\n2: $2\n3: $3\n";
# 1: foo(bar(baz)+baz(bop))
# 2: (bar(baz)+baz(bop))
# 3: bar(baz)+baz(bop)

```

Capture state inside recursion

When a group recurses into itself, captures set inside the recursion are *not* visible to the caller after the recursion returns. The recursive call has its own capture state, which is discarded on return. This is why most recursive patterns wrap a secondary capture group around the recursive call when the matched text is needed:

```

/(?<NAME>(?(NAME_PAT)))(?<ADDR>(?(ADDRESS_PAT)))
  (?(DEFINE)
    (?<NAME_PAT>....)
    (?<ADDRESS_PAT>....)
  )/x

```

Here `$+{NAME}` is the outer capture; `$+{NAME_PAT}` is undefined because it lived only

inside the recursion.

The recursion-depth ceiling is compiled into the engine. Patterns that recurse without consuming input fail immediately rather than running forever - the engine detects the cycle.

(DEFINE) blocks

The `(?(DEFINE)...)` block holds named subpatterns that never run on their own - they are only invoked through `(?&NAME)` . This is how to write a regex that looks like a small grammar:

```
my $email = qr/
  \A (?&LOCAL) @ (?&DOMAIN) \z
  (? (DEFINE)
    (?<LOCAL>    [\w.+ -]+ )
    (?<DOMAIN>   (?&LABEL) (?: \. (?&LABEL) )+ )
    (?<LABEL>    [a-zA-Z0-9] (?: [a-zA-Z0-9 -]* [a-zA-Z0-9] )? )
  )
/x;
```

The DEFINE block at the end keeps the named subpatterns in one place; the top of the pattern reads as a clean specification.

Two cautions:

- Subpatterns in a DEFINE block *count toward* the absolute and relative numbering of capture groups in the surrounding pattern. Name everything in the DEFINE so you don't have to count.
- The optimiser is less effective on DEFINE-style patterns than on the equivalent in-lined form. They run, but not always at top speed.

Conditional patterns: `(?(cond)yes|no)`

A conditional pattern picks between two sub-patterns based on a runtime test:

- `(?(N)YES|NO)` - match YES if group N matched something, else NO.
- `(?(<NAME>)YES|NO)` - same, but by name.
- `(?(?=LOOK)YES|NO)` - match YES if the lookahead succeeds.
- `(?(?{CODE})YES|NO)` - match YES if the code block returns true.
- `(?(R)YES|NO)` - match YES if currently inside a recursion.
- `(?(R1)YES|NO)` - match YES if recursing through group 1.
- `(?(R&NAME)YES|NO)` - match YES if recursing through named group.

The NO branch is optional; missing NO is treated as “always match”.

Worked example: optionally-quoted text. If the input opens with `(`, require a closing `)`. Otherwise allow no parens:

```
m{ ( \ ( )?           # optional opening paren, group 1
    [^()]+           # body (no parens)
    (? (1) \ )       # if group 1 matched, require closing paren
}x;
```

Without conditional patterns, this would need an alternation covering both shapes; the conditional version exposes the intent.

Worked example: balanced grammar with named recursion. Useful inside a `DEFINE` block for context-sensitive validation:

```
qr/
    (?<expr>
        (?<atom> [a-z]+ | \ ( (?&expr) \ )
        (? : \s* [+] \s* (?&atom) ) *
    )
/x
```

Conditionals are the part of regex that most resembles a real programming language; reach for them when an alternation of mutually-exclusive shapes would be unwieldy.

Branch reset: `(?|...)`

Inside `(?|...)`, every alternative branch starts numbering its captures at the same slot. After the group, numbering resumes one past the maximum across all branches. The full treatment is in the [alternation](#) chapter; the capture-numbering implications belong here.

```
# Before -----branch-reset----- after
/ ( a ) ( ? | x ( y ) z | ( p ( q ) r ) | ( t ) u ( v ) ) ( z ) /x
# 1          2          2 3          2      3      4
```

After this pattern matches:

- `$1` is always the leading `a`.
- `$2` is `y`, `(p q r)`, or `t` depending on branch.
- `$3` is undef from branch 1, `q` from branch 2, `v` from branch 3.
- `$4` is the trailing `z`.

If you use named captures inside `(?|...)`, use the *same names in the same order* in every branch:

```
/(?|  (?<a> x ) (?<b> y )
    |  (?<a> z ) (?<b> w )) /x;
```

Mixing names across branches works (Perl resolves to the leftmost defined name) but produces surprising results: every name references the same slot, so `${a}` and `${b}` may have the same value across different branches.

Position arrays: @- and @+

After a successful match, @- and @+ hold the start and end offsets of the whole match and of each capture group:

- \$-[0], \$+[0] - offsets of the whole match.
- \$-[n], \$+[n] - offsets of the nth capture, or undef if the group did not participate.

```
my $s = "Mmm...donut, thought Homer";
if ($s =~ /^(Mmm|Yech)\.\.\.(donut|peas)/) {
    for my $i (1 .. $#-) {
        printf "Match %d: %s at (%d,%d)\n",
            $i,
            substr($s, $-[$i], $+[$i] - $-[$i]),
            $-[$i], $+[$i];
    }
}
# Match 1: Mmm at (0,3)
# Match 2: donut at (6,11)
```

Offsets are often easier than substrings when you need to modify the original string at the matched position.

@{^CAPTURE} - captures as an array

Perl exposes all numbered captures as a single array @{^CAPTURE}, indexed from 0 (where index 0 is \$1, index 1 is \$2, ...):

```
$string =~ /$pattern/ and my @captured = @{^CAPTURE};
```

This is convenient when the number of captures is variable or unknown - code that takes captures from a user-supplied pattern no longer has to count (to know how many \$1/\$2/... to read).

Subscripting requires the demarcated-curly form:

```
print "${^CAPTURE[0]}"; # equivalent to $1
```

%{^CAPTURE_ALL} and %{^CAPTURE_NAMES} exist for named-capture introspection - see [perlvar](#).

Prematch, match, postmatch

Perl sets three special scalars after each match that expose the surrounding text:

- \$` - everything before the match (the *pre-match*).
- \$& - the match itself.
- \$(' - everything after the match (the *post-match*).

```
"the cat caught the mouse" =~ /cat/;
# $` = 'the '
```

(continues on next page)

(continued from previous page)

```
# $&    = 'cat'
# $'    = ' caught the mouse'
```

The named variants `${^PREMATCH}`, `${^MATCH}`, `${^POSTMATCH}` are equivalent. Both forms are zero-cost; use whichever reads better.

`$+` and `$$N`

`$+` holds the match of the highest-numbered capture group that succeeded.

`$$N` holds the match of the most-recently-closed capture group - the rightmost `)` that completed. This is exactly what you want inside a `(?{...})` code block to access the latest capture without counting parens:

```
$_ = "The brown fox jumps over the lazy dog";
/the (\S+)(?{ $color = $$N }) (\S+)(?{ $animal = $$N })/i;
print "color = $color, animal = $animal\n";
# color = brown, animal = fox
```

See the *performance* chapter for the full embedded- code story.

POSIX subexpression maximisation - what other engines do

Some engines (any conformant POSIX NFA, plus engine implementations designed to mimic POSIX, plus most DFA hybrids) follow a different rule for which match wins when multiple are possible: *longest-leftmost*, with the constraint that *each subexpression captures the longest substring it can*.

For the pattern `(to|top)(o|polo)?(gical|o?logical)` against `topological`:

- Perl (and PCRE2, and any Traditional NFA) tries alternatives left-to-right. `(to|top)` matches `to`; the engine pushes on. `(o|polo)` matches `o`; pushes on. `(gical|o?logical)` matches `logical`. Done.
- POSIX semantics requires the *overall* match be longest, and among matches of equal length, each individual group's capture to be longest. The result is `top polo gical`, where each capture is at its widest.

Modern engines essentially never use POSIX-NFA semantics, but some DFA-based tools (`grep`, `awk`) approximate it. The *cross-engine* chapter has the full table.

Cross-engine: captures and backreferences

This is the one feature where engines diverge most. The full table is in the *cross-engine* chapter; the relevant rows extracted:

Feature	Perl 5.42	PCRE2	Emac:	POSIX BRE	POSIX ERE	RE2/Go
Numbered captures	yes	yes	yes	yes	strict no, GNU yes	yes (no backrefs)
Backreferences in pattern (\1-)	yes	yes	yes	yes (\1-\9)	strict no	NO
Named captures (? <name> . . .)	yes	yes	NO	NO	NO	yes
Branch reset (? ...)	yes	yes	NO	NO	NO	NO
@{^CAPTURE} array	yes	NO	NO	NO	NO	NO (different API)
Recursive subpatterns (?R) etc.	yes	yes	NO	NO	NO	NO
Conditional patterns (? (c) y n)	yes	yes	NO	NO	NO	NO

The single most important takeaway: **RE2 / Go regexp does not support backreferences at all**. This is not an oversight; it is the price RE2 pays for guaranteed linear-time matching. Patterns that need backreferences are not regular languages and cannot be matched by a DFA in linear time.

If you are porting Perl regexps to a Go service, audit them for backreferences before assuming the conversion is mechanical. Patterns like `/(?<a>\w+)` and `\k<a>/` simply cannot be expressed in RE2.

See also

- The [alternation](#) chapter - branch reset and the alternative-numbering implications of `(? |...)`.
- The [performance](#) chapter - atomic groups, recursive patterns in worked examples, embedded code.
- The [modifiers](#) chapter - `/n` to suppress all captures.
- `perlvar` - `%+`, `%-`, `$&`, `$$^N`, `@{^CAPTURE}` and friends.

Alternation

Alternation is the `|` operator. It picks between two or more sub-patterns at the same position.

```
"cats and dogs" =~ /cat|dog|bird/;    # matches 'cat'
"cats and dogs" =~ /dog|cat|bird/;    # matches 'cat'
```

The order of the alternatives does not change *where* the overall pattern matches. The engine still honours “earliest position wins” - both patterns above match at position 0 because that’s the earliest position where any alternative can match.

Leftmost alternative wins at a given position

Within a single starting position, alternatives are tried left to right and the first one that succeeds is used:

```
"cats" =~ /c|ca|cat|cats/;    # matches 'c' - first alternative
↳ wins
"cats" =~ /cats|cat|ca|c/;    # matches 'cats' - first wins, longer
```

If one alternative is a prefix of another and you want the longer match, put it first. The engine does not look past the first successful alternative at the current position.

This is *Traditional NFA* behaviour, and it is what Perl, PCRE2, Python, and most modern engines implement. POSIX-conformant engines (notably some `awk` and `grep` implementations) follow the *longest-leftmost* rule instead - they would match `cats` regardless of alternative order. The *cross-engine* chapter has the comparison.

Friedl puts it succinctly:

Greedy alternation is non-greedy in a Traditional NFA.

The pattern `tour|to|tournament` against three tournaments won matches `tour`, not `tournament`. The first alternative succeeds and the engine commits to it; longer alternatives further along the list are never tried.

Implications for ordering

An alternation that is part of a larger pattern can have its performance and correctness shaped by the order of alternatives:

- **Specificity first.** When you want the longest of several prefixes, put the longest first. `/web|website|websites/` matches `web` even on input `website`; `/websites|website|web/` matches `websites`.
- **Common case first.** Alternation is tried left-to-right; if 90% of your inputs hit alternative 3, the engine wastes time on alternatives 1 and 2 every time. Reorder by frequency.
- **Sibling captures.** When the alternatives capture, the order affects which `$n` is set - see *Alternation and capturing* below.

Combining-pieces formal rule

`perlre`'s "Combining RE Pieces" gives the precise statement underlying "leftmost wins". For two pattern pieces `S` and `T`:

When `S` can match, it is a better match than when only `T` can match.

That is the formal version of the rule. "Better" means the engine prefers it. For two `S` matches, the same internal ordering applies (greediness rules within the alternative); likewise for `T` matches. Across alternatives, *the existence of a successful `S` match excludes consideration of `T`.*

This is why `S|T` cannot be reordered by the engine on its own: Perl does not search for the *best* alternative across the disjunction - it commits to `S` whenever `S` succeeds.

Grouping vs. alternation precedence

| has very low precedence. It splits the pattern at the *outermost* level containing it:

```
/ab|cd/;      # 'ab' OR 'cd'
/^ab|cd$/;    # '^ab' OR 'cd$' - probably not what you meant!
/^(ab|cd)$/;  # '^' + ('ab' or 'cd') + '$' - what you meant
```

To constrain alternation to part of a pattern, wrap it in a group. Non-capturing (?:...) is preferred unless you need the capture:

```
/house(?:cat|keeper)/;  # 'housecat' or 'housekeeper'
/house(cat|keeper)/;    # same, but $1 will be 'cat' or 'keeper'
```

The group creates a local scope for |. Outside the group | resumes its top-level role:

```
/^(?:foo|bar|baz)$|^xyz$/;  # ('foo'/'bar'/'baz') or 'xyz'
```

Empty alternatives

An empty alternative matches the empty string - a useful trick for “this or nothing”:

```
/house(cat|)/;      # 'housecat' or 'house'
/(19|20|)\d\d/;    # '19xx', '20xx', or just 'xx'
```

Modern style prefers (?:...)? over (?:...|); they are equivalent, but the ? form is clearer:

```
/house(?:cat)?/;    # same as house(cat|), no capture
```

Watch for the backtracking cost when an empty alternative is combined with a quantifier - the engine can re-explore the same position many times. See the [performance](#) chapter on zero-length-match termination.

Cross-engine note

Strict POSIX, lex, and most older awk implementations *disallow* empty alternatives - (this|that|) is a syntax error. Perl 5.42, PCRE2, Rust’s regex crate, Python re, and modern engine implementations accept them. If a pattern needs to be portable to older POSIX tools, write (?:this|that)? instead - it expresses the same idea and is portable across the engines that accept neither form.

Alternation inside character classes

Character classes are almost always what you want when alternating between single characters. /a|b|c/ and /[abc]/ match the same strings, but [abc] is faster, terser, and clearer:

```
/a|b|c/;      # works, but verbose
/[abc]/;      # use this
```

Alternation is for alternatives longer than one character (or ones that are themselves patterns). When each alternative is a single character, reach for a class. The engine

treats a class as a *single* atomic choice; an alternation as N choices to be explored in order.

Common-prefix factoring

A pattern like `/this|that|then|those/` defeats the engine's fixed-string-check optimisation: there is no literal prefix the engine can scan for cheaply. Refactoring to expose the common prefix turns the alternation into something the optimiser can work with:

```
/this|that|then|those/;      # no common prefix visible
/th(?:is|at|en|ose)/;      # common prefix 'th' exposed
```

The two patterns match the same strings, and the second is materially faster on large inputs. The engine scans for `th` using Boyer-Moore, then runs the small alternation only at candidate positions.

The general technique:

1. Find the longest literal prefix common to all alternatives.
2. Lift it outside the alternation.
3. Wrap the remainder in `(?:...)` so the alternation stays localised.

For lists of words with a few different prefixes, you can apply the rewrite recursively. For very long lists, see the [performance](#) chapter's section on matching many strings - once the list is in the thousands, a specialised matcher beats any alternation.

Alternation and capturing

Only one alternative inside a group can match at a time, so the group captures the matching alternative:

```
if ("bert" =~ /(cat|dog|bert|ernie)/) {
    print "matched $1\n";      # matched bert
}
```

Sibling groups outside the alternation retain their normal numbering:

```
/^(\w+):\s*(yes|no|maybe)$/;
# $1 = the key, $2 = the verdict
```

Inside a nested alternation, the groups are numbered left to right by opening paren, even across branches:

```
/(a)|(b)/;
# On match of 'a': $1 = 'a', $2 undef
# On match of 'b': $1 undef, $2 = 'b'
```

Check with defined `$n`, not truth - an empty capture is different from an absent capture.

Branch reset: (?|...)

Parallel-capture patterns are the usual reason you pick up (?|...). Inside (?|...), every branch starts numbering its captures at the same slot. After the group, numbering resumes at one past the maximum across all branches.

```
# Without (?|...): need to know which branch matched.
if ($time =~ /(\d\d|\d):(\d\d)|(\d\d)(\d\d)/) {
    my ($h, $m) = ($1, $2);
    ($h, $m) = ($3, $4) unless defined $h;
}

# With (?|...): $1 and $2 come from whichever branch matched.
if ($time =~ /(?|(\d\d|\d):(\d\d)|(\d\d)(\d\d))/) {
    my ($h, $m) = ($1, $2);
}
```

With a trailing fixed piece:

```
if ($time =~ /(?|(\d\d|\d):(\d\d)|(\d\d)(\d\d))\s+([A-Z]{3})/) {
    # $1 = hours, $2 = minutes, $3 = zone (numbered after the group)
    print "hour=$1 minute=$2 zone=$3\n";
}
```

Rules inside (?|...):

- Each branch independently numbers its capturing groups from the current group count.
- After the group, the outer numbering continues at one higher than the maximum count reached in any branch.
- Named groups keep their names; you can repeat a name across branches. Use the *same names in the same order* in every branch, or surprises ensue (see the [groups and captures](#) chapter).

Branch reset is the cleanest way to express “parse X in one of several equivalent formats, then reach for the same variables afterwards.”

Alternation in split

split takes a regexp pattern, so alternation works there too:

```
my @words = split /\s+|-/, "one-two three four-five";
# ('one', 'two', 'three', 'four', 'five')
```

If the separator pattern contains capturing groups, split includes the captured text in the output list - often surprising. Use (?:...) unless you want that:

```
split /(?:\s+|-)/, "a-b c";    # ('a', 'b', 'c')
split /(\s+|-)/, "a-b c";    # ('a', '-', 'b', ' ', 'c')
```

The capturing form is occasionally what you want - preserving the exact separators between fields. Mostly you want non-capturing.

Summary

- `|` separates alternatives; leftmost that matches at the current position wins.
- Wrap alternations in `(?:...)` to localise them.
- Prefer character classes over single-character alternations.
- Lift common prefixes outside the alternation when you want the engine's literal-scan optimisation to fire.
- `(?:|...)` resets capture numbering across branches - use it when the branches capture the same conceptual fields.

See also

- The *groups and captures* chapter - capture numbering inside `(?:|...)`, and the rest of the named- capture machinery.
- The *performance* chapter - ordering by likelihood, common-prefix factoring, and matching many strings.
- The *cross-engine* chapter - alternation syntax differences across engine families.
- `split` - alternation as separator syntax.

Modifiers

Modifiers change how a pattern is matched without changing the pattern itself. They appear after the closing delimiter of a match, substitution, or `qr//`:

```
"Hello" =~ /hello/i;      # case-insensitive match
$x =~ s/foo/bar/g;        # global substitution
my $re = qr/\d+/i;        # compiled pattern with /i
```

The same modifiers can be embedded inside a pattern with `(?flags)` or `(?flags:...)`, localising their effect to part of the pattern.

The full modifier set

Modifier	Effect
/i	case-insensitive matching
/m	multi-line: ^ and \$ match at every \n
/s	single-line: . matches \n too
/x	extended: ignore whitespace and # comments
/xx	extended-extended: also ignore whitespace inside [...]
/g	global: match as many times as possible
/c	do not reset pos on failure (with /g)
/r	s///r returns the result instead of modifying
/e	s///e evaluates the replacement as Perl code
/ee	s///ee evaluates, then evaluates the result too
/n	non-capturing: (...) act like (? :...)
/p	no-op; accepted for compat
/o	compile the pattern once (rarely needed; prefer qr/)
/a	\d, \w, \s restricted to ASCII
/aa	/a, plus /i does not cross ASCII/non-ASCII boundary
/u	Unicode semantics regardless of use utf8
/l	use current locale
/d	dual-mode semantics - avoid in new code

The first eight are the everyday modifiers; the charset modifiers /a, /u, /l, /d are covered in detail in the [unicode](#) chapter and summarised again here.

/i - case-insensitive

```
"Hello" =~ /hello/i;      # matches
"HELLO" =~ /[A-Z]/i;      # matches - class is insensitive too
"Grüße" =~ /GRÜSSE/i;    # matches under Unicode semantics
```

Unicode case folding includes mappings like ß → ss, German Eszett casefolding, and so on. The full table lives in the Unicode standard; for ASCII, /i does what you expect.

/i carries a negligible performance penalty for ASCII. The implementation folds case at *compile time* - the pattern carries the case-folded character set, the input is read once. Unicode case-folding is more involved (sequences like ß ↔ ss), but the extra work is bounded and rarely shows up in practice.

/m - multi-line

Changes where ^ and \$ match. Without /m, they match only at the outer ends of the string. With /m, they match at every embedded newline too.

```
my $x = "first\nsecond\nthird";

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

(continues on next page)

(continued from previous page)

```
$x =~ /first$/m;    # matches
$x =~ /third$/m;    # matches
```

`\A`, `\z`, `\Z` remain absolute string anchors even under `/m`

- see the *anchors and assertions* chapter.

`/s` - single-line

Makes `.` match newline characters too.

```
my $x = "a\nb";
$x =~ /a.b/;      # does not match - . does not cross \n
$x =~ /a.b/s;     # matches
```

`/m` and `/s` are independent. They can both be used on the same match. Despite the names, they do not conflict:

```
$x =~ /^a.b$/sm;  # . matches newline AND ^,$ are line-aware
```

`/x` - extended pattern

Ignores literal whitespace and lets `#` introduce end-of-line comments. Crucial for any pattern more than a line long.

Before `/x`:

```
/^[+-]?\d+(\.\d*)?([eE][+-]?\d+)?$/;
```

After:

```
/^
  [+ -]?          # optional sign
  \d+             # integer part
  ( \. \d* )?     # optional fraction
  ( [eE] [+ -]? \d+ )? # optional exponent
$/x;
```

Whitespace in the *pattern* is ignored; whitespace you want to match becomes `\s`, `\` , or `[ ]`:

```
/\w+ \s+ \w+/x;      # three tokens: word, space, word (literal
→ spaces ignored)
/\w+\s+\w+/x;        # equivalent
/key:[ ]value/x;     # literal space via bracket class
/key:\ value/x;      # literal space via backslash
```

`#` starts a comment that ends at the next newline. To match a literal `#` under `/x`, escape it or put it in a class.

The “Pattern White Space” set under `/x` follows Unicode UAX#31: SPACE, CHARACTER TABULATION, LINE FEED, LINE TABULATION, FORM FEED, CARRIAGE RETURN,

NEXT LINE, PARAGRAPH SEPARATOR, LINE SEPARATOR. In practice you only meet ASCII space, tab, and newline.

/xx - extended in classes too

Inside [...] whitespace is *not* ignored under plain /x - [ab c] matches a, b, ' ', or c. To also ignore whitespace inside classes, use /xx:

```
/[ab c]/xx;           # matches 'a', 'b', or 'c' - space ignored
/[ab\ c]/xx;         # the \ is needed to match a literal space
/[ ! @ ]/xx;         # matches '!' or '@' - visible spacing
```

/xx is a superset of /x. It is convenient for character classes laid out for readability, especially with shorthand classes:

```
/[ \d \s \-_,. ]+/xx; # digits, whitespace, common punctuation
```

/xx was added in Perl 5.26. Patterns built before that read fine; new code can reach for it freely.

/g - global

In scalar context, keeps a position in the string (pos \$x) and advances each time the pattern matches, allowing iteration:

```
my $x = "cat dog house";
while ($x =~ /(\w+)/g) {
    print "$1 at ", pos($x), "\n";
}
# cat at 3
# dog at 7
# house at 13
```

In list context, returns all matches at once:

```
my @words = $x =~ /(\w+)/g; # ('cat', 'dog', 'house')
```

If the pattern contains no captures, list context returns the whole matched text for each match:

```
my @digits = "abc123def456" =~ /\d+/g; # ('123', '456')
```

With multiple captures, each iteration returns the tuple in order:

```
my @pairs = "a=1,b=2,c=3" =~ /(\w)=(\d)/g;
# ('a', '1', 'b', '2', 'c', '3')
```

/c - preserve position on failure

By default, a failed /g match resets pos to undef. Under /gc, pos stays at its previous value - crucial for hand-rolled lexers:

```
my $s = "123abc";
while (1) {
    if ($s =~ /\G(\d+)/gc) { print "num $1\n"; next; }
    if ($s =~ /\G([a-z]+)/gc) { print "word $1\n"; next; }
    last; # nothing matched; exit
}
```

See the *anchors and assertions* chapter for \G.

/r - non-destructive substitution

s/// normally modifies the target string and returns the count. s///r leaves the target alone and returns the result:

```
my $name = "  Alice  ";
my $trimmed = $name =~ s/^\s+|\s+$//gr;
# $name is still "  Alice  "
# $trimmed is "Alice"
```

Enables substitution chains without intermediate variables:

```
my $clean = $input
    =~ s/\s+/ /gr          # collapse runs of whitespace
    =~ s/^\s+|\s+$//gr    # trim ends
    =~ s/[\x00-\x7f]//gr; # drop non-ASCII
```

Each s///r returns the transformed string, which the next one receives.

/e - evaluate the replacement

The replacement half of s///e is Perl code, not a double-quoted string. The return value of the code replaces the match:

```
my $x = "numbers: 1 2 3 4";
$x =~ s/(\d+)/$1 * 2/ge;
# $x is now "numbers: 2 4 6 8"
```

/ee evaluates twice: the code returns a string, which is then evaluated as Perl again. Rarely useful and easy to misuse - only reach for it when you are sure. The *substitution* chapter has worked examples.

/n - non-capturing

Makes every ordinary (...) behave like (?:...). Useful when a long pattern has parentheses purely for grouping and you want to keep \$1, \$2, ... unset:

```
"hello" =~ /(hi|hello)/n;    # matches, but $1 is not set
```

Named captures (`?<name>...`) still capture under `/n`. Added in Perl 5.22.

`/p` - no-op

`/p` is a silent no-op. `${^PREMATCH}`, `${^MATCH}`, `${^POSTMATCH}` are always available. Accepted for backward compatibility, but contributes nothing in new code.

`/o` - compile once

`/o` disables re-compilation of a pattern that interpolates variables. Compiled patterns are also cached automatically, and `qr//` gives explicit control. `/o` is rarely the right tool; `qr//` is clearer and composes.

Charset modifiers - `/a`, `/aa`, `/u`, `/l`, `/d`

These control how the shorthand classes (`\d`, `\w`, `\s`), case folding, and the POSIX class set behave under Unicode and locale considerations. Brief summary here; the full treatment is in the [unicode](#) chapter.

Modifier	Behaviour
<code>/u</code>	full Unicode semantics. Default under use v5.12+.
<code>/a</code>	restrict <code>\d</code> , <code>\w</code> , <code>\s</code> , <code>[[:...:]]</code> to ASCII.
<code>/aa</code>	<code>/a</code> plus <code>/i</code> does not cross ASCII/non-ASCII.
<code>/l</code>	follow the current use locale POSIX locale.
<code>/d</code>	dual-mode semantics. Avoid in new code.

`/a`, `/u`, `/l`, `/d` are mutually exclusive - at most one can be in effect at a time. `/aa` is a refinement of `/a`. They are *set-once* modifiers: an inner (`?d:...`) cannot un-set an outer `/u`.

The everyday choice in modern code: use v5.12 (or later) selects `/u` automatically, which is what you want for Unicode-correct text. Add `/a` only when the pattern must reject non-ASCII characters - typically because the input is known to be ASCII-only and you want to enforce that.

Inline modifiers

`(?flags)` turns flags on for the rest of the enclosing group (or pattern):

```
/(?i)yes/;    # case-insensitive, same as /yes/i
```

`(?flags:...)` scopes the flags to the inner group only and is non-capturing:

```
/Answer: ((?i)yes)/;    # only the 'yes' is case-insensitive
/Answer: ((?i:yes))/;    # clearer: scope is the group's contents
```

`(?-flags)` turns flags off. They can be combined:

```
/(?i-m:pattern)/; # turn on /i, turn off /m, within this group
```

Inline modifiers are the right tool when different parts of a long pattern need different modifiers. They are also useful in patterns built by interpolation, where the embedded `(?i)` travels with the pattern fragment.

Caret form: `(?^...)`

`(?^flags)` is shorthand for “reset all flags, then apply these”. The expansion is `d-imnsx` followed by the listed flags. So `(?^x:...)` means “default flags except `/x` is on, regardless of what flags were in scope outside”.

The caret form is what Perl uses when stringifying compiled patterns; you may see it in error messages or `qr//` output. In hand-written code it is occasionally useful when interpolating a pattern fragment that should not inherit modifiers from its surroundings:

```
my $strict = qr/(?^:foo|bar)/; # always default flags
# ... no matter how this is interpolated.
```

A negative flag is not legal after the caret (it would be redundant - the caret already cleared everything).

Mutually-exclusive flags

`/a` and `/aa` override each other (last one wins); same for `/x` and `/xx`. They are not additive. `(?xx-x:...)` turns *all* `x` behaviour off, not “subtract one `x` from two”.

The charset family `/a`, `/d`, `/l`, `/u` is mutually exclusive; specifying one un-specifies the others. They cannot be turned *off* with a leading `-`, only switched between. So `(?-d:...)` is a fatal error; `(?dl:...)` is also fatal (two charset flags together).

`/p` is special: its presence anywhere in a pattern has a global effect (and that effect is a no-op).

Order of modifiers

On a match or substitution, the order of trailing modifiers is not significant. `/mgis` and `/sgmi` are identical. Pick a house style and stick to it.

How a regex is read - the four phases

For modifiers and inline forms to work, you have to understand that Perl reads a regex in *phases*. The phases:

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

The order matters because phases B and D operate on different representations:

- `\Q$dir\E` cooks at Phase B, before the regex compiler sees it. By Phase D the variable's contents have already been metaquoted; the regex compiler sees a literal pattern.
- `\U...\E` is interpreted by Phase B as a string-cook directive (uppercase the contents), which is almost certainly *not* what you want inside a regex. Use `\Q` and `\E` for regex purposes; `\U` only when you want the pattern's *literal text* to be uppercased.
- `(?#comment)` is removed before Phase B sees the pattern at all. A literal `#` inside `(?#...)` is fine. A literal `)` is not - the comment ends at the first `)`.

The `/x` modifier applies at Phase C, between interpolation and compilation. That means whitespace introduced by an interpolated variable is *not* stripped by `/x`:

```
my $sub = " a b ";      # contains literal spaces
"axb" =~ / x $sub x /x; # the spaces in $sub are NOT stripped
```

If you want the interpolated content to obey `/x`, you have to strip its whitespace at the source.

use re 'strict'

`use re 'strict'` raises a number of normally-tolerated regex sloppiness conditions to compile-time errors. It is per-lexical- scope. In strict mode:

- A `{` in a non-quantifier position is an error, not a literal.
- `[a-]` (dash at end of class) is an error.
- A useless negation of an always-on flag is an error.
- A `(?-d)` (trying to turn off `/d`) is an error rather than a warning.
- Unmatched `[` in a string is an error.

Strict mode is recommended for new regex-heavy code, especially generated patterns where small typos go unnoticed in a normal build. It is not default because it would break working older patterns.

```
use re 'strict';

/abc{,1/;      # error in strict; warning otherwise
/[a-]/;       # error in strict
```

See also

- The [unicode](#) chapter - `/a`, `/aa`, `/u`, `/l`, `/d` in detail, plus what they do to character classes and case folding.
- The [substitution](#) chapter - `/e`, `/r`, `/g`, and `/c` in their substitution-specific roles.
- The [basics](#) chapter - the four-phase parsing model in its first appearance.
- The [anchors and assertions](#) chapter - `\A`, `\z`, `\Z` interactions with `/m`.
- The [performance](#) chapter - `/o` discussion and why `qr//` superseded it.

- `m`, `s`, `qr` - operator references.

Substitution

`s///` is the search-and-replace operator. Its general form is

```
$target =~ s/pattern/replacement/modifiers;
```

It finds the first position where *pattern* matches in *target*, replaces that match with *replacement*, and returns the number of substitutions made. With `/g`, it does that for every non-overlapping match.

```
my $x = "Time to feed the cat!";  
$x =~ s/cat/hacker/;           # $x is now "Time to feed the hacker!"
```

If the pattern does not match, the target is unchanged and `s///` returns a false value.

Operating on `$_`

Like `m//`, `s///` defaults to `$_` when no `=~` is given:

```
for (@lines) {  
    s/\s+$/;/           # trim trailing whitespace from each line  
}
```

Counting substitutions

The return value of `s///` in scalar context is the number of replacements:

```
my $count = ($text =~ s/\btodo\b/done/gi);  
print "cleared $count todos\n";
```

In boolean context, zero means no match.

The replacement string

The replacement is a double-quoted string, so variables and escape sequences interpolate:

```
my $suffix = ".bak";  
$file =~ s/$/$suffix/;    # append .bak to the name  
  
$x =~ s/\t/    /g;        # each tab becomes four spaces
```

Inside the replacement, `$1`, `$2`, `&`, `+` and friends refer to what the *current* match just captured:

```
my $y = "'quoted words'";  
$y =~ s/^(.*)'$/1/;      # strip surrounding single quotes  
# $y is now "quoted words"
```

Named captures work the same way through `${name}`:

```
"2026-04-23" =~ s/(?<y>\d{4})-(?<m>\d{2})-(?<d>\d{2})/${y}/${m}/${d}/;
# result: "23/04/2026"
```

Escaping in the replacement

The replacement is interpreted as a double-quoted string, so the double-quote rules apply to *it*, independently of the pattern. Metacharacters like `*`, `+`, `?`, `$` retain their double-quoted meanings - only `$` and `@` trigger interpolation:

```
$x =~ s/(\w+)/[$1]/g;      # wrap every word in brackets
$x =~ s/\$/USD/g;         # replace literal '$' with 'USD'
```

Backslash escape sequences in the replacement follow the double-quote rules - `\t`, `\n`, `\x{...}` all work. The special case-modification escapes `\l`, `\u`, `\L`, `\U`, `\E`, `\Q` apply in the replacement too:

```
$x =~ s/(\w+)/\u$1/g;      # capitalise the first letter of each word
$x =~ s/(\w+)/\U$1\E/g;    # uppercase whole word
```

Warning: `\1` versus `$1` in the replacement

A long-grandfathered idiom: writing `\1`, `\2`, ... in the replacement to mean “the first capture”, “the second capture”:

```
$pattern =~ s/(\W)/\\1/g;    # works, but is a trap
```

Perl accepts `\1` through `\9` in the replacement of `s///` for historical compatibility with `sed`. **Use `$1` through `$9` instead.** The reason is that the replacement is a double-quoted string, where `\1` *normally* means a control-A character. The substitution operator’s special-case rule kludges that double-quote interpretation to mean “first capture”, but the edges are sharp.

Specifically: combining `\1` with `/e` re-introduces the double-quote meaning. Under `/e`, the replacement is *Perl code*; in Perl code, `"\1"` is control-A:

```
s/(\d+)/ \1 + 1 /eg;    # warning: \1 better written as $1
```

A second sharp edge: `\1` followed by digit characters is ambiguous. `\1000` could mean “first capture, then literal `000`”, or it could mean “octal `0x40` (`@`) - and you cannot disambiguate by writing `\{1\}000`”:

```
s/(\d+)/\1000/;        # ambiguous, best avoided
s/(\d+)/${1}000/;      # unambiguous
```

The rule for new code: **always use `$1` in the replacement, never `\1`.** The `\1` form is grandfathered, not recommended.

/g - replace all

Without /g, `s///` replaces only the first match. With /g, it replaces every non-overlapping match from left to right:

```
my $x = "I batted 4 for 4";
$x =~ s/4/four/;           # "I batted four for 4"
$x = "I batted 4 for 4";
$x =~ s/4/four/g;          # "I batted four for four"
```

/g in substitution is unrelated to /g in matching. Here it just means “do this again and again until the pattern stops matching.”

Zero-length matches and `s///g`

A pattern that can match the empty string presents a problem under /g: every position would match again forever. Perl breaks this loop with a specific rule: **after a zero-length match, the next attempt at the same position is forbidden to also be zero-length.** The engine takes the second-best match instead.

The cleanest demonstration:

```
$_ = 'bar';
s/\w??/<$&>/g;
# Result: <><b><><a><><r><>
```

The non-greedy `\w??` prefers the empty match. After producing one, the next iteration is forbidden from being zero-length too, so the engine takes the second-best match - a single `\w` character. Then another empty match, then another character, and so on, alternating until the string is consumed.

The pattern: zero-length matches alternate with one-character matches, and the substitution does not loop forever.

A useful corollary: `s/(\d{3})/$1,/g` does not insert a comma after every set of three digits *correctly* - it inserts after the first three from the left, then the next three, regardless of where you wanted the commas. To insert thousands separators, match from the right or use 1 while:

```
my $n = "1234567";
1 while $n =~ s/^(\\d+)(\\d{3})/$1,$2/;
# n is now "1,234,567"
```

The 1 while repeats until the substitution returns false (no more matches), which is exactly what we want for “every position, after settling”.

/e - evaluate the replacement

Under /e, the replacement is Perl code, not a string. The code runs for every match and its return value replaces the match.

```
my $x = "numbers: 1 2 3 4";
$x =~ s/(\\d+)/$1 * 2/ge;      # "numbers: 2 4 6 8"
```

The match variables \$1, \$2, ... are visible inside the code block.

Practical uses:

```
# Uppercase the first letter of every sentence.
$text =~ s/^(^|\s+)(\w)/$1U$2/g;

# Hex-encode non-ASCII bytes.
$bytes =~ s/([\x80-\xff])/sprintf "\\x%02x", ord $1/ge;

# Apply a lookup table.
my %subst = (red => 0xff0000, green => 0x00ff00, blue => 0x0000ff);
$css =~ s/\b(red|green|blue)\b/sprintf "%06x", $subst{$1}/ge;
```

/ee - double-eval

/ee evaluates *twice* - first as code, then the result is itself evaluated as Perl. The use case is dynamic variable lookup by name:

```
our $greeting = "hello";
my $s = "greeting";
$s =~ s/(\w+)/"\$1"/ee;           # first: '$greeting'; second:
    ↪ 'hello'
# $s is now "hello"
```

The replacement is a Perl expression - the explicit double-quotes make it produce the *string* "\$greeting". The first e evaluates that expression to the string \$greeting. The second e treats that string as Perl, looking up the variable \$greeting and yielding hello. The match position is replaced with the value hello.

The use case is narrow and the security implications are substantial: any input that reaches an /ee substitution is effectively executed as Perl. Treat /ee like eval - never on user-supplied data.

/r - non-destructive substitution

/r returns the new string instead of modifying the target. The target is untouched:

```
my $orig = "I like dogs";
my $new = $orig =~ s/dogs/cats/r;
# $orig is still "I like dogs"
# $new is "I like cats"
```

If the pattern does not match, /r returns the original string unchanged:

```
my $x = "I like dogs";
my $y = $x =~ s/elephants/cougars/r;
# $y eq "I like dogs"
```

The big win is chaining:

```
my $slug = $title
    =~ s/[\w\s-]//gr      # drop punctuation
    =~ s/\s+/-/gr        # collapse spaces to dash
    =~ s/--+/-/gr;        # collapse runs of dashes
```

Each arrow returns a new string, which the next `s///r` receives.

\K in substitution

`\K` (covered in detail in the *anchors and assertions* chapter) shines in substitution. The pattern matches a prefix that establishes context, then `\K` excludes the prefix from the replacement target:

```
# Without \K - the prefix has to be re-output:
$_ =~ s/(foo)bar/$1/g;

# With \K - the prefix is matched but not part of $&:
$_ =~ s/foo\Kbar//g;
```

Both replace `foobar` with `foo`. The `\K` form is cleaner: no capture is needed, and the substitution pattern is shorter and faster.

`\K` works similarly in any substitution where a fixed prefix should match but not be replaced. It is the substitution-side twin of `lookbehind`.

Substitution and `pos()`

Successful `s///g` advances `pos($string)` past each match. A zero-length match advances `pos` by one to break the loop. After substitution, `pos` is reset.

A subtle interaction: `s///g` will not re-scan a region that has just been matched. After each successful replacement the match position advances past the *matched* text - the length of the replacement does not matter. So:

```
my $x = "aaa";
$x =~ s/a/AA/g;
# $x is "AAAAAA" - three a's, each replaced with AA
# It is NOT an infinite loop: pos() moves past each matched 'a',
# so the just-inserted 'A's are never re-scanned.
```

This is sometimes surprising for substitutions whose replacement contains characters that the pattern would also match. The rule is “advance past the matched text, never re-scan it”.

Alternate delimiters

Both halves of `s///` can use matched brackets; the delimiters do not have to be the same:

```
s{pattern}{replacement}g;
s<pattern><replacement>g;
s[pattern][replacement]g;
s(pattern)(replacement)g;    # legal, but avoid mixing for clarity
```

Any printable punctuation works in a single-character form:

```
s!/usr/local/!/opt!/g;
s#pattern#replacement#;
```

Useful when the pattern or replacement contains /.

Single-quoted substitution

`s'pattern'replacement'` treats both halves as single-quoted - no variable interpolation, no backslash escapes apart from `\'` and `\\`:

```
s'@users'@admins'g;      # literal '@users' becomes literal '@admins'
```

Rarely needed; mention it for completeness.

Combining with /m, /s, /x

Substitution modifiers compose with match modifiers, so you can get the full toolbox:

```
# Trim each line.
$x =~ s/^\s+|\s+$//mg;

# Collapse blank lines.
$x =~ s/\n{2,}/\n\n/g;

# Replace C-style comments across newlines.
$src =~ s/{/*.*?*/}{}gs;  # /s lets . cross newlines
```

With /x the pattern is readable, with /s it spans newlines, with /g it repeats.

A real example

Turn a block of English into a slug - lowercase, hyphens instead of spaces, drop non-letters, collapse and trim:

```
sub slug {
    my ($s) = @_ ;
    return lc($s)
        =~ s/[\^w\s-]//gr      # remove punctuation
        =~ s/\s+/-/gr         # spaces to hyphens
        =~ s/-+/-/gr          # collapse hyphens
        =~ s/^-|-$/gr;        # trim leading/trailing hyphens
}

print slug("Hello, World! -- Regex Tutorial");
# prints: hello-world-regex-tutorial
```

Friedl's framing - "anything that isn't required will always be considered successful"

A pattern's optional pieces will always succeed if they can match nothing. This applies inside substitution as forcefully as in matching:

```
my $x = "no horizontal rule here";
$x =~ s/-*/<HR>/;
# Result: "<HR>no horizontal rule here"
```

The pattern `-*` matches zero or more dashes - and zero is always available, at the start of the string. `<HR>` is inserted at position 0. The pattern has succeeded by the engine's definition; it has failed by yours.

The fix is to require at least one dash:

```
$x =~ s/-+/<HR>/;      # no match; $x unchanged
```

The slogan: **always consider what will happen if there is no match**. Optional pieces are not free; they always succeed by matching nothing somewhere.

Things that look like substitution but aren't

`tr///` (also written `y///`) does character-by-character transliteration, not regexp substitution. It accepts two lists of characters; each occurrence of the *n*th character in the first list becomes the *n*th character in the second list. No regexp features apply:

```
my $x = "Hello";
$x =~ tr/A-Za-z/a-zA-Z/;      # swap case
(my $hex = $bytes) =~ tr/\x00-\xff//d;  # delete all bytes -
    ↪pointless
```

`tr///` is faster than `s///` for character-level work because it is not a regexp engine. See [tr](#) for full semantics.

See also

- [s](#) - substitution operator reference.
- [tr](#) - character transliteration; not substitution but often confused for it.
- [quotemeta](#) - escape a string for safe interpolation into a pattern.
- The [modifiers](#) chapter - `/g`, `/e`, `/r`, `/ee`, and how they interact.
- The [anchors and assertions](#) chapter - `\K` for “match this prefix but don't replace it”.
- The [performance](#) chapter - `s///g` zero-length match termination in detail.

Unicode

Perl's strings are Unicode strings. Every code point from `\x{0000}` through `\x{10FFFF}` is a valid character; regardless of the underlying byte encoding. Regexp's operate on characters, not bytes - so `\w`, `\d`, `\s`, `.`, and `\X` all understand the Unicode world.

This chapter covers how to write Unicode characters in a pattern, how to match by Unicode property, and how the *charset modifiers* `/a`, `/u`, `/l`, `/d` tune the default character-class behaviour. It closes with *Script Runs*, the mechanism for rejecting strings that mix scripts when they shouldn't.

Writing Unicode in a pattern

Four ways to denote a specific code point:

Form	Example	Notes
<code>\xHH</code>	<code>\x41 (A)</code>	Two hex digits, 0–255
<code>\x{...}</code>	<code>\x{263a} (☺)</code>	Any code point
<code>\o{...}</code>	<code>\o{101} (A)</code>	Octal
<code>\N{name}</code>	<code>\N{GREEK SMALL LETTER SIGMA}</code>	Unicode character name

```
/\x{263a}/;           # WHITE SMILING FACE
/\N{GREEK SMALL LETTER SIGMA}/; # σ
/\N{U+03C3}/;         # same codepoint via U+ form
```

Whitespace and underscores inside `\N{...}` and `\p{...}` are ignored, so you can write `\N{GREEK SMALL LETTER SIGMA}` or `\N{greek_small_letter_sigma}`.

Inside a character class, the same escapes work:

```
/[\x{0370}-\x{03ff}]/; # any Greek block character
/[\N{SECTION SIGN}\N{PILCROW SIGN}]/;
```

`\N{NAME}` (named character) is *not* the same as `\N` (any character except `\n` - see the *character classes* chapter). The presence or absence of the brace is what selects.

Matching by property: `\p{...}` and `\P{...}`

Unicode classifies every code point by dozens of properties. The most useful in everyday patterns:

Property	Matches
<code>\p{L}</code>	any letter
<code>\p{Ll}</code>	lowercase letter
<code>\p{Lu}</code>	uppercase letter
<code>\p{Lt}</code>	titlecase letter
<code>\p{N}</code>	any numeric character
<code>\p{Nd}</code>	decimal digit
<code>\p{P}</code>	punctuation
<code>\p{M}</code>	mark (combining character)
<code>\p{S}</code>	symbol
<code>\p{Z}</code>	separator (including spaces)
<code>\p{C}</code>	control, format, unassigned, private-use
<code>\p{ASCII}</code>	code point 0–127
<code>\p{White_Space}</code>	any Unicode whitespace

`\P{X}` is the negation of `\p{X}` - matches any code point *not* in property X.

Short single-letter properties drop the braces:

```
/\pL/;    # letter - same as \p{L}
/\pN/;    # number
/\pM/;    # mark
```

Compound properties

Some properties have multiple values. The compound form `\p{Property=Value}` (or `\p{Property:Value}`) spells them out:

```
/\p{General_Category=Uppercase_Letter}/;    # same as \p{Lu}
/\p{Block=Greek_and_Coptic}/;
/\p{Numeric_Type=Decimal}/;
/\p{Bidi_Class=R}/;                          # right-to-left
↪characters
```

Case, spaces, and underscores are ignored in the property name and value, so `\p{gc=lu}` is the same as `\p{General_Category=Uppercase_Letter}`.

For the full list of properties and values, see [perluniprops](#) - that list is generated from Unicode data and is not duplicated here.

Matching by script

Scripts (writing systems) are properties too:

```
/\p{Latin}/;
/\p{Greek}/;
/\p{Cyrillic}/;
/\p{Arabic}/;
/\p{Han}/;           # Chinese, Japanese kanji, Korean hanja
/\p{Hiragana}/;
/\p{Katakana}/;
/\p{Hangul}/;
/\p{Script=Greek}/;  # compound form, same meaning
```

For linguistic classification, prefer `\p{Script_Extensions=...}` - it handles code points used by multiple scripts more sensibly than `\p{Script=...}`. Perl's `\p{Greek}` shorthand actually uses `Script_Extensions` under the hood.

Grapheme clusters: \X

A *grapheme cluster* is what users perceive as a single character. A Danish å can be one code point (U+00E5) or two (A followed by combining ring U+030A). Both look identical on screen. `\X` matches either - one atomic user-visible character.

```
my $s = "A\u{030A}";    # A + combining ring
length($s);              # 2 - code-point count
```

(continues on next page)

(continued from previous page)

```
$s =~ /. /;           # matches just the 'A'
$s =~ /\X/;           # matches the whole cluster
```

Use `\X` when iterating visibly across user-facing text: cursor movement, column counting, truncation, anywhere “characters” means “what the user sees”.

Extended bracketed character classes - `(? [...])`

The standard `[...]` form takes unions of characters; `(? [...])` adds set-algebra operators that combine Unicode properties:

```
# Letters that are Greek:
/(?[ \p{Letter} & \p{Greek} ])/;

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

# Hex digits without 'a' through 'f':
/(?[ [0-9A-Fa-f] - [a-f] ])/;
```

Operators: `+` (or `|`) for union, `&` for intersection, `-` for difference, `^` for symmetric difference, `!` for complement. The construct is itself a character class - it matches one character - and quantifies normally. The *character classes* chapter covers it in full.

The charset modifiers

The flags `/a`, `/u`, `/l`, `/d` control how the shorthand classes (`\d`, `\w`, `\s`) and case folding behave. They are orthogonal to `/i`, `/m`, `/s`, `/x`. At most one can be in effect at a time; `/aa` is a refinement of `/a`.

`/u` - Unicode semantics

`\d`, `\w`, `\s`, `\b`, case folding, and `[[:alpha:]]` use full Unicode definitions. This is the default when `use v5.12` or later is in effect.

```
"item\x{0660}" =~ /\w+\d/u;   # matches - U+0660 is an Arabic-Indic
↪zero
"naïve" =~ /\w+/u;           # matches the whole word
```

Modern code should use `/u`. `use v5.12+` selects it automatically.

`/a` - ASCII-only classes

Restricts `\d`, `\w`, `\s`, `[[:...:]]` to the ASCII range. Useful when you *want* English-like text and no more:

```
"item\x{0660}" =~ /\w+\d/a;   # does not match - \d is [0-9] under /
↪a
"item0"           =~ /\w+\d/a; # matches
```

/aa - strict ASCII

/aa (doubled) additionally prevents case-insensitive matches crossing the ASCII/non-ASCII boundary. Under plain /ia, K could match \x{212A} (KELVIN SIGN) because the Kelvin sign casefolds to k. Under /iaa it does not:

```
"K"           =~ /k/i;           # matches
"\x{212A}"    =~ /k/i;           # matches - Kelvin sign casefolds to 'k'
"\x{212A}"    =~ /k/iaa;         # does NOT match - ASCII-only i
```

/aa is the strict form when the input is genuinely ASCII-only and you want the engine to refuse Unicode matches even under /i.

/l - locale semantics

\d, \w, \s, case folding, and [[:alpha:]] follow the current POSIX locale. Only sensible after use locale. Rarely what you want - locale behaviour is implementation-specific and harder to reason about than /a or /u. Provided for legacy code that genuinely needs locale-dependent matching.

/d - dual-mode semantics

Under /d, \d, \w, \s depend on whether the target string is flagged as UTF-8 internally. This is the source of most “works on my machine” Unicode regexp bugs. **Avoid /d in new code.** The default under use v5.12 is /u.

If you inherit code where /d matters, the failure mode is: the same input string produces different match results depending on whether some upstream operation flagged the string UTF-8. The fix is use feature 'unicode_strings' (or use v5.12+), which makes the rules independent of the UTF-8 flag.

Which modifier is in effect?

Resolution order:

1. Explicit modifier on the regex (/a, /u, /l, /d).
2. use re '/u' (or other) in the lexical scope.
3. use locale selects /l.
4. use feature 'unicode_strings' (and use v5.12+) selects /u.
5. Otherwise, /d (the dual-mode fallback; avoid).

Programs should use use v5.12; (or later) at the top of each file; this gives you /u, which is what you want.

Pragma interactions

```
use v5.12;                               # implicit feature 'unicode_
→strings'
use feature 'unicode_strings';           # explicit
```

(continues on next page)

(continued from previous page)

```
use re '/u';           # set default modifier for this
↳scope
```

use feature 'unicode_strings' (included in use v5.12) makes all strings Unicode for character-class purposes, regardless of whether they are marked UTF-8 internally. Turn it on at the top of every file.

Case folding

Under /iu (the default), case-insensitive matching uses the full Unicode case-folding tables:

```
"Grüße" =~ /grüsse/iu;      # matches - ß folds to 'ss'
"Σίγμα" =~ /σίγμα/iu;       # matches - upper Σ ↔ lower σ
"İstanbul" =~ /istanbul/iu;  # matches - dotted I folds
```

Under /ia, fold only ASCII letters. Under /iaa, plus the Kelvin/K rule above.

The folding happens at compile time - pperl does not lowercase the input string at match time.

The \b{...} word-boundary variants

Beyond plain \b, Perl offers semantic variants that handle real-world text better:

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

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

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

The [anchors and assertions](#) chapter has the broader boundary discussion; this chapter is the home for the Unicode-aware variants because their definitions are Unicode-driven.

Script runs

Different writing systems have visually-similar characters. The Latin a, the Cyrillic a (U+0430), and the Greek α (U+03B1) look indistinguishable. Used together - typically in a URL or identifier - they enable a *homograph attack*:

paypal.com

Could be all Latin (legitimate). Could be a Cyrillic a embedded in otherwise-Latin text (a phishing attack pointing to a different domain). The browser's URL renders identically; the bytes do not.

A *script run* is a sequence of characters all from the same Unicode script (per `Script_Extensions` and Unicode UTS 39). Most legitimate words are script runs; mixed-script strings are rare outside specific multilingual contexts and almost always suspicious.

Perl provides four constructs to require that a matched sub-pattern be a script run:

Form	Meaning
<code>(*script_run: PAT) / (*sr: PAT)</code>	match PAT, then check it is a script run
<code>(*atomic_script_run: PAT) / (*asr: PAT)</code>	same, but atomic (faster, less backtracking)

```
# Match a domain label only if it is a script run:
$label =~ /( *sr: \w+ )/x or warn "mixed-script label";

# Atomic version: faster on adversarial input.
$label =~ /( *asr: \w+ )/x or warn "mixed-script label";
```

The atomic form is what you want in most production code - it prevents the script-run check from being defeated by catastrophic backtracking on a malicious input. `(*sr: ...)` is the non-atomic equivalent (`( *sr: (?>PAT) )` is the same as `( *asr: PAT )`).

Script run rules

A sequence is a script run if and only if all of:

1. **No code point in the sequence has the `Script_Extension` property `Unknown`.** This excludes private-use and surrogate code points; you cannot smuggle them through.
2. **All characters come from the `Common` script, the `Inherited` script, and at most one other script.**
3. **All decimal digits come from the same set of ten.** Many scripts have their own digits (Arabic-Indic, Devanagari, etc.); a string mixing 1 (ASCII) with ١ (Arabic-Indic one) is *not* a script run, even if other characters are compatible.

Three pseudo-scripts get special handling:

- **Common** - punctuation, ASCII digits 0–9, mathematical symbols, emoji, full-width digits. These can appear in any script run *except* that all decimal digits in the sequence must still come from the same set of ten.
- **Inherited** - combining marks (accents, diacritics). These attach to the previous character and inherit its script.
- **Unknown** - unassigned code points and similar. A single-character string of unknown is allowed; a longer string containing one is not.

The relaxation for Common and Inherited is what allows real-world text to be a script run despite punctuation and combining marks. The strict-equality rule for digits is what defeats homograph attacks that mix digits from look-alike scripts.

When script runs help

- **URL parsing:** refuse domain labels that are not script runs.
- **Identifier validation:** programming-language identifiers should normally be script runs.
- **Form input sanitisation:** real names are script runs; mixed-script names are usually attacks or test data.

Script runs are a 2018-and-later feature. They are what Perl's regex engine adds to a problem the rest of the engine cannot solve directly: a property of the *whole* match, not of any single character.

Cross-engine: Unicode coverage

Feature	Perl 5.42	PCRE2	Emacs (limited)	POSIX	RE2 / Go
\p{General_Category}	yes	yes	partial	no	yes
\p{Script}	yes	yes	partial	no	yes
\p{Property=Value}	yes	yes	partial	no	yes
\X grapheme cluster	yes	yes	no	no	yes ((?-X: . . .) is something else)
\b{wb}, \b{sb}	yes	no	no	no	no
Script Runs (*sr:...)	yes	no	no	no	no
Default to Unicode \d/\w	under /u	no	no	no	no (Unicode under (?u))

Perl is a leader here: \b{wb} and Script Runs are essentially unique to Perl among the comparison set. RE2 / Go and PCRE2 both default to ASCII for \d, \w, \s; RE2 enables Unicode under the (?u) flag. PCRE2 has property support but lacks the boundary variants and script runs. Emacs has property support that varies by build.

The full table is in the *cross-engine* chapter.

Encoding vs. character

A Perl string is always a sequence of code points. The bytes on disk are the I/O layer's concern - use `utf8` for source- code literals, `:encoding(UTF-8)` on filehandles, and `decode/encode` from Encode when crossing the boundary. Patterns never operate on the encoded bytes; they always operate on code points.

```
use utf8;                # source code is UTF-8
use feature 'unicode_strings';
use open ':std', ':encoding(UTF-8)'; # STDIN/STDOUT/STDERR

while (my $line = <>) {
    $line =~ /\p{Lu}/ and print "has uppercase letter\n";
}
```

Get the I/O right once, and patterns just work on characters for the rest of the program.

See also

- The *character classes* chapter - non-Unicode class basics, (?[...]).
- The *anchors and assertions* chapter - \b, \B, plus the \b{...} family in their boundary role.
- The *modifiers* chapter - /i, /x, /m, /s, and how they combine with charset modifiers.
- The *cross-engine* chapter - Unicode coverage across regex engines.
- *quotemeta* - Unicode-aware metacharacter escaping.

Performance

Most regexps run in roughly linear time over the string. A well-written pattern that matches is fast; a well-written pattern that fails is also fast. The trouble comes from patterns that *could have* matched in many different ways - the engine explores those possibilities one after another, and pathological patterns can explore exponentially many.

This chapter is about recognising that trouble, avoiding it, and when you cannot avoid it, damping it with possessive quantifiers, atomic groups, and careful structure. The second half is about the machinery the engine quietly applies on your behalf, and how to write patterns that let those optimisations fire.

Perl uses a *Traditional NFA* - an engine that walks the pattern under the direction of the input rather than the other way around, saving its place at every choice point so it can resume after a dead end. That property is what makes captures, backreferences, and lookahead possible; it is also what makes the worst case exponential. Everything in this chapter is a corollary of that one fact.

How backtracking works

A regexp engine walks the string and the pattern in lockstep. When it reaches a choice - an alternation, a quantifier, an optional group - it makes a decision, remembers the choice on a stack, and continues. If a later position in the pattern fails, it pops the most recent choice, tries the next available decision, and resumes. This is *backtracking*.

Every ?, *, +, {...}, and | is a potential backtrack point. A short string and few choice points: fast. A long string, nested choice points, and no match: exponential.

A worked example from perlre. To pull the word following foo out of Food is on the foo table.:


```
$_ = "Food is on the foo table.";
if (/\\b(foo)\\s+(\\w+)/i) {
    print "$2 follows $1.\\n";
}
# table follows foo.
```

The engine first matches `\\b(foo)` against `Food`, loading `$1` with `Foo`. It then looks for `\\s+` and finds `d` - failure. It backtracks, advances the start position, and continues searching until it finds the literal `foo` followed by whitespace, then a word.

A more revealing case. The pattern says “everything between `foo` and `bar`”:

```
$_ = "The food is under the bar in the barn.";
if (/foo(.*)bar/) {
    print "got <$1>\\n";
}
# got <d is under the bar in the >
```

Greedy `.*` ran all the way to the end of the string, then crawled back left until `bar` could finally match - at the *last* bar, not the first. Use `.*`? to stop at the first.

perlre has a longer version of this with eight variants - most wrong, all instructive - under Backtracking. The lesson is the same in each: a greedy quantifier always overshoots first; the correct *shape* (which anchors? non-greedy or negated class?) is what fixes the match, not adjusting the quantifier in place.

The state-saving cost

Friedl’s mechanical model is worth keeping in mind. For an NFA, the *saved-state count* is roughly the number of characters that quantifier-governed subexpressions matched. `[0-9]+` against `1234` saves four states - one per digit, marking each as a “match could end here” point. When the engine eventually fails further along the pattern, it pops those four states one at a time, trying to find a way to make the match succeed.

A subtle detail: the `+` quantifier *requires* its first match, so it doesn’t save a state for “match nothing here”. `*` does, because zero matches are allowed. That asymmetry shows up in benchmark microcomparisons of `+` vs `*` patterns; on inputs where the quantifier really must match at least one thing, `+` is slightly faster because it has one fewer state to manage on failure.

Catastrophic backtracking

Consider this pattern trying to match a long string of a’s:

```
"aaaaaaaaaaaaaaaaaaaaa" =~ /(a+)+b/;
```

The pattern cannot match - there is no `b`. But the engine has to *prove* it. Each `a` in the string can be assigned to the inner `+` or the outer `+` in many ways. For a 20-character string that is millions of combinations; for a 30-character string, billions. The engine tries them all.

The general shape to watch for: **nested indeterminate quantifiers on overlapping classes**.

```
/(a+)/;           # dangerous
/(a|b)*c/;       # dangerous on a+b strings without trailing c
/(\w+)*\s/;      # dangerous on a run of word chars with no
↳space
/("[^"]|\\.)*/;   # dangerous on long unterminated strings
```

Every one of these has the same shape: an outer quantifier on a group whose inner content is itself indeterminate. Friedl’s canonical scary example, the doublequoted-string regex `"(\. | [^"\\]+)*"` applied where it cannot match, is documented to take roughly 3.2×10^{29} backtracks before giving up - many universe-lifetimes if the engine really did try them all. In practice the recursion-depth ceiling fires first; the program does not so much hang as fall over in slow motion.

The disease is the same in every case. For each extra character the input grows, the number of ways the engine can apportion characters between the inner and outer quantifier *doubles*.

The cure is *determinism*: write the regex so that for any input there is exactly one way the engine can divide the work between inner and outer quantifier. The next sections cover the three techniques that achieve this.

Possessive quantifiers

A possessive quantifier (`++`, `*+`, `?+`, `{n,m}+`) is greedy *and* never gives back. It tells the engine: “once I’ve matched these characters, they are mine; do not revisit this decision on failure.”

```
/(a++)+b/;      # at most one backtrack point
/(\w++)*\s/;    # bounded - \w++ takes all word chars in one shot
```

The trade-off: if a later part of the pattern needs those characters back, the match simply fails. Possessive is correct when there really is no other way for the match to succeed at that prefix.

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

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

The possessive notation is shorter; the atomic-group notation generalises further (you can wrap arbitrary alternations, not just quantified atoms).

A consequence of “possessive equals atomic”: some combinations make no sense and Perl rejects them.

Illegal form	Equivalent legal form
<code>X??+</code>	<code>X{0}</code>
<code>X+?+</code>	<code>X{1}</code>
<code>X{n,m}?+</code>	<code>X{n}</code>

Non-greedy and possessive are contradictory. Use `X{0}`, `X{1}`, or `X{n}` if you really wanted to match exactly that many.

Atomic groups

`(?>...)` is a non-capturing group whose contents, once matched, are committed. Like a possessive quantifier, no backtracking happens inside it. The difference is that atomic groups wrap *arbitrary* patterns, not just quantified atoms.

```
"aaab" =~ /a*ab/;      # matches - the a* gives back one 'a'
"aaab" =~ /(??>a*)ab/;  # does not match - a* takes all, none to
→give
```

Atomic groups are the most general tool for controlling backtracking. Use them around a block of pattern that should either fully match or not participate.

A classic application: balanced parentheses, fast on bad input.

```
# Unbounded: dangerous on unbalanced input
/ \ ( ( [^()]+ | \ ([^()]*\ ) )+ \ ) /x;

# With atomic group: fast failure on bad input
/ \ ( ( ??> [^()]+ ) | \ ([^()]*\ ) )+ \ ) /x;
```

The inner `(??>[^()]+)` says: gobble as many non-paren characters as possible and don't come back. That removes the ambiguity that makes nested `+` expensive.

`(?>...)` does not disable backtracking *outside* it. The engine can still backtrack past the construct as a whole, just not into it. So `((?>a*)|(?>b*))ar` against bar still matches: the outer alternation can backtrack between branches.

Nested atomic groups are *not* no-ops. Each extra layer further constrains what the inner pieces can give up. `(?>a[bc]*c)` matches `abc`; `(?>a(?>[bc]*c)` does not, because the inner `(?>[bc]*)` ate the trailing `c` and refuses to release it.

The long-form spelling `(*atomic:...)` is also accepted; it reads better in pattern code that already uses verbs like `(*positive_lookahead:...)`.

Matching quoted strings - the canonical case

The “match a quoted string with backslash escapes” problem is the one everyone gets wrong on the first try.

Naive:

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

Given a long unterminated string like "aaaaaaaaa (no closing quote), this backtracks quadratically over every partition of the a's between the two alternatives. The safe form:

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

The inner ++ takes all non-quote, non-backslash characters in one bite. The outer ++ refuses to give back across iterations. A failed match fails in linear time. The atomic-group equivalent is identical in behaviour:

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

- and demonstrates why the possessive form is preferable when it applies. It's terser without losing anything.

Unrolling the loop

The single most useful technique for replacing a non-deterministic pattern with a deterministic one. Friedl named it; engines have internalised it; every working regex programmer should know it.

The pattern shape:

```
opening normal* (special normal*)* closing
```

- where normal and special are subexpressions that consume disjoint sets of characters. The construction guarantees there is exactly one way the engine can split the input between them, so no exponential exploration ever begins.

Three rules, all mandatory:

1. **special and normal cannot start a match at the same position.** If both can match the same character, the engine has freedom to choose; that freedom is exactly what causes exponential blowup.
2. **special cannot match the empty string.** A special that matches nothingness is invisible to the engine and removes the checkpoint that makes the construction deterministic.
3. **One application of special cannot be coverable by multiple applications of special.** If special is \s+ and the input has three consecutive spaces, the engine could match them as one special, three specials, or various combinations - exponential blowup is back.

Worked example: doublequoted strings

Naive form, prone to catastrophic backtracking:

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

Unrolled form, no nested ambiguity:

```
/"[^\\"\\]*(\\. [^\\"\\]*)*"/;
```

[^\\"\\]* is the normal* - it eats every non-special character. \\. is the special - it consumes a backslash plus exactly the following character. The two cannot overlap

(normal* cannot contain \\; special cannot match a non-\\ character), and special always consumes exactly two characters.

The unrolled form runs in linear time on any input - matching or not. The naive form is exponential on inputs with many backslashes and no closing quote.

The same problem with possessive quantifiers also runs in linear time:

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

Pick whichever the rest of your team reads more easily. Both work.

Worked example: C comments

The “match a C comment, surviving asterisks inside” problem. Naive comment match `\\/\\*.*?\\*/` works but is slow on long comments because `. *?` has to expand one character at a time.

Unrolled:

```
/\\/\\*[^*]*\\*+( [^/*] [^*]*\\*+)*\\//;
```

Reading from left to right:

- `\\/\\*` - opening `/*`.
- `[^*]*` - normal*, zero or more non-asterisk characters.
- `\\*+` - at least one asterisk.
- `( [^/*] [^*]*\\*+)*` - zero or more iterations of: a non-slash, non-asterisk character, then more non-asterisks, then more asterisks. (The non-slash bit is what stops the run: an asterisk followed by a slash is the terminator we want.)
- `\\/` - closing slash.

The construction makes every iteration of the outer `*` consume at least one character that no other iteration could have consumed. Result: linear time, no catastrophic-backtracking pattern.

The pedagogical value of working through this is in the *thought process*, not memorising the final regex. The asterisks-inside complication forces the extra `[^/*]` term; recognising why is what teaches you to spot the next case.

Worked example: the freeflowing regex

When you have multiple “skip me” subexpressions (strings, comments, constants in a parser-like context), build a single regex with all of them in alternation and an `OTHER+` clause for the bulk of the data:

```
my $DOUBLE = qr/"[^"\\]*(?:\\. [^"\\]*)"/;
my $SINGLE  = qr/'[^'\\"]*(?:\\. [^'\\"']*)'/;
my $COMMENT = qr\\/\\*[^*]*\\*+( [^/*] [^*]*\\*+)*\\//;
my $LINECOM = qr\\/\\/[^\n]*/;
my $OTHER  = qr/[^\s'"/]+/;
```

(continues on next page)

(continued from previous page)

```
while ($source =~ / $DOUBLE | $SINGLE | $COMMENT | $LINECOM | $OTHER
↪/xg) {
    ...
}
```

The \$OTHER arm consumes the bulk of the input in long runs, which means the engine spends almost all its time inside one linear-time loop, not on the bump-along trying alternatives at every uninteresting character. The technique is decisive on parser-like workloads.

Reducing choice

If your pattern works but is slow, look for spurious choice:

- **Anchor where you can.** `^pat` and `pat$` let the engine reject most starting positions immediately. `\A` is even better than `^` when you mean “start of string” - the engine can refuse to retry at every position rather than only at line starts.
- **Replace alternation of single characters with a class.** `/a|b|c/` is slow compared to `/[abc]/`; the class is a single choice, the alternation is three.
- **Put the common alternative first.** Alternation is tried left-to-right, and Perl stops at the first success. If 90% of your inputs hit alternative 3, reorder.
- **Expose common prefixes.** `/this|that|them/` defeats the fixed-string optimisation; `/th(?:is|at|em)/` hands the engine the literal `th` to scan for first.
- **Prefer character classes to `.`** `[^\n]` is cheaper than `.` because the engine knows it will never match across lines. Same for `[^"]` instead of `.` between quotes.
- **Negated character classes beat `.`** The non-greedy form has to leave the inner loop on every iteration to test what follows; the negated class participates in the simple-repetition fast path. `<([^\>]+)>` is several times faster than `<(.*?)>` and also more correct: the negated class *cannot* cross a `>` even under backtracking pressure.

Compile once, match many

Every time a pattern with interpolated variables is used, Perl checks whether the text of the pattern changed. If it has not, the compiled form is reused. For patterns that really are fixed, the overhead is small but nonzero.

`qr` compiles a pattern once and produces a reusable object:

```
my $word = qr/\b[a-z]+\b/;
for my $text (@corpus) {
    while ($text =~ /$word/g) {
        ...
    }
}
```

Using `qr` in a hot loop saves the recompile-or-check step and makes the code clearer. It also lets you build complex patterns out of named pieces, which is decisive for any regex

over about ten lines.

The `/o` modifier (“compile once”) was the Perl 4 mechanism for this. It still works but is rarely the right tool today: `qr//` is clearer, scopes better, and composes.

Avoiding capture overhead

Capturing groups cost a little - the engine has to track the start and end positions, and the surrounding optimisations (simple repetition in particular) may be disabled. If you don’t use the capture, don’t ask for it.

```
# Capturing even though $1 is never read:
/(\d{4})-(\d{2})-(\d{2})/;

# Non-capturing:
/(?:\d{4})-(?:\d{2})-(?:\d{2})/;
```

The `/n` modifier turns every `(...)` into `(?:...)` without editing the pattern - useful for long patterns. Named captures still capture under `/n`.

Even non-capturing parens may suppress the simple-repetition optimisation in some engines, so `(?:[a-z])+` is occasionally slower than `[a-z]+`. Drop unnecessary grouping; flatten patterns where you can without hurting readability.

Recursive patterns

Perl regexps support recursion - a pattern can refer back to one of its own groups. The construct is `(?R)` (or `(?0)`) for the whole pattern, `(?N)` for group `N`, `(?-N)` and `(?+N)` for relative references, and `(?&NAME)` for named groups.

A recursive pattern for a balanced parenthesis, any depth:

```
my $bal = qr/
    (? (DEFINE)
        (?<paren>
            \(                # opening paren
            (?:
                [^()]+        # run of non-paren, possessive
                | (?&paren)    # or a balanced sub-group, _
            ↪recursively
            ) *+
            \)                # closing paren
        )
    )
    (?&paren)
/x;

"((a)(b(c)))" =~ /^$bal$/;  # matches
```

A named subpattern declared in a `(?(DEFINE) . . .)` block recurses into itself, not into the whole enclosing pattern. Using `(?R)` here would force every level of recursion to re-apply the call site’s anchors (`^` and `$`), which is rarely what you want.

Recursion makes genuinely recursive structures matchable without falling out of the regex DSL. It is also a strong signal that you might be reaching for the wrong tool. A real parser is often clearer; reach for recursive regexps when you want a one-line match-or-fail decision, not a parse tree.

There is a recursion-depth limit compiled into the engine (it can be raised by rebuilding from source). Patterns that recurse without consuming input fail immediately rather than running forever - the engine detects the cycle.

The interaction with capture groups deserves attention. When a group recurses into itself, captures set inside the recursion are *not* visible to the caller after the recursion returns. This is the practical reason most recursive patterns use possessive quantifiers and atomic groups for the inner content: capturing it serves no purpose because the data won't be there when the outer match finishes.

Embedded code

(?`{ code }`) runs Perl code at the point the engine reaches it. The code's return value is set as `$^R`. Useful for instrumentation and for assertions whose condition is more naturally written in Perl than in regex syntax:

```
my $tries = 0;
"aaaaab" =~ /a*(?{ $tries++ })b/;
print "engine tried $tries times\n";
```

Two important properties:

- The block is *backtracking-safe* if you assign to local variables. `local $cnt = $cnt + 1` undoes itself when the surrounding match backtracks past the block. A bare assignment does not.
- The block sees the same lexical scope it was compiled in. A `qr//` containing `(?{ ... })` closes over the lexicals in scope at compile time, *not* at match time. This is the same rule that governs closures everywhere else in Perl.

`$^N` is particularly useful inside `(?{ ... })` - it holds the text matched by the most-recently-closed capture group, so you can save matches without counting parentheses:

```
$_ = "The brown fox jumps over the lazy dog";
/the (\S+)(?{ $color = $^N }) (\S+)(?{ $animal = $^N })/i;
print "color = $color, animal = $animal\n";
# color = brown, animal = fox
```

The big cost: `(?{ ... })` *globally disables* certain optimisations in the pattern. Patterns containing it can run much slower than patterns without it, even on inputs where the code block runs zero times.

`(*{ code })` is the *optimistic* form, introduced in Perl 5.37.7. Same syntax, same behaviour, except it does not disable optimisations. The trade-off: the engine may execute the block fewer times than the strict semantics of `(?{ ... })` would require. On a failing match it might not run at all. Use `(?{ ... })` when you depend on side effects firing in a documented order; use `(*{ ... })` when you don't.

(`??{ code }`) is the *postponed regex*: the code's return value is treated as a pattern, compiled if it is a string (or used directly if it is a `qr//` object), and matched at that position. It enables genuinely runtime-shaped patterns:

```
my $inner = '(.)\1';
"ABBA" =~ /^.(.)??{ $inner }\1/;
print $1;    # 'A'
```

The inner pattern has its own capture state - captures inside (`??{ ... }`) are not visible to the outer pattern. For balanced parens or other recursive structures, (`?R`)/(`?N`) is more efficient and also clearer.

User-supplied input must never reach (`?{ ... }`) or (`??{ ... }`). Perl forbids it by default; use `re 'eval'` removes the safety catch and you should not.

Embedded-code execution frequency

The exact number of times (`?{ ... }`) and (`??{ ... }`) execute during a match is *unspecified*. The engine guarantees only:

- On a successful match, blocks on the accepting path execute in left-to-right order, the appropriate number of times for the quantifier they sit under.
- Blocks may execute zero, one, or many times during failed matches, depending on what optimisations the engine applies.

For example, in `"aaabcbdeeeee" =~ /a(?{print "a"})b(?{print "b"})cde/`, the precise number of a's and b's printed is unspecified for *failed* matches. For a successful match each block runs at least once on the accepting path; if b was printed, an a was necessarily printed at least once before it.

Quantified blocks behave intuitively on success:

```
"good" =~ /g(?o(?{ print "o" }))*d/;
# prints o twice
```

Code blocks intended to count, log, or trace must use the (`?{ ... }`) form (deterministic on success); blocks that are purely advisory may use (`*{ ... }`) (and accept the lower call count in exchange for not breaking optimisations).

Special backtracking control verbs

Perl provides verbs that direct the engine more precisely than ordinary atomic groups can. They all use the form (`*VERB`) or (`*VERB:NAME`). They are zero-width (they consume no input) and only do something when *backtracked into* on failure.

The verbs interact with two package variables: `$REGERROR` and `$REGMARK`. After a match involving any verb that takes a NAME:

- `$REGMARK` is set to the name of the last (`*MARK:NAME`) executed, on success.
- `$REGERROR` is set to the name of the verb that caused the failure, on failure.

These are package globals (not magic variables; not localised automatically); use `local` if you need scoping.

(*PRUNE) and (*PRUNE:NAME)

Prunes the backtracking tree at the current point. After A(*PRUNE)B: if B fails, the engine does not backtrack into A. The whole match fails *at the current starting position*; it does not advance and retry.

```
'aaab' =~ /a+b?(?{ print "$&\n"; $count++ })(*FAIL)/;
# Prints aaab, aaa, aa, a, aab, aa, a, ab, a - 9 times.

$count = 0;
'aaab' =~ /a+b?(*PRUNE)(?{ print "$&\n"; $count++ })(*FAIL)/;
# Prints aaab, aab, ab - 3 times.
```

(*PRUNE) is a more powerful (?>...): it controls backtracking across pattern segments rather than within a fixed sub-pattern.

(*SKIP) and (*SKIP:NAME)

Like (*PRUNE), plus: signals that the text leading up to the verb cannot be part of *any* match. The engine advances the start position to where the verb fired and retries from there:

```
'aaabaaab' =~ /a+b?(*SKIP)(?{ print "$&\n"; $count++ })(*FAIL)/;
# Prints aaab, aaab - 2 times.
```

If (*SKIP:NAME) fires and a (*MARK:NAME) was set earlier, the new start position is the mark's position, not the verb's. This is the only sane way to do “if branch X failed, advance past where branch X started” in a single pattern.

(*MARK:NAME)

A named pseudo-checkpoint. Does nothing on its own - but (*SKIP:NAME) uses it as a target, and \$REGMARK after a successful match holds the most recently executed mark name. The shorthand (*:NAME) is identical to (*MARK:NAME).

This lets a pattern record which alternative branch matched without using a separate capture group per branch:

```
/(?:foo(*MARK:F)|bar(*MARK:B)|baz(*MARK:Z))/;
# After a successful match, $REGMARK is "F", "B", or "Z".
```

The optimiser handles MARK/SKIP more efficiently than /(?:(foo)|(bar)|(baz))/ because it does not have to track three parallel capture states.

(*THEN) and (*THEN:NAME)

Inside an alternation: on failure of the rest of the pattern, backtrack to the next alternative of the *innermost enclosing group with alternations*, skipping branches above that level. Outside an alternation, equivalent to (*PRUNE). Loosely: “this branch and only this branch; if it fails, try the next sibling branch.”

```
/(COND1 (*THEN) ACT1 | COND2 (*THEN) ACT2 | COND3 (*THEN) ACT3)/x;
```

Reads like an if/elsif/else inside the regex.

(*COMMIT) and (*COMMIT:arg)

The strongest brake. Like (*SKIP), but instead of advancing the start position, the engine *gives up entirely*. No further attempts to find a match anywhere in the string.

```
'aaabaaab' =~ /a+b?(*COMMIT)(?{ print "$&\n"; $count++ })(*FAIL)/;
# Prints aaab - 1 time.
```

Use when a partial match has consumed enough to prove that no match exists anywhere later either.

(*FAIL) / (*F)

Always fails. Equivalent to (?!) but more readable. Useful only combined with (?{ . . . }): when you want to count or log every attempted match without ever accepting one. (?!) is internally optimised into (*FAIL).

(*ACCEPT) and (*ACCEPT:arg)

The opposite of (*FAIL). Always succeeds, ending the match immediately. Inside nested patterns or recursion, only the innermost pattern is ended. Capture groups that have opened but not yet closed are marked as ending where (*ACCEPT) fires:

```
'AB' =~ /(A (A|B(*ACCEPT)|C) D)(E)/x;
# matches; $1 is "AB", $2 is "B", $3 is undef
```

(*ACCEPT) is most useful inside (??{ . . . }) or (? (condition) . . .) constructs where you want a fast-path success without finishing the pattern.

Zero-length-match termination

A repeated pattern that can match the empty string is a problem. Trivially:

```
'foo' =~ m{ ( o? )* }x;
```

Each iteration of the outer * matches the empty string, the position does not advance, the next iteration matches empty again - infinite loop in waiting. Perl breaks this loop in two different ways depending on which kind of repetition is involved.

Lower-level loops - the quantifiers *, +, {n,m} - are *interrupted* when the engine detects that an iteration matched zero-length. The construct

```
(?: NON_ZERO_LENGTH | ZERO_LENGTH )*
```

is silently treated as

```
(?: NON_ZERO_LENGTH ) * (?: ZERO_LENGTH ) ?
```

The zero-length branch can run, but only *once*, after the non-zero branch is done.

Higher-level loops - the `/g` modifier on `m//` and `s///`, and the `split` operator - preserve a flag across iterations recording whether the previous match was zero-length. After a zero-length match the next attempt is forbidden to *also* be zero-length; the engine takes the second-best match instead. The canonical demonstration:

```
$_ = 'bar';  
s/\w??/<$&>/g;  
# Result: <><b><><a><><r><>
```

The non-greedy `\w??` prefers an empty match. After producing one, the next iteration is forbidden from being zero-length too, so the engine takes the *second*-best match - a single `\w` character. Then another empty match, then another character, and so on, alternating until the string is consumed.

The state of “previous match was zero-length” is associated with the matched string and is reset by any assignment to `pos`. Zero-length matches at the end of the previous match are also ignored during `split`, which is why `split //` produces individual characters and not a trailing empty fragment.

Combining patterns: better and worse

perlre’s “Combining RE Pieces” section gives a precise account of which match the engine prefers when a pattern admits more than one. The rules are mechanical, and worth knowing for the cases where intuition fails.

For two pattern pieces concatenated, `ST`:

- If `S` matches `A` better than `A'`, then `AB` is better than `A' B'` (the leftmost piece’s preference dominates).
- If `S` matches both `A` and `A'` equally, `T`’s preference decides.

For alternation, `S | T`:

- `S` matching is always better than only `T` matching (left-to- right choice - Traditional NFA semantics).

For greedy and non-greedy quantifiers:

- `S{n,m}` is equivalent to trying `S{m}`, then `S{m-1}`, ..., then `S{n}` (greedy: maximum first).
- `S{n,m}?` is equivalent to trying `S{n}`, then `S{n+1}`, ..., then `S{m}` (non-greedy: minimum first).

For atomic groups, `(?>S)`:

- *Only the best match for `S` is considered*; backtracking is forbidden inside.

For lookahead, `(?=S)` / `(?<=S)`:

- *Only the best match for `S` is considered* (the same as atomic). This matters only if `S` has capturing parens whose contents are used outside.

The above rules describe *which match is chosen at a fixed starting position*. One additional rule covers position selection: **a match at an earlier position is always better than a match at a later position** - the bump-along property is more important than any preference within a single position.

Engine internals - what the optimiser does for you

A regex engine's compiler runs a small army of analyses on the pattern before matching begins. Knowing them lets you write patterns that *enable* them, rather than ones that defeat them.

First-character discrimination

If the pattern must begin with one of a small set of characters (`am|pm` $\Rightarrow$ `[ap]`), the bump-along can scan the input for one of those characters using a fast inner loop, invoking the full engine only at candidate positions.

A pattern starting with `.`, or with a top-level alternation the engine cannot see through, defeats this. Modern engines (PCRE2, RE2, the Rust regex crate) go far beyond first-character discrimination - Boyer-Moore on prefix literals, byte-level SIMD scanning, Aho-Corasick fallback for many literal alternatives. Perl 5.42 does the simpler form well; the more aggressive techniques are why a benchmark may show RE2 outperforming Perl on inputs dominated by literal scanning.

Fixed-string check

If the pattern requires some literal substring (e.g. `Subject: in ^Subject: (Re: )?(.*)`), the engine uses Boyer-Moore to skip to candidate positions cheaply. A naive pattern like `/this|that|these|those/` defeats this; rewriting as `/th(?:is|at|ese|ose)/` exposes the common prefix `th` for the optimiser to find.

Simple-repetition fast path

`x*`, `[a-f]+`, `.?` and other “single atom, simple quantifier” constructs are dispatched to a fast inner loop that bypasses the general engine machinery. Wrapping the atom in *capturing* parens disables this. Even non-capturing parens may, depending on the exact construct.

Length cognizance

If the pattern requires at least `N` characters, the engine can refuse to attempt matches in the last `N` positions of the string, and refuse to attempt matches at all in strings shorter than `N`.

Anchor implication

A pattern starting with `^` is attempted only once (or once per logical line under `/m`). Less obviously, a pattern starting with `.*` followed by an anchorable subpattern *can* be implicitly anchored, because by the time `.*` has consumed the whole string, no later starting position can possibly succeed. Modern engines recognise this; do not write `.*` at the start unless you mean it.

Compile caching

Compiling a regex to its internal form is non-trivial. Perl caches compiled patterns aggressively - patterns embedded in code are compiled once when the surrounding sub is compiled, and patterns assembled from interpolated variables are recompiled only when the resulting text actually changes. `qr//` makes this explicit. The `/o` modifier was the older mechanism; `qr//` has superseded it in idiom.

Matching many strings efficiently

If you have a fixed list of strings to match and the list is stable, building one big alternation is often the simplest win:

```
my @words = qw(the quick brown fox);
my $any = join '|', map quotemeta, @words;
my $re = qr/\b(?:$any)\b/;
for my $line (@lines) {
    next unless $line =~ /$re/;
    ...
}
```

`quotemeta` protects each word from containing accidental metacharacters. The `(?:...)` is non-capturing - unnecessary captures would cost more than they pay. If the list is large (thousands of strings), a specialised module that builds an Aho-Corasick or trie matcher will beat a giant alternation; the built-in engine compiles the alternation to an NFA that is fast but not asymptotically optimal for that workload.

Measuring

Claims about regexp speed are cheap; measurements are not. `Benchmark::timethese` makes it easy to compare two forms on real data:

```
use Benchmark qw(timethese);

my $text = "... " x 10000;

timethese(1000, {
    greedy      => sub { $text =~ /"(?:[^\\"\\]|\\.)*"/ },
    possessive => sub { $text =~ /"(?:[^\\"\\]+|\\.)*+"/ },
});
```

Always profile before optimising. “Obviously slow” patterns often aren’t; “obviously fast” ones sometimes are pathological on real input. Friedl’s framing remains the best summary: the fastest program is the one that finishes first.

Rules of thumb

- Write the pattern that reads best; profile.
- If matching is slow, look for nested quantifiers on overlapping classes.
- Fix them with possessive quantifiers, atomic groups, or unrolling.

- Anchor with `^`, `$`, `\A`, `\z`, or `\G` whenever the pattern can only legitimately start or end in those positions.
- Prefer classes to alternations of single characters.
- Prefer negated classes to non-greedy quantifiers when they describe the same set.
- Use `qr//` for patterns that run in a loop.
- Use `(?:...)` for groups you don't capture, or `/n` to suppress captures wholesale.
- Use `(*{...})` instead of `(?{...})` when the code block is advisory.
- When a real parser would be clearer than a pattern, write the parser.

See also

- The [quantifiers](#) chapter - greedy, non-greedy, possessive forms.
- The [groups and captures](#) chapter - atomic groups, recursive subpatterns, conditional patterns.
- The [alternation](#) chapter - ordering, leftmost alternative wins.
- The [substitution](#) chapter - `/g` and zero-length match termination in substitution.
- `qr` - precompile patterns for repeated use.
- `pos` - read or set the `/g` match position; also resets the zero-length-match flag.

Cross-engine comparison

Perl regexps are not the only kind. A reader who knows Perl well will, sooner or later, meet a regex written for another tool - sed, awk, an Emacs Lisp buffer search, a Go service, an embedded PCRE2 library - and be expected to read it. The syntactic differences are usually small, the semantic differences are sometimes substantial, and the architectural differences (which inputs make the engine catastrophically slow) can be decisive.

This chapter is the single page that answers “what does X look like elsewhere?” for the five engine families a Perl-trained reader is most likely to encounter. It is not a porting tutorial; it is the comparison reference.

Engines covered

Five families, picked because each is widespread in its niche *and* materially different from Perl. Engines that are widespread but largely Perl-shaped (Java's `java.util.regex`, Python's `re`, ECMAScript, .NET) are out of scope; the differences are minor and a Perl reader can read them on sight.

Engine	Where you meet it	Family
PCRE2	nginx, PHP preg_*, command-line pcre2grep	Perl-extended
Emacs	Emacs Lisp regex search, M-x replace-reg-exp	Its own family
POSIX BRE	sed (default), grep (default), ed	POSIX Basic
POSIX ERE	awk, egrep / grep -E, sed -E / sed -r	POSIX Extended
RE2 / Go reg-exp	Go's standard library, Cloud Bigtable, code search	Linear-time hybrid

Versions anchored against in this chapter: PCRE2 10.44, Emacs 30.x, POSIX 1003.1-2024, Go 1.22 (RE2 tracks Go's regexp engine). The values in the tables below were verified against these versions; they will drift over time, and engine-specific updates should re-verify before relying on a row.

What's not in scope

The following engines are deliberately omitted from the tables. Each is widespread enough that you may meet it; in each case, the differences from Perl are small enough that a Perl reader can read the regex with little adjustment.

- **Java java.util.regex** - Perl-derived; differences are flag spellings and a few escape conventions.
- **Python re** - Perl-derived; biggest delta is variable-length lookbehind support arrived later than in Perl.
- **ECMAScript** (JavaScript) - Perl-derived; lookbehind only since ES2018, no possessive quantifiers, sticky flag y instead of \G. Modern V8 / SpiderMonkey converge on Perl's feature set.
- **.NET regex** - Perl-shaped, plus *balanced groups* ((?<name-pop>...)), which are unique. Niche enough to skip.
- **Rust regex** - RE2 lineage; the story is essentially Go's with a slightly different feature surface (it has fixed-width lookbehind, RE2 does not).
- **Vim** - different magic-mode semantics; Vim-internal.
- **Boost.Regex** - C++ niche; closely tracks PCRE2.

The five families, in a paragraph each

PCRE2

Perl-Compatible Regular Expressions, version 2 (PCRE2 replaced the original PCRE library in 2015). The reference implementation for “Perl regex outside of Perl”. Reads almost identically to Perl 5.42 patterns and supports nearly the same feature set: back-references, lookaround, atomic groups, possessive quantifiers, recursive subpatterns, named captures, conditional patterns, callouts. PCRE2 also has a JIT compiler that is sometimes faster than perl's.

Where PCRE2 differs from Perl 5.42:

- `(?{...})` and `(??{...})` - Perl runs arbitrary Perl code; PCRE2 has *callouts* `((?C)/(?Cn))` which call back into the host application. Not a security difference; just a different interface.
- A handful of flag spellings differ (PCRE2_CASELESS vs `/i`, etc.) at the API level, not in pattern syntax.
- Recursive subpattern atomicity: `(?R)` is *atomic* in PCRE2 by default; Perl allows backtracking into a recursed group.
- `(*VERB)` set is largely the same, with PCRE2 adding `(*UTF)`, `(*UCP)`, and other startup options.

For most patterns, “looks the same” is correct.

Emacs

GNU Emacs’s regex engine is its own family. On the surface it looks BRE-shaped - `\( . . \)` for grouping, `\{m,n\}` for counts - but inside it has Perl-lite features (non-greedy quantifiers, word boundaries) and Emacs-specific features (syntax classes `\sw`, category classes `\cC`) that no other engine has.

Notable Emacs-specific constructs:

- **Buffer anchors:** `\`` (backtick) is start-of-buffer, `\'` is end-of-buffer. Closer to Perl’s `\A / \z` than to `^ / $`.
- **Syntax classes:** `\sw` matches a “word-syntax” character as defined by the current buffer’s syntax table; `\s-` matches whitespace-syntax. This is mode-dependent - the same regex matches different characters in C-mode vs Lisp-mode.
- **Category classes:** `\cC` matches a character in category C. Used heavily for CJK text classification.
- **No `\d` shorthand** - Emacs requires `[0-9]` or `[[:digit:]]`. (`\w`, `\s`, `\b` exist, but `\d` does not.)
- **No lookahead, lookbehind, or `\K`.**

If you read Emacs Lisp in 2026 you will see `\\(\\)` patterns constantly - that is the doubled escape required because Emacs Lisp string literals consume one level of backslash before the regex parser sees the rest.

POSIX BRE - sed, grep, ed

POSIX Basic Regular Expressions. The engine you get from `sed`, `grep`, and `ed` *with no flags*. The most surprising thing for a Perl reader: the metacharacter set is *smaller*. Bare `+`, `?`, `\{m,n\}`, `(, )`, and `|` are *literal characters*. The metacharacter forms are the backslashed variants: `\+`, `\?`, `\{m,n\}`, `\(, \)`, `\|` (the last is a GNU extension; strict POSIX BRE has no alternation at all).

```
# POSIX BRE, finds the literal text "a+b":
echo 'a+b' | grep 'a+b'

# To match "one or more a", under BRE:
echo 'aaab' | grep 'a\+b'
```

Backreferences (`\1–\9`) work in BRE (this is older than POSIX ERE’s no-backreferences rule). POSIX classes `[[:digit:]]` etc. work. `\<` and `\>` (GNU extension) provide word boundaries. Shortened character escapes like `\d`, `\s`, `\w` are *not* supported.

`grep` and `sed` ship the same engine but flag-default to different families: GNU `grep -P` invokes PCRE2, `grep -E` is ERE, default `grep` is BRE. `sed -E` is ERE, default `sed` is BRE.

POSIX ERE - `awk`, `egrep`, `sed -E`

POSIX Extended Regular Expressions, the engine `awk` defaults to and the one selected by `egrep` (now `grep -E`) and `sed -E` / `sed -r`. Bare `+`, `?`, `{m,n}`, `(...)`, and `|` are metacharacters here; this is the family closest to “what a Perl-trained programmer expects” among the POSIX-conformant set.

What strict POSIX ERE does *not* have, by spec:

- **Backreferences in the pattern** (`\1`, `\2`, ...). They are legal in `sed` substitution replacements but not in match patterns. GNU extensions add them; portable scripts should not rely on this.
- Lookaround.
- Possessive quantifiers, atomic groups.
- `\d`, `\s`, `\w`, `\b` (use `[0-9]`, `[[:space:]]`, etc.).

The most common surprise: an ERE `\(` is a *literal opening paren*. Backslashing punctuation is sometimes a metacharacter and sometimes a no-op, with the rule being the opposite of BRE. Memorise: BRE backslash *enables* metacharacters; ERE backslash *disables* them.

RE2 / Go `regexp`

Google’s RE2 is a fundamentally different architecture. RE2 constructs an NFA at compile time and uses a *lazy DFA* (constructing DFA states on demand and caching them) at match time. The result is *guaranteed linear time* in the input length, no matter the pattern.

The trade-off: RE2 cannot support features that take regular languages out of the regular-language family. Specifically:

- **No backreferences in patterns.** `\1`, `\g{1}`, `\k<name>` - none. A pattern like `(...)\1` simply cannot be compiled.
- **No general lookaround.** RE2 supports anchors and `\b` (zero-width assertions), but not `(?=...)`, `(?<=...)`, `(?!...)`, or `(?<!...)`.
- **No atomic groups, no possessive quantifiers, no recursive subpatterns, no embedded code.**

What RE2 *does* support, and what makes it pleasant for the common case:

- Named captures with the Python-shaped syntax `(?P<name>...)`.
- ASCII by default for `\d`, `\w`, `\s`; Unicode mode enabled with `(?u)`.
- The `(?flags)` and `(?flags:...)` modifier syntaxes.

- A swap-default-greediness flag (?U) - under it, * is non-greedy and *? is greedy. The opposite of Perl.

If you have ever written a Perl regex that catastrophically backtracked, RE2 is what you write next. The cost is the missing features above; the upside is the engine cannot be DoSed by a malicious input.

Go's standard regexp package wraps RE2 and adds a few Go-isms, none of which change the fundamentals.

Feature comparison tables

The remainder of this chapter is row-major: rows are features, columns are engines. This layout optimises for the “I know one engine, what does X look like in another?” use case. Cell values are concise and may simplify edge cases - verify against the engine's own reference for production code.

Quantifiers, grouping, alternation

Feature	Perl 5.42	PCRE2	Emacs	POSIX BRE	POSIX ERE	RE2 / Go
* zero-or-more	meta	meta	meta	meta	meta	meta
+ one-or-more	meta	meta	meta	literal (use \+)	meta	meta
? zero-or-one	meta	meta	meta	literal (use \?)	meta	meta
{m,n} interval	meta	meta	\{m,n\}	\{m,n\}	meta	meta
alternation	meta	meta	meta	\ (GNU)	meta	meta
(...) grouping	meta	meta	\(...\)	\(...\)	meta	meta
(?:...) non-capturing	yes	yes	no	no	no	yes
Possessive *+, ++, ?+	yes	yes	no	no	no	no
Non-greedy *?, +?	yes	yes	yes	no	no	yes
Atomic group (?>...)	yes	yes	no	no	no	no

The single highest-value row for a Perl reader meeting BRE: + and ? are *literal*. A regex like `colou?r` reads as “matches `colou?r` exactly” under BRE. The rule is consistent - BRE's metacharacter set is small; ERE expanded it; Perl expanded it further.

Character shorthand classes

What does `\d` match? It depends. The columns below show what character set each engine treats as digits, word characters, and whitespace under default settings.

Short-hand	Perl 5.42 (default)	Perl 5.42 + /u	PCRE2 (default)	Emacs	POSIX BRE	POSIX ERE	RE2 / Go (default)
\d	ASCII (or Unicode under /u)	Unicode	ASCII	NO	NO	NO	ASCII; Unicode under (?u)
\w	ASCII or Unicode	Unicode	ASCII	yes (syntax-table)	NO	NO	ASCII; Unicode under (?u)
\s	ASCII or Unicode	Unicode	ASCII	yes	NO	NO	ASCII; Unicode under (?u)
\b word boundary	yes	yes	yes	yes (\</\> traditional)	NO	NO	yes
\h horizontal WS	yes	yes	yes	NO	NO	NO	NO
\v vertical WS	yes	yes	yes	NO	NO	NO	NO

Two surprises here for a Perl reader:

1. POSIX BRE and ERE both lack \d, \w, \s. The portable forms are `[0-9]`, `[[:alnum:]]`, `[[:space:]]`. Tools that accept \d (GNU grep, awk in some implementations) do so as a non-portable extension.
2. Emacs has \w and \s but no \d. The \s character is followed by a syntax-class character - \s - for whitespace, \sw for word - which is unique to Emacs.

POSIX bracket classes like `[[:digit:]]` and `[[:alpha:]]` work in every engine in the table; they are the portable spelling.

Anchors and newline handling

Feature	Perl 5.42	PCRE2	Emacs	POSIX BRE / ERE	RE2 / Go
<code>^</code> start-of-string	yes	yes	start-of-buffer	yes	yes
<code>^</code> after <code>\n</code> under multi-line	/m	(?m)	always (line-oriented)	not specified	(?m)
<code>\$</code> end-of-string	yes	yes	end-of-buffer	yes	yes
<code>\$</code> before final <code>\n</code>	yes	yes	no	varies	yes
<code>.</code> matches <code>\n</code>	/s	(?s)	no	no	(?s)
<code>\A</code> / <code>\z</code>	yes	yes	<code>\`</code> / <code>\'</code>	no	yes (<code>\A</code> , <code>\z</code>)
<code>\Z</code> (before optional final <code>\n</code>)	yes	yes	no	no	no (use <code>\z</code>)
<code>\G</code> (last-match position)	yes	yes	no	no	no

Emacs uses `\`` and `\'` for the buffer anchors, where most modern engines use `\A` and `\z`. The Emacs spelling is older; it predates POSIX. Mixing up `^` and `\A` matters in line-oriented Emacs code more than in Perl, because Emacs matches are often invoked in contexts that scan an entire buffer.

`\G` is Perl-specific (and PCRE2). RE2 / Go uses a different API on the *match object* (a `Loc` method that returns the next position) rather than embedding the position in the pattern.

Lookaround and atomic constructs

Feature	Perl 5.42	PCRE2	Emacs	POSIX	RE2 / Go
Lookahead <code>(?=...)</code> / <code>(?!...)</code>	yes	yes	NO	NO	NO
Fixed-width lookbehind <code>(?<=...)</code>	yes	yes	NO	NO	NO
Variable-length lookbehind	yes (exp.)	yes	NO	NO	NO
<code>\K</code> keep-left	yes	yes	NO	NO	NO
Atomic group <code>(?>...)</code>	yes	yes	NO	NO	NO
Possessive quantifiers	yes	yes	NO	NO	NO

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

Captures and backreferences

Feature	Perl 5.42	PCRE2	Emac	POSIX BRE	POSIX ERE	RE2 / Go
Numbered captures	yes	yes	yes	yes	strict no, GNU yes	yes
Backreferences in pattern (<code>\1–</code>)	yes (1–99)	yes	yes	yes (<code>\1–\9</code>)	strict no, GNU yes	NO
Named captures (<code>? <name>...</code>)	yes	yes	NO	NO	NO	yes ((<code>? P<name>...</code>))
Branch reset (<code>? ...</code>)	yes	yes	NO	NO	NO	NO
Recursive subpatterns (<code>?R</code>)	yes	yes	NO	NO	NO	NO
Conditional patterns (<code>?(c)y n</code>)	yes	yes	NO	NO	NO	NO
(<code>{...}</code>) embedded code	yes	call-outs	NO	NO	NO	NO
@ <code>{^CAPTURE}</code> array	yes	NO	NO	NO	NO	NO (different API)

The single most important takeaway: **RE2 / Go regexp does not support backreferences**. Patterns like `/(.)\1/` cannot be compiled. This is not an engine bug; it is the price RE2 pays for guaranteed linear time. Patterns containing backreferences are not regular languages and a true DFA cannot match them in linear time.

If you are porting Perl regexps to a Go service, the audit question is “does this pattern use backrefs?” - if so, the conversion is not mechanical.

Engine-architecture summary

Property	PCRE2 / Perl 5.42	Emacs	POSIX BRE / ERE (typical impl)	RE2 / Go
Algorithm	Traditional NFA + JIT (PCRE2)	Traditional NFA	DFA (GNU) / NFA (others)	Hybrid NFA → lazy DFA
Worst-case time	Exponential	Exponential	Linear (DFA)	Linear (guaranteed)
Backtracking model	yes	yes	DFA: no	no
Catastrophic backtracking risk	yes	yes	no (DFA)	no
Backreferences supported	yes	yes	varies	NO

POSIX BRE/ERE is implementation-dependent: GNU `grep` and `awk` use a hybrid DFA-with-NFA-fallback that is fast and not subject to catastrophic backtracking; older traditional implementations (historical `awk`, traditional `grep` on some BSDs) use a backtracking NFA. The portable assumption is “linear time, but verify if your input may be

hostile”.

Selecting an engine when you have the choice

If you are writing the program (not reading code that already chose), the engine you pick is mostly determined by the language you are programming in. The exceptions are when the same library ships multiple engines, and when an external regex tool is your choice:

- **A Perl program:** Perl’s built-in. No reason to reach outside.
- **A C program embedding regex:** PCRE2. The standard library’s `regex.h` is POSIX BRE/ERE; PCRE2 is what you want for Perl-like behaviour.
- **A Go service handling untrusted input:** `regexp` (RE2). The linear-time guarantee is the headline feature; the missing backreferences and lookahead are usually acceptable.
- **A Rust program:** the `regex` crate. RE2-lineage; same linear-time guarantee.
- **Shell scripting:** prefer `grep -E` (ERE) and `sed -E` (ERE) over the BRE defaults for Perl-readability. Use `grep -P` (PCRE2) only when ERE genuinely is not enough.
- **An Emacs Lisp function:** Emacs’s. Otherwise leave Emacs.

See also

- The [character classes](#) chapter - the shorthand-by-engine row appears in its L3 sidebar.
- The [anchors and assertions](#) chapter - the lookahead and anchor rows appear in its L3 sidebar.
- The [groups and captures](#) chapter - the captures-and-backreferences row appears in its L3 sidebar.
- The [performance](#) chapter - Traditional NFA model and catastrophic backtracking, the architectural reason RE2 exists.
- **Basics** - `m//`, `s///`, the binding operators `=~` and `!~`, what counts as a metacharacter, how to escape, the four-phase parsing model.
- **Character classes** - bracketed classes `[...]`, negated classes, shorthand `\d \w \s`, POSIX classes, the extended bracketed form `(? [...])`, the IP-address worked example.
- **Anchors and assertions** - `^`, `$`, `\b`, `\A`, `\Z`, `\G`, `\K`, lookahead, lookbehind, the `\b{...}` Unicode boundaries, juxtaposition-as-AND.
- **Quantifiers** - `*`, `+`, `?`, `{n,m}`, greedy vs. non-greedy vs. possessive, the principles of the match.
- **Groups and captures** - `(...)`, `(?:...)`, named captures, backreferences, atomic groups, recursive subpatterns, conditional patterns, branch reset.
- **Alternation** - `|`, precedence, common-prefix factoring, empty alternatives, branch reset.

- **Modifiers** - /i, /m, /s, /x, /xx, /g, /c, /r, /e, /n, /p, /o, /a, /aa, /u, /l, /d, in-line forms (?i...), scoped forms (?i:...), the four-phase parsing model, use re 'strict'.
- **Substitution** - s/// in depth: the replacement string, /e, /ee, /r, \K in substitution, zero-length match termination.
- **Unicode** - \p{...}, \P{...}, \X, scripts, the charset modifiers, case folding, script runs, encoding vs. character.
- **Performance** - backtracking, catastrophic patterns, unrolling the loop, atomic groups, possessive quantifiers, recursive patterns, embedded code, special backtracking control verbs, internal optimisations the engine applies on your behalf.
- **Cross-engine** - comparison against PCRE2, Emacs, POSIX BRE, POSIX ERE, and RE2 / Go.

4.3.3 A first round-trip

The shortest useful regexp program - find three-letter words repeated back to back, separated by a single space:

```
my $text = "I said the the other day";
if ($text =~ /\b(\w{3})\s\1\b/) {
    print "Repeated: $1\n";      # Repeated: the
}
```

- \b is a word boundary - the pattern only fires at word edges.
- (\w{3}) captures exactly three word characters into \$1.
- \s is one whitespace character between the two copies.
- \1 is the backreference: match the same three characters again.

Every more complicated pattern in the guide is layered on that idea - anchor where you need to, describe what you want, capture what you want back.

4.3.4 Conventions in the examples

- Example output appears as an inline # ... comment beside the expression that produces it.
- Examples assume default modifiers unless they show /i, /x, etc.
- Where a pattern is shown on its own (no =~), treat it as a fragment the surrounding text will combine with a string.
- When a Unicode character is needed, examples use \x{263a} or \N{GREEK SMALL LETTER SIGMA} so the rendered text matches the source.

4.3.5 Related reference pages

The guide is a teaching reference; these are the operator references the guide refers to throughout.

- [m](#) - the match operator.

- `s` - substitution.
- `qr` - compile a pattern for reuse.
- `split` - split a string on a regexp.
- `pos` - read or set the /g match position.
- `quotemeta` - escape metacharacters in a string.
- `tr` - character-by-character translation. Often confused with regexp substitution but unrelated in semantics.

4.4 Performance

How to make Perl run fast on pperl, and - just as important - how to tell whether it already does. This guide is the actionable layer: where the classic Perl performance advice still holds, where pperl's JIT and auto-parallelizer have quietly made it irrelevant, and where pperl rewards code shaped in ways upstream Perl never cared about.

It is deliberately not a tour of the machinery. The JIT, the parallel scheduler, the native-module dispatch and the FFI each have their own chapter in the [architecture guide](#); this guide tells you what to *write* so they fire, and points at those chapters for the *why*.

4.4.1 Who this is for

A working Perl programmer with a program that is too slow, or one they want to keep fast. You know how to write the code; you want to know which habits pay off here, which are now dead weight, and how to measure the difference without guessing.

You do not need to understand Cranelift, Rayon, or the interpreter's inner loop. Where a behaviour depends on them, this guide states the behaviour and links the chapter that explains it.

4.4.2 The one rule that survives everything

Measure first. Every other rule in this guide is conditional on a measurement you have not yet taken. Perl performance intuition, honed on the upstream interpreter, mispredicts on pperl in both directions: code you expect to be slow is sometimes compiled to native machine code and runs faster than C; code you expect the JIT to rescue sometimes falls back to the interpreter because one string operation crept into a numeric loop.

The profiler does not lie and your intuition, ported from another runtime, does. Start at [Measuring](#).

4.4.3 How this guide is organised

Measuring

You cannot tune what you have not measured, and on pperl the usual measuring tools split into three groups: the ones that work unchanged, the ones that work because they are plain Perl running on a faster interpreter, and the one everybody reaches for that does not work at all. This chapter sorts them out and shows how to read what the working ones tell you.

The tool that does not work: Devel::NYTProf

Devel::NYTProf is the standard line-and-subroutine profiler on upstream Perl. It is an XS module - it hooks the interpreter through a C extension. pperl ships no XS layer, so Devel::NYTProf does not load. Neither does any other XS-based profiler (Devel::DProf, Devel::Profiler's XS paths).

This is not a gap to be worked around with a pure-Perl reimplementaion of NYTProf's output format. pperl has its own profiler, built into the runtime, that sees what the interpreter actually executes - op counts and the time spent in each. Use it.

The pperl profiler

ppperl's profiler is compiled into a dedicated build and switched on at run time by an environment variable. It is off by default and costs nothing in a normal build.

Build once with the profile feature, then run with PPERL_PROFILE set:

```
cargo build --release --features=profile
PPERL_PROFILE=1 ./target/release/ppperl script.pl
```

When the program exits - or on SIGINT / SIGTERM, so a long-running or hung program can still be profiled by interrupting it

- the report prints to STDERR. It has two sections.

Subroutine timings, the top entries by total time:

```
-- Subroutine Timings (top 20 by total time) --
```

Sub	Calls	Total ms
→ Avg μs		

→ -----		
main::parse_record	10000	412.318
→ 41.23		
main::normalise	120000	88.004
→ 0.73		

Op timings, the top entries by total time, naming the interpreter operation rather than a line of your source:

```
-- Op Timings (top 30 by total time) --
```

Op	Count	Total ms	Avg ns

helem	12400000	210.114	16
padsv	31000000	180.700	5

Read the subroutine section first: it tells you *where* in your code the time goes. Read the op section when the subroutine section has pointed you at a routine and you want to know *what kind* of work dominates it - a routine heavy in helem is doing hash lookups; one heavy in padsv is touching lexicals; one whose time is in subcall is paying call overhead worth inlining.

The profiler counts what the interpreter executes. Code that the JIT compiled to native machine code does not run through the interpreter's op dispatch, so it does not show

up op-by-op. A loop that has vanished from the op timings is a loop the JIT took over - which is the answer you wanted. To profile interpreter behaviour in isolation, disable the JIT with `--no-jit` (see below).

Whole-program timing: the benchmark runner

When the question is “how fast is pperl on this, versus system perl, with and without JIT and parallelism”, the project benchmark runner answers it directly. `bench/run-perlbench` always runs `/usr/bin/perl` as the baseline (normalised to 100) and compares pperl in up to four configurations.

```
./bench/run-perlbench          # no JIT, no parallel
./bench/run-perlbench +J      # add the JIT-enabled run
./bench/run-perlbench +P      # add the parallel run
./bench/run-perlbench +J+P    # all four combinations
./bench/run-perlbench -t arith # only benchmarks matching
→ "arith"
```

Every runner shares one iteration count, derived from the slowest runner’s empty-loop rate, so the reported ratios compare like with like. The `+J` / `+P` columns are exactly the comparison you want when deciding whether a piece of code benefits from the JIT or the parallelizer: if `+J` is no faster than the plain run, the loop is not being compiled, and *Writing fast pperl* explains why and what to change.

Plain-Perl tools that work: Benchmark and Time::HiRes

The classic in-program measuring tools are plain Perl. They run on pperl through the normal module path - `Benchmark` is reachable on the standard include path and executes as Perl on the pperl interpreter; `Time::HiRes` is provided natively. Both behave exactly as a Perl programmer expects.

Benchmark for A/B comparison of two ways to write the same routine:

```
use Benchmark qw(cmpthese);

my @data = (1 .. 100_000);

cmpthese(-3, {
    grep_count => sub { my $n = grep { $_ % 2 == 0 } @data },
    loop_count => sub { my $n = 0; $_ % 2 or $n++ for @data },
});
```

`Time::HiRes` for ad-hoc wall-clock timing of a single block:

```
use Time::HiRes qw(time);

my $t0 = time;
do_the_work();
printf "took %.3f s\n", time - $t0;
```

A caution that matters more on pperl than on upstream: a `Benchmark` micro-comparison that wraps the code in a `sub { ... }` measures the code *as a subroutine body*. Subrou-

tine bodies are not JIT-compiled - the JIT targets loops, not call frames (see [JIT compilation](#)). So a tight numeric loop that would be compiled at file scope may run interpreted inside a Benchmark callback, and the comparison understates the loop's real speed. When the thing you are timing is a loop that should JIT, prefer `bench/run-perlbench` or a `Time::HiRes` span around the loop in situ.

Turning the optimisers off to isolate a cause

Two flags let you attribute a speedup - or a slowdown - to the right layer:

```
pperl --no-jit script.pl           # interpreter only, no native
↳ compilation
ppperl --no-parallel script.pl     # single-threaded, no Rayon
↳ dispatch
```

Run a program three ways - default, `--no-jit`, `--no-parallel` - and the differences tell you which optimiser is carrying the work. If `--no-jit` barely changes the time, the JIT was not engaging on this code, and the loop shape is the thing to look at. If `--no-parallel` barely changes it, the parallelizer declined the loop - usually because of a side effect it could not rule out (see [Parallel execution](#)).

A measuring discipline

1. Run the program once under the profiler build. Read the subroutine timings. The top two or three lines are your budget.
2. If a hot routine is a loop you expected to be compiled, confirm with `run-perlbench +J` or by timing it `--no-jit` versus default. A loop that does not speed up under the JIT is shaped wrong, not slow.
3. Only now change code. Re-measure the same way. Keep the change if the number moved and the program still produces the same output.

Performance claims are cheap; the profiler report and the benchmark ratio are not. Trust the measurement.

See also

- [Writing fast pperl](#) - once you have found the hot loop, how to shape it so the JIT and parallelizer take it.
- [Idioms](#) - which classic optimisations are still worth applying before you reach for the profiler.
- [JIT compilation](#) - what the JIT compiles and why interpreted op timings disappear for compiled loops.
- [Parallel execution](#) - what the `+P` column and `--no-parallel` flag are exercising.
- [Regex performance](#) - if the profiler points at a regex, the cause and cure live there, not here.

Idioms

Perl carries decades of performance folklore - habits that paid off on the upstream interpreter and got passed down as rules. Some of that advice is timeless. Some of it is now actively pointless on pperl, because the JIT or the compile-time folder already does the work. And a few new habits matter here that no upstream Perl programmer ever needed.

This chapter triages the classic idioms into three buckets:

- **Still valid** - the advice holds; the cost it avoids is real on pperl too.
- **Made moot** - pperl's compiler or JIT already eliminates the cost, so the contortion buys nothing and only hurts readability.
- **New** - relevant *because* of how pperl compiles, with no upstream precedent.

A blanket caveat runs through the whole “made moot” bucket: the JIT only takes a loop when the loop is *shaped* for it. A loop heavy with subroutine calls, regexp matches, or string building is not compiled and runs on the interpreter - where the classic advice re-applies in full. So “the JIT handles it” is true *for loops the JIT handles*. [Writing fast pperl](#) is the companion chapter on making sure your loop is one of them.

Still valid

These survive the move to pperl unchanged. The cost they avoid is paid on the interpreter, and most hot code touches the interpreter somewhere.

- **Hoist invariant work out of loops.** Computing the same value every iteration is wasted work whether the loop is interpreted or compiled. The JIT does not hoist arbitrary expressions for you; move the invariant out yourself.
- **Avoid needless copies of large structures.** Passing a big array by value, or slicing it just to count, copies. Pass a reference, use an aliased loop variable, or work in place. pperl's value model has the same copy costs Perl always had.
- **Precompile regexps used in a loop with `qr`.** The regex engine and its compile cache work as on upstream; a pattern built fresh each iteration pays to recompile. This is regex territory - see [regex performance](#).
- **Choose the right data structure.** A hash lookup beats a linear scan at the same sizes it always did. Algorithmic complexity is unaffected by any runtime; an $O(n^2)$ loop is slow even fully JIT'd.
- **Read files in the right-sized chunks.** Line-by-line versus slurp-and-split is an I/O question, and I/O is not JIT-compiled. The upstream trade-offs carry over directly.
- **Prefer built-in list operations to hand-rolled loops where the built-in is native.** `sum`, `min`, `max`, `first` from `List::Util` run at built-in speed with no XS layer (see [native modules](#)). A hand-written accumulator loop *may* JIT to the same speed, but the native built-in gets you there without depending on the loop shape.

Made moot

These were real wins on upstream Perl. On pperl the compiler or JIT already removes the cost, so applying the idiom by hand buys nothing and usually costs clarity.

Hand-inlining hot arithmetic

The old habit of unrolling a tight arithmetic loop, or manually inlining a small arithmetic helper to dodge call overhead, targets a cost the JIT removes. A numeric `for` or `while` loop over integer or float accumulators is compiled to native machine code, with the loop variables held in CPU registers - see [JIT compilation](#). Hand-unrolling such a loop makes it longer and harder to read without beating the compiler.

```
# Don't hand-unroll this. The JIT compiles the natural form to
# native code with $i and $sum in registers.
my $sum = 0;
for my $i (1 .. 1_000_000) {
    $sum += $i * $i;
}
```

The contortion only pays off if the loop does *not* JIT - and if it doesn't, the fix is to make it JIT (remove the call or string op that disqualified it), not to unroll it by hand.

DEBUG-constant logging guards

The idiom of guarding expensive logging behind a compile-time false constant -

```
use constant DEBUG => 0;
# ...
warn "state: @stuff\n" if DEBUG;
```

- costs exactly nothing on pperl, which is the whole point of the idiom, and pperl delivers it completely. When `DEBUG` is a false constant, the compiler folds the `if` `DEBUG` guard at compile time: the disabled statement collapses to a no-op, with no condition test and no branch left in the compiled program. The interpolation `"@stuff"` is never built. You can leave guarded logging in hot paths with a clear conscience; a false `DEBUG` constant erases it.

This is the idiom working *as designed* - keep writing it. What is moot is any further cleverness on top: you do not need to comment out the logging, move it behind a sub, or strip it for production. The constant does it.

Folding constant expressions yourself

Precomputing a constant arithmetic result, or a constant call to a pure built-in, and storing it in a variable to “avoid recomputing it” buys nothing - the compiler folds constant expressions and constant-argument pure built-ins (`sin`, `cos`, `sqrt`, `chr`, `ord`, and the like) at compile time, as long as the call is in a valid domain. `my $two_pi = 2 * 3.14159265358979`; and `my $root = sqrt(2)`; are computed once, at compile time, whether or not you “help” by hoisting them.

The one nuance: folding is skipped for arguments outside a function's domain (`sqrt` of

a negative, log of zero), so those fall through to run time and croak there exactly as they should. You never need to fold a constant by hand to gain speed.

Stripping dead constant-condition loops

A `while (0) { ... }` or a block guarded by a constant-false condition is eliminated at compile time - the body is removed, not merely skipped at run time. (A C-style `for (; 0; ...)` is handled slightly differently: the body is never entered, though the loop scaffolding op may remain.) Conditionally-compiled-out code behind a false constant carries no run-time weight; you do not need to physically delete it to make the program faster.

New

These have no upstream analogue. They matter because of how pperl compiles and parallelizes, and getting them wrong silently forfeits the largest speedups pperl offers.

Keep numeric hot loops numeric

The single most valuable new habit. The JIT compiles a numeric loop and, when parallelization is enabled (the default) and the loop is a clean reduction over a known range, dispatches it across threads - the difference between a 76× and a 431× speedup on the Mandelbrot benchmark (see [parallel execution](#)).

A string operation inside that loop forfeits the parallel half. When the JIT sees a string variable in the loop - a `.` = concat target, a string accumulator - it demotes the loop to a *sequential* compiled loop: still native code, but no longer spread across threads.

```
# Numeric reduction - JIT-compiled and eligible for parallel
→dispatch.
my $total = 0;
for my $i (0 .. $n) {
    $total += compute($i);      # (only if compute() itself stays
→numeric/inlinable)
}

# A string built in the same loop demotes it to sequential JIT -
# the parallel dispatch is gated off the moment a string var
→appears.
my $log = "";
for my $i (0 .. $n) {
    $total += $i * $i;
    $log .= "$i ";              # forfeits parallel dispatch for this
→loop
}
```

If you need both the numeric reduction and the string accumulation, split them into two loops: the numeric one parallelizes, the string one runs as a sequential compiled loop. Mixing them costs you the parallel speedup on the numeric work for no gain on the string work.

Shape reductions as accumulate-never-reset

The parallelizer recognises a reduction variable by seeing it accumulated and never reset inside the loop body (the analysis subtracts reset patterns from accumulation patterns; a variable that is both is not treated as a reduction). Writing your accumulator in the plain `$sum += ...` form, declared *before* the loop and never re-initialised inside it, is what lets the loop be claimed for parallel execution.

```
my $sum = 0;                                # declared outside - a reduction
for my $x (@data) {
    $sum += weight($x);                     # accumulate, never reset →
    ↪parallelizable
}
```

Re-initialising the accumulator inside the loop, or interleaving a reset, defeats the recognition and the loop runs sequentially. This is covered in full under *reduction detection*.

Let the loop JIT - don't hide it in a sub

The JIT compiles *loops*, not subroutine bodies. A hot numeric loop written at file scope or inside a larger routine compiles fine; the same arithmetic refactored so each iteration is a subroutine call does not - the call frame is interpreter territory, and the per-call overhead is exactly what the JIT cannot remove. When a loop is hot and numeric, keep its body inline rather than factoring each iteration into a sub for tidiness. Factor for clarity elsewhere; keep the inner loop flat.

See also

- *Writing fast pperl* - the positive program: how to shape a loop so the JIT and parallelizer take it.
- *Measuring* - confirm an idiom actually moved the number before and after applying it.
- *JIT compilation* - what compiles, why hand-inlining arithmetic is redundant.
- *Parallel execution* - the reduction analysis and the side-effect gate behind the “new” idioms.
- *Sorting* - the sort-specific idioms (Schwartzian, Guttman-Rosler) get their own chapter.

Sorting

`sort` is where a surprising amount of a program's time hides, because the cost is not the sort - it is the comparator, called $O(n \log n)$ times. A comparator that does real work per call multiplies that work by the log of the list size. This chapter is about recognising when the comparator is the bottleneck and the two transforms that fix it.

The reference page for `sort` defines the operator, the comparator contract, and the `$a/$b` mechanics. This chapter assumes you know those and focuses on speed.

The comparator is the cost

sort itself is a stable mergesort - fast, and not something you can improve from Perl. What you control is how much work happens *inside* each comparison. Two facts set the budget:

- The comparator runs $O(n \log n)$ times. For a 100,000-element list that is well over a million calls.
- On pperl, the comparator block is a callback, and callbacks run on the interpreter. The JIT compiles loops, not sort comparator bodies (see *JIT compilation*), so you cannot rely on the JIT to rescue an expensive comparator the way it rescues a numeric loop. The comparator's cost is paid in full, every call.

That second fact is the pperl-specific reason the transforms below matter as much here as on upstream Perl - arguably more, because the JIT's speedups elsewhere make an un-transformed sort stand out more sharply in a profile.

The diagnosis is always the same: if *Measuring* shows time concentrated in a comparator, the key is being recomputed on every comparison. Compute it once instead.

The Schwartzian transform

When the sort key is expensive to derive from each element - a substring extraction, a field lookup, a length, a computation - the naive sort recomputes it on every comparison:

```
# Slow: the key (-M $_) is computed twice per comparison, O(n log n)
times.
my @sorted = sort { -M $a <=> -M $b } @files;
```

The Schwartzian transform computes each key exactly once, sorts on the precomputed key, then discards it - a map, a sort, a map:

```
my @sorted =
  map { $_->[1] }           # 3. strip the key, keep
  ↪ the element
  sort { $a->[0] <=> $b->[0] } # 2. sort on the
  ↪ precomputed key
  map { [ -M $_, $_ ] }     # 1. compute the key once
  ↪ per element
  @files;
```

Read it bottom to top: the lower map builds a [key, element] pair for each element (n key computations), the sort compares only the cheap precomputed $\$_->[0]$ (no key computation in the comparator), and the upper map throws the key away. The expensive key is computed n times instead of $O(n \log n)$ times.

Use it whenever the key costs more than a numeric or string compare to derive. For a 100,000-element list sorted by a key that takes a microsecond to compute, the naive form spends seconds in key computation alone; the transform spends a tenth of a second.

The pieces are `map` and `sort` - the `sort` reference page shows the same transform as a worked example.

The Guttman-Rosler transform

The Schwartzian transform sorts arrayrefs, which costs an arrayref allocation and a dereference per comparison. When the key can be encoded as a fixed-width *string* prefix, the Guttman-Rosler transform (GRT) goes further: it packs the key and the element into a single string, sorts those strings with the default string sort (no comparator block at all), then strips the prefix.

```
my @sorted =
    map { substr($_, 8) }                # 3. strip the
    ↪8-char key prefix                    # 2. plain
    sort                                # 1. fixed-width
    ↪string sort, no comparator          key + payload
    map { pack('N', $_->{priority}) . $_->{name} }
    ↪@records;
```

The win has two parts. First, the key is precomputed once, as in the Schwartzian. Second - and this is the GRT's distinct advantage - the sort has *no comparator block*, so there is no per-comparison callback at all. Default string sort compares the packed strings directly, which avoids the callback cost entirely. Removing the comparator callback matters more on pperl precisely because that callback would run on the interpreter.

The cost of the GRT is that the key must be encodable so that a bitwise string comparison produces the order you want:

- Integers must be packed big-endian (`pack 'N' / pack 'Q>'`) so that byte order matches numeric order, and offset to be non-negative (string comparison has no sign bit).
- Floats and signed numbers need bias encoding to sort correctly as bytes - fiddly enough that the Schwartzian is usually the better choice unless the sort is genuinely hot.
- Every key must be the *same width*, or a short key's payload bleeds into the comparison.

Reach for the GRT when the sort is hot, the key is an unsigned integer or a string, and the Schwartzian still shows up in the profile. For anything else, the Schwartzian is simpler and almost always fast enough.

Cheaper comparators without a transform

Not every slow sort needs a transform. Sometimes the comparator is just written more expensively than it needs to be.

- **Compare numbers with `<=>`, strings with `cmp`.** Using `cmp` on numbers stringifies both sides every comparison; using `<=>` on strings is a numification trap. Match the operator to the data.
- **Order tiebreakers cheapest-first.** A multi-key comparator with `||` stops at the first non-zero result. Put the discriminating, cheap key first so most comparisons never reach the expensive one:

```

sort { $a->{rank} <=> $b->{rank}          # cheap, decides most_
↪pairs
    || $a->{name} cmp $b->{name} }         # only when ranks tie
    @rows;

```

- **Sort once, reverse separately.** To sort descending, sorting ascending and calling `reverse` is often clearer and no slower than inverting the comparator, and it keeps the comparator in its cheapest form.

Decision order

1. Profile. If *Measuring* does not put the time in the sort, leave it alone.
2. If the comparator recomputes a key, apply the Schwartzian transform. This fixes the great majority of slow sorts.
3. If the sort is still hot and the key is an unsigned integer or a string, consider the Guttman-Rosler transform to drop the comparator callback entirely.
4. Re-measure after each change.

See also

- `sort` - the operator reference: comparator contract, `$a/$b`, stability.
- `map` - the transform half of both the Schwartzian and Guttman-Rosler forms.
- `reverse` - descending sorts without inverting the comparator.
- *Measuring* - confirm the comparator is the cost before transforming it.
- *JIT compilation* - why a sort comparator block is not JIT-compiled, so its cost is paid in full.

Writing fast pperl

The other chapters in this guide are about removing cost. This one is about earning the speedups pperl offers that no other Perl has: the JIT that compiles hot loops to native code, and the parallelizer that spreads a qualifying loop across CPU cores. Both are automatic - you do not call them, annotate for them, or configure them. But they only fire on code *shaped* the right way, and a single misplaced operation can shut either one off silently.

This chapter is the positive program: write the loop like this, and the compiler takes it. The *JIT* and *parallel execution* chapters explain the machinery; this one tells you what to type.

The shape the JIT wants

The JIT targets **loops** - `for` and `while` - whose bodies are arithmetic and comparison over numeric variables. When it recognises that shape, it compiles the body to native machine code and holds the loop variables in CPU registers instead of on the interpreter stack. That is where the order-of-magnitude speedups come from.

A loop the JIT compiles cleanly:

```
my $sum = 0;
for my $i (1 .. 1_000_000) {
    $sum += $i * $i;
}
```

Integer and floating-point arithmetic (+, -, *, /), comparisons (<, >, <=, >=, ==), and control flow (if, last) inside the loop all compile. Nested loops compile too, several levels deep, each level connected to the next. The natural, idiomatic form is the fast form - you do not unroll, hoist into temporaries, or otherwise distort it.

What pushes a loop back onto the interpreter

The JIT declines a loop whose body does work it cannot express in native code. When it declines, the loop runs on the interpreter - at interpreter speed, which is still fast for non-numeric work, but without the native-code multiplier. The disqualifiers:

- **Subroutine calls.** The JIT compiles loop bodies, not call frames. A loop that calls a sub each iteration runs interpreted - the call overhead is the very thing the JIT cannot remove. If the per-iteration work is small and hot, inline it into the loop body rather than factoring it into a sub.
- **Regex operations.** The regex engine has its own optimiser and its own path; a match or substitution in the loop body is not JIT-compiled. See [regex performance](#) for tuning that work on its own terms.
- **I/O.** A print, a file read, a system call - I/O-bound work does not benefit from native compilation and is not compiled.
- **Complex data-structure access.** Hash and array operations with dynamic keys fall outside the JIT's numeric model.

None of these is an error; the loop simply runs the interpreted way. But if a loop you expected to be fast is not, run it `--no-jit` versus default ([Measuring](#)): if the times match, the JIT was already declining it, and one of the above is why.

Keep strings out of numeric loops

The JIT *does* handle one string operation - `.` = concat-assign, and clearing a string with `$x = ""` - through calls back into the runtime. So a string-building loop still compiles. But this is the one case where the shape of the loop changes *which optimisers* apply, and it is the highest-value rule in this chapter.

When the JIT sees a string variable in a loop, the loop compiles as a **sequential** native loop - but parallel dispatch is gated off. The parallelizer needs the loop body free of the shared-mutable-state that string building requires, so the moment a string variable appears, the loop is no longer spread across threads. It still runs as compiled native code; it just runs on one core.

The practical consequence: a numeric reduction that *would* parallelize loses that parallelism if you build a string in the same loop.

```
# Parallelizable: pure numeric reduction.
my $total = 0;
```

(continues on next page)

(continued from previous page)

```

for my $i (0 .. $n) {
    $total += $i * $i;
}

# Same arithmetic, but the string build demotes it to a
# single-core compiled loop - the parallel speedup is forfeited.
my $report = "";
my $total = 0;
for my $i (0 .. $n) {
    $total += $i * $i;
    $report .= "$i ";
}

```

If you need both, split the work into two loops. The numeric loop parallelizes; the string loop runs as a sequential compiled loop. Mixing them costs the parallel speedup on the numeric half and gains nothing on the string half.

The shape the parallelizer wants

When parallelization is enabled - it is on by default; `--no-parallel` turns it off - a JIT-compiled loop is dispatched across threads if the analysis can prove it safe. Two conditions carry the most weight:

A reduction shaped as accumulate-never-reset. The parallelizer recognises a reduction variable by seeing it accumulated and never reset inside the loop. Declare the accumulator before the loop, use the plain `+=` form, and never re-initialise it inside the body:

```

my $sum = 0;                                # declared outside the loop
for my $x (@data) {
    $sum += score($x);                       # accumulate only → recognised as a
    ↪reduction
}

```

Re-initialising `$sum` inside the loop, or interleaving a reset, defeats the recognition (the analysis subtracts reset patterns from accumulation patterns) and the loop runs sequentially. Full detail in [reduction detection](#).

A loop body free of observable side effects. The analyzer is deliberately conservative. Any of these disqualifies a loop from parallel dispatch:

- I/O (print, open, file reads)
- Writes to package/global variables
- Subroutine calls not proven pure
- Side-effecting regex operations (`s///`)

A missed parallelization is safe - the loop just runs sequentially. An *incorrect* one would be a bug, so the analysis errs toward declining. That conservatism is why keeping the loop body small, local, and free of globals is what gets it parallelized.

Built-ins that parallelize

`map` and `grep` with pure callbacks over a large-enough collection can run in parallel, with output order preserved:

```
my @out      = map { expensive($_) } @big;
my @filtered = grep { costly_test($_) } @big;
```

The same gates apply: the callback must be free of side effects and shared mutable state, and the collection must clear the parallelization threshold (tunable with `--parallel-threshold`). A pure transform over a large array is the easiest parallel win pperl offers - no loop to shape, just a clean callback.

When the work is not a loop

Not every program is a numeric loop, and the JIT and parallelizer have nothing to offer code that is call-bound, I/O-bound, or string-shaped. For that code the rest of this guide applies in full:

- Pick native built-ins over hand-rolled loops where they exist - `List::Util` runs at built-in speed.
- Apply the classic still-valid idioms from *Idioms*: hoist invariants, avoid copies, pre-compile regexps.
- Transform expensive sorts (*Sorting*).
- Call C through the *FFI* only when the call frequency is low - the per-call marshalling cost makes it the wrong tool inside a hot inner loop.

A checklist for a hot loop

1. Is the body arithmetic and comparison over numeric variables? If yes, the JIT compiles it. If not, the disqualifier is a call, a regex, I/O, or dynamic data access - address that first.
2. Is there a string operation in the loop? If so, and the loop is otherwise a numeric reduction, split the string work out so the numeric loop can parallelize.
3. Is the accumulator declared outside the loop and only accumulated, never reset? That is what makes it a recognised reduction.
4. Is the body free of I/O, global writes, and impure calls? That is what lets the loop parallelize.
5. Measure default versus `--no-jit` versus `--no-parallel` (*Measuring*) to confirm which optimiser is engaging.

See also

- *JIT compilation* - the compilation pipeline, the variable-type model, the caching.
- *Parallel execution* - the reduction analysis, the side-effect gate, the thread controls.
- *Measuring* - how to confirm the JIT and parallelizer are actually engaging on your code.

- *Idioms* - which classic optimisations still matter and which the compiler already handles.
- *Sorting* - the one common hot path the JIT does not reach.
- **Measuring** - the tools that actually work under pperl. The upstream XS profilers (Devel::NYTProf) do not load; the built-in `--features=profile` build and the `bench/run-perlbench` runner do. How to read their output and how to A/B two versions of a routine.
- **Idioms** - a triage of the classic Perl performance idioms. Each is tagged: still valid, made moot by the JIT or parallelizer, or newly relevant because of how pperl compiles. The idioms that were folklore on upstream Perl are not all folklore here.
- **Sorting** - sort in depth: the Schwartzian transform, the Guttman-Rosler transform, sort subroutine cost, and when the comparator is the bottleneck.
- **Writing fast pperl** - the positive program: how to structure a hot loop so the JIT compiles it, how to shape a reduction so the parallelizer claims it, and the one mistake - mixing string work into a numeric loop - that silently disables both.

4.4.4 What this guide does not cover

The mechanisms live elsewhere and this guide does not repeat them:

- *JIT compilation* - what gets compiled, the caching model, the variable-type analysis.
- *Parallel execution* - the reduction analysis, the side-effect gate, the CLI thread controls.
- *Regex performance* - backtracking, catastrophic patterns, the engine's optimiser. Regex speed is its own discipline; nothing in this guide duplicates it.
- *Native modules* - why `List::Util` and friends run at built-in speed with no XS layer.
- *Auto-FFI* - calling C without XS, and the per-call marshalling cost that makes it the wrong tool for a hot inner loop.

4.4.5 A note on numbers

Concrete speedups in this guide are reproduced from the architecture and concurrency chapters, which carry the measured figures and the benchmark that produced them. Where this guide states a speedup without a number, the qualitative behaviour is established but the exact figure depends on your machine, your data, and your build - run `bench/run-perlbench` and trust your own measurement over any printed ratio.

4.5 Object-oriented programming

Perl has two object systems living side by side: the **modern class/field/method** syntax introduced in 5.38 (codename *Corinna*), and the **classical bless/@ISA** model that predates it by three decades. pperl supports both, and both appear in real code you will read and maintain.

The short version:

- For **new code**, use `class`. The syntax is first-class, the constructor is generated for you, fields are genuinely private, and single inheritance has a clean declarative form.
- For **legacy code**, understand `bless`, `@ISA`, and the `$self = shift` convention. You will meet them in every pre-5.38 codebase, every mainline CPAN distribution, and much of core Perl itself.
- **Moose**, **Moo**, **Class::Accessor**, **Object::Pad** are the CPAN ecosystems that papered over the classical model's rough edges while the core feature was still being designed. They are a historical chapter; new code reaches for core `class` instead.

This how-to tree covers both systems end-to-end. Read the chapters in order on a first pass, or jump to the one that matches the code in front of you.

4.5.1 Which chapter you want

- *Modern classes* - `class`, `field`, `method`, `ADJUST`, `:param`, `:reader`. The recommended way to write OO in pperl.
- *Classical OO* - `bless`, `@ISA`, `sub new`, accessors by hand, `SUPER::`. The model you will maintain in existing code.
- *Inheritance and method resolution* - how method lookup works, `isa`, `can`, `DOES`, `MRO`, and the pitfalls of multiple inheritance.
- *Roles and delegation* - composition patterns for behaviour shared across unrelated classes.
- *Migrating from classical to modern* - a mechanical recipe for converting a `bless`-based class to the `class` feature, and the places where the translation is not mechanical.

4.5.2 When to use OO at all

OO is a tool, not a style. Reach for it when:

- The problem has a small vocabulary of nouns (user, session, request, record) and a larger vocabulary of verbs acting on each.
- A piece of data carries invariants that should be enforced at one point in the code, not at every call site.
- You expect the implementation behind a set of operations to change while the operations themselves stay stable.
- You are modelling a category with variants and want polymorphism to pick the right behaviour.

Do not reach for OO when:

- The program is small, linear, and has no reusable nouns.
- A plain hash of hashes or a closure is enough.
- The only reason to write a class is “it felt more professional.” A twelve-line script does not need a class.

4.5.3 A note on scope

This tree is a tutorial, not a reference. The exhaustive syntax tables live in the per-keyword reference pages:

- `class`
- `field`
- `method`
- `bless`
- `ref`
- `isa`

Link into them whenever you need the full signature surface; this tree gives you the working-programmer overview.

Modern classes

The `class` feature is the recommended way to write object-oriented code in pperl. It replaces the hand-rolled constructor, the manual accessor generation, the hash-slot book-keeping, and the `use parent` dance with a single declarative syntax. You describe *what* the class is; the runtime arranges *how* it is implemented.

This chapter walks through the feature as a working programmer would meet it: first a minimum class, then fields and attributes, then methods, then construction-time hooks, then inheritance. Each section is short on purpose; when you want the full surface of a keyword, follow the cross-link into its reference page.

Enabling the feature

The feature is flagged experimental in Perl 5.42 and pperl mirrors that status. Every file that uses it opens with:

```
use v5.38;
use feature 'class';
no warnings 'experimental::class';
```

The `use v5.38` line already enables the `class` feature on its own, but being explicit makes the dependency visible to anyone reading the file.

A minimal class

```
use v5.38;
use feature 'class';
no warnings 'experimental::class';

class Point {
    field $x :param;
    field $y :param;
```

(continues on next page)

(continued from previous page)

```
method distance_from_origin {  
    return sqrt($x ** 2 + $y ** 2);  
}  
}  
  
my $p = Point->new(x => 3, y => 4);  
say $p->distance_from_origin; # 5
```

Three things to notice:

- There is no sub `new`. The runtime generates the constructor from the `field` declarations. A user-written sub `new` inside a class body is a compile-time error on purpose - the generated one knows how to wire up `:param`, run initialisers, and trigger ADJUST blocks in the right order.
- Fields are referenced by their *declared name* inside methods, not through `$self->{x}`. The instance storage is genuinely private; no code outside the class body can poke at it.
- The `method` keyword takes care of `$self`. You never write `my $self = shift`.

Fields

A `field` is per-instance storage. Every object of the class gets its own slot; every method and every ADJUST block sees the field as if it were a lexical.

```
class Counter {  
    field $value = 0;  
    field $step  = 1;  
  
    method tick { $value += $step }  
    method value { $value }  
}
```

Fields accept scalars, arrays, and hashes:

```
class Registry {  
    field $name;  
    field @items;  
    field %seen;  
}
```

Attributes on fields

Field attributes take the declarative work off your hands.

- **`:param`** - bind this field to a constructor argument. If the caller omits it, the field is either left unset or takes the declared default.
- **`:param(alt)`** - bind to a different constructor name.
- **`:reader`** - generate a read-only accessor with the field's name minus the sigil.

- **:reader(custom)** - generate a reader under a different name.
- **:writer** - generate a setter (set_FIELDNAME).

```
class User {
    field $name :param :reader;
    field $email :param :reader :writer;
    field $role :param :reader = 'guest';
}

my $u = User->new(name => 'Ada', email => 'ada@example.org');
say $u->name;                               # Ada
$u->set_email('ada@analytical.org');
say $u->role;                               # guest
```

Default values

Defaults use one of three operators, each with a different rule:

- **=** - apply when the caller *omitted* the parameter entirely.
- **//=** - also apply when the caller passed undef.
- **||=** - also apply when the caller passed any false value.

```
class Window {
    field $title :param          = 'untitled';
    field $width :param         //= 640;
    field $shown :param         ||= 1;
}
```

Pick = unless you have a concrete reason to allow the stronger forms. Silent promotion of undef or 0 to a default is a bug magnet.

Fields are not lexicals in the usual sense

Fields look like lexicals inside a method body, but they have tighter rules:

- You cannot **our** a field, you cannot **local** one, and you cannot take a reference to one and export it.
- Fields are private to the class. Subclasses do **not** inherit a parent's fields; if a subclass needs a parent's state, it asks for it through a method.
- A field's initialiser runs before \$self is bound. Inside field \$x = EXPR, you can use earlier fields by name and you can use `__CLASS__`, but you cannot use \$self. Anything that needs the fully constructed instance belongs in an ADJUST block.

Methods

A **method** is a subroutine that auto-binds \$self and reads the enclosing class's fields by name.

```
class File {
  field $path :param :reader;

  method slurp {
    open my $fh, '<', $path or die "open $path: $!";
    local $/;
    return <$fh>;
  }

  method rename ($new) {
    rename $path, $new or die "rename: $!";
    $path = $new;
  }
}
```

Methods use signatures automatically - use feature 'signatures' is implied inside a class body. `$self` is the method's implicit first argument and does **not** appear in the signature.

Private methods

A method declared with `my` is lexical to the class body and is invoked through the `->&` operator:

```
class Safe {
  field $secret :param;

  my method check ($n) { $n == $secret }

  method unlock ($n) {
    return $self->&check($n) ? 'open' : 'denied';
  }
}
```

`my` method is the closest pperl has to a hard-private method; no external caller can reach it because the name does not live in the package at all.

Construction hooks: ADJUST

An ADJUST block runs after field initialisation, with `$self` already bound. It is the place for:

- Post-construction validation.
- Derived fields that depend on other fields.
- Opening a handle or connecting to a resource whose lifetime matches the object.

```
class TempFile {
  field $prefix :param = 'tmp';
  field $path;
```

(continues on next page)

(continued from previous page)

```

field $fh;

ADJUST {
    $path = "/tmp/$prefix-$$-" . int(rand 1_000_000);
    open $fh, '>', $path or die "open $path: $!";
}

method write ($data) { print {$fh} $data }
method path { $path }
}

```

Multiple ADJUST blocks in one class run in declaration order. In an inheritance chain, the parent's ADJUST blocks run before the child's.

Inheritance

A class inherits from at most one parent using the `:isa` attribute:

```

class Shape {
    field $colour :param :reader = 'black';
    method area { 0 }
}

class Circle :isa(Shape) {
    field $radius :param :reader;
    method area { 3.14159265 * $radius ** 2 }
}

```

Only **methods** are inherited. Fields are per-class storage. A subclass that needs the parent's state asks through a reader:

```

class Labelled :isa(Shape) {
    field $label :param :reader;
    method describe { "$label (" . $self->colour . ') ' }
}

```

Multiple inheritance is not supported. `:isa(A, B)` is a syntax error, and a class cannot carry more than one `:isa` attribute. The *inheritance chapter* explains why, and covers the role-based alternative.

Calling up the chain

`SUPER::` still works, exactly as it does for `bless`-based classes:

```

class Dog :isa(Animal) {
    method introduce {
        $self->SUPER::introduce;
        say "and I can fetch";
    }
}

```

`__CLASS__`

Inside a field initialiser or a method body, `__CLASS__` is the class currently being constructed. In a subclass, `__CLASS__` refers to the subclass even inside an inherited method or an inherited field initialiser. Use it for factory-style hooks:

```
class Base {
  sub DEFAULT_X { 10 }
  field $x = __CLASS__->DEFAULT_X;
  method x { $x }
}

class Tuned :isa(Base) {
  sub DEFAULT_X { 99 }
}

say Tuned->new->x;                                     # 99
```

Interoperation with classical OO

A class-declared class is a regular package. Code that uses `ref`, the `isa` operator, `UNIVERSAL::isa`, and `UNIVERSAL::can` keeps working:

```
my $c = Circle->new(radius => 2);
say ref $c;                                     # Circle
say $c isa Shape ? 'yes' : 'no';                # yes
say $c->can('area') ? 'yes' : 'no';             # yes
```

A classical class can subclass a modern class and vice versa, as long as the classical side goes through methods (never through `$self->{fieldname}`) - see the [migration chapter](#) for the full interop story.

Edge cases worth knowing

- **Do not write `sub new`.** The constructor is generated. A user-written `new` inside a class body is rejected at compile time.
- **Treat `@ISA` as read-only** for class-declared classes. Runtime manipulation is undefined behaviour.
- **`$self` is read-only** inside a method. Assigning to `$self` is a runtime error.
- **package cannot reopen a class.** Once a namespace has been declared with `class`, a later `package` statement for the same name is a compile-time error, and vice versa.
- **Statement-form scope.** `class Foo;` consumes the rest of the enclosing block, typically the rest of the file. A subsequent `class` or `package` ends the body; there is no way to close a statement-form class early.

Where to go next

- *Inheritance and method resolution* - the full story of how method lookup works across `:isa` chains and across mixed classical/modern hierarchies.
- *Roles and delegation* - the recommended answer to “I want to share behaviour across unrelated classes.”
- *Migrating from classical to modern* - if you are staring at a `bless`-based class and wondering what the equivalent `class` block looks like.

Classical OO

Before the `class` feature landed in 5.38, Perl’s OO was built from three primitives: a package, a reference, and the `bless` function. Those three primitives are still present - the modern `class` keyword sits on top of the same machinery - and every pre-5.38 codebase, plus most of CPAN, still uses them directly.

This chapter is for reading and maintaining that code. If you are writing a new class, start with the *modern class chapter* instead.

The three primitives

1. A **package** defines a namespace. Any package can be a class; the difference is purely how it is used.
2. A **reference** holds instance data. The convention is a hash reference, but any reference type works.
3. **bless** tags the referent with a class name, so that method dispatch through the reference looks up subs in that package.

That is the whole model. Everything else - constructors, accessors, inheritance, roles - is a convention built on top.

A minimal class

```
package File;

sub new {
    my ($class, %args) = @_;
    my $self = {
        path    => $args{path},
        content => $args{content},
    };
    return bless $self, $class;
}

sub path    { $_[0]{path} }
sub content { $_[0]{content} }

sub print_info {
    my $self = shift;
```

(continues on next page)

(continued from previous page)

```
    print "This file is at $self->{path}\n";
}

1;
```

Key conventions, each load-bearing:

- **sub new** is the constructor. It is an ordinary method; the language has no constructor keyword. By convention it is named `new` and returns the blessed reference.
- **my (\$class, %args) = @_** unpacks the invocant first, then the named arguments. A class-method call `File->new(path => ...)` passes the class name as the first argument.
- **bless \$self, \$class** attaches the class tag. Using `$class` (the actual invocant) rather than a hard-coded `'File'` is what makes the constructor *inheritance-safe*: a subclass inheriting `new` will bless into the subclass's name.
- **return is usually omitted** after `bless` because `bless` returns its first argument. `bless $self, $class` as the last expression in the sub is idiomatic. We write it explicitly here for readability.

Accessor methods by hand

The model gives you no accessors; you write them. The terse form exploits the fact that `$_[0]` is the invocant:

```
sub path    { $_[0]{path} }
sub content { $_[0]{content} }
```

The named-parameter form is clearer and what you should write when the accessor does anything non-trivial:

```
sub path {
    my $self = shift;
    return $self->{path};
}
```

A read/write accessor:

```
sub name {
    my $self = shift;
    $self->{name} = shift if @_;
    return $self->{name};
}
```

This is the boilerplate the *modern class feature* replaces with `:reader` and `:writer`.

Why hash references

Hash references dominate because they let you add and rename fields without changing the layout, and because named keys survive a `Data::Dumper` or `JSON::encode` round-trip. Array references are faster and more compact but fragile: renumbering a slot breaks every access site.

Hash-backed instances have one serious drawback: no encapsulation. Any caller can reach into the object's internals with `$obj->{path}`. The classical model has no tool to stop them - self-discipline and code review are the only defences. This is the single biggest reason to prefer the *modern class feature* for new code.

Inheritance with @ISA

Inheritance is a package-level list named `@ISA`. Method dispatch walks it when the current package does not define the method.

```
package File::MP3;
our @ISA = ('File');

sub print_info {
    my $self = shift;
    $self->SUPER::print_info;
    print "Title: $self->{title}\n";
}

1;
```

`SUPER::` means “look for this method starting one step up the ISA chain from the package the current method was *compiled in*”

- not from the object's class. That subtlety matters in diamond inheritance; the *inheritance chapter* covers it.

Declaring the parent - four ways

There are four spellings of the same idea. Prefer the third for new code you are still maintaining in the classical style.

```
# 1. Raw @ISA assignment.
package Child;
our @ISA = ('Parent');

# 2. use base.
package Child;
use base 'Parent';           # deprecated in modern Perl

# 3. use parent.
package Child;
use parent 'Parent';         # current recommendation for classical

# 4. @ISA with a require.
```

(continues on next page)

(continued from previous page)

```
package Child;
require Parent;
our @ISA = ('Parent');
```

use parent handles the load of the parent module and the @ISA push in one line. use base is an older variant that also imports Exporter-style fields; avoid it.

The \$self invocant

Every method is a subroutine whose first argument is either the class name (for class methods like new) or the object (for instance methods). The convention is to shift it into a variable named \$self:

```
sub method {
    my $self = shift;
    ...
}
```

Or unpack it together with other arguments:

```
sub method {
    my ($self, $arg1, $arg2) = @_;
    ...
}
```

There is no runtime enforcement. If you forget to shift \$self, your first user-supplied argument silently becomes \$self and chaos ensues. Static analysers catch this; the language does not.

Destructors

A DESTROY method, if defined, runs when the object's refcount falls to zero. Use it to release external resources:

```
sub DESTROY {
    my $self = shift;
    close $self->{fh} if defined $self->{fh};
}
```

Pitfalls that bite every project at least once:

- DESTROY runs at unpredictable times. Do not rely on it for correctness-critical cleanup - explicit \$obj->close beats implicit destruction.
- DESTROY can see a partially constructed object if new croaked mid-way. Defensive defined checks are cheap insurance.
- During global destruction (END time), dependencies may already be gone. A DESTROY that walks references can fail with “Attempt to free unreferenced scalar” or similar. Wrap risky cleanup in eval { ... }.

AUTOLOAD

If a method call finds no matching sub in the class or its ancestors, Perl looks for an AUTOLOAD sub. The called method's name appears in our \$AUTOLOAD:

```

our $AUTOLOAD;
sub AUTOLOAD {
    my $self = shift;
    my $name = $AUTOLOAD;
    $name =~ s/.*::://;
    return if $name eq 'DESTROY';      # critical
    ...
}

```

The DESTROY guard is mandatory: without it, object destruction triggers AUTOLOAD with \$name = 'DESTROY' and you either misbehave or crash.

AUTOLOAD is powerful and easy to abuse. The cases it earns its keep:

- Lazy accessor generation - synthesise an accessor on first call, then install it as a real sub via typeglob assignment so the next call is direct.
- Forwarding - delegate unknown methods to a contained object (but see [Roles and delegation](#) for a cleaner approach).

Cases where it hurts more than it helps: pretending to be a full-blown MOP, catching typos as silent no-ops, implementing “classes” whose entire surface is AUTOLOAD.

A worked example

A classical Counter class with read/write accessor and inheritance:

```

package Counter;

sub new {
    my ($class, %args) = @_;
    my $self = {
        value => $args{value} // 0,
        step  => $args{step}  // 1,
    };
    return bless $self, $class;
}

sub value {
    my $self = shift;
    $self->{value} = shift if @_;
    return $self->{value};
}

sub step { $_[0]{step} }

sub tick {
    my $self = shift;

```

(continues on next page)

(continued from previous page)

```
    $self->{value} += $self->{step};  
}  
  
1;
```

A subclass that doubles the step:

```
package Counter::Double;  
use parent 'Counter';  
  
sub new {  
    my ($class, %args) = @_;  
    $args{step} //= 2;  
    return $class->SUPER::new(%args);  
}  
  
1;
```

Compared with the *modern class* version, the classical form is about three times as many lines and every one of them is a place to get it wrong.

When classical OO is the right call

Despite everything, there are legitimate reasons to stick with bless-based classes:

- **You are maintaining existing code.** Rewriting a working classical class into class form is a migration project, not a free lunch - see the *migration chapter*.
- **You need to support older Perl.** If your code must run on Perl before 5.38, the class feature is unavailable.
- **You need array-backed or code-backed objects.** class only creates hash-backed instances. If you are building a performance-critical structure with array-reference instances, classical bless is still the answer.
- **You are doing something the class feature deliberately forbids** - reopening the package from outside, multiple inheritance, or deep metaprogramming via @ISA manipulation.

A historical note on CPAN OO systems

Because the classical model leaves so much to convention, CPAN grew an ecosystem of modules that layer declarative syntax on top:

- **Moose** - feature-complete OO system with a type miniature, roles, method modifiers, a full introspection API, and a large extension ecosystem. Heavy at load time; the dominant choice for large applications built between 2007 and roughly 2020.
- **Moo** - a subset of Moose's API without the introspection layer. Lighter, pure-Perl, API-compatible enough to let a Moo class and a Moose class share an inheritance tree.

- **Class::Accessor** - generates simple accessors and new. Minimal, pure-Perl, no roles.
- **Class::Tiny** - the smallest of the accessor-generator modules. All accessors are read/write, no type checks.
- **Object::Pad** - the experimental prototype that became the core `class` feature. Still installable on CPAN for code that needs the feature on older Perls.
- **Role::Tiny** - role composition for projects that do not use Moose but still want does-style sharing.

These systems are **historical context** in pperl. New code in pperl should use the core `class` feature, which covers the ground that Moose/Moo pioneered without the load-time and dependency cost. When you inherit a codebase that uses one of them, learn just enough of its surface to stay productive and plan the migration with the *migration chapter*.

Inheritance and method resolution

Method lookup in Perl is simple in the single-inheritance case and genuinely subtle once diamonds or mixed class systems enter the picture. This chapter covers the full story: the MRO algorithms, `SUPER::`, the `isa` operator, `can`, `DOES`, and the interaction between classical and modern classes in one hierarchy.

The lookup algorithm

When you write `$obj->foo(@args)`, Perl does the following:

1. Determine the class. For an object, it is the class the referent was blessed into; for a class-method call, it is the invocant itself.
2. Walk the class's **method resolution order** (MRO) in sequence. For each package in the order, look for a sub named `foo`.
3. The first match wins. If no match is found in any class on the MRO, walk again looking for `AUTOLOAD`.
4. If no `AUTOLOAD` is found either, throw `Can't locate object method "foo" via package "Class"`.

The MRO is what the class's `@ISA` chain linearises to. The linearisation rule depends on which MRO is in effect.

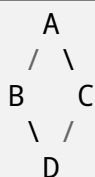
The two MROs: `dfs` and `c3`

Perl ships two MRO algorithms:

- **dfs** - depth-first search of `@ISA`. The default. Walk each parent, and each parent's parents, left-to-right, diving fully into each subtree before moving on.
- **c3** - the C3 linearisation used by Python, Raku, and many others. Guarantees monotonicity: a subclass never appears later than one of its ancestors in the lookup order.

For single inheritance the two are identical: the list walks straight up the chain.

For multiple inheritance they diverge. Consider the diamond:



- **dfs:** D, B, A, C, A (collapsed to D, B, A, C). A method on C that overrides A is **shadowed** by A because A is reached through B first.
- **c3:** D, B, C, A. C's override is visible.

C3 is almost always what you want under multiple inheritance. Opt in per-class:

```
use mro 'c3';
package Diamond::D;
our @ISA = ('Diamond::B', 'Diamond::C');
```

For classes declared with `class`, multiple inheritance is not supported at all, so the MRO choice is purely a linear walk and the default dfs is fine.

SUPER::

`$self->SUPER::foo(@args)` calls `foo` in one of the parent classes of *the package the current method was compiled in*. That last phrase is the one everyone gets wrong.

```
package Animal;
sub speak { 'some sound' }

package Dog;
our @ISA = ('Animal');
sub speak { 'woof' }

package Puppy;
our @ISA = ('Dog');
sub speak {
    my $self = shift;
    return $self->SUPER::speak;
}
```

`Puppy::speak` calls `SUPER::speak`. Perl looks up from **Puppy** (the compilation package), not from `ref($self)`. The search starts in Puppy's `@ISA`, so it finds `Dog::speak` and returns 'woof'.

This matters when methods are inherited and the child calls `SUPER::` from within. Because `SUPER::` is anchored to the compile-time package, a method inherited from `Dog` and invoked on a `Puppy` object will still call `Animal::` via its `SUPER::`, not `Dog`. Usually that is the right behaviour; when it is not, reach for explicit chain navigation with `next::method`, covered below.

next::method - the MRO-aware SUPER

`mro::next::method` (exposed as `$self->next::method(@args)`) walks the live MRO from the current method's *actual* call position, not from the compile-time package:

```

use mro 'c3';

sub speak {
    my $self = shift;
    return $self->next::method;
}

```

Under C3 with a diamond, `next::method` is what you want; `SUPER::` can skip a sibling branch because it only consults the left-most ancestor.

The isa operator

The `isa` operator tests class membership without the footgun of `UNIVERSAL::isa` called as a function:

```

if ($obj isa 'File::MP3') { ... }
if ($obj isa File::MP3)    { ... }    # bareword form

```

Both bless-based and `class`-declared classes answer `isa` correctly, and subclass relationships flow through `:isa` and `@ISA` identically.

Three rules worth remembering:

- `isa` returns false (not an error) when given a non-object. `42 isa Foo` is false. `undef isa Foo` is false.
- `isa` respects the live MRO at the moment of the call. If code pushes onto `@ISA` at runtime, `isa` sees it. (This is another reason pperl rejects `@ISA` manipulation for class-declared classes - the guarantees would become muddy.)
- Prefer the `isa` operator over `UNIVERSAL::isa($obj, 'Class')`. The operator short-circuits on non-objects instead of croaking; the function-call form is also defeated by overloaded objects.

can

`$obj->can('method')` returns a code reference if the object has a method by that name, `undef` otherwise:

```

if (my $code = $obj->can('save')) {
    $code->($obj, @args);
}

```

Uses:

- Feature-detection before calling: avoids the Can't locate object method error.
- Dispatching to a method by name from a string held in a variable - more readable than the `$obj->$name` form when the method might not exist.

Non-uses:

- Do **not** use `can` to decide whether an object implements a conceptual role. A class can define a `save` method that does something completely unrelated to persistence. For role membership, see *roles and delegation*.

DOES

`$obj->DOES('RoleOrClass')` is the role-aware counterpart to `isa`. By default `DOES` delegates to `isa`, so for classes without explicit role machinery the two behave the same. Role systems (classical Moose, Moo, and the future core role feature) override `DOES` to also return true for roles the class consumes.

```
if ($obj->DOES('Printable')) { ... }
```

In pperl, `DOES` is the right method to check when you want to ask “does this object implement the contract?” rather than “does this object inherit from this class?”

Mixed classical / modern hierarchies

class-declared and bless-based classes interoperate in both directions, subject to a handful of rules.

A modern class inheriting from a classical class

```
package Shape;                                # classical
sub new {
    my ($class, %args) = @_;
    return bless { colour => $args{colour} // 'black' }, $class;
}
sub colour { $_[0]{colour} }

use feature 'class';
class Circle :isa(Shape) {                    # modern inherits classical
    field $radius :param :reader;
    method area { 3.14159265 * $radius ** 2 }
}
```

This works. The modern class’s generated constructor delegates to the parent’s `new`, and methods inherited from the classical parent appear on the modern child. The parent’s hash-slot state lives *alongside* the child’s field storage.

Caveat: the modern child cannot reach into the parent’s hash. `$self->{colour}` is not legal inside a method of `Circle`; you must call `$self->colour`.

A classical class inheriting from a modern class

```
use feature 'class';
class Animal {
    field $name :param :reader;
```

(continues on next page)

(continued from previous page)

```

    method speak { 'some sound' }
}

package Dog;                                # classical inherits modern
our @ISA = ('Animal');
sub new {
    my ($class, %args) = @_;
    return $class->SUPER::new(%args);
}
sub speak { 'woof' }
```

This also works - the modern parent's generated constructor is inherited, and `SUPER::new` resolves to it. The classical child must not try to mutate the parent's fields directly; fields are private to the class that declared them.

What fails in mixed hierarchies

- **Reaching `$self->{fieldname}` across the boundary.** Modern fields are not hash slots, and classical hash slots are not visible as fields. Always cross the boundary through a method.
- **Multiple inheritance into a modern class.** The `:isa` attribute only takes one target. If you need to mix behaviour from multiple places, see [roles and delegation](#).
- **Runtime `@ISA` manipulation on a modern class.** Undefined behaviour. Treat `@ISA` as read-only for any class declared with `class`.

Why multiple inheritance usually hurts

Every language with multiple inheritance eventually discovers the same pathologies. Method lookup becomes order-dependent; diamond problems force you to think about linearisation; the “I inherited twice from the same root” case tangles initialisation. Roles - composition of behaviour without an inheritance relationship - solve the underlying sharing problem without the pathologies.

In pperl the `class` feature forbids multiple inheritance deliberately. Classical code that uses it should be migrating towards a role model; the [roles chapter](#) is the next stop.

Further reading

- `class` - full syntax of modern class declarations including `:isa`.
- `bless` - the primitive under classical inheritance.
- `ref` - reports a blessed reference's class name.
- `isa` - the preferred class- membership operator.
- [Roles and delegation](#) - the composition- based alternative to multiple inheritance.
- [Migration](#) - walking an existing hierarchy from classical to modern.

Roles and delegation

Inheritance answers “what is this object?” Roles and delegation answer a different question: “what can this object do?” When two unrelated classes need to share behaviour, forcing them under a common parent is the wrong tool. Composition - giving each class the behaviour it needs without pretending they are related - is the right one.

This chapter covers the three composition patterns that pperl code uses in practice:

1. **Mixin-style sharing** via a tiny helper package.
2. **Role composition** via `Role::Tiny` or a Moose-family system in legacy code.
3. **Delegation** - forwarding methods to a contained object.

The core `class` feature does not yet ship a role primitive. That is a deliberate staging decision upstream: the `class` feature landed first, roles are a separate proposal. Until the core role feature ships, the patterns in this chapter are the working answers.

The problem

Imagine a `Radio` class and a `Computer` class. Both have an on/off switch. They are not related - a radio is not a computer, and neither specialises a hypothetical `Machine`. A `HasOnOffSwitch` parent class would be an artificial grouping whose only purpose is reuse of two methods.

Three solutions, in increasing order of ambition:

- Copy the `turn_on` / `turn_off` methods into each class. Tolerable for two classes, painful for ten.
- Extract them into a helper package, use one of the sharing mechanisms below.
- Use a role system with formal composition semantics.

Pattern 1: manual mixin

The smallest possible “share these methods” mechanism is a plain package that installs subroutines into the consuming class at use time:

```
package HasOnOffSwitch;

sub import {
    my $target = caller;
    no strict 'refs';
    *{"${target}::turn_on"} = sub { $_[0]->{is_on} = 1 };
    *{"${target}::turn_off"} = sub { $_[0]->{is_on} = 0 };
    *{"${target}::is_on"} = sub { $_[0]->{is_on} };
}

1;
```

Used:

```
package Radio;
use HasOnOffSwitch;

sub new { bless { is_on => 0 }, shift }
```

This works. It is also brittle: it reaches into the consuming class's hash directly (`$_[0]->{is_on}`), which is exactly the encapsulation violation that classical hash-backed OO is famous for. It breaks if the consumer is a `class`-declared class, because fields are not hash slots.

Use this pattern only for very small shared behaviour in a classical codebase. For anything larger, use a real role system.

Pattern 2: `Role::Tiny`

`Role::Tiny` is the lightweight role system that works with plain classical classes, with Moo, and with Moose (on the consuming side). It is the closest thing to a portable role primitive in the CPAN ecosystem.

```
package HasOnOffSwitch;
use Role::Tiny;

sub turn_on { $_[0]->is_on(1) }
sub turn_off { $_[0]->is_on(0) }

requires 'is_on';      # the consumer must provide this accessor

1;
```

A consumer picks the role up with with:

```
package Radio;
use Role::Tiny::With;
with 'HasOnOffSwitch';

sub new { bless { is_on => 0 }, shift }
sub is_on { @_ > 1 ? $_[0]{is_on} = $_[1] : $_[0]{is_on} }
```

What `Role::Tiny` gives you over the manual mixin:

- **Requirements.** `requires 'is_on'` fails at composition time if the consumer does not supply `is_on`. The manual mixin fails silently at the first method call.
- **Conflict detection.** Consuming two roles that both define the same method is a composition-time error, not silent last- definition-wins.
- **Method modifiers** - before, after, around - if you want them.

`Role::Tiny` is **not** the same thing as Moose's role system. Its reach is intentionally narrow. If the codebase already uses Moose, use `Moose::Role`; if it uses Moo, use `Moo::Role`.

Pattern 3: Moose-family roles

Legacy codebases built on Moose or Moo use `Moose::Role` and `Moo::Role` respectively. The surface looks like the role system the core `class` feature will eventually gain:

```
package HasOnOffSwitch;
use Moose::Role;

has is_on => (is => 'rw', isa => 'Bool', default => 0);

sub turn_on { $_[0]->is_on(1) }
sub turn_off { $_[0]->is_on(0) }

1;

package Radio;
use Moose;
with 'HasOnOffSwitch';

1;
```

In pperl, Moose still works. New code should not reach for it - the runtime cost and dependency weight are both large, and the core `class` feature now covers the declarative side of what Moose was built for. When the core role feature lands, migrating from `Moose::Role` to the core role will be the natural next step.

Pattern 4: delegation

Delegation says “I don’t do this myself, but I hold something that does.” Instead of inheriting behaviour, contain an object that provides it, and forward methods to it.

The manual form:

```
use feature 'class';

class Player {
    field $radio :param;

    method turn_on { $radio->turn_on }
    method turn_off { $radio->turn_off }
    method is_on { $radio->is_on }
}
```

Each forwarded method is three lines of mechanical plumbing. For two or three methods that is fine; for a wide surface, reach for `Class::Method::Delegate` or a Moose handles accessor:

```
package Player;
use Moose;

has radio => (
    is => 'ro',
```

(continues on next page)

(continued from previous page)

```
handles => ['turn_on', 'turn_off', 'is_on'],
);
```

handles generates the forwarders at class-build time.

When to delegate vs inherit

- **Delegate** when the contained object is a *collaborator* - something the owner *has* rather than *is*. A Player has a Radio; a Player is not a kind of Radio.
- **Inherit** when the child genuinely *is* a specialised parent. A Circle is a kind of Shape; a File::MP3 is a kind of File.

If the answer to “is-a or has-a?” is not immediate, the relation is probably has-a. Defaulting to composition is almost always the less wrong call - you can refactor composition into inheritance with less pain than the reverse.

Delegation with lazy construction

A common pattern: the collaborator is expensive to build, so build it on first access. With the modern `class` feature:

```
use feature 'class';

class Logger {
    field $path :param;
    field $fh;

    method fh {
        unless ($fh) {
            open $fh, '>>', $path or die "open $path: $!";
        }
        return $fh;
    }

    method log ($msg) {
        print {$self->fh} $msg, "\n";
    }
}
```

fh is a reader that also handles the one-shot construction. Callers see a plain method; the lazy cache is private.

What not to do

- **Multiple inheritance to share two methods.** You pay the full MRO cost for a tiny win, and the next person to inherit from your class inherits the MRO pain too. Use a role or a mixin.
- **AUTOLOAD as a delegator.** Convenient in the five-minute demo, impossible to introspect, and it eats typos. Write explicit forwarders or use handles.

- **Roles that carry state.** A role that defines both methods and attributes is really a mini-class. The cleanest role defines behaviour that the consuming class's state supports; the role requires the accessors it needs and trusts the consumer to provide them. The manual-mixin example in pattern 1 got this wrong by reaching into `$_[0] -> {is_on}` directly.
- **Deep delegation chains.** If `$player->radio->volume->level` appears repeatedly, add a `volume_level` accessor on `Player`. Train-wreck method chains are a composition smell.

Further reading

- *Modern classes* - the base layer that composition sits on top of.
- *Inheritance and method resolution* - when the answer really is inheritance.
- *Migration* - how role-based legacy code translates into the core class model (spoiler: mostly unchanged; roles migrate last).
- `class` - reference for the modern class syntax.
- `isa` - class-membership test; pair with `DOES` for role membership.

Migrating from classical to modern

If you are maintaining a `bless`-based codebase and wondering what the equivalent `class` block looks like, this chapter is the recipe. The translation is mostly mechanical, but the places where it is not mechanical are where real design decisions surface - and where the migration pays for itself.

Before starting, read the *modern class chapter* for the syntax you are translating *into*, and skim the *classical OO chapter* for any constructs the existing code uses that you may not remember in detail.

When to migrate

Migration is not free. Do it when you get real value:

- The class is under **active development**. You are about to add fields, add methods, or change the constructor contract. Doing that work against the `class` feature is less error-prone than doing it against hand-rolled hash slots.
- The class has **unclear encapsulation**. Callers reach into `$obj->{fieldname}` directly, and you want to stop that without rewriting every call site at once. The `class` feature makes the violation a compile-time error the moment you switch.
- The class sits on top of **Moose or Moo**, and you are paying the load-time cost without using the dependency-heavy features. The modern `class` is a straightforward replacement for most Moose/Moo classes that only use `has` and `with`.

Do **not** migrate a class that is working and frozen. A stable classical class that will not see new code for another five years is exactly as good as its modern equivalent, and the migration itself will introduce bugs that the stable class does not have.

The mechanical translation

Take this classical class:

```

package Counter;

sub new {
    my ($class, %args) = @_;
    my $self = {
        value => $args{value} // 0,
        step  => $args{step}  // 1,
    };
    die "step must be positive" if $self->{step} <= 0;
    return bless $self, $class;
}

sub value {
    my $self = shift;
    $self->{value} = shift if @_;
    return $self->{value};
}

sub step { $_[0]{step} }

sub tick {
    my $self = shift;
    $self->{value} += $self->{step};
}

1;

```

The modern form:

```

use v5.38;
use feature 'class';
no warnings 'experimental::class';

class Counter {
    field $value :param :reader :writer = 0;
    field $step  :param :reader          = 1;

    ADJUST {
        die "step must be positive" if $step <= 0;
    }

    method tick { $value += $step }
}

1;

```

The mapping, piece by piece:

Classical construct	Modern construct
<code>package Counter;</code>	<code>class Counter { ... }</code>
<code>sub new { ... bless ... }</code>	<i>(deleted - generated)</i>
Constructor argument defaulting with <code>//</code>	<code>= DEFAULT</code> or <code>:param ... = DE-FAULT</code>
<code>\$args{foo}</code> stored in <code>\$self->{foo}</code>	<code>field \$foo :param</code>
Read-only accessor	<code>:reader</code>
Read/write accessor	<code>:reader :writer</code>
Validation at the end of <code>new</code>	<code>ADJUST</code> block
<code>\$self->{foo}</code> inside a method	<code>\$foo</code> (the field name)
<code>my \$self = shift</code>	<i>(deleted - implicit)</i>
<code>use parent 'Base'; /our @ISA = ('Base')</code>	<code>class Sub :isa(Base) { ... }</code>
<code>\$self->SUPER::foo(@args)</code>	unchanged - still works

The non-mechanical parts

Everything above is rote translation. The places where you need to think are all variants of “the classical code was doing something the class feature does not do the same way.”

Multiple inheritance

class does not support `:isa(A, B)`. If the classical class has `@ISA = ('Parent1', 'Parent2')`, you have three options:

1. **Collapse to single inheritance.** Usually the second parent is a mixin or role in disguise. Migrate it into a *role* and consume it with `with`.
2. **Collapse to composition.** If the second parent is a collaborator rather than an ancestor, migrate to delegation instead of inheritance.
3. **Do not migrate the class.** If the multiple inheritance is genuinely load-bearing and neither of the above fits, the class is a poor fit for the modern feature. Leave it classical until the core role feature lands.

In practice option 1 or 2 applies to about 95% of the multiple- inheritance cases in real code.

Direct hash access from callers

If external callers do `$counter->{value}` rather than `$counter->value`, migrating the class will break them. The compiler error is clean and local - the error message names the site - but the migration is no longer a drop-in.

Mitigation strategy:

1. **Audit first.** Grep for `$obj_name->{` patterns across the codebase. The count is usually small.
2. **Add an accessor** to the classical class that covers every accessed field.
3. **Replace** `$obj->{fieldname}` call sites with `$obj->fieldname()` in a dedicated commit.

4. Then migrate the class to modern form.

Splitting the refactor into two phases keeps each commit small and bisectable.

AUTOLOAD

AUTOLOAD in a classical class has no modern equivalent. The `class` feature does not let you catch missing method calls.

If the AUTOLOAD is doing one of the common patterns, translate it:

- **Lazy accessor synthesis** - declare the accessors explicitly as fields with `:reader` or `:writer`. The `class` feature's code generation replaces what AUTOLOAD was doing.
- **Delegation to a contained object** - replace with explicit forwarder methods (see [roles and delegation](#)) or with a Moose-style `handles` accessor if the codebase can depend on Moose.
- **A generic proxy** - keep the class classical. The `class` feature is not the right tool.

DESTROY

`class` does not introduce a first-class destructor form. If the classical class has a `DESTROY`, write it as a plain method inside the `class` block - it is invoked through the same `refcount` hook:

```
class Logger {
    field $fh;

    ADJUST { ... }

    sub DESTROY {
        my $self = shift;
        close $self->{fh} if $self->{fh};
    }
}
```

Note the wart: inside `DESTROY` you cannot name the field by its declared name, because sub bodies do not see field bindings. Either use a regular method that `DESTROY` delegates to, or accept the slight asymmetry.

A cleaner pattern when the lifetime is under your control is to add an explicit `close` method and call it deterministically, leaving `DESTROY` as a belt-and-braces backup.

Non-hash instances

If the classical class does `bless \@self, $class` or `bless \$self, $class` - an array-backed or scalar-backed instance - the modern feature has no equivalent. Hash-like instance layout is baked in. Keep these classes classical.

Tied hashes / overloading

use overload and tie integrations work unchanged in a class block. Nothing special to do.

Migrating Moose / Moo classes

The translation from Moose to class is cleaner than from classical because Moose is already declarative. The rough map:

Moose / Moo	Modern class
use Moose; / use Moo;	use feature 'class'; plus block
has foo => (is => 'ro', ...)	field \$foo :param :reader
has foo => (is => 'rw', ...)	field \$foo :param :reader :writer
has foo => (default => sub { ... })	field \$foo = EXPR (or ADJUST)
extends 'Parent'	class Sub :isa(Parent) { ... }
BUILD { ... }	ADJUST { ... }
with 'Role::Name'	(wait for core roles)
has foo => (isa => 'Int', ...)	(no direct equivalent)
method_modifiers (before/after/around)	(no direct equivalent)

The gap items - type constraints, method modifiers, roles - are what keeps some Moose codebases on Moose for now. If your Moose use is limited to has, extends, and BUILD, the migration is straightforward. If it depends heavily on Moose's type system or method modifiers, the migration is a project, not a commit.

Testing the migration

A class migration that changes no external behaviour should pass the existing test suite unchanged. Steps:

1. Ensure the class has tests covering its public API. If it does not, write them **before** migrating - the tests are your migration safety net.
2. Migrate the class.
3. Run the tests. A clean pass is the completion signal.
4. Grep the codebase for \$obj->{ ... } reaches into the migrated class's instances. Fix any hits.
5. Remove any now-dead use parent, use base, explicit sub new, or accessor boilerplate the old form needed.

If step 3 fails, the failures usually fall into a short list:

- **Direct hash access** in a method that the old form tolerated and the new form rejects. Replace with field references.
- **A constructor argument that was silently accepted.** The generated constructor is strict about :param names. Either declare the parameter or change the call site.

- **A field initialiser that referenced `$self`.** Field initialisers run before `$self` is bound. Move the dependent logic into an ADJUST block.

What you get out of it

The classical-to-modern migration is unusual in that the result is both shorter and more correct:

- **Shorter.** The Counter example above drops from ~20 lines to ~10, and the removed lines are all boilerplate.
- **Private fields.** External code can no longer reach in. The encapsulation you used to defend in review is now defended by the compiler.
- **No forgotten bless.** The generated constructor is always inheritance-safe; you cannot ship a constructor that accidentally blesses into the wrong class.
- **No forgotten `$self = shift`.** The implicit bind cannot go missing.
- **A named feature flag.** no warnings 'experimental::class' documents at the top of the file what the class depends on.

Further reading

- *Modern classes* - reference for the syntax you are migrating into.
- *Classical OO* - reference for the syntax you are migrating out of.
- *Inheritance and method resolution* - the mixed- hierarchy chapter, useful while the codebase is half-migrated.
- *Roles and delegation* - where multiple inheritance ends up in the migrated code.
- `class`, `field`, `method` - the reference pages for the three keywords at the heart of the migration.
- `bless`, `ref`, `isa` - the primitives that underpin both old and new forms and that keep working across mixed hierarchies.

4.6 Debugging Perl programs

Perl ships with an unusually complete debugging surface: a built-in statement-by-statement debugger, a regex-engine tracer, a documented API for writing alternative debuggers, and a CPAN ecosystem of profilers, leak finders, post-mortem tools, and IDE integrations. pperl's reimplementations preserves the interpreter's hook points, so the entire stack works: `perl -d`, `Devel::NYTProf`, `Devel::Cover`, `Perl::LanguageServer`, and the rest of the `-d:Module` family.

4.6.1 Which chapter do I want?

Symptom	Start here
Program misbehaves and I want to inspect it live	<i>interactive-debugger</i>
Program crashes with a cryptic message	<i>exceptions</i>
I want to sprinkle prints and dumps	<i>print-and-die</i>
I want to turn more mistakes into compile errors	<i>preflight</i>
I need to stop on a specific line or condition	<i>breakpoints</i>
I need to see what a data structure actually holds	<i>inspecting-state</i>
I want to see every line / sub as it executes	<i>tracing</i>
I want breakpoints in VS Code / Neovim	<i>ide-dap</i>

4.6.2 The three layers of this guide

- **Orientation** - this page.
- **Working programmer** - the chapters listed above. Each is self-contained; reading a single one is enough to solve the problem it names.

4.6.3 Known gaps

Debugging surfaces where the ecosystem itself is thin, or where pperl’s implementation has not reached parity with upstream. Named here so readers do not waste time on them.

- **Regex tracer output format.** pperl’s regex engine is a native reimplementa-tion; use `re 'debug'` produces a trace but node names and field formats are not byte-identical to perl5’s. Consumers that parse the trace (few exist) will need ad-justment.
- **B::* introspection modules.** B::Deparse, B::Concise, and kin expose perl5’s op graph. pperl’s op representation differs; native equivalents are in progress.
- **XS-dependent introspection modules.** Devel::Peek, Devel::FindRef, De-vel::Gladiator, Devel::Refcount, Devel::Size, Devel::MAT::Dumper, De-vel::Leak family - all reach the SV through XS (C-level macros against the SV pointer). pperl mirrors perl5’s SV layout faithfully, but XS loading is not sup-ported; these modules therefore need native (Rust) reimplementations that expose the same Perl-level API. Pure-Perl Devel::Peek-workalike functionality is already provided natively by the built-in Devel::Peek; the others are in progress.
- **Enbugger on Perl ≥ 5.38.** Last release 2012; works anecdotally on modern perls but has no active maintainer. The `gdb-inject-perl` path is more reliable.
- **Async provenance.** No Perl equivalent to Node’s `AsyncLocalStorage`: a `Future` that fails in a distant callback carries no automatic record of who scheduled it.
- **Sentry breadcrumbs.** Sentry::Raven is functional but maintenance-only; no first-class breadcrumb support. The OpenTelemetry Perl SDK is the forward path for new installations.
- **Perl::LanguageServer attach-to-running-process.** Launch is supported; post-hoc attach is not.

4.6.4 See also

- `perlfunc/die`, `perlfunc/warn`, `perlfunc/eval`, `perlfunc/caller` - the language primitives the debugger and the stack-trace tools are built on.
- *PPerl Architecture guide* - how pperl is built and why the debugger surface works as it does.

Preflight: catching bugs before runtime

This chapter covers the pragmas, compile-time checks, and runtime hygiene settings that prevent most bugs from reaching the debugger at all. A clean preflight is the shortest path to a short debugging session.

Readers who already use `v5.36` or later run under strict and warnings by default and can skip straight to the diagnostics and autoflush sections. The rest of this chapter assumes you might be maintaining older code where neither is implicit.

Strict and warnings

Since Perl 5.36, use `v5.36`; (and any declaration of a higher version) implicitly enables `use strict`; and `use warnings`;. Modern scripts do not write those lines - the version declaration covers both:

```
use v5.40;
```

`use strict` refuses three classes of sloppy code at compile time: undeclared variables (`strict vars`), symbolic references (`strict refs`), and barewords used as strings (`strict subs`). `use warnings` turns on runtime diagnostics for uninitialised reads, lossy numeric conversions, filehandles used after close, and several dozen other sharp edges.

Legacy code that uses neither still benefits from both: add them, fix the symbols they complain about, commit.

Narrow opt-outs remain useful:

```
no strict 'refs';           # genuine symbolic deref
no warnings 'uninitialized'; # known-safe defaulting
no warnings 'experimental::try'; # on 5.36, 5.38 - see below
```

Never blanket-disable: `no warnings`; with no category list is almost always wrong.

Promote warnings to fatal for a region:

```
local $SIG{__WARN__} = sub { die $_[0] };
```

Useful at program top during development; the first warning becomes a stack-trace-producing crash.

use diagnostics

Expands each warning or fatal message into the full paragraph from perldiag:

```
use diagnostics;
```

The output is verbose - too verbose for production - but during debugging it explains what a terse message like “Odd number of elements in hash assignment” actually means and how to fix it.

Post-hoc version: pipe a program’s stderr through the `splain` command to annotate warnings after the fact.

Compile-time checks

`perl -c script.pl` parses the script and runs every `BEGIN` / `use`, then stops before the first runtime statement. A clean `-c` means the file and its dependencies compile; it does not mean the program runs correctly.

```
perl -c script.pl
script.pl syntax OK
```

Combine with `-w` to enable warnings globally for the compile phase and any compile-time code:

```
perl -cw script.pl
```

`perl -MO=Deparse script.pl` prints what perl actually parsed - precedence mis-parses, constant-folded branches, for rewritten as `while`. Use it to verify that perl saw what you meant:

```
perl -MO=Deparse script.pl
```

`perl -MO=Deparse, -p` forces full parenthesisation and makes precedence explicit.

`perl -MO=Concise script.pl` prints a compact op-tree dump; mostly for readers doing deep optimisation work.

autodie: make failure throw

Built-ins that signal failure via false return values (`open`, `close`, `chdir`, `unlink`) are easy to forget to check. `autodie` replaces them with versions that throw:

```
use autodie qw(open close chdir);

open my $fh, '<', $path;      # now dies on failure; no `or die`
↪needed
```

`autodie` is lexical: only calls in its scope are affected. Narrow the opt-out for a specific call site:

```
{
    no autodie 'open';
```

(continues on next page)

(continued from previous page)

```
open my $fh, '<', $path or warn "cannot read $path: $!";
}
```

Prefer `autodie` to the retired `Fatal` module, which did not cover functions returning counts.

Unbuffered output

When diagnostic output vanishes before a crash, or arrives interleaved oddly with `warn` messages, the cause is almost always block-buffered `stdout`. Set `$|` at the top of the script:

```
$| = 1;
```

Every `print` now flushes immediately. The cost is a small I/O-throughput hit; the gain is that what you see on the terminal matches what the program has actually done, in order.

For a specific filehandle without changing the selected one:

```
use IO::Handle;
$log_fh->autoflush(1);
```

Common early diagnostics and what they mean

Diagnostic	Meaning
Global symbol "\$x" requires explicit package name	Strict is on and <code>\$x</code> was never declared; add <code>my</code> .
Odd number of elements in hash assignment	A list assigned to a hash has odd count; a <code>qw(...)</code> or string with an embedded space produced two values where one was expected.
Use of uninitialized value in ...	An <code>undef</code> reached an operator. Usually a typo in a variable name or a missing return.
Name "main::x" used only once: possible typo	A package variable was referenced exactly once. Almost always a typo or a missing <code>my</code> .
Can't locate Foo.pm in @INC	A <code>use</code> or <code>require</code> could not find the module. Check spelling, <code>@INC</code> , <code>PERL5LIB</code> .
Reference found where even-sized list expected	<code>my %h = { ... }</code> - braces build a hashref; use parens.

Find out more

- [print-and-die](#) - when preflight is not enough and you need structured runtime diagnostics.
- [exceptions](#) - turning failure paths into typed exceptions the caller can match on.

Print debugging, warn, and the Carp family

This chapter covers non-interactive diagnostics: inserted prints, variable dumps, stack traces from inside the program. These techniques remain the fastest path to understanding “what does my code actually see here?” for bugs that are cheap to reproduce.

Readers who already know about `print` and `warn` will find here the structured variants that earn their keep: the Carp caller-reporting family, `Data::Dumper` and its modern replacements, and the stack-trace-everywhere one-liners.

warn vs print - pick the right stream

`warn` writes to STDERR; `print` defaults to STDOUT. Use `warn` for diagnostics: it keeps program output clean, survives STDOUT redirection, and is the stream loggers already capture.

```
warn "*** foo=[$foo] bar=[@bar]\n";
```

Wrap values in literal brackets so trailing whitespace and newlines are visible in the output. `"foo=[$foo]"` reveals `foo=[ ]` where `"foo=$foo"` just shows `foo= .`

Without a trailing `\n`, `warn` (and `die`) append at `FILE line N.` from the caller’s perspective. With `\n`, they do not. Rule of thumb: append `\n` for user-facing messages; omit it for developer diagnostics where you want location.

Cheap location tags from the compiler:

```
warn "reached $0 " . __FILE__ . ':' . __LINE__ . "\n";
warn "in @{$[__SUB__->name]}\n";           # 5.16+
```

The Carp family - report from the caller

When a library function fails, pointing at the library’s own source is almost never useful. Carp’s family reports from the caller’s frame:

Function	Reports from	Fatal?	Stack trace?
<code>carp</code>	caller	no	no
<code>croak</code>	caller	yes	no
<code>cluck</code>	caller	no	full
<code>confess</code>	caller	yes	full

```
use Carp;

sub open_config {
    my $path = shift // croak "open_config: missing path";
    open my $fh, '<', $path or croak "cannot read $path: $!";
    return $fh;
}
```

Library code should use `croak/carp`, not `die/warn` - the error message points at the caller’s line, which is where the fix goes.

Stack traces without touching the code

Carp::Always installs \$SIG{__DIE__} and \$SIG{__WARN__} to route through Carp::confess / Carp::cluck. Run a misbehaving script with:

```
perl -MCarp::Always script.pl
```

Every die and warn produces a full stack. No source changes. This is the first thing to reach for when a crash gives a useless one-line message.

Devel::Confess is a drop-in equivalent with fewer edge cases around overloaded objects:

```
perl -d:Confess script.pl
```

Both modules add cost to every warn and die; use them while investigating, not in steady production.

For a permanent in-code install:

```
use Carp;
$SIG{__DIE__} = sub { return if $^S; Carp::confess(@_) };
$SIG{__WARN__} = sub { Carp::cluck(@_) };
```

return if \$^S; - “return if we are inside eval”. Without this guard every caught exception pays the trace cost. See *exceptions* for the full \$SIG{__DIE__} pattern.

Dumping data structures

Pass a **reference** to every dumper; dumping a bare hash or array prints its flat-list form and loses the structure.

Data::Dumper - universal, core, eval-roundtrippable

```
use Data::Dumper;

$Data::Dumper::Sortkeys = 1;
$Data::Dumper::Indent   = 1;
$Data::Dumper::Terse     = 1;
$Data::Dumper::Deepcopy  = 1;

warn Dumper(\%config);
```

- Sortkeys makes output diff-stable.
- Indent=1 is the readable setting (0 = flat, 2 = default, 3 = with array indices).
- Terse=1 suppresses \$VAR1 = prefixes when you do not need round-trip-ability.
- Deepcopy=1 disables the “seen this ref before” \$VAR1->{ . . . } tracking - prints the data, not a graph reconstruction.

Data::Dumper::Concise is a drop-in with sensible defaults pre-applied; the DBIx::Class / Moose community standard.

Data::Printer - coloured, modern, readable

```
use DDP;
p %config;                                # not a reference - DDP handles both
```

Data::Printer prints better-looking output than Data::Dumper, colours by type, handles objects specially (class, attributes, overloads), and does not try to round-trip. Prefer it when the goal is reading the dump, not re-ingesting it.

Data::Dump - compact one-liner

```
use Data::Dump qw(dd);
dd \%config;                             # prints to STDERR in compact form
```

Gisle Aas's Data::Dump. Useful when the default Data::Dumper output is too verbose for a single warn line.

When to reach for which

- Data::Dumper when you might feed the output back into `eval` or compare two dumps.
- Data::Printer (DDP) when a human is reading the output.
- Data::Dump when you want a one-line summary.
- Data::Dumper::Concise on projects already using it.

Conditional debug prints

A compile-time constant is optimised out entirely when false:

```
use constant DEBUG => 0;
warn "state=$state\n" if DEBUG;
```

With `DEBUG => 0`, the `if DEBUG` test is folded at compile time and the `warn` never appears in the op tree - zero runtime cost.

Wire to the environment so the flag is flippable without editing:

```
use constant DEBUG => $ENV{APP_DEBUG} // 0;
```

Multiple channels via bitmask:

```
use constant { WEB => 1, SQL => 2, REGEX => 4 };
my $DEBUG = $ENV{APP_DEBUG} // 0;

sub dbg { my $c = shift; warn "[DBG] @_ \n" if $DEBUG & $c }

dbg WEB,    "request $req_id start\n";
dbg SQL,    "query: $sql\n";
```

(continues on next page)

(continued from previous page)

```
dbg REGEX, "matched: $&\n";           # (unless you care about perf -
↪see below)
```

Source-filter alternative: Smart::Comments

```
use Smart::Comments;

my @stuff = compute();
### @stuff           # prints: @stuff: [...]
### Checking key: $key # print, plus asserts $key is truthy
### Processing |=| ... for @stuff
```

Lines beginning `###` are rewritten into `prints` (and, with special syntax, progress bars and assertions). The module is a source filter - it rewrites the program text before compile - so reported line numbers can drift by one. When not imported, the `###` comments compile to nothing. The drift is the price of not cluttering the code with explicit warn lines.

Taint and \$& - incidental costs to know about

Referencing `$&`, `$``, or `$'` **anywhere** in the program forces perl to maintain these match-context globals for **every** regex match in the entire program, including matches inside loaded modules. Historically this imposed a noticeable slowdown; on modern perls the cost is smaller but non-zero, and the rule still applies: do not touch them if you can avoid it.

Use `@-` / `@+` (match position arrays), named captures, or the `/p` modifier with `${^MATCH}`, `${^PREMATCH}`, `${^POSTMATCH}` instead. The `/p` variants impose the cost only for matches that opt in.

Timestamped logging

Hand-rolled warn with `localtime` is adequate for one-off investigations:

```
use Time::HiRes qw(gettimeofday);
sub t { my ($s, $us) = gettimeofday; sprintf "%d.%06d", $s, $us }
warn sprintf "[%s] state=%s\n", t(), $state;
```

For anything more structured, use a logging framework rather than rolling your own:

- `Log::Any` - library-safe; emit logs without picking a backend. Pair with an adapter (`Log::Any::Adapter::Log4perl`, `::Stdout`, `::Syslog`) in the application.
- `Log::Log4perl` - category / level / appender / layout; config file-driven.
- `Log::Dispatch` - lower-level router; often used under `Log::Any`.

Standard levels, lowest to highest: `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`. Gate `TRACE` / `DEBUG` behind an env flag in production.

Find out more

- *exceptions* - `die`, `$@`, `$SIG{__DIE__}`, typed exception classes, the modern `try/catch`.
- *interactive-debugger* - when inserting prints costs more than launching `perl -d`.
- *tracing* - full-program line-by-line trace without editing the source.

Exceptions, die, and typed error objects

This chapter covers how to raise, catch, and classify runtime failures: `die` and `eval`, the native `try/catch`, special-variable hygiene (`$@`, `$!`, `$?`), `$SIG{__DIE__}`, and the design of exception classes.

Readers who know `die` and `eval { }` will find here the modern replacements for string-dispatching on exception messages and the hazards of `$SIG{__DIE__}` that the stock `eval` pattern silently hides.

die and eval

`die` throws an exception; the nearest enclosing `eval { }` catches it and leaves the message in `$@`:

```
eval {  
    risky();  
};  
if (my $err = $@) {  
    warn "caught: $err";  
}
```

Without a trailing `\n`, `die`'s argument gets at `FILE line N.` appended from the caller's frame. With `\n`, no location is appended - use for messages that will be shown to users, omit for messages you want to trace.

Never use the string form `eval "..."` for exception handling. It compiles its argument and has every hazard of user-supplied code.

The native try/catch

On Perl 5.34 and later, a dedicated `try/catch` is available behind a feature flag. Since 5.40 it is non-experimental:

```
use v5.40;  
use feature 'try';  
  
try {  
    risky();  
}  
catch ($e) {  
    warn "caught: $e";  
}
```

finally remains experimental as of 5.42. Expect:

```
use feature 'try';
no warnings 'experimental::try';
```

on perls between 5.34 and 5.40.

For code that must run on 5.34 or 5.36 and also on newer perls, `Feature::Compat::Try` picks the core feature where available and falls back to `Syntax::Keyword::Try`:

```
use Feature::Compat::Try;

try { risky(); }
catch ($e) { warn "caught: $e" }
```

Why native try over Try::Tiny

`Try::Tiny` still works and remains common. Prefer the native form for new code: it avoids the sub-call frame that `Try::Tiny` introduces (which shows up in stack traces), has zero startup cost, and composes with the `isa` operator cleanly.

These are retired - do not adopt:

- `TryCatch` - abandoned.
- `Error.pm` - superseded.
- `Exception::Class::TryCatch` - superseded by `Throwable`.

`$@` clobbering - the classic `eval` hazard

`$@` is set by `eval`; it is also observable globally. Any code that runs between `eval`'s return and your check on `$@` - a `DESTROY` method, a tied-variable handler, an unrelated nested `eval` - can overwrite it. Always copy first:

```
eval { risky(); };
my $err = $@;
return unless $err;
```

Native `try/catch` and `Syntax::Keyword::Try` avoid this pitfall by giving you the error as the catch parameter; raw `eval` does not. Prefer the former for any code that takes exceptions seriously.

Special variables adjacent to exceptions

Variable	Set by	Read immediately because...
<code>\$@</code>	<code>eval { } / die</code>	any intervening call can overwrite.
<code>\$!</code>	Last failed system call	any following syscall overwrites.
<code>\$?</code>	<code>system / backticks</code> child	the next child overwrites.
<code>^E</code>	Extended OS error	on non-Unix; on Linux usually mirrors <code>\$!</code> .

`$?` decoding after `system / backticks`:

```

if ($? == -1)          { warn "child failed to start: $!" }
elsif ($? & 127)        { warn "child killed by signal " . ($? & 127) }
    ↳}
elsif ($? & 128)        { warn "core dumped" }           # combined with
    ↳above
else                    { my $exit = $? >> 8; ... }

```

Classifying exceptions: dispatch on type

For non-trivial programs, distinguishing “user not found” from “database is down” from “programming error” by parsing error strings is fragile. Throw typed objects and match on them:

```

use v5.40;
use feature 'try';
use Scalar::Util qw(blessed);

try {
    do_business();
}
catch ($e) {
    if ($e isa MyApp::Err::NotFound) { return http_404($e) }
    elsif ($e isa MyApp::Err::AuthFailure) { return http_403($e) }
    elsif ($e isa MyApp::Err) { return http_500($e) }
    elsif (blessed($e) && $e->isa('DBIx::Class::Exception')) {
        return http_500($e);
    }
    else {
        MyApp::Err::Unknown->throw(cause => "$e");
    }
}

```

The `isa` operator (stable on 5.36+) does the right thing for overloaded objects and is zero-cost when the left operand is not a reference. Pair it with a typed wrapper at the outermost boundary so everything downstream is a `MyApp::Err`.

Designing exception classes

The 2026 default: Throwable role from CPAN, optionally via `Throwable::SugarFactory` for declarative hierarchies.

Minimum-viable class:

```

package MyApp::Err::NotFound;
use Moo;
with 'Throwable';

has resource => ( is => 'ro', required => 1 );
has id       => ( is => 'ro', required => 1 );

```

(continues on next page)

(continued from previous page)

```

sub message {
    my $self = shift;
    sprintf 'not found: %s/%s', $self->resource, $self->id;
}

1;

```

Usage:

```
MyApp::Err::NotFound->throw(resource => 'user', id => $uid);
```

throw is provided by the role; it dies with \$self (the blessed object). ->new without throw builds without raising.

Declarative form via Throwable::SugarFactory:

```

package MyApp::Err;
use Throwable::SugarFactory;

exception 'GenericError'      => 'something bad happened';
exception 'NotFound'          => 'resource not found'
    => (has => [ id => (is => 'ro')
    ↪]);
exception 'AuthFailure'       => 'auth failed'
    => (has => [ reason => (is => 'ro
    ↪') ]]);
exception 'AuthFailureExpired' => 'session expired'
    => (extends => 'AuthFailure');

1;

```

This generates: the class, a not_found(id => ...) constructor- and-throw helper, a is_not_found(\$e) predicate, and ->to_hash on every instance for JSON serialisation.

What a class must have

1. **A class name** - so \$e isa MyApp::Err::X dispatches.
2. **Structured fields** - id, resource, code.
3. **A human message** - without it, warn \$e prints MyApp::Err=HASH(0x...).
4. **Serialisation** - ->to_hash for logging / aggregators / JSON APIs.
5. **Origin** - file + line captured at throw. Throwable captures these automatically.
6. **A stack trace** if the error is worth investigating later: with 'StackTrace::Auto' adds ->stack_trace.

When not to use a class

Plain-string die remains correct for short scripts, one-offs, and any case where no caller will dispatch on the error:

```
die "permission denied: $path\n";
```

The trailing `\n` suppresses file-and-line appending; the message is user-facing.

Hash-ref die - `die { type => 'not_found', id => $uid }` - is a transitional form. The moment a caller writes `if (ref $e eq 'HASH' && $e->{type} eq 'not_found')` three times, refactor to a class.

Retired exception frameworks - do not adopt:

- `Error.pm` - use `Throwable` (role-based) or `Exception::Class`.
- `Class::Throwable` - functional; `Throwable` is the modern choice.
- `Exception::Class::TryCatch` - superseded by native `try`.

`$SIG{__DIE__}` - handle with care

`$SIG{__DIE__}` fires on every die, **including** die inside `eval`. A naive install turns every caught exception into noise:

```
# WRONG: fires on every eval that catches an error
$SIG{__DIE__} = sub { Carp::confess(@_) };
```

The `^S` special variable tells you whether you are inside an `eval`: `true` means “yes, someone is going to catch this”; `false` means “this is reaching the top”. Guard on it:

```
$SIG{__DIE__} = sub {
    return if ^S;           # inside eval - not for us
    my ($err) = @_;
    eval { ship_to_sentry($err) }; # never let the handler itself
    die
    warn "reporter failed: $@" if $@;
    die $err;               # re-raise
};
```

For Sentry / aggregator integration, wrap the reporter call in its own `eval` - a failing error reporter must not obscure the original exception.

Before installing on top of an existing handler, chain:

```
my $previous = $SIG{__DIE__};
$SIG{__DIE__} = sub {
    return if ^S;
    my ($err) = @_;
    eval { ship_to_sentry($err) };
    warn "reporter failed: $@" if $@;
    goto &$previous if $previous;
    die $err;
};
```


`$SIG{__WARN__}` - turn warnings fatal, or route them

Symmetric to `$SIG{__DIE__}`, fires on every warn:

```
$SIG{__WARN__} = sub { die $_[0] };      # promote warnings to
↳exceptions
```

Useful during development as a tripwire; rarely wanted in production. More common production use: route warnings through a logger.

Find out more

- *print-and-die* - the Carp family and stack-trace-everywhere switches.
- `perlfunc/die`, `perlfunc/eval`, `perlfunc/warn`, `perlvar` - reference pages for the language primitives.

The interactive debugger

This chapter covers `perl -d`: launching a program under the debugger, the prompt, the essential motion and inspection commands, and the stack-navigation vocabulary. After this chapter you can investigate any running program interactively.

Readers start from knowing `print` / `warn` work but are not enough: you need to stop, look around, peek at variables, walk the call stack, then resume. That is what this chapter delivers.

Launching

For any script `app.pl`, prepend `-d`:

```
perl -d app.pl --verbose input.txt
```

Arguments after the script name pass to the script as normal.

Common launch forms:

Form	Purpose
<code>perl -d script.pl args</code>	Debug the given script.
<code>perl -d -e 'CODE'</code>	Debug a one-liner expression.
<code>perl -de 0</code>	Interactive REPL sandbox.
<code>perl -d:Module script.pl</code>	Load <code>Devel::Module</code> as debugger.
<code>perl -d:NYTProf script</code>	Profile via NYTProf plugin.
<code>perl -S -d script.pl</code>	Search <code>\$PATH</code> for the script.

`-d` causes the compiler to emit per-statement hook points; the interpreter preloads `perl5db.pl` before the first runtime line.

Never pipe input to `perl -d` and never use `/dev/stdin` - the debugger needs a TTY to read commands from. Launch from a real terminal.

The prompt

```
Loading DB routines from perl5db.pl version 1.77
Editor support available.
```

```
Enter h or 'h h' for help, or 'man perldebug' for more help.
```

```
main::(app.pl:4):      my $key = 'welcome';
DB<1>
```

The line `main::(app.pl:4):` shows the next line **about to execute** - nothing in the program has run yet. Compile-time code (BEGIN, use) already ran; the first prompt comes before the first runtime statement.

DB<1> - the number is a command counter that increments on each command. !N recalls command number N. DB<<N>> with double brackets means you are inside a nested debugger (one expression you stepped into called another breakable expression).

The debugger always shows the line about to execute, never the line just executed.

Quitting

```
DB<1> q
```

Not quit, not exit. Typing `exit` runs Perl's `exit` inside the debugger's `eval`, which is not the same. `^D` at the prompt works on most terminals.

Help

```
DB<1> h          # one-screen summary
DB<1> h h        # full help (long - use the pager)
DB<1> h b        # help on one command
DB<1> |h h       # pipe full help through $ENV{PAGER}
```

Where am I?

`v` views a window around the current line. `l` lists a specified range:

```
DB<1> v
1      #!/usr/bin/perl
2      use v5.40;
3
4==>   my $key = 'welcome';
5:     my %data = (
6         'this'    => qw(that),
7         'tom'     => qw(and jerry),
8         'welcome' => q>Hello World),
9         'zip'     => q(welcome),
10    );
```

- `==>` marks the next line to execute.

- `:` marks executable lines - you can set a breakpoint there.
- Lines without `:` are not breakable (comments, blanks, continuations of a multi-line statement).

Navigation:

Command	Effect
<code>v</code>	View a window around the current line.
<code>v LINE</code>	View a window around LINE.
<code>l</code>	List the next window; repeat advances.
<code>l LINE</code>	List just LINE.
<code>l MIN-MAX</code>	List a range.
<code>l SUB</code>	List the first window of SUB.
<code>-</code>	Previous window.
<code>.</code>	Snap view back to the current line.
<code>f FILE</code>	Switch display to FILE (regex against %INC).
<code>/PAT/</code>	Forward search in source.
<code>?PAT?</code>	Backward search.

Motion: `s`, `n`, `Enter`, `c`, `r`

The four essential motion commands:

Command	Effect
<code>s</code>	Step - execute next statement; descend into subroutines.
<code>n</code>	Next - execute next statement; step over subroutines.
<code><Enter></code>	Repeat the last <code>s</code> or <code>n</code> .
<code>c</code>	Continue until next breakpoint or end.
<code>c LINE</code>	Continue to a one-shot breakpoint at LINE, then stop.
<code>c SUB</code>	Continue to the first line of SUB.
<code>r</code>	Return from the current sub; stop at the caller.

```
DB<1> s           # descend into the call
DB<2> n           # stay at this level
DB<3>             # <Enter> repeats the last motion
DB<4> c 42        # run to line 42 once
DB<5> r           # pop the current frame
```

A multi-line statement steps atomically - one `s` executes a whole multi-line hash assignment.

On a line that calls no subroutines, `s` and `n` are indistinguishable.

Printing values: p and x

```
DB<1> p $key
welcome

DB<2> p join ',', sort keys %data
this,tom,welcome,zip
```

p EXPR evaluates EXPR in the program's current package and prints the scalar-context result followed by a newline. p is flat - for a reference it prints HASH(0x...), not contents.

For structures, use x:

```
DB<3> x \%data
0 HASH(0x5651f9a82358)
  'this' => 'that'
  'tom' => 'and'
  'welcome' => 'jerry'
  'zip' => 'Hello World'
```

Always pass a **reference** (x \%h, x \@a) - x \%h dumps the flat key/value list, which hides the structure.

See *inspecting-state* for the full inspection vocabulary (V, X, y, m, M, i, depth controls).

The call stack: T, s, r

T prints a backtrace - the chain of sub calls that led to the current line:

```
DB<1> T
$ = main::handle_request(ref(HTTP::Request)) called from file 'app.
→pl' line 47
$ = main::dispatch('GET', '/users/42') called from file 'app.pl'
→line 19
@ = main::main() called from file 'app.pl' line 12
```

- \$ - called in scalar context.
- @ - called in list context.
- . - void context.
- Arguments are shown with Data::Dumper-style truncation; the exact length cap is the maxTraceLen option.

Navigate the stack with s (into a call) and r (out of the current frame). There is no “frame up / frame down” command: re-step if you go too deep, or use b to set a breakpoint at the caller and c to reach it.

To inspect an outer frame's lexicals, use y LEVEL from *inspecting-state* - it reads via Pad-Walker and shows the enclosing scope's my variables.

Evaluating Perl at the prompt

Anything the debugger does not recognise as a command is eval'd as Perl code in package DB at the current lexical scope:

```

DB<1> $counter = 0           # mutate state
DB<2> @ARGV = qw(new args)   # mutate @ARGV
DB<3> $obj->reset             # call a method
DB<4> m/(\w+)/ && p $1        # match against $_

```

A fresh my \$x at the prompt vanishes when the command returns - the prompt has its own lexical scope. Use a bareword, our, or chain on one line:

```

DB<1> our $tmp = compute();   # survives
DB<2> $tmp = compute(); p $tmp # one-line chain

```

Force Perl interpretation when an expression looks like a debugger command:

```

DB<1> ; h EXPR                # ; prefix forces eval
DB<2> + h EXPR                # + prefix works too

```

Restart and rerun

R restarts the program via exec (). History, breakpoints, actions, options, and the command line are preserved:

```
DB<1> R
```

rerun re-executes without leaving the debugger, preserving state:

Form	Effect
rerun	Restart, replaying no prior commands.
rerun -N	Restart and replay up to N commands back.
rerun N	Restart from history entry N.

Redefining a sub mid-session

Paste the entire sub NAME { ... } at the prompt. Or edit a replacement file and load it:

```
DB<1> do 'fixed.pl'
```

Subsequent calls use the new definition. Existing stack frames continue with the old body.

Commands so far

Command	Effect
h, h CMD, h h	Help: summary / one command / full.
CMD	Pipe the output through \$ENV{PAGER}.
q	Quit.
v, v LINE	View source around current / LINE.
l, l LINE, l SUB, l MIN-MAX	List.
.	Snap to current line.
-	Previous window.
f FILE	Switch display to FILE.
/PAT/, ?PAT?	Search forward / backward.
s	Step into.
n	Step over.
<Enter>	Repeat last s / n.
r	Return from current sub.
c, c LINE, c SUB	Continue / one-shot breakpoint.
p EXPR	Print (scalar context).
x EXPR	Dump (structured, pass a reference).
T	Stack backtrace.
R	Restart via exec ().
rerun [N -N]	Restart with history replay.

Find out more

- *breakpoints* - b, conditionals, actions, watches, timing matrix.
- *inspecting-state* - V, X, y, m, M, i, depth controls.
- *tracing* - t, AutoTrace, line-info output to a file.

Breakpoints, actions, and watches

This chapter covers stopping the debugger at a specific point - a line, a subroutine, a condition, a value change - and the prompt-hook machinery that lets you run code automatically at each stop.

Readers arriving here know how to launch `perl -d` and step one line at a time. The next skill is to skip straight to the interesting place and stop only when the problem is about to manifest.

Breakpoints on lines and subs

Command	Effect
<code>b</code>	Breakpoint on the current line.
<code>b LINE</code>	Breakpoint at LINE in the current file.
<code>b LINE COND</code>	Conditional; break when COND is true.
<code>b FILE:LINE</code>	Cross-file; FILE matches a %INC entry.
<code>b FILE:LINE COND</code>	Conditional cross-file.
<code>b SUB</code>	Break at the first line of SUB (SUB must be compiled).
<code>b SUB COND</code>	Conditional sub breakpoint.
<code>b \$coderef</code>	Break at the sub referenced by \$coderef (no COND supported).
<code>b compile SUB</code>	Break after SUB finishes compiling, before its body runs.
<code>b postpone SUB</code>	Break the first time SUB is called, even if not yet loaded.
<code>b load FILE</code>	Break when FILE is required / do'd.
<code>B LINE</code>	Delete breakpoint at LINE.
<code>B *</code>	Delete all breakpoints.
<code>disable</code>	Disable without removing.
<code>[FILE:]LINE</code>	
<code>enable</code>	Re-enable.
<code>[FILE:]LINE</code>	
<code>L</code>	List breakpoints, actions, and watches.
<code>L b</code>	List breakpoints only.

Conditions are **raw expressions**, not `if (...)`:

```
DB<1> b 237 $count > 10
DB<2> b 42 /unauthorized/i
DB<3> b validate_payload ref($obj) eq 'HASH' && keys %$obj
```

Breakpoints settle only on lines that contain executable code. A blank line, a pure comment, or a continuation of an earlier statement refuses:

```
DB<1> b 3
Line 3 not breakable.
```

A line carries one breakpoint and one action at most; adding a second overwrites the first.

Timing matrix

Where in the program lifecycle a breakpoint fires matters - especially for code in BEGIN blocks or modules that are required lazily.

Goal	Command
Stop at the first runtime statement	Default - the debugger already stops there.
Stop inside a BEGIN { } block	Insert <code>\$DB::single = 1;</code> inside the block.
Stop before a sub's body runs (after it compiles)	<code>b compile SUB</code>
Stop on first call of a sub not yet loaded	<code>b postpone SUB</code>
Stop when a file is required or do'd	<code>b load /full/path/from/%INC</code>
Stop at a specific line	<code>b LINE</code>
Stop at a line only when a condition holds	<code>b LINE COND</code>

`$DB::single = 1;` inside a BEGIN block is the trick for compile-time debugging: writing `b` at the prompt only works after the debugger is running, but BEGIN code runs before the first prompt. Drop the assignment into the block, launch under `-d`, and the debugger stops at the next breakable line inside the block.

Accessing `$DB::single` when not under `-d` autovivifies `DB::` harmlessly - the pattern is safe to commit.

Listing and managing breakpoints

```
DB<1> L
app.pl:
42:   $user = User->load($id);
      break if (1)
91:   dispatch($req);
      break if ($req->method eq 'POST')

DB<2> disable 42
DB<3> enable 42
DB<4> B 42
DB<5> B *
```

`disable` / `enable` toggle a breakpoint without forgetting its condition - useful when you want to skip a hot breakpoint temporarily and re-arm it later.

Actions: run code automatically at a line

An **action** runs Perl code before the specified line executes, without stopping. Use for passive observation:

```
DB<1> a 42 print "entering: user_id=$user_id\n"
DB<2> a 91 warn "POST body:\n" . Dumper($body) if $req->method eq
→ 'POST'
DB<3> L a
app.pl:
42: ...
      action: print "entering: user_id=$user_id\n"
```

(continues on next page)

(continued from previous page)

```
DB<4> A 42          # delete action at line 42
DB<5> A *           # delete all actions
```

Ordering at a breakable line:

1. Breakpoint condition is tested.
2. Trace output is printed (if tracing on).
3. Action runs.
4. Prompt appears (if breakpoint hit or stepping).
5. The line itself executes.

Actions and breakpoints coexist on the same line - one of each maximum.

Watches: break on value change

A watch reports the old and new value when a tracked expression changes, and drops you at the prompt:

```
DB<1> w $global_count
DB<2> c
Watchpoint 0:  $global_count changed:
    old value:  42
    new value:  43
main::tick(app.pl:118):  ++$global_count;
DB<3> W $global_count    # delete this watch
DB<4> W *                # delete all watches
```

Watches are evaluated at every breakable line. Expressions must reference globals or accessible aliases; lexical my variables are not watchable directly. When a lexical matters, use `Tie::Watch` in the code to install a watcher at tie time, or assign via a package-scoped alias.

Watches add per-statement cost proportional to the number of watches and the expense of each expression. Keep them narrow and remove them when done.

Prompt hooks: run code at every stop

`< / >` install Perl code to run automatically before / after every prompt. `{ / { {` do the same for debugger commands:

Command	Effect
<code>< CMD</code>	Perl code before each prompt (= after the line just executed).
<code><< CMD</code>	Append to the pre-prompt list.
<code>< ?</code>	List pre-prompt code.
<code>< *</code>	Delete all pre-prompt code.
<code>> CMD / >> CMD</code>	Perl code after each prompt (= before next line).
<code>{ DBCMD</code>	Debugger command (not Perl) pre-prompt.
<code>{ { DBCMD</code>	Append to the pre-prompt debugger-command list.

Typical use: timestamp every step:

```
DB<1> < print "stop: ", scalar localtime, "\n"
DB<2> > print "start: ", scalar localtime, "\n"
```

Or always show the current line and a watched variable:

```
DB<1> { v
DB<2> { p $important
```

Multi-line: escape the newline with \:

```
DB<1> < for my $f (@important_files) { \
cont:     print "  $f: ", (-s $f) // 'n/a', "\n"; \
cont: }
```

Aliases

Persistent command shortcuts via =:

```
DB<1> = len s/^len(.*)/p length($1)/
DB<2> = quit s/^quit(\s*)/exit/
DB<3> =                               # list all aliases
```

Aliases live in %DB::alias and apply at dispatch time. Define once in .perl5db for across-session persistence.

One-shot breakpoints

c LINE sets a breakpoint at LINE, continues, and clears the breakpoint when it hits:

```
DB<1> c 42
```

Simplest way to say “just run to line 42 and stop there”. No manual b / B pairing.

Breaking on import / first-call lazily

When a sub lives in a module that may not be loaded yet, b SUB fails:

```
DB<1> b MyApp::Worker::dispatch
Subroutine not yet compiled.
```

b postpone waits for the sub to finish compiling:

```
DB<1> b postpone MyApp::Worker::dispatch
DB<2> c
```

For files loaded by require / do, b load breaks on first load:

```
DB<1> b load /usr/share/perl5/MyApp/Worker.pm
```

The filename must match the full path in %INC, not just the module name.

Find out more

- *inspecting-state* - what to look at once a breakpoint fires.
- *tracing* - when you want to watch what happens *without* stopping.

Inspecting state

This chapter covers the debugger’s data-inspection vocabulary: `x`, `V`, `X`, `y`, `m`, `M`, `i`, and the options that control how deeply aggregate values are dumped. After this chapter you can ask any running program “what does this variable, this package, this object actually look like right now?”

Readers arrive here knowing `p EXPR` prints a scalar. The rest of the inspection commands earn their keep the moment the answer “what is this?” requires more than a single value.

`x`: dump structures

`x EXPR` pretty-prints in list context, recursing into references:

```
DB<1> x \%config
0  HASH(0x5651f9a82358)
   'debug' => 1
   'dsn' => 'dbi:SQLite::memory:'
   'timeout' => 30
   'users' => ARRAY(0x5651f9b28a10)
       0  'alice'
       1  'bob'
       2  'carol'
```

Always pass a **reference** for aggregates. `x %h` dumps the flat key/value list, which hides the structure. `x \%h` does what you want.

```
DB<2> x \@list
0  ARRAY(0x5651f9b28a10)
   0  'alice'
   1  'bob'
   2  'carol'

DB<3> x $obj
0  MY::Class=HASH(0x5651fa01c2b0)
   'id' => 42
   'prefs' => HASH(0x5651fa01c330)
   'theme' => 'dark'
```

For deep or cyclic structures, cap the recursion:

```
DB<4> x 2 \%tree      # recurse 2 levels deep
DB<5> x 1 $obj        # one level
```

If the first token of the expression is numeric, parenthesise:

```
DB<6> x (3 + $offset)      # without parens, 3 is mis-parsed as _  
↪ depth
```

V and X: package variables

V [PKG [VARS]] dumps the globals of a package:

```
DB<1> V main  
$counter = 42  
@ARGV = (  
    0  '--verbose'  
    1  'input.txt'  
)  
%ENV = (  
    'PATH' => '/usr/bin:/bin'  
    ...  
)
```

With a second argument, filter variables:

```
DB<2> V MyApp $config $last_user      # only these  
DB<3> V MyApp ~^config                # regex: names matching /^  
↪ config/  
DB<4> V MyApp !^_                     # regex: names NOT matching _  
↪ /^_/
```

X is V for the current package:

```
DB<5> X                                # dump current package's globals  
DB<6> X $foo @bar                     # filtered
```

V / X do **not** see lexicals (my variables). For those, use y.

y: lexicals in enclosing scope

y LEVEL [VARS] reads lexical my variables via PadWalker:

```
DB<1> y 0                                # lexicals of the current sub  
my $req = HTTP::Request=HASH(0x...)  
my $user_id = 42  
my @errors = ()  
  
DB<2> y 1                                # one frame up (the caller's lexicals)  
my $request_count = 118  
my $dispatcher = MyApp::Dispatcher=HASH(0x...)  
  
DB<3> y 0 $req $user_id                # only those
```

y requires PadWalker version 0.08 or later. It works only inside a sub (the outer script body has no pad to walk).

m: methods of an object or class

m EXPR lists methods callable on an object or class:

```
DB<1> m $user
via UNIVERSAL: isa can DOES VERSION DESTROY
via MyApp::Model: save load delete update
via MyApp::Role::Auditable: audit_log record_event

DB<2> m 'MyApp::Worker'
via UNIVERSAL: ...
via MyApp::Worker: dispatch retry cancel
```

Groups methods by the package that provides them - useful for spotting which role or superclass a method comes from.

M and i: modules and inheritance

M lists loaded modules with versions:

```
DB<1> M
'DBI' (version 1.643)
'JSON::PP' (version 4.16)
'MyApp::Model' (version undef)
...
```

i CLASS or i \$obj prints the inheritance tree:

```
DB<2> i $user
MyApp::User 0.12
  MyApp::Model 0.42
    Moo::Object ?
```

Useful for “where is this method actually coming from?” when method resolution crosses several roles.

S: subroutine names

S [[!]PAT] lists sub names matching (or not matching) a pattern:

```
DB<1> S ^MyApp::Worker::
MyApp::Worker::cancel
MyApp::Worker::dispatch
MyApp::Worker::retry

DB<2> S !^(main|DB|UNIVERSAL)::      # everything except these
```

S alone lists everything - long.

p vs x: which to use when

Want	Use
A single scalar's value, plain	p \$x
A joined / formatted expression	p join ' ', sort keys %h
A hash or array	x \%h, x \@a
An object's fields	x \$obj
Only the top level	x 1 \$deep_tree
Package variables of Pkg	V Pkg
Lexicals of current sub	y 0
Method list of \$obj	m \$obj
@ISA chain of \$obj	i \$obj

Special case: p is a flat print. For a reference, p \$ref prints HASH(0x...). Use x \$ref for the contents.

Depth and compactness options

Dump output is shaped by options set via the o command. The relevant settings for inspection:

Option	Effect
dumpDepth	Recursion depth (negative = unlimited). Default 4.
arrayDepth	First N array elements only; empty = all.
hashDepth	Same, for hashes.
compactDump	Short arrays on one line.
veryCompact	Tighter yet.
globPrint	Dump glob contents.
DumpReused	Fully expand repeated references (beware cycles).
quote	" or ' or auto (default).
undefPrint	How undef renders.
bareStringify	Print overload-stringified form.

Set an option in a session:

```
DB<1> o dumpDepth=6
DB<2> o compactDump=1
DB<3> o arrayDepth=20
DB<4> o compactDump?           # query current value
```

Set options at startup via PERLDB_OPTS or in .perlddb:

```
PERLDB_OPTS="dumpDepth=6 compactDump=1" perl -d script.pl
```

Inspecting SV internals

`Devel::Peek` exposes the scalar’s internal storage - type flags, refcount, PV/NV/IV layout, magic chain - when the question is “what kind of scalar is this really?”:

```
DB<1> use Devel::Peek; Dump($sv)
```

Primarily for investigating tied variables, overloaded objects, or performance anomalies where the answer depends on whether a scalar is currently stored as an integer, a string, or both.

Note: `Devel::Peek` in `ppperl` is a compat layer - it reports a `ppperl`-specific Sv layout rather than `perl5`’s SV layout. Fields like `REFCNT`, `FLAGS`, `PV`, `IV`, `NV` have equivalents; the exact output differs. See the [ppperl-architecture guide](#) for layout details.

Evaluating at the prompt

Anything unrecognised as a command is evaluated as Perl. Use it to drill deeper than a single command:

```
DB<1> p keys %{ $obj->{prefs} }
theme,locale,tz

DB<2> x [ grep { !defined $_->{parent} } @nodes ]
DB<3> x { map { $_->id => $_->name } @users }
```

For one-off assertions without stopping:

```
DB<1> die "unexpected" if ref $obj ne 'MyApp::User'
```

Program I/O vs debugger I/O

The debugger’s own input and output go through `$DB::IN` and `$DB::OUT`, which are opened to `/dev/tty` directly. The program can redirect `STDIN`/`STDOUT`/`STDERR` freely; the debugger prompt still shows up. Conversely, `p EXPR` output does not contaminate a program’s captured stdout.

Find out more

- [breakpoints](#) - stop where you can inspect; this chapter is about what to look at once stopped.
- [tracing](#) - watch values change over time without stopping at every step.

Tracing: watching execution flow

This chapter covers traces that record what a program did without stopping it: the debugger’s `t` command and trace options, the `Devel::Trace` module family, and the regex-engine tracer. Use tracing when the bug is “which code path actually ran?” rather than “what value does X have at line N?”.

Readers coming in know how to set breakpoints and inspect state. A trace is the complement - “follow every step and tell me about it” rather than “stop at one interesting place”.

t: trace inside the debugger

t toggles per-statement tracing. Every breakable line prints a location marker before executing:

```
DB<1> t
Trace = on

DB<2> c
main::(app.pl:5):      my $key = 'welcome';
main::(app.pl:6):      my %data = (
main::handle_request(app.pl:42):  my $user = load_user($req);
main::load_user(app.pl:87):      my ($req) = @_;
main::load_user(app.pl:88):      return unless $req;
...
```

Restricting the depth:

```
DB<3> t 2           # toggle with maximum sub-depth 2
DB<4> t 0 my_work() # trace one expression only
```

The frame option controls what the trace prints at sub calls and returns:

Bit	Flag	Effect
1	enter	Print on sub entry.
2	exit	Print on sub exit.
4	ctx+args	Include call context and arguments.
8	tied/ref	Mark tied-variable and reference sub calls.
16	return	Print the return value.

Set frame via o:

```
DB<5> o frame=15      # enter + exit + ctx/args + tied/ref
```

Common combinations:

- frame=2: just entry/exit traces, one line per call - the call-tree view.
- frame=6: add caller context and argument dump - full trace.
- frame=30: like frame=6 plus return values.

Trace output goes to the debugger’s output stream. Redirect it to a file so it does not drown the interactive session:

```
DB<6> o LineInfo=trace.out
```

Read the file in another terminal:


```
tail -f trace.out
```

Non-interactive tracing: NonStop + AutoTrace

For “record a full execution trace with no interaction”, combine `NonStop=1` (no prompts) and `AutoTrace=1` (tracing on from start):

```
PERLDB_OPTS="NonStop=1 AutoTrace=1 frame=2 LineInfo=trace.out" \
perl -d script.pl
```

The program runs straight through; `trace.out` captures every breakable line and every sub entry. Ideal for CI-mode failure capture, headless servers, or any environment where a TTY prompt is unavailable.

Add `inhibit_exit=0` to let the program fall off the end without the “Debugged program terminated” prompt.

Devel::Trace: every line, without the debugger

`Devel::Trace` traces every line of every file to `STDERR`. No command-line interaction. No breakpoints. Simple and loud:

```
perl -d:Trace script.pl 2> trace.out
```

```
>> app.pl:4: my $key = 'welcome';
>> app.pl:5: my %data = (
>> app.pl:11: print "$data{$key}\n";
```

Run it once, `grep` the output for the puzzle. Output size is proportional to program run length - use on a minimal reproducer, not the whole test suite.

Variants:

- `Devel::DumpTrace` - same plus per-line variable values.
- `Devel::TraceUse` - “which modules were loaded, in what order, by whom”:

```
perl -d:TraceUse script.pl
```

- `Devel::TraceINC` - narrower: just the require chain with timing.
- `Devel::Trace::Subs` - call-flow and stack-trace generator for “which subs actually ran”.

Tracing a specific region

The `$DB::trace` variable toggles tracing from Perl code:

```
$DB::trace = 1;
interesting_region();
$DB::trace = 0;
```

Assigning to `$DB::trace` when not under `-d` autovivifies the variable harmlessly - the code is safe to commit and runs as a no-op outside the debugger.

Same pattern for single-stepping a region:

```
$DB::single = 1;           # break here when next statement runs under
↪-d
```

Regex tracing: use `re 'debug'`

The regex engine has its own tracer. Enable lexically inside a region, or on the command line:

```
{
    use re 'debug';
    $str =~ /(\w+)\s+(\w+)/;
}
```

```
perl -Mre=debug -e '"hello world" =~ /(\w+)\s+(\w+)/'
```

Compile-phase output shows what the optimiser did:

- Compiling REX 'PATTERN' - the source pattern.
- size N - internal node-table size.
- anchored 'SUB' at K / floating 'SUB' at K..LIMIT - which substring the engine will look for as a pre-filter.
- stclass TYPE - the starting-character class.
- minlen N - no match possible on input shorter than N.
- Node listing - the compiled program.

Runtime output shows each step:

```
POS <PRE-MATCH> <POST-MATCH> | NODE_NUM: OPNAME
```

Indentation reflects backtracking depth; failed... and failed, try continuation... mark backtracks; Match successful! / Match failed. close the trace.

If no runtime output appears at all, the optimiser decided the match without entering the engine - either minlen pre-filtered or the pattern was folded to a constant test.

pperl's regex engine is a native reimplementaion, not a port of perl5's. The trace is functionally equivalent but the exact node names and field formats are not byte-identical to upstream. Parsing the trace programmatically (few tools do) will need adjustment.

Partial switches narrow the output:

```
use re 'debugcolor';           # ANSI colour
use re debug => 'EXECUTE';     # runtime only, no compile
↪output
```

Devel::TraceUse for module-load bugs

A program that runs slowly at startup, or a “why is this module loaded at all?”, is almost always a module-graph question. `Devel::TraceUse` prints the full use/require tree:

```
perl -d:TraceUse script.pl
Modules used from script.pl:
  1. strict 1.12, script.pl line 2 [main]
  2. warnings 1.58, script.pl line 2 [main]
  3. DBI 1.643, script.pl line 5 [main]
    4. Carp 1.52, DBI line 13 [DBI]
    5. DynaLoader 1.47, DBI line 14 [DBI]
  ...
```

Indentation shows the dependency chain. Cross-reference with the line number in the parent file to see where each load happens.

Combining traces with the debugger

Trace to a file, step interactively:

```
PERLDB_OPTS="frame=6 LineInfo=trace.out" perl -d app.pl
```

The debugger prompt appears normally; `trace.out` accumulates the call-tree as you step and continue. Useful when a call chain is too deep to follow by stepping but you still want to stop at specific points to inspect.

Find out more

- *breakpoints* - stopping at a specific point rather than recording every step.
- *inspecting-state* - the commands that answer “what is this value?”.

IDE and DAP integration

This chapter covers editor-driven debugging: `Perl::LanguageServer` in VS Code and its counterparts in Neovim / Emacs / IntelliJ, remote-over-SSH debugging, and container-mode debugging via `kubectl` or `docker`.

Readers who reach for an IDE rather than a terminal prompt will find here the minimum viable setup, the config keys that matter, and the known fragility points (Coro, path mapping, attach vs launch).

The landscape

Two wire protocols, both with Perl backends:

Protocol	Perl backend	Status
DAP	<code>Perl::LanguageServer</code>	Active.
DBGp	<code>Devel::Debug::DBGp</code>	Stagnant since 2018.

For new setups, use DAP via `Perl::LanguageServer`. DBGp survives only where an existing client (vim vdebug, pugdebug) is already in use.

`Perl::LanguageServer` bundles an LSP (code intelligence, diagnostics, go-to-definition) and a DAP (debugger) adapter in one process. The LSP half works on any Perl file in the editor; the DAP half runs only during a debug session.

Minimal VS Code setup

CPAN side - install into the Perl the debuggee will run under:

```
cpanm Perl::LanguageServer
```

Editor side - install the `richterger.perl` extension:

```
Ctrl-P → ext install richterger.perl
```

`.vscode/launch.json`:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "perl",
      "request": "launch",
      "name": "Debug current file",
      "program": "${workspaceFolder}/${relativeFile}",
      "stopOnEntry": true,
      "args": [],
      "cwd": "${workspaceFolder}",
      "env": {},
      "reloadModules": true
    }
  ]
}
```

Open a `.pl` or `.pm`, set breakpoints in the gutter, press F5. The Debug Console accepts any Perl expression, evaluated in the current stack frame.

`.vscode/settings.json` for `@INC` and interpreter selection:

```
{
  "perl.perlInc": [
    "${workspaceFolder}/lib",
    "${workspaceFolder}/local/lib/perl5"
  ],
  "perl.perlCmd": "/usr/bin/perl"
}
```

Capabilities

Perl::LanguageServer supports the DAP features that matter:

- Launch, pause, step in / over / out, return.
- Conditional and unconditional breakpoints, added at any time.
- Variable inspection across stack frames.
- **Set variable** - change a value mid-session from the editor.
- Watch expressions.
- Evaluate-in-context at the Debug Console.
- Module hot-reload via `reloadModules: true`.

Not supported:

- **Attach to a running process.** `request: "attach"` is not implemented; only `"launch"` (the LS starts the debuggee itself).

Remote debugging over SSH

Edit locally; debug on a remote host. `settings.json`:

```
{
  "perl.sshCmd": "ssh",
  "perl.sshAddr": "deploy@host.example",
  "perl.sshUser": "deploy",
  "perl.sshWorkspaceRoot": "/srv/app",
  "perl.pathMap": [
    ["/srv/app", "${workspaceFolder}"]
  ]
}
```

Perl::LanguageServer runs on the remote host; the editor stays local. `pathMap` translates filenames the debuggee reports (`/srv/app/lib/X.pm`) to local editor paths (`${workspaceFolder}/lib/X.pm`).

Breakpoints that silently fail to bind are almost always a `pathMap` mismatch.

Container-mode debugging

`launch.json` accepts container keys that invoke `docker`, `podman`, `docker-compose`, or `kubectl`:

Key	Values
<code>containerCmd</code>	"docker", "docker-compose", "podman", "kubectl"
<code>containerMode</code>	"run" (fresh container) or "exec" (existing)
<code>containerName</code>	Image (for run) or container name (for exec)
<code>containerArgs</code>	Extra args passed to the container runtime.
<code>pathMap</code>	[["/in/container", "/on/host"], ...]

Attach to an already-running Docker container:

```
{
  "type": "perl",
  "request": "launch",
  "name": "Debug in container",
  "program": "/app/bin/worker.pl",
  "containerCmd": "docker",
  "containerMode": "exec",
  "containerName": "my-running-worker",
  "pathMap": ["/app", "${workspaceFolder}"]
}
```

Kubernetes pod:

```
{
  "type": "perl",
  "request": "launch",
  "name": "Debug in pod",
  "program": "/app/bin/worker.pl",
  "containerCmd": "kubectl",
  "containerMode": "exec",
  "containerName": "my-pod-abc123",
  "containerArgs": ["-n", "staging", "--container", "app"],
  "pathMap": ["/app", "${workspaceFolder}"]
}
```

Constraints:

- **Perl::LanguageServer must be installed inside the image.** Bake it into the Dockerfile or build a dev-image variant.
- pathMap is the single biggest failure mode. Breakpoints that never bind mean the debuggee reports a path that does not translate.

Other editors

- **Neovim** - use nvim-dap with Perl::LanguageServer as the adapter. The adapter spec is a short Lua snippet that points at perl -MPerl::LanguageServer -e '...'
- **Emacs** - dap-mode with the same adapter.
- **IntelliJ** - Devel::Camelcadedb + the JetBrains Perl plugin. Not DAP; proprietary protocol. Viable but outside the Perl::LanguageServer ecosystem.
- **Sublime Text** - DAP clients exist via plugins; use Perl::LanguageServer as the adapter.

Special-mode launches

`launch.json` keys for non-default invocation:

Key	Effect
<code>useTaintForDebug</code>	Injects <code>-T</code> into the debuggee’s perl invocation.
<code>sudoUser</code>	Re-execs the debuggee as that user (needs passwordless sudo).
<code>reloadModules</code>	Enables module hot-reload during the session.
<code>stopOnEntry</code>	Break at the first statement.

Gotchas

- **BEGIN-time errors** surface as DAP “output” events, not as stopped-at-breakpoint events. Use `stopOnEntry: true` to get a stop at all if the script dies in BEGIN.
- **Application `$SIG{__DIE__}`** can swallow exceptions before the debugger’s handler runs. Wrap the app’s handler in an `if ($^S)` guard (see [exceptions](#)).
- **Watch expressions re-evaluate on every step.** Avoid expensive or side-effect expressions.
- **No source maps.** Perl has no concept equivalent to a JavaScript source map. Source filters (`Smart::Comments`) can shift reported line numbers by one; breakpoints in filtered files may land on the wrong line.
- **Future::AsyncAwait stepping.** Works for simple cases; stepping through complex await chains is unreliable. Drop to `perl -d` for deep async investigation, or use logging.
- **Windows native** - `Perl::LanguageServer`’s stdin handling is broken on Windows. WSL works.
- **Coro.** `Perl::LanguageServer` relies on Coro internally; the most-cited fragility point is Coro interacting badly with unusual build configurations. If the server refuses to start, check Coro’s install log first.

Find out more

- [interactive-debugger](#) - `perl -d` directly, when the IDE stack is overkill or unavailable.

4.7 The Definitive Perl One-Liner Guide

A Perl one-liner is a complete program delivered as an argument to `ppperl -e` (or `-E`), run directly from the shell. It occupies the same niche as `awk`, `sed`, and ad-hoc shell pipelines, with a richer regex engine, a full standard library, and the ability to grow into a real script the moment one line stops being enough.

This guide is for a Linux shell user who already composes `grep/sed/awk/xargs` pipelines and has written a small Perl script or two. It teaches the command-line surface end

to end: the switches, a progressive recipe catalogue, regex-driven transformations, numeric work, how to wrap recipes into reusable shell aliases, and the traps that cost time.

The binary is `pperl`. Every example in this guide is copy-pasteable into a Linux shell. `pperl` is a Rust reimplementation of Perl 5; its command-line interface matches Perl 5.42.

4.7.1 Which chapter do I want?

You want to ...	Start here
Know what each switch (<code>-n</code> , <code>-p</code> , <code>-a</code> , ...) does	<i>switches</i>
Browse recipes from trivial substitutions up to data reductions	<i>progression</i>
Match, extract, or rewrite text with regexps	<i>regex-recipes</i>
Sum, average, generate primes, convert IPs, do date arithmetic	<i>numeric</i>
Turn a one-liner into a reusable shell function or alias	<i>aliases</i>
Debug quoting, encoding, in-place edits, and record-separator surprises	<i>gotchas</i>

4.7.2 The three layers of this guide

- **Orientation** - this page.
- **Working programmer** - the six chapters above. Each stands on its own; skim the ones relevant to your current problem.
- **Deep reference** - the compact switch/variable table at the end of [*switches*](#).

4.7.3 Two examples before you start

A `sed/awk` user reading this guide wants to see what a one-liner buys them. Two demonstrations.

Sum the numbers in the last column of a whitespace-separated file

```
pperl -lane '$s += $F[-1]; END { print $s }' data.txt
```

`-l` chomps input newlines and appends one on output. `-a` splits each line into `@F`. `-n` wraps the program in an implicit read loop. `$F[-1]` is the last column. `END { ... }` runs once after input is exhausted. No temporary file, no `awk 'BEGIN{...}END{...}'`, no boilerplate.

Extract every unique IPv4 address from a log, newest first

```
pperl -lne 'print for /\b(?:\d{1,3}\.){3}\d{1,3}\b/g' access.log \
| awk '!seen[$0]++'
```

The one-liner handles the hard part (finding all matches per line, not just the first), the `awk` keeps the first sighting of each. Written as two short tools, each doing one job - that is the working mode of this guide.

4.7.4 Read in this order on a first pass

1. *switches* - the vocabulary.
2. *progression* - the recipes, ordered by difficulty.
3. *regex-recipes*, *numeric* - applied pieces.
4. *aliases* - lifting what you use daily into shell functions.
5. *gotchas* - the traps, once you are running into them.

On later passes, go directly to whichever chapter names the problem you are holding.

4.7.5 Bibliography

This guide is not derived from any single book. It was researched the way a technical writer would research the topic - perldoc, CPAN module documentation, conference talks, blog posts, mailing-list archives, and the books listed below - and then distilled in several passes: removing references to tools and versions nobody runs anymore, collapsing the same advice repeated by different authors into one statement, and merging multiple partial treatments of one topic into a single complete treatment. The result covers more ground than any single source we consulted, because no single source covers the whole field. It is also more densely cross-linked than any of them: when the regex chapter touches something the regex guide already explains in depth, that is one click away - not a chapter you have to find by yourself in another volume.

The two books below are the closest published companions in print form. Buy them if the one-liner idiom is a daily tool for you and you want a printed reference on the shelf.

- Peteris Krumins, *Perl One-Liners: 130 Programs That Get Things Done*, No Starch Press, 2014.
- Sundeep Agarwal, *Perl One-Liners Guide* (learnbyexample), 2023.

The switches

Every one-liner is a combination of a program (passed to `-e` or `-E`) and a handful of switches that wrap implicit behaviour around it. This chapter covers the switches you will reach for. Each one earns its keep in the progression, regex, and numeric chapters that follow.

Readers coming from `awk` or `sed` already know what per-line iteration, field splitting, and in-place editing look like; the question is which `ppperl` flag turns each on.

`-e` - run a program given on the command line

```
ppperl -e 'print "hello\n"'
```

`-e` takes the Perl source as its argument. Multiple `-e` chunks concatenate; they behave as one program with semicolons between them.

```
ppperl -e 'my $x = 41;' -e 'print $x + 1, "\n"' # 42
```

Use single quotes around the program so the shell leaves `$_`, `$1`, `\n`, and backticks alone. See *gotchas* for what goes wrong with double quotes.

-E - -e plus the modern feature set

-E is -e with all optional features enabled: `say`, `state`, `fc`, the `//` defined-or operator, unicode string semantics under use feature `'unicode_strings'`, and more.

```
ppperl -E 'say "hello"' # instead of print "hello\n"
ppperl -E 'say "pi = " . (atan2(1,1)*4)'
```

Reach for -E when writing new one-liners. Reach for -e only when copy-paste compatibility with very old scripts matters.

-n - wrap an implicit read loop around the program

-n turns the program into the body of a loop over every line of every input file (or STDIN if none are given). The loop is exactly:

```
while (<>) {
    # your program
}
```

`$_` holds each input line, including its trailing newline. Nothing is printed automatically - you decide what reaches STDOUT.

```
# Print lines containing "error"
ppperl -ne 'print if /error/' app.log
```

Equivalent long form, shown once so the mechanical expansion is clear:

```
while (<>) {
    print if /error/;
}
```

Every subsequent -n recipe in this guide relies on that expansion - assume it without re-stating.

`BEGIN { ... }` runs before the loop, `END { ... }` runs after. Both are ordinary Perl blocks and compose with -n cleanly.

```
# Count lines matching a pattern
ppperl -ne '$n++ if /WARN/; END { print "$n\n" }' app.log
```

-p - like -n, but print \$_ after each iteration

```
ppperl -pe 's/foo/bar/g' input.txt
```

Mechanical expansion:

```
while (<>) {
    # your program
} continue {
```

(continues on next page)

(continued from previous page)

```
print or die "-p destination: $!";
}
```

Reach for `-p` when the output is the (possibly modified) input. Reach for `-n` when the output is a summary, a filtered subset, or nothing at all.

-l - line-end handling

`-l` does two things at once:

1. On input, `chomp` each line as it is read under `-n` / `-p`. `$_` no longer ends in `"\n"`.
2. On output, set `$\` (the output record separator) so every `print` appends `"\n"`.

```
# Print just the first field of each tab-separated line
ppperl -F'\t' -lane 'print $F[0]' table.tsv
```

Without `-l`, you would either write `print "$F[0]\n"` (the newline moves into the program) or produce glued output.

`-l` accepts an optional octal argument that sets `$\` to a different character: `-l012` keeps newline (the default), `-l0` sets `$\` to NUL, and so on. The common use is the no-argument form.

-a - autosplit into @F

Combined with `-n` or `-p`, `-a` splits `$_` on whitespace and puts the fields into `@F`. The default split is `split(' ', $_)` - it trims leading whitespace and treats any run of whitespace as one separator.

```
# Third column of every line
ppperl -lane 'print $F[2]' data.txt

# Reverse column order
ppperl -lane 'print "@{[reverse @F]}"' data.txt
```

`@F` is zero-indexed. `$F[-1]` is the last field. `scalar @F` is the field count.

-F - change the split pattern

`-F` sets the delimiter used by `-a`. The argument is a regex; a literal comma is `-F,`, a colon is `-F:`, a tab is `-F'\t'`.

```
# Second field of a colon-separated file
ppperl -F: -lane 'print $F[1]' /etc/passwd

# Split on any run of commas or semicolons
ppperl -F'[;,]+' -lane 'print scalar @F' mixed.csv
```

`-F` implies nothing about `-a`; you still need `-a`. The common idiom `-F: -lane` is colon-split, chomped, with `@F` available.

Quoting a tab requires a literal tab in the shell or the regex form:

```
ppperl -F'\t' -lane 'print $F[0]' data.tsv      # regex form, always s
↪safe
ppperl -F'      ' -lane 'print $F[0]' data.tsv  # literal tab - s
↪fragile
```

-i - edit files in place

-i rewrites the file pperl is reading from, replacing its contents with whatever -p prints. The whole file's output goes to a temporary file; when the loop finishes, that temporary replaces the original.

```
# Replace CRLF with LF in every .txt under cwd (no backup)
ppperl -i -pe 's/\r\n/\n/g' *.txt

# Same, keeping a .bak copy beside each original
ppperl -i.bak -pe 's/\r\n/\n/g' *.txt
```

Always use -i.bak (or another suffix) until you have verified the edit against a representative file. A bug in the program with bare -i destroys the input. See *gotchas*.

-i has no effect without -p (or explicit prints under -n): if nothing is printed, the replacement file is empty.

-M and -m - load modules before the program runs

-Mmodule is equivalent to use module; at the top of the program. -Mmodule=sym1, sym2 is use module qw(sym1 sym2);.

```
ppperl -MList::Util=sum -lane 'print sum @F' data.txt
ppperl -MPOSIX=strftime -le 'print strftime("%F", localtime)'
ppperl -MData::Dumper -e 'print Dumper({a=>1, b=>[2,3]})'
```

Lower-case -m is no strict ... rather than use ...; it is rarely needed on the command line.

Modules most common in one-liners: List::Util (sum/min/max/uniq/any), POSIX (strftime, mktime, fmod), Data::Dumper (inspect structures), JSON::PP (parse/emit JSON), Text::CSV (quoted CSV), MIME::Base64, Digest::MD5, Digest::SHA, Socket.

-0 - set the input record separator

Perl normally reads a line at a time. -0NNN sets \$/ to the character with the given octal code, changing what “one record” means for <>, <STDIN>, and the -n / -p loop.

Common forms:

Form	What \$/ becomes	Effect
-0	"\0" (NUL)	Read NUL-delimited records (matches <code>find -print0</code>).
-0777	undef	Slurp the whole file as one record.
-00	" "	Paragraph mode: blank lines separate records.
-0NNN	<code>chr(0NNN)</code>	Any single octal byte.

```
# Count files from find -print0
find . -type f -print0 | pperl -0 -ne '$n++; END { print "$n\n" }'

# Slurp, then run a cross-line regex
ppperl -0777 -ne 'print "yes\n" if /BEGIN.*?END/s' text.txt

# Print every paragraph containing "TODO"
ppperl -00 -ne 'print if /TODO/' notes.txt
```

Paragraph mode (`-00`) interacts with `-l`: without `-l`, the blank separator stays attached to each record; with `-l`, it is chomped.

-C - Unicode on stdio and/or @ARGV

`-C` enables Unicode handling on specified streams. The argument is either a number (bitmask) or a string of letters.

```
ppperl -CSD -ne 'print' input.txt      # stdin, stdout, stderr all
↳ UTF-8
ppperl -CA -e 'print $ARGV[0]' äöü    # @ARGV treated as UTF-8
ppperl -CSA -nE 'say length'          # S + A
```

Letter codes: I = stdin, O = stdout, E = stderr, S = I+O+E, A = @ARGV, D = set default I/O layers (includes file opens).

For a fully UTF-8 session, `-CSD`. For one-off processing of named UTF-8 files, `-CSAD`. See *gotchas* for the symptoms that tell you `-C` is missing.

Reference table

Every switch, its mechanical expansion, and the chapter where it first shows up in a recipe.

Switch	Expands to	First used in
-e CODE	Run CODE as the program	progression
-E CODE	-e plus all current features enabled	progression
-n	while (<>) { CODE }	progression
-p	while (<>) { CODE } continue { print }	progression
-l	chomp input under -n/-p; set \$\ = "\n" for output	progression
-a	Auto-split \$_ into @F	progression
-F PAT	Change -a split pattern to regex PAT	progression
-i[EXT]	Edit in place; with EXT, keep backup original + EXT	progression
-M MOD	use MOD; before program	numeric
-M MOD=X	use MOD qw(X);	numeric
-0	\$/ = "\0" - NUL-delimited records	progression
-00	\$/ = "" - paragraph mode	progression
-0777	\$/ = undef - slurp entire file	progression
-C	Enable Unicode on stdio, @ARGV, and/or default layers	gotchas

Special variables the switches set up

Short catalogue; full definitions live in [perlvar](#).

Variable	Role under -n/-p
\$_	The current input line (chomped if -l, raw otherwise).
\$.	Current input line number (not reset between files by default).
\$/	Input record separator. -0 family changes it.
\$_	Output record separator appended by print. -l sets it to "\n".
\$,	Output field separator inserted between print's arguments.
\$"	Separator between array elements when interpolated in "@array".
@F	Auto-split fields under -a.
@ARGV	Remaining command-line arguments; filenames are consumed as processed.
\$ARGV	Name of the file currently being read (" - " means STDIN).

Find out more

- [progression](#) applies every switch in recipes of increasing complexity.
- [perlvar](#) lists every special variable pperl defines, not just the ones the switches touch.
- [perlop](#) covers qr//, the flip-flop operator, and the diamond operator used by the -n / -p loops.

Progressive recipes

This chapter works from trivial substitutions up through reductions, extraction, and structured output. Each section reuses the switches introduced in the previous one, so a reader going straight through builds up the idiom vocabulary without ever meeting a new flag mid- recipe.

Readers coming from awk/sed should recognise the first sections; the interesting material starts around *Reductions across lines* and *Context-windowed filters*.

Every recipe is copy-pasteable. When a sample input matters, it is shown inline so the expected output is unambiguous.

The baseline: -e and -E

The shortest one-liner. No I/O loop, no field splitting - just a program that runs once.

```
ppperl -e 'print 2 ** 32, "\n"'           # 4294967296
ppperl -E 'say 2 ** 32'                   # same, with -E/
↪say
ppperl -E 'say for 1 .. 5'                 # 1\n2\n3\n4\n5
```

The , in print separates arguments; the output field separator \$, (default empty) goes between them.

Filtering: -n + print if

-n wraps the program in a read loop over every line of every input file (or STDIN). Nothing is printed unless the program says so.

```
# Sample input
$ printf 'gate\napple\nwhat\nkite\n' > words.txt

ppperl -ne 'print if /at/' words.txt       # gate, what
ppperl -ne 'print unless /e/' words.txt    # what
ppperl -ne 'print if /^[aeiou]/' words.txt # apple
```

The regex match is against \$_ by default; /at/ is short for \$_ =~ m/at/.

Multiple conditions

```
# Lines containing both "ERROR" and "timeout"
ppperl -ne 'print if /ERROR/ && /timeout/' app.log

# Lines containing neither
ppperl -ne 'print if !/ERROR/ && !/timeout/' app.log

# Events A, B, C appearing in order on one line
ppperl -ne 'print if /A.*B.*C/' events.log
```

By line number

```
pperl -ne 'print if $. == 1'           # head -n 1
ppperl -ne 'print if $. <= 10'         # head -n 10
ppperl -ne 'print if 17 .. 30'         # lines 17-30
↪(range op)
ppperl -ne 'print if /START/ .. /END/'  # regex range,
↪inclusive
ppperl -ne 'print unless $. % 2'       # even lines
```

The range operator `..` in scalar context is a flip-flop: compared against `$.` when both endpoints are integers, against `$_` when they are patterns. See [perlop](#).

By length

Use `-l` so the counted length excludes the trailing newline.

```
pperl -lne 'print if length >= 80' src.c      # long lines
ppperl -lne 'print if length() < 5' words.txt  # short lines
```

Bare length with no argument applies to `$_`.

Substitution: `-p + s///`

`-p` is `-n` plus an implicit `print` after each iteration. Use it when the output is the (possibly modified) input.

```
# Replace every "foo" with "bar"
ppperl -pe 's/foo/bar/g' input.txt

# First occurrence per line only
ppperl -pe 's/foo/bar/' input.txt

# Conditional: replace only on lines that also match BAZ
ppperl -pe 's/foo/bar/g if /BAZ/' input.txt
```

Case transformations

```
pperl -nle 'print uc'           # to uppercase
ppperl -nle 'print lc'           # to lowercase
ppperl -nle 'print ucfirst lc'   # Sentence case
ppperl -ple 's/(\w+)/\u$1/g'     # Title Case Per
↪Word
```

`uc`, `lc`, and `ucfirst` with no argument operate on `$_`.

Whitespace normalisation

```

ppperl -ple 's/^\s+//'          # trim left
ppperl -ple 's/\s+$//'          # trim right
ppperl -ple 's/^\s+|\s+$//g'    # trim both
ppperl -ple 's/\s+/ /g'         # collapse runs
ppperl -ne 'print if /\S/'      # drop blank
↪ lines

```

Line-ending conversion

```

ppperl -pe 's/\r\n/\n/g' win.txt      # DOS → UNIX
ppperl -pe 's/(?<!\r)\n/\r\n/g' unix.txt # UNIX → DOS

```

The lookbehind (`(?<!\r)`) avoids producing `\r\r\n` when the input is already DOS-encoded. See *regex-recipes* for lookarounds in depth.

Field-oriented work: -a and -F

`-a` auto-splits each line into `@F`. `-F` changes the delimiter.

```

$ cat table.txt
brown bread mat hair 42
blue cake mug shirt -7
yellow banana window shoes 3.14

# Second field
ppperl -lane 'print $F[1]' table.txt
# bread, cake, banana

# Last field
ppperl -lane 'print $F[-1]' table.txt
# 42, -7, 3.14

# Count fields per line
ppperl -lane 'print scalar @F' table.txt
# 5, 5, 5

# Lines whose last field is negative
ppperl -lane 'print if $F[-1] < 0' table.txt
# blue cake mug shirt -7

```

Custom delimiter

```

# /etc/passwd: print username (field 1) and shell (last field)
ppperl -F: -lane 'print "$F[0] $F[-1]"' /etc/passwd

# Tab-separated
ppperl -F'\t' -lane 'print $F[2]' metrics.tsv

```

(continues on next page)

(continued from previous page)

```
# Comma or semicolon, one or more
ppperl -F'[;,]+' -lane 'print scalar @F' mixed.csv
```

Proper CSV (quoted fields with embedded commas) is not a job for `-F`. Use `-MText::CSV` - see *numeric* for the pattern.

Rejoining fields

Interpolating `@F` inside double quotes uses `$` (default: one space) as the separator.

```
# Reverse field order, space-separated
ppperl -lane 'print "@{[reverse @F]}"' table.txt

# Reverse field order, colon-separated
ppperl -F: -lane 'local $" = ":"; print "@{[reverse @F]}"' /etc/
↪passwd
```

`@{[ EXPR ]}` is the array-ref-then-deref idiom that lets arbitrary expressions be interpolated into a double-quoted string.

Reductions across lines

The move from per-line filters to cross-line computation.

Counts

```
# wc -l
ppperl -lne 'END { print $. }' big.txt

# grep -c pattern
ppperl -lne '$n++ if /pattern/; END { print $n + 0 }' file

# Count blank lines
ppperl -lne '$n++ if /^$/; END { print $n + 0 }' file

# Total fields across the file
ppperl -lane '$t += @F; END { print $t }' table.txt
```

The `$n + 0` idiom prints `0` rather than empty when the counter was never incremented.

Sums, minima, maxima

Use `List::Util` via `-M`.

```
# Per line
ppperl -MList::Util=sum -lane 'print sum @F' data.txt
ppperl -MList::Util=min -lane 'print min @F' data.txt
ppperl -MList::Util=max -lane 'print max @F' data.txt
```

(continues on next page)

(continued from previous page)

```
# Across all lines
pperl -MList::Util=sum -lane '$s += sum @F; END { print $s }' data.
→txt
ppperl -MList::Util=max -alne '$m = max($m // (), @F); END { print $m.
→}' data.txt
```

`$m // ()` returns an empty list when `$m` is undefined, so the first iteration does not pass undef to max. See *numeric* for the full arithmetic chapter.

Longest and shortest line

```
ppperl -ne '$l = $_ if length > length $l; END { print $l }' file
ppperl -ne '$s = $_ if $. == 1 || length < length $s; END { print $s.
→}' file
```

Unique lines (first sighting only)

```
ppperl -ne 'print unless $seen{$_}++' file

# Or the List::Util one-liner (slurps):
ppperl -MList::Util=uniq -e 'print uniq <>' file
```

The first form streams - memory scales with distinct lines, not total lines. The second slurps everything into memory, then emits. Prefer the first for large inputs.

Duplicates only

```
# Print each duplicate line once, when first seen for the second.
→time
ppperl -ne 'print if ++$seen{$_} == 2' file
```

Context-windowed filters

Printing the line *around* a match, not just the match itself.

```
# Line immediately before each match (grep -B 1, with caveats)
ppperl -ne '/pattern/ && $prev && print $prev; $prev = $_' file

# Line immediately after each match (grep -A 1)
ppperl -ne 'print if $p; $p = /pattern/' file

# tail -n 10
ppperl -ne 'push @a, $_; shift @a if @a > 10; END { print @a }' file
```

For rich `-A` / `-B` / `-C` behaviour, `grep` is the right tool. One-liners fit when the window rule is unusual (e.g. “print the line matching X, plus every line until the next blank line”).

```
# Print from /SECTION/ to the next blank line
ppperl -ne 'print if /SECTION/ .. /^$/' file
```

Record mode: **-0**, **-00**, **-0777**

Switch away from line-based reading when the natural record is not a line.

Slurp with **-0777**

```
# Single-shot regex across the whole file
ppperl -0777 -ne 'print "match\n" if /BEGIN.*?END/s' text.txt

# Replace only the first occurrence across the file
ppperl -0777 -i.bak -pe 's/TODO/DONE/' notes.txt
```

-0777 sets `$/ = undef`. The entire file comes in as one `$_`. The `/s` modifier makes `.` match newlines. See *regex-recipes* for the full multi-line story.

Paragraph mode with **-00**

-00 sets `$/ = ""`. Each read returns everything up to the next blank-line delimiter.

```
# Paragraphs containing "TODO"
ppperl -00 -ne 'print if /TODO/' notes.md

# Reverse paragraph order
ppperl -00 -e 'print reverse <>' notes.md
```

NUL-delimited records with **-0**

```
# Count files produced by find -print0
find . -type f -print0 | pperl -0 -ne '$n++; END { print "$n\n" }'

# Convert NUL-delimited stream to newline-delimited
find . -type f -print0 | pperl -0 -pe 's/\0/\n/'
```

In-place editing with **-i**

-i overwrites each input file with what **-p** prints. **-i.bak** keeps a backup.

```
# Lowercase every occurrence of HOSTNAME in every .conf (keep
↳ backup)
ppperl -i.bak -pe 's/HOSTNAME/\L$&/g' *.conf

# Delete every line matching DEPRECATED (no backup - only do this
# after you've validated the pattern on one file)
ppperl -i -ne 'print unless /DEPRECATED/' file.txt
```

The backup files pile up. Clean them up with `rm *.bak` once verified. See [gotchas](#) for the classic in-place traps.

Structured output

CSV-like (lightweight - commas are not quoted)

```
# Rebuild a TSV as CSV (naive, assumes no commas in fields)
ppperl -F'\t' -lane 'print join ",", @F' data.tsv

# Add a column of line numbers
ppperl -lane 'print join "\t", $., @F' data.tsv
```

JSON (stdlib)

```
# Emit one JSON object per line (no commas in values assumed)
ppperl -MJSON::PP -F: -lane '
    print JSON::PP->new->encode({
        user => $F[0], uid => 0+$F[2], shell => $F[-1]
    })
' /etc/passwd
```

Multi-line one-liners like this are where an *alias* starts to pay for itself.

Numbered input

```
ppperl -pe '$_ = "$. $_"' # 1 gate, 2 apple, ...
↪...
ppperl -pe 's/^/sprintf "%5d ", $./e' # padded width
```

The `/e` modifier on `s///` treats the replacement as Perl code. See [regex-recipes](#).

Find out more

- [regex-recipes](#) - the regex-dependent one-liners: captures, lookahead, named groups, `/e`, `/r`.
- [numeric](#) - arithmetic, statistics, randoms, date math.
- [aliases](#) - turn the recipes you use weekly into shell functions.
- [gotchas](#) - quoting, encoding, `-i` backups, record- separator surprises.
- [perlop](#) - the flip-flop operator, the diamond operator, `s///`, `tr///`, quoting forms.

Regex-driven one-liners

The previous chapter kept regexps at the level `sed` or `awk` users already know: literal patterns inside `m//` and `s///`. This chapter covers one-liners whose shape depends on a regex feature beyond plain matching - extractions, captures, lookarounds, code-in-replacement, and non-destructive rewrites.

Readers who already know the constructs `((?:...)`, `(?<name>...)`, `(?=...)`, the `/e` and `/r` modifiers) can skim for the command-line framing. Readers who know what `/g` does but have never written `/e` will get the most out of reading straight through.

For the regex language itself, see the *regular expressions guide*.

Extracting all matches

`/g` in list context returns every match, not just the first.

```
# Every integer in the input
ppperl -ne 'print "$_\n" for /-?\d+/g' file.txt

# Every IPv4-shaped substring
ppperl -lne 'print for /\b(?:\d{1,3}\.){3}\d{1,3}\b/g' access.log

# Every HTTP header value (naive: no folded headers)
ppperl -nE 'say $1 if /^[\w-]+\s*(.+)/' headers.txt
```

`say` (enabled by `-E`) appends a newline, saving one `"\n"` and keeping the program short.

Non-greedy quantifiers

`*?`, `+?`, `??`, `{n,m}?` - match as few characters as possible.

```
# Everything between the first < and the next > (tag-by-tag)
ppperl -lne 'print for /<(.*?)>/g' markup.html

# Every quoted string (double quotes, no escape handling)
ppperl -lne 'print for /"([^\"]*)"/g' config.txt
```

The negated character class `[^"]*` is usually a faster and safer non-greedy than `.*` when you know what must not appear inside.

Captures in substitution

The right-hand side of `s///` can reference left-hand captures.

```
# Swap every "word1 word2" pair
ppperl -pe 's/(\w+)\s+(\w+)/$2 $1/g' file.txt

# Quote every unquoted key in key=value pairs
ppperl -pe 's/(\w+)=/$1="/g; s/=(("[^\n"]+))/"$1"/g' config.txt

# Increase every trailing number by one
ppperl -pe 's/(\d+)/$1 + 1/e' file.txt
```

/e - replacement is Perl code

The /e modifier treats the replacement as code; its return value is what gets substituted in.

```
# Number every word in the file (global counter)
ppperl -pe 's/(\w+)/++$i . ".$1"/ge' file.txt

# Number words per line (counter reset each line)
ppperl -pe '$i = 0; s/(\w+)/++$i . ".$1"/ge' file.txt

# Wrap every integer in base 16
ppperl -pe 's/\d+/sprintf "0x%x", $&/ge' file.txt

# Convert Celsius-labelled numbers to Fahrenheit
ppperl -pe 's/(\d+)C\b/sprintf "%.1fF", $1 * 9 / 5 + 32/ge' weather.
→txt
```

/ge together: evaluate the right-hand side as code, do it for every match. \$& is the whole matched text; \$1, \$2, ... are captures.

/ee evaluates twice - the first e produces a string, the second evaluates that string. Useful when the replacement string is itself dynamic code read from input; needed rarely, a hazard often.

/r - non-destructive substitution

/r returns the modified string instead of rewriting \$_. The original stays intact.

```
# Print filename and its .bak variant side by side
ls *.txt | pperl -pe '$_ = $_ . ($_ =~ s/$/.bak/r)'

# Compute a transformed version without touching $_
ppperl -ne 'my $upper = $_ =~ tr/a-z/A-Z/r; print $_, $upper' file.
→txt
```

Reach for /r when a single line needs to be printed in two forms, or when the transformation is conditional and you want the untouched line back in the failure branch.

Named captures

(?<name>...) captures into \${name} and into %+.

```
# Parse combined-log-format access lines
ppperl -nE '
    if (/^(?<ip>\S+).*(?<method>\S+) (?<path>\S+)/) {
        say "${ip} ${method} ${path}";
    }
' access.log
```

Named captures pay for themselves once the pattern has more than two groups, or when the same group needs to be referenced both in the pattern and in the replacement code.

Lookahead and lookbehind

Zero-width assertions: match a position where the surrounding text satisfies a condition, without consuming the condition itself.

```
# Digits preceded by $ (e.g. "$42" → 42, ignore "42")
ppperl -lne 'print for /(?!=\$)\d+/g' prices.txt

# Words NOT followed by "!"
ppperl -lne 'print for /\b\w+\b(?!)/g' shouts.txt

# Insert CRLF only where LF is not already preceded by CR
ppperl -pe 's/(?!\\r)\\n/\\r\\n/g' unix.txt
```

Common pairs:

Lookaround	Meaning
(?=X)	Next characters match X
(?!X)	Next characters do not match X
(?<=X)	Preceding characters match X
(?<!X)	Preceding characters do not match X

Lookbehind in pperl supports variable-length patterns; see *regular expressions: anchors and assertions*.

Multi-line matches across slurped input

-0777 slurps the entire file into \$_. The regex modifiers that matter in that mode:

- /s - . matches \n.
- /m - ^ and \$ match at every line start/end, not just the start/end of the whole string.

```
# Every BEGIN...END block (non-greedy), each on one output line
ppperl -0777 -lne '
    while (/BEGIN(?:.*?)END/sg) {
        (my $body = $1) =~ s/\\n/ /g;
        print $body;
    }
' text.txt

# Paragraphs where every line starts with ">"
ppperl -00 -ne 'print if /\A(?:>.*\\n?)+\\z/m' mail.txt
```

\A and \z anchor the whole string start and end, unaffected by /m. Use them when /m is on but you really do mean the whole string.

Transliteration with `tr`

`tr` (also spelt `y`) is character-by-character substitution. Not a regex, but a close neighbour - and the right tool for character-set jobs.

```

ppperl -pe 'tr/A-Za-z/N-ZA-Mn-za-m/'           # ROT13
ppperl -pe 'tr/A-Za-z/a-zA-Z/'                 # swap case
ppperl -pe 'tr/a-z//d'                         # delete lowercase
→letters
ppperl -pe 'tr/ \t/ /s'                         # squeeze
→whitespace runs to one space

# Count commas on each line (tr returns the count)
ppperl -ne 'print tr/,//, "\n"' data.csv

```

The `d` flag deletes characters in the left set that have no counterpart on the right. The `s` flag squeezes runs. The `c` flag complements the left set (everything NOT listed).

Parsing delimited substrings

CSV with quoted commas

A quick-and-dirty split that honours double-quoted fields:

```

ppperl -ne '
    my @F = /"([^\"]*)"|([^\,]+)/g;
    @F = grep defined, @F;
    print join("|", @F), "\n";
' quoted.csv

```

Correct and fast enough for a one-off. For anything that must handle embedded quotes ("he said "hi""), use `Text::CSV`; see *numeric*.

URL query strings

```

# Each key=value pair on its own line
ppperl -lne 'print for /([^\?&=]+)=([^\&]*)/g' urls.txt

```

Practical extraction one-liners

Words by length

```

# Words of exactly 5 characters
ppperl -lne 'print for /\b\w{5}\b/g' file.txt

```

Numbers with units

```

ppperl -lne 'print for /\b\d+(?:\.\d+)?\s*(?:ms|s|kb|mb|gb)\b/gi'
→timings.txt

```

HTTP User-Agent from a request log

```
ppperl -nE 'say $1 if /^User-Agent: (.+)$/ ' request.log
```

Balanced braces (as many as pperl's regex engine supports)

ppperl's regex engine supports recursion via `(?R)` and `(?1)`, so single-level brace balancing is a one-liner:

```
ppperl -0777 -nE 'say $& while /\{(?:[^\}]+|(?R))*\}/g' code.txt
```

For deeply nested structures, reach for a parser. Regex is the right tool up to the complexity where it stops being one.

Find out more

- [regular expressions guide](#) - the language underneath every recipe here, including modifiers, performance, and Unicode.
- `m` - the match operator.
- `s` - substitution, the full set of modifiers.
- `tr` - transliteration reference.
- [progression](#) - simpler `s///` recipes without captures or code.
- [perlop](#) - the `qr//` form for precompiling patterns referenced by name inside a one-liner.

Numeric and date one-liners

ppperl as a calculator, stats tool, and date-arithmetic engine. This chapter collects the recipes that turn numeric columns, timestamps, and IP addresses into useful output without a script file.

The assumed baseline is a reader who has read [switches](#) and [progression](#). The `-lane / -Mlist::Util=...` idioms used below were introduced there.

Where this chapter reaches for performance: large numeric reductions are one of the cases where pperl's JIT earns its cost. A `for (1 .. 100_000_000) { $s += $_ }` accumulation runs as compiled machine code after the interpreter warms up. You do not opt in - it triggers automatically on arithmetic-heavy numeric loops.

Plain-arithmetic one-liners

```
ppperl -E 'say 2 ** 64' # 1.
↪84467440737096e+19
ppperl -E 'say 2 ** 64 - 1' # rounds (float)
ppperl -MMath::BigInt -E 'say Math::BigInt->new(2)->bpow(64)'
#
↪18446744073709551616
```

(continues on next page)

(continued from previous page)

```

ppperl -E 'say 7 / 3'                # 2.33333333333333
ppperl -E 'say int(7 / 3)'           # 2
ppperl -E 'say 7 % 3'                # 1
ppperl -E 'say sqrt(2)'              # 1.4142135623731
ppperl -E 'say atan2(1, 1) * 4'      # 3.14159265358979
↪(pi)

```

`sprintf` controls output width and precision:

```

ppperl -E 'printf "%.4f\n", 22/7'    # 3.1429
ppperl -E 'printf "%10.2f\n", 42.5'  #          42.50
ppperl -E 'printf "%08d\n", 42'      # 00000042
ppperl -E 'printf "%x %o %b\n", 255, 255, 255' # ff 377 11111111

```

Sums, minima, maxima, averages

List::Util ships with Perl 5 - no install needed.

Per line

```

ppperl -MList::Util=sum -lane 'print sum @F'      data.txt
ppperl -MList::Util=min -lane 'print min @F'      data.txt
ppperl -MList::Util=max -lane 'print max @F'      data.txt

# Per-line mean
ppperl -MList::Util=sum -lane 'print sum(@F) / @F' data.txt

```

Across all lines

```

# Total sum
ppperl -MList::Util=sum -lane '$s += sum @F; END { print $s }' data.
↪txt

# Running min / max using // (defined-or)
ppperl -MList::Util=min -alne '$m = min($m // (), @F); END { print $m.
↪}' data.txt
ppperl -MList::Util=max -alne '$m = max($m // (), @F); END { print $m.
↪}' data.txt

# Mean of one column (third), ignoring blank lines
ppperl -lane 'next unless /\S/; $s += $F[2]; $n++;
                END { printf "%.4f\n", $s / $n }' data.txt

```

`$m // ()` returns an empty list when `$m` is undefined, so the first iteration does not taint min/max with an undef. See `perlop` for `//`.

Sum a single column

```
# Sum of column 3 (zero-indexed: $F[2])
ppperl -lane '$s += $F[2]; END { print $s }' data.txt

# Negative or zero entries only
ppperl -lane '$s += $F[2] if $F[2] <= 0; END { print $s }' data.txt
```

Standard deviation

```
ppperl -lane '
    $n++; $s += $_; $s2 += $_**2 for @F;
    END {
        my $n2 = $n * @F;    # total count (assumes rectangular
        ↪data)
        my $mean = $s / $n2;
        printf "mean=%.4f sd=%.4f\n",
            $mean, sqrt($s2 / $n2 - $mean ** 2);
    }
' data.txt
```

For anything beyond mean+sd, reach for `Statistics::Descriptive`:

```
ppperl -MStatistics::Descriptive -lane '
    BEGIN { our $stat = Statistics::Descriptive::Full->new }
    $stat->add_data(@F);
    END { printf "median=%.2f mean=%.2f sd=%.2f\n",
        $stat->median, $stat->mean, $stat->standard_deviation }
' data.txt
```

Random numbers and sampling

```
# 10 random integers in [5, 15)
ppperl -E 'say join ", ", map { int(rand(10)) + 5 } 1 .. 10'

# 8-character random lowercase password
ppperl -E 'say join "", map { ("a".."z")[rand 26] } 1 .. 8'

# 8-character random alphanumeric
ppperl -E 'say join "", map { ("a".."z", 0..9)[rand 36] } 1 .. 8'

# Random UUID (using Data::UUID)
ppperl -MData::UUID -E 'say Data::UUID->new->create_str'

# Shuffle the fields of each line
ppperl -MList::Util=shuffle -alne 'print join " ", shuffle @F' data.
↪txt

# Pick one random line from a file (reservoir sample, size 1)
```

(continues on next page)

(continued from previous page)

```
ppperl -ne 'rand($.) < 1 && ($pick = $_); END { print $pick }' file.
→txt
```

The reservoir trick is the right answer when the file is too large to slurp or its line count is unknown - every line has an equal probability of being the one printed, with exactly one pass.

Factorials, GCD, LCM, primes

```
# Factorial
ppperl -E '$f = 1; $f *= $_ for 1 .. 20; say $f'
ppperl -MMath::BigInt -E 'say Math::BigInt->new(100)->bfac'

# GCD / LCM (built into Math::BigInt under bgcd / blcm)
ppperl -MMath::BigInt=bgcd -anle 'print bgcd(@F)' nums.txt
ppperl -MMath::BigInt=blcm -anle 'print blcm(@F)' nums.txt

# Euclid's algorithm for two numbers
ppperl -le '$n=20; $m=35; ($m,$n) = ($n, $m % $n) while $n; print $m'
```

Primality:

```
# Regex-based primality test (Abigail). Input: one integer per line.
ppperl -lne '(1 x $_) !~ /^1?$|^(11+?)\1+$/ && print "$_ prime"'
→nums.txt

# Reliable for large numbers: use Math::Prime::Util
ppperl -MMath::Prime::Util=is_prime -lne '
    print "$_ prime" if is_prime($_)
' nums.txt
```

The regex trick is a curiosity; `Math::Prime::Util` is the serious answer.

IP ↔ integer

```
# Dotted-quad → 32-bit integer
ppperl -MSocket -le 'print unpack "N", inet_aton "127.0.0.1"'
# 2130706433

# Integer → dotted-quad
ppperl -MSocket -le 'print inet_ntoa pack "N", 2130706433'
# 127.0.0.1
```

`unpack "N"` reads four bytes as a big-endian 32-bit unsigned integer. The matching `pack "N"` writes one.

Date and time

Today, now, epoch

```
pperl -E 'say time'                                # epoch seconds
ppperl -E 'say scalar localtime'                   # Mon Apr 22
↪14:31:07 2026
ppperl -E 'say scalar gmtime'                       # same, UTC
ppperl -MPOSIX=strftime -E 'say strftime "%F %T", localtime'
                                                    # 2026-04-22
↪14:31:07
```

`strftime` is the clean way to format; `localtime` in list context returns (sec, min, hour, mday, mon, year, wday, yday, isdst).

Date arithmetic

`POSIX::mktime` normalises out-of-range fields (month 13 becomes month 1 of next year, day 32 becomes day 1 of next month, etc.), so date arithmetic means “subtract N from a field, `mktime`, `strftime`”.

```
# Yesterday's date
ppperl -MPOSIX=strftime -MPOSIX=mktime -E '
    my @t = localtime;
    $t[3] -= 1;
    say strftime "%F", localtime mktime @t
'

# 30 days ago
ppperl -MPOSIX=strftime -MPOSIX=mktime -E '
    my @t = localtime;
    $t[3] -= 30;
    say strftime "%F", localtime mktime @t
'

# First Monday of next month
ppperl -MPOSIX=strftime -MPOSIX=mktime -E '
    my @t = localtime;
    $t[3] = 1; $t[4]++;                                # first of next month
    my $epoch = mktime @t;
    my @nm = localtime $epoch;
    $epoch += ((8 - $nm[6]) % 7 || 7) * 86400 if $nm[6] != 1;
    # ^ if not already Monday, advance to next Monday
    say strftime "%F", localtime $epoch
'
```

For anything more elaborate (time zones, business-day math, ISO weeks), reach for `DateTime`:

```
ppperl -MDateTime -E '
    my $d = DateTime->now->subtract(days => 30);
```

(continues on next page)

(continued from previous page)

```
say $d->strftime("%F %T %Z")
,
```

Timestamps in logs

```
# Sum the durations (last field, milliseconds) of lines matching
→ "slow"
ppperl -lane '$s += $F[-1] if /slow/; END { print $s, " ms total" }' .
→ app.log

# Count events per hour (assumes ISO timestamp first field)
ppperl -lane '
    (my $hour = $F[0]) =~ s/:\d{2}:\d{2}.*//;
    $h{$hour}++;
    END { print "$_ $h{$_}" for sort keys %h }
' events.log
```

CSV and TSV

TSV with no embedded tabs: `-F '\t'` is fine. Real CSV needs `Text::CSV`:

```
# Sum the third column of a properly quoted CSV
ppperl -MText::CSV -E '
    my $csv = Text::CSV->new({ binary => 1 });
    my $total = 0;
    while (my $row = $csv->getline(*STDIN)) {
        $total += $row->[2];
    }
    say $total
' < data.csv

# Re-emit a CSV with only selected columns
ppperl -MText::CSV -E '
    my $csv = Text::CSV->new({ binary => 1, eol => "\n" });
    while (my $row = $csv->getline(*STDIN)) {
        $csv->print(*STDOUT, [ @{$row}[0, 2, 4] ]);
    }
' < data.csv > trimmed.csv
```

The `binary => 1` option keeps bytes intact; the `eol => "\n"` option makes print append newlines.

JSON

`JSON::PP` ships with Perl; `Cpanel::JSON::XS` is faster if it is installed.

```
# Extract the "name" field from each line of NDJSON
ppperl -MJSON::PP -lne '
```

(continues on next page)

(continued from previous page)

```
my $o = JSON::PP->new->decode($_);
print $o->{name};
' events.ndjson

# Convert each line of TSV to JSON
ppperl -MJSON::PP -F'\t' -lane '
    print JSON::PP->new->encode({ id => $F[0], name => $F[1] })
' data.tsv

# Pretty-print a single JSON document
ppperl -MJSON::PP -0777 -ne '
    print JSON::PP->new->pretty->encode(JSON::PP->new->decode($_))
' small.json
```

For large JSON, streaming parsers (`JSON::Streaming::Reader`) beat slurping; single-document files under a few hundred megabytes are fine with the above.

Base64, URL encoding, hex

```
# Base64 encode/decode (of a whole file)
ppperl -MMIME::Base64 -0777 -ne 'print encode_base64($_)' file
ppperl -MMIME::Base64 -0777 -ne 'print decode_base64($_)' file.b64

# URL encode/decode
ppperl -MURI::Escape -lne 'print uri_escape($_)' < urls.txt
ppperl -MURI::Escape -lne 'print uri_unescape($_)' < encoded.txt

# Hex ↔ integer
ppperl -E 'printf "%x\n", 255'           # ff
ppperl -E 'say hex "ff"'                 # 255

# Hex dump of a file (like xxd -p)
ppperl -0777 -ne 'print unpack "H*", $_' file.bin

# Reverse: hex dump → bytes
ppperl -0777 -ne 'chomp; print pack "H*", $_' file.hex > file.bin
```

Find out more

- *aliases* - wrap the recipes you use weekly into shell functions with positional arguments.
- *progression* - the reduction idioms these numeric recipes build on.
- `sprintf`, `printf` - formatted output.
- `time`, `localtime`, `gmtime` - the built-ins underneath the date recipes.

From one-liner to shell alias

A one-liner is disposable by design. The moment you type the same one twice, something has earned a place in your shell config. This chapter covers the mechanics of lifting a pperl recipe into a bash or zsh shell function, how to parametrise it, and when to stop and write a real script instead.

Assumed baseline: you have a bash or zsh shell, a `~/ .bashrc` or `~/ .zshrc` (or an `rc.d / functions` directory your shell sources), and the patience to close and reopen one terminal after editing it.

Alias versus function

Bash and zsh have two mechanisms. They are not interchangeable.

- `alias` is textual substitution. It runs at the start of a command line, replacing the alias name with its value before the shell re-parses. No arguments, no pipelining quirks beyond what substitution gives you.
- A shell function is a named piece of shell code. It takes positional arguments (`$1`, `$2`, `$@`), reads stdin like any other command, and can pipe into and out of other commands.

Rule of thumb. If the recipe takes no arguments, alias is enough. If it needs any argument (a column number, a regex, a filename), use a function.

Alias - the zero-argument recipe

```
# In ~/.bashrc
alias sum-stdin='ppperl -MList::Util=sum -lane '""'"print sum @F'""'"
alias count-lines='ppperl -lne '""'"END { print $. }'""'"
```

The quoting is ugly because a bash single-quoted string cannot contain a single quote; the idiom `'""'"` closes-and-reopens. This is the main reason to prefer functions: they let you quote the Perl once, cleanly.

Function - the common case

```
# In ~/.bashrc
sum-stdin() { pperl -MList::Util=sum -lane 'print sum @F'; }
count-lines() { pperl -lne 'END { print $. }'; }
```

Usage is the same:

```
$ printf '1 2 3\n4 5 6\n' | sum-stdin
6
15
```

Parametrised functions

The shell interpolates `$1`, `$@`, `"$1"` into the function body before executing it. Inside the Perl program, the shell-level `$1` appears as a literal value - Perl's own `$1` (the regex capture) is untouched by shell interpolation only if you quote with single quotes.

The mixture is solvable with care. Two idioms cover almost every case.

Idiom 1: shell-interpolated column / value

Use double quotes around the Perl program so the shell expands `$1`, then escape Perl's own `$` with `\$`.

```
# Sum column N of stdin: sumcol N
sumcol() {
    pperl -MList::Util=sum -lane "\$s += \$F[$1]; END { print \$s }"
}

# Usage
$ cat data.tsv
10 apple 3
15 pear 7
20 plum 2
$ sumcol 2 < data.tsv
12
```

Every `\$` is a literal dollar sign for Perl; every `$1` (unescaped) is the first positional argument expanded by the shell.

Idiom 2: program in single quotes, data via environment

Single-quoted Perl stays readable, and values reach the program via `%ENV`.

```
# Grep with a pperl regex: pgrep PATTERN [FILE...]
pgrep() {
    local pat=$1
    shift
    pat="$pat" pperl -ne 'print if /$ENV{pat}/' "$@"
}

$ pgrep '\berror\b' app.log
```

This keeps `$1` (Perl capture) usable and avoids the backslash cascade.

Idiom 3: mixing both

```
# Replace column N's commas with semicolons, in place: csv-fix N
↪FILE...
csv-fix() {
    local col=$1
    shift
```

(continues on next page)

(continued from previous page)

```
col="$col" pperl -i.bak -F'\t' -lane '
    my $c = $ENV{col};
    $F[$c] =~ tr/,/,/;
    print join("\t", @F);
    ' "$@"
}
```

The column number arrives in `$ENV{col}` - no escaping dance. The file list propagates via `"$@"` so spaces in filenames survive.

Practical aliases worth keeping

Annotated list of recipes that earn their place in a shell config for anyone who processes text at the command line daily.

sumcol N - sum column N (whitespace-separated)

```
sumcol() {
    pperl -MList::Util=sum -lane "\$s += \$F[1]; END { print \$s }"
}
```

sumcolf N FS - sum column N with custom field separator

```
sumcolf() {
    local col=$1 fs=$2
    col="$col" pperl -MList::Util=sum -F"$fs" -lane '
        $s += $F[$ENV{col}]; END { print $s }
    '
}

$ sumcolf 3 : < /etc/passwd      # sum of UIDs
```

colstats N - min / max / mean / count for column N

```
colstats() {
    col="$1" pperl -MList::Util=qw(min max sum) -lane '
        my $c = $ENV{col};
        push @v, $F[$c];
        END {
            printf "n=%d min=%g max=%g mean=%.4f\n",
                scalar @v, min(@v), max(@v), sum(@v) / @v;
        }
    '
}
```

uniq-stream - streaming unique, keeping first sighting

```
uniq-stream() { pperl -ne 'print unless $seen{$_}++;' ; }
```

Unlike `sort -u`, order is preserved. Unlike `uniq`, input need not be sorted.

extract PAT - print every match of a regex, one per line

```
extract() { pat="$1" pperl -lne 'print for /$ENV{pat}/g'; }

$ extract '\b\d{3}-\d{4}\b' < phonebook.txt
555-1212
555-9876
```

between A B - print everything between regex A and regex B, inclusive

```
between() {
    a="$1" b="$2" pperl -ne 'print if /$ENV{a}/ .. /$ENV{b}/'
}

$ between 'BEGIN cert' 'END cert' < bundle.pem
```

json-pretty - pretty-print a single JSON document

```
json-pretty() {
    pperl -MJSON::PP -0777 -ne '
        print JSON::PP->new->pretty->canonical->encode(
            JSON::PP->new->decode($_)
        )
    '
}
```

passwd-json - /etc/passwd as NDJSON

```
passwd-json() {
    pperl -MJSON::PP -F: -lane '
        print JSON::PP->new->encode({
            user => $F[0], uid => 0+$F[2], gid => 0+$F[3],
            home => $F[5], shell => $F[-1]
        })
    ' < /etc/passwd
}
```

words-by-freq - count word frequencies, sorted

```
words-by-freq() {
    pperl -ne '
        $w{lc $_}++ for /\b[[:alpha:]]+\b/g;
        END { print "$w{$_} $_\n" for sort { $w{$b} <=> $w{$a} } }
    ↪keys %w }
}

$ words-by-freq < essay.txt | head -20
```

strip-comments LANG - strip comments for a few common languages

```
strip-comments() {
    case "$1" in
        sh|py|rb|pl) pperl -pe 's/(?<!\n)#.*//' ;;
        ↪c|cpp|rs|java|js) pperl -0777 -pe 's{//[^\n]*|/\n.*?\/}{}gs
        ' ;;
        sql|lua) pperl -pe 's/--[^\n]*//' ;;
        *) echo "unknown language: $1" >&2; return 1 ;;
    esac
}
```

When to stop and write a script

The one-liner → alias transition ends when:

- The program needs an if / else branch longer than a single ? : conditional.
- There are more than two or three -M modules.
- You find yourself debugging via trial-and-error on the command line.
- The program is the only alias in your config that gets a comment above it explaining what it does.
- A colleague is going to run it.

Move to a script file. Drop the switches into the shebang line and let the program read naturally:

```
#!/usr/bin/env pperl
use strict;
use warnings;
use List::Util qw(sum);

my $col = shift @ARGV // die "usage: $0 COLUMN [FILE...]\n";
my $total = 0;
while (<>) {
    my @f = split;
    $total += $f[$col];
}
```

(continues on next page)

(continued from previous page)

```
}  
print "$total\n";
```

Give it a sensible name, `chmod +x`, and move on. The one-liner did its job: it was the shortest path from “I have an idea” to “I know this works”; the script is the path from “I know this works” to “my colleague can read it.”

Find out more

- *progression* - the recipes these aliases wrap.
- *gotchas* - quoting inside shell functions and aliases.
- *numeric* - more candidates for the alias list, especially the date-arithmetic ones.

Traps and surprises

The ways pperl one-liners silently do something other than what the shell user meant. Each section names a trap, shows its symptom, and gives the fix. Read straight through once to prime the pattern-match; come back when a recipe is misbehaving.

Quoting hell

The single biggest source of wasted time in one-liner work.

Single quotes are your friend

Under bash and zsh, text inside single quotes is not interpolated. No `$VAR` expansion, no backtick execution, no `!` history substitution (zsh doesn't do `!`; bash does, and it bites). Use single quotes for Perl programs unless you have a reason not to.

```
# Correct  
ppperl -ne 'print if /$var/' file           # Perl sees literal  
→$var  
  
# Trap  
ppperl -ne "print if /$var/" file          # shell expands $var  
→before pperl sees it
```

The ! surprise in interactive bash

```
$ pperl -E 'say "hello!"'  
bash: !": event not found
```

Bash's history expansion fires on unquoted `!` (and some escaped forms). Workarounds:

- `set +H` in `~/ .bashrc` disables history expansion.
- Escape the `!` when writing the command: `"hello\!"`.

- Keep the shell's single-quoted string around the program - ! is inert inside ' . . . ' once history expansion is off.

A single quote inside a single-quoted string

Bash has no escape for it. The workaround is the four-character close-open dance: `'\''` reads as close-quote, literal-quote, open-quote.

```
# Want Perl to see: print "it's here"
ppperl -e 'print "it'\''s here\n"'
```

For any program with more than one embedded apostrophe, give up on ' . . . ' and use `q/ . . . /` inside double quotes:

```
ppperl -e "print q/it's here/, qq/\n/"
```

or move the program into a file.

The \$ escape cascade under double quotes

If you quote the Perl program in double quotes (because you want `$1` to interpolate from the shell), Perl's own variables need `\$`:

```
col=3
ppperl -lane "\$s += \$F[$col]; END { print \$s }"      # works
```

That is legible once. After the third such one-liner in a row, move to environment variables:

```
col=3
col=$col pppperl -lane '$s += $F[$ENV{col}]; END { print $s }'
```

See *aliases#parametrised-functions*.

Locale and encoding

ppperl's default I/O is bytes. If your input is UTF-8, Perl still sees a byte string until you ask for otherwise.

The symptom

```
$ echo 'h      ' | pppperl -lne 'print length'
6                                     # "h" + 2 bytes for   
  + 3 more = 6
$ echo 'h      ' | pppperl -CSD -lne 'print length'
5                                     # correct
```

The fix is `-C` (see *switches*):

- `-CSD` - STDIN/STDOUT/STDERR and file opens all UTF-8.
- `-CS` - stdio only.

- -CA - treat @ARGV as UTF-8.

Regex matching on UTF-8

Without -C, \w matches ASCII word characters only. With -C, \w matches Unicode letters/digits as defined by Perl's Unicode property tables.

```
# Count "word" characters in a multilingual file
ppperl -CSD -lne '$c += () = /\w/g; END { print $c }' file.txt
```

Mojibake on the way out

If pperl is reading UTF-8 cleanly but the terminal shows garbled characters, the problem is further down - likely LC_ALL or the terminal's own encoding. locale and tput diagnose it; this is not a pperl problem.

-i in-place editing

Always .bak until verified

```
# Destructive if the pattern is wrong
ppperl -i -pe 's/OLD/NEW/g' *.conf

# Safe
ppperl -i.bak -pe 's/OLD/NEW/g' *.conf
# ...verify...
rm *.conf.bak
```

A wrong pattern with bare -i replaces the file's contents with something useless. There is no recovery short of version control. Use -i.bak, confirm, delete.

The empty-file trap

-i writes whatever -p (or -n plus explicit prints) emits. If the program prints nothing, the file becomes empty.

```
# Wrong: -i with -n and no print - empties the file
ppperl -i -ne '/KEEP/ && print' file.txt      # prints only lines
↳ matching KEEP

# Same thing, probably what you meant: keep the matching lines
ppperl -i -ne 'print if /KEEP/' file.txt
```

Both are correct. The point: under -n, you own every byte of output.

In-place across many files with ownership caveats

-i creates a new file and renames. On Linux, this means the new file inherits the calling user's ownership and umask. Root-owned configs edited by non-root users will fail; root-owned configs edited by root will keep the mode but may lose SELinux context. Know your threat model before pointing -i at /etc.

The diamond operator <> and magic open

<> (the “diamond”) is what -n / -p reads from. It reads from each argument in @ARGV as a filename, falling back to STDIN if @ARGV is empty.

The “-” is STDIN

If an argument is “-”, <> reads STDIN at that point. Useful for interleaving:

```
echo prefix | pperl -ne 'print' - trailer.txt
# prints: prefix, then contents of trailer.txt
```

Filename metacharacters (and why they matter less than they used to)

Historically, bare <> treated “>foo” as “open foo for writing” - a two-argument open that read the mode from the filename. Modern Perl’s <> uses three-argument open internally, so filenames starting with <, >, |, or containing pipes are treated as plain filenames. No attack surface on input filenames with modern pperl.

Filename is in \$ARGV

```
ppperl -lne 'print "$ARGV: $_" if /\bERROR\b/' *.log
```

\$ARGV updates as each file opens. “-” is the special value for STDIN.

\$. keeps counting across files

\$. is not reset when <> moves to the next file. If you want per-file line numbers:

```
ppperl -lne 'close ARGV if eof; print "$ARGV$.: $_" if /TODO/' *.pl
```

close ARGV if eof resets \$. when the current file ends.

Record-separator traps

-l removes then restores - once

-l chomps input and sets \$\
But -l applied twice (say, you added -l after -0) confuses people:

```
# Read NUL-delimited input, output newline-terminated
ppperl -0 -lpe '' # correct: -l sets $\  
↪= "\n"

# But if you write -l0 Perl parses that as -l with argument 0:
ppperl -l0 -pe '' # $\  
↪probably NOT what you wanted
```

Put -l after -0 (no argument to -l) when you want the standard pair.

Paragraph mode keeps trailing blanks

`-00` (paragraph mode) sets `$/ = ""`. Perl reads up to and including the blank-line separator. Without `-1`, the separator stays attached to each paragraph; printing it back out preserves the blank line. With `-1`, the separator is chomped and `$\ = "\n"` appends one newline on output - paragraphs collapse.

```
$ printf 'a\n\nb\n\nc\n' | pperl -00 -pe ''
a

b

c
$ printf 'a\n\nb\n\nc\n' | pperl -00 -lpe ''
a
b
c
```

Know which one you want.

`-0777` and `$.`

Under `-0777`, each file is one record. `$.` increments per record, so `$.` is the number of files processed, not the number of lines.

```
ppperl -0777 -e 'while (<>) { print "file $.: length=", length, "\n"
→} ' *.txt
# file 1: length=1024
# file 2: length=2048
```

The `-F` interpretation of its argument

`-F` takes a regex. Perl versions before 5.10 required you to write the delimiters; modern `ppperl` accepts both forms:

```
ppperl -F:      -lane '...'      # bare: Perl quotes it
ppperl -F/:/    -lane '...'      # explicit delimiters
ppperl -F'\t'   -lane '...'      # tab - regex form
```

The common mistake is a literal tab in a shell where the terminal swallowed it:

```
ppperl -F'      ' -lane '...'      # fragile: tab may
→be a space in your paste
```

Write `-F'\t'` unless you are pasting from known-good source.

Operator precedence around `print`

`print` has lower precedence than most operators. This trips people whose `awk` reflex is wrong for Perl:

```
# Wrong: prints into a filehandle named "..."  
ppperl -ne 'print "out.txt" $_'  
  
# Right: parentheses or commas  
ppperl -ne 'print $_, "\n"'  
ppperl -ne 'print("foo\n")'  
  
# Wrong: the binary || applies to print's argument list  
ppperl -ne 'print if $_ || "never"' # works but obscure  
  
# One particular case worth memorising:  
ppperl -ne 'print (1+1)*2' # prints 2, not 4  
# ^ print(1+1) is the function call; *2 is applied to its return  
↪value
```

When in doubt, put parentheses around print's arguments or end the program with an explicit semicolon.

tr is not a regex

tr / y take character sets, not patterns. tr/\d/X/ does NOT match digits - it replaces literal backslash-d characters.

```
ppperl -pe 'tr/0-9/X/' # replace digits with  
↪X (character range)  
ppperl -pe 's/\d/X/g' # replace digits with  
↪X (regex)
```

Use tr for character-class work (case conversion, deletion, ranges) and s/// when you need any regex machinery.

BEGIN and END scope

BEGIN { ... } and END { ... } blocks are part of the program, not the loop body. Variables declared with my inside BEGIN are not visible to the loop body. Use our or global variables for state that needs to persist from BEGIN into the loop:

```
# Wrong: $greeting is not in scope inside the loop  
ppperl -ne 'BEGIN { my $greeting = "Hi"; } print "$greeting $_"'  
  
# Right: declare without my, or use our  
ppperl -ne 'BEGIN { $greeting = "Hi"; } print "$greeting $_"'
```

In practice, most BEGIN blocks do setup with side effects (\$\ = "\n", setting a format) rather than bindings.

die and the \n convention

`die` without a trailing `\n` appends " at -e line N.\n" to its message. With a trailing `\n`, it prints the message as-is. Choose deliberately:

```
# Developer diagnostic - want the location
ppperl -E 'die "unreachable" if $x > 100'

# User-facing message - don't leak program structure
ppperl -E 'die "bad input\n" unless /\d+/'
```

The same rule applies to `warn`.

Missing newline at end of file

If the last line of an input file has no trailing newline, `-l` still chomps what's there (nothing), and `$_` still lacks a `\n`. `-p` reprints it without a final newline, and the next line of shell output runs into it. Rarely a correctness problem; often a display problem.

```
ppperl -pe '' no-final-newline.txt           # output lacks
↪trailing \n
```

Find out more

- [switches](#) - the mechanical expansions of every flag mentioned here.
- [perlvar](#) - the full definition of every special variable that the traps above depend on (`$/`, `$\`, `$.`, `$ARGV`, `@ARGV`).
- [perlop](#) - the diamond operator, `tr`, and the quoting forms referenced above.

4.8 Tying variables

A **tied** variable looks like an ordinary scalar, array, hash, or filehandle, but every read, write, and iteration on it is rerouted through methods you write. The variable is the public face; a class of your design is the machinery behind it. Assigning to a tied scalar calls your `STORE`; reading a tied hash key calls your `FETCH`; running keys over a tied hash drives your `FIRSTKEY` and `NEXTKEY`. The caller writes plain Perl and never sees the class.

That indirection is what makes tie powerful. A hash can be a façade over an on-disk database. A scalar can recompute itself on every read. A filehandle can collect everything printed to it into a string. The code using the variable does not change; only the class behind it does.

This guide is about **writing the class** - the part `perltie` calls the implementation. The built-in side ([tie](#), [tied](#), [untie](#)) and the full method menus per variable type are already documented on the [tie](#) reference page. This guide links there for signatures and concentrates on what those pages do not teach: how to build a working tie class, what each method must return, and exactly when the runtime calls it.

4.8.1 Which chapter you want

Each variable type has its own constructor and its own method set. Read the chapter that matches the sigil in front of you.

- *Tied scalars* - TIESCALAR, FETCH, STORE, DESTROY. The smallest tie interface; start here to learn the shape of all the others.
- *Tied hashes* - TIEHASH plus the iteration protocol (FIRSTKEY / NEXTKEY), EXISTS, DELETE, CLEAR, and SCALAR. The richest of the data types.
- *Tied arrays* - TIEARRAY, the size protocol (FETCHSIZE / STORESIZE / EXTEND), and the mutators (PUSH, POP, SHIFT, UNSHIFT, SPLICE).
- *Tied filehandles* - TIEHANDLE and the I/O hooks (PRINT, PRINTF, WRITE, READLINE, READ, GETC, and friends).

4.8.2 How each chapter is built

Every chapter follows the same shape:

- **The contract by example.** Rather than restate the method menu (the [tie](#) reference page already lists every signature), each chapter builds one complete, runnable class and explains each method as it appears.
- **Call timing.** The non-obvious part of tie is *when* the runtime invokes each hook - which operator triggers which method, in which context, with which arguments. Each chapter spells this out.
- **Minimum viable vs. full.** The fewest methods you can define and still have a usable tie, then the rest of the menu and what each one buys you.
- **The easy way.** Each data type ships a `Tie::*` base class that implements the tedious methods in terms of a few core ones. Inherit from it and override only the interesting hooks.

4.8.3 What ties cost

Every operation on a tied variable is a method call. A tied hash lookup is a FETCH dispatch where a plain hash lookup is a single memory access - visibly slower in a tight loop. Tie is the right tool when the indirection earns its keep (a variable backed by a file, a computed value, a logging façade), not when you simply want a fancier hash. For bulk work over a tied container, read it into a plain array or hash once, operate, and write back.

4.8.4 Differences from upstream

The tie mechanism is Perl 5.42's own, so tie classes behave exactly as they do on upstream Perl.

Tied scalars

A tied scalar is the smallest tie interface and the best place to learn the shape of all the others. You define a constructor and two accessor methods; from then on, every read of the variable calls one of them and every write calls the other.

The class needs four methods, two of them optional:

- TIESCALAR - the constructor, called once by `tie`.
- FETCH - called on every read of the variable.
- STORE - called on every write to the variable.
- DESTROY - called when the tied variable goes away (optional).

The full signature list is on the `tie` reference page. This chapter shows what each method does by building one.

A complete class

A counter that clamps writes so the value never drops below zero:

```
use v5.36;

package Counter {
    sub TIESCALAR {
        my ($class, $start) = @_;
        my $n = $start // 0;
        return bless \$n, $class;
    }
    sub FETCH { my $self = shift; return $$self }
    sub STORE {
        my ($self, $value) = @_;
        $$self = $value < 0 ? 0 : $value;    # never go below zero
    }
    sub DESTROY { }
}

tie my $hits, 'Counter', 10;
say $hits;           # 10
$hits = 25;
say $hits;           # 25
$hits = -7;
say $hits;           # 0
```

The object behind the scalar can be anything blessed - here a reference to a plain number. The LIST after the class name in the `tie` call (10) is passed straight to TIESCALAR.

The method contract

TIESCALAR *classname*, LIST

The constructor. It receives the class name and whatever extra arguments were passed to `tie`. It must return a blessed reference; that reference becomes the tied object and the return value of the `tie` call. A common shortcut is a blessed scalar ref holding the value, as above, but a blessed hash ref is just as valid when the class needs more state.

FETCH *this*

Called every time the variable is read. It takes only the object. Whatever it returns

is what the reader sees. FETCH is where a computed scalar earns its keep: it can recompute, look something up, or log the access and return a stored value.

STORE this, value

Called every time the variable is assigned. It receives the object and the single new value. Its return value is ignored - the assignment expression's value comes from a subsequent FETCH, not from STORE. This is where validation, clamping, or write-through to a backing store lives.

DESTROY this

Called when the tied variable is destroyed (scope exit, or the last reference dropping). Define it only when the class holds a resource that needs releasing - an open handle, a lock. A counter has nothing to clean up, so its DESTROY is empty or omitted entirely.

Call timing

- Reading in any context calls FETCH. say \$hits, \$x = \$hits + 1, and "\$hits" each trigger one FETCH.
- Assignment calls STORE with the right-hand value. \$hits = 25 calls STORE(\$obj, 25).
- Compound assignment reads then writes: \$hits += 5 calls FETCH, adds 5, then STORE. So does \$hits++.
- The value of an assignment expression comes from FETCH, not STORE's return. After my \$r = (\$hits = -7), \$r is 0 because the clamp ran in STORE and the expression's value was fetched back.

Minimum viable class

The smallest useful tied scalar defines exactly TIESCALAR, FETCH, and STORE. Omit DESTROY unless you allocate something. A class with only TIESCALAR and FETCH produces a read-only scalar: it ties fine, but the first assignment dies with Can't locate object method "STORE".

The easy way: inherit from Tie::StdScalar

Tie::StdScalar (shipped inside Tie::Scalar) implements TIESCALAR, FETCH, STORE, and DESTROY over a blessed scalar ref. Inherit from it and override only the hook you want to change:

```
package Uppercase {
    use Tie::Scalar;
    our @ISA = ('Tie::StdScalar');
    sub STORE { my ($self, $v) = @_; $$self = uc $v }
}

tie my $shout, 'Uppercase';
$shout = 'hello';
say $shout;           # HELLO
```

STORE upper-cases on the way in; FETCH and the constructor come from the base class unchanged.

Tie::Scalar itself is the *abstract* base - it defines FETCH and STORE that croak, so subclassing it directly means supplying your own TIESCALAR. Reach for Tie::StdScalar when you want a working constructor for free; reach for Tie::Scalar only when you intend to write every method anyway and want the interface documented by inheritance.

See also

- [tie](#) - the built-in, with the full per-type method menu
- [tied](#) - recover the object backing the variable to call class-specific methods on it
- [untie](#) - dissolve the binding
- [Tied hashes](#) - the next step up: a constructor plus an iteration protocol
- Tie::Scalar - the base-class module; Tie::StdScalar is the concrete version you usually want

Tied hashes

A tied hash is the richest of the tie types, because a hash does more than read and write single elements - it iterates, it reports its size, it tests for key existence, and it clears. Each of those is a separate hook. Hashes were the first Perl data type to be tied (the original use case was binding %hash to an on-disk DBM file), and the interface still reflects that heritage.

The full method menu is on the [tie](#) reference page. This chapter builds one class and explains each hook, with attention to the part that trips people up: the iteration protocol.

A complete class

A hash whose keys are case-insensitive - \$h{Name} and \$h{NAME} address the same slot:

```
use v5.36;

package CaseInsensitive {
    sub TIEHASH { my $class = shift; return bless {}, $class }
    sub STORE   { my ($self, $key, $val) = @_; $self->{lc $key} =
        ↪$val }
    sub FETCH   { my ($self, $key) = @_; return $self->{lc $key} }
    sub EXISTS  { my ($self, $key) = @_; return exists $self->{lc
        ↪$key} }
    sub DELETE  { my ($self, $key) = @_; return delete $self->{lc
        ↪$key} }
    sub CLEAR   { my $self = shift; %$self = () }
    sub FIRSTKEY { my $self = shift; my $reset = keys %$self; return
        ↪each %$self }
    sub NEXTKEY { my $self = shift; return each %$self }
    sub SCALAR  { my $self = shift; return scalar %$self }
}
```

(continues on next page)

(continued from previous page)

```

}

tie my %h, 'CaseInsensitive';
$h{Name} = 'Ada';
$h{EMAIL} = 'ada@example.org';

say $h{name};           # Ada
say $h{email};          # ada@example.org
say exists $h{NAME} ? 'yes' : 'no';  # yes
say scalar %h ? 'nonempty' : 'empty'; # nonempty

for my $k (sort keys %h) {
    say "$k => $h{$k}";
}
# email => ada@example.org
# name => Ada

```

The object is a blessed hash ref used as the real storage; the hooks lower-case the key before touching it. Notice that iteration returns the *stored* (lower-cased) keys, which is exactly what a case-insensitive hash should do.

The method contract

TIEHASH classname, LIST

The constructor. Returns a blessed reference - usually but not necessarily a hash ref - that becomes the tied object.

FETCH this, key / STORE this, key, value

Single-element read and write, the same pair as for scalars but with a key argument. STORE's return value is ignored.

EXISTS this, key / DELETE this, key

Back the `exists` and `delete` built-ins. DELETE's return value becomes delete's return value; to match a plain hash, return what FETCH would have returned for that key before removing it.

CLEAR this

Triggered when the whole hash is emptied, typically by assigning the empty list (`%h = ()`). Remove everything.

FIRSTKEY this / NEXTKEY this, lastkey

The iteration protocol - see below. Together they drive `keys`, `values`, and `each`.

SCALAR this

Called when the hash is evaluated in scalar or boolean context (`scalar %h`, if `(%h)`, and since 5.28 `keys %h` in boolean context). Return a value that is true when the hash is non-empty and false when empty.

DESTROY this and UNTIE this

Optional cleanup hooks, identical in role to the scalar case. UNTIE is covered under the untie gotcha below.

The iteration protocol

This is the part of the hash interface that has no analogue in the scalar one, and the part that most often goes wrong.

When a `keys`, `values`, or `each` iteration begins, the runtime calls `FIRSTKEY` to get the first key, then `NEXTKEY` repeatedly, passing the previously returned key each time, until a method returns the empty list or `undef`. Both are always called in scalar context and should return just a key; the runtime calls `FETCH` itself to get each value.

Two rules keep iteration correct:

- **FIRSTKEY must reset any internal iterator.** When the backing store is a real hash, evaluate `keys %$self` in void or scalar context first - that resets Perl's per-hash each cursor - then call `each`. The throwaway `my $reset = keys %$self` in the example does exactly this. Skip it and a fresh `keys %h` may resume from wherever the last iteration stopped.
- **Signal the end with the empty list.** When the backing store is not a real hash and has no `each` of its own, return the empty list (or `undef`) once you have handed out the last key.

Why SCALAR is worth defining

Without a `SCALAR` method, the runtime guesses at the hash's truth in boolean context, and the guess can be wrong - notably, it may report a hash as non-empty right after you have emptied it by repeated `DELETE`. If `if (%tied_hash)` matters to your callers, define `SCALAR` and return the real answer, as the example does with `scalar %$self`.

Call timing

- `$h{k}` reads $\rightarrow$ `FETCH`; `$h{k} = $v` $\rightarrow$ `STORE`.
- `exists $h{k}` $\rightarrow$ `EXISTS`; `delete $h{k}` $\rightarrow$ `DELETE`.
- `%h = ()` $\rightarrow$ `CLEAR`; `%h = (a => 1)` $\rightarrow$ `CLEAR` then `STORE`.
- `keys %h` / `values %h` / `each %h` $\rightarrow$ `FIRSTKEY` once, then `NEXTKEY` per step. `keys` and `values` in list context call `FETCH` for each key as well.
- `scalar %h`, `if (%h)` $\rightarrow$ `SCALAR` (if defined).
- A hash slice `@h{qw(a b)}` calls `FETCH` once per key; there is no slice hook.

Minimum viable class

A read/write tied hash needs at least `TIEHASH`, `FETCH`, and `STORE`. If callers will ever iterate it, add `FIRSTKEY` and `NEXTKEY` - without them, `keys` and `each` see an empty hash. Add `EXISTS` and `DELETE` if callers use those built-ins; without them, `exists/delete` die at the point of use. `SCALAR` and `CLEAR` round out correctness for boolean tests and bulk emptying.

The easy way: inherit from `Tie::StdHash`

`Tie::StdHash` (shipped inside `Tie::Hash`) implements the entire menu over a blessed hash ref. Inherit from it and override only the hooks that should behave differently:

```

package LoudHash {
    use Tie::Hash;
    our @ISA = ('Tie::StdHash');
    sub STORE { my ($self, $k, $v) = @_; $self->{$k} = uc $v }
}

tie my %loud, 'LoudHash';
$loud{greeting} = 'hi';
say $loud{greeting};           # HI

```

Only STORE changes; FETCH, EXISTS, the iteration protocol, and the constructor all come from the base class. `Tie::Hash` (the parent) is the abstract version that supplies a default TIEHASH and new but leaves the access methods for you to write; `Tie::StdHash` is the concrete one you usually want.

See also

- [tie](#) - the built-in and the full hash method menu
- [tied](#) - reach the backing object for class-specific methods
- [each](#) - the iterator that FIRSTKEY / NEXTKEY implement; its own page documents the end-of-iteration contract
- [exists](#) and [delete](#) - the built-ins behind EXISTS and DELETE
- [Tied scalars](#) - the simpler interface this one builds on
- `Tie::Hash` - the base-class module; `Tie::StdHash` is the concrete version

Tied arrays

A tied array has two layers of interface. The lower layer - FETCH, STORE, FETCHSIZE, STORESIZE, CLEAR - is mandatory and handles element access and resizing. The upper layer - PUSH, POP, SHIFT, UNSHIFT, SPLICE, DELETE, EXISTS - is optional and lets the array respond to the matching built-ins; without a given hook, that built-in either falls back to the lower layer or dies.

The full menu is on the [tie](#) reference page. This chapter builds a class, then explains the size protocol and the optional mutators, with their call timing verified against the running interpreter.

A complete class

A numeric array whose empty slots read back as `0` instead of `undef`, and whose STORE-SIZE pads new slots with `0`:

```

use v5.36;

```

(continues on next page)

(continued from previous page)

```

package ZeroFill {
    sub TIEARRAY { my $class = shift; return bless { data => [] },
    ↪$class }
    sub FETCH    { my ($self, $i) = @_; return $self->{data}[$i] //
    ↪0 }
    sub STORE    { my ($self, $i, $v) = @_; $self->{data}[$i] = $v;
    ↪}
    sub FETCHSIZE { my $self = shift; return scalar @{$self->{data}};
    ↪} }
    sub STORESIZE {
        my ($self, $n) = @_;
        my $data = $self->{data};
        if ($n > @$data) {
            push @$data, (0) x ($n - @$data); # pad with 0, not
    ↪undef
        } else {
            $$data = $n - 1;
        }
    }
    sub CLEAR { my $self = shift; $self->{data} = [] }
    sub PUSH  { my $self = shift; push @{$self->{data}}, @_;
    ↪return scalar @{$self->{data}} }
    sub POP   { my $self = shift; return pop @{$self->{data}} }
}

tie my @a, 'ZeroFill';
$a[0] = 11;
$a[2] = 33; # FETCH on the skipped slot 1 reads back
    ↪0
say scalar @a; # 3
say $a[1];    # 0
$a[4] = 4;    # STORESIZE grows the array, padding with
    ↪0
say "@a";    # 11 0 33 0 0
push @a, 44;
say pop @a;  # 44

```

The object is a blessed hash ref wrapping a real array in its data slot - a common pattern, because the class usually wants room for state beyond the array itself.

The mandatory layer

TIEARRAY classname, LIST

The constructor; returns a blessed reference that becomes the tied object.

FETCH this, index / STORE this, index, value

Single-element read and write. A negative index is translated to a positive one - the runtime calls FETCHSIZE first and adds it to the index - unless the class sets \$NEGATIVE_INDICES to a true value, in which case the raw negative index reaches FETCH and STORE.

FETCHSIZE this

Returns the number of elements. This backs scalar `@array` and the high-water mark for `$#array`. It is the single most-called method on a tied array, since the runtime needs the size to normalise negative indices and to bound iteration.

STORESIZE this, count

Sets the array's length. Growing should fill the new slots with the class's notion of "empty" (here, `0`); shrinking should drop the excess. This backs `$#array = N` and assignment-driven resizing.

CLEAR this

Empties the array - triggered by `@array = ()`.

The size protocol in practice

`FETCHSIZE` and `STORESIZE` are where tied arrays differ most from tied hashes. Three things to keep straight:

- **`$#a` reads through `FETCHSIZE` and writes through `STORESIZE`.** Setting `$#a = 4` calls `STORESIZE($obj, 5)`.
- **Skipped slots are the class's problem.** Assigning `$a[2]` when the array has length 1 calls `STORE($obj, 2, ...)` directly - the runtime does *not* call `STORESIZE` to fill the gap. If a read of slot 1 should return `0` rather than `undef`, `FETCH` has to say so (the `// 0` in the example).
- **EXTEND is advisory.** When the runtime expects an array to grow
 - for instance just before a large assignment - it may call `EXTEND this, count` as a hint to pre-allocate. It is always safe to leave `EXTEND` undefined or empty; it is purely an optimisation opportunity, and making it behave like `STORESIZE` is a bug, because the runtime calls `EXTEND` at times when it does not actually want the visible length to change.

The optional mutators

Each of `push`, `pop`, `shift`, `unshift`, `splice`, `delete`, and `exists` dispatches to the uppercase hook of the same name when the class defines it. Their call timing, confirmed against the interpreter:

- `push @a, 1, 2` → one `PUSH` call with the whole list.
- `pop @a` → `POP`; `shift @a` → `SHIFT`; `unshift @a, 0` → `UNSHIFT`.
- `splice @a, 1, 1, 'x', 'y'` → one `SPLICE` call carrying offset, length, and the replacement list; its return value becomes `splice`'s.
- `exists $a[0]` → `EXISTS`; `delete $a[0]` → `DELETE`.

When a mutator hook is **absent**, behaviour depends on the operator. `Tie::Array` (below) supplies fallbacks for the common five in terms of the mandatory layer; without that base class, calling an unimplemented mutator dies at the point of use.

Call timing

- `$a[$i] → FETCH; $a[$i] = $v → STORE.`
- `scalar @a`, normalising a negative index → `FETCHSIZE`.
- `$#a = $n, @a = (...)` resizing → `STORESIZE` (and `CLEAR` first for whole-array assignment).
- `push/pop/shift/unshift/splice/delete/exists` → the matching uppercase hook when defined.
- An array slice `@a[1, 3]` calls `FETCH` once per index; there is no slice hook.

Minimum viable class

The smallest tied array defines `TIEARRAY`, `FETCH`, `STORE`, `FETCHSIZE`, and `STORESIZE`. With those five, element access, `scalar @a`, `$#a`, and iteration all work. Add `CLEAR` so `@a = ()` behaves. Add the mutators only for the built-ins your callers actually use - or inherit them.

The easy way: inherit from `Tie::Array`

`Tie::Array` implements `PUSH`, `POP`, `SHIFT`, `UNSHIFT`, and `SPLICE` in terms of the mandatory five, so a subclass that supplies `TIEARRAY`, `FETCH`, `STORE`, `FETCHSIZE`, and `STORESIZE` gets all the list mutators for free:

```
package Logging {
    use Tie::Array;
    our @ISA = ('Tie::Array');
    sub TIEARRAY { my $class = shift; return bless { data => [] },
    ↪ $class }
    sub FETCH    { my ($self, $i) = @_; return $self->{data}[$i] }
    sub STORE    { my ($self, $i, $v) = @_; warn "store $i\n";
    ↪ $self->{data}[$i] = $v }
    sub FETCHSIZE { my $self = shift; return scalar @{ $self->{data} }
    ↪ } }
    sub STORESIZE { my ($self, $n) = @_; ${ $self->{data} } = $n -
    ↪ 1 }
}

tie my @log, 'Logging';
push @log, 'x', 'y';      # PUSH inherited; calls STORE twice
    ↪ (warns)
say "@log";               # x y
```

`push @log, 'x', 'y'` works without a `PUSH` method because the inherited one expands into two `STORE` calls - which is why the example warns twice. `Tie::Array` does not define `TIEARRAY` itself, so the subclass must supply the constructor; its default `DELETE` and `EXISTS` simply croak, so override them if your callers need `delete` and `exists`.

See also

- [tie](#) - the built-in and the full array method menu
- [tied](#) - reach the backing object
- [splice](#) - the most intricate mutator the SPLICE hook stands in for
- [push](#) and [pop](#) - the mutators Tie::Array derives from the mandatory layer
- [Tied hashes](#) - the other rich container interface
- Tie::Array - the base-class module that supplies the list mutators

Tied filehandles

A tied filehandle reroutes I/O the way a tied scalar reroutes reads and writes. Print to it and your PRINT runs; read a line and your READLINE runs. The classic uses are redirecting output somewhere unusual (into a string, across a network, through a filter) and embedding Perl in a host program that needs STDOUT and STDERR handled specially.

The interface is larger than the others but mostly optional: you implement only the operations the handle will actually receive. The full menu is on the [tie](#) reference page; this chapter builds a class and explains the hooks and their timing.

A complete class

A write-only handle that collects everything printed to it into a string:

```
use v5.36;

package Collector {
    sub TIEHANDLE { my $class = shift; my $buf = ''; return bless \
        ↪$buf, $class }
    sub PRINT     { my $self = shift; $$self .= join($, // ' ', @_); ↪
        ↪return 1 }
    sub PRINTF    { my $self = shift; my $fmt = shift; $$self .= ↪
        ↪sprintf($fmt, @_); return 1 }
    sub buffer    { my $self = shift; return $$self }
}

tie *OUT, 'Collector';
print OUT "hello ";
print OUT "world\n";
printf OUT "%d items\n", 3;
my $obj = tied *OUT;
print $obj->buffer;
# hello world
# 3 items
```

The object is a blessed scalar ref holding the accumulated text. The extra buffer method is reached through [tied](#), not through any I/O operation - that is the standard way a tie class exposes operations with no filehandle syntax.

Tying the glob, not the scalar

The first argument to `tie` for a handle must be a glob - it begins with an asterisk:

- To tie a named handle: `tie *OUT, 'Collector'`.
- To tie a handle held in a scalar `$fh`: `tie *$fh, 'Collector'`. Writing `tie $fh, ...` ties the *scalar* `$fh`, not the handle inside it - a common and confusing mistake.

The method contract

TIEHANDLE classname, LIST

The constructor; returns a blessed reference of any kind. It can hold whatever internal state the handle needs.

PRINT this, LIST

Backs `print` and `say`. It receives the printed list. Honour `$,` (the output field separator) between items and `$\` (the output record terminator) at the end if you want full fidelity - `print` does not insert them for you.

PRINTF this, format, LIST

Backs `printf`. It receives the format string and arguments; run them through `sprintf` yourself.

WRITE this, scalar, length, offset

Backs `syswrite` - the raw, buffer-oriented write, distinct from `PRINT`.

READLINE this

Backs `<$fh>` and `readline`. In scalar context return the next line or `undef` at end of input; in list context return all remaining lines or an empty list. The returned strings should include the input record separator `$/`, matching what a real handle gives you.

READ this, scalar, length, offset / GETC this

Back `read` / `sysread` and `getc` respectively.

OPEN, CLOSE, EOF, BINMODE, FILENO, SEEK, TELL

Back the corresponding built-ins when used on the handle. Implement the ones that make sense for your backing store and leave the rest out.

DESTROY this / UNTIE this

Optional cleanup, identical in role to the other tie types. A handle class often closes its backing resource here.

Call timing

- `print FH @list` and `say FH @list` → `PRINT`. `say` additionally localises `$\` to `"\n"` around the call, so a `PRINT` that honours `$\` handles `say` with no special case.
- `printf FH $fmt, @args` → `PRINTF`.
- `syswrite FH, $buf, $len, $off` → `WRITE`.
- `<FH> / readline FH` → `READLINE`, in the surrounding context.
- `read/sysread` → `READ`; `getc FH` → `GETC`.

- `close FH`, `eof FH`, `binmode FH`, `seek`, `tell`, `fileno`, `open FH`, ... → the matching uppercase hook when defined.

A note on `STDERR`: when `STDERR` is tied, its `PRINT` is what issues warnings and error messages. The runtime disables the tie for the duration of that call, so you can warn from inside `PRINT` without triggering infinite recursion.

Minimum viable class

There is no single mandatory method beyond `TIEHANDLE` - you implement whichever operations the handle will receive. An output-only handle needs `TIEHANDLE` plus `PRINT` (and usually `PRINTF`). An input-only handle needs `TIEHANDLE` plus `READLINE` (and perhaps `GETC` / `READ`). Operations whose hook is missing die at the point of use, so define exactly the set your callers exercise.

The easy way: inherit from `Tie::StdHandle`

`Tie::StdHandle` (shipped inside `Tie::Handle`) ties a handle onto a real underlying one: its `OPEN` opens an actual file and the I/O hooks forward to it. Inherit from it and override only the operations you want to intercept:

```
package Doubler {
    use Tie::StdHandle;
    our @ISA = ('Tie::StdHandle');
    sub READLINE {
        my $self = shift;
        my $line = $self->SUPER::READLINE;
        return defined $line ? $line . $line : undef; # echo each
        ↪line twice
    }
}

my $data = "a\nb\n";
tie *DUP, 'Doubler';
tied(*DUP)->OPEN('<', \ $data);
print while defined($_ = <DUP>);
# a
# a
# b
# b
```

`READLINE` calls up to the base class for the real next line, then doubles it; every other operation flows through `Tie::StdHandle` unchanged. `Tie::Handle` (the parent) is the abstract base that documents the interface and supplies a default `PRINT`/`PRINTF` in terms of `WRITE`; `Tie::StdHandle` is the concrete, ready-to-use one.

See also

- `tie` - the built-in and the full handle method menu
- `tied` - the only way to reach a handle class's non-I/O methods, like `buffer` above
- `print` and `printf` - the built-ins behind `PRINT` and `PRINTF`

- [readline](#) - its page documents the scalar/list return contract `READLINE` must honour
- [Tied scalars](#) - the simplest tie type, a good warm-up for the handle interface
- `Tie::Handle` - the base-class module; `Tie::StdHandle` is the concrete version

4.9 Coming from another language

You already know how to program. You reach for a hash, a loop, a function, a regex without thinking - in some other language. These guides teach Perl by mapping what you already do onto how Perl does it, one construct at a time. Your dictionary is a Perl hash; here is the syntax, and here is the one place the analogy leaks.

Each guide assumes you are **fluent in the source language and new to Perl**. It does not teach the source language back to you. It teaches the Perl side of each pairing: the syntax, the idiom, and - most useful of all - the gotcha that the surface similarity hides.

Every guide in this series covers the **same topics in the same order**, so the mapping is comparable across languages and the series grows cleanly as new languages are added. The order moves from what every language shares (variables, strings, numbers) through Perl's distinctive shapes (sigils, the array/hash split, and **context** - the one idea no other language has), into structure (subroutines, references, modules, objects), and out through the working surface (regex, I/O, error handling, idioms), ending with a per-language collection of the traps that actually bite.

4.9.1 Which guide you want

- [Python to Perl](#) - sigils where Python has none, the array/hash split where Python has list and dict, comprehensions rewritten as map/grep, and the context concept Python has no word for.
- [PHP to Perl](#) - you already have \$, but in Perl it shifts by access, PHP's one array splits into @array and %hash, and . versus + (and == versus eq) stop being interchangeable.
- [Ruby to Perl](#) - blocks become Perl blocks and closures, method-calls-on-everything become plain functions, and Ruby's "only nil and false are falsy" inverts into Perl's truthiness.
- [Lua to Perl](#) - the one table type splits into @array and %hash, indexing turns 0-based, Lua patterns become real regex, and multiple-return-values are your bridge to Perl's context.
- [Sed to Perl](#) - the same s/// you already know, sed's y/// becomes tr///, address ranges become the /a/ . . /b/ flip-flop, and the hold space becomes an ordinary Perl variable.
- [Awk to Perl](#) - Perl grew out of awk, so almost everything maps directly: fields become @F and split, NR becomes \$., associative arrays become hashes, and BEGIN/END keep their names.
- [Shell to Perl](#) - for when a shell script outgrew itself: \$(cmd) becomes qx{}, the []/-f file tests carry straight over, and the \${var#...} parameter expansions become real regex.

- *Tcl to Perl* - a carry-over for Tcl maintainers: `set x` becomes `my $x`, `[expr {}]` disappears into native operators, lists and array become Perl arrays and hashes, and `regsub` becomes `s///`.

4.9.2 The one concept to read first

If you read only one section of whichever guide matches your background, read **Context**. Every other language in this series collapses a collection to a count, a first element, or a boolean with explicit syntax (`len(x)`, `x[0]`, `bool(x)`). Perl does it through *context*: the same expression yields a list in one place and a single scalar in another, decided by where you wrote it. It is the source of most “why did my array become the number 3?” confusion, and once it clicks the rest of Perl falls into place.

4.9.3 A note on the runtime

These guides target `pperl`, a faithful Rust reimplementation of Perl 5.42. Every Perl example that runs as a script was executed on `pperl` and shows real output. The text-processing guides (`sed`, `awk`, `shell`) also show the classic `perl -ne/perl -pe` one-liner forms, which are the idiomatic Perl equivalent; the implicit-loop switches behind them are not yet recognised by this `pperl` build, so those one-liners are shown for reference and paired with an explicit-loop script you can run today.

Python to Perl

You know Python. You write list comprehensions in your sleep, you reach for a dict without thinking, and `for x in xs:` is muscle memory. This guide maps each of those habits onto Perl, so you can be productive in `pperl` in an afternoon instead of relearning programming from scratch.

Two shocks land in the first five minutes, so meet them now:

- **Perl uses braces and semicolons, not indentation.** There is no significant whitespace. A block is `{ ... }`; a statement ends with `;`. Indent for your own sanity, but the parser ignores it.
- **Variables wear sigils.** A scalar is `$x`, an array is `@x`, a hash is `%x`. The sigil is part of how you read the variable, not just its name. This is the single biggest surface difference from Python, and section one is about nothing else.

Every example here runs on `pperl`, a Rust reimplementation of Perl 5.42. Each Perl snippet was executed on `pperl`; the `# ...` comments show its real output. Start your scripts with:

```
use v5.42;
```

That one line turns on `strict` (no accidental globals), warnings, the `say` built-in (a `print` with a trailing newline, like Python’s `print`), subroutine signatures, and the modern feature set used throughout this guide.

How to read this

- *Variables and sigils* - the \$/@/% shift.
- *Strings* - interpolation and concatenation.
- *Numbers* - one surface type, float division.
- *Arrays and lists* - your list, plus map/grep.
- *Hashes* - your dict.
- *Context* - the concept Python has no word for. Read this one.
- *Truthiness* - what counts as false, with surprises.
- *Control flow* - if, loops, the postfix forms.
- *Subroutines* - def becomes sub.
- *References* - nested data and closures.
- *Regular expressions* - regex as syntax, not a library.
- *File and stream I/O* - open, line reads, slurp.
- *Modules and packages* - import becomes use.
- *Objects* - the modern class syntax.
- *Error handling* - try/except becomes eval or try/catch.
- *Idioms* - the daily one-liners.
- *Gotchas coming from Python* - scan before you write.

Variables and sigils

In Python a name just names a value. In Perl the **sigil tells you the shape**: \$ for a single value, @ for an ordered list, % for a key/value map. `my` declares a lexically scoped variable, the Perl equivalent of an ordinary local binding.

```
name = "Ada"
langs = ["Python", "Perl"]
ages = {"Ada": 36, "Linus": 56}
```

```
use v5.42;
my $name = "Ada";
my @langs = ("Python", "Perl");
my %ages = (Ada => 36, Linus => 56);
say $name;           # Ada
say $langs[0];       # Python
say $ages{Ada};      # 36
```

The gotcha: when you reach *into* an aggregate for one element, the sigil follows the value you get back, not the container. A single element of `@langs` is a scalar, so it is `$langs[0]`, not `@langs[0]`; one value from `%ages` is `$ages{Ada}`. The `@/%` form is the whole collection; the `$` form is one piece of it.

Strings

Double quotes interpolate variables, single quotes do not. There is no f-string because the default string is an f-string. Concatenation is `.`, not `+` (which is reserved for numbers).

```
name = "Ada"
greeting = f"Hello, {name}"
shout = "AB" * 3
```

```
use v5.42;
my $name = "Ada";
my $greeting = "Hello, $name";    # Hello, Ada
my $literal = 'Hello, $name';    # Hello, $name (no interpolation)
my $joined = "AB" . "CD";        # ABCD
my $shout = "AB" x 3;            # ABABAB
```

The gotcha: string repetition is `x`, not `*`, and the join operator is `.`, not `+`. Mixing them up is the most common first-day typo for a Python developer. Array elements and hash values interpolate inside double quotes too: `"First: $langs[0]"` works as written.

Numbers

Perl has one numeric type at the surface; you do not declare `int` vs `float`. Division always produces a float, exactly like Python 3.

```
print(7 / 2)      # 3.5
print(7 // 2)     # 3
print(2 ** 10)    # 1024
print(10 % 3)     # 1
```

```
use v5.42;
say 7 / 2;        # 3.5
say int(7 / 2);   # 3
say 2 ** 10;      # 1024
say 10 % 3;       # 1
```

The gotcha: comparison and equality operators come in two families, and you pick by the *operands' type*, not the variables'. Numbers use `==`, `!=`, `<`, `>`, `<=>`; strings use `eq`, `ne`, `lt`, `gt`, `cmp`. `"10" == "10.0"` is true (numeric), but `"10" eq "10.0"` is false (string). Using `==` on strings, or `eq` on numbers, is a frequent and quiet bug.

Arrays and lists

A Perl array is your Python list. The verbs are renamed but the operations are the same, and negative indices work as you expect.

```
nums = [1, 2, 3, 4, 5]
nums.append(6)
last = nums.pop()
print(len(nums))    # 5
print(nums[1:3])    # [2, 3]
```

```

use v5.42;
my @nums = (1, 2, 3, 4, 5);
push @nums, 6;
my $last = pop @nums;      # 6
say scalar @nums;          # 5
say "@nums[1,2]";         # 2 3   (a slice)

```

Comprehensions become map and grep. map transforms (Python's `[f(x) for x in xs]`); grep filters (Python's `[x for x in xs if p(x)]`). `$_` is the current element, Perl's implicit loop variable.

```

squares = [x * x for x in nums]
evens    = [x for x in nums if x % 2 == 0]
desc     = sorted(nums, reverse=True)

```

```

use v5.42;
my @nums      = (1, 2, 3, 4, 5);
my @squares   = map { $_ * $_ } @nums;    # 1 4 9 16 25
my @evens     = grep { $_ % 2 == 0 } @nums; # 2 4
my @desc      = sort { $b <=> $a } @nums;  # 5 4 3 2 1

```

The gotcha: sort compares as strings by default, so `sort (10, 9, 100)` gives `10 100 9`. For numbers you must pass a comparator: `sort { $a <=> $b } @nums`. `$a` and `$b` are the two elements being compared, and `<=>` is the numeric three-way operator (`cmp` is its string sibling).

Hashes

A Perl hash is your Python dict. Declared with `%`, accessed one value at a time with `$h{key}`.

```

h = {"apple": 3, "pear": 5, "plum": 2}
print(h["apple"])           # 3
print(sorted(h.keys()))     # ['apple', 'pear', 'plum']
print("pear" in h)         # True
del h["plum"]
for k in sorted(h):
    print(k, "=>", h[k])

```

```

use v5.42;
my %h = (apple => 3, pear => 5, plum => 2);
say $h{apple};              # 3
say join ", ", sort keys %h; # apple,pear,plum
say exists $h{pear} ? "yes" : "no"; # yes
delete $h{plum};
for my $k (sort keys %h) {
    say "$k => $h{$k}";
}

```

The gotcha: membership is `exists $h{key}`, not Python's `in`, and the fat comma `=>` is the idiomatic key/value separator. `=>` also auto-quotes the bareword on its left, so `apple`

=> 3 needs no quotes around apple. A hash has no order; iterate sort keys %h when you want a stable one.

Context

This is the chapter to read twice. Python has no equivalent, and almost every “Perl is weird” story traces back to it.

Every expression in Perl runs in either **list context** or **scalar context**, decided by where it appears, not by what it is. The same array behaves differently in each. Assigning an array to a *scalar* gives its length; assigning it to a *list* (a parenthesised set of variables) copies elements.

```
use v5.42;
my @list = (10, 20, 30);

my $count = @list;      # scalar context -> 3 (the length)
say $count;             # 3

my ($first) = @list;    # list context -> first element copied
say $first;            # 10

my ($a, $b) = @list;    # 10 and 20
say "$a $b";           # 10 20
```

The parentheses on the left are the whole difference. `my $x = @list` asks “how many?” and gets 3. `my ($x) = @list` asks “give me the elements” and gets 10. A Python developer reads both as “assign the list,” which is exactly the trap.

A subroutine can ask which context it was called in, with `wantarray`, and return different things accordingly:

```
use v5.42;
sub items {
    return wantarray ? (1, 2, 3) : "three items";
}

my @got = items();      # list context
my $msg = items();      # scalar context
say "@got";             # 1 2 3
say $msg;               # three items
```

Built-ins use this too. `scalar @array` forces scalar context to get a count; a regex match with `/g` returns the list of matches in list context and a true/false in scalar context. The classic idiom for “count the matches” forces a list assignment in scalar context with the so-called *countof* form:

```
use v5.42;
my $n = () = "a1b2c3" =~ /\d/g;
say $n;      # 3
```

The gotcha: there is no single answer to “what does this array do” - it depends on the surrounding expression. When something returns a number you expected to be a list

(or vice versa), context is the first suspect. Reach for `scalar` to force a count and parentheses on the left to force element copying.

Truthiness

Perl's false values are a short, specific list: the number `0`, the empty string `"`, the string `"0"`, and `undef` (Perl's `None`). **Everything else is true** - and that includes some strings a Python developer assumes are false.

```
use v5.42;
for my $v (0, 1, "", "0", "00", "0.0", "false") {
    say "'$v' is ", ($v ? "true" : "false");
}
```

This prints, exactly:

```
'0' is false
'1' is true
'' is false
'0' is false
'00' is true
'0.0' is true
'false' is true
```

The gotcha: `"00"` and `"0.0"` are **true** in Perl, even though they look numerically zero, because only the exact string `"0"` is false. The string `"false"` is true (it is a non-empty, non-`"0"` string). And unlike Python, an empty array or empty hash is not itself a false *value* - in boolean context an array yields its length, so `if (@list)` is true when the list is non-empty, which is the test you want. For a “use this if defined, else a default” choice, use defined-or `//`, which falls back only on `undef` (not on `0` or `"`):

```
use v5.42;
my %cfg = (host => "localhost");
my $port = $cfg{port} // 8080;    # key absent -> undef -> 8080
say $port;                      # 8080
```

Control flow

The keywords are familiar; the spelling differs. `elif` is `elsif`, and there is no `:` and no indentation - blocks are braced.

```
if n < 0:
    sign = "neg"
elsif n == 0:
    sign = "zero"
else:
    sign = "pos"

for x in nums:
    print(x)
```

(continues on next page)

(continued from previous page)

```
while running:
    step()
```

```
use v5.42;
my $sign;
if ($n < 0) { $sign = "neg" }
elsif ($n == 0) { $sign = "zero" }
else { $sign = "pos" }

for my $x (@nums) { say $x }

while ($running) { step() }
```

Perl adds two things Python does not have. **Postfix modifiers** let a simple statement carry its own condition or loop, which reads naturally for one-liners:

```
use v5.42;
say $_ for @nums;           # loop, one statement
say "negative" if $n < 0;    # guarded statement
$sum += $_ for @nums;       # accumulate
```

And unless/until are the negated forms of if/while, for when the positive phrasing reads awkwardly: say "ok" unless \$error.

The gotcha: the ternary chains the same as Python's, but with ?/:, not if/else: my \$sign = \$n < 0 ? "neg" : \$n == 0 ? "zero" : "pos". The C-style ?: is the idiom; there is no x if c else y form.

Subroutines

def becomes sub. Modern Perl has signatures that look much like Python's parameter lists, including defaults.

```
def greet(name, greeting="Hello"):
    return f"{greeting}, {name}!"

def total(*nums):
    return sum(nums)
```

```
use v5.42;
sub greet ($name, $greeting = "Hello") {
    return "$greeting, $name!";
}
say greet("Ada");           # Hello, Ada!
say greet("Linus", "Hi");   # Hi, Linus!

sub total (@nums) {
    my $sum = 0;
    $sum += $_ for @nums;
```

(continues on next page)

(continued from previous page)

```

    return $sum;
}
say total(1, 2, 3, 4);      # 10

```

Keyword arguments map to a hash parameter, since `key => value` pairs flatten into a list that a `%opts` signature gathers up:

```

use v5.42;
sub connect_to (%opts) {
    return "host=$opts{host} port=$opts{port}";
}
say connect_to(host => "localhost", port => 8080);
# host=localhost port=8080

```

The gotcha: a sub returns the value of its last expression even without `return`, and the return is **context-sensitive** (see [Context](#)). If you return `@list`, the caller gets the list in list context but the *count* in scalar context. Be deliberate about what you return when it might be used either way.

References

Perl's arrays and hashes do not nest directly; you nest them through **references**, which are the Perl analogue of Python's object handles (in Python every container is already a reference, so this step is invisible to you there). A reference is a scalar that points at an aggregate. Take one with `\`, follow it with `->`.

```

data = {"name": "Ada", "langs": ["Perl", "Python"]}
print(data["langs"][1])      # Python
data["langs"].append("Ruby")

```

```

use v5.42;
my $data = { name => "Ada", langs => ["Perl", "Python"] };
say $data->{name};            # Ada
say $data->{langs}[1];        # Python
push $data->{langs}->@*, "Ruby"; # postfix deref to push
say "@{$data->{langs}}";      # Perl Python Ruby

```

`{ ... }` builds an anonymous hash reference and `[ ... ]` an anonymous array reference - the literal building blocks of nested data, matching Python's `{}` and `[]` literals. The `->` arrow follows a reference one step; between two subscripts it is optional, so `$data->{langs}[1]` and `$data->{langs}->[1]` are the same.

Anonymous subs are closures, exactly like Python's `lambda` but with no single-expression limit:

```

use v5.42;
my $double = sub ($n) { $n * 2 };
say $double->(21);          # 42

```

The gotcha: the postfix deref forms (`->@*`, `->%*`, `->$*`) are the modern, readable way to dereference a whole aggregate, and `"@{ ... }"` is how you interpolate a dereferenced

array into a string. The older `@$ref` and `$$ref{key}` forms still work and appear in existing code, but reach for the arrow and postfix forms in new code.

Regular expressions

In Python regex is the `re` library. In Perl it is **syntax**: `=~` binds a pattern to a string, `m//` matches, `s///` substitutes. Capture groups land in `$1`, `$2`, and named captures in `%+`.

```
import re
m = re.search(r"(\d{4})-(\d{2})-(\d{2})", s)
if m:
    year, month, day = m.groups()
t = re.sub(",", ";", "a,b,c")
parts = "one,two,three".split(",")
```

```
use v5.42;
my $s = "2026-06-20";
if ($s =~ /(\d{4})-(\d{2})-(\d{2})/) {
    say "$1 $2 $3";          # 2026 06 20
}
if ($s =~ /(<year>\d{4})/) {
    say "${year}";          # 2026
}
(my $t = "a,b,c") =~ s/,/,/g;    # a;b;c
my @parts = split /,/, "one,two,three"; # one two three
```

A `/g` match in list context returns *all* captures, which is the direct analogue of `re.findall`:

```
use v5.42;
my @nums = "x1y2z3" =~ /(\d)/g;    # 1 2 3
say "@nums";
```

The gotcha: `s///` modifies its target *in place* and returns the number of substitutions, not a new string. That is why the example copies first with `(my $t = "a,b,c")` and then substitutes on the copy - otherwise you would be editing a string literal. To get a new string without touching the original, use the `/r` flag: `my $t = $s =~ s/,/,/gr`.

File and stream I/O

The idiom is the three-argument `open` with a lexical filehandle, the direct counterpart of Python's with `open(...)` as `f`:. The angle-bracket operator `<$fh>` reads a line, the same role as iterating a Python file object.

```
with open("/tmp/sample.txt", "w") as f:
    for i in range(1, 4):
        f.write(f"line {i}\n")

with open("/tmp/sample.txt") as f:
    for line in f:
        print("got:", line.rstrip())
```

```
use v5.42;
my $path = "/tmp/sample.txt";

open my $wh, '>', $path or die "cannot write: $!";
print {$wh} "line $_\n" for 1 .. 3;
close $wh;

open my $rh, '<', $path or die "cannot read: $!";
while (my $line = <$rh>) {
    chomp $line;
    say "got: $line";          # got: line 1 ... got: line 3
}
close $rh;
```

To slurp every line at once, read the handle in list context:

```
use v5.42;
open my $fh, '<', "/tmp/sample.txt" or die $!;
my @lines = <$fh>;          # all lines, including newlines
say scalar @lines;          # 3
```

The gotcha: a line read with <\$fh> keeps its trailing newline; call `chomp` to strip it (Python's `line.rstrip()` reflex). And `open` returns false on failure rather than raising, so the `or die "...: $!"` idiom is mandatory - `$!` is the system error message, Perl's `errno`.

Modules and packages

`import` becomes `use`. The namespace separator is `::`, not `.`, and a module `Foo::Bar` lives at `Foo/Bar.pm` on the include path `@INC` (Perl's `sys.path`). A module's import list is a selection of names, much like `from module import name`.

```
from math import sqrt
import json
data = json.loads(text)
```

```
use v5.42;
use List::Util qw(sum0 max first);
say sum0(1 .. 10);          # 55
say max(3, 7, 2);           # 7
say first { $_ % 2 == 0 } (1, 3, 4, 5); # 4
```

The standard library is large and much of what you would reach to PyPI for ships in core or on CPAN, the Perl package archive. `List::Util`, `Scalar::Util`, `POSIX`, `File::Spec`, and `Data::Dumper` are the everyday ones.

The gotcha: `use` runs at *compile time*, not at the point in the file where it appears, so the order of `use` lines relative to your code does not matter the way `import` placement does. Importing is also opt-in per name: `use List::Util qw(sum0 max)` brings exactly `sum0` and `max` into your namespace, nothing else.

Objects

For new code, Perl 5.42 has a first-class `class` syntax that will look familiar coming from Python. Fields are declared with `field`, methods with `method`, and `$self` is available inside every method just as `self` is in Python (you do not list it as a parameter).

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def to_string(self):
        return f"({self.x}, {self.y})"
    def move(self, dx, dy):
        self.x += dx
        self.y += dy
        return self

p = Point(x=3, y=4)
print(p.to_string())      # (3, 4)
p.move(1, 1)
print(p.to_string())      # (4, 5)
```

```
use v5.42;
use feature 'class';

class Point {
    field $x :param = 0;
    field $y :param = 0;
    method to_string { return "($x, $y)" }
    method move ($dx, $dy) {
        $x += $dx;
        $y += $dy;
        return $self;
    }
}

my $p = Point->new(x => 3, y => 4);
say $p->to_string;      # (3, 4)
$p->move(1, 1);
say $p->to_string;      # (4, 5)
```

The constructor is generated for you: `:param` marks a field that the auto-built `new` accepts as a named argument, and `= 0` is its default. Inheritance is `:isa`:

```
use v5.42;
use feature 'class';

class Point3D :isa(Point) {
    field $z :param = 0;
    method z { return $z }
}
```

(continues on next page)

(continued from previous page)

```
my $q = Point3D->new(x => 1, y => 2, z => 3);
say $q->to_string, " z=", $q->z;    # (1, 2) z=3
```

The gotcha: the class feature is still marked experimental in Perl 5.42, and on pperl the “class is experimental” notices print to **stderr** even with no warnings 'experimental::class' in scope. They do not touch your program’s output, so a script that prints to stdout behaves exactly as shown above; just expect the notices on the error stream. For the older bless-based object model you will meet in existing code, and for roles, inheritance details, and migration, see the *Object-oriented Programming guide*.

Error handling

Python’s try/except has two counterparts. The classic one wraps code in `eval { }` and inspects `$@` (the caught error) afterward:

```
try:
    risky(-1)
except Exception as e:
    print("caught:", e)
```

```
use v5.42;
my $result = eval { risky(-1) };
if ($@) {
    print "caught: $@";    # caught: negative!
}
```

The modern form is try/catch, which reads almost exactly like Python’s:

```
use v5.42;
use feature 'try';

try {
    risky(-4);
}
catch ($e) {
    print "caught: $e";    # caught: negative!
}
```

where the raising side is die:

```
use v5.42;
sub risky ($n) {
    die "negative!\n" if $n < 0;
    return sqrt $n;
}
```

The gotcha: die with a string that ends in `\n` produces exactly that message; die with a string that does *not* end in a newline gets " at SCRIPT line N.\n" appended automatically. End your error strings with `\n` when you want a clean message and omit

it when you want the file-and-line trace. Note also the two error variables: `$@` is the last caught exception (Python's `except ... as e`), while `$!` is the last *system* error (a failed open, the `errno` string).

Idioms

A few patterns you will see daily and want in your fingers.

Sort a list by a computed key (the Schwartzian transform - Python's `sorted(xs, key=f)` done by hand, decorate then sort then undecorate):

```
use v5.42;
my @words = qw(apple fig pear);
my @by_length =
    map { $_->[1] }
    sort { $a->[0] <=> $b->[0] }
    map { [ length $_, $_ ] } @words;
say "@by_length";           # fig pear apple
```

Build a hash from a list, the dict-comprehension analogue (`{w: len(w) for w in words}`):

```
use v5.42;
my @words = qw(apple pear fig);
my %len = map { $_ => length $_ } @words;
say "$_ => $len{$_}" for sort keys %len;
# apple => 5
# fig => 3
# pear => 4
```

And `ppperl` is a command-line tool in the `awk/sed` niche. The implicit-loop one-liner is the Perl developer's everyday reflex:

```
ppperl -lane '$s += $F[-1]; END { print $s }' data.txt
```

The gotcha: read the *One-Liners guide* for the switches (`-n`, `-p`, `-a`, `-l`, `-e`) and the dozens of recipes built on them; it is one of the highest-leverage things to learn coming from a shell-and-Python background.

Gotchas coming from Python

The traps, collected for a final scan before you start writing:

- **Sigils change with usage.** One element of `@a` is `$a[0]`; one value of `%h` is `$h{k}`. The container sigil and the element sigil differ.
- **. joins strings, + adds numbers.** Never `+` two strings to concatenate; that coerces them to numbers and warns.
- **Two comparison families.** `==/</<=>` for numbers, `eq/lt/ cmp` for strings. Pick by operand type, not by variable.
- **sort is string-sorted by default.** Pass `{ $a <=> $b }` for numeric order.

- **Context.** `my $x = @a` is the *length*; `my ($x) = @a` is the *first element*. The parentheses are the whole difference. Re-read [Context](#) when a value comes back the wrong shape.
- **Truthiness surprises.** `"0"` is false, but `"00"` and `"0.0"` are **true**. Only `0`, `"`, `"0"`, and `undef` are false.
- **s/// edits in place** and returns a count; use the `/r` flag to get a new string instead.
- **open does not raise.** Use `open ... or die "...: $!"`.
- **chomp your line reads.** `<$fh>` keeps the trailing newline.
- **Non-ASCII source needs use utf8;** Under `use v5.42`; a literal non-ASCII byte in your source is a compile error without it. Keep string data ASCII unless you specifically add `use utf8;`.
- **The class feature prints experimental notices to stderr** on `pperl`. Correct output, noisy error stream.

PHP to Perl

You know PHP. You reach for `$arr['key']` without thinking, `foreach ($items as $item)` is muscle memory, and you join strings with `.` because of course you do. Perl and PHP are close cousins: both are sigil-using, pragmatic, get-the-job-done languages, and PHP borrowed heavily from Perl in its early days. This guide maps your PHP habits onto Perl so you can be productive in `pperl` in an afternoon.

Two shocks land in the first five minutes, so meet them now:

- **The \$ sigil is not the whole story.** In PHP every variable is `$x`, always. In Perl the sigil changes with how you *access* the value: `@a` is the whole array but `$a[0]` is one element; `%h` is the whole hash but `$h{k}` is one value. The dollar you already know marks a single scalar; `@` and `%` are new and section one is about nothing else.
- **PHP's one array becomes Perl's two types.** A PHP array is an ordered map that does duty as both list and dictionary. Perl splits that into `@array` (integer-indexed) and `%hash` (string-keyed), and you choose up front which one you have. This split drives the [Arrays](#) and [Hashes](#) sections.

Every example here runs on `pperl`, a Rust reimplementation of Perl 5.42. Each Perl snippet was executed on `pperl`; the `# ...` comments show its real output. Start your scripts with:

```
use v5.42;
```

That one line turns on `strict` (no accidental globals), warnings, the `say` built-in (a `print` with a trailing newline, like PHP's `echo` plus `PHP_EOL`), subroutine signatures, and the modern feature set used throughout this guide.

How to read this

- [Variables and sigils](#) - the `$/@/%` shift.
- [Strings](#) - interpolation and `.` concatenation.
- [Numbers](#) - `==` vs `eq`, the classic PHP trap.

- *Arrays and lists* - half of your array.
- *Hashes* - the other half of your array.
- *Context* - the concept PHP has no word for. Read this one.
- *Truthiness* - what counts as false, with surprises.
- *Control flow* - if, loops, the postfix forms.
- *Subroutines* - function becomes sub.
- *References* - nested data and closures.
- *Regular expressions* - regex as syntax, not preg_*.
- *File and stream I/O* - fopen, line reads, slurp.
- *Modules and packages* - require/use becomes use.
- *Objects* - the modern class syntax.
- *Error handling* - try/catch, the same idea.
- *Idioms* - the daily one-liners.
- *Gotchas coming from PHP* - scan before you write.

Variables and sigils

You already use \$, so the first half of Perl's sigils is free. The new idea is that **the sigil marks the access, not the variable**. The whole array is @a; one element is \$a[0]. The whole hash is %h; one value is \$h{k}. PHP keeps \$ on everything and varies the brackets (\$a[0], \$h['k']); Perl varies the sigil instead. my declares a lexically scoped variable, the equivalent of an ordinary local binding.

```
$name = "Ada";
$langs = ["PHP", "Perl"];
$ages = ["Ada" => 36, "Rasmus" => 56];
```

```
use v5.42;
my $name = "Ada";
my @langs = ("PHP", "Perl");
my %ages = (Ada => 36, Rasmus => 56);
say $name;           # Ada
say $langs[0];       # PHP
say $ages{Ada};      # 36
say scalar @langs;   # 2
```

The gotcha: when you reach *into* an aggregate for one element, the sigil follows the value you get back, not the container. A single element of @langs is a scalar, so it is \$langs[0], not @langs[0]; one value from %ages is \$ages{Ada}. Read the leading sigil as “what shape am I getting” - \$ for one thing, @ for a list, % for the whole map - and the brackets as “from where.”

Strings

Concatenation is `.`, exactly as in PHP. Double quotes interpolate variables, single quotes do not - the same split PHP makes between `"..."` and `'...'`. There is no `sprintf`-into-the-string syntax to learn because the default double-quoted string already interpolates.

```
$name = "Ada";
$greeting = "Hello, $name";
$literal = 'Hello, $name';
$joined = "AB" . "CD";
$shout = str_repeat("AB", 3);
```

```
use v5.42;
my $name = "Ada";
my $greeting = "Hello, $name";    # Hello, Ada
my $literal = 'Hello, $name';    # Hello, $name (no interpolation)
my $joined = "AB" . "CD";        # ABCD
my $shout = "AB" x 3;            # ABABAB
```

The gotcha: string repetition is the `x` operator, not a function - `"AB" x 3`, not `str_repeat`. Interpolating an array or hash element needs no curly-brace escape: `"First: $langs[0]"` and `"Name: $h{name}"` both work as written, where PHP would make you write `"${$arr['k']}"`. To interpolate the result of an *expression*, wrap it in `@{ [ ... ] }`: `"sum: @{ [ 2 + 3 ] }"` prints `sum: 5`.

Numbers

Perl has one numeric type at the surface; you do not declare `int` vs `float`. The thing that will actually bite you is operators, so read the gotcha twice.

```
echo 7 / 2;           // 3.5
echo intdiv(7, 2);    // 3
echo 2 ** 10;         // 1024
echo 10 % 3;          // 1
```

```
use v5.42;
say 7 / 2;            # 3.5
say int(7 / 2);       # 3
say 2 ** 10;          # 1024
say 10 % 3;           # 1
```

The gotcha: Perl has **two operator families and you pick by operand type, not by the variable**. Numbers use `==`, `!=`, `<`, `>`, `<=>`; strings use `eq`, `ne`, `lt`, `gt`, `cmp`. And `+` is *always* numeric - there is no operator overloading the way PHP's `+` does array union. Concatenate with `.`, add with `+`, and never cross them:

```
use v5.42;
say "12" . "3";      # 123    (string concat)
say "12" + "3";      # 15     (numeric add)
```

(continues on next page)

(continued from previous page)

```
say "10" == "10.0" ? "same" : "diff"; # same (numeric compare)
say "10" eq "10.0" ? "same" : "diff"; # diff (string compare)
```

PHP retired the loose `==` mess in modern versions; Perl never had it, but it solves the same problem by making you say which comparison you mean. `"10" == "10.0"` is true (numeric), `"10" eq "10.0"` is false (string). Using `==` on strings, or `eq` on numbers, is the single most common PHP-to-Perl bug.

Arrays and lists

Here is where PHP's one array splits in two. When you used an array as a numerically indexed list - `$items[0]`, `array_push`, `foreach` over values - that is a Perl **array**, declared with `@`. The verbs are renamed but the operations are the same, and negative indices work as you expect.

```
$nums = [1, 2, 3, 4, 5];
$nums[] = 6;
$last = array_pop($nums);
echo count($nums);           // 5
print_r(array_slice($nums, 1, 2)); // [2, 3]
```

```
use v5.42;
my @nums = (1, 2, 3, 4, 5);
push @nums, 6;
my $last = pop @nums;      # 6
say scalar @nums;          # 5
say "@nums[1,2]";         # 2 3 (a slice)
```

Your `array_map` and `array_filter` become `map` and `grep`. `map` transforms (PHP's `array_map`); `grep` filters (PHP's `array_filter`). `$_` is the current element, Perl's implicit loop variable, the way the callback's parameter is in PHP - except you do not name it.

```
$squares = array_map(fn($x) => $x * $x, $nums);
$evens   = array_filter($nums, fn($x) => $x % 2 == 0);
rsort($nums); // descending, in place
```

```
use v5.42;
my @nums   = (1, 2, 3, 4, 5);
my @squares = map { $_ * $_ } @nums; # 1 4 9 16 25
my @evens  = grep { $_ % 2 == 0 } @nums; # 2 4
my @desc   = sort { $b <=> $a } @nums; # 5 4 3 2 1
```

The gotcha: `sort` compares as strings by default, so `sort (10, 9, 100)` gives `10 100 9`. For numbers you must pass a comparator: `sort { $a <=> $b } @nums`. `$a` and `$b` are the two elements being compared, and `<=>` is the numeric three-way operator (`cmp` is its string sibling). Note also that `grep` keeps elements, returning a new list, where PHP's `array_filter` preserves keys - Perl arrays have no keys to preserve, so the result is simply re-indexed.

Hashes

The other half of PHP's array - the part you used with string keys, `$config['host']`, `array_key_exists`, `foreach ($h as $k => $v)` - is a Perl **hash**, declared with `%` and accessed one value at a time with `$h{key}`.

```
$h = ["apple" => 3, "pear" => 5, "plum" => 2];
echo $h["apple"];           // 3
$keys = array_keys($h);
sort($keys);
echo array_key_exists("pear", $h) ? "yes" : "no"; // yes
unset($h["plum"]);
foreach ($h as $k => $v) {
    echo "$k => $v\n";
}
```

```
use v5.42;
my %h = (apple => 3, pear => 5, plum => 2);
say $h{apple};           # 3
say join ", ", sort keys %h; # apple,pear,plum
say exists $h{pear} ? "yes" : "no"; # yes
delete $h{plum};
for my $k (sort keys %h) {
    say "$k => $h{$k}"; # apple => 3 ... pear => 5
}
```

The gotcha: membership is `exists $h{key}` (the key is present) or `defined $h{key}` (the key is present *and* its value is not undef) - between them they cover PHP's `array_key_exists` and `isset`. The fat comma `=>` is the idiomatic key/value separator and auto-quotes the bareword on its left, so `apple => 3` needs no quotes around `apple`. A Perl hash has **no insertion order** (unlike a PHP array, which keeps it); iterate `sort keys %h` when you want a stable order. To walk key and value together, PHP's `foreach ($h as $k => $v)` becomes either `for my $k (keys %h)` or, when you want both at once, `while (my ($k, $v) = each %h)`.

Context

This is the chapter to read twice. PHP has no equivalent, and almost every “Perl is weird” story traces back to it.

Every expression in Perl runs in either **list context** or **scalar context**, decided by where it appears, not by what it is. The same array behaves differently in each. Assigning an array to a *scalar* gives its length; assigning it to a *list* (a parenthesised set of variables) copies elements.

```
use v5.42;
my @list = (10, 20, 30);

my $count = @list; # scalar context -> 3 (the length)
say $count;        # 3
```

(continues on next page)

(continued from previous page)

```

my ($first) = @list;      # list context -> first element copied
say $first;               # 10

my ($x, $y) = @list;      # 10 and 20
say "$x $y";              # 10 20

```

The parentheses on the left are the whole difference. `my $x = @list` asks “how many?” and gets 3 - this is where `count($arr)` lives, just spelled by context instead of a function. `my ($x) = @list` asks “give me the elements” and gets 10. A PHP developer reads both as “assign the array,” which is exactly the trap.

A subroutine can ask which context it was called in, with `wantarray`, and return different things accordingly:

```

use v5.42;
sub items {
    return wantarray ? (1, 2, 3) : "three items";
}
my @got = items();      # list context
my $msg = items();      # scalar context
say "@got";             # 1 2 3
say $msg;               # three items

```

Built-ins use this too. `scalar @array` forces scalar context to get a count - that is the explicit `count()` you were looking for. A regex match with `/g` returns the list of matches in list context and a true/false in scalar context.

The gotcha: there is no single answer to “what does this array do” - it depends on the surrounding expression. When something returns a number you expected to be a list (or vice versa), context is the first suspect. Reach for scalar to force a count and parentheses on the left to force element copying.

Truthiness

Perl’s false values are a short, specific list: the number 0, the empty string "", the string "0", and undef (Perl’s null). **Everything else is true.** This is close to PHP’s rules but not identical, and the differences are quiet.

```

use v5.42;
for my $v (0, 1, "", "0", "00", "0.0", "false") {
    say "'$v' is ", ($v ? "true" : "false");
}

```

This prints, exactly:

```

'0' is false
'1' is true
'' is false
'0' is false
'00' is true

```

(continues on next page)

(continued from previous page)

```
'0.0' is true
'false' is true
```

The gotcha: "00" and "0.0" are **true** in Perl, even though PHP treats "0" and only "0" as the falsy string too - Perl agrees on "0" but not on its look-alikes. The string "false" is true (it is a non-empty, non-"0" string), the same as in PHP. And unlike PHP's empty-array-is-falsy rule, an empty Perl array is not itself a false *value*: in boolean context an array yields its length, so `if (@list)` is true exactly when the list is non-empty, which is the test you want. For “use this if defined, else a default,” use `defined-or //`, the equivalent of PHP's `??`, which falls back only on `undef` (not on `0` or `""`):

```
use v5.42;
my %cfg = (host => "localhost");
my $port = %cfg{port} // 8080;    # key absent -> undef -> 8080
say $port;                      # 8080
```

Control flow

The keywords are familiar; the spelling differs in two small ways. `elseif` is `elsif` (one e), and conditions need parentheses while blocks need braces - there is no `:/endif` alternate syntax.

```
if ($n < 0) {
    $sign = "neg";
} elseif ($n == 0) {
    $sign = "zero";
} else {
    $sign = "pos";
}

foreach ($nums as $x) {
    echo $x;
}

while ($running) {
    step();
}
```

```
use v5.42;
my $sign;
if ($n < 0) { $sign = "neg" }
elsif ($n == 0) { $sign = "zero" }
else { $sign = "pos" }

for my $x (@nums) { say $x }

while ($running) { step() }
```

Perl adds two things PHP does not have. **Postfix modifiers** let a simple statement carry its own condition or loop, which reads naturally for one-liners:

```

use v5.42;
say $_ for @nums;           # loop, one statement
say "negative" if $n < 0;    # guarded statement
$sum += $_ for @nums;       # accumulate

```

And unless/until are the negated forms of if/while, for when the positive phrasing reads awkwardly: say "ok" unless \$error.

The gotcha: the ternary is `?/:`, the same as PHP's, and chains the same way: `my $sign = $n < 0 ? "neg" : $n == 0 ? "zero" : "pos"`. Perl has no switch/match statement in core; a chained ternary or a dispatch hash (`$handler{$key}->()`) is the idiom, and a for-on-a- single-value block is the common substitute for a switch.

Subroutines

PHP's function becomes Perl's sub. Modern Perl has signatures that look much like PHP's parameter lists, including default values.

```

function greet($name, $greeting = "Hello") {
    return "$greeting, $name!";
}

function total(...$nums) {
    return array_sum($nums);
}

```

```

use v5.42;
sub greet ($name, $greeting = "Hello") {
    return "$greeting, $name!";
}
say greet("Ada");           # Hello, Ada!
say greet("Rasmus", "Hi");  # Hi, Rasmus!

sub total (@nums) {
    my $sum = 0;
    $sum += $_ for @nums;
    return $sum;
}
say total(1, 2, 3, 4);      # 10

```

The variadic `...$nums` becomes a trailing array parameter `@nums`, which gathers all remaining arguments. PHP's named arguments (or the common `$opts` array convention) map to a hash parameter, since `key => value` pairs flatten into a list that a `%opts` signature gathers up:

```

use v5.42;
sub connect_to (%opts) {
    return "host=$opts{host} port=$opts{port}";
}
say connect_to(host => "localhost", port => 8080);
# host=localhost port=8080

```

The gotcha: a sub returns the value of its last expression even without `return`, and the return is **context-sensitive** (see [Context](#)). If you return `@list`, the caller gets the list in list context but the *count* in scalar context. Be deliberate about what you return when it might be used either way.

References

PHP arrays nest directly: an array can hold another array and you just keep subscripting. Perl arrays and hashes do not nest directly; you nest them through **references**, scalars that point at an aggregate. This is the same machinery as PHP's `&$x` references and its object handles, but in Perl you reach for it whenever you build nested data. Take a reference with `\`, follow it with `->`.

```
$data = ["name" => "Ada", "langs" => ["Perl", "PHP"]];
echo $data["langs"][1];          // PHP
$data["langs"][] = "Ruby";
```

```
use v5.42;
my $data = { name => "Ada", langs => ["Perl", "PHP"] };
say $data->{name};                # Ada
say $data->{langs}[1];            # PHP
push $data->{langs}->@*, "Ruby";  # postfix deref to push
say "@{$data->{langs}}";          # Perl PHP Ruby
```

`{ ... }` builds an anonymous hash reference and `[ ... ]` an anonymous array reference - the literal building blocks of nested data, matching the `[ ... ]` and `[k => v]` literals you already write in PHP. The `->` arrow follows a reference one step; between two subscripts it is optional, so `$data->{langs}[1]` and `$data->{langs}->[1]` are the same. The arrow is also exactly the method-call arrow you know from PHP OO - same symbol, both for “reach inside” and “call a method.”

Anonymous subs are closures, exactly like PHP's `fn` and `function()` use `(...)`, but with no `use` clause needed - they capture lexicals automatically:

```
use v5.42;
my $double = sub ($n) { $n * 2 };
say $double->(21);                # 42
```

The gotcha: the postfix deref forms (`->@*`, `->%*`, `->$*`) are the modern, readable way to dereference a whole aggregate, and `"@{ ... }"` is how you interpolate a dereferenced array into a string. The older `@$ref` and `${$ref}{key}` forms still work and appear in existing code, but reach for the arrow and postfix forms in new code.

Regular expressions

In PHP regex is the `preg_*` function family with pattern strings like `'/\d+/'`. In Perl it is **syntax**: `=~` binds a pattern to a string, `m//` matches, `s///` substitutes - no delimiters-inside-a-string, no function call. Capture groups land in `$1`, `$2`, and named captures in `%+`.


```
if (preg_match('/(\d{4})-(\d{2})-(\d{2})/', $s, $m)) {
    [$_, $year, $month, $day] = $m;
}
$t = preg_replace('/', '/', ';', "a,b,c");
$parts = explode(",", "one,two,three");
```

```
use v5.42;
my $s = "2026-06-20";
if ($s =~ /(\d{4})-(\d{2})-(\d{2})/) {
    say "$1 $2 $3";           # 2026 06 20
}
if ($s =~ /(?!<year>\d{4})/) {
    say $+{year};            # 2026
}
(my $t = "a,b,c") =~ s/,/,/g;    # a;b;c
my @parts = split /,/, "one,two,three"; # one two three
```

A /g match in list context returns *all* captures, the direct analogue of `preg_match_all`:

```
use v5.42;
my @nums = "x1y2z3" =~ /(\d)/g;    # 1 2 3
say "@nums";
```

The gotcha: `s///` modifies its target *in place* and returns the number of substitutions, not a new string - unlike `preg_replace`, which returns a fresh string. That is why the example copies first with `(my $t = "a,b,c")` and then substitutes on the copy - otherwise you would be editing a string literal. To get a new string without touching the original (the `preg_replace` reflex), use the /r flag: `my $t = $s =~ s/,/,/gr`.

File and stream I/O

The idiom is the three-argument `open` with a lexical filehandle, the counterpart of PHP's `fopen` into a `$handle`. The angle-bracket operator `<$fh>` reads a line, the same role as `fgets`.

```
$wh = fopen("/tmp/sample.txt", "w");
for ($i = 1; $i <= 3; $i++) {
    fwrite($wh, "line $i\n");
}
fclose($wh);

$rh = fopen("/tmp/sample.txt", "r");
while (($line = fgets($rh)) != false) {
    echo "got: " . rtrim($line) . "\n";
}
fclose($rh);
```

```
use v5.42;
my $path = "/tmp/sample.txt";
```

(continues on next page)

(continued from previous page)

```
open my $wh, '>', $path or die "cannot write: $!";
print {$wh} "line $_\n" for 1 .. 3;
close $wh;

open my $rh, '<', $path or die "cannot read: $!";
while (my $line = <$rh>) {
    chomp $line;
    say "got: $line";          # got: line 1 ... got: line 3
}
close $rh;
```

To slurp every line at once, read the handle in list context - the moral equivalent of `file()`:

```
use v5.42;
open my $fh, '<', "/tmp/sample.txt" or die $!;
my @lines = <$fh>;          # all lines, including newlines
say scalar @lines;          # 3
```

The gotcha: a line read with `<$fh>` keeps its trailing newline; call `chomp` to strip it (the `rtrim` reflex, but `chomp` removes only the record separator). And `open` returns false on failure rather than raising or warning, so the `or die "...: $!"` idiom is mandatory - `$!` is the system error message, Perl's `errno` (PHP's `error_get_last()`).

Modules and packages

PHP's `require/include` and Composer's autoloader become Perl's `use`. The namespace separator is `::`, not `\`, and a module `Foo::Bar` lives at `Foo/Bar.pm` on the include path `@INC` (Perl's `include_path`). A module's import list is a selection of names to bring into your namespace.

```
require 'vendor/autoload.php';
use App\Util>ListHelper;
$sum = ListHelper::sum(range(1, 10));
```

```
use v5.42;
use List::Util qw(sum0 max first);
say sum0(1 .. 10);          # 55
say max(3, 7, 2);           # 7
say first { $_ % 2 == 0 } (1, 3, 4, 5); # 4
```

PHP's huge flat standard-function namespace (`count`, `in_array`, `array_map`, `implode`, `sprintf`, ...) maps in Perl partly to built-in operators and partly to a few everyday modules. The common translations:

PHP	Perl
<code>count(\$a)</code>	<code>scalar @a</code>
<code>in_array(\$x, \$a)</code>	<code>grep { \$_ eq \$x } @a</code>
<code>array_map(\$f, \$a)</code>	<code>map { ... } @a</code>
<code>array_filter(\$a, \$f)</code>	<code>grep { ... } @a</code>
<code>implode(",", \$a)</code>	<code>join ",", @a</code>
<code>explode(",", \$s)</code>	<code>split /,/, \$s</code>
<code>sprintf(...)</code>	<code>sprintf(...)</code> (same)
<code>isset(\$h[\$k])</code>	<code>exists \$h{\$k}/defined \$h{\$k}</code>
<code>array_sum(\$a)</code>	<code>sum0 @a (List::Util)</code>

List::Util, Scalar::Util, POSIX, File::Spec, and Data::Dumper are the everyday modules; most of what you would pull from Packagist ships in core or on CPAN, the Perl package archive.

The gotcha: use runs at *compile time*, not at the point in the file where it appears, so the order of use lines relative to your code does not matter the way require placement can in PHP. Importing is also opt-in per name: use List::Util qw(sum0 max) brings exactly sum0 and max into your namespace, nothing else - there is no global function table the way PHP exposes every built-in everywhere.

(Web request data - PHP's \$_GET, \$_POST, \$_SERVER superglobals - has no language-level analogue in Perl; that is the job of a CGI or PSGI layer, not the core language, and is out of scope for this guide.)

Objects

For new code, Perl 5.42 has a first-class class syntax that will look familiar coming from PHP. Fields are declared with `field`, methods with `method`, and `$self` is available inside every method just as `$this` is in PHP (you do not list it as a parameter).

```
class Point {
    public function __construct(
        public int $x = 0,
        public int $y = 0,
    ) {}
    public function to_string(): string {
        return "({$this->x}, {$this->y})";
    }
    public function move(int $dx, int $dy): static {
        $this->x += $dx;
        $this->y += $dy;
        return $this;
    }
}

$p = new Point(x: 3, y: 4);
echo $p->to_string();           // (3, 4)
$p->move(1, 1);
echo $p->to_string();           // (4, 5)
```

```

use v5.42;
use feature 'class';

class Point {
    field $x :param = 0;
    field $y :param = 0;
    method to_string { return "($x, $y)" }
    method move ($dx, $dy) {
        $x += $dx;
        $y += $dy;
        return $self;
    }
}

my $p = Point->new(x => 3, y => 4);
say $p->to_string;           # (3, 4)
$p->move(1, 1);
say $p->to_string;           # (4, 5)

```

The constructor is generated for you, like PHP's promoted constructor properties: `:param` marks a field that the auto-built `new` accepts as a named argument, and `= 0` is its default. The method-call arrow `->` is the same one you use in PHP. Inheritance is `:isa`, the counterpart of `extends`:

```

use v5.42;
use feature 'class';

class Point3D :isa(Point) {
    field $z :param = 0;
    method z { return $z }
}

my $q = Point3D->new(x => 1, y => 2, z => 3);
say $q->to_string, " z=", $q->z;  # (1, 2) z=3

```

The gotcha: the class feature is still marked experimental in Perl 5.42, and on `perl` the “class is experimental” notices print to **stderr** even with no warnings `'experimental::class'` in scope. They do not touch your program's output, so a script that prints to `stdout` behaves exactly as shown above; just expect the notices on the error stream. The older object model - a package, a hash blessed into it with `bless`, and `@ISA` for inheritance - is what you will meet in existing code and most of CPAN; for it, for roles, and for migration advice, see the *Object-oriented Programming guide*.

Error handling

PHP's `try/catch` has a direct counterpart in modern Perl, and it reads almost identically. The raising side is `die` (PHP's `throw`):

```

function risky($n) {
    if ($n < 0) {

```

(continues on next page)

(continued from previous page)

```

        throw new Exception("negative!");
    }
    return sqrt($n);
}

try {
    risky(-4);
} catch (Exception $e) {
    echo "caught: " . $e->getMessage() . "\n";
}

```

```

use v5.42;
use feature 'try';

sub risky ($n) {
    die "negative!\n" if $n < 0;
    return sqrt $n;
}

try {
    risky(-4);
}
catch ($e) {
    print "caught: $e";      # caught: negative!
}

```

The older, still-common form wraps code in `eval { }` and inspects `$@` (the caught error) afterward - the same pattern as PHP code written before exceptions were idiomatic:

```

use v5.42;
my $result = eval { risky(-1) };
if ($@) {
    print "caught: $@";      # caught: negative!
}

```

The gotcha: `die` with a string that ends in `\n` produces exactly that message; `die` with a string that does *not* end in a newline gets " at SCRIPT line N.\n" appended automatically. End your error strings with `\n` when you want a clean message and omit it when you want the file-and-line trace. Note also the two error variables: `$@` is the last caught exception (PHP's `$e`), while `$!` is the last *system* error (a failed open, the `errno` string).

Idioms

A few patterns you will see daily and want in your fingers.

Sort a list by a computed key (the Schwartzian transform - PHP's `usort` with a precomputed key, done in one expression: `decorate, sort, undecorate`):

```

use v5.42;
my @words = qw(apple fig pear);
my @by_length =
    map { $_->[1] }
    sort { $a->[0] <=> $b->[0] }
    map { [ length $_, $_ ] } @words;
say "@by_length";           # fig pear apple

```

Build a hash from a list, the `array_combine/array_column` analogue (here, word to its length):

```

use v5.42;
my @words = qw(apple pear fig);
my %len = map { $_ => length $_ } @words;
say "$_ => $len{$_}" for sort keys %len;
# apple => 5
# fig => 3
# pear => 4

```

And `pperl` is a command-line tool in the `awk/sed` niche - the same job you might reach for a quick PHP CLI script to do. The implicit-loop one-liner is the Perl developer's everyday reflex:

```
pperl -lane '$s += $F[-1]; END { print $s }' data.txt
```

The gotcha: read the *One-Liners guide* for the switches (`-n`, `-p`, `-a`, `-l`, `-e`) and the dozens of recipes built on them; it is one of the highest-leverage things to learn coming from a PHP-and-shell background.

Gotchas coming from PHP

The traps, collected for a final scan before you start writing:

- **Sigils change with usage.** One element of `@a` is `$a[0]`; one value of `%h` is `$h{k}`. PHP keeps `$` everywhere; Perl varies the sigil by what shape you are getting.
- **One PHP array is two Perl types.** Decide up front: integer keys and order means `@array`; string keys means `%hash`. They are not interchangeable.
- **Two comparison families.** `==/</>=` for numbers, `eq/lt/cmp` for strings. Pick by operand type, not by variable. `==` on strings or `eq` on numbers is the classic bug.
- **`.` joins strings, `+` only adds numbers.** Perl's `+` never does array union or concatenation; mixing `.` and `+` corrupts your data silently or warns.
- **`sort` is string-sorted by default.** Pass `{ $a <=> $b }` for numeric order.
- **Hashes have no order.** Unlike a PHP array, iteration order is not insertion order; `sort keys %h` when you need stability.
- **Context.** `my $x = @a` is the *count*; `my ($x) = @a` is the *first element*. The parentheses are the whole difference. Re-read *Context* when a value comes back the wrong shape.

- **Truthiness surprises.** `"0"` is false, but `"00"` and `"0.0"` are **true**. Only `0`, `" "`, `"0"`, and `undef` are false. An empty array is not a false value - `if (@a)` tests non-empty.
- **`s///` edits in place** and returns a count, unlike `preg_replace`; use the `/r` flag to get a new string instead.
- **`open` does not raise.** Use `open ... or die "...: $!"`.
- **`chomp` your line reads.** `<$fh>` keeps the trailing newline.
- **Web superglobals are not core.** `$_GET/$_POST` belong to a CGI or PSGI layer, not the language.
- **Non-ASCII source needs `use utf8`;** Under `use v5.42`; a literal non-ASCII byte in your source is a compile error without it. Keep string data ASCII unless you specifically add `use utf8`;
- **The `class` feature prints experimental notices to `stderr`** on `ppperl`. Correct output, noisy error stream.

Ruby to Perl

You know Ruby. You pass blocks to each and map without thinking, you reach for a Hash by reflex, and `arr.select { |x| ... }` is muscle memory. This guide maps each of those habits onto Perl, so you can be productive in `ppperl` in an afternoon instead of relearning programming from scratch.

Two shifts land in the first five minutes, so meet them now:

- **Sigils mark the type, not the scope.** Ruby uses `@var` for instance state, `$var` for globals, and a bare `var` for locals. Perl uses `$x` for a single value, `@x` for an ordered list, `%x` for a key/value map - the sigil tells you the *shape* of the data, and it is the same whether the variable is local or global. Section one is about nothing else.
- **Not everything is an object.** In Ruby `"foo".upcase` and `[1, 2].sum` are method calls because numbers, strings, and arrays are all objects. Perl is pragmatic: a scalar is not an object, there is no universal base class, and those operations are plain functions - `uc "foo"`, `sum @a`. You call methods only on things you blessed into a class.

Every example here runs on `ppperl`, a Rust reimplementation of Perl 5.42. Each Perl snippet was executed on `ppperl`; the `# ...` comments show its real output. Start your scripts with:

```
use v5.42;
```

That one line turns on `strict` (no accidental globals), warnings, the `say` built-in (a `print` with a trailing newline, like Ruby's `puts`), subroutine signatures, and the modern feature set used throughout this guide.

How to read this

- *Variables and sigils* - the `$/@/%` shift.
- *Strings* - interpolation and concatenation.
- *Numbers* - one surface type, float division.

- *Arrays and lists* - your Array, plus block-taking map/grep.
- *Hashes* - your Hash, and where symbols go.
- *Context* - the concept Ruby has no word for. Read this one.
- *Truthiness* - what counts as false, and it differs sharply.
- *Control flow* - if, loops, the postfix forms.
- *Subroutines* - def becomes sub.
- *References* - nested data, and blocks as values.
- *Regular expressions* - regex as syntax, like Ruby's.
- *File and stream I/O* - open, line reads, slurp.
- *Modules and packages* - require becomes use.
- *Objects* - the modern class syntax.
- *Error handling* - begin/rescue becomes eval or try/catch.
- *Idioms* - the daily one-liners.
- *Gotchas coming from Ruby* - scan before you write.

Variables and sigils

In Ruby a sigil tells you *scope*: @x is an instance variable, \$x a global, x a local. In Perl the **sigil tells you shape**: \$ for a single value, @ for an ordered list, % for a key/value map. my declares a lexically scoped variable, the Perl equivalent of an ordinary local binding - it is how you get a local in Perl, where Ruby just uses a bare name.

```
name = "Ada"
langs = ["Ruby", "Perl"]
ages = {"Ada" => 36, "Yukihiro" => 60}
```

```
use v5.42;
my $name = "Ada";
my @langs = ("Ruby", "Perl");
my %ages = (Ada => 36, Yukihiro => 60);
say $name;           # Ada
say $langs[0];       # Ruby
say $ages{Ada};      # 36
```

The gotcha: when you reach *into* an aggregate for one element, the sigil follows the value you get back, not the container. A single element of @langs is a scalar, so it is \$langs[0], not @langs[0]; one value from %ages is \$ages{Ada}. The @/% form is the whole collection; the \$ form is one piece of it. (And forget the Ruby mapping: a Perl \$x is never “a global” the way Ruby’s \$x is - the \$ here just means “one value.”)

Strings

Double quotes interpolate variables, single quotes do not, exactly like Ruby's " . . ." versus ' . . . '. The Ruby `{expr}` form has a direct counterpart for plain variables and a slightly longer one for arbitrary expressions. Concatenation is `.`, not `+`.

```
name      = "Ada"
greeting  = "Hello, #{name}"
total     = "Total: #{6 * 7}"
shout     = "AB" * 3
```

```
use v5.42;
my $name = "Ada";
my $greeting = "Hello, $name";      # Hello, Ada
my $literal = 'Hello, $name';      # Hello, $name (no
  ↳ interpolation)
my $total    = "Total: @ {[ 6 * 7 ]}"; # Total: 42
my $joined   = "AB" . "CD";          # ABCD
my $shout    = "AB" x 3;             # ABABAB
```

The gotcha: a bare `"$name"` interpolates a variable just like Ruby's `{name}`, but to interpolate an *expression* you wrap it in the `@{ [ . . . ] }` form (it builds a throwaway array and inlines it). String repetition is `x`, not `*`, and the join operator is `.`, not `+`; `+` on strings coerces them to numbers and warns. Array elements and hash values interpolate inside double quotes too: `"First: $langs[0]"` works as written.

Numbers

Perl has one numeric type at the surface; you do not distinguish Integer from Float. Division always produces a float, like Ruby's Float division when either operand is a float.

```
puts 7.0 / 2      # 3.5
puts 7 / 2        # 3    (integer division in Ruby)
puts 2 ** 10      # 1024
puts 10 % 3       # 1
```

```
use v5.42;
say 7 / 2;        # 3.5    (always float)
say int(7 / 2);   # 3
say 2 ** 10;      # 1024
say 10 % 3;       # 1
```

The gotcha: comparison and equality operators come in two families, and you pick by the *operands' type*, not the variables'. Numbers use `==`, `!=`, `<`, `>`, `<=>`; strings use `eq`, `ne`, `lt`, `gt`, `cmp`. This has no Ruby analogue - Ruby's `==` dispatches on the object's class, while Perl's `==` always means *numeric* equality. `"10" == "10.0"` is true (numeric), but `"10" eq "10.0"` is false (string). Using `==` on strings, or `eq` on numbers, is a frequent and quiet bug.

Arrays and lists

A Perl array is your Ruby Array. The big shift is method calls becoming functions: Ruby's `arr.push(x)`, `arr.pop`, `arr.length` become `push @arr, x`, `pop @arr`, `scalar @arr` - the array is an argument, not a receiver.

```
nums = [1, 2, 3, 4, 5]
nums.push(6)
last = nums.pop          # 6
puts nums.length         # 5
puts nums[1, 2].join(" ") # 2 3
```

```
use v5.42;
my @nums = (1, 2, 3, 4, 5);
push @nums, 6;
my $last = pop @nums;      # 6
say scalar @nums;          # 5
say "@nums[1,2]";         # 2 3   (a slice)
```

Ruby's block-taking iterators are the richest part of this mapping. `map`, `select`, and `sort_by` become Perl `map`, `grep`, and `sort`, which take a `{ ... }` block exactly the way Ruby methods do. Inside the block, `$_` is the current element - Perl's implicit block variable, the role Ruby's `|x|` parameter plays.

```
squares = nums.map    { |x| x * x }
evens    = nums.select { |x| x.even? }
desc     = nums.sort   { |a, b| b <=> a }
```

```
use v5.42;
my @nums    = (1, 2, 3, 4, 5);
my @squares = map { $_ * $_ } @nums; # 1 4 9 16 25
my @evens   = grep { $_ % 2 == 0 } @nums; # 2 4
my @desc    = sort { $b <=> $a } @nums; # 5 4 3 2 1
```

The gotcha: `sort` compares as strings by default, so `sort (10, 9, 100)` gives `10 100 9`. For numbers you must pass a comparator: `sort { $a <=> $b } @nums`. `$a` and `$b` are the two elements being compared (the equivalent of Ruby's `|a, b|`), and `<=>` is the numeric three-way operator - the very same spaceship you know from Ruby (`cmp` is its string sibling).

Hashes

A Perl hash is your Ruby Hash. Declared with `%`, accessed one value at a time with `$h{key}`. The fat comma `=>` you already use in Ruby hash literals works here too, and it auto-quotes the bareword on its left.

```
h = {apple: 3, pear: 5, plum: 2}
puts h[:apple]           # 3
puts h.keys.sort.join(",") # apple,pear,plum
puts h.key?(:pear)       # true
```

(continues on next page)

(continued from previous page)

```
h.delete(:plum)
h.sort.each { |k, v| puts "#{k} => #{v}" }
```

```
use v5.42;
my %h = (apple => 3, pear => 5, plum => 2);
say $h{apple};           # 3
say join ", ", sort keys %h; # apple,pear,plum
say exists $h{pear} ? "yes" : "no"; # yes
delete $h{plum};
for my $k (sort keys %h) {
    say "$k => $h{$k}";
}
```

The gotcha: Perl has **no symbol type**. Where Ruby reaches for `:apple` as a lightweight, interned key, Perl just uses the string `"apple"` - and the `=>` fat comma auto-quotes it, so `apple => 3` and `$h{apple}` need no quotes. A Ruby symbol and a Ruby string are different objects; in Perl there is only the string, so the `:sym` versus `"sym"` distinction simply vanishes. Membership is `exists $h{key}`, not Ruby's `key?`, and a hash has no order - iterate `sort keys %h` when you want a stable one.

Context

This is the chapter to read twice. Ruby has no equivalent, and almost every “Perl is weird” story traces back to it.

Every expression in Perl runs in either **list context** or **scalar context**, decided by where it appears, not by what it is. The same array behaves differently in each. Assigning an array to a *scalar* gives its length; assigning it to a *list* (a parenthesised set of variables) copies elements. Ruby never overloads assignment this way - `a = arr` always just aliases the array - so this is genuinely new.

```
use v5.42;
my @list = (10, 20, 30);

my $count = @list;      # scalar context -> 3 (the length)
say $count;             # 3

my ($first) = @list;    # list context -> first element copied
say $first;            # 10

my ($a, $b) = @list;    # 10 and 20
say "$a $b";           # 10 20
```

The parentheses on the left are the whole difference. `my $x = @list` asks “how many?” and gets 3. `my ($x) = @list` asks “give me the elements” and gets 10. The Ruby reflex reads both as “assign the array,” which is exactly the trap.

A subroutine can ask which context it was called in, with `wantarray`, and return different things accordingly:

```

use v5.42;
sub items {
    return wantarray ? (1, 2, 3) : "three items";
}
my @got = items();           # list context
my $msg = items();           # scalar context
say "@got";                  # 1 2 3
say $msg;                    # three items

```

Built-ins use this too. `scalar @array` forces scalar context to get a count; a regex match with `/g` returns the list of matches in list context and a true/false in scalar context. The classic idiom for “count the matches” forces a list assignment in scalar context with the so-called *countof* form:

```

use v5.42;
my $n = () = "a1b2c3" =~ /\d/g;
say $n;           # 3

```

The gotcha: there is no single answer to “what does this array do” - it depends on the surrounding expression. When something returns a number you expected to be a list (or vice versa), context is the first suspect. Reach for scalar to force a count and parentheses on the left to force element copying.

Truthiness

This is the sharpest break from Ruby in the whole guide. In Ruby only `nil` and `false` are falsy - `0` and `""` are both **true**. In Perl the false values are a short, specific list: the number `0`, the empty string `""`, the string `"0"`, and `undef` (Perl’s `nil`). Everything else is true.

```

use v5.42;
for my $v (0, 1, "", "0", "00", "0.0", "false") {
    say "'$v' is ", ($v ? "true" : "false");
}

```

This prints, exactly:

```

'0' is false
'1' is true
'' is false
'0' is false
'00' is true
'0.0' is true
'false' is true

```

The gotcha: `0` and `""` are **false** in Perl but **true** in Ruby - the single most disruptive habit to unlearn. A Ruby guard like `if count` fires for `count == 0`; the Perl `if ($count)` does not. On the other side, `"00"` and `"0.0"` are **true** in Perl even though they look numerically zero, because only the exact string `"0"` is false. And unlike Ruby, where you might lean on `nil` checks, Perl distinguishes “false” from “undefined” with `defined-or //`, which falls back only on `undef` (not on `0` or `""`):

```

use v5.42;
my %cfg = (host => "localhost");
my $port = $cfg{port} // 8080;    # key absent -> undef -> 8080
say $port;                       # 8080

```

Control flow

The keywords are familiar; the spelling differs. `elsif` is Perl's `elsif` (Ruby's `elsif` too - one of the few exact matches), and blocks are braced, with no `then` and no terminating `end`.

```

if      n < 0 then sign = "neg"
elsif  n == 0 then sign = "zero"
else           sign = "pos"
end

```

```

nums.each { |x| puts x }

```

```

while running
  step
end

```

```

use v5.42;
my $sign;
if      ($n < 0) { $sign = "neg" }
elsif  ($n == 0) { $sign = "zero" }
else           { $sign = "pos" }

```

```

for my $x (@nums) { say $x }

```

```

while ($running) { step() }

```

Perl's **postfix modifiers** are the direct counterpart of Ruby's trailing `puts x if cond` and `do_it while cond` - a simple statement carrying its own condition or loop:

```

use v5.42;
say $_ for @nums;           # loop, one statement
say "negative" if $n < 0;    # guarded statement
$sum += $_ for @nums;       # accumulate

```

And `unless/until` are the negated forms of `if/while`, just like Ruby's `unless/until`: `say "ok" unless $error`.

The gotcha: the ternary uses `?/:`, not Ruby's `cond ? a : b` - which is actually identical, so this one is free. There is no `x if c else y` expression form; chain ternaries instead: `my $sign = $n < 0 ? "neg" : $n == 0 ? "zero" : "pos"`.

Subroutines

`def` becomes `sub`. Modern Perl has signatures that look much like Ruby's parameter lists, including defaults. And the good news a Ruby developer wants to hear: **the last evaluated expression is the return value**, exactly as in Ruby - `return` is optional.

```
def greet(name, greeting = "Hello")
    "#{greeting}, #{name}!"
end

def total(*nums)
    nums.sum
end
```

```
use v5.42;
sub greet ($name, $greeting = "Hello") {
    return "$greeting, $name!";
}
say greet("Ada");           # Hello, Ada!
say greet("Yukihiro", "Hi"); # Hi, Yukihiro!

sub area ($w, $h) { $w * $h } # no return needed; last expr wins
say area(3, 4);             # 12

sub total (@nums) {
    my $sum = 0;
    $sum += $_ for @nums;
    return $sum;
}
say total(1, 2, 3, 4);      # 10
```

Ruby keyword arguments map to a hash parameter, since `key => value` pairs flatten into a list that a `%opts` signature gathers up:

```
use v5.42;
sub connect_to (%opts) {
    return "host=$opts{host} port=$opts{port}";
}
say connect_to(host => "localhost", port => 8080);
# host=localhost port=8080
```

The gotcha: a sub returns the value of its last expression even without `return` (the Ruby reflex carries over), but in Perl that `return` is also **context-sensitive** (see [Context](#)). If you `return @list`, the caller gets the list in list context but the *count* in scalar context. Be deliberate about what you return when it might be used either way.

References

Perl's arrays and hashes do not nest directly; you nest them through **references**, a scalar that points at an aggregate. In Ruby every container is already a reference, so nesting is invisible to you there; in Perl you take a reference with `\` (or build one with `[ ]/{ }`) and follow it with `->`.

```
data = {name: "Ada", langs: ["Perl", "Ruby"]}
puts data[:langs][1]          # Ruby
data[:langs].push("Python")
```

```
use v5.42;
my $data = { name => "Ada", langs => ["Perl", "Ruby"] };
say $data->{name};              # Ada
say $data->{langs}[1];          # Ruby
push $data->{langs}->@*, "Python"; # postfix deref to push
say "@{$data->{langs}}";        # Perl Ruby Python
```

`{ ... }` builds an anonymous hash reference and `[ ... ]` an anonymous array reference - the literal building blocks of nested data, matching Ruby's `{}` and `[]` literals. The `->` arrow follows a reference one step; between two subscripts it is optional, so `$data->{langs}[1]` and `$data->{langs}->[1]` are the same.

The payoff for a Ruby developer is blocks-as-values. A Ruby block, `proc`, or `lambda` becomes a Perl anonymous sub: a closure stored in a scalar, called with `->`. It closes over its lexicals exactly like a Ruby block:

```
use v5.42;
my $doubler = sub { map { $_ * 2 } @_ };
say join ", ", $doubler->(1, 2, 3); # 2,4,6

my $factor = 10;
my $scaler = sub ($x) { $x * $factor }; # closes over $factor
say $scaler->(5);                       # 50
```

The gotcha: a Perl anonymous sub is a value you must *store and call explicitly* with `->( )` - there is no implicit `yield`, no block passed invisibly alongside the arguments. When a builtin wants a block (`map/grep/sort`, or `first/reduce` from `List::Util`), you write the `{ ... }` inline; when you want to pass a callable around as data, you make a sub `{ ... }` and call it through its scalar. The postfix deref forms (`->@*`, `->%*`, `->$*`) are the modern, readable way to dereference a whole aggregate.

Regular expressions

Good news: Perl regex will feel like home. Ruby borrowed its regex literal `/.../`, its `=~` operator, and `$1/$2` capture variables straight from Perl. The match operator `m//`, the substitution `s///`, and named captures in `%+` round it out.

```
s = "2026-06-20"
if s =~ /(\d{4})-(\d{2})-(\d{2})/
  puts "#{ $1 } #{ $2 } #{ $3 }"
end
t = "a,b,c".gsub(",", ";")
parts = "one,two,three".split(",")
```

```
use v5.42;
my $s = "2026-06-20";
```

(continues on next page)

(continued from previous page)

```

if ($s =~ /\d{4})-(\d{2})-(\d{2})/) {
    say "$1 $2 $3";          # 2026 06 20
}
if ($s =~ /(?!<year>\d{4})/) {
    say "${year}";          # 2026
}
(my $t = "a,b,c") =~ s/,/,/g;    # a;b;c
my @parts = split /,/, "one,two,three"; # one two three

```

A `/g` match in list context returns *all* captures, the direct analogue of Ruby's `String#scan`:

```

use v5.42;
my @nums = "x1y2z3" =~ /\d/g;    # 1 2 3
say "@nums";

```

The gotcha: where Ruby's `gsub` returns a new string and leaves the original alone, Perl's `s///` modifies its target *in place* and returns the number of substitutions. That is why the example copies first with `(my $t = "a,b,c")` and then substitutes on the copy. To get a new string without touching the original - the `gsub` behaviour you expect - add the `/r` flag: `my $t = $s =~ s/,/,/gr`.

File and stream I/O

The idiom is the three-argument `open` with a lexical filehandle, the counterpart of Ruby's `File.open(path, mode) do |f| ... end`. The angle-bracket operator `<$fh>` reads a line, the same role as Ruby's `f.each_line` or `f.gets`.

```

File.open("/tmp/sample.txt", "w") do |f|
    (1..3).each { |i| f.write("line #{i}\n") }
end

File.open("/tmp/sample.txt") do |f|
    f.each_line { |line| puts "got: #{line.chomp}" }
end

```

```

use v5.42;
my $path = "/tmp/sample.txt";

open my $wh, '>', $path or die "cannot write: $!";
print {$wh} "line $_\n" for 1..3;
close $wh;

open my $rh, '<', $path or die "cannot read: $!";
while (my $line = <$rh>) {
    chomp $line;
    say "got: $line";    # got: line 1 ... got: line 3
}
close $rh;

```


To slurp every line at once, read the handle in list context:

```
use v5.42;
open my $fh, '<', "/tmp/sample.txt" or die $!;
my @lines = <$fh>;           # all lines, including newlines
say scalar @lines;           # 3
```

The gotcha: a line read with <\$fh> keeps its trailing newline; call `chomp` to strip it - Ruby's `line.chomp` is the same verb, and Perl borrowed the name. And `open` returns false on failure rather than raising the way Ruby's `File.open` does, so the `or die "`. . . : \$!" idiom is mandatory; `$!` is the system error message, Perl's equivalent of the message in Ruby's `Errno` exception.

Modules and packages

`require/include` becomes `use`. The namespace separator is `::`, which you already know from Ruby constants like `Set::Foo`; Perl uses it everywhere, for both packages and the names inside them. A module `Foo::Bar` lives at `Foo/Bar.pm` on the include path `@INC` (Perl's `$LOAD_PATH`). A module's import list is a selection of names, much like pulling specific methods into scope.

```
require "set"
nums = [1, 2, 3, 4].sum
biggest = [3, 7, 2].max
```

```
use v5.42;
use List::Util qw(sum0 max first);
say sum0(1 .. 10);           # 55
say max(3, 7, 2);           # 7
say first { $_ % 2 == 0 } (1, 3, 4, 5); # 4
```

The standard library is large, and much of what you would reach to RubyGems for ships in core or on CPAN, the Perl package archive. `List::Util`, `Scalar::Util`, `POSIX`, `File::Spec`, and `Data::Dumper` are the everyday ones.

The gotcha: `use` runs at *compile time*, not at the point in the file where it appears, so the order of `use` lines relative to your code does not matter the way `require` placement can. Importing is opt-in per name: `use List::Util qw(sum0 max)` brings exactly `sum0` and `max` into your namespace, nothing else - there is no monkey-patching of built-in types the way Ruby lets you reopen `Array`.

Objects

Ruby is everything-is-an-object; Perl is pragmatic, and the difference showed up already in *Arrays* (functions, not methods). But when you *do* want objects, Perl 5.42 has a first-class class syntax that will look familiar. Fields are declared with `field`, methods with `method`, and `$self` is available inside every method just as `self` is in Ruby (you do not list it as a parameter).

```
class Point
  attr_accessor :x, :y
```

(continues on next page)

(continued from previous page)

```

def initialize(x: 0, y: 0)
  @x = x
  @y = y
end
def to_string
  "(#{@x}, #{@y})"
end
def move(dx, dy)
  @x += dx
  @y += dy
  self
end
end

p = Point.new(x: 3, y: 4)
puts p.to_string          # (3, 4)
p.move(1, 1)
puts p.to_string          # (4, 5)

```

```

use v5.42;
use feature 'class';

class Point {
  field $x :param = 0;
  field $y :param = 0;
  method to_string { return "($x, $y)" }
  method move ($dx, $dy) {
    $x += $dx;
    $y += $dy;
    return $self;
  }
}

my $p = Point->new(x => 3, y => 4);
say $p->to_string;          # (3, 4)
$p->move(1, 1);
say $p->to_string;          # (4, 5)

```

The constructor is generated for you, the equivalent of Ruby's `attr_accessor` plus `initialize` collapsed into the field declarations: `:param` marks a field that the auto-built `new` accepts as a named argument, and `= 0` is its default. Inheritance is `:isa`, the counterpart of Ruby's `class Point3D < Point`:

```

use v5.42;
use feature 'class';

class Point3D :isa(Point) {
  field $z :param = 0;
  method z { return $z }
}

```

(continues on next page)

(continued from previous page)

```

}

my $q = Point3D->new(x => 1, y => 2, z => 3);
say $q->to_string, " z=", $q->z;    # (1, 2) z=3

```

The gotcha: the class feature is still marked experimental in Perl 5.42, and on pperl the “class is experimental” notices print to **stderr** even with no warnings 'experimental::class' in scope. They do not touch your program’s output, so a script that prints to stdout behaves exactly as shown above; just expect the notices on the error stream. Note also that field gives you a :reader for read access but no Ruby-style attr_accessor setter by default - you write a method to mutate, as move does above. For the older bless-based object model you will meet in existing code, and for roles, inheritance details, and migration, see the *Object-oriented Programming guide*.

Error handling

Ruby’s begin/rescue/ensure has two counterparts. The classic one wraps code in eval { } and inspects \$@ (the caught error) afterward - note that eval here is exception *trapping*, not Ruby’s eval of a string:

```

begin
    risky(-1)
rescue => e
    puts "caught: #{e.message}"
end

```

```

use v5.42;
my $result = eval { risky(-1) };
if ($@) {
    print "caught: $@";    # caught: negative!
}

```

The modern form is try/catch/finally, which reads almost exactly like begin/rescue/ensure:

```

use v5.42;
use feature 'try';

try {
    risky(-4);
}
catch ($e) {
    print "caught: $e";    # caught: negative!
}
finally {
    say "done";            # done    (runs either way, like ensure)
}

```

where the raising side is die - Perl’s raise:

```
use v5.42;
sub risky ($n) {
    die "negative!\n" if $n < 0;
    return sqrt $n;
}
```

The gotcha: die with a string that ends in `\n` produces exactly that message; die with a string that does *not* end in a newline gets " at SCRIPT line N.\n" appended automatically. End your error strings with `\n` when you want a clean message and omit it when you want the file-and-line trace. The `finally` block, like Ruby's `ensure`, runs whether or not an exception was thrown. Note also the two error variables: `$@` is the last caught exception (Ruby's `rescue => e`), while `$!` is the last *system* error (a failed open). Under `pperl`, the `try/catch/finally` feature prints an "experimental" notice on `stderr`; `stdout` is unaffected.

Idioms

A few patterns you will see daily and want in your fingers.

Sort a list by a computed key. Ruby has `sort_by` built in; the Perl hand-rolled form is the Schwartzian transform - `decorate, sort, undecorate` - and is worth recognising on sight:

```
use v5.42;
my @words = qw(apple fig pear);
my @by_length =
    map { $_->[1] }
    sort { $a->[0] <=> $b->[0] }
    map { [ length $_, $_ ] } @words;
say "@by_length";           # fig pear apple
```

Build a hash from a list, the Ruby `words.to_h { |w| [w, w.length] }` analogue:

```
use v5.42;
my @words = qw(apple pear fig);
my %len = map { $_ => length $_ } @words;
say "$_ => $len{$_}" for sort keys %len;
# apple => 5
# fig => 3
# pear => 4
```

And `pperl` is a command-line tool in the `awk/sed` niche - the role `ruby -e` and `ruby -ne` fill for shell glue. The implicit-loop one-liner is the Perl developer's everyday reflex:

```
pperl -lane '$s += $F[-1]; END { print $s }' data.txt
```

The gotcha: read the *One-Liners guide* for the switches (`-n`, `-p`, `-a`, `-l`, `-e`) and the dozens of recipes built on them; it is one of the highest-leverage things to learn coming from a shell-and-Ruby background.

Gotchas coming from Ruby

The traps, collected for a final scan before you start writing:

- **Sigils mark shape, not scope.** `$x/@x/%x` say single/list/map; they are not Ruby's instance/global/local markers. One element of `@a` is `$a[0]`; one value of `%h` is `$h{k}`.
- **Not everything is an object.** `uc "foo"` and `sum @a` are functions, not methods on a string or array. You call methods only on blessed references.
- **Truthiness is the big one.** `0` and `"` are **false** in Perl but **true** in Ruby. `if ($count)` is false when `$count` is `0`. Only `0`, `"`, `"0"`, and `undef` are false; `"00"` and `"0.0"` are **true**.
- **No symbols.** `:name` has no Perl equivalent; hash keys are plain strings, and `=>` auto-quotes the bareword on its left.
- **. joins strings, + adds numbers.** Never `+` two strings to concatenate; that coerces them to numbers and warns. String repeat is `x`, not `*`.
- **Two comparison families.** `==/</>=` for numbers, `eq/lt/ cmp` for strings. Pick by operand type, not by variable - unlike Ruby's class-dispatched `==`.
- **sort is string-sorted by default.** Pass `{ $a <=> $b }` for numeric order.
- **Context.** `my $x = @a` is the *length*; `my ($x) = @a` is the *first element*. The parentheses are the whole difference. Re-read [Context](#) when a value comes back the wrong shape.
- **s/// edits in place** and returns a count, unlike Ruby's `gsub`; use the `/r` flag to get a new string instead.
- **open does not raise.** Use `open ... or die "...: $!"`.
- **Blocks are not implicit.** A stored sub `{ ... }` is called explicitly with `->( )`; there is no `yield` and no invisible block argument.
- **Non-ASCII source needs use utf8;.** Under `use v5.42`; a literal non-ASCII byte in your source is a compile error without it. Keep string data ASCII unless you specifically add `use utf8;`.
- **The class feature prints experimental notices to stderr** on `ppperl`. Correct output, noisy error stream.

Lua to Perl

You know Lua. You build a table without thinking, you reach for `setmetatable` to get objects and operators, and for `k, v in pairs(t)` is muscle memory. This guide maps each of those habits onto Perl, so you can be productive in `ppperl` in an afternoon instead of relearning programming from scratch.

Two shocks land in the first five minutes, so meet them now:

- **Lua's one table becomes two Perl types.** Lua has a single aggregate, the table, that you use as both an array (`{1, 2, 3}`) and a map (`{k = "v"}`). Perl splits that into two distinct things: an ordered `@array` keyed by integers and a `%hash` keyed by strings. Deciding “is this a list or a lookup?” up front is the central new habit, and the [Arrays](#) and [Hashes](#) sections are about nothing else.

- **Variables wear sigils.** Lua names are bare: `x`, `t`, `f`. In Perl a scalar is `$x`, an array is `@x`, a hash is `%x`. The sigil is part of how you read the variable, not just decoration. This is the single biggest surface difference from Lua, and section one is about nothing else.

Every example here runs on `ppperl`, a Rust reimplementation of Perl 5.42. Each Perl snippet was executed on `ppperl`; the `# ...` comments show its real output. Start your scripts with:

```
use v5.42;
```

That one line turns on `strict` (every variable must be declared with `my`, the cure for Lua's accident-prone implicit globals), warnings, the `say` built-in (a `print` with a trailing newline, like Lua's `print`), subroutine signatures, and the modern feature set used throughout this guide.

How to read this

- *Variables and sigils* - the `$/@/%` shift, and `local x` becoming `my $x`.
- *Strings* - interpolation, and `..` concatenation becoming `..`.
- *Numbers* - one surface type, float division.
- *Arrays and lists* - half of your table, 0-based not 1-based.
- *Hashes* - the other half of your table.
- *Context* - the concept Lua has no word for. Read this one.
- *Truthiness* - what counts as false, and it differs sharply from Lua.
- *Control flow* - `if`, loops, the postfix forms.
- *Subroutines* - `function` becomes `sub`; real closures.
- *References* - nested data and closures.
- *Regular expressions* - real regex, not Lua patterns.
- *File and stream I/O* - `open`, line reads, slurp.
- *Modules and packages* - `require` becomes `use`.
- *Objects* - metatables become the modern `class` syntax.
- *Error handling* - `pcall` becomes `eval` or `try/catch`.
- *Idioms* - the daily one-liners.
- *Gotchas coming from Lua* - scan before you write.

Variables and sigils

In Lua a name just names a value, whatever its shape. In Perl the **sigil tells you the shape**: `$` for a single value, `@` for an ordered list, `%` for a key/value map. `my` declares a lexically scoped variable, the exact role of Lua's `local`. Where Lua lets a bare assignment silently create a global, `use v5.42` turns that into a compile error: every variable is `my` unless you say otherwise.

```
local name = "Ada"
local langs = {"Lua", "Perl"}
local ages = {Ada = 36, Linus = 56}
```

```
use v5.42;
my $name = "Ada";
my @langs = ("Lua", "Perl");
my %ages = (Ada => 36, Linus => 56);
say $name;           # Ada
say $langs[0];       # Lua
say $ages{Ada};      # 36
```

The gotcha: when you reach *into* an aggregate for one element, the sigil follows the value you get back, not the container. A single element of `@langs` is a scalar, so it is `$langs[0]`, not `@langs[0]`; one value from `%ages` is `$ages{Ada}`. The `@/%` form is the whole collection; the `$` form is one piece of it. In Lua the table name `t` never changes whether you write `t`, `t[1]`, or `t.key`; in Perl the leading mark changes with what you are accessing.

Strings

Double quotes interpolate variables, single quotes do not. There is no `string.format` reflex needed for the common case, because the default string already interpolates. The big swap: concatenation is `.`, where Lua uses `..`.

```
local name = "Ada"
local greeting = "Hello, " .. name
local shout = string.rep("AB", 3)
```

```
use v5.42;
my $name = "Ada";
my $greeting = "Hello, $name";    # Hello, Ada
my $literal = 'Hello, $name';    # Hello, $name (no interpolation)
my $joined = "AB" . "CD";        # ABCD
my $shout = "AB" x 3;             # ABABAB
```

The gotcha: Lua's concatenation operator `..` does not exist in Perl; the single dot `.` joins strings and the `x` operator repeats them (Lua's `string.rep`). Array elements and hash values interpolate inside double quotes too, so `"First: $langs[0]"` works as written, which removes most of the `..` chaining a Lua developer reaches for.

Numbers

Perl has one numeric type at the surface; you do not pick between integer and float, much as Lua 5.3+ presents a single number to most code. Division always produces a float.

```
print(7 / 2)          -- 3.5
print(math.floor(7 / 2)) -- 3
```

(continues on next page)

(continued from previous page)

```
print(2 ^ 10)      -- 1024.0
print(10 % 3)      -- 1
```

```
use v5.42;
say 7 / 2;          # 3.5
say int(7 / 2);     # 3
say 2 ** 10;        # 1024
say 10 % 3;         # 1
```

The gotcha: comparison and equality operators come in two families, and you pick by the *operands' type*, not the variables'. Numbers use `==`, `!=`, `<`, `>`, `<=>`; strings use `eq`, `ne`, `lt`, `gt`, `cmp`. `"10" == "10.0"` is true (numeric), but `"10" eq "10.0"` is false (string). Lua has only `==` and coerces nothing across the string/number line; Perl instead asks *you* to choose the comparison, and using `==` on strings (or `eq` on numbers) is a frequent quiet bug.

Arrays and lists

Here is half of the great table split. A Lua sequence (`{10, 20, 30}`, the table used as an array) becomes a Perl `@array`. The first and loudest difference: **Perl arrays are 0-based**, where Lua arrays conventionally start at index 1.

```
local t = {"x", "y", "z"}
print(t[1])    -- x    (Lua: first element is index 1)
print(t[3])    -- z
print(#t)      -- 3    (length)
```

```
use v5.42;
my @t = ("x", "y", "z");
say $t[0];     # x    (Perl: first element is index 0)
say $t[2];     # z
say $t[-1];    # z    (negative indices count from the end)
say scalar @t; # 3    (length; @t in scalar context is its size)
```

The mutators are renamed but familiar. `table.insert/table.remove` become `push/pop/shift/unshift`:

```
use v5.42;
my @nums = (10, 20, 30);
push @nums, 40;          # add to the end
my $last = pop @nums;    # 40, removed from the end
say scalar @nums;        # 3
say "@nums[0,1]";        # 10 20    (a slice)
```

Lua reaches for explicit `for i = 1, #t do ... end` loops to transform or filter; Perl has `map` (transform) and `grep` (filter), with `$_` as the current element:

```
use v5.42;
my @nums = (1, 2, 3, 4, 5);
```

(continues on next page)

(continued from previous page)

```
my @squares = map { $_ * $_ } @nums; # 1 4 9 16 25
my @evens   = grep { $_ % 2 == 0 } @nums; # 2 4
my @desc    = sort { $b <=> $a } @nums; # 5 4 3 2 1
```

The gotcha: beyond 0-based indexing, `#t` has no single Perl spelling - you use scalar `@a` for an array's length and keys `%h` for a hash's. And `sort` compares as strings by default, so `sort (10, 9, 100)` gives `10 100 9`. For numbers you must pass a comparator: `sort { $a <=> $b } @nums`. `$a` and `$b` are the two elements being compared, and `<=>` is the numeric three-way operator (`cmp` is its string sibling).

Hashes

Here is the other half of the table split. A Lua table used as a map (`{name = "Ada", age = 36}`, or `t["key"]`) becomes a Perl `%hash`. Declared with `%`, accessed one value at a time with `$h{key}`.

```
local h = {apple = 3, pear = 5, plum = 2}
print(h.apple)           -- 3
print(h["pear"])         -- 5
h.plum = nil             -- delete a key
for k, v in pairs(h) do
    print(k, v)
end
```

```
use v5.42;
my %h = (apple => 3, pear => 5, plum => 2);
say $h{apple};           # 3
say join ", ", sort keys %h; # apple,pear,plum
say exists $h{pear} ? "yes" : "no"; # yes
delete $h{plum};
for my $k (sort keys %h) {
    say "$k => $h{$k}"; # apple => 3 ... pear => 5
}
```

The gotcha: Lua merges the array and the map into one table and keeps both an array part and a hash part inside it; Perl keeps them strictly apart, so you decide up front whether a thing is an `@array` (integer positions, ordered) or a `%hash` (string keys, unordered). Membership is `exists $h{key}`, the counterpart of Lua's `h[k] ~= nil` test, and the fat comma `=>` is the idiomatic key/value separator that auto-quotes the bareword on its left (so `apple => 3` needs no quotes). A hash has no order, so iterate `sort keys %h` when you want a stable one; Lua's `pairs` is unordered for the same reason.

Context

This is the chapter to read twice. Lua has no general equivalent, and almost every “Perl is weird” story traces back to it. But Lua *does* give you the one bridge that makes it click: **multiple return values**.

In Lua, `local a, b = f()` spreads a function's several return values across several variables, and `#t` or `select("#", ...)` collapses a list to a count. Perl generalises that

idea into a property of *every* expression. Every expression runs in either **list context** or **scalar context**, decided by where it appears. The same array behaves differently in each: assigned to a *scalar* it gives its length, assigned to a *list* (parenthesised variables) it copies elements.

```
use v5.42;
my @list = (10, 20, 30);

my $count = @list;      # scalar context -> 3 (the length)
say $count;             # 3

my ($first) = @list;    # list context -> first element copied
say $first;             # 10

my ($a, $b) = @list;    # 10 and 20
say "$a $b";           # 10 20
```

The parentheses on the left are the whole difference. `my $x = @list` asks “how many?” and gets 3. `my ($x) = @list` asks “give me the elements” and gets 10. The list form `my ($a, $b) = @list` is exactly Lua’s `local a, b = ...` - the bridge you already know.

A subroutine returning several values is the same as a Lua function with multiple return values; the caller’s context decides what comes back:

```
use v5.42;
sub minmax (@nums) {
    my @s = sort { $a <=> $b } @nums;
    return ($s[0], $s[-1]);
}
my ($lo, $hi) = minmax(4, 1, 9, 2);
say "$lo $hi";          # 1 9
```

A sub can even ask which context it was called in, with `wantarray`, and answer differently - something Lua functions cannot do:

```
use v5.42;
sub items {
    return wantarray ? (1, 2, 3) : "three items";
}
my @got = items();       # list context
my $msg = items();       # scalar context
say "@got";              # 1 2 3
say $msg;                # three items
```

The gotcha: there is no single answer to “what does this array do” - it depends on the surrounding expression, where Lua resolves multiple-value spreading at the specific call site only. When something returns a number you expected to be a list (or vice versa), context is the first suspect. Reach for scalar to force a count and parentheses on the left to force element copying.

Truthiness

This is a sharp break from Lua, so slow down. In Lua **only nil and false are falsy** - the number 0 and the empty string "" are both *true*. Perl draws the line in a completely different place. Its false values are: the number 0, the empty string "", the string "0", and undef (Perl's nil). **Everything else is true.**

```
use v5.42;
for my $v (0, 1, "", "0", "00", "0.0", "false") {
    say "'$v' is ", ($v ? "true" : "false");
}
```

This prints, exactly:

```
'0' is false
'1' is true
'' is false
'0' is false
'00' is true
'0.0' is true
'false' is true
```

The gotcha: a Lua developer's instinct that 0 is true will silently flip the meaning of every if (\$count). In Perl 0 and "" *are* false, so if (\$n) is false when \$n is zero - usually what you want, but the opposite of Lua. Going the other way, "00" and "0.0" are **true** even though they look numerically zero, because only the exact string "0" is false. For a “use this if defined, else a default” choice - Lua's `x = y or default` idiom, which trips on 0 and false - use Perl's defined-or `//`, which falls back only on undef (Perl's nil):

```
use v5.42;
my %cfg = (host => "localhost");
my $port = $cfg{port} // 8080;    # key absent -> undef -> 8080
say $port;                       # 8080
```

Control flow

The keywords are familiar; the spelling differs. Lua's `elseif` is Perl's `elsif`, there is no `then/do/end`, and blocks are braced rather than keyword-delimited.

```
if n < 0 then
    sign = "neg"
elseif n == 0 then
    sign = "zero"
else
    sign = "pos"
end

for _, x in ipairs(nums) do
    print(x)
end
```

(continues on next page)

(continued from previous page)

```
while running do
    step()
end
```

```
use v5.42;
my $sign;
if ($n < 0) { $sign = "neg" }
elsif ($n == 0) { $sign = "zero" }
else { $sign = "pos" }

for my $x (@nums) { say $x }

while ($running) { step() }
```

Perl adds two things Lua does not have. **Postfix modifiers** let a simple statement carry its own condition or loop, which reads naturally for one-liners:

```
use v5.42;
say $_ for @nums;           # loop, one statement
say "negative" if $n < 0;    # guarded statement
$sum += $_ for @nums;       # accumulate
```

And unless/until are the negated forms of if/while, for when the positive phrasing reads awkwardly: say "ok" unless \$error.

The gotcha: Perl has no `x if c else y` statement; the conditional expression is the C-style ternary with `?:`, which Lua usually fakes with `c and x or y`. The Perl form is direct and chains: `my $sign = $n < 0 ? "neg" : $n == 0 ? "zero" : "pos"`. Loop control is last/next where Lua has `break` and (in 5.4) `goto continue`.

Subroutines

function becomes sub. Modern Perl has signatures that name their parameters, including defaults, rather than Lua's positional ... and manual arg handling.

```
local function greet(name, greeting)
    greeting = greeting or "Hello"
    return greeting .. ", " .. name .. "!"
end

local function total(...)
    local sum = 0
    for _, n in ipairs({...}) do sum = sum + n end
    return sum
end
```

```
use v5.42;
sub greet ($name, $greeting = "Hello") {
    return "$greeting, $name!";
}
```

(continues on next page)

(continued from previous page)

```

}
say greet("Ada");           # Hello, Ada!
say greet("Linus", "Hi");   # Hi, Linus!

sub total (@nums) {
    my $sum = 0;
    $sum += $_ for @nums;
    return $sum;
}
say total(1, 2, 3, 4);      # 10

```

Lua's named-argument idiom (passing a single table, `f{host=..., port=...}`) maps to a hash parameter, since `key => value` pairs flatten into a list that a `%opts` signature gathers up:

```

use v5.42;
sub connect_to (%opts) {
    return "host=$opts{host} port=$opts{port}";
}
say connect_to(host => "localhost", port => 8080);
# host=localhost port=8080

```

The gotcha: a sub returns the value of its last expression even without `return` (like Lua's explicit-only `return`, except Perl also has the implicit last-value form), and the `return` is **context-sensitive** (see [Context](#)). If you `return @list`, the caller gets the list in list context but the *count* in scalar context - a degree of freedom Lua's multiple-return does not have.

References

Perl's arrays and hashes do not nest directly; you nest them through **references**, a scalar that points at an aggregate. In Lua every table is already a reference (assigning a table shares it, never copies), so this concept is invisible to you there; in Perl you take a reference explicitly with `\`, and follow it with `->`.

```

local data = {name = "Ada", langs = {"Perl", "Lua"}}
print(data.langs[2])           -- Lua
table.insert(data.langs, "Ruby")

```

```

use v5.42;
my $data = { name => "Ada", langs => ["Perl", "Lua"] };
say $data->{name};              # Ada
say $data->{langs}[1];          # Lua
push $data->{langs}->@*, "Ruby"; # postfix deref to push
say "@{$data->{langs}}";       # Perl Lua Ruby

```

`{ ... }` builds an anonymous hash reference and `[ ... ]` an anonymous array reference - the literal building blocks of nested data. The `->` arrow follows a reference one step; between two subscripts it is optional, so `$data->{langs}[1]` and `$data->{langs}->[1]` are the same.

Anonymous subs are real closures, exactly like Lua's closures over upvalues - one of the cleanest carry-overs in the whole guide:

```
use v5.42;
sub counter ($start) {
    my $n = $start;
    return sub { return $n++ };
}
my $c = counter(10);
say $c->();           # 10
say $c->();           # 11
say $c->();           # 12
```

The gotcha: the postfix deref forms (`->@*`, `->%*`, `->$*`) are the modern, readable way to dereference a whole aggregate, and `"@{ ... }"` is how you interpolate a dereferenced array into a string. The older `@$ref` and `${$ref}{key}` forms still work and appear in existing code, but reach for the arrow and postfix forms in new code.

Regular expressions

This is an upgrade, not a translation. Lua has **patterns** - `string.match`, `string.gsub`, character classes like `%d` - which are deliberately *not* regular expressions and lack alternation, quant, and backreferences. Perl has full regex as **syntax**: `=~` binds a pattern to a string, `m//` matches, `s///` substitutes. Capture groups land in `$1`, `$2`, and named captures in `%+`.

```
local s = "2026-06-20"
local y, m, d = string.match(s, "(%d+)-(%d+)-(%d+)")
local t = string.gsub("a,b,c", ",", ";")    -- a;b;c
```

```
use v5.42;
my $s = "2026-06-20";
if ($s =~ /(\d{4})-(\d{2})-(\d{2})/) {
    say "$1 $2 $3";           # 2026 06 20
}
if ($s =~ /(?<year>\d{4})/) {
    say $+{year};             # 2026
}
(my $t = "a,b,c") =~ s/,/,/g;    # a;b;c
my @parts = split /,/, "one,two,three"; # one two three
```

A `/g` match in list context returns *all* the matches, the direct analogue of looping `string.gmatch`:

```
use v5.42;
my @nums = "x1y2z3" =~ /(\d)/g;    # 1 2 3
say "@nums";
```

The gotcha: the metacharacters differ from Lua patterns - Perl uses `\d` where Lua uses `%d`, `*/+/?` quantifiers, `|` alternation, and `(?<name>...)` named groups, none of which Lua patterns support. And `s///` modifies its target *in place* and returns the number of substitutions, not a new string, where Lua's `gsub` returns a new string. That is why the

example copies first with `(my $t = "a,b,c")` and then substitutes on the copy. To get a new string without touching the original, use the `/r` flag: `my $t = $s =~ s/,/,;/gr`.

File and stream I/O

The idiom is the three-argument `open` with a lexical filehandle, the counterpart of Lua's `io.open`. The angle-bracket operator `<$fh>` reads a line, the role of Lua's `fh:read("l")` or the `fh:lines()` iterator.

```
local f = io.open("/tmp/sample.txt", "w")
for i = 1, 3 do
    f:write("line " .. i .. "\n")
end
f:close()

f = io.open("/tmp/sample.txt", "r")
for line in f:lines() do
    print("got:", line)
end
f:close()
```

```
use v5.42;
my $path = "/tmp/sample.txt";

open my $wh, '>', $path or die "cannot write: $!";
print {$wh} "line $_\n" for 1 .. 3;
close $wh;

open my $rh, '<', $path or die "cannot read: $!";
while (my $line = <$rh>) {
    chomp $line;
    say "got: $line";          # got: line 1 ... got: line 3
}
close $rh;
```

To slurp every line at once, read the handle in list context:

```
use v5.42;
open my $fh, '<', "/tmp/sample.txt" or die $!;
my @lines = <$fh>;          # all lines, including newlines
say scalar @lines;          # 3
```

The gotcha: a line read with `<$fh>` keeps its trailing newline (Lua's `fh:lines()` strips it, `fh:read("L")` keeps it); call `chomp` to strip it. And `open` returns false on failure rather than raising, much like Lua's `io.open` returning `nil`, `err`, so the `or die "...: $!"` idiom is mandatory - `$!` is the system error message, Perl's `errno`.

Modules and packages

Lua's `require` becomes Perl's `use`. The namespace separator is `::`, not `.`, and a module `Foo::Bar` lives at `Foo/Bar.pm` on the include path `@INC` (Lua's `package.path`). Unlike Lua's `require`, which returns a table you assign yourself, `use` imports selected names directly into your namespace.

```
local sqrt = math.sqrt
local sum = 0
for i = 1, 10 do sum = sum + i end
```

```
use v5.42;
use List::Util qw(sum0 max first);
say sum0(1 .. 10);           # 55
say max(3, 7, 2);           # 7
say first { $_ % 2 == 0 } (1, 3, 4, 5); # 4
```

Here is a pleasant surprise coming from Lua's deliberately tiny standard library: Perl ships a large built-in set, and much of what you would hand-roll or pull from LuaRocks is in core or one use away on CPAN, the Perl package archive. `List::Util`, `Scalar::Util`, `POSIX`, `File::Spec`, and `Data::Dumper` are the everyday ones.

The gotcha: `use` runs at *compile time*, not at the line where it appears, so its placement relative to your code does not matter the way a `require` call's does. Importing is opt-in per name: `use List::Util qw(sum0 max)` brings exactly `sum0` and `max` into your namespace, nothing else.

Objects

Lua has no built-in object system; you build one from tables and metatables - `setmetatable(t, {__index = Class})` for method lookup and inheritance, `__add`/`__eq` for operators. Perl 5.42 gives you a first-class class syntax instead, so the common case needs no metatable wiring. Fields are declared with `field`, methods with `method`, and `$self` is available inside every method (you do not pass it as the first parameter the way Lua's `self` / colon-call does).

```
local Point = {}
Point.__index = Point

function Point.new(x, y)
    return setmetatable({x = x or 0, y = y or 0}, Point)
end
function Point:to_string()
    return "(" .. self.x .. ", " .. self.y .. ")"
end
function Point:move(dx, dy)
    self.x = self.x + dx
    self.y = self.y + dy
    return self
end
```

(continues on next page)

(continued from previous page)

```

local p = Point.new(3, 4)
print(p:to_string())      -- (3, 4)
p:move(1, 1)
print(p:to_string())      -- (4, 5)

```

```

use v5.42;
use feature 'class';

class Point {
    field $x :param = 0;
    field $y :param = 0;
    method to_string { return "($x, $y)" }
    method move ($dx, $dy) {
        $x += $dx;
        $y += $dy;
        return $self;
    }
}

my $p = Point->new(x => 3, y => 4);
say $p->to_string;          # (3, 4)
$p->move(1, 1);
say $p->to_string;          # (4, 5)

```

The constructor is generated for you, replacing Lua’s hand-written `Point.new: :param` marks a field that the auto-built `new` accepts as a named argument, and `= 0` is its default. Inheritance, which Lua builds from a chain of `__index` metatables, is the `:isa` attribute:

```

use v5.42;
use feature 'class';

class Point3D :isa(Point) {
    field $z :param = 0;
    method z { return $z }
}

my $q = Point3D->new(x => 1, y => 2, z => 3);
say $q->to_string, " z=", $q->z;    # (1, 2) z=3

```

The gotcha: the `class` feature is still marked experimental in Perl 5.42, and on `ppperl` the “class is experimental” notices print to **stderr** even when you try to silence them. They do not touch your program’s output, so a script that prints to `stdout` behaves exactly as shown above; just expect the notices on the error stream. Lua’s operator metamethods (`__add`, `__eq`, and `friends`) map to Perl’s overload pragma rather than to `class`; for that, the older `bless`-based object model, roles, and inheritance details, see the *Object-oriented Programming guide*.

Error handling

Lua's protected call `pcall(f)` - which returns `false`, `err` instead of unwinding - has two Perl counterparts. The classic one wraps code in `eval { }` and inspects `$@` (the caught error) afterward:

```
local ok, err = pcall(function() risky(-1) end)
if not ok then
    print("caught: " .. err)
end
```

```
use v5.42;
my $result = eval { risky(-1) };
if ($@) {
    print "caught: $@";      # caught: negative!
}
```

The modern form is `try/catch`, which reads more like a conventional exception block than `pcall`'s boolean-return dance:

```
use v5.42;
use feature 'try';

try {
    risky(-4);
}
catch ($e) {
    print "caught: $e";      # caught: negative!
}
```

where the raising side is `die`, Perl's error:

```
use v5.42;
sub risky ($n) {
    die "negative!\n" if $n < 0;
    return sqrt $n;
}
```

The gotcha: `die` with a string that ends in `\n` produces exactly that message; `die` with a string that does *not* end in a newline gets " at SCRIPT line N.\n" appended automatically (Lua's error does a similar position-prefixing unless you pass level 0). End your error strings with `\n` when you want a clean message and omit it when you want the file-and-line trace. Note also the two error variables: `$@` is the last caught exception, while `!` is the last *system* error (a failed open, the `errno` string).

Idioms

A few patterns you will see daily and want in your fingers.

Sort a list by a computed key (the Schwartzian transform - `decorate`, `sort`, `undecorate` in one pipeline, the Perl reflex where Lua passes a comparator function to `table.sort`):

```
use v5.42;
my @words = qw(apple fig pear);
my @by_length =
    map { $_->[1] }
    sort { $a->[0] <=> $b->[0] }
    map { [ length $_, $_ ] } @words;
say "@by_length";           # fig pear apple
```

Build a hash from a list, turning a sequence into a lookup table in one step:

```
use v5.42;
my @words = qw(apple pear fig);
my %len = map { $_ => length $_ } @words;
say "$_ => $len{$_}" for sort keys %len;
# apple => 5
# fig => 3
# pear => 4
```

And pperl is a command-line tool in the awk/sed niche, a role Lua does not target. The implicit-loop one-liner is the Perl developer's everyday reflex:

```
ppperl -lane '$s += $F[-1]; END { print $s }' data.txt
```

The gotcha: read the *One-Liners guide* for the switches (-n, -p, -a, -l, -e) and the dozens of recipes built on them; it is one of the highest-leverage things to learn coming from a scripting background where this niche was someone else's job.

Gotchas coming from Lua

The traps, collected for a final scan before you start writing:

- **One table becomes two types.** A Lua table is an @array (integer-keyed, ordered) or a %hash (string-keyed, unordered) in Perl, never both at once. Decide which you have before you reach for it.
- **Arrays are 0-based.** Lua's `t[1]` is Perl's `t[0]`; the last element is `t[-1]`. This is the most common porting off-by-one.
- **Sigils change with usage.** One element of @a is `$a[0]`; one value of %h is `$h{k}`. The container sigil and the element sigil differ - the bare Lua name `t` has no such shift.
- **Concatenation is ., not ...** Lua's `..` does not exist; `.` joins strings and `x` repeats them.
- **Truthiness is inverted at zero.** In Lua `0` and `""` are *true*; in Perl they are *false*. Only `0`, `""`, `"0"`, and `undef` are false, so `"00"` and `"0.0"` are surprisingly *true*.
- **Two comparison families.** `==/</>=` for numbers, `eq/lt/cmp` for strings. Pick by operand type; Lua's single `==` has no equivalent split.
- **sort is string-sorted by default.** Pass `{ $a <=> $b }` for numeric order.
- **Context.** `my $x = @a` is the *length*; `my ($x) = @a` is the *first element*. The list form is your friend `local a, b = f()`; the scalar form is new. Re-read *Context*

when a value comes back the wrong shape.

- **Regex is not Lua patterns.** Perl has full regex (`\d`, `|`, `*`, named groups); Lua patterns (`%d`, no alternation) are a different, smaller language. `s///` edits in place and returns a count; use `/r` for a new string.
- **open does not raise.** Use `open ... or die "...: $!",` the way Lua's `io.open` returns `nil, err`.
- **chomp your line reads.** `<$fh>` keeps the trailing newline that Lua's `lines()` would have stripped.
- **Non-ASCII source needs `use utf8`;** Under `use v5.42`; a literal non-ASCII byte in your source is a compile error without it. Keep string data ASCII unless you specifically add `use utf8`;
- **The `class` feature prints experimental notices to `stderr`** on `ppperl`. Correct output, noisy error stream. Operator metamethods map to the `overload` pragma, not to `class`.

Sed to Perl

You know `sed`. You think in `s/old/new/`, you address lines with `/pattern/` and `$`, you reach for `-n` and `p` to print selectively, and `-i` to edit files in place. This guide maps each of those habits onto Perl, so you can keep every reflex you have and pick up the things `sed` makes hard.

The one idea to carry in from the first line: **Perl is a near-superset of sed for text work.** Everything `sed` does, Perl does - usually with the *same* `s///` syntax - plus a real regex engine, real variables, and arbitrary code in the replacement. The awkward corners of `sed` (the hold space, multi-line tricks, anything resembling arithmetic) become ordinary Perl that you would write without thinking. This guide leads with the one-to-one mappings and then shows where Perl simply removes a `sed` limitation.

Every runnable example here was executed on `ppperl`, a Rust reimplement of Perl 5.42; the `# ...` comments show its real output. Start scripts with:

```
use v5.42;
```

That one line turns on `strict`, warnings, and the `say` built-in (a `print` with a trailing newline), the modern feature set used throughout.

Note

One-liner switches on this `ppperl` build `sed`'s natural Perl twin is the one-liner: `perl -ne` and `perl -pe`. Those switches are **correct perl5 5.42** and are shown throughout this guide as the idiomatic form. But the implicit-loop switch family (`-n`, `-p`, `-a`, `-F`, `-i`, `-l`) is **not yet recognised by this pppperl build** - only `-e` is. So the one-liner examples below are shown *without* output, and each is paired with an explicit-loop full-script form (`while (<$fh>) { ... }`) that **does** run on `ppperl` today and carries the verified output. Use the full-script form until the switches land.

How to read this

- *The line cycle* - sed's implicit loop becomes `while (<$fh>)` and `$_`.
- *Substitution*: `s///` - the same operator, with real regex.
- *Transliteration*: `y///` becomes `tr///` - the `y` command, renamed.
- *Addresses and ranges* - line numbers, patterns, and `/a/ .. /b/`.
- *Deleting, printing, quitting*: `d`, `p`, `q` - next, print, last.
- *The hold space becomes a variable* - where sed is hard and Perl is trivial.
- *In-place editing and I/O* - `-i`, reading files, pipes.
- *One-liners and switches* - `-n`, `-p`, `-e`, `BEGIN/END`.
- *Gotchas coming from sed* - scan before you write.

The line cycle

sed runs your script once per input line, with the current line in its pattern space, and prints the pattern space at the end of the cycle (unless `-n`). Perl's equivalent is an explicit loop with the line in `$_`, the implicit topic variable. The `-p` switch reproduces “auto-print every line”; `-n` reproduces “print nothing unless I say so.”

```
sed 's/bar/BAR/' input.txt
```

```
perl -pe 's/bar/BAR/' input.txt
```

The runnable, full-script form of that same loop - the one that runs on pperl today - opens the file and iterates it explicitly. `$_` is the current line, exactly like sed's pattern space:

```
use v5.42;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (<$fh>) {
    # each line lands in $_
    s/a/A/g;           # operate on $_ by default
    print;             # print $_ (note: keeps the newline)
}
# Apple
# bAnAnA
# cherry
```

(The input file held apple, banana, cherry.) The gotcha: a line read with `<$fh>` keeps its trailing newline, just as sed's pattern space does. `print` with no argument prints `$_` including that newline, so it round-trips cleanly; `chomp` only when you intend to strip it. The line number is in `$.`, the sed analogue of the line counter you reach for with `$=` in GNU sed scripts.

Substitution: s///

This is the headline: **sed's s/// and Perl's s/// are the same operator**. The flags line up too - g is global, i is case-insensitive - and Perl adds flags sed has no equivalent for.

```
sed 's/colou\?r/COLOR/g' input.txt
```

```
perl -pe 's/colou?r/COLOR/g' input.txt
```

By default s/// edits its target in place and returns the count of substitutions. Captured groups are \$1, \$2, ... and you can run **arbitrary code** in the replacement with the /e flag - something sed cannot do at all:

```
use v5.42;
my $s = "items: 3 and 4";
(my $t = $s) =~ s/(\d+)/$1 * 10/ge;    # /e: replacement is Perl code
say $t;                               # items: 30 and 40
```

To rearrange captures (sed's \1, \2 backreferences in the replacement), use \$1, \$2 and, when you want a *new* string instead of editing in place, the /r flag, which returns the result and leaves the original untouched:

```
use v5.42;
my $date = "2026-06-21";
my $us = $date =~ s/(\d+)-(\d+)-(\d+)/{$2/$3/$1}r;    # /r: return a
→copy                                                # 06/21/2026
say $us;                                              # 2026-06-21
say $date;
```

The gotcha: bare s/// modifies its target and returns a *count*, not the new string. That is why the examples above either copy first ((my \$t = \$s) =~ s/.../.../) or use /r. Writing my \$new = (\$s =~ s/a/b/) puts the substitution count in \$new, not the edited string - a classic first-day surprise. Note also that Perl delimiters are free: s{...}{...} and s#...#...# avoid backslashing slashes in a path pattern.

Transliteration: y/// becomes tr///

sed's y/abc/xyz/ is Perl's tr/abc/xyz/. Perl keeps y/// as an exact alias, so y/a-z/A-Z/ A-Z/ works verbatim, but tr is the spelled name you will see in Perl code.

```
sed 'y/abc/xyz/' input.txt
```

```
use v5.42;
my $s = "hello";
(my $t = $s) =~ tr/a-z/A-Z/;    # uppercase, like y/a-z/A-Z/
say $t;                        # HELLO
```

tr also does things y cannot. With an empty replacement it *counts* matches, and the s modifier squeezes runs of a character - the common tr -s idiom:

```

use v5.42;
my $s = "hello";
my $vowels = ($s =~ tr/aeiou//); # count, replacement empty
say $vowels;                     # 2

my $dots = "a....b...c";
(my $squeezed = $dots) =~ tr/././s; # squeeze repeats
say $squeezed;                   # a.b.c

```

The gotcha: `tr///` is **not** a regex - the left side is a literal character set, the same as `sed's` `y`. `tr/a-z//` means the range `a` through `z`, but `tr/.*//` means the two literal characters `.` and `*`, not “any string.” For pattern work, reach for `s///`. And like `s///`, `tr///` edits in place and returns a count, so copy first or use the `/r` flag for a non-destructive transliteration.

Addresses and ranges

A `sed` command can be prefixed by an address: a line number, `$` for the last line, a `/pattern/`, or a `addr1, addr2` range. In Perl an address is just a condition on the statement - a postfix `if`. Line number is `$.`; the pattern test is `/pat/` against `$_`; the last line is `eof`.

```

sed -n '2p' input.txt          # print line 2
sed -n '/banana/p' input.txt   # print matching lines

```

```

use v5.42;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (<$fh>) {
    print if $. == 2;          # sed -n '2p'
}
# banana

```

A `sed` range `/start/,/end/` becomes Perl's **flip-flop** range operator `/start/ .. /end/`, which is true from the line matching the left pattern through the line matching the right, inclusive:

```

use v5.42;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (<$fh>) {
    print if /banana/ .. /cherry/; # sed '/banana/,/cherry/p' with
    ↪ -n
}
# banana
# cherry

```

The gotcha: in a boolean test `..` is the flip-flop (range) operator, **not** the list-range operator (which is `1 .. 10` building a list). Perl tells them apart by context - in the scalar/boolean position of an `if`, `..` is the line-range selector that mirrors `sed's` `addr1, addr2`. It is stateful: it remembers whether it has seen the start pattern, so it does exactly what a `sed` range does across the loop.

Deleting, printing, quitting: d, p, q

sed's single-letter cycle commands map to Perl loop keywords. d (delete pattern space, skip to next cycle) is next. q (quit) is last. p (print) is print. Because Perl's loop is explicit, these read as exactly what they do.

```
sed '/banana/d' input.txt      # drop matching lines
sed '/banana/q' input.txt     # stop after the match
```

```
use v5.42;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (<$fh>) {
    next if /banana/;          # sed '/banana/d'
    print;
}
# apple
# cherry
```

```
use v5.42;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (<$fh>) {
    print;
    last if /banana/;         # sed '/banana/q'
}
# apple
# banana
```

The gotcha: under sed's -p-style auto-print, d must skip the auto-print, which it does by aborting the cycle. In an explicit Perl loop you control printing yourself, so “delete” is simply *not printing* the line - next if /pat/ before the print, or guard the print with unless. There is no hidden auto-print to fight; what you print is what comes out.

The hold space becomes a variable

This is the section where sed stops being convenient and Perl stops being hard. sed has exactly one auxiliary buffer - the hold space - and shuffles data into it with h, H, g, G, x. Anything needing two pieces of remembered state turns into hold-space contortions. In Perl you just declare a variable. Or an array. Or a hash.

The canonical hold-space exercise is reversing a file (tac), which in sed is an opaque 1!G;h;\$!d incantation. In Perl it is one obvious line:

```
sed -n '1!G;h;$!d' input.txt  # reverse the file (tac)
```

```
use v5.42;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
my @hold;                                # the "hold space" is just an
    ↪ array
while (<$fh>) {
    unshift @hold, $_;                  # newest line to the front
}
```

(continues on next page)

(continued from previous page)

```
print @hold;
# cherry
# banana
# apple
```

The gotcha: there is no gotcha - that is the point. Once you have real variables, the entire category of sed problems that exist only because there is a single hold space disappears. Need to remember the previous line? my \$prev. Need a running total? my \$sum. Need to dedup? A my %seen hash. When a sed script reaches for h/H/x, the Perl translation is almost always “introduce a named variable and write the obvious thing.”

In-place editing and I/O

sed's -i rewrites a file in place. The idiomatic Perl is the same switch on a -p one-liner. Reading several files is sed's default; Perl spells it <>, the diamond operator, which reads each file named in @ARGV in turn.

```
sed -i 's/red/crimson/' file.txt # edit in place
```

```
perl -i -pe 's/red/crimson/' file.txt
```

The runnable equivalent today is an explicit read-modify-write: slurp the lines, transform them, write them back. This is exactly what -i does under the hood, and it runs on perl now:

```
use v5.42;
my $path = '/tmp/inplace_data.txt'; # holds: red green blue

open my $in, '<', $path or die "read: $!";
my @lines = <$in>; # slurp all lines (list_
    context)
close $in;

s/e/E/g for @lines; # transform each line's $_

open my $out, '>', $path or die "write: $!";
print {$out} @lines; # write them back
close $out;
# file now holds:
# rEd
# grEEen
# bluE
```

For pipes - sed in the middle of a shell pipeline - Perl opens a pipe filehandle directly: open my \$fh, '-|', 'ls', '-l' reads a command's output, and open my \$fh, '|-', 'sort' writes to a command's input. The gotcha: open returns false on failure rather than raising, so the or die "...: \$!" idiom is mandatory - \$! is the system error string, sed's silent failure made explicit. And reading multiple files with <> resets \$. per file only if you check eof explicitly; the plain \$. keeps counting across the concatenated stream.

One-liners and switches

The one-liner is sed's home turf and Perl's too. The switch family that makes it work: -e supplies the program, -n wraps it in a no-auto-print loop (sed's -n), -p wraps it in an auto-print loop (sed's default), -i edits in place, -l handles line endings, and -a/-F autosplit each line into @F (awk territory, useful here for field edits). BEGIN { } and END { } blocks run before and after the loop, the direct analogue of sed's 1i/\$a setup-and-teardown tricks.

```
sed -n '/error/p' log.txt           # print error lines
sed 's/[0-9]\+/N/g' input.txt      # mask numbers
```

```
perl -ne 'print if /error/' log.txt # print error lines
perl -pe 's/\d+/N/g' input.txt      # mask numbers
perl -nE 'END { say $n } $n++ if /error/' log.txt # count, BEGIN/
↪END
```

These are shown without output because the implicit-loop switches are not recognised by this pperl build yet (see the note at the top). Each maps to a verified explicit-loop form: perl -ne 'CODE' is while (<\$fh) { CODE }, and perl -pe 'CODE' is the same with a trailing print. The runnable “count error lines” form:

```
use v5.42;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
my $n = 0;
while (<$fh) {
    $n++ if /banana/;           # the body of -ne
}
say $n;                        # 1
```

The gotcha: until the switches land on pperl, write the explicit loop (it runs today and is what the switches expand to anyway), and consult the *One-Liners guide* for the full switch catalogue and dozens of recipes - it is the highest-leverage thing to read coming from a sed background. Perl also has full subroutines, references, modules, and OO if your script outgrows a one-liner; see the *Object-oriented Programming guide* for that side of the language.

Gotchas coming from sed

The traps, collected for a final scan before you write:

- **s/// returns a count, not the new string.** my \$x = \$s =~ s/a/b/ puts the substitution count in \$x. Copy first ((my \$t = \$s) =~ s/.../.../) or use the /r flag to get a new string.
- **tr/// (sed's y) is a literal character set, not a regex.** tr/.*/ matches the two characters . and *. For patterns, use s///.
- **.. in a boolean is the flip-flop range, not a list range.** In an if, /a/ .. /b/ is the sed addr1,addr2 selector; 1 .. 10 in list context builds a list. Same operator, decided by context.
- **<\$fh> keeps the trailing newline,** like sed's pattern space. print round-trips it; chomp only when you mean to strip it.

- **There is no hidden auto-print.** sed's -p default is something you spell yourself with print. "Delete a line" is just not printing it (next if /pat/), not a special command.
- **open does not raise on failure.** Use open my \$fh, '<', \$path or die "...: \$!". \$! is the system error string.
- **The hold space is gone - use a variable.** Any sed script reaching for h/H/x translates to a named Perl variable, array, or hash. This is a simplification, not a loss.
- **The implicit-loop switches (-n/-p/-a/-F/-i/-l) are not yet recognised by this pperl build.** Only -e works today. Write the explicit while (<\$fh) { ... } loop, which is exactly what the switches expand to, until they land.

Awk to Perl

You know awk. You think in pattern { action }, you split every line into \$1, \$2, \$3 without asking, you count things with a[key]++, and you frame the whole job with BEGIN { } and END { }. Perl was written partly as "awk, but bigger" - almost everything you do in awk has a direct Perl form, and then Perl keeps going: real data structures, full regex, modules, and a language that does not run out of room when the script grows.

The one idea to carry in from the first line: **Perl is a near-superset of awk.** The record loop, the fields, the associative arrays, the BEGIN/END keywords, print and printf, length/substr/split

- they all map across, often by the *same name*. What changes is that fields become a real array @F you can push and pop, the associative array becomes a Perl hash you can nest, and the regex engine is PCRE rather than POSIX ERE. This guide leads with the one-to-one mappings and points out the few places the analogy leaks.

Every runnable example here was executed on pperl, a Rust reimplementation of Perl 5.42; the # ... comments show its real output. Start scripts with:

```
use v5.42;
```

That one line turns on strict, warnings, and the say built-in (a print with a trailing newline), the modern feature set used throughout.

Note

One-liner switches on this pperl build awk's natural Perl twin is the one-liner: `perl -ne`, `perl -ane`, and `perl -F, -ane`. Those switches are **correct perl5 5.42** and are shown throughout this guide as the idiomatic form. But the implicit-loop switch family (`-n`, `-p`, `-a`, `-F`, `-i`, `-l`) is **not yet recognised by this pperl build** - only `-e` is. So the one-liner examples below are shown *without* output, and each is paired with an explicit-loop full-script form (`while (<$fh>) { ... }`) that **does** run on pperl today and carries the verified output. Use the full-script form until the switches land.

How to read this

- *The line/record loop* - awk's implicit `{ ... }` becomes `while (<$fh>)` and `$_`.
- *Fields and splitting* - `$1/$NF/NF/FS` become `@F`, `$F[0]`, `split`, `-F`.
- *Regex and substitution* - `gsub/sub/ match` become `s///=~`, a strict superset.
- *Addresses and ranges* - `/re/`, `cond { }`, and `/a/`, `/b/` become postfix `if` and the flip-flop.
- *I/O, pipes, in-place edit* - `getline`, redirection, and `-i`.
- *One-liners and switches* - `-n`, `-a`, `-F`, `BEGIN/END`.
- *Control flow* - `next`, `for`, the postfix forms.
- *Variables and data* - associative arrays become hashes, the happy mapping.
- *Gotchas coming from awk* - scan before you write.

The line/record loop

awk runs your program once per input record, with the record in `$0` and the action in `{ ... }`. Perl's equivalent is an explicit loop with the record in `$_`, the implicit topic variable. The `-n` switch reproduces awk's "run this block per line"; awk's NR (record number) is Perl's `$.`

```
awk '{ print }' input.txt
```

```
perl -ne 'print' input.txt
```

The runnable, full-script form of that same loop - the one that runs on pperl today - opens the file and iterates it explicitly. `$_` is the current record, exactly like awk's `$0`:

```
use v5.42;
open my $fh, '<', '/tmp/scores.txt' or die "open: $!";
while (<$fh>) {
    # each line lands in $_, like awk's $0
    chomp;
    say "line $. : $_";    # $. is awk's NR
}
# line 1 : alice 85 90
# line 2 : bob 70 60
```

(continues on next page)

(continued from previous page)

```
# line 3 : carol 95 100
# line 4 : alice 50 55
```

(The input file held alice 85 90, bob 70 60, carol 95 100, alice 50 55.) The gotcha: a line read with `<$fh>` keeps its trailing newline, unlike awk's `$0`, which has the record separator stripped. Call `chomp` to strip it (`chomp` is the closest thing to “awk already removed the newline for you”). The record number is `$. = awk's NR`; when you read several files with `<>`, `$.` keeps counting across them, so the awk FNR (per-file counter) reset is something you arrange yourself by checking `eof`.

Fields and splitting

This is awk's heart, and Perl gives you the same picture with one twist: the fields live in a real array `@F`, so awk's `$1` is Perl's `$F[0]` (zero-based), `$0` is `$_`, and `NF` is the count scalar `@F`. Under the `-a` switch Perl autosplits each line into `@F` exactly as awk autosplits into `$1..$NF`; the runnable equivalent is one explicit `split`.

```
awk '{ print $1, NF }' input.txt      # first field, field count
awk -F, '{ print $1 }' input.txt     # comma separator (FS)
```

```
perl -ane 'print "$F[0] @{$[scalar @F]}\n"' input.txt  # -a fills @F
perl -F, -ane 'print "$F[0]\n"' input.txt             # -F sets the
→split pattern
```

The full-script form does the split itself. Bare `split` with no arguments splits `$_` on whitespace and discards leading blanks - exactly awk's default FS:

```
use v5.42;
open my $fh, '<', '/tmp/scores.txt' or die "open: $!";
while (<$fh>) {
    my @F = split;                # awk's automatic field_
→split
    say "$F[0] has ", scalar(@F) - 1, " scores"; # $F[0]=$1, @F=Nf
}
# alice has 2 scores
# bob has 2 scores
# carol has 2 scores
# alice has 2 scores
```

Arithmetic across fields is the everyday awk action `print $1, $2+$3`, and it reads almost identically:

```
use v5.42;
open my $fh, '<', '/tmp/scores.txt' or die "open: $!";
while (<$fh>) {
    my @F = split;
    say "$F[0]: ", $F[1] + $F[2]; # awk: print $1, $2+$3
}
# alice: 175
```

(continues on next page)

(continued from previous page)

```
# bob: 130
# carol: 195
# alice: 105
```

awk's `$NF` (the last field) is Perl's negative index `$F[-1]`, and a non-default FS is the pattern argument to `split`:

```
use v5.42;
open my $fh, '<', '/tmp/data.csv' or die "open: $!"; # alice,85,90
→...
while (<$fh) {
    chomp;
    my @F = split /,/;      # awk -F,
    say $F[-1];            # awk's $NF: last field
}
# 90
# 60
# 100
```

To join fields back with a separator (awk's OFS), set `$,` (the output field separator that `print`/`say` inserts between its list arguments):

```
use v5.42;
open my $fh, '<', '/tmp/data.csv' or die "open: $!";
while (<$fh) {
    chomp;
    my @F = split /,/;
    local $, = "|";        # OFS
    say @F;                # join the list with OFS
}
# alice|85|90
# bob|70|60
# carol|95|100
```

The gotcha: fields are **zero-based**. awk's `$1` is `$F[0]`, `$2` is `$F[1]`, and `$0` (the whole record) is `$_`, not an element of `@F`. `NF` is scalar `@F` (or `$#F + 1`); the last field is `$F[-1]`, not `$F[NF]`. And `split` with **one space as a string** (`split ' '`) is the special whitespace-and-strip mode; `split / /` (a regex with one space) is literal and does *not* strip leading blanks. Plain `split` with no argument is the awk-default mode you want most of the time.

Regex and substitution

awk's `sub`, `gsub`, `match`, and `~` all map onto Perl's single regex syntax: `=~` binds a pattern to a string, `s///` substitutes, `s///g` is global (awk's `gsub` vs `sub`). Perl's engine is PCRE, a strict superset of awk's POSIX ERE, so every awk pattern works and Perl adds backreferences, lookahead, named captures, and the `/e`, `/r`, `/x` flags awk has no equivalent for.

```
awk '/^alice/ { sub(/alice/, "ALICE"); print }' input.txt
```

```
perl -ne 'if (/^alice/) { s/alice/ALICE/; print }' input.txt
```

The full-script form:

```
use v5.42;
open my $fh, '<', '/tmp/scores.txt' or die "open: $!";
while (<$fh>) {
    next unless /^alice/;    # awk: /^alice/ { ... }
    s/alice/ALICE/;          # awk: sub(/alice/, "ALICE") (first_
    ↪match)
    print;
}
# ALICE 85 90
# ALICE 50 55
```

Use `s///g` for awk's `gsub` (every match), and capture groups land in `$1`, `$2` just as awk's match populates `RSTART`/`RLENGTH` plus the gawk array. The `/e` flag runs **arbitrary Perl** in the replacement, which awk cannot do:

```
use v5.42;
my $s = "items: 3 and 4";
(my $t = $s) =~ s/(\d+)/$1 * 10/ge;    # /e: replacement is Perl code
say $t;                                # items: 30 and 40
```

length, substr, index, and the case functions map by name - watch the indexing:

```
use v5.42;
my $s = "alice";
say length $s;                # 5          (awk length($s))
say substr $s, 0, 3;          # ali        (awk substr($s,1,3) - awk is 1-
    ↪based!)
say uc $s;                    # ALICE      (awk toupper($s))
```

The gotcha: `substr` is **zero-based** where awk's `substr` is one-based, so awk `substr($s,1,3)` is Perl `substr($s,0,3)`. And `s///` edits its target in place and returns the **count** of substitutions, not the new string - so `my $new = ($s =~ s/a/b/)` puts a number in `$new`. Copy first `((my $t = $s) =~ s/.../.../)` or use the `/r` flag to get a fresh string. POSIX character classes `[[:digit:]]` work, but the Perl shorthands `\d`, `\w`, `\s` are shorter and what you will see in Perl code.

Addresses and ranges

An awk rule is `pattern { action }`: the action runs only on records matching the pattern. In Perl that pattern is just a condition on the statement - a postfix `if` testing `$_`. A bare `/re/` matches against `$_`, a numeric test uses `$`. (awk's `NR`), and awk's range `/start/,/end/` is Perl's **flip-flop** operator `/start/ .. /end/`.

```
awk 'NR==2'          input.txt      # print record 2
awk '/banana/'       input.txt      # print matching records
awk '/bob/,/carol/'  input.txt      # range, inclusive
```

```
perl -ne 'print if $. == 2'          input.txt
perl -ne 'print if /banana/'         input.txt
perl -ne 'print if /bob/ .. /carol/' input.txt
```

The full-script flip-flop, true from the line matching the left pattern through the line matching the right, inclusive - exactly awk's two-pattern range:

```
use v5.42;
open my $fh, '<', '/tmp/scores.txt' or die "open: $!";
while (<$fh) {
    print if /bob/ .. /carol/;    # awk: /bob/,/carol/
}
# bob 70 60
# carol 95 100
```

The gotcha: a bare pattern with no action in awk means “print the record”; in Perl the loop is explicit, so you write the print (or say) yourself. And in a boolean test `..` is the flip-flop (range) operator, **not** the list-range operator (`1 .. 10` builds a list). Perl tells them apart by context: in the scalar position of an if, `..` is the stateful line-range selector that mirrors awk's `addr1,addr2`.

I/O, pipes, in-place edit

awk reads every file named on its command line automatically and has `getline` for explicit reads. Perl's `<>` (the diamond operator) reads each file in `@ARGV` in turn - the same default - and `<$fh>` is the explicit `getline`. Writing to a command or reading from one (awk's `print | "cmd"` and `"cmd" | getline`) is a pipe filehandle.

```
awk '{ print > "out.txt" }' input.txt      # redirect output
awk '{ print | "sort" }'    input.txt      # pipe to a command
```

```
use v5.42;
open my $out, '>', '/tmp/out.txt' or die "write: $!";
open my $in,  '<', '/tmp/scores.txt' or die "read: $!";
while (<$in) {
    print {$out} $_;          # awk: print > "out.txt"
}
close $out;
```

For pipes, `open my $fh, '|-', 'sort'` writes to a command's input and `open my $fh, '-|', 'ls', '-l'` reads a command's output - awk's `| "cmd"` in both directions.

awk has no built-in in-place edit; you reach for a tmp-file shuffle or GNU `gawk -i in-place`. Perl spells it `-i` on a one-liner:


```
perl -i -pe 's/red/crimson/' file.txt      # edit in place
```

The runnable equivalent today is an explicit read-modify-write - slurp the lines, transform, write back - which is exactly what `-i` does under the hood:

```
use v5.42;
my $path = '/tmp/inplace.txt';          # holds: red green blue

open my $in, '<', $path or die "read: $!";
my @lines = <$in>;                      # slurp all lines (list_
    context)
close $in;

s/e/E/g for @lines;                    # transform each line's $_

open my $out, '>', $path or die "write: $!";
print {$out} @lines;                  # write them back
close $out;
# file now holds: rEd / grEEEn / bluE
```

The gotcha: `open` returns false on failure rather than aborting, so the `or die "...: $!"` idiom is mandatory - `$!` is the system error string, `awk`'s silent failure made explicit. And `print {$out} ...` needs the braces around the filehandle when it is a variable; the bare `print FH ...` form (no comma) is for named handles, which is a common first-day stumble.

One-liners and switches

The one-liner is `awk`'s home turf and Perl's too. The switch family that reproduces `awk`: `-e` supplies the program, `-n` wraps it in a per-record loop (`awk`'s implicit `{ }`), `-p` does the same with an auto-print at the end of each cycle, `-a` autosplits into `@F` (`awk`'s automatic field split), `-F` sets the split pattern (`awk`'s `FS`), and `-l` handles line endings (`awk`'s `ORS/RS` convenience). `BEGIN { }` and `END { }` are the **same keywords** as `awk`, running before and after the loop.

```
awk '/error/ { print }'      log.txt      # print error_
    records
awk '{ s += $NF } END { print s }' input.txt # sum the last field
awk -F, '{ print $1 }'      input.txt     # first CSV field
```

```
perl -ne 'print if /error/'      log.txt
perl -ane 'END { print "$s\n" } $s += $F[-1]' input.txt
perl -F, -ane 'print "$F[0]\n"'  input.txt
```

These are shown without output because the implicit-loop switches are not recognised by this `pperl` build yet (see the note at the top). Each maps to a verified explicit-loop form: `perl -ne 'CODE'` is `while (<$fh>) { CODE }`, `-ane` adds `my @F = split` at the top, and `BEGIN/END` blocks work in a full script exactly as in `awk`. The runnable “sum a column with `BEGIN/END`” form:

```

use v5.42;
my $n = 0;
my $sum = 0;
BEGIN { say "start" }           # runs before the loop, like awk
↳BEGIN
END { say "lines=$n sum=$sum" } # runs after, like awk END
open my $fh, '<', '/tmp/scores.txt' or die "open: $!";
while (<$fh>) {
    my @F = split;
    $n++;
    $sum += $F[1];              # awk: sum += $2
}
# start
# lines=4 sum=300

```

The gotcha: until the switches land on pperl, write the explicit loop (it runs today and is what the switches expand to anyway), and consult the *One-Liners guide* for the full switch catalogue and dozens of recipes - it is the highest-leverage thing to read coming from an awk background. Perl also has full subroutines, references, modules, and OO if your script outgrows a one-liner; see the *Object-oriented Programming guide* and the other coming-from guides.

Control flow

awk's control keywords map onto Perl's almost name-for-name. awk's next (skip to the next record) is Perl's next; break/continue in awk loops are Perl's last/next; awk's for (k in arr) becomes a Perl for over keys. The braces and ; are the same; Perl adds the postfix forms that read like awk patterns.

```
awk '{ if ($2 < 70) next; print }' input.txt
```

```

use v5.42;
open my $fh, '<', '/tmp/scores.txt' or die "open: $!";
while (<$fh>) {
    my @F = split;
    next if $F[1] < 70;          # awk: if ($2 < 70) next
    print;
}
# alice 85 90
# carol 95 100

```

The gotcha: Perl's comparison operators come in two families and you pick by the *operands' type*, where awk has only </= for both. Numbers use ==, <, >, <=>; strings use eq, lt, gt, cmp. awk decides numeric-vs-string comparison by the values at run-time; Perl makes you choose the operator. Using == on strings (or eq on numbers) is a frequent, quiet bug - "abc" == "xyz" is true in Perl (both numify to 0), which is almost never what you meant.

Variables and data

This is the happiest mapping in the guide: awk's associative array is a Perl **hash**, and the daily idioms are near-identical. awk's `count[$1]++` is Perl's `$count{$F[0]}++`; `for (k in count)` is `for my $k (keys %count)`; `if (k in count)` is `exists $count{$k}`. The one shift is the sigil - the whole hash is `%count`, but one value is `$count{$key}`.

```
awk '{ total[$1] += $2 + $3 }
     END { for (k in total) print k, total[k] }' input.txt
```

```
use v5.42;
open my $fh, '<', '/tmp/scores.txt' or die "open: $!";
my %total;
while (<$fh>) {
    my @F = split;
    $total{$F[0]} += $F[1] + $F[2];    # awk: total[$1] += $2 + $3
}
for my $name (sort keys %total) {    # awk END loop over the array
    say "$name $total{$name}";
}
# alice 280
# bob 130
# carol 195
```

The two-file `NR==FNR` idiom - load keys from the first file, then test membership in the second - is one explicit loop per file in Perl, which is clearer than the awk one-liner it replaces:

```
use v5.42;
my %seen;
open my $f1, '<', '/tmp/data.csv' or die "open: $!";
while (<$f1>) {
    my ($k) = split /,/;            # first field only
    $seen{$k} = 1;
}
close $f1;
open my $f2, '<', '/tmp/scores.txt' or die "open: $!";
while (<$f2>) {
    my ($k) = split;
    print if $seen{$k};             # awk: ($1 in seen)
}
# alice 85 90
# bob 70 60
# carol 95 100
# alice 50 55
```

The gotcha: a hash has no inherent order (awk's `for (k in arr)` is also unordered, so this will not surprise you), so iterate `sort keys %h` when you want a stable order. Membership is `exists $h{$k}`, not awk's `k in arr` - and merely *reading* `$h{$k}` to test it will autovivify the key into existence in some contexts, where `exists` never does. Numbers and strings auto-coerce in Perl as in awk, but remember the `==` vs `eq` split from

Control flow.

Gotchas coming from awk

The traps, collected for a final scan before you write:

- **Fields are zero-based.** awk's \$1 is \$F[0], \$2 is \$F[1], and the whole record \$0 is \$_. NF is scalar @F; the last field is \$F[-1], not \$F[NF].
- **<\$fh> keeps the trailing newline**, unlike awk's \$0. chomp to strip it.
- **substr is zero-based**, where awk's is one-based: awk substr(\$s,1,3) is Perl substr(\$s,0,3).
- **s/// returns a count, not the new string.** my \$x = \$s =~ s/a/b/ puts the substitution *count* in \$x. Copy first, or use the /r flag to get a new string.
- **Two comparison families.** Numbers use ==/</<=>; strings use eq/lt/cmp. awk's single ==/< is split in two, chosen by operator, not by value. "abc" == "xyz" is true (both numify to 0).
- **Bare split is the awk-default mode.** split (no args) splits \$_ on whitespace and strips leading blanks. split / / (a regex with one space) is literal and does *not* strip - use plain split for awk's default FS.
- **Membership is exists \$h{\$k}, not k in arr.** Testing a key by reading \$h{\$k} can autovivify it; exists never does.
- **open does not abort on failure.** Use open my \$fh, '<', \$path or die "... : \$!". \$! is the system error string.
- **The implicit-loop switches (-n/-p/-a/-F/-i/-l) are not yet recognised by this pperl build.** Only -e works today. Write the explicit while (<\$fh) { my @F = split; ... } loop, which is exactly what the switches expand to, until they land.

Shell to Perl

You know the shell. You glue programs together with pipes, you reach for sed, awk, grep, and cut in a pipeline without thinking, you capture command output with \$(...), and you script control flow with for f in *, while read line, and case. This guide maps each of those habits onto Perl, so you can keep the pipeline reflexes and pick up the things the shell makes painful.

When a shell script grows past about 50 lines of quoting and sed/awk/grep pipelines, Perl is the natural next step: it does all of it in one language, with real data structures instead of whitespace-delimited strings. The framing to carry in: **Perl is the shell pipeline, but the filter language is a real language.** You still open pipes, you still glob, you still run commands and read their output. What changes is that between the pipes you now have @arrays, %hashes, real regex, and arithmetic that needs no \$(). The shell's one data type is the string; Perl gives you actual containers.

Every runnable example here was executed on pperl, a Rust reimplementation of Perl 5.42; the # ... comments show its real output. Start scripts with:

```
use v5.42;
```

That one line turns on `strict`, warnings, and the `say` built-in (a `print` with a trailing newline), the modern feature set used throughout.

Note

One-liner switches on this pperl build The shell developer's natural Perl twin is the one-liner: `perl -ne` and `perl -pe`, the in-the-pipeline filter. Those switches are **correct perl5 5.42** and are shown throughout this guide as the idiomatic form. But the implicit-loop switch family (`-n`, `-p`, `-a`, `-F`, `-i`, `-l`) is **not yet recognised by this pperl build** - only `-e` is. So the one-liner examples below are shown *without* output, and each is paired with an explicit-loop full-script form (`while (<$fh>) { ... }`) that **does** run on pperl today and carries the verified output. Use the full-script form until the switches land.

How to read this

- *The record loop* - `while read line` becomes `while (<$fh>)` and `$_`.
- *Fields and splitting* - `cut/IFS/awk` fields become `split` and a real array.
- *Regex and substitution* - the shell's `${var#...}/${var/a/b}` parameter expansions become real regex.
- *Addresses and ranges* - `grep -n`, `awk NR==2`, and line ranges.
- *I/O, pipes, and in-place edit* - the heart: `open`, `-||-`, redirection, here-docs, `$(...)`.
- *One-liners and switches* - `-n`, `-p`, `-e`, `BEGIN/END`.
- *Control flow* - `for/while/case/if`, file tests, exit status.
- *Variables and data* - `$var` becomes `my $var`, plus the `@arrays` and `%hashes` the shell never had.
- *Gotchas coming from the shell* - scan before you write.

The record loop

The shell's line-at-a-time idiom is `while read line; do ... done`, which reads one line per iteration into `$line` (and strips it according to `IFS`). Perl's equivalent is `while (<$fh>)`, which reads one line per iteration into `$_`, the implicit topic variable. The crucial difference: Perl keeps the trailing newline, so you `chomp` it yourself, exactly where the shell's `read` would have eaten it.

```
while read line; do
    echo "got: $line"
done < data.txt
```

```
use v5.42;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (my $line = <$fh>) {      # each line, with its newline
    chomp $line;               # strip it, like read does
    say "got: $line";
```

(continues on next page)

(continued from previous page)

```

}
close $fh;
# got: apple
# got: banana
# got: cherry

```

(The input file held apple, banana, cherry.) Read with the bare `<$fh>` and the line lands in `$_`, the default that most line built-ins operate on. The line number is in `$.` This is the form to reach for when you are translating a filter:

```

use v5.42;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (<$fh>) {
    chomp;          # line lands in $_
    say "$. : $_";  # $. is the line number
}
close $fh;
# 1 : apple
# 2 : banana
# 3 : cherry

```

The gotcha: `read` splits on IFS and trims; `<$fh>` does neither. A Perl line read is the whole raw line including the `\n`, so `chomp` is your reflex, and field-splitting is a separate, explicit step (the next section). What you lose in automatic trimming you gain in not having IFS quietly mangle your data.

Fields and splitting

In the shell you reach fields with `cut -d: -f1`, with `awk '{print $1}'`, or by setting IFS and letting `read` split for you. Perl's one answer is `split`: give it a separator and a string, get a real array back. Positional fields that the shell numbers `$1`, `$2` become array elements `$f[0]`, `$f[1]` - **0-based**, unlike `awk`'s 1-based `$1`.

```

line='root:x:0:0:root:/root:/bin/bash'
user=$(echo "$line" | cut -d: -f1)
shell=$(echo "$line" | cut -d: -f7)
echo "$user $shell"

```

```

use v5.42;
my $line = "root:x:0:0:root:/root:/bin/bash";
my @f = split /:/, $line;
say "user: $f[0]";          # user: root
say "shell: $f[-1]";       # shell: /bin/bash (negative index)
say "fields: ", scalar @f;  # fields: 7

```

To split on runs of whitespace - the shell's default word-splitting, and `awk`'s default - use the special pattern `split ' '` (a literal single-space string, which also skips leading blanks). To join fields back, `join`:

```
use v5.42;
my @words = split ' ', " the quick brown ";
say scalar @words;           # 3 (leading/inner blanks collapsed)
say join "|", @words;        # the|quick|brown
```

The gotcha: `split /:/` with a literal pattern keeps empty trailing fields only up to a point - by default Perl drops trailing empty fields, so `split /:/, "a::"` yields `("a")`, not `("a", "", "")`. Pass a negative limit (`split /:/, $s, -1`) to keep them, the equivalent of caring about trailing empty cut fields. And `split ' '` (the magic-whitespace form) is **not** the same as `split / /` (a single literal space): the first collapses runs and trims, the second does not.

Regex and substitution

The shell's `${}` parameter expansions become real regex - and this is a large upgrade. `${var#prefix}`, `${var%suffix}`, `${var/a/b}`, and `${var//a/b}` are each a special-purpose, limited string edit. Perl replaces the whole family with one operator, `s///`, that takes an actual regular expression and the full set of flags.

```
f='report.txt.bak'
base=${f%.bak}           # strip suffix
path='/usr/local/bin'
tail=${path##*/}         # basename
dir=${path%/*}           # dirname
s='a-b-c'
one=${s/-/_}             # replace first
all=${s//-/_}            # replace all
```

```
use v5.42;
my $f = "report.txt.bak";
(my $base = $f) =~ s/\.bak$//;    # ${f%.bak}
say $base;                       # report.txt

my $path = "/usr/local/bin";
(my $tail = $path) =~ s{.*//}{};  # ${path##*/}  basename
say $tail;                       # bin
(my $dir = $path) =~ s{/[^\/*]*$}{}; # ${path%/*}  dirname
say $dir;                        # /usr/local

my $s = "a-b-c";
(my $one = $s) =~ s/-/_/;         # ${s/-/_}    first only
say $one;                         # a_b-c
(my $all = $s) =~ s/-/_/g;        # ${s//-/_}  /g = global
say $all;                        # a_b_c
```

Because it is a real regex, the things the shell cannot do at all are trivial: capture groups land in `$1`, `$2`; the `/e` flag runs Perl code in the replacement; the `/r` flag returns a copy instead of editing in place; and `/i`, `/x` modify matching.


```

use v5.42;
my $s = "items: 3 and 4";
(my $t = $s) =~ s/(\d+)/$1 * 10/ge; # /e: replacement is Perl code
say $t;                               # items: 30 and 40

my $date = "2026-06-21";
my $us = $date =~ s{(\d+)-(\d+)-(\d+)}{$2/$3/$1}r; # /r: return a
→copy
say $us;                                     # 06/21/2026
say $date;                                   # 2026-06-21 (unchanged)

```

The gotcha: bare `s///` edits its target and returns the *count* of substitutions, not the new string - so the examples copy first with `(my $t = $s) =~ ...` or use the `/r` flag. Writing `my $new = ($s =~ s/a/b/)` puts a number in `$new`. This is the single most common first-day surprise. Note also that the delimiters are free: `s{...}{...}` and `s#...#...#` save you backslashing the slashes in a path pattern, the way the shell's `${path##*/}` already avoids them.

Addresses and ranges

Selecting lines by number or pattern - `grep`, `grep -n`, `awk 'NR==2'`, `sed -n '/a/,/b/p'` - becomes a postfix `if` on a condition inside the loop. The line number is `$_`; the pattern test is `/pat/` against `$_`.

```

grep -n an data.txt           # number + matching lines
awk 'NR==2' data.txt         # the second line

```

```

use v5.42;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (<$fh>) {
    chomp;
    say "$. : $_" if /an/;      # grep -n 'an'
}
close $fh;
# 2 : banana

```

A line range - the `sed/awk /start/,/end/` selector - becomes Perl's **flip-flop** range operator `/start/ .. /end/`, true from the line matching the left pattern through the line matching the right, inclusive:

```

use v5.42;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (<$fh>) {
    print if /banana/ .. /cherry/; # the /start/,/end/ range
}
close $fh;
# banana
# cherry

```

The gotcha: in a boolean test `..` is the flip-flop (range) operator, **not** the list-range operator (which is `1 .. 10` building a list). Perl tells them apart by context - in the boolean

position of an `if`, `..` is the stateful line-range selector; in list context it builds a sequence. Same two dots, decided by where you write them.

I/O, pipes, and in-place edit

This is the heart of the guide, because it is the heart of shell scripting. Everything you do with `|`, `>`, `>>`, `2>&1`, `<<EOF`, and `$(...)` has a direct Perl spelling, and once a script outgrows a single pipeline this is the section that pays off.

Command substitution. The shell's `$(cmd)` and backticks become Perl's backticks or `qx{...}`. The capture is context-sensitive: in scalar context you get the whole output as one string; in list context you get one element per line.

```
out=$(echo hello)
mapfile -t lines < <(seq 1 3)
```

```
use v5.42;
my $out = `echo hello`;           # scalar: whole output, one string
chomp $out;
say "got: $out";                  # got: hello

my @lines = qx{seq 1 3};          # list: one element per line
chomp @lines;
say "lines: @lines";              # lines: 1 2 3
```

When you do not need the output, `system()` runs a command and gives you its exit status (see *Control flow* for `$?`). Pass `system` a *list* and no shell is involved - the safe form that avoids quoting hell:

```
use v5.42;
system('mkdir', '-p', '/tmp/sh_demo'); # list form: no shell, no
→quoting
say "made it" if -d '/tmp/sh_demo';     # made it
```

Pipes. A pipeline stage - reading a command's output or feeding a command's input - is an open with a pipe mode. `open my $fh, '-|', LIST` reads a command's stdout; `open my $fh, '|-', LIST` writes to a command's stdin. The list form (command plus args as separate strings) runs no shell, so there is nothing to quote.

```
seq 1 3 | while read n; do echo "n=$n"; done # read from a
→command
printf '%s\n' banana apple cherry | sort     # write to a
→command
```

```
use v5.42;
open my $ph, '-|', 'seq', '1', '3' or die "pipe: $!"; # read
→command output
my @nums = <$ph>;
close $ph;
chomp @nums;
say "got: @nums";                  # got: 1 2 3
```

```

use v5.42;
open my $sort, '|-', 'sort' or die "pipe: $!"; # write to command_
↳stdin
print {$sort} "$_\n" for qw(banana apple cherry);
close $sort;
# apple
# banana
# cherry

```

Redirection. `>` and `>>` are the open modes `'>'` (truncate) and `'>>'` (append); a three-arg open with a lexical handle is the idiom. `print {$fh} ...` writes to a handle. To merge a child's stderr into the captured stream (`2>&1`), keep the redirection in the command you hand to backticks:

```

use v5.42;
open my $fh, '>', '/tmp/sh_out.txt' or die "write: $!";
print {$fh} "line $_\n" for 1 .. 3;
close $fh;

my $both = `sh -c 'echo to-out; echo to-err 1>&2' 2>&1`; # merge_
↳stderr
print $both;
# to-out
# to-err

```

Here-docs. The shell's `<<EOF` (interpolating) and `<<'EOF'` (literal) map one-to-one to Perl's `<<"END"` and `<<'END'`. Double quotes on the terminator interpolate variables; single quotes do not.

```

use v5.42;
my $name = "world";
print <<"END"; # interpolating, like <<EOF
hello $name
END
print <<'END'; # literal, like <<'EOF'
no $interpolation here
END
# hello world
# no $interpolation here

```

In-place edit. `sed -i` rewrites a file in place; the idiomatic Perl is the same switch on a `-p` one-liner. The runnable equivalent today is the explicit read-modify-write that `-i` performs under the hood - slurp, transform, write back:

```

use v5.42;
my $path = '/tmp/inplace_data.txt'; # holds: red green blue

open my $in, '<', $path or die "read: $!";
my @lines = <$in>; # slurp all lines
close $in;

```

(continues on next page)

(continued from previous page)

```
s/e/E/g for @lines;           # transform each line's $_

open my $out, '>', $path or die "write: $!";
print {$out} @lines;         # write them back
close $out;
# file now holds:
# rEd
# grEEn
# bluE
```

The gotcha: open returns false on failure rather than raising, so the `or die "...: $!"` idiom is mandatory - `$!` is the system error string, the shell's silent `2>/dev/null` failure made loud and explicit. And when you build a command from variables, prefer the **list form** of `system/open('ls', '-l', $dir)` over a single interpolated string: the list form bypasses the shell entirely, so there is no word splitting, no glob expansion, and no quoting to get wrong - the cure for the shell's quoting hell.

One-liners and switches

The one-liner is the shell developer's home turf and Perl's too: the filter you drop into a pipeline. The switch family that makes it work: `-e` supplies the program, `-n` wraps it in a no-auto-print loop, `-p` wraps it in an auto-print loop, `-i` edits files in place, `-l` handles line endings, and `-a/-F` autosplit each line into `@F` (the direct `awk` analogue, fields in `@F` instead of `$1..$NF`). `BEGIN { }` and `END { }` run before and after the loop, like `awk`'s `BEGIN/END`.

```
grep error log.txt           # filter lines
awk -F: '{print $1}' /etc/passwd # first field
awk '{s += $NF} END {print s}' data.txt # sum last field
```

```
perl -ne 'print if /error/' log.txt # filter lines
perl -F: -ane 'print "$F[0]\n"' /etc/passwd # first field, -F sets
→split
perl -lane '$s += $F[-1]; END { say $s }' data.txt # sum last field
```

These are shown without output because the implicit-loop switches are not recognised by this `ppperl` build yet (see the note at the top). Each maps to a verified explicit-loop form: `perl -ne 'CODE'` is `while (<$fh) { CODE }`, and `perl -pe 'CODE'` is the same with a trailing `print`. The runnable “first field of each line” form:

```
use v5.42;
open my $fh, '<', '/tmp/passwd.txt' or die "open: $!";
while (<$fh) {           # the implicit loop of -ane
    chomp;
    my @F = split /\:/;   # what -F: gives you in @F
    say $F[0];           # print the first field
}
close $fh;
# root
```

(continues on next page)

(continued from previous page)

```
# daemon
# bin
```

The gotcha: until the switches land on pperl, write the explicit loop (it runs today and is what the switches expand to anyway), and consult the *One-Liners guide* for the full switch catalogue and dozens of recipes - it is the highest-leverage thing to read coming from a shell background, where the one-liner is the everyday reflex.

Control flow

The shell's for, while, case, and if/test map onto Perl loops and conditionals, and several of the mappings are pleasantly direct.

Loops. for f in *; do becomes for my \$f (glob "*"); the file glob *.txt is glob "*.txt" or the readline form <*.txt>.

```
for f in /tmp/sh_*.txt; do
    echo "file: $f"
done
```

```
use v5.42;
for my $f (sort glob "/tmp/sh_*.txt") {
    say "file: $f";
}
# file: /tmp/sh_data.txt
# file: /tmp/sh_passwd.txt
```

Conditionals and test. This is a happy mapping: the shell's [-e], [-d], [-f] file tests are the **same -X operators** in Perl. String and numeric comparison split into two families, as they do between ["\$a" = "\$b"] and ["\$a" -eq "\$b"]: Perl uses eq/ne for strings and ==/!= for numbers. The shell's -z/-n (empty / non-empty string) become !length / length.

```
if [ -f "$f" ] && [ -n "$name" ]; then
    echo ok
fi
[ "$a" = "$b" ]           # string equality
[ "$x" -eq "$y" ]        # numeric equality
```

```
use v5.42;
my $f = '/tmp/sh_data.txt';
my $name = 'set';
say "ok" if -f $f && length $name;  # same -f, same idea
# ok

say "abc" eq "abc" ? "eq" : "ne";  # string compare
say 10 == 10.0 ? "num-eq" : "num-ne"; # numeric compare
# eq
# num-eq
```

case. The shell's case has no single keyword twin; the idiomatic Perl is a for/if chain or a ternary cascade (or, for many branches, a dispatch hash mapping keys to code refs).

```
case "$cmd" in
  start)    echo starting ;;
  stop)     echo stopping ;;
  *)        echo "unknown: $cmd" ;;
esac
```

```
use v5.42;
for my $cmd (qw(start stop bogus)) {
  my $r =
    $cmd eq 'start' ? "starting"
    : $cmd eq 'stop' ? "stopping"
    : "unknown: $cmd";
  say "$cmd -> $r";
}
# start -> starting
# stop -> stopping
# bogus -> unknown: bogus
```

Exit status. After a `system()`, the child's status is in `$?`, exactly the shell variable. But `$?` holds the raw wait status, not the bare exit code: shift right by 8 to get the code the shell would show as `$?`. The loop keywords `last` and `next` are the shell's `break` and `continue`.

```
use v5.42;
system('sh', '-c', 'exit 3');
say "raw $? = $?";           # raw $? = 768
say "exit code = ", $? >> 8; # exit code = 3
```

The gotcha: `$? >> 8` is the exit code; the low 8 bits hold the signal that killed the child (if any). The shell hides this behind its own `$?`; in Perl you do the shift yourself. Forgetting it is why `if ($? == 1)` after a command that exited 1 is almost always wrong - the raw value is 256, not 1.

Variables and data

A shell variable is always a string - `var=value`, used as `$var` or `${var}`, and quoted as `"$var"` to survive word splitting. Perl's scalar is `my $var`, lexically scoped, and **no quoting dance is needed**: a Perl scalar holds one value and does not get re-split on use. Where the shell forces everything through strings, Perl adds the two containers the shell never had: ordered `@arrays` and key/value `%hashes`.

```
name="Ada"
greeting="Hello, ${name}"
langs="Perl Python"           # a "list" is really a string + IFS
```

```
use v5.42;
my $name = "Ada";
my $greeting = "Hello, $name"; # interpolation, like "${name}"
```

(continues on next page)

(continued from previous page)

```
my @langs = ("Perl", "Python"); # a real list, not a split string
say $greeting;                  # Hello, Ada
say "first: $langs[0]";         # first: Perl
say "count: ", scalar @langs;   # count: 2
```

Script arguments. The shell's positional parameters map to @ARGV. The quoting minefield of "\$@" versus \$* simply does not exist: @ARGV is a real array, each argument a clean element no matter what spaces or globs it contained. \$1..\$9 become \$ARGV[0]..\$ARGV[8], \$# (the count) becomes scalar @ARGV, and \$0 (the script name) becomes \$0.

```
echo "count: $#"  
echo "first: $1"  
echo "all: $@"
```

```
use v5.42;  
say "count: ", scalar @ARGV; # like $#  
say "first: $ARGV[0]";      # like $1  
say "all: @ARGV";           # like "$@"  
# invoked as: pperl script.pl one two three  
# count: 3  
# first: one  
# all: one two three
```

Associative arrays. A Perl hash is the shell's declare -A done right - declared with %, accessed one value at a time with \${key}, and far more ergonomic than the shell's bolted-on associative arrays.

```
use v5.42;  
my %count;  
$count{$_}++ for qw(apple pear apple fig apple pear);  
for my $k (sort keys %count) {  
    say "$k: $count{$k}";  
}  
# apple: 3  
# fig: 1  
# pear: 2
```

The gotcha: a shell variable is global by default and a Perl my variable is lexically scoped to its block - the opposite default. The shell's local (dynamic scoping inside a function) is closest to Perl's local, but for ordinary variables you want my, which is true lexical scope. And interpolation only happens in double quotes: "\$name" interpolates, '\$name' is the literal four characters, just like the shell.

Perl also has full subroutines, references, modules, and OO if your script outgrows a one-liner - see the *Object-oriented Programming guide* and the other coming-from guides.

Gotchas coming from the shell

The traps, collected for a final scan before you write:

- **<\$fh> keeps the trailing newline.** Shell read trims; Perl does not. `chomp` is your reflex on every line read.
- **split is a separate step, and it is 0-based.** Fields are `$f[0]`, `$f[1]`, not awk's 1-based `$1`. `split ' '` (magic whitespace) trims and collapses; `split / /` (a literal space) does not.
- **s/// returns a count, not the new string.** `my $x = $s =~ s/a/b/` puts the substitution *count* in `$x`. Copy first (`my $t = $s`) `=~ s/.../.../`) or use the `/r` flag.
- **\$? >> 8 is the exit code.** The raw `$?` after `system` is the full wait status; the exit code is the high 8 bits. `if ( $? == 1 )` is almost always a bug; you want `if ( ( $? >> 8 ) == 1 )`.
- **open does not raise on failure.** Use `open my $fh, '<', $path` or `die "....: $! "`. `$!` is the system error string - the shell's silent failure made explicit.
- **Prefer the list form of system/open over an interpolated string.** `system('ls', '-l', $dir)` runs no shell: no word splitting, no glob surprise, no quoting hell. The string form (`system "ls -l $dir"`) reintroduces every shell-quoting trap you came here to escape.
- **my is lexical; shell variables are global.** Declare with `my` and scope to the block; reach for `local` only for the shell's dynamic-scoping behaviour, which is rarely what you want.
- **String vs numeric comparison is two families.** `eq/ne` for strings, `==/!=` for numbers - the `[ = ]` vs `[ -eq ]` split, but the operators are reversed from the shell, so read them carefully.
- **The implicit-loop switches (-n/-p/-a/-F/-i/-l) are not yet recognised by this pperl build.** Only `-e` works today. Write the explicit `while (<$fh) { ... }` loop, which is exactly what the switches expand to, until they land.

Tcl to Perl

You know Tcl. You write `set x 5`, you substitute with `$x`, you reach for `[expr { ... }]` to do arithmetic, you build lists with `lappend` and walk them with `foreach`, and you bend strings with `regexp` and `regsub`. This guide maps each of those habits onto Perl, so you can keep every reflex you have and pick up the things Tcl makes hard.

Tcl and Perl share the same roots: Unix scripting, a string-first worldview, and `$`-substitution as the way you reach a variable. Most of what you know maps almost directly - and Perl gives you real data structures (`@arrays` and `%hashes` as first-class types, not lists pressed into service) and a vast module ecosystem to land in. Two specific pieces of friction get the weight in this guide: Perl's distinct array/hash types where Tcl has lists and assoc arrays, and Perl's `regex` (a superset of Tcl's). The line-at-a-time and one-liner material is lighter here, because Tcl is not a one-liner language; you will spend your time in variables, data, control flow, and `regex`.

Every runnable example here was executed on `ppperl`, a Rust reimplementation of Perl 5.42; the `# ...` comments show its real output. Start scripts with:


```
use v5.42;
```

That one line turns on `strict`, warnings, and the `say` built-in (a `print` with a trailing newline, the rough twin of Tcl's `puts`), the modern feature set used throughout.

Note

One-liner switches on this pperl build Perl's one-liner switches -`perl -ne`, `perl -pe`, -`a`, -`i`, -`l` - are **correct perl5 5.42** and are the idiomatic form for ad-hoc text work. But the implicit-loop switch family (-`n`, -`p`, -`a`, -`F`, -`i`, -`l`) is **not yet recognised by this pperl build** - only -`e` is. So the one-liner examples below are shown *without* output, and each is paired with an explicit-loop full-script form (`while (<$fh>) { . . }`) that **does** run on pperl today and carries the verified output. Use the full-script form until the switches land.

How to read this

- *Variables and substitution* - `set` becomes `my`, `[expr]` disappears, `$x` substitution stays.
- *Lists and arrays* - `lindex/lappend/lsort` become Perl array operations.
- *Assoc arrays and hashes* - `set a(k) v` becomes `$a{k}`; `dict` becomes a hash.
- *Procs and references* - `proc` becomes `sub`; `upvar/uplevel` become references.
- *Regex and substitution* - `regexp/regsub/string map` become `m///s///tr///`.
- *Control flow* - `if/while/foreach/switch` become Perl's, plus `{}` vs `"` quoting.
- *The line/record loop* - reading a file line by line with `$_` and `$..`
- *Fields and splitting* - `split/join`, the Tcl `split` analogue.
- *I/O, pipes, and in-place edit* - `open/gets/puts/exec` become `open/<$fh>/print/qx`.
- *One-liners and switches* - -`n`, -`p`, -`e`, kept brief.
- *Gotchas coming from Tcl* - scan before you write.

Variables and substitution

In Tcl, `set x 5` creates a variable and `$x` substitutes its value. In Perl, `my $x = 5;` declares a lexically scoped variable and `$x` substitutes it - the substitution syntax you already know carries over verbatim. The relief is that Perl has native operators: there is no `[expr { . . }]`. Arithmetic is just arithmetic.

```
set x 5
set name "world"
set sum [expr {$x + 3}]
puts "x is $x"
puts "sum is $sum"
```



```

use v5.42;
my $x = 5;                # set x 5
my $name = "world";
my $sum = $x + 3;         # native operators, no [expr] needed
say "x is $x";           # x is 5
say "sum is $sum";       # sum is 8

```

Quoting maps cleanly, and this is a happy mapping: Tcl's `"..."` (substitution happens) is Perl's double-quoted string, and Tcl's `{...}` (literal, no substitution) is Perl's single-quoted string.

```

use v5.42;
my $name = "world";
say "hello, $name";      # hello, world    (interpolates)
say 'no $interp here';  # no $interp here (literal)

```

The gotcha: Tcl is “everything is a string” and Perl is close to it - a scalar auto-coerces between number and string by context - but Perl splits *comparison* into two operator families, and you pick by the operation, not the value. Numbers use `==`, `!=`, `<`, `>`; strings use `eq`, `ne`, `lt`, `gt`.

```

use v5.42;
my $x = "5";
say $x + 3;              # 8      (numeric context coerces "5")
say $x . "3";            # 53     (string context)
say "10" == "10.0" ? "num-equal" : "num-differ"; # num-equal
say "10" eq "10.0" ? "str-equal" : "str-differ"; # str-differ

```

`==` compares `"10"` and `"10.0"` as numbers (equal); `eq` compares them as strings (different). Using `==` where you meant `eq`, or the reverse, is the one quiet bug the string-first worldview sets you up for.

Lists and arrays

A Tcl list is a Perl array, and the verbs line up one for one. The big shift: in Tcl *everything is a list*, including strings you split on the fly; in Perl an array (`@a`) is a distinct, first-class type with its own sigil. When you reach for one element, the sigil follows the *element*, so it is `$a[0]` (a scalar), not `@a[0]`.

```

set items [list apple pear fig]
puts [lindex $items 0]
puts [llength $items]
lappend items plum
set sorted [lsort $items]
foreach it $items { puts "item: $it" }

```

```

use v5.42;
my @items = ("apple", "pear", "fig");
say $items[0];          # apple          (lindex $items 0)
say scalar @items;      # 3              (llength $items)

```

(continues on next page)

(continued from previous page)

```

push @items, "plum";           # lappend items plum
my @sorted = sort @items;      # lsort
say "@sorted";                 # apple fig pear plum
for my $it (@items) {          # foreach it $items
    say "item: $it";
}

```

`lsearch` - “find the index of a matching element” - has no single built-in; the idiomatic Perl is `grep` over the index range, with `$#items` as the last index:

```

use v5.42;
my @items = ("apple", "pear", "fig", "plum");
my ($idx) = grep { $items[$_] eq "fig" } 0 .. $#items; # lsearch
say "fig at $idx";                                     # fig at 2

```

The gotcha: scalar `@items` is how you get the length (Tcl’s `llength`); writing `my $n = @items` works because assigning an array to a scalar yields its count, but spell it scalar `@items` when you want the length inside a larger expression. And `sort` compares as *strings* by default, so `sort (10, 9, 100)` gives `10 100 9`; pass a numeric comparator `sort { $a <=> $b } @nums` for numeric order, where `$a` and `$b` are the two elements and `<=>` is the numeric three-way operator.

Assoc arrays and hashes

A Tcl associative array - set `a(red) 1` - is a Perl hash. The element access maps directly: Tcl’s `$a(red)` is Perl’s `$a{red}`, and the whole collection is `%a` (Tcl has no syntax for the array as a single value; Perl does). Modern Tcl’s `dict` maps to the same Perl hash, or to a hash reference when you need to nest or pass it around.

```

set a(red) 1
set a(green) 2
set a(blue) 3
puts $a(red)
puts [array exists a]
foreach k [lsort [array names a]] { puts "$k => $a($k)" }
unset a(blue)

```

```

use v5.42;
my %a = (red => 1, green => 2, blue => 3);
say $a{red};           # 1      ($a(red))
say exists $a{green} ? "yes" : "no"; # yes  (info exists
    ↪a(green))
for my $k (sort keys %a) {          # array names a
    say "$k => $a{$k}";
}
delete $a{blue};             # unset a(blue)

```

That prints:

```
1
yes
red => 1
green => 2
blue => 3
```

The gotcha: membership is `exists $a{key}`, not a separate command, and the fat comma `=>` is the idiomatic key/value separator in a hash literal. `=>` also auto-quotes the bareword on its left, so `red => 1` needs no quotes around `red`. A hash has no inherent order (neither does a Tcl array); iterate `sort keys %a` when you want a stable one, exactly as you reach for `lsort [array names a]`.

Procs and references

`proc name {args} {body}` becomes Perl's `sub name { ... }`. The classic Tcl way to read arguments - positional, by name - has a direct Perl twin: `my ($a, $b) = @_`; unpacks the argument array `@_`. Modern Perl also has signatures, which read like Tcl's `{args}` list and add defaults.

```
proc add {a b} {
    return [expr {$a + $b}]
}
puts [add 3 4]

proc greet {who {greeting Hello}} {
    return "$greeting, $who!"
}
puts [greet Tcl]
puts [greet Perl Hi]
```

```
use v5.42;
sub add {
    my ($a, $b) = @_;          # unpack the argument array
    return $a + $b;
}
say add(3, 4);                # 7

sub greet ($who, $greeting = "Hello") { # signature with default
    return "$greeting, $who!";
}
say greet("Tcl");              # Hello, Tcl!
say greet("Perl", "Hi");       # Hi, Perl!
```

Tcl's pass-by-name - `upvar`, where a proc reaches up and mutates a caller's variable - becomes a Perl **reference**: take one with `\`, follow it with `$$ref`. You pass the reference explicitly instead of naming the variable, which makes the data flow visible at the call site.

```
use v5.42;
sub increment {
```

(continues on next page)

(continued from previous page)

```

    my ($ref) = @_;
    $$ref++;                # mutate through the reference (Tcl:
    ↪upvar)
}
my $count = 10;
increment(\$count);         # pass a reference to $count
say $count;                # 11

```

The gotcha: `@_` is the argument array, and `my ($a, $b) = @_;` *copies* the arguments into named locals - the line you almost always want at the top of a sub. Reaching for upvar-style aliasing is rarely needed in Perl; when you do need to mutate a caller's data, pass a reference (`\$x`, `\@arr`, `\%h`) deliberately rather than aliasing by name. References are also how Perl nests data (an array of hashes, a hash of arrays); the same mechanism that replaces upvar is the one that replaces Tcl's string-encoded nested lists.

Regex and substitution

This is a payload section. Tcl's `regexp` and `regsub` use Advanced Regular Expressions; Perl's regex is a **superset**, so the patterns you know carry over and gain features. Perl makes regex *syntax*, not a command: `=~` binds a pattern to a string, `m//` matches, `s///` substitutes, and `tr///` transliterates.

```

set s "order 42 shipped"
if {[regexp {(\\d+)} $s -> num]} {
    puts "number: $num"
}
set t [regsub -all {o} "foo boo zoo" 0]
puts $t
set u [string map {a A b B} "abcabc"]
puts $u

```

```

use v5.42;
my $s = "order 42 shipped";
if ($s =~ /(\\d+)/) {          # regexp; capture lands in $1
    say "number: $1";          # number: 42
}
(my $t = "foo boo zoo") =~ s/o/0/g;    # regsub -all -> s///g
say $t;                             # f00 b00 z00
(my $u = "abcabc") =~ tr/ab/AB/;      # string map (char-for-char)
say $u;                             # ABcABc

```

The capture maps too: Tcl's `-> num` match variable becomes Perl's `$1`, `$2`, and so on, with named captures landing in `%+`:

```

use v5.42;
if ("2026-06-21" =~ /(\\<y>\\d{4})-(\\<m>\\d{2})/) {
    say "$+{y} / $+{m}";        # 2026 / 06
}

```

Tcl's `string match` is glob matching, not `regex(string match *.txt $name)`. In Perl,

anchor a regex instead - `/\.txt$/` is the equivalent - because Perl's one pattern language covers both jobs.

```
use v5.42;
my $name = "report.txt";
say "matches" if $name =~ /\.txt$/;      # string match *.txt $name
# matches
```

The gotcha: `s///` modifies its target *in place* and returns the *count* of substitutions, not the new string - which is why the examples above copy first with `(my $t = "...")` before substituting. To get a new string and leave the original untouched, use the `/r` flag: `my $new = $s =~ s/o/O/gr`. And `tr///` (the string map analogue for single characters) is **not** a regex: its left side is a literal character set, so `tr/ab/AB/` maps a and b only. For pattern work, reach for `s///`.

Control flow

The control structures map almost one to one; the spelling shifts and the braces stay. Tcl's `if {cond} {body}` becomes Perl's `if (cond) { body }`, `elseif` becomes `elsif`, and `foreach` becomes `for` (the two spellings are aliases in Perl; `foreach` works too).

```
if {$n < 0} {
    set sign neg
} elseif {$n == 0} {
    set sign zero
} else {
    set sign pos
}

set i 0
while {$i < 3} { puts "i=$i"; incr i }

foreach w {a b c} { puts "w=$w" }
```

```
use v5.42;
my $n = 0;
my $sign;
if ($n < 0) { $sign = "neg" }
elsif ($n == 0) { $sign = "zero" } # elseif -> elsif
else { $sign = "pos" }
say $sign;                        # zero

my $i = 0;
while ($i < 3) { say "i=$i"; $i++ } # incr i -> $i++

for my $w (qw(a b c)) { say "w=$w" } # foreach w {a b c}
```

Tcl's `switch` has no single keyword in Perl; the idiomatic replacement is a **dispatch hash** mapping each case to a code reference, which is faster and more flexible than a chain of `if`:

```

use v5.42;
my $cmd = "start";
my %dispatch = (
    start => sub { "starting" },
    stop  => sub { "stopping" },
);
say $dispatch{$cmd}->();      # starting

```

The gotcha: the {} vs "" distinction you rely on in Tcl for deferring versus substituting maps to Perl's ' ' vs "" for strings (covered above), but Perl's *block* braces are not quoting - a Perl { ... } after if/while/for is a code block, always evaluated, never a deferred string. There is no “the body is a string until I run it” step; Perl parses the block at compile time. Postfix modifiers are a bonus Tcl has no form for: say \$_ for @items; and say "neg" if \$n < 0; carry the loop or condition after the statement.

The line/record loop

Tcl reads a file with set fh [open f r]; while {[gets \$fh line] >= 0} {...}. Perl's equivalent is a while (<\$fh>) loop, with each line landing in \$_ (the implicit topic variable) or in a named scalar. The line number is in \$..

```

set fh [open /tmp/tcl_data.txt r]
while {[gets $fh line] >= 0} {
    puts "got: $line"
}
close $fh

```

```

use v5.42;
open my $fh, '<', '/tmp/tcl_data.txt' or die "open: $!";
while (my $line = <$fh>) {      # gets $fh line
    chomp $line;                # <$fh> keeps the newline; strip it
    say "got: $line";           # puts
}
close $fh;
# got: apple
# got: banana
# got: cherry

```

(The input file held apple, banana, cherry.) Using \$_ and the line counter \$. reads even closer to a record loop:

```

use v5.42;
open my $fh, '<', '/tmp/tcl_data.txt' or die "open: $!";
while (<$fh>) {                # each line lands in $_
    chomp;                     # chomp defaults to $_
    say "$. : $_";             # $. is the current line number
}
close $fh;
# 1 : apple
# 2 : banana

```

(continues on next page)

(continued from previous page)

3 : cherry

The gotcha: Tcl's gets strips the trailing newline for you; Perl's <\$fh> keeps it, so call chomp when you do not want it. With no argument, chomp and say/print both default to \$_, which is why the second loop never names the line variable.

Fields and splitting

Tcl's split \$line \$sep returns a list; Perl's split /sep/, \$line returns an array, and join puts it back together. The difference: a Perl split separator is a *regex*, so you get the full pattern language for free.

```
set line "a:b:c"
set parts [split $line ":"]
puts [llength $parts]
puts [lindex $parts 1]
puts [join $parts ","]
```

```
use v5.42;
my $line = "a:b:c";
my @parts = split /:/, $line; # split $line ":" (separator is a
    ↪ regex)
say scalar @parts;           # 3      (llength $parts)
say $parts[1];               # b      (lindex $parts 1)
say join ", ", @parts;       # a,b,c  (join $parts ",")
```

The gotcha: split ' ' with a literal single-space string is a special case that splits on *any run of whitespace* and discards leading empty fields - the awk-style “split into words” behaviour, and usually what you want for free-form input. split / / (a regex with one space) is the literal “split on each single space” and does not collapse runs. Reach for split ' ' to tokenise words:

```
use v5.42;
my @words = split ' ', " the quick brown ";
say "@words"; # the quick brown
```

I/O, pipes, and in-place edit

Tcl's open/gets/puts/read/close map to Perl's open/<\$fh>/ print/slurp/close. The idiom is the three-argument open with a lexical filehandle. Tcl's exec - run a command and capture its output - becomes Perl's qx{...} (backticks) or system when you do not need the output.

```
set out [exec echo hello from shell]
puts "exec said: $out"
```

```
use v5.42;
my $out = qx{echo hello from shell}; # exec -> qx// (backticks)
chomp $out;
```

(continues on next page)

(continued from previous page)

```
say "exec said: $out";           # exec said: hello from_
↪shell
```

Tcl has no in-place file edit built in; you read, transform, and write back. Perl's `-i` switch is the one-liner form of exactly that, and the runnable full-script equivalent is a read-modify-write you can write today:

```
use v5.42;
my $path = '/tmp/tcl_inplace.txt';      # holds: red green blue

open my $in, '<', $path or die "read: $!";
my @lines = <$in>;                      # slurp all lines (list_
↪context)
close $in;

s/e/E/g for @lines;                    # transform each line's $_

open my $out, '>', $path or die "write: $!";
print {$out} @lines;                  # write them back
close $out;
# file now holds:
# rEd
# grEEEn
# bluE
```

The gotcha: `open` returns false on failure rather than raising (Tcl throws an error you catch with `catch`), so the `or die "$!"` idiom is mandatory - `$!` is the system error string. For pipes, Perl opens a pipe filehandle directly: `open my $fh, '|-', 'ls', '-l'` reads a command's output and `open my $fh, '|-', 'sort'` writes to a command's input, the structured form of Tcl's `open "|command"`.

One-liners and switches

Tcl is not a one-liner language, so this section is brief - but Perl is, and once you leave Tcl behind it is worth knowing the reflex exists. The switch family: `-e` supplies the program, `-n` wraps it in a no-auto-print loop, `-p` wraps it in an auto-print loop, `-i` edits in place, and `-a/-F` autosplit each line into `@F`. `BEGIN { }` and `END { }` blocks run before and after the loop.

```
perl -ne 'print if /error/' log.txt      # print matching lines
perl -pe 's/\d+/N/g' input.txt          # mask numbers
perl -nE 'END { say $n } $n++ if /error/' log.txt # count, END_
↪block
```

These are shown without output because the implicit-loop switches are not recognised by this pperl build yet (see the note at the top). Each maps to a verified explicit-loop form: `perl -ne 'CODE'` is `while (<$fh>) { CODE }`, and `perl -pe 'CODE'` is the same with a trailing `print`. The runnable “count matching lines” form:


```

use v5.42;
open my $fh, '<', '/tmp/tcl_data.txt' or die "open: $!";
my $n = 0;
while (<$fh>) {
    $n++ if /banana/;           # the body of -ne
}
say $n;                        # 1

```

The gotcha: until the switches land on pperl, write the explicit loop (it runs today and is what the switches expand to anyway), and consult the *One-Liners guide* for the full switch catalogue and recipes. Perl also has full subroutines, references, modules, and OO if your script outgrows a glue task: Tcl's namespace `eval` and `[incr Tcl]` classes map to Perl packages and the modern class syntax; see the *Object-oriented Programming guide* for that side of the language.

Gotchas coming from Tcl

The traps, collected for a final scan before you write:

- **Two comparison families.** `==`/`<`/`<=>` compare numerically; `eq`/`lt`/`cmp` compare as strings. `"10" == "10.0"` is true, `"10" eq "10.0"` is false. Pick by the operation, not the value - the one quiet bug the string-first worldview invites.
- **The sigil follows the element.** One element of `@items` is `$items[0]` (a scalar), not `@items[0]`; one value of `%a` is `$a{red}`. The container sigil and the element sigil differ.
- **sort is string-sorted by default**, like comparing with `eq`. Pass `{ $a <=> $b }` for numeric order.
- **s/// returns a count, not the new string**, and edits in place. Copy first (`(my $t = $s) =~ s/.../.../`) or use the `/r` flag for a fresh string.
- **tr/// (the string map analogue) is a literal character set, not a regex.** `tr/ab/AB/` maps single characters. For patterns, use `s///`.
- **split ' ' and split / / differ.** The literal-space string splits on whitespace runs and drops leading empties (the word tokeniser); the one-space regex does not. Use `split ' '` for free-form input.
- **<\$fh> keeps the trailing newline**, unlike Tcl's `gets`. Call `chomp` to strip it.
- **open does not raise on failure.** Use `open my $fh, '<', $path or die "...: $!".` `$!` is the system error string; there is no catch to wrap it.
- **References replace upvar.** To mutate a caller's variable, pass a reference (`\$x`, `\@arr`, `\%h`) deliberately rather than aliasing by name. References are also how Perl nests data.
- **A { ... } block is code, not a deferred string.** Perl parses blocks at compile time; there is no "the body is a string until I run it" step. Only `'/'`/`"` carry Tcl's `{}/"` substitution distinction.
- **The implicit-loop switches (-n/-p/-a/-F/-i/-l) are not yet recognised by this pperl build.** Only `-e` works today. Write the explicit `while (<$fh>) { ... }` loop, which is exactly what the switches expand to, until they land.

4.10 The Definitive Guide to Graphics with Perl

Perl is not the language people reach for when they think “graphics”, and that reputation is mostly an accident of history rather than a statement about the tools. The raster workhorse (Imager) is actively maintained and fast, the vector engine everyone else uses under the hood (Cairo) has clean bindings, and there is a self-contained GUI toolkit with its own graphics core (Prima) that draws to a real window with no external toolkit to fight. What Perl has lacked is not libraries but a single place that puts them in order, from the pixel up to a running program.

This guide is that place, for **two dimensions**. It starts at the theory every graphics programmer eventually needs (how a line becomes pixels, what a color space actually is, how transparency composites), moves through the working CPAN toolkits, and then goes one step further than the usual tour: it uses pperl’s native FFI to reach C graphics libraries directly, the same way pperl already reaches libgmp to give you `Math::GMP`. It ends with two things you can run: a drawing application and a 2D game.

This is the first version. Three dimensions, OpenGL, game engines, and the GPU are a planned sequel, not part of this book. The final chapter draws that line explicitly.

4.10.1 Which chapter do I want?

I want to ...	Start here
understand how pixels, paths, and color actually work	<i>foundations</i>
implement a line, circle, fill, or blur myself	<i>raster-algorithms</i>
draw on, load, or save a bitmap image	<i>raster-imager</i>
use a GD or ImageMagick codebase, or their specific features	<i>raster-gd-magick</i>
produce resolution-independent vector paths, or SVG output	<i>vector-cairo-svg</i>
read image metadata, or draw a chart	<i>metadata-and-charts</i>
draw in a live window on Linux	<i>gui-toolkits</i>
reach a C graphics library that has no CPAN binding	<i>ffi-graphics-primer</i>
see how pperl’s shipped native Cairo binding works	<i>ffi-cairo-showcase</i>
build a real drawing application	<i>capstone-drawing-app</i>
build a real 2D game	<i>capstone-2d-game</i>
find 3D, game engines, or a complete SDL2 binding	<i>whats-next</i>

4.10.2 The three arcs of this guide

- **Foundations** - *foundations* and *raster-algorithms*. The mental model and the classic algorithms, written in plain Perl so you own them before any library hides them. Read these once; every later chapter assumes the vocabulary.
- **Toolkits** - the five chapters from *raster-imager* through *gui-toolkits*. The working CPAN 2D stack: what each library is good at and where one wins over another. Each chapter is self-contained; reach for the one that names your problem.
- **Reaching further** - *ffi-graphics-primer* onward. pperl’s native FFI, the two capstone projects, and an honest map of where the road currently ends.

4.10.3 A note on the runtime

These chapters target `ppperl`, a faithful Rust reimplementaion of Perl 5.42. The CPAN modules in the toolkit chapters are used exactly as they are on any Perl 5, and the FFI chapters use `ppperl`'s own `Peta::FFI`, which has no `perl5` equivalent.

4.10.4 Known gaps

Everything three-dimensional, every GPU-programming topic, computer vision and image analysis, and connectors to hosted image-generation services are out of scope for this version. They are named, with the reason and the plan, in *what's next*. The FFI chapters also flag, at the point where it matters, which C interfaces are reachable today and which wait on the next layer of `Peta::FFI`.

4.10.5 See also

- *Auto-FFI (Peta::FFI)* - the native FFI layer the “reaching further” arc is built on: `dlopen`, `call`, the type-signature codes, and the curated-binding pattern.
- *PPerl Architecture guide* - the JIT and how `ppperl` provides C-backed modules natively.

Foundations: pixels, paths, and color

Before any library, four ideas carry the whole subject: the difference between a raster and a vector, the coordinate system a drawing lives in, what a color actually is once you have to store it, and how two colors combine when one is partly transparent. Every chapter after this one leans on these without re-explaining them. None of it is Perl-specific, but all of the examples are Perl, because the point is to make the ideas concrete in code you can run.

Raster versus vector

A **raster** image is a grid of colored cells. It has a fixed width and height in pixels, and every pixel holds a color. Scaling it up past its resolution invents nothing: the cells just get bigger. Photographs, screenshots, and anything with continuous tone are rasters. `Imager`, `GD`, and `ImageMagick` are raster tools.

A **vector** image is a set of shapes described mathematically: “a line from here to there”, “a circle of this radius”, “fill this path”. There is no resolution until you draw it, so it scales to any size cleanly. Fonts, logos, and diagrams are vectors. `Cairo` and `SVG` are vector tools.

The two meet at **rasterization**: turning a vector description into pixels. A vector library still has to decide which pixels a diagonal line touches, and that decision is *the first algorithm in the next chapter*. Everything you draw ends up as pixels on a screen; the question is only whether you commit to a resolution early (raster) or late (vector).

A raster in Perl is, at its plainest, an array of rows:

```
my $w = 4;
my $h = 3;
# each pixel is [r, g, b], each channel 0..255
```

(continues on next page)

(continued from previous page)

```
my @img = map { [ map { [0, 0, 0] } 1 .. $w ] } 1 .. $h;

$img[1][2] = [255, 0, 0]; # row 1, column 2, red
```

That nested-array picture is all a raster is. Real libraries store the same grid in a packed C buffer for speed, but the mental model does not change: a pixel is addressed by an integer row and column, and it holds a color.

The coordinate system

Graphics coordinates are not math-class coordinates. The origin (0, 0) is the **top-left** corner, x increases to the right, and y increases **downward**. This catches everyone once: a larger y is lower on the screen, not higher.

```
# (0,0) top-left          x ->
# +-----+
# | (0,0)      (w-1, 0)
# y |
# | |
# v | (0, h-1)  (w-1, h-1)
```

A pixel is a little square, and there is a lasting ambiguity about whether the coordinate (3, 5) names the square's corner or its center. Raster libraries usually treat integer coordinates as pixel centers; vector libraries treat them as the grid lines between pixels, which is why a one-unit-wide vertical line at $x = 3$ can come out straddling two columns of pixels at half intensity. When a thin line or a rectangle edge looks blurry or one pixel off, this is almost always why, and the fix is a half-pixel offset. The chapters that hit it say so.

Sizes and positions are integers in raster space and real numbers in vector space. That is the practical difference you feel: in Imager you set pixel (120, 47); in Cairo you can move to (120.5, 47.25) and the rasterizer works out the coverage.

Color: what a pixel holds

A color on a screen is three numbers: how much **red**, **green**, and **blue** light to emit. Add all three at full strength and you get white; leave them all off and you get black. This is the **RGB** model, and it is additive because you are adding light, not mixing paint.

Each channel is almost always stored in 8 bits, so 0 .. 255 per channel, and a color is three bytes. That is the #RRGGBB hex notation from the web: #ff0000 is (255, 0, 0), pure red.

```
sub rgb_to_hex {
    my ($r, $g, $b) = @_;
    return sprintf "%02x%02x%02x", $r, $g, $b;
}

print rgb_to_hex(255, 128, 0), "\n"; # #ff8000, an orange
```

RGB is how pixels are stored, but it is a terrible model for a human choosing a color. “Make this a bit more orange but keep it the same brightness” is not an obvious move in RGB. **HSV** (hue, saturation, value) is the model for that: **hue** is the position on the color wheel in degrees 0 .. 360 (0 red, 120 green, 240 blue), **saturation** is how vivid versus gray 0 .. 1, and **value** is how bright 0 .. 1. Rotating the hue while holding saturation and value walks the rainbow at constant vividness and brightness.

Every color-picker and gradient tool works in HSV or a close relative, then converts to RGB to store. The conversion is worth having in plain sight, because you will reimplement it more than once:

```
# h in 0..360, s and v in 0..1 -> (r, g, b) in 0..255
sub hsv_to_rgb {
    my ($h, $s, $v) = @_;
    my $c = $v * $s;                               # chroma
    my $hp = $h / 60;
    my $x = $c * (1 - abs($hp % 2 - 1 + ($hp - int $hp)));
    my ($r, $g, $b) =
        $hp < 1 ? ($c, $x, 0) :
        $hp < 2 ? ($x, $c, 0) :
        $hp < 3 ? (0, $c, $x) :
        $hp < 4 ? (0, $x, $c) :
        $hp < 5 ? ($x, 0, $c) :
        ($c, 0, $x);
    my $m = $v - $c;
    return map { int(($_ + $m) * 255 + 0.5) } ($r, $g, $b);
}

my @rainbow = map { [ hsv_to_rgb($_, 1, 1) ] } 0, 60, 120, 180, 240,
→300;
# red, yellow, green, cyan, blue, magenta
```

There are other spaces you will meet: grayscale (one channel, a brightness), CMYK (subtractive, for print), and perceptually uniform spaces like CIELAB where equal numeric steps look like equal perceptual steps. For screen drawing in two dimensions, RGB for storage and HSV for choosing cover almost everything.

Alpha and compositing

A fourth channel, **alpha**, records opacity: 0 fully transparent, 255 (or 1.0) fully opaque. A pixel with alpha is **RGBA**. Alpha is what lets you draw one image over another and have the top one’s transparent parts show through.

Combining a foreground pixel over a background pixel is **compositing**, and the standard rule is **source-over**. With alpha as a fraction 0 .. 1:

```
# composite one RGBA pixel over another (all channels 0..255,
# alpha treated as 0..1 for the blend)
sub over {
    my ($fg, $bg) = @_;
    my $a = $fg->[3] / 255;                # source alpha
    my @out = map {
```

(continues on next page)

(continued from previous page)

```
        int($fg->[_] * $a + $bg->[_] * (1 - $a) + 0.5)
    } 0 .. 2;
    return \@out;
}

my $red_half = [255, 0, 0, 128];    # red at 50% opacity
my $white    = [255, 255, 255, 255];
my $result   = over($red_half, $white);
print "@$result\n";                # 255 128 128, a pink
```

Red at half opacity over white gives pink, exactly as you would expect paint-thinned-with-water to look. That one formula, applied per pixel, is how every layer, brush stroke, and sprite with soft edges lands on the canvas beneath it. When you stack many semi-transparent layers, you apply it repeatedly, bottom to top.

The subtlety that bites later is **premultiplied alpha**: storing each color channel already multiplied by its alpha. It makes repeated compositing cheaper and filtering correct at edges, and it is why a library's raw pixel bytes sometimes are not the RGB you expect. Cairo uses premultiplied alpha internally; the *Cairo chapter* notes where it shows.

The one 3D idea, named and deferred

You will hear about the **Z-buffer** the moment 3D comes up, and it is worth one paragraph here so the term is not a mystery. In 2D, “what is on top” is decided by draw order: the last thing drawn wins, which is exactly the source-over compositing above. In 3D, objects have a depth (a z coordinate), and the thing nearest the viewer should win regardless of draw order. A **Z-buffer** is a second grid, the same size as the image, holding the depth of the closest surface seen so far at each pixel; before drawing a pixel you check whether the new surface is nearer than what is recorded, and skip it if not.

That is the whole concept, and it is as far as this 2D guide goes with it. Depth buffering, projection, and the rest of the third dimension belong to the *planned sequel*. In two dimensions, draw order is your depth buffer.

Where this goes next

You now have the vocabulary the rest of the guide speaks: raster and vector, top-left coordinates, RGB and HSV, alpha and source-over compositing. The *next chapter* spends that vocabulary on the classic drawing algorithms, implemented from scratch, so that when a library draws a line or fills a polygon you know exactly what it is doing on your behalf.

Drawing primitives from scratch

Every graphics library hands you line, circle, fill, and blur. This chapter implements them in plain Perl so that when a library draws them for you, nothing is magic. These are the classic algorithms, and they are classic because they are the right answers: fast, integer where they can be, and correct at the awkward cases. You will not ship these implementations (a C library will always be faster), but knowing them tells you why a library behaves the way it does, and gives you the tools for the one case the library did not anticipate.

Throughout, the canvas is the nested-array raster from the *foundations chapter*: a grid of [r, g, b] pixels addressed by integer (x, y), origin top-left.

```
sub new_canvas {
    my ($w, $h) = @_;
    return { w => $w, h => $h,
             px => [ map { [ map { [255, 255, 255] } 1 .. $w ] } 1 ..
             ↪. $h ] };
}

sub set_px {
    my ($c, $x, $y, $color) = @_;
    return if $x < 0 || $y < 0 || $x >= $c->{w} || $y >= $c->{h};
    $c->{px}[$y][$x] = $color;
}
```

Lines: Bresenham

A line from (x0, y0) to (x1, y1) has to choose, for each column it crosses, which row of pixels to light. The naive approach computes $y = m \cdot x + b$ with floating point per step; **Bresenham's algorithm** does it with integer addition only, by tracking an error term that says how far the ideal line has drifted from the pixels chosen so far.

The insight is that at each step you either stay on the same cross-axis row or move one, and a running integer error decides which. The version below handles every slope and direction in one loop by working with absolute deltas and step signs:

```
sub draw_line {
    my ($c, $x0, $y0, $x1, $y1, $color) = @_;
    my $dx = abs($x1 - $x0);
    my $dy = -abs($y1 - $y0);
    my $sx = $x0 < $x1 ? 1 : -1;
    my $sy = $y0 < $y1 ? 1 : -1;
    my $err = $dx + $dy;                                # error accumulator
    while (1) {
        set_px($c, $x0, $y0, $color);
        last if $x0 == $x1 && $y0 == $y1;
        my $e2 = 2 * $err;
        if ($e2 >= $dy) { $err += $dy; $x0 += $sx; }    # step in x
        if ($e2 <= $dx) { $err += $dx; $y0 += $sy; }    # step in y
    }
}
```

No multiplication, no division, no floating point in the loop. That is why Bresenham survived from 1962: it is the line-drawing you would build in hardware. Every library's thin, aliased line is this or a direct descendant.

Circles: the midpoint method

A circle has eight-fold symmetry: compute one octant and mirror it into the other seven. The **midpoint circle algorithm** walks one octant with the same integer-error idea as Bresenham, deciding at each step whether to move straight or diagonally by testing a decision variable against zero.

```
sub draw_circle {
    my ($c, $cx, $cy, $r, $color) = @_;
    my ($x, $y) = ($r, 0);
    my $err = 1 - $r;                                # decision variable
    while ($x >= $y) {
        # eight mirrored points, one per octant
        for my $p ([ $x, $y ], [ $y, $x ], [ -$x, $y ], [ -$y, $x ],
                    [ $x, -$y ], [ $y, -$x ], [ -$x, -$y ], [ -$y, -$x ]) {
            set_px($c, $cx + $p->[0], $cy + $p->[1], $color);
        }
        $y++;
        if ($err < 0) { $err += 2 * $y + 1; }
        else          { $x--; $err += 2 * ($y - $x) + 1; }
    }
}
```

Compute one eighth, draw eight. The same symmetry trick, with the error test adjusted, gives ellipses and arcs.

Filling polygons: the scanline method

Drawing a shape's outline is one problem; filling its interior is another. The **scanline fill** sweeps a horizontal line down the image, and for each scanline finds where it crosses the polygon's edges. Those crossings come in pairs, and the interior is between each pair: the **even-odd rule**. Sort the x crossings, then fill between the first and second, the third and fourth, and so on.

```
# poly is an arrayref of [x, y] vertices, in order, implicitly
→closed
sub fill_polygon {
    my ($c, $poly, $color) = @_;
    my ($ymin, $ymax) = ($poly->[0][1], $poly->[0][1]);
    for my $v (@$poly) {
        $ymin = $v->[1] if $v->[1] < $ymin;
        $ymax = $v->[1] if $v->[1] > $ymax;
    }
    for my $y ($ymin .. $ymax) {
        my @xs;
        for my $i (0 .. $#poly) {
            my $a = $poly->[$i];
            my $b = $poly->[( $i + 1 ) % @$poly];
            my ($ax, $bx) = ($a->[0], $b->[0]);
            # does this edge straddle the scanline?
            next if ($ax <= $y) == ($bx <= $y);
```

(continues on next page)

(continued from previous page)

```

        # x where the edge crosses y (linear interpolation)
        my $t = ($y - $ay) / ($by - $ay);
        push @xs, $a->[0] + $t * ($b->[0] - $a->[0]);
    }
    @xs = sort { $a <=> $b } @xs;
    # fill between crossing pairs
    for (my $i = 0; $i + 1 < @xs; $i += 2) {
        for my $x (int($xs[$i] + 0.5) .. int($xs[$i + 1] - 0.5)) {
            set_px($c, $x, $y, $color);
        }
    }
}

```

The `($ay <= $y) == ($by <= $y)` test is the crossing check written to count each vertex exactly once, so a scanline passing through a shared vertex does not double-count and leak the fill. That half-open convention is the detail that separates a correct fill from one with occasional stray lines, and it is why libraries and this code agree on where an edge “belongs”.

Anti-aliasing: coverage instead of on-or-off

The line and circle above are **aliased**: each pixel is fully on or fully off, and a diagonal edge looks like a staircase. **Anti-aliasing** softens that by setting a pixel’s intensity to how much of it the shape actually covers. A pixel the edge crosses at 40 percent coverage gets 40 percent of the shape color blended with 60 percent of the background, which is exactly the *source-over compositing* from the last chapter with alpha equal to coverage.

The cheapest way to approximate coverage is **supersampling**: draw at several times the resolution with the aliased algorithm, then average each block of high-resolution pixels down to one. Averaging a 2x2 block where three sub-pixels are on gives 75 percent coverage for free:

```

# downsample a 2x-oversampled canvas by averaging 2x2 blocks
sub downsample_2x {
    my ($big) = @_;
    my ($w, $h) = ($big->{w} / 2, $big->{h} / 2);
    my $out = new_canvas($w, $h);
    for my $y (0 .. $h - 1) {
        for my $x (0 .. $w - 1) {
            my @sum = (0, 0, 0);
            for my $dy (0, 1) {
                for my $dx (0, 1) {
                    my $p = $big->{px}[2 * $y + $dy][2 * $x + $dx];
                    $sum[$_] += $p->[$_] for 0 .. 2;
                }
            }
            $out->{px}[$y][$x] = [ map { int($_ / 4 + 0.5) } @sum ];
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

    }
    return $out;
}

```

Real libraries compute coverage analytically rather than by brute-force supersampling, but the effect is identical: the staircase becomes a smooth ramp of intensities. When Cairo gives you a crisp anti-aliased curve and Imager offers an `aa` flag on its primitives, this is what they are doing.

Tessellation: breaking shapes into triangles

Hardware and many algorithms only really know how to fill **triangles**; everything more complex is first broken into triangles, a process called **tessellation** or **triangulation**. A convex polygon triangulates trivially by fanning from one vertex:

```

# fan triangulation of a convex polygon -> list of [v0, v1, v2]
→ triangles
sub triangulate_convex {
    my ($poly) = @_;
    my @tris;
    for my $i (1 .. $#poly - 1) {
        push @tris, [ $poly->[0], $poly->[$i], $poly->[$i + 1] ];
    }
    return @tris;
}

```

Concave polygons need a real algorithm (ear clipping is the standard one: repeatedly find a triangle that pokes out with no other vertex inside it, snip it off, repeat). The convex fan is enough to see the idea, and it is the idea that carries straight into 3D, where every surface is ultimately a mesh of triangles. In two dimensions you meet tessellation whenever you fill a complex vector path; in the *planned 3D sequel* it is the foundation of everything.

Convolution: blur, sharpen, and edges

A whole family of image effects comes from one operation: **convolution**, sliding a small grid of weights (a **kernel**) over the image and replacing each pixel with the weighted sum of its neighbors. A kernel of all equal weights averages, which blurs. A kernel that subtracts the neighbors and adds a large center sharpens.

```

# apply a 3x3 kernel to a canvas; divisor normalises the weights
sub convolve_3x3 {
    my ($c, $kernel, $divisor) = @_;
    $divisor ||= 1;
    my $out = new_canvas($c->{w}, $c->{h});
    for my $y (1 .. $c->{h} - 2) {
        for my $x (1 .. $c->{w} - 2) {
            my @sum = (0, 0, 0);
            for my $ky (0 .. 2) {

```

(continues on next page)

(continued from previous page)

```

        for my $kx (0 .. 2) {
            my $wgt = $kernel->[$ky][$kx];
            my $p = $c->{px}[$y + $ky - 1][$x + $kx - 1];
            $sum[$_] += $p->[$_] * $wgt for 0 .. 2;
        }
    }
    $out->{px}[$y][$x] =
        [ map { my $v = int($_ / $divisor + 0.5);
                $v < 0 ? 0 : $v > 255 ? 255 : $v } @sum ];
    }
}
return $out;
}

my $box_blur = [[1,1,1],[1,1,1],[1,1,1]]; # divisor 9
my $sharpen  = [[0,-1,0],[-1,5,-1],[0,-1,0]]; # divisor 1

```

The box blur divides by 9 to keep brightness constant; the sharpen kernel sums to 1 so it needs no divisor. Swap in a kernel that responds to intensity change and you get edge detection, which is where image processing shades into computer vision. That boundary is deliberate: this guide uses convolution as a **drawing** tool (blur a shadow, sharpen a downscaled sprite) and stops there. Analysis, feature detection, and recognition are a different subject, out of scope here.

What the libraries do for you

Everything above is what a graphics library implements in optimized C, usually with SIMD and careful edge handling you would not want to write by hand. Knowing the algorithms means you can read a library’s documentation and know what “anti-aliased”, “even-odd fill rule”, “convolution matrix”, and “triangulated path” mean, and you can drop to your own pixel loop for the rare effect no library ships. The *next chapter* picks up Imager, where these primitives are one method call away and running at C speed.

Raster drawing with Imager

Imager is the raster workhorse: create an image, draw on it, filter it, save it. It is actively maintained, it links its own copies of the format libraries so PNG, JPEG, TIFF, GIF, and WEBP all work out of the box, and its drawing primitives are the ones from the *algorithms chapter* running at C speed. If you are starting a raster project on Perl today and have no reason to prefer something else, start here.

Every drawing method returns false on failure and sets `errstr`, so the idiom is to check and report:

```

use Imager;

my $img = Imager->new(xsize => 400, ysize => 300, channels => 4)
    or die Imager->errstr;

```

`channels => 4` asks for RGBA (red, green, blue, alpha); use 3 for opaque RGB. The alpha channel is what lets the *compositing* from the foundations chapter work when you draw

one image onto another.

Colors

Anywhere a method takes color =>, you can pass a name, a hex string, an Imager::Color object, or a plain array reference:

```
my $red    = Imager::Color->new('red');
my $red2   = Imager::Color->new('#FF0000');
my $red3   = Imager::Color->new(255, 0, 0);
my $trans  = Imager::Color->new(255, 0, 0, 128); # 50% alpha

$img->box(color => [0, 128, 255], filled => 1); # array-ref, no
↳object
```

The array-reference form is the least ceremony for a one-off color and is what most of the examples below use.

The primitives

Note the parameter names carefully, because they are not all the same shape. Boxes use corner coordinates xmin/ymin/xmax/ymax; circles and arcs use a center x/y with a radius r; arcs add start and end degrees d1/d2.

```
# fill the whole canvas white first
$img->box(color => 'white', filled => 1);

# a line; aa => 1 requests anti-aliasing
$img->line(x1 => 0, y1 => 0, x2 => 399, y2 => 299,
          color => [255, 0, 0], aa => 1);

# an outlined box (no fill) and a filled one
$img->box(xmin => 20, ymin => 20, xmax => 120, ymax => 90,
          color => 'black');
$img->box(xmin => 140, ymin => 20, xmax => 240, ymax => 90,
          color => [200, 220, 255], filled => 1);

# a filled circle: center (200,150), radius 60
$img->circle(x => 200, y => 150, r => 60, color => [0, 128, 255]);

# an arc is a pie slice from d1 to d2 degrees
$img->arc(x => 200, y => 150, r => 80, d1 => 0, d2 => 120,
         color => 'yellow');

# a polygon from a list of [x, y] points
$img->polygon(points => [[300, 40], [380, 40], [340, 110]],
             color => [120, 200, 120]);

# a polyline is an open path; it takes no fill
$img->polyline(points => [[300, 130], [340, 200], [380, 130]],
              color => 'black', aa => 1);
```

(continues on next page)

(continued from previous page)

```
# flood fill from a seed point out to a border
$img->flood_fill(x => 5, y => 5, color => [245, 245, 245]);
```

Two traps worth stating once. `box` is filled only when you pass `filled => 1` (or a fill `=> pattern`); without it you get an outline. `polyline` draws an open path and does not accept fill, unlike `polygon` which closes and can fill. When a shape comes out as an outline you expected filled, or a fill silently does nothing, one of these two is why.

Loading, scaling, and saving

Reading and writing infer the format from the filename extension, or you can force it with `type =>`:

```
my $photo = Imager->new;
$photo->read(file => 'input.jpg') or die $photo->errstr;

# scale by a factor, or to fit a box
my $thumb = $photo->scale(scalefactor => 0.25);
my $fit    = $photo->scale(xpixels => 200, ypixels => 200, type =>
    'min');

$thumb->write(file => 'thumb.png') or die $thumb->errstr;
```

`scale` returns a new image; it does not modify the original. The `type => 'min'` form scales so the image fits inside the given box preserving aspect ratio; `'max'` fills the box (cropping the overflow when you then crop); `'nonprop'` ignores aspect ratio and stretches to exactly the pixels you named.

Filters

`filter` applies an effect in place, named by `type =>`. The *convolution* from the algorithms chapter is here as built-in Gaussian blur and unsharp masking, plus a general conv for your own kernel:

```
# Gaussian blur; larger stddev is more blur
$img->filter(type => 'gaussian', stddev => 2.5);

# unsharp mask to sharpen; both params optional
$img->filter(type => 'unsharpmask', stddev => 2.0, scale => 1.0);

# your own 1-D convolution coefficients (a horizontal blur here)
$img->filter(type => 'conv', coef => [0.25, 0.5, 0.25]);

# contrast, mosaic (pixelate), and inversion are all filters too
$img->filter(type => 'contrast', intensity => 1.4);
$img->filter(type => 'mosaic', size => 8);
$img->filter(type => 'hardinvert');
```

The full set includes `gaussian`, `gaussian2` (separate x and y deviations), `conv`, `un-`

sharpmask, contrast, mosaic, noise, hardinvert, autolevels, postlevels, and the procedural texture generators fountain, gradgen, radnoise, and turbnoise. Reach for the named filter before writing your own pixel loop; only drop to a conv coefficient list or manual access when nothing built in fits.

A complete program

Putting it together: a small poster with a background, some shapes, a blur pass, and a saved thumbnail.

```
use strict;
use warnings;
use Imager;

my $img = Imager->new(xsize => 400, ysize => 300, channels => 4)
    or die Imager->errstr;

$img->box(color => [30, 30, 60], filled => 1);           # dark_
    ↳bg
$img->circle(x => 120, y => 150, r => 70, color => [255, 200, 60]);
$img->circle(x => 260, y => 150, r => 70, color => [60, 200, 255]);
$img->filter(type => 'gaussian', stddev => 1.2);          # soften
$img->line(x1 => 0, y1 => 250, x2 => 399, y2 => 250,
    color => 'white', aa => 1);

$img->write(file => 'poster.png') or die $img->errstr;
$img->scale(scalefactor => 0.5)->write(file => 'poster-thumb.png')
    or die $img->errstr;
```

When Imager is the wrong tool

Imager is raster, so it has the raster limitation: it has a fixed resolution, and scaling past it degrades. For resolution-independent output (a logo that must print crisply at any size, a diagram destined for the web as markup) you want a vector tool, covered in *Cairo and SVG*. For an existing codebase built on GD or ImageMagick, or for their specific strengths, see the *next chapter*. And when the drawing needs to appear in a live window rather than a file, that is the *GUI toolkits chapter*. For everything that is “make or modify a bitmap on disk”, Imager is the answer.

GD and ImageMagick when you need them

Imager is the default, but two other raster libraries are worth knowing. GD is older, smaller, and everywhere: a great deal of existing Perl draws with it, and its color-index model is worth understanding even if you never choose it new. ImageMagick, through `Image::Magick`, is the heavyweight: the widest format support and the deepest bag of transformations, at the cost of a large dependency. This chapter shows each, and says when it beats Imager.

GD

GD wraps libgd. Its defining quirk is the **color index**: on a classic (palette) image you allocate each color once and then draw with the integer index it returns, not with the color itself. This is a relic of 256-color GIF images, and it still shows in the API even though truecolor images are now the norm.

```
use GD;

# the third argument requests a truecolor image; omit it for palette
my $im = GD::Image->new(200, 200, 1);

# even on truecolor, colors come from colorAllocate
my $white = $im->colorAllocate(255, 255, 255);
my $blue  = $im->colorAllocate(0, 0, 255);
my $red   = $im->colorAllocate(255, 0, 0);

$im->filledRectangle(0, 0, 199, 199, $white);
$im->line(0, 0, 199, 199, $blue);
```

The primitives are positional, and two of them differ from what you might expect. `arc` and `ellipse` take a center plus **width and height** (the full extents, not a radius), and angles in degrees:

```
# center (100,100), 80 wide, 80 tall, full circle
$im->arc(100, 100, 80, 80, 0, 360, $blue);
$im->filledArc(100, 100, 120, 60, 0, 180, $red); # a filled half-
→ellipse

$im->rectangle(10, 10, 60, 60, $blue);
$im->filledRectangle(140, 10, 190, 60, $red);
$im->setPixel(100, 100, $blue);
$im->fill(30, 30, $red); # flood fill
```

Polygons are drawn from a `GD::Polygon` object, not raw coordinates:

```
my $poly = GD::Polygon->new;
$poly->addPt(50, 10);
$poly->addPt(150, 10);
$poly->addPt(100, 90);
$im->polygon($poly, $blue);
$im->filledPolygon($poly, $red);
```

Output is a method per format that returns the encoded bytes, so you print them to a filehandle yourself:

```
open my $fh, '>', 'out.png' or die $!;
binmode $fh;
print $fh $im->png; # or ->jpeg($quality), ->gif
close $fh;
```

When GD wins: you are maintaining code that already uses it (a large amount of older Perl does), you want the smallest dependency that still draws, or you specifically need

libgd behavior to match another tool in a pipeline that also uses libgd. For new drawing work with no such constraint, Imager's named-color and hash-argument API is friendlier and its filter set is richer.

ImageMagick (Image::Magick)

Image::Magick, also called PerlMagick, ships with ImageMagick itself and exposes its full engine. Its strengths are **format breadth** (it reads and writes formats nothing else on CPAN touches) and **transformations** (hundreds of operations from Resize and Blur through Distort, Morphology, and format-specific tricks). Its weakness is the dependency: you need a matching ImageMagick library installed, and it is large.

The API is verb-oriented: you create an object, Read or size it, then call named operations, each returning an error string that is empty on success.

```
use Image::Magick;

my $im = Image::Magick->new;
$im->Set(size => '200x200');
$im->ReadImage('canvas:white');           # a blank white canvas

# drawing is one Draw call per primitive, SVG-path-like
$im->Draw(primitive => 'line',    points => '0,0 199,199',
          stroke => 'blue');
$im->Draw(primitive => 'circle',  points => '100,100 100,40',
          stroke => 'red', fill => 'none');
$im->Draw(primitive => 'polygon', points => '50,10 150,10 100,90',
          fill => 'skyblue');

# transformations are named methods
$im->Blur(radius => 0, sigma => 1.5);
$im->Resize(geometry => '100x100');

my $err = $im->Write('out.png');
warn "$err" if $err;
```

The circle primitive takes the center followed by any point on the perimeter (so 100, 100 100,40 is center (100,100) with radius 60), which is ImageMagick's convention rather than a radius. Every geometry argument is ImageMagick's string mini-language ('200x200', '100x100+10+10'), so it is worth a glance at ImageMagick's own geometry documentation for the details.

When ImageMagick wins: you need a format nothing else reads, you are doing heavy batch transformation (thumbnailing thousands of files, format conversion pipelines), or you want one of its many operations that would be a project to reimplement. For interactive drawing or a tight dependency footprint, it is more machine than you need, and Imager or GD fit better.

Choosing

You want ...	Reach for
New raster drawing, friendly API, good filters	Imager
Smallest dependency, or an existing GD codebase	GD
Widest formats, heavy batch transforms	Image::Magick

All three are raster and share the raster ceiling: fixed resolution, degradation on upscaling. When you need output that scales cleanly, the *next chapter* moves to vector graphics with Cairo and SVG.

Vector graphics: Cairo and SVG

Raster tools commit to a resolution; vector tools describe shapes and decide the pixels later, or never. This chapter covers the two vector routes in Perl. Cairo is a full anti-aliased 2D engine that rasterizes paths to a pixel surface (or to PDF and SVG): you draw with real-number coordinates and it works out the coverage. SVG generates SVG markup, a text description of shapes that a browser or renderer draws later at whatever size it likes. Use Cairo when you need pixels now with the quality of a proper rasterizer; use SVG when you want resolution-independent output as a file that other tools consume.

Cairo also returns in *chapter 9*: pperl ships a native reimplementaion of it, so the same code runs with no CPAN Cairo installed, and that chapter shows how the native binding works. Here it is the ordinary CPAN module.

Cairo: paths and the drawing model

Cairo's model is a **path** built up from segments, then either **stroked** (the outline drawn) or **filled** (the interior painted). You set a source color, build a path with `move_to` and `line_to` and curves, and commit it. Colors are floats in 0 .. 1, not 0 .. 255.

```
use Cairo;

# an ARGB surface is a pixel buffer with alpha
my $surface = Cairo::ImageSurface->create('argb32', 200, 200);
my $cr = Cairo::Context->create($surface);

# paint a white background: set source, then paint fills everything
$cr->set_source_rgb(1, 1, 1);
$cr->paint;

# a stroked path
$cr->set_source_rgb(0, 0, 0);
$cr->set_line_width(3);
$cr->move_to(20, 20);
$cr->line_to(180, 20);
$cr->curve_to(180, 100, 20, 100, 20, 180); # a bezier curve
$cr->stroke;
```

(continues on next page)

(continued from previous page)

```
# a filled, semi-transparent circle (arc angles are in radians)
$cr->arc(100, 100, 40, 0, 6.28318);          # full circle = 2*pi
$cr->set_source_rgba(1, 0, 0, 0.5);          # red at 50% alpha
$cr->fill;

$surface->write_to_png('output.png');
```

Three things to internalize. Coordinates are real numbers, so `move_to(100.5, 47.25)` is meaningful and the *half-pixel offset* from the foundations chapter is how you get crisp thin lines. Arc angles are in **radians**, not degrees, so a full circle is $2 * \pi$. And `set_source_rgba`'s alpha gives you the *compositing* for free: fill or stroke with alpha below 1 and Cairo blends over what is already there.

`stroke` and `fill` consume the current path. If you want to both fill and stroke the same shape, use `fill_preserve` (or `stroke_preserve`), which keeps the path for a second operation:

```
$cr->arc(100, 100, 40, 0, 6.28318);
$cr->set_source_rgb(0.4, 0.8, 1);
$cr->fill_preserve;                # fill, but keep the path
$cr->set_source_rgb(0, 0, 0);
$cr->set_line_width(2);
$cr->stroke;                       # now outline the same circle
```

Premultiplied alpha

The `argb32` surface stores pixels with *premultiplied alpha*: each color channel is already multiplied by its alpha. This is invisible when you draw through Cairo's API, but it matters the moment you read raw pixel bytes off the surface or feed Cairo's buffer to another library, because the bytes are not the straight RGBA you might assume. When you cross that boundary, un-multiply first. Within Cairo's own drawing it never comes up.

SVG: markup, not pixels

SVG is pure Perl and does something different: it builds an XML document describing shapes, and produces text. Nothing is rasterized; the consumer (a browser, Inkscape, a PDF converter) draws it. That makes it perfect for output that must stay sharp at any zoom, or that other tools will process further.

```
use SVG;

my $svg = SVG->new(width => 200, height => 200);

# a group carries shared style; children inherit it
my $g = $svg->group(id => 'shapes',
                   style => { stroke => 'black', fill => 'none' });

$g->line(x1 => 0, y1 => 0, x2 => 200, y2 => 200);
```

(continues on next page)

(continued from previous page)

```

$g->circle(cx => 100, cy => 100, r => 50, style => { fill =>
  ↳ 'skyblue' });
$g->rect(x => 10, y => 10, width => 40, height => 40);
$g->path(d => 'M10 190 L100 100 L190 190 Z'); # a triangle via_
  ↳ path data

$svg->text(x => 60, y => 110)->cdata('Hello');

print $svg->xmlify;

```

The style hash maps to SVG's presentation attributes: stroke, fill, stroke-width, opacity, and the rest. A group (aliased `g`) with a style is how you avoid repeating attributes on every element; children inherit and can override. Text is attached with a chained `cdata` because the character content is a child node, not an attribute.

For a polygon from computed points, `get_path` assembles the points string for you:

```

my @x = (50, 150, 100);
my @y = (10, 10, 90);
my $points = $svg->get_path(x => \@x, y => \@y, -type => 'polygon');
$g->polygon(%$points, style => { fill => 'coral' });

```

`xmlify` (also spelled `to_xml` or `render`) returns the finished document as a string. Write it to a `.svg` file and it opens in any browser; hand it to a converter for PDF or PNG.

Cairo or SVG?

You want ...	Reach for
Pixels now, anti-aliased, from real-number coordinates	Cairo
Vector output as a file for browsers or other tools	SVG
To render the same drawing to PNG, PDF, and SVG from one API	Cairo
No compiled dependency at all (pure Perl)	SVG

The two also compose: generate structure as SVG markup, then render it to pixels with a rasterizer when you need a bitmap. For applied vector output in the specific shape of charts and graphs, and for reading the metadata already inside image files, the *next chapter* covers the pragmatic bookends of raster and vector work.

Image metadata and applied 2D charting

Two pragmatic corners of 2D work round out the toolkit arc. First, reading the data already inside image files: dimensions, format, and the EXIF block a camera writes. Second, the most common reason people draw at all, which is turning numbers into a chart. Neither is glamorous, and both come up constantly.

Reading metadata: Image::Info

Before you draw on or resize an image, you often just need to know about it: how big is it, what format, what did the camera record. `Image::Info` is pure Perl, reads the headers without decoding the whole image, and returns a plain hash reference.

```
use Image::Info qw(image_info dim);

my $info = image_info('photo.jpg');
die $info->{error} if $info->{error};

my ($w, $h) = dim($info);
print "format: $info->{file_media_type}\n";      # image/jpeg
print "size:   ${w}x${h}\n";
print "created: $info->{DateTime}\n" if $info->{DateTime};
```

`dim` is a convenience that pulls width and height out of the hash. The rest of the keys depend on the format and on what the file carries: resolution, color type, and the common EXIF fields for photographs. For a quick “what is this file” it is all you need, and it costs almost nothing because it never decodes the pixels.

Reading and writing metadata: Image::ExifTool

When you need the full metadata surface, and especially when you need to **write** it, `Image::ExifTool` is the definitive tool. It handles the metadata of essentially every image and media format, reads thousands of tags, and can edit them in place without touching the image data.

```
use Image::ExifTool qw(:Public);

my $et = Image::ExifTool->new;
$et->ExtractInfo('photo.jpg');

my $model = $et->GetValue('Model');           # camera model
my $when  = $et->GetValue('DateTimeOriginal');
print "shot on $model at $when\n" if $model;

# rewrite a tag and save
$et->SetNewValue('Artist' => 'Jane Doe');
$et->WriteInfo('photo.jpg');                  # edits metadata
↳only
```

`Image::Info` answers “how big, what format” cheaply; `Image::ExifTool` is the one to reach for when the metadata itself is the job (stripping location tags for privacy, batch-tagging a photo library, reading maker-note fields). Both leave the pixels alone.

Charting: the honest state of the art

Charting is applied 2D drawing: axes, bars, lines, and labels composed from the primitives you already know. The Perl charting landscape is thinner than it once was. Several once-popular modules are now unmaintained, so this section leads with what still works and names the rest plainly.

GD::Graph for quick raster charts

GD::Graph builds on GD and is still maintained. It is the shortest path to a bar, line, or pie chart as a PNG. You give it data as a set of parallel arrays (the first is the x labels, the rest are data series) and it renders.

```
use GD::Graph::bars;

my @data = (
    ['Jan', 'Feb', 'Mar', 'Apr'],      # x-axis labels
    [ 12,   19,   8,   23 ],          # one data series
);

my $chart = GD::Graph::bars->new(400, 300);
$chart->set(
    title      => 'Widgets shipped',
    y_max_value => 25,
    dclrs      => ['blue'],
) or die $chart->error;

my $gd = $chart->plot(\@data) or die $chart->error;

open my $fh, '>', 'chart.png' or die $!;
binmode $fh;
print $fh $gd->png;                    # plot() returns a
GD::Image
close $fh;
```

GD::Graph::bars, ::lines, ::pie, and ::points are the workhorses. plot returns a GD::Image, so everything from the *GD chapter* about output applies. This is the pragmatic default for a chart on disk.

SVG for vector charts

When the chart must stay sharp at any size, or live in a web page as markup, build it as SVG directly. There is no chart abstraction to learn: you compute the bar rectangles and axis lines yourself and emit them. That is more code than GD::Graph, but it is resolution-independent and needs no compiled dependency.

```
use SVG;

my @values = (12, 19, 8, 23);
my $svg = SVG->new(width => 400, height => 300);
my $base = 260;                      # baseline y
my $bw   = 60;                       # bar width

for my $i (0 .. $#values) {
    my $h = $values[$i] * 8;          # scale to pixels
    my $x = 40 + $i * ($bw + 20);
    $svg->rect(x => $x, y => $base - $h, width => $bw, height => $h,
               style => { fill => 'steelblue' });
}
```

(continues on next page)

(continued from previous page)

```
}  
$svg->line(x1 => 30, y1 => $base, x2 => 380, y2 => $base,  
          style => { stroke => 'black' });  
  
print $svg->xmlify;
```

For anything beyond a simple bar or line chart the hand-rolled SVG grows quickly, but for exactly the common cases it is clean and dependency-free.

The legacy options, named

Two charting modules you will find referenced online are effectively unmaintained and are noted here only so their state is not a surprise. **Chart::Clicker** produces attractive Cairo-rendered charts but has not seen a release in years and carries a large dependency tree (including Moose); if you already run it, it works, but it is not a choice for new code. **Chart::Gnuplot** drives an external gnuplot binary and is likewise dormant; if you have gnuplot installed and want its plotting power, calling it directly (or through a thin wrapper of your own) is more maintainable than depending on the abandoned module. For scientific plotting specifically, the PDL ecosystem's own graphics modules are the better-supported path and belong with PDL rather than here.

Where this leaves the toolkit arc

You can now create, load, draw on, filter, save, vectorize, inspect, and chart 2D images entirely on disk. The one thing missing is a **screen**: getting a drawing into a live, interactive window. That is the *next chapter*, and it is also the bridge to the two capstone projects, which both need a window to run in.

2D on a screen: GUI canvases

Everything so far drew to a file. This chapter draws to a **window**: an interactive canvas that repaints, responds to the mouse and keyboard, and can animate. Every GUI toolkit works the same way in outline: you create a window, you give it a paint handler that draws its contents, and the toolkit calls that handler whenever the window needs re-drawing. Drawing inside the handler uses primitives that are, by now, familiar: lines, ellipses, polygons, fills.

The Perl GUI landscape has a clear best option for graphics and several others with trade-offs. This chapter leads with **Prima**, because it is actively maintained, self-contained, and has a real graphics core; then covers Tk and Gtk3 for when you have a reason to prefer them; and is honest about Wx, which is no longer a healthy choice.

Prima: the self-contained option

Prima is a GUI toolkit written for Perl with its own graphics engine. It does not wrap an external toolkit you have to install and match versions with; it draws natively (on X11 on Linux) and ships its own primitives, image handling, and fonts. It is maintained, and it is the recommendation for new graphics-heavy GUI work in Perl.

A minimal window with a paint handler:

```

use Prima qw(Application);

my $win = Prima::MainWindow->new(
    text    => 'Prima demo',
    size    => [400, 300],
    onPaint => sub {
        my ($self, $canvas) = @_;          # the handler gets a canvas
        $canvas->color(c1::White);
        $canvas->bar(0, 0, $canvas->size);   # bar is a filled
        ↪rectangle
        $canvas->color(c1::Red);
        $canvas->fill_ellipse(200, 150, 120, 90);
        $canvas->color(c1::Black);
        $canvas->line(0, 0, 400, 300);
    },
);

Prima->run;

```

Two naming notes that catch newcomers. Prima's **filled rectangle** is `bar`; plain rectangle draws the outline. The polygon fill is `fillpoly` (one word), taking an array reference of coordinates. Colors come from the `c1::` constants (`c1::Red`, `c1::White`, and so on) or a packed RGB integer, and are set as the drawing state with `color` before you draw, rather than passed to each primitive.

The `Prima::Drawable` primitives are the full set you expect: `line`, `ellipse` and `fill_ellipse`, `arc`, `sector` and `fill_sector`, `polyline`, `fillpoly`, `bar`, `rectangle`, `flood_fill`, and `put_image` for blitting a bitmap. Drawing state includes `color`, `backColor`, `lineWidth`, and `fill` patterns.

Offscreen drawing and saving

The same primitives work on a `Prima::Image`, an offscreen surface, so you can draw without a window and save the result. This is how a Prima program exports what it drew:

```

my $img = Prima::Image->new(width => 200, height => 200, type =>
    ↪im::RGB);
$img->begin_paint;
$img->color(c1::Blue);
$img->fill_ellipse(100, 100, 80, 80);
$img->end_paint;
$img->save('out.png');

```

`begin_paint` and `end_paint` bracket drawing onto the offscreen image, and `save` writes it in a format inferred from the extension. This dual nature (same drawing code for the screen and for a file) is why the *capstone chapters* build on Prima.

Tk: the classic Canvas

Tk (Perl/Tk) is the long-standing option, and its Canvas widget has a different model worth knowing: it is **retained-mode**. Instead of a paint handler that redraws everything, you create canvas items (a line, an oval, a polygon) that the widget remembers and redraws for you, and you move or reconfigure them by their id. This suits diagrams and editors where objects persist and get manipulated.

```
use Tk;

my $mw = MainWindow->new;
my $canvas = $mw->Canvas(-width => 400, -height => 300)->pack;

$canvas->createLine(0, 0, 400, 300, -fill => 'blue', -width => 2);
my $oval = $canvas->createOval(160, 100, 240, 180, -fill => 'red');
$canvas->createPolygon(50, 10, 150, 10, 100, 90, -fill => 'green');

# move the remembered oval later, by id
$canvas->move($oval, 20, 0);

MainLoop;
```

Tk is stable and works, though it is in maintenance rather than active development and carries a few inherited CVEs in its bundled Tcl/libpng. For a persistent-object diagram or a quick classic GUI it is a reasonable pick; for a redraw-every-frame animation the immediate-mode Prima model fits better.

Gtk3: drawing through Cairo

Gtk3 binds GTK 3 through GObject introspection. Its drawing story is that a `Gtk3::DrawingArea` widget emits a draw signal handed a *Cairo* context, so you draw with the Cairo API from the vector chapter inside the handler. If your application is already a GTK application, this is the natural way to add custom graphics; if it is not, pulling in the whole GTK stack (the GTK 3 development libraries and GObject introspection data) is a large dependency for graphics alone.

```
use Gtk3 -init;

my $window = Gtk3::Window->new('toplevel');
my $area = Gtk3::DrawingArea->new;
$window->add($area);
$window->set_default_size(400, 300);

$area->signal_connect(draw => sub {
    my ($widget, $cr) = @_;           # $cr is a Cairo context
    $cr->set_source_rgb(0.2, 0.4, 0.9);
    $cr->arc(200, 150, 80, 0, 6.28318);
    $cr->fill;
    return 0;
});
```

(continues on next page)

(continued from previous page)

```
$window->show_all;
Gtk3->main;
```

Everything you know about Cairo applies unchanged inside that handler. Gtk3 is usable but in slow maintenance, and the dependency weight means it earns its place mainly when GTK is already in the picture.

Wx: mention, not recommendation

You will find Wx (wxPerl) referenced as the way to build native-looking cross-platform GUIs in Perl, and historically it was. It is included here so its current state is clear: Wx and its `Alien::wxWidgets` build dependency have not had a release in years, and a large fraction of automated build reports fail against current `wxWidgets`. It is not a choice for new work. If you need a GUI with custom drawing today, `Prima` is the maintained self-contained option and `Tk` the maintained classic; reach for either of those instead.

Choosing

You want ...	Reach for
Maintained, self-contained, graphics-heavy new GUI	Prima
A retained-mode canvas of persistent objects	Tk
Custom drawing inside an existing GTK application	Gtk3
Per-frame animation and a realtime loop	Prima

With a window and a paint handler in hand, the toolkit arc is complete. The *next arc* goes further than any single CPAN module: it uses `ppperl`'s native FFI to reach C graphics libraries directly, and then spends everything in the guide on two projects you can run, both built on the `Prima` foundation from this chapter.

Reaching further: `Peta::FFI` for graphics

The toolkit chapters covered what CPAN already wraps. This chapter, and the rest of the guide, goes further: reaching a C graphics library that has no CPAN binding at all, using `ppperl`'s native FFI. This is a `ppperl`-exclusive capability. Binding a C library the traditional way means writing a C shim, compiling it, and shipping a build toolchain with your code; `ppperl` calls the C library directly, the same mechanism that already gives you `Math::GMP` with no compiled binding to build at all.

This is not `FFI::Platypus`. `Peta::FFI` is built into the runtime, and its curated bindings are native Rust, not Perl-side declarations. The distinction matters for graphics because a real graphics binding needs to manage C-allocated objects (a surface, a context) across many calls, and `ppperl`'s model handles that the way the C library itself does.

The authoritative reference for the FFI layer is the *Auto-FFI page*; this chapter applies it to graphics specifically.

The three levels

Peta::FFI is best understood as three levels, from raw to curated.

Level 1, raw calls. You name a library, a function, and a type-signature string, and pperl dispatches the call through libffi. Nothing is generated ahead of time; any signature you can spell works.

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

my $lib = dlopen("libm.so.6");
my $r = call($lib, "sqrt", "(d)d", 2.0);    # sqrt(2) = 1.4142...
dlclose($lib);
```

The signature "(args)ret" uses one letter per type: v void, i int, l long, L unsigned long or size_t, d double, f float, p a C string in, and P a mutable output buffer. That set covers a surprising amount of a C API: anything whose arguments and return are scalars, strings, or a byte buffer.

Level 2, a data-description layer. Many C functions take or return **structs**, not scalars, and the raw type codes cannot describe a struct's layout. The planned answer is a declarative layer where you describe a record's fields once and let the FFI marshal it. This is the emerging middle path; it is not a shipping module yet, and this guide is explicit at each point about where a binding would need it.

Level 3, curated bindings. The top level is a hand-written native binding: a Rust module that opens the library once, resolves every symbol it needs, and presents a clean Perl API, managing the C-allocated objects behind Perl values. `Math::GMP` is exactly this: use `Math::GMP` and you get arbitrary-precision integers with operator overloading, and nowhere did you touch `dlopen` or a signature string. The [next chapter](#) takes one of these that pperl already ships for a graphics library, Cairo, and shows how it works.

What raw calls can and cannot reach

The dividing line for graphics is the struct. A great deal of a C graphics API is reachable at Level 1 right now, because it is scalars and opaque handles:

- Functions that take and return **numbers** (set a line width, a radius, a color component) map straight onto i, d, f, and friends.
- Functions that take a **C string** (load a font by name, open a file) use p.
- Functions that fill a **caller-provided buffer** (read pixels into memory you own) use P.
- An **opaque handle** returned by the library (a surface pointer, a context pointer) is just an address. pperl already represents a library handle as an integer, and the curated pattern in the next chapter stores these object handles the same way: as an integer inside a blessed Perl reference.

Where raw calls stop:

- **Structs passed or returned by value.** A function taking a `Color { r, g, b, a }` by value, or returning a `Rectangle`, cannot be described by the scalar type codes. This is the Level 2 gap.

- **Callbacks.** A library that calls back into your code (an event loop that invokes a handler, a draw signal) needs a C function pointer that re-enters Perl. That is beyond the raw layer.
- **Unions.** An event type that is a union of many struct shapes (the classic windowing event record) combines both problems above.

These are not permanent walls; they are the frontier the FFI layers are growing toward. This guide's value is telling you exactly which side of the line a given task falls on, so you neither avoid what works nor promise what does not.

Why a curated binding, not raw calls, for real work

You could drive a graphics library entirely through Level 1 calls, and for a quick experiment that is fine. For anything real, a Level 3 curated binding is the right unit, for three reasons.

First, **object lifetime**. A graphics context is allocated by the library and must be freed exactly once. A curated binding ties that lifetime to a Perl value's DESTROY, so the C object is released when the Perl object goes away, with no manual bookkeeping. Raw calls leave that entirely to you.

First-class Perl **ergonomics** is the second reason: `$surface->fill` reads better and is harder to get wrong than a call with a hand-written signature at every use site.

Third, **the signature is written once**. In a curated binding the "(ppd)i" strings live in one Rust module, checked once; every caller just uses the method. Scattering raw signatures through application code is how you get a mismatch that corrupts the stack.

The *Cairo showcase* makes all three concrete with the Cairo binding pperl ships, a real vector library whose API is almost entirely scalars and one opaque handle, which is precisely the shape that reaches cleanly today.

The shape of what follows

The next chapter builds a curated Cairo binding the way `Math::GMP` is built, so you can see a graphics library brought up on pperl end to end. Then the two capstones use the maintained toolkit stack from Arc B, anchored on Prima, to build a drawing application and a 2D game you can run, with the FFI thread woven in where it earns its place and honestly flagged where the frontier begins. The *final chapter* draws the map of what is left.

A native Cairo binding, shipped

The *vector chapter* drew with Cairo as an ordinary CPAN module. This chapter shows the other side of that same code: pperl ships a **native reimplementation of Cairo**, so use Cairo works on pperl with no CPAN Cairo distribution installed and no C compiler involved. The Perl you write is identical; what backs it is pperl's own FFI reaching libcairo directly. Cairo makes the ideal worked example because its C API is almost entirely scalars and one opaque handle, which is precisely the shape that reaches cleanly today.

This is a real, shipped binding, not a sketch. Reading it tells you what a curated native graphics binding is, how the pieces fit, and how to recognize (or ask for) the same treat-

ment for another library. Building such a binding is a runtime task; what you write as a user is the Perl on the far side of it, and that Perl is just the [Cairo API](#) you already saw.

The Perl API: exactly the CPAN one

The native binding exposes the **real Cairo package names and methods**: `Cairo::ImageSurface`, `Cairo::Context`, `Cairo::PdfSurface`, and the rest, with the upstream method signatures. Code written for CPAN Cairo runs unchanged; ported Cairo test suites pass against it. The program from the vector chapter is the program here:

```
use Cairo;

my $surface = Cairo::ImageSurface->create('argb32', 200, 200);
my $cr      = Cairo::Context->create($surface);

$cr->set_source_rgb(1, 1, 1);
$cr->paint;                                # white background
$cr->set_source_rgb(0, 0, 0);
$cr->set_line_width(3);
$cr->move_to(20, 20);
$cr->line_to(180, 20);
$cr->stroke;
$cr->arc(100, 100, 40, 0, 6.28318);
$cr->set_source_rgba(1, 0, 0, 0.5);
$cr->fill;

$surface->write_to_png('out.png');
# the surface and context free their C objects when they go out of
↳ scope
```

Nothing in that program hints at how it is implemented. There is no `dlopen`, no signature string, no binding object: just Cairo. The rest of this chapter is what makes use Cairo do that on pperl, and the pattern is the same one behind [Math::GMP](#).

Step 1: open the library once, resolve the symbols

The binding opens `libcairo.so.2` a single time, when the module loads, and resolves each C function it needs into a table of function pointers held for the life of the process. This is the same shape as the [Math::GMP](#) binding, which opens `libgmp` once and resolves its `mpz_*` symbols. For Cairo the resolved symbols are the ones behind the methods above:

```
open libcairo.so.2 once when the module loads
resolve and store, as function pointers:
    cairo_image_surface_create(format, width, height) -> surface*
    cairo_create(surface*)                           -> context*
    cairo_move_to(context*, x, y)
    cairo_line_to(context*, x, y)
    cairo_arc(context*, xc, yc, radius, a1, a2)
    cairo_set_source_rgb(context*, r, g, b)
    cairo_set_source_rgba(context*, r, g, b, a)
```

(continues on next page)

(continued from previous page)

```

    cairo_set_line_width(context*, w)
    cairo_stroke(context*)  cairo_fill(context*)  cairo_
    ↪ paint(context*)
    cairo_surface_write_to_png(surface*, path) -> status
    cairo_destroy(context*)  cairo_surface_destroy(surface*)

```

If libcairo is not installed on the system, the binding reports it once with a helpful message rather than crashing. Every one of those C functions has a signature the FFI's type codes already express: the arguments are the opaque handle (an address, so an integer), doubles, and a string path. There is no by-value struct in the core drawing API, which is why Cairo is the clean example.

Cairo is a large library, so the real binding is organized by package: `Cairo::Surface`, `Cairo::Context`, `Cairo::Path`, `Cairo::Pattern`, `Cairo::Matrix`, `Cairo::Font`, and `Cairo::Region` each resolve the symbols they need. That is an implementation convenience; the user sees one Cairo.

Step 2: the opaque handle as a blessed integer

Cairo hands back a `cairo_surface_t*` and a `cairo_t*`: pointers to C structs whose insides you never touch. The binding does not need to understand their layout, only to hold the pointer and pass it back on every call. `ppperl` represents such a handle the same way `Math::GMP` holds its `mpz_t` pointer: as an **integer inside a blessed reference**.

```

Cairo::ImageSurface object = bless(\ (surface_ptr as integer),
    ↪ 'Cairo::ImageSurface')
Cairo::Context object      = bless(\ (context_ptr as integer),
    ↪ 'Cairo::Context')

```

When a method is called, the binding reads the integer back out of the blessed reference, casts it to the C pointer, and calls the function. The blessing gives you method dispatch (`$cr->stroke`), inheritance (every concrete surface class is a `Cairo::Surface` subclass), and, crucially, a DESTROY hook. The only difference from `Math::GMP` is that the opaque thing is a drawing surface instead of a big integer.

Step 3: lifetime tied to DESTROY

A C graphics object must be freed exactly once. Leak it and you burn memory every drawing; free it twice and you corrupt the heap. The binding ties the C object's life to the Perl object's: when the blessed reference goes out of scope, Perl calls DESTROY, and the binding calls the matching C destructor, respecting cairo's reference counting.

```

Cairo::Context DESTROY -> cairo_destroy(context_ptr)
Cairo::Surface DESTROY -> cairo_surface_destroy(surface_ptr)

```

`Math::GMP` does precisely this: its destructor calls `mpz_clear` and frees the `mpz_t`. You, the user, never call a free function; the drawing objects clean up when the Perl variables holding them go away. The binding enforces the ordering cairo requires, releasing a context before the surface it draws to.

Step 4: the methods

Each Perl method is a small native function that reads the handle, reads the scalar arguments, and makes one C call. `Cairo::Context::move_to` reads the context pointer and two doubles and calls `cairo_move_to`; `Cairo::Context::set_source_rgba` reads the pointer and four doubles and calls `cairo_set_source_rgba`; `Cairo::Surface::write_to_png` reads the pointer and a path string. The mapping is one method to one C call:

```
Cairo::Context::move_to($cr, $x, $y)
    -> read context ptr from $cr; call cairo_move_to(ptr, x, y)

Cairo::Context::set_source_rgba($cr, $r, $g, $b, $a)
    -> call cairo_set_source_rgba(ptr, r, g, b, a)

Cairo::Surface::write_to_png($surface, $path)
    -> call cairo_surface_write_to_png(ptr, path)
```

The C-level signatures live in the binding and are settled there once. A user calling `$cr->set_source_rgba(1, 0, 0, 0.5)` never writes a signature and cannot get one wrong. That single point of truth for the C calls is the third of the *three reasons* a curated binding beats scattered raw calls: object lifetime, ergonomics, and the signature written down once.

What this reaches, and what it does not

With those four pieces (open once, handle as a blessed integer, DESTROY frees, methods over the resolved function table) pperl delivers a real, memory-safe Cairo with no CPAN distribution installed, no C compiler, and no build step. The full *vector chapter* drawing model is available, and the objects manage themselves.

What the pattern does not, by itself, reach is the same frontier the *primer* named. Cairo avoids all of it in its core drawing API, which is why it is the first graphics library brought up this way. A library that passes colors or points **by value as structs**, that **calls back** into your code, or that hands you **union-typed events** needs more than this pattern has: the by-value and union cases wait on the *data-description layer*, and callbacks need function-pointer support that re-enters Perl. The *2D game capstone* hits exactly that wall with windowing and event libraries, and says so at the point it does.

The takeaway

A graphics library whose C API is scalars plus opaque handles can be brought up on pperl as a curated native binding, and Cairo is the first one shipped: open the library once, hold each C object as a blessed integer, free it in DESTROY, and back each method with its C call settled in one place. The result is the upstream Cairo API, running natively. Other libraries of the same shape are on the way. The *next chapter* stops describing and starts building: a drawing application on the maintained toolkit stack, assembling the whole guide into something that runs.

Capstone: a drawing application

This chapter assembles the guide into a program you run: a small drawing application. Click and drag to paint, pick a color, clear the canvas, and save the result to a PNG. It is built on *Prima*, the maintained self-contained toolkit from the GUI chapter, because it gives us a window, mouse events, and an offscreen surface that saves itself, all without an external toolkit to install.

The design is the one every paint program uses. There is an **offscreen image** that holds the actual painting, and a **window** that displays it. The mouse draws onto the offscreen image; the window's paint handler just copies that image to the screen. Separating the two means the painting survives window redraws (resize, occlusion) for free, and saving is trivial because the real drawing already lives in an image object.

The model: an offscreen image

The painting itself is a `Prima::Image`, drawn on with the same primitives you would use on a window canvas. We start it white:

```
use strict;
use warnings;
use Prima qw(Application Buttons);

my $W = 640;
my $H = 480;

# the real drawing lives here, offscreen
my $canvas = Prima::Image->new(width => $W, height => $H, type =>
    ↪im::RGB);
$canvas->begin_paint;
$canvas->color(c1::White);
$canvas->bar(0, 0, $W, $H);          # bar is Prima's filled
    ↪rectangle
$canvas->end_paint;

my $brush_color = c1::Black;        # current paint color
my $last;                          # last mouse point while
    ↪dragging
```

`begin_paint` and `end_paint` bracket any drawing onto the offscreen image. `bar` fills a rectangle (recall from the *GUI chapter* that Prima's filled rectangle is `bar`, not `rect`-`angle`). We hold the current brush color and the previous drag point as ordinary Perl variables.

The window: display and input

The window's job is to show `$canvas` and to translate mouse motion into paint strokes. Its `onPaint` handler copies the offscreen image to the screen with `put_image`; its mouse handlers draw onto the offscreen image and ask the window to repaint.

```

my $window = Prima::MainWindow->new(
    text => 'pperl paint',
    size => [$W, $H + 40],           # extra height for the buttons

    onPaint => sub {
        my ($self, $canvas_widget) = @_;
        $canvas_widget->put_image(0, 40, $canvas); # blit the
        drawing
    },

    onMouseDown => sub {
        my ($self, $btn, $mod, $x, $y) = @_;
        $last = [$x, $y - 40];      # translate for the button
        strip
    },

    onMouseMove => sub {
        my ($self, $mod, $x, $y) = @_;
        return unless $last;        # only while a button is held
        my @pt = ($x, $y - 40);
        stroke($last, \@pt);
        $last = \@pt;
        $self->repaint;
    },

    onMouseUp => sub { $last = undef; },
);

```

The `y - 40` offset keeps a strip at the bottom of the window for the buttons; the drawing area is everything above it. `onMouseMove` fires continuously, so we draw a short line segment from the previous point to the current one and remember the current point. Dragging fast still produces a connected stroke because we always join consecutive points rather than dotting individual pixels.

The brush: drawing onto the offscreen image

`stroke` draws one segment of the brush onto the offscreen `$canvas`. Because the painting is a normal `Prima::Image`, we use the same drawing calls as anywhere else, bracketed by `begin_paint` and `end_paint`:

```

sub stroke {
    my ($from, $to) = @_;
    $canvas->begin_paint;
    $canvas->color($brush_color);
    $canvas->lineWidth(3);
    $canvas->line($from->[0], $from->[1], $to->[0], $to->[1]);
    # round the joints so fast strokes stay smooth
    $canvas->fill_ellipse($to->[0], $to->[1], 3, 3);
    $canvas->end_paint;
}

```


Setting `lineWidth` before the line gives a thicker stroke; the small `fill_ellipse` at the endpoint rounds off the joins so a fast zig-zag does not show gaps at the corners. This is the anti-aliasing-adjacent detail from the [algorithms chapter](#) showing up in practice: thick lines meeting at an angle leave a notch unless you round the joint.

Controls: color, clear, and save

A row of buttons along the bottom gives the tool its verbs. Each is a `Prima::Button` with an `onClick`. Color buttons set `$brush_color`; clear repaints the offscreen image white; save writes it to disk.

```
my @palette = (
    ['Black', cl::Black], ['Red', cl::Red],
    ['Green', cl::Green], ['Blue', cl::Blue],
);
my $x = 5;
for my $entry (@palette) {
    my ($label, $color) = @$entry;
    $window->insert(Button =>
        text      => $label,
        origin    => [$x, 5],
        size      => [60, 30],
        onClick   => sub { $brush_color = $color },
    );
    $x += 65;
}

$window->insert(Button =>
    text => 'Clear', origin => [$x, 5], size => [60, 30],
    onClick => sub {
        $canvas->begin_paint;
        $canvas->color(cl::White);
        $canvas->bar(0, 0, $W, $H);
        $canvas->end_paint;
        $window->repaint;
    },
);
$x += 65;

$window->insert(Button =>
    text => 'Save', origin => [$x, 5], size => [60, 30],
    onClick => sub {
        $canvas->save('painting.png')
        and print "saved painting.png\n";
    },
);

Prima->run;
```

`insert(Button => ...)` adds a child widget to the window. `$canvas->save` writes the offscreen image to a PNG, format inferred from the extension, returning true on

success. Because the drawing has lived in an image object the whole time, saving is a single call with nothing to reconstruct.

Running it

Assemble the pieces above into one file and run it with `ppperl`. A window opens with a white canvas and a button strip; drag to paint, click a color to switch, Clear to reset, Save to write `painting.png`. Every piece traces back through the guide: the offscreen image and its primitives from the [GUI chapter](#), the stroke-joining from the [algorithms chapter](#), the color model from [foundations](#), and the save path that is one method because the painting was always a real image.

Where the FFI thread would enter

This capstone stays on the maintained CPAN stack, and deliberately so: it is meant to run today, unmodified. The [FFI thread](#) enters when you outgrow what the stack offers, for instance wanting a C brush-engine library with no Perl binding for pressure-sensitive strokes. That is reached exactly as the [Cairo showcase](#) showed: a curated binding presenting the C library as Perl objects. For a paint program, Prima's own primitives are more than enough, and the honest engineering choice is to use them. The [next capstone](#) is where the FFI frontier genuinely comes into play, because realtime windowing and events push past what the current bindings reach.

Capstone: a 2D game

The second capstone is a small but complete 2D game: a paddle, a ball, a score, a game loop that runs at a steady frame rate, and keyboard control. It is built on [Prima](#) again, because Prima gives us the three things a game needs (a timer for the loop, keyboard events, and fast drawing) in a maintained package that runs today. At the end of the chapter we look honestly at the alternative, a dedicated game library like SDL2, and why reaching it well is still frontier work for `ppperl`'s FFI.

A game is a loop: read input, update the world, draw the world, wait for the next frame, repeat. Everything else is detail. We build that loop first, then hang a game on it.

The game loop: a timer

A realtime loop needs a heartbeat. Prima provides `Prima::Timer`, which fires an `onTick` handler on an interval; setting it to roughly 16 milliseconds gives about 60 frames per second. Each tick updates the world and asks the window to repaint.

```
use strict;
use warnings;
use Prima qw(Application);

my $W = 480;
my $H = 360;

# the world state, plain data
my %game = (
    ball => { x => 240, y => 180, dx => 3, dy => 3, r => 8 },
```

(continues on next page)

(continued from previous page)

```
paddle => { x => 200, w => 80, h => 12 },
score  => 0,
over   => 0,
);
```

The entire game world is a plain hash. That is deliberate: the update logic is ordinary Perl on ordinary data, and the drawing reads the same data. Nothing about the loop is graphics-specific until the draw step.

Update: moving the world

Each tick advances the ball, bounces it off the walls and the paddle, and ends the game if the ball passes the paddle. This is pure arithmetic on the world hash, no drawing:

```
sub update {
    return if $game{over};
    my $b = $game{ball};
    $b->{x} += $b->{dx};
    $b->{y} += $b->{dy};

    # bounce off left, right, and top walls
    $b->{dx} = -$b->{dx} if $b->{x} - $b->{r} < 0 || $b->{x} + $b->
    ↪{r} > $W;
    $b->{dy} = -$b->{dy} if $b->{y} - $b->{r} < 0;

    # paddle collision near the bottom
    my $p = $game{paddle};
    my $paddle_y = $H - 30;
    if ($b->{y} + $b->{r} >= $paddle_y
        && $b->{x} >= $p->{x} && $b->{x} <= $p->{x} + $p->{w}) {
        $b->{dy} = -abs($b->{dy});          # always bounce upward
        $game{score}++;
    }

    # missed the paddle: game over
    $game{over} = 1 if $b->{y} - $b->{r} > $H;
}
```

The collision test is the axis-aligned overlap from the *foundations* coordinate system: the ball has crossed the paddle's row and lies within its horizontal span. Bouncing is negating a velocity component. A whole genre of games is this and not much more.

Draw: rendering the world

Drawing reads the world hash and paints it. Prima calls `onPaint` with a canvas; we clear it, then draw the paddle, the ball, and the score using the primitives from the *GUI chapter*:

```

sub draw {
    my ($canvas) = @_;
    $canvas->color(c1::Black);
    $canvas->bar(0, 0, $W, $H);           # clear to black

    my $p = $game{paddle};
    $canvas->color(c1::White);
    $canvas->bar($p->{x}, $H - 30, $p->{x} + $p->{w}, $H - 30 + $p->
    ↪{h});

    my $b = $game{ball};
    $canvas->color(c1::Yellow);
    $canvas->fill_ellipse($b->{x}, $b->{y}, $b->{r} * 2, $b->{r} *
    ↪2);

    $canvas->color(c1::White);
    $canvas->text_out("Score: $game{score}", 10, $H - 20);
    $canvas->text_out('GAME OVER', $W / 2 - 40, $H / 2) if $game
    ↪{over};
}

```

`text_out` draws a string; `fill_ellipse` and `bar` are the ball and the paddle. Clearing to black every frame and redrawing everything is the immediate-mode style: the screen is a function of the world state, recomputed each frame, which is simple to reason about and fast enough for a game this size.

Input: the keyboard

The paddle follows the arrow keys. Prima delivers key presses through an `onKeyDown` handler, which we use to slide the paddle left and right, clamped to the window:

```

sub on_key {
    my ($self, $code, $key, $mod) = @_;
    my $p = $game{paddle};
    $p->{x} -= 25 if $key == kb::Left;
    $p->{x} += 25 if $key == kb::Right;
    $p->{x} = 0 if $p->{x} < 0;
    $p->{x} = $W - $p->{w} if $p->{x} > $W - $p->{w};
}

```

The `kb::Left` and `kb::Right` constants name the arrow keys. Clamping keeps the paddle on screen. For smooth continuous motion a game often tracks which keys are currently held and moves in the update step; a step-per-press is simpler and fine here.

Wiring it together

The window ties the three handlers to the loop. The timer drives update and repaint; the paint handler draws; the key handler steers:

```

my $window = Prima::MainWindow->new(
    text      => 'pperl bounce',
    size      => [$W, $H],
    onPaint   => sub { my ($self, $canvas) = @_; draw($canvas); },
    onKeyDown => \&on_key,
);

$window->insert(Timer =>
    timeout => 16,                      # ~60 fps
    onTick  => sub {
        update();
        $window->repaint;
    },
)->start;

Prima->run;

```

Assemble the pieces into one file and run it with `ppperl`. A window opens with a bouncing ball and a paddle you steer with the arrow keys; each successful bounce scores a point, and missing ends the game. Every part traces through the guide: the loop is a timer, the world is plain data, the update is arithmetic on coordinates from *foundations*, and the draw step is *Prima primitives*.

The frontier: SDL2 and dedicated game libraries

Prima runs this game well, and for many 2D games it is the right tool. But a serious game usually wants a dedicated library: SDL2, for hardware -accelerated sprites, sound, gamepads, and a tuned event loop. Here the guide has to be honest about where `ppperl` stands.

The Perl SDL story is thin. The old SDL distribution binds SDL 1.2, which is end-of-life, and it is itself effectively unmaintained. There is no solid, maintained SDL2 binding on CPAN; the one FFI-based attempt is alpha-grade, with an incomplete test suite and shaky threading. So SDL2 is exactly the kind of modern C library the *FFI thread* exists to reach, and it is also exactly where the current FFI frontier bites.

Some of SDL2 is reachable with raw `Peta::FFI` calls today. Creating a window and a renderer returns opaque handles (pointers, so integers), and many calls take only scalars:

```

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

my $sdl = dlopen("libSDL2.so");
call($sdl, "SDL_Init", "(L)i", 0x20);          # SDL_INIT_
→VIDEO
my $win = call($sdl, "SDL_CreateWindow", "(piiiiL)l",
    "ppperl", 100, 100, 480, 360, 4);          # returns a
→handle
# ... and so on for the renderer, using the returned handle

```

That much works: scalar arguments, string arguments, and opaque handles returned as integers are precisely what the raw type codes cover. But a real game loop needs the

part that does not yet work. `SDL_PollEvent` fills an `SDL_Event`, which is a **union** of many struct shapes, and the raw type codes cannot describe a struct, let alone a union. Reading which key was pressed means decoding that struct's bytes, which is what the *Level 2 data-description layer* is being built to do. And a fully idiomatic binding would want callbacks, which the raw layer also does not reach.

So the honest position is this: a **complete, comfortable** SDL2 binding on pperl is frontier work, waiting on the FFI's struct and callback support. It is reachable, it is on the roadmap, and it is exactly the kind of binding the *Cairo showcase* pattern will produce once the data-description layer lands. Until then, Prima is the capstone's honest, running answer, and the SDL2 route is a real but partial path clearly marked as frontier.

Where the guide ends

You have now built two programs that run: a drawing application and a game, both on the maintained toolkit stack, with the FFI thread woven in where it earns its place and flagged where the road currently ends. The *final chapter* draws that map explicitly: what a second version would add, and which frontiers, the third dimension chief among them, are deliberately left for the sequel.

Known gaps and what's next

This guide covered two dimensions thoroughly and stopped there on purpose. This closing chapter draws the boundary explicitly: what a second version would add, what belongs in separate guides, and where the FFI frontier currently ends. Naming these keeps you from hunting for coverage that is not here, and sets out the road ahead.

The third dimension (planned v2)

Everything three-dimensional is deferred to a sequel, not omitted by accident. The *foundations chapter* named the one concept that carries across (the Z-buffer, which decides what is nearest the viewer), and the *algorithms chapter* noted that tessellation into triangles is the bridge into 3D, but neither went further. A second version would cover:

- **OpenGL**, through a maintained binding, for hardware-accelerated rendering: the modern shader pipeline, vertex and index buffers, and the math (projection, the model-view transform) that turns a 3D scene into a 2D image.
- **The depth buffer for real**, not just the concept: how draw order stops mattering and depth decides visibility.
- **A 3D game capstone**, the third project the original plan named, built on whatever 3D path proves most honest to run at the time.

3D is a large subject and doing it justice needs its own arc; folding a thin version into this guide would have shortchanged both dimensions.

Deliberately out of scope

Three areas are excluded on purpose and would each be a separate guide, not a chapter here:

- **GPU programming.** Writing compute or graphics shaders and running general-purpose work on the GPU is a distinct discipline with its own toolchain. It touches graphics but is not drawing, and it earns its own treatment.
- **Computer vision and image analysis.** This guide used *convolution* as a drawing tool (blur, sharpen) and stopped precisely at the line where the same math becomes detection and recognition. Feature detection, segmentation, and recognition are analysis, not drawing, and belong elsewhere.
- **Hosted image-generation services.** Connectors to external image-generation APIs are network and service-integration work, not graphics programming, and are out of scope regardless of dimension.

The FFI frontier

The *reaching-further arc* was candid at each step about where pperl's native FFI currently reaches. Collected here, the frontier is:

- **The Level 2 data-description layer.** Raw `Peta::FFI` calls handle scalars, strings, buffers, and opaque handles, which covers a large part of many C graphics APIs and all of a library like *Cairo*. What they do not yet describe is a **struct** passed or returned by value, or a **union** such as a windowing event record. The declarative data-description layer is being built to close exactly this gap, and it is the missing piece between today's raw calls and a comfortable binding for struct-heavy libraries.
- **Callbacks.** A library that calls back into your code (an event loop invoking a handler, a draw signal) needs a C function pointer that re-enters Perl. That is beyond the raw layer today.
- **A complete SDL2 binding.** The *game capstone* showed SDL2 is partly reachable now (window and renderer creation, the scalar calls) and blocked on the two items above for the rest (event unions, and ideally callbacks). It is the natural first big curated graphics binding once the data-description layer lands, following the *Cairo showcase* pattern.

None of these are walls. They are the order in which the FFI is growing, and this guide's job was to tell you which side of the current line each task sits on.

What you can do now

Set against the frontier, the ground already covered is broad. With this guide you can, today, on pperl:

- Draw raster images with primitives and filters, and read, scale, and save them (*Imager*, *GD* and *ImageMagick*).
- Produce resolution-independent vector output, as pixels through Cairo or as markup through SVG (*Cairo* and *SVG*).
- Read and write image metadata, and draw charts (*metadata* and *charts*).
- Open a window, respond to the mouse and keyboard, and animate (*GUI toolkits*).
- Reach a scalar-and-handle C graphics library through a curated native binding, with no C compiler and no build step (*Cairo showcase*).

- Build and run real programs: a *drawing application* and a *2D game*.

That is a complete 2D graphics capability in Perl, from the pixel to a running program. The third dimension, the GPU, and the vision side are the roads out from here, and they are on the map.

4.11 pack and unpack - a tutorial

`pack` and `unpack` are Perl's low-level serialisers. They convert between Perl values and the raw byte strings that binary formats - network protocols, file headers, C structures, fixed-width records - actually carry on the wire.

Most Perl programmers meet them rarely and forget them quickly. This tutorial is here so that when you do meet them, you already know the shape of the problem.

4.11.1 When you need pack and unpack

Reach for them when you are:

- **Speaking a wire protocol** - DNS queries, TCP/IP headers, MQTT frames, anything with “big-endian 16-bit length” in its spec.
- **Parsing a binary file format** - PNG, GIF, WAV, ELF, GZip, Intel HEX, all the formats whose first four bytes are a magic number followed by well-known integer fields.
- **Slicing fixed-width text records** - log files, accounting ledgers, reports from mainframes. `unpack` is cleaner than chained `substr` calls and handles blank-field ambiguity that `split` collapses.
- **Passing data to a system call** - `syscall`, `ioctl`, and the rare `sockaddr_*` constructor expect bytes laid out as a C struct.

For text serialisation (JSON, YAML, CSV, XML), reach for the appropriate module instead. `pack` is for binary.

4.11.2 How this tutorial is organised

Each chapter teaches one topic and stands on its own. You can read them in order, or jump to the one that answers your question.

Bytes and widths

By the end of this chapter you will be able to pick the right integer directive for a given field, count the bytes a template produces, and tell when you need a fixed width versus a native width.

Binary formats are specified by *byte counts*. “A 16-bit unsigned length field followed by 4-byte packets” is an instruction about bytes, not about numbers. The first job of a `pack` template is to match those counts exactly.

The fixed-width integer directives

The spine of every template:

Bytes	Unsigned	Signed
1	C	c
2	S	s
4	L	l
8	Q	q

Each directive consumes one value from the list and produces exactly that many bytes:

```
length pack "C", 65 # 1
length pack "S", 12345 # 2
length pack "L", 1_000_000_000 # 4
length pack "Q", 1_000_000_000_000 # 8 (if 64-bit Perl)
```

A repeat count packs that many of the same directive in a row:

```
length pack "C4", 1, 2, 3, 4 # 4
length pack "L3", 10, 20, 30 # 12
```

Compose them freely; the total length is the sum:

```
length pack "C S L", 65, 1000, 2_000_000 # 1 + 2 + 4 = 7
```

C is your default one-byte directive

For protocol bytes - flag fields, version numbers, opcodes, type tags - C (unsigned 8-bit, 0 to 255) is almost always the right choice. c (signed, -128 to 127) turns up only when the spec says “signed”. W is a variant of C that allows values above 255 when your template is in U0 (UTF-8) mode - ignore it until you hit that case.

IP addresses are the textbook example:

```
my $raw = pack "C4", 192, 168, 1, 42;
#      ^-- four unsigned bytes, in the order the list gives them
```

Native vs. fixed width

The directives s / S / l / L are specified to be exactly 2 and 4 bytes regardless of the C compiler. Their native-width counterparts add !:

```
length pack "l", 0 # 4 (always)
length pack "l!", 0 # 4 on most 32-bit and 64-bit systems,
# 8 on 64-bit Alpha, some legacy Unixes.
```

i and I are *always* native (whatever sizeof(int) returns on this machine, with a 32-bit minimum). i! is an alias for i.

Rule of thumb:

- For interoperable binary (wire formats, on-disk files, data crossing machines) use **fixed width** (l, L, ...) plus an explicit endianness - or the *portable shortcuts*.
- For in-process work that matches some local C struct you are about to hand to `ioctl` or `syscall`, use **native width** (l!, L!, i, I).

Mixing the two is fine, but stop and ask yourself why each time.

Counting the length of a template

Before you send a record, check the byte count is what the spec says. `length pack(TEMPLATE, ...)` with placeholder values is the simplest way:

```
my $sz = length pack "n N L C", 0, 0, 0, 0;  
# 2 + 4 + 4 + 1 = 11
```

For templates whose length depends on native sizes, `length pack TEMPLATE, 0, 0, ...` gives you the answer on the current machine.

Worked example: a minimal packet header

A hypothetical protocol specifies:

- 8-bit version
- 8-bit flags
- 16-bit big-endian length (payload follows)

Three fields, four bytes total. The template writes itself:

```
my $hdr = pack "C C n", $version, $flags, length $payload;  
my $pkt = $hdr . $payload;  
  
length $hdr      # 4
```

Two things to note:

1. We did not spell out `C1 C1 n1` - a bare directive letter has an implicit repeat count of 1.
2. The length in the third field is computed from `$payload` in the same expression. A later chapter shows the */form* that ties the two together so you cannot forget.

Edge cases worth remembering

- **Values out of range:** `pack "C"`, 256 silently truncates (you get `\x00`). There is no warning. Check ranges yourself if the data is untrusted.
- **Signed vs. unsigned round-trip:** a value packed as `C` and unpacked as `c` (or vice versa) flips sign for values ≥ 128 :

```
unpack "c", pack "C", 200      # -56
```

- **Too few values:** missing values pack as 0 for numeric directives, "" for strings. No warning.

The next chapter handles the question every multi-byte integer raises: which end of the number goes first?

Endianness

By the end of this chapter you will be able to choose between native, big-endian, and little-endian byte order; use the portable `n / N / v / V` shortcuts; and apply the `< / >` modifiers to any integer or float.

A multi-byte integer has to be written somehow. `0x12345678` is four bytes - but in which order? Two conventions dominate:

- **Big-endian** - highest-order byte first: 12 34 56 78. Used by most network protocols, by PowerPC and SPARC CPUs, and by file formats that care about portability (PNG, Java `.class`, most TCP/IP headers).
- **Little-endian** - lowest-order byte first: 78 56 34 12. Used by x86, x86-64, ARM (in its common modes), and by file formats from the Intel lineage (BMP, WAV, TIFF in its common variant, all of FAT).

Mix these up and you will read nonsense. `pack` gives you three ways to pin down the byte order.

The portable shortcuts: `n / N / v / V`

Four letters cover the most common wire-format cases:

Directive	Bytes	Endian	Meaning
<code>n</code>	2	big-endian	Network (port numbers)
<code>N</code>	4	big-endian	Network (IPv4 addrs)
<code>v</code>	2	little-endian	"VAX"
<code>V</code>	4	little-endian	"VAX"

"Network byte order" is big-endian. If you are reading an RFC, the answer is almost always `n` or `N`.

```
my $port = pack "n", 443;           # "\x01\xbb"
my $ip    = pack "N", 0x7F000001;    # "\x7f\x00\x00\x01" - 127.0.0.1
```

These four are unsigned only. With the `!` modifier they become signed:

```
my $neg = pack "n!", -1;             # "\xff\xff"
```

If your field fits one of these four shapes, use them. They are portable, they are readable, and their meaning does not depend on the CPU running the code.

The general modifiers: < and >

Any integer directive (s, S, l, L, i, I, q, Q, j, J) and every float (f, d, F, D) accepts an endianness modifier:

Modifier	Meaning	Mnemonic
>	Big-endian	The “big end” touches the directive letter
<	Little-endian	The “little end” touches the directive

Read l> as “long, big end” and l< as “long, little end”.

```
my $be32 = pack "l>", -1_000_000;    # big-endian 32-bit signed
my $le64 = pack "q<", 2 ** 40;      # little-endian 64-bit signed
my $bed  = pack "d>", 3.14159;      # big-endian IEEE 754 double
```

For the common 16- and 32-bit unsigned cases, n / N / v / V are shorter and should be preferred. Use < / > when:

- the field is signed - n and N are unsigned by default
- the field is 64-bit - there are no n/N equivalents for q/Q
- the field is a float - n/N cover integers only
- you want to apply endianness to a whole group (see below)

Applying endianness to a group

If every integer in a sub-structure shares the same byte order, write the endianness modifier once on a group:

```
my $rec = pack "(s l l)<", $flags, $x, $y;
# same as "s<l<l<"
```

A group-level modifier cascades into every byte-ordered directive inside, including nested groups. Directives that do not accept a byte-order modifier (such as C) are silently ignored.

Native order: when it *is* what you want

The bare directives s / S / l / L / i / I / q / Q produce the host CPU’s native byte order. That is useful when:

- You are building a C struct in memory for an ioctl call on the *same* machine.
- You are writing a scratch file that only this program will read back.
- The protocol spec explicitly says “host byte order” (rare).

It is **wrong** when data leaves the process: a file written on a laptop, read on a server, must not use native order unless both machines happen to agree.

Detecting the host's byte order

Useful once, as a sanity check. Pack 1 as a 16-bit integer and look at the first byte:

```
my $little_endian = unpack("c", pack("s", 1)) == 1;
my $big_endian    = unpack("c", pack("s", 1)) == 0;
```

On x86 the first is true; on a big-endian host the second is true.

Floats have endianness too

IEEE 754 does not pin down byte order. A double written on a little-endian machine comes out reversed on a big-endian one:

```
# send a float in big-endian order - safe across machines that use
→IEEE 754
my $buf = pack "d>", 2.718281828;
```

Mismatched hardware (non-IEEE float formats, exotic long double layouts) cannot be repaired by `</>` alone. For portability across odd platforms, send floats as text or use a byte-exact format like the output of `sprintf "%a"`.

Worked example: parsing a BMP header

The BMP bitmap format begins with the ASCII magic BM followed by three little-endian 32-bit fields (file size, reserved, pixel data offset):

```
my ($magic, $filesize, $reserved, $pixoff) =
    unpack "A2 V V V", $bmp_header;

die "not a BMP" unless $magic eq "BM";
```

V is little-endian 32-bit unsigned - exactly what the spec demands. A2 reads two bytes as a text string without stripping. Every field is the right shape; no byte-swapping required, no platform-dependent behaviour.

What to remember

- **Wire format?** Use `n / N / v / V` - or `< / >` if you need signed, 64-bit, or float.
- **Stay inside one process?** Native is fine - but flag the choice deliberately.
- **IEEE 754 is not enough.** If a float crosses a machine, pin the byte order.

With byte counts and byte order under control, the remaining directives - strings, positioning, groups - fall out naturally.

Strings, bits, and nybbles

By the end of this chapter you will be able to pack and unpack fixed-width strings, C-style NUL-terminated strings, bit strings, and hex strings - the three text-ish families of directives.

Binary formats carry string data in several distinct flavours: raw bytes, space-padded text, NUL-terminated C strings, bit fields, hex-encoded numbers. `pack` has a directive for each.

The three string directives: `a` / `A` / `Z`

All three pack exactly one value into a fixed width. They differ in what they pad with, and - crucially - in what `unpack` strips back off:

Letter	Pad byte	Unpack returns	Typical use
<code>a</code>	<code>"\0"</code> (NUL)	All bytes unchanged	Arbitrary binary
<code>A</code>	<code>" "</code> (space)	Trailing whitespace and NUL stripped	ASCII fixed-width text
<code>Z</code>	<code>"\0"</code> (NUL)	Bytes up to the first NUL	C-style NUL-terminated

The width is the **repeat count**, not a count of values:

```
pack "a4", "hi"           # "hi\0\0"
pack "A4", "hi"           # "hi  "
pack "Z4", "hi"           # "hi\0\0"

pack "a4", "abcdef"       # "abcd" - truncated
pack "A*", "hello"        # "hello" - whatever the value's length is
```

`Z` guarantees a trailing NUL - with a caveat

`Z` always reserves space for at least one terminating NUL:

```
pack "Z*", "hello"        # "hello\0" - a free NUL appended
pack "Z5", "hello"        # "hell\0" - truncated to make room!
```

If a C program on the other side expects a zero-terminated `char[32]`, use `Z32`. It will pack up to 31 bytes of data plus the terminator.

Picking between `a` and `A` for text

- The data is ASCII, padded with spaces on disk → `A`.
- The data is arbitrary bytes (might contain spaces or NULs as valid content) → `a`.

The classic trap: you pack a name with `A20` and the wire format uses NUL padding. `unpack "A20"` strips both spaces *and* NULs, so it looks fine - but `pack "A20", "bob"` will round-trip to `"bob "` padded with spaces. If the spec says NUL padding, use `a` on the pack side.

Worked example: fixed-width text records

A ledger file lays data out in columns:

0	1	2	3	4
0123456789012345678901234567890123456789012345678				
2026-04-22	coffee at the station			3.50
2026-04-23	train to Brussels			42.00

Column 1-10 is the date, 12-38 the description, 40-47 the amount. The gaps are single spaces (byte 11 and byte 39). `unpack` with `A10 x A27 x A*` peels each record apart:

```
while (<$ledger>) {
    chomp;
    my ($date, $desc, $amount) = unpack "A10 x A27 x A*", $_;
    print "  $date | $desc | $amount\n";
}
```

`x` skips one byte; `A27` eats the description column and strips trailing spaces; `A*` greedy-eats the remaining bytes as the amount. The output side uses wider fields so spaces survive the round-trip:

```
my $line = pack "A11 A28 A8 A*", $date, $desc,
               sprintf("%.2f", $amt_left),
               sprintf("%12.2f", $amt_right);
```

Notice the extra byte on each `A` width - that single extra byte is the column gap. A consistent byte budget per column is what makes fixed-width records tractable at all.

Bit strings: `b` and `B`

A bit string is a string of `"0"` and `"1"` characters that packs into actual bits. Two directives, differing only in which direction you read each byte:

Directive	Bit order within each byte
<code>b</code>	LSB first - bit 0, 1, 2, 3, 4, 5, 6, 7
<code>B</code>	MSB first - bit 7, 6, 5, 4, 3, 2, 1, 0

The repeat count is the number of **bits**, not bytes:

```
pack "B8", "10001100"      # "\x8c" - MSB first, bit 7 set, bit 3 &
                             ↪ 2 set
pack "b8", "00110001"      # "\x8c" - LSB first, same byte
```

`B` matches the usual “left-to-right is high-to-low” convention you see in binary dumps; `b` matches the convention used by some hardware registers and the `vec` built-in. Use `B` when you are writing out a “normal” binary number, `b` when you are mirroring a data sheet that numbers bit 0 on the right.

Counting set bits

One of the more surprising uses of `unpack`: count set bits in a buffer with a single call. `%32b*` unpacks the bits and asks for a 32-bit sum, which is the count:

```
my $n_bits = unpack "%32b*", $mask;
```

The `%N` prefix is `unpack`-only - see the [positioning chapter](#) for the other uses.

Hex strings: `h` and `H`

Hex strings are the text representation most developers read - two hex digits per byte. Two directives, differing in nybble order:

Directive	Nybble order
<code>h</code>	Low nybble first
<code>H</code>	High nybble first

`H` is the “normal” hex dump - read left to right, high digit then low:

```
pack "H*", "deadbeef"      # "\xde\xad\xbe\xef"
unpack "H*", "\xde\xad\xbe\xef" # "deadbeef"
```

`h` reverses each byte’s nybbles - rarely what you want unless a spec explicitly says so. If the round-trip you expect is *not* what you get, try the other letter.

```
pack "h*", "ef"            # "\xfe" (nybbles swapped)
pack "H*", "ef"            # "\xef"
```

Choosing between them

You have	Reach for
ASCII in a fixed-width column on disk	<code>A</code>
Arbitrary bytes in a fixed-width slot	<code>a</code>
A C-style NUL-terminated string	<code>Z</code>
A binary number as a string of 0 / 1	<code>B</code>
Same, with the register-convention bit order	<code>b</code>
A hex string in the normal reading order	<code>H</code>
A hex string with nybbles swapped	<code>h</code>

Bits, nybbles, and string directives all share one rule: the repeat count specifies the width of the *single* value being packed, not the number of values. That’s the one fact to carry forward.

Next up: `()` groups and repeat counts, the tools for applying a pattern of directives to a list of values of unknown length.

Grouping and counts

By the end of this chapter you will be able to repeat a pattern of directives across a list of values, use the `*` and `[...]` forms of repeat count, and write self-describing length-prefixed records with the `/` directive.

A bare repeat count applies to *one* directive letter: `C4` packs four bytes. The moment the repeating unit is more than one directive

- “pack a short and two bytes, many times” - you need a group.

() - a group is a sub-template

Parentheses gather a sequence of directives so that a repeat count or endianness modifier applies to the whole. Compare:

```
pack "C S C S C S", @a, @b, @c, @d, @e, @f    # repeat by hand
pack "(CS)3",       @a, @b, @c, @d, @e, @f    # same thing, grouped
pack "(CS)*",       @pairs                    # repeat as often as
↪values last
```

A group has no byte cost of its own - it is a syntactic device. The total packed length is what the directives inside it would produce without the parentheses.

Grouping with endianness

The single most practical use of a group - a sub-structure whose every integer shares one byte order:

```
my $rec = pack "(l s s)<", $id, $x, $y;
# same as "l<s<s<"
```

The `<` cascades into every byte-ordered directive inside, including nested groups. Directives that do not accept a byte-order modifier (like `C`, `a`, `Z`) are silently unaffected.

Repeat counts in full

After any directive or group, you may write:

Form	Meaning
<code>N</code>	Apply the directive/group <code>N</code> times
<code>*</code>	Apply as often as values last. For <code>x</code> , <code>X</code> , <code>@</code> : equivalent to 0. For <code>u</code> : 45.
<code>[N]</code>	Same as <code>N</code>
<code>[templ]</code>	Repeat count is the packed-byte length of the bracketed template

The `[template]` form is the tool for expressing “as many bytes as a *foo* takes”:

```
pack "x[L]"      # skip 4 bytes (sizeof a packed long)
pack "x[d]"      # skip 8 bytes (sizeof a packed double)
pack "a[Q]"      # one string 8 bytes wide
```

It is especially useful for alignment (see the [positioning chapter](#)) and for templates whose widths must track the platform-dependent size of a native integer:

```
pack "a[l!]", $native_long_buf      # room for one native long
```

*** is per-value-group, not “swallow everything”**

A single * counts “remaining values” *for this directive*. Two A*s in a row do not race:

```
pack "A*A*", "hello", "world"      # "helloworld"
```

The first A* packs all of "hello"; the second packs all of "world". Each * consumes *one* value from the list, at that value’s full length. This is the general rule: **every directive corresponds to one piece of data**, whatever its repeat count.

The / directive - length and data together

Wire formats frequently store a count immediately before the thing being counted: “a 2-byte length, then that many bytes of payload”. The / directive ties the two into one step.

In pack: *length-item/sequence-item*

Write *two* directives separated by a slash. The first packs the length; the second packs the payload. pack computes the length for you:

```
my $msg = pack "n/a*", "hello, world";  
# "\x00\x0chello, world"  
# ^^^^^^^ big-endian 16-bit length = 12  
#          ^^^^^^^^^ the payload itself
```

The *length-item* may be any numeric directive - C, n, N, w, S<, and so on - or a string directive such as A4 when the protocol writes the length as ASCII:

```
my $buf = pack "A4/A*", "Humpty-Dumpty";  
# "13 Humpty-Dumpty" - 4-char ASCII length, then the string
```

In unpack: */item*

The unpack form is simpler: a bare / before the item. The count is taken from the most recent integer directive:

```
my ($payload) = unpack "n/a*", $msg;      # "hello, world"
```

Reading that template: “read a n integer, call it L; then read L bytes as an a*-style string.” The length itself does not appear in the output list.

The common mistake: A* after a /

You cannot put another A* or a* after a /-introduced field in unpack and expect it to behave - the * is greedy:

```
# Wrong - $prio will be undef, $sm gets everything left
my ($src, $dst, $sm, $prio) = unpack "Z* Z* C A* C", $buf;

# Right - use /A* so $sm knows where to stop
my ($src, $dst, $sm, $prio) = unpack "Z* Z* C/A* C", $buf;
```

In the second template, C/A* reads a byte-count then that many bytes. Everything after the slash respects it, and the trailing C picks up the next byte as expected.

Worked example: key-value pairs

A protocol stores a dictionary as count followed by count pairs of (length, key, length, value):

```
my %env = ( HOST => "example.com",
            PORT => "443",
            USER => "alice" );

my $blob = pack "S (S/A* S/A*)*",
               scalar keys %env,
               %env;
```

Reading it back:

```
my %parsed = unpack "S/(S/A* S/A*)", $blob;
```

The pack template says: “one 16-bit count (of pairs), then repeat the sub-template S/A* S/A* for each pair.” The unpack template reads the count and applies it to the group directly - the count is no longer in the output list.

Edge cases and constraints

- **/ makes no sense with a fixed-length item.** The second directive must be variable-width: a*, A*, Z*, /A\$n, or the like. Perl will reject a fixed-length second item.
- **()* with pack cannot be matched by()* on unpack.** Pack has the values, so it can say “repeat until done”. Unpack does not know how many repetitions are encoded in the buffer unless a count directive precedes the group.
- **Nested groups** are legitimate and common:

```
pack "((CC)(S))<", @records
```

Endianness modifiers cascade through every nesting level.

- **Repeat count on a group** applies to the whole group and takes that many repetitions’ worth of values. (CS)3 consumes six list values, not three.

Next chapter: the directives that move around inside a template without producing a value - x, X, @, ..

Positioning and padding

By the end of this chapter you will be able to skip bytes, insert padding, jump to absolute offsets, and align fields to arbitrary boundaries - the four directives x, X, @, and ..

Some directives produce a byte without consuming a value from the list (pad bytes). Some consume a value without producing a byte (positioning commands). Taken together they let you model the gaps and alignments that real binary formats contain.

x - skip forward / insert NUL

x is the simplest of the four. In pack it inserts one NUL byte without consuming a value; in unpack it skips one byte without producing a value.

```
pack    "C x C", 65, 66      # "A\0B"
unpack  "C x C", "A\0B"     # (65, 66)
```

With a repeat count, it inserts or skips that many bytes:

```
pack "C x3 C", 65, 66      # "A\0\0\0B"
```

x* means “skip to the end of the string” in unpack, and is equivalent to x0 (no-op) in pack.

x does not mean “a space”

The confusing thing about x is that it pads with NUL, not with space. If you want space padding for text, use A with a wider repeat count instead:

```
# Wrong - Perl inserts a NUL, not a space
pack "A10 x A10", "date", "label"
# "date      \0label    "

# Right - widen the field
pack "A11 A10", "date", "label"
# "date      label     "
```

X - back up

X moves the write/read head backward by one byte. With a repeat count, backward by that many bytes. Unlike x, X has no effect on the output bytes of pack - it just rewinds so a subsequent directive overwrites them.

```
pack "CC X C", 65, 66, 67   # "AC" - the second C overwrites 66
↪with 67
```

X is mostly useful with unpack: peek forward, then rewind to re-read:

```
# Read a 16-bit length, then re-read the same 2 bytes as two
↳separate bytes
my ($len, $hi, $lo) = unpack "n XX CC", $frame;
```

X before the start of the string is a fatal error.

@ - jump to absolute position

@N sets the current position to byte N **within the innermost () group** (or from the start of the string if there is no enclosing group). In pack, this zero-fills from the current position forward, or truncates if the current position is past N:

```
pack "C @4 C", 65, 66      # "A\0\0\0B" - byte 0 is 'A', byte 4
↳is 'B'
```

This is the template-level equivalent of `offsetof(struct, field)`: if a C header file tells you field 2 lives at offset 12, @12 parks you there regardless of what the preceding directives did.

@ inside a group

Within each repetition of a group, @ restarts at 0. That can surprise you:

```
pack '@1A(@2A@3A)', qw(X Y Z)
# "\0X\0\0YZ"
```

Walking through: the outer @1A puts "X" at byte 1 (with a NUL at byte 0). The inner group starts a new coordinate system at "X"'s position; inside it, @2A puts "Y" at the group's byte 2 (two further NULs in between), then @3A puts "Z" at byte 3 of the group. Read the spec carefully before using @ inside groups.

. - absolute position from the data

. is the value-driven cousin of @. Instead of a fixed repeat count, the position comes from the next value in the list:

```
pack "C . C", 65, 4, 66    # "A\0\0\0B" - position 4 from the next
↳value
```

Useful when the offset is computed at runtime - say, from a header field you just read:

```
my $buf = pack "C .N C", $type, $offset, $value;
# $offset bytes of NUL fill before $value
```

In unpack, . returns the current position rather than zero-filling:

```
my ($a, $pos) = unpack "C .", "hello";
# $a = 104 ('h'), $pos = 1
```

With .*, the offset is measured from the start of the whole string; with .N for integer N ≥ 1, from the start of the enclosing N-th group; with .0, relative to the current position.

The ! modifier: alignment

`x!N` and `X!N` turn `x` and `X` into alignment commands: advance (or rewind) to the nearest multiple of `N`. This is how you respect C struct padding rules.

```
# struct { char c; double d; char cc[2]; }  
my $s = pack "c x![d] d c2", $c, $d, $c1, $c2;
```

Reading `x![d]`: “align forward to a multiple of the width a packed `d` would take” - i.e. 8 bytes. That is exactly the rule most C compilers apply to a double field after a char. Use `[!]`, `[d]`, `[Q]` to name the alignment requirement without having to know the numeric width.

A bare `x!0` or `x!1` is a no-op.

Checksums - %N on unpack only

`%N` is a prefix on a single directive, not a standalone directive. It says “instead of returning the values the next directive would produce, return their `N`-bit sum”. Common uses:

```
my $bytesum = unpack "%32C*", $buf;    # sum every byte  
my $setbits = unpack "%32b*", $mask;   # count set bits
```

Exists only in `unpack`. There is no `%` in `pack` - you cannot pack a checksum directly; compute it yourself and pack the result with whatever integer directive you need.

Worked example: reading a variable-offset payload

Some formats start with a header of fixed size but put the actual data at an offset the header contains. The classic recipe:

```
# Header: magic (4 bytes), version (1 byte), data-offset (4 bytes  
→BE),  
# then data starts at the byte-offset the header pointed at.  
my ($magic, $ver, $off) = unpack "A4 C N", $file;  
  
die "bad magic" unless $magic eq "MAGI";  
  
# Jump to the data section  
my $data = unpack "x$off a*", $file;
```

Two separate `unpack` calls - the first reads the offset, the second uses it in the template. Template strings are not regular expressions; they have no way to look ahead and use a computed value within the same call. `Unpack` in stages when the shape of the data depends on what you have just read.

With positioning and alignment in the toolbox, the two worked examples in the next chapters should read as straightforward applications: parsing a real network header and a real file format.

Network protocols - a DNS query

By the end of this chapter you will be able to build and parse a DNS query header and question section using `pack` and `unpack` - and take the same approach to any RFC-defined protocol.

A “network protocol” is, in the end, a sequence of byte-exact fields described in English prose. If you can read the field table, you can write the template. This chapter walks one concrete example - the DNS query packet - from the RFC wire diagram to a fully working pair of `encode` / `decode` subroutines.

The problem

We want to ask a DNS server for the A record of `example.com`. The UDP payload we send is a **DNS message** (RFC 1035), consisting of:

1. A 12-byte **header**.
2. A **question section** - the domain we are asking about, and the record type we want.

(A real client also parses the **answer section** in the response. We stop at encoding and showing a sketch of the decode.)

The header

From RFC 1035 section 4.1.1, the header is six consecutive 16-bit big-endian fields:

Offset	Field	Meaning
0	ID	Arbitrary 16-bit identifier we choose
2	Flags	Opcode, RD bit, response flags, rcode
4	QDCOUNT	Number of entries in question section
6	ANCOUNT	Number of resource records in answer
8	NSCOUNT	Authority records
10	ARCOUNT	Additional records

Six unsigned big-endian shorts means six `n` directives:

```
sub dns_header {
    my (%opts) = @_;
    pack "n6",
        $opts{id}      // 0,
        $opts{flags}   // 0,
        $opts{qd}       // 0,
        $opts{an}       // 0,
        $opts{ns}       // 0,
        $opts{ar}       // 0;
}
```

That is 12 bytes, exactly as the spec requires. `n6` is shorthand for `n n n n n n`.

The question section

A question is:

1. A **name**, encoded as a sequence of length-prefixed labels terminated by a zero-length label.
2. A 16-bit **QTYPE** (1 = A record).
3. A 16-bit **QCLASS** (1 = IN, Internet).

For example `.com` the name encodes as:

```
\x07 e x a m p l e   \x03 c o m   \x00
```

Each label begins with a length byte; the whole name ends with a zero-length label. Two things to notice: the length is one byte (C), not two; and there is a trailing NUL but it is *not* exactly what `Z` produces - it terminates the list of labels, not a single string.

We can build this by hand from a domain name:

```
sub encode_name {
    my ($name) = @_;
    my $out = "";
    for my $label (split /\./, $name) {
        die "label too long" if length($label) > 63;
        $out .= pack "C/a*", $label;
    }
    $out .= "\0";           # zero-length terminator
    return $out;
}
```

Two directives in the loop body: `C/a*` - a single-byte length followed by the label bytes, computed automatically (see the [grouping-and-counts chapter](#) for the `/` form). After the loop, a literal `"\0"` terminates the list.

With `encode_name` in place, the whole question section is two concatenations and one pack:

```
sub dns_question {
    my ($name, $qtype, $qclass) = @_;
    return encode_name($name) . pack "n n", $qtype, $qclass;
}
```

Assembling the packet

```
use constant {
    QR_QUERY      => 0,
    OPCODE_QUERY => 0,
    RD            => 1 << 8,           # recursion desired
    TYPE_A        => 1,
    CLASS_IN      => 1,
};
```

(continues on next page)

(continued from previous page)

```

sub dns_query_for_A {
    my ($name) = @_;
    my $id      = int rand 65536;
    my $flags   = RD;                      # standard query, recursion_
    ↪desired

    my $pkt = dns_header(
        id      => $id,
        flags   => $flags,
        qd      => 1,                      # one question
    );
    $pkt .= dns_question($name, TYPE_A, CLASS_IN);

    return ($id, $pkt);
}

my ($id, $pkt) = dns_query_for_A("example.com");
# send $pkt over a UDP socket to port 53

```

29 bytes total: 12 header + 13 for the name (`\x07example\x03com\x00` is 13) + 2 + 2 for QTYPE and QCLASS. Run `length $pkt` to confirm.

Parsing the response header

The response has the same header shape - only the flag bits change. Decoding is the reverse template:

```

sub parse_dns_header {
    my ($buf) = @_;
    my ($id, $flags, $qd, $an, $ns, $ar) = unpack "n6", $buf;

    my %hdr = (
        id      => $id,
        qr      => ($flags >> 15) & 1,
        op      => ($flags >> 11) & 0x0f,
        aa      => ($flags >> 10) & 1,
        tc      => ($flags >> 9) & 1,
        rd      => ($flags >> 8) & 1,
        ra      => ($flags >> 7) & 1,
        rcode   => $flags & 0x0f,
        qd      => $qd,
        an      => $an,
        ns      => $ns,
        ar      => $ar,
    );
    return \%hdr;
}

```

`n6` pulls the six shorts out at one go; the individual flag bits come out of shifts and masks

on `$flags`. This pattern is universal: use `pack` / `unpack` for the byte-level layout, plain Perl for bit-level decoding of the flag fields.

Parsing a name

Names in the answer section use the same length-prefix encoding, plus a **pointer compression** mechanism we will skip. For a fresh name (no compression) the decoder mirrors the encoder:

```
sub decode_name {
    my ($buf, $offset) = @_;
    my @labels;
    while (1) {
        my $len = unpack "x$offset C", $buf;
        last if $len == 0;
        die "compression pointer" if $len >= 0xc0;
        my $label = unpack "x${\ ($offset + 1)} a$len", $buf;
        push @labels, $label;
        $offset += 1 + $len;
    }
    return (join(".", @labels), $offset + 1);
}
```

Three `unpack` calls, each with an `x$offset` prefix to skip to the right byte. The name alternates between one-byte lengths and variable-width labels, and the loop keeps reading until it hits a zero-length label. The `x$offset a$len` trick - compose a template from Perl values, call `unpack` - is the standard pattern when the next field's width depends on the previous field's value.

What to carry forward

- **An RFC byte-layout table maps directly to a template string.** Big-endian shorts → `n`, big-endian longs → `N`, bytes → `C`.
- **Length-prefixed fields use `C/a*` / `n/a*` / ...** - one expression, no off-by-one counting by hand.
- **Bit-level fields inside a byte get decoded in plain Perl** after `unpack` has handed you the byte or short that contains them.
- **Variable-width data requires staged unpacking.** Read a length, then build the template for the data from that length, then call `unpack` again.

The next chapter applies the same approach to a real file format - a GIF image header - and introduces the “magic number check” idiom.

File formats - a GIF header

By the end of this chapter you will be able to identify a binary file by its magic number, parse its fixed-layout header, and decode bit-packed flag bytes. The example is the GIF image format, but the approach transfers directly to PNG, WAV, BMP, ELF, ZIP, and every other format built around a fixed header.

The GIF header and logical screen descriptor

A GIF file starts with two fixed-size blocks, thirteen bytes in total. From the GIF89a specification:

Header (6 bytes)

Offset	Bytes	Field
0	3	Signature - "GIF"
3	3	Version - "87a" or "89a"

Logical Screen Descriptor (7 bytes)

Offset	Bytes	Field	Byte order
6	2	Logical screen width	little-endian
8	2	Logical screen height	little-endian
10	1	Packed flags	-
11	1	Background colour index	-
12	1	Pixel aspect ratio	-

The packed flags byte has five sub-fields:

Bit:	7	6	5	4	3	2	1	0
	[GCT]	[CR	]	[Sort]	[GCT	size]		

- bit 7 - global colour table flag
- bits 6-4 - colour resolution (minus 1)
- bit 3 - sort flag
- bits 2-0 - global colour table size (actual size = $2^{(\text{value}+1)}$)

Every format specification looks roughly like this: fixed-width fields in a known order, with the occasional packed-bits byte.

Reading it in

The template writes itself field by field:

```
sub parse_gif_header {
    my ($fh) = @_;
    binmode $fh;

    my $header;
    read $fh, $header, 13
        or die "short read: $!";
}
```

(continues on next page)

(continued from previous page)

```
my ($sig, $ver, $w, $h, $flags, $bg, $aspect) =  
    unpack "A3 A3 v v C C C", $header;  
  
die "not a GIF" unless $sig eq "GIF";  
die "unknown version: $ver" unless $ver eq "87a" or $ver eq "89a"  
↪ ;  
  
return {  
    version => $ver,  
    width   => $w,  
    height  => $h,  
    flags   => $flags,  
    bg      => $bg,  
    aspect  => $aspect,  
};  
}
```

Directive by directive:

- A3 A3 - six bytes of ASCII, split into signature and version. A (not a or Z) because the spec guarantees exactly three characters with no padding - A returns them unchanged and has the safer strip-on-unpack behaviour.
- v v - two little-endian 16-bit unsigned integers. The GIF format is little-endian throughout; v is the exact match.
- C C C - three unsigned bytes.

Total width: 3 + 3 + 2 + 2 + 1 + 1 + 1 = 13 bytes. We read exactly that many before unpacking; any less is an error.

The magic-number check

Two lines above illustrate a pattern you will repeat for every file format:

```
die "not a GIF" unless $sig eq "GIF";  
die "unknown version: $ver" unless $ver eq "87a" or $ver eq "89a";
```

Check the magic first, unpack the rest second. Reversing the order lets nonsense data through into later stages where the error messages are less clear. For formats with a longer magic (PNG is 8 bytes, ELF is 4, WAV's "RIFF...WAVE" sandwich is 12), the same idea applies: unpack enough to identify the file, abort on mismatch, then unpack the rest.

Decoding the packed flags byte

pack and unpack cannot decode sub-byte bit fields directly (with the exception of the string-shaped b / B directives, see the [strings chapter](#)). The pragmatic approach is shifts and masks in plain Perl on the byte unpack returned:

```

sub decode_gif_flags {
    my ($flags) = @_;
    return {
        gct_present => ($flags >> 7) & 0x01,
        color_res   => (($flags >> 4) & 0x07) + 1,
        sort_flag   => ($flags >> 3) & 0x01,
        gct_size    => 2 ** ((($flags) & 0x07) + 1),
    };
}

```

The pattern - “unpack the byte, pick the bits apart with shifts”

- is so common it deserves to be named. Anything smaller than a byte belongs outside the template.

Worked run

```

open my $fh, "<:raw", "sample.gif" or die $!;
my $hdr = parse_gif_header($fh);
my $gct = decode_gif_flags($hdr->{flags});

printf "GIF%s %dx%d\n", $hdr->{version}, $hdr->{width}, $hdr->
    ↳{height};
printf "global colour table: %s (%d entries)\n",
    $gct->{gct_present} ? "yes" : "no",
    $gct->{gct_size};

```

On a real GIF89a file of dimensions 640×480, this prints:

```

GIF89a 640x480
global colour table: yes (256 entries)

```

Writing a header

The inverse is shorter still - every directive we unpacked, we can pack. One detail: the flags byte has to be re-assembled from its sub-fields first.

```

sub encode_gif_flags {
    my (%f) = @_;
    return (((f{gct_present} & 0x01) << 7)
        | (((log2(f{color_res})) & 0x07) << 4)
        | ((f{sort_flag} & 0x01) << 3)
        | (log2(f{gct_size}) - 1));
}

sub write_gif_header {
    my ($fh, $hdr, $gct) = @_;
    my $flags = encode_gif_flags(%$gct);
    my $bytes = pack "A3 A3 v v C C C",
        "GIF", $hdr->{version},

```

(continues on next page)

(continued from previous page)

```
$hdr->{width}, $hdr->{height},  
$flags, $hdr->{bg}, $hdr->{aspect};  
print {$fh} $bytes;  
}
```

Same template on the pack side, same 13 bytes out, matches the GIF reader on the other end. Round-tripping with a test vector is the quickest way to gain confidence you have the template right: unpack an existing file, re-pack the result, compare byte-for-byte.

Scaling up to larger formats

The approach is compositional. A format like WAV has multiple chunks, each with its own header (id + size + data). You write:

- A low-level “read one chunk header” helper - two unpack calls on 8 bytes: 4-byte FourCC and 4-byte little-endian size.
- A dispatcher that looks at the FourCC and calls a per-chunk parser.
- Per-chunk parsers that unpack the chunk body according to that chunk’s own byte layout.

Every layer is a small unpack against a tiny template. The moment a field’s width depends on a value you just read, you stop, compose the new template from that value, and call unpack again - the same staged pattern we used for DNS names.

What you have learned

- **Magic number first, body second.** Fail fast on the wrong format.
- **Little-endian file formats use v / V**, big-endian use n / N; < / > handle the signed or 64-bit cases.
- **Sub-byte fields are plain-Perl shifts**, not template directives.
- **Larger formats decompose into many small templates.** The work of the programmer is to keep each template small and obvious.

With the reference pages as your lookup and these seven chapters under your belt, the next binary format that crosses your desk should read as a solved problem. The [pack](#) and [unpack](#) pages hold the full directive table when you need to confirm a width or a modifier.

- **Bytes and widths** - counting bytes, fixed-width integer directives, the native-vs-fixed distinction.
- **Endianness** - byte order, the < / > modifiers, the portable n / N / v / V shortcuts.
- **Strings** - a / A / Z (binary, text, C string), bit strings, hex strings.
- **Grouping and counts** - () groups, repeat counts, [...] forms, the / item-count-prefix directive.
- **Positioning** - x / X / @ / . for moving around inside a template without producing a value.
- **Network protocols** - build and parse a DNS query header.

- **File formats** - parse a GIF header, the canonical “magic number plus fixed fields” file.

Two chapters (*network-protocols*, *file-formats*) are full worked examples. Walk through one of them once you have read enough of the other chapters to recognise the directives.

4.11.3 A first round-trip

The shortest useful example of pack and unpack working together - packing a dotted-quad IP address into four raw bytes and reading it back:

```
my $raw      = pack    "C4", 192, 168, 1, 42;    # "\xc0\xa8\x01\x2a"
my @octets   = unpack  "C4", $raw;              # (192, 168, 1, 42)
```

C packs one unsigned byte; C4 packs four of them in a row. On the way out, the same template pulls them back into a list of four integers. Every more complicated binary format is layered on top of that idea.

4.11.4 Conventions in the examples

- Expected output appears as an inline # ... comment on the same line as the expression that produced it, or directly below the code block.
- Examples run under a byte-oriented interpreter - no use utf8, no use open pragma changing layers. If a filehandle appears, assume binmode \$fh has already been called.
- Multi-byte constants such as "\xc0\xa8\x01\x2a" are written with hex escapes rather than raw byte values so the printed text matches what you see in the code.

4.12 Opening files

Everything a Perl program does with a file, a pipe, or a buffer goes through a **filehandle**. This chapter is a recipe collection for the open built-in: how to read, write, append, pipe, encode, and recover from errors - each recipe small enough to copy, each warning pinned to the line that bites.

The pre-opened handles - STDIN, STDOUT, STDERR, ARGV - are available from the first line of a program:

```
print STDERR "debug: entered phase 2\n";
print STDOUT "name? ";
my $name = <STDIN> // die "no input";
while (<ARGV>) { ... }
```

Every other handle you open yourself. The modern form is always three arguments - handle, mode, target - and the handle is a lexical scalar:

```
open my $fh, "<", $path or die "open $path: $!";
```

Read that form until it is muscle memory. Everything else on the following pages is a variation on it.

4.12.1 What this chapter does not cover

- **Locking.** Opening does not lock. See `flock` when you need advisory locking across processes.
- **Low-level descriptor control.** For `O_EXCL`, `O_NONBLOCK`, and explicit permission bits, see `sysopen`.
- **Directory iteration.** `open` is for files, pipes, and in-memory buffers. Directory reading uses `opendir` / `readdir`.

4.12.2 Where to go next

Reading, writing, appending

The three everyday tasks - read an existing file, create one from scratch, add to the end of one - are three different modes of the same `open` call. This page shows each in the form that actually belongs in new code.

The three-argument form

```
open my $fh, MODE, EXPR or die "open: $!";
```

- `my $fh` is a fresh lexical. `open` fills it in with a handle reference when it succeeds; on failure it leaves `$fh` undefined and sets `$!`.
- `MODE` is a short string - "<", ">", ">>", and a handful of others - that says what you want to do.
- `EXPR` is the pathname. It is treated as a plain string: leading or trailing whitespace is preserved, and the string is never reinterpreted as a shell command.

The older two-argument form (`open $fh, "<$path"`) still works, but it re-parses the string looking for mode characters and for the “magic” pipe syntax. Do not feed it a filename that came from a user.

Reading

```
my $path = "/etc/hostname";
open my $fh, "<", $path or die "open $path: $!";

while (my $line = <$fh>) {
    chomp $line;
    # ... work with $line ...
}

close $fh or die "close $path: $!";
```

Two observations that trip up beginners:

- `<$fh>` calls `readline` under the hood. In boolean context - as inside the `while` head - the result is tested for being defined, not truthy. That matters because a file containing the line `"0\n"` would otherwise stop the loop early. `while (<$fh>)` is

a special-case spelling that Perl rewrites to `while (defined($_ = <$fh>))` for you.

- `close` can fail. On a read handle that is rare, but on a write handle it is the moment buffered data is finally flushed, so a closing error is a real error. Check it.

If you want the whole file as one string, slurp it:

```
open my $fh, "<", $path or die "open $path: $!";
local $/;                                # undef = slurp mode
my $all = <$fh>;
close $fh                                or die "close $path: $!";
```

The local `$/` is the idiomatic “for the rest of this scope, turn off line splitting”. When control leaves the enclosing block, `$/` returns to its previous value.

Writing (clobber)

```
open my $fh, ">", $path or die "open $path: $!";
print $fh "first line\n";
print $fh "second line\n";
close $fh                                or die "close $path: $!";
```

- `>` **creates** the file if it does not exist, and **truncates** it to zero bytes if it does. There is no confirmation step.
- The filehandle does not need a comma after it in a `print` invocation. `print $fh "text"` is correct; `print $fh, "text"` is a different expression that treats `$fh` as a list element.
- Buffered writes are held in memory until `close`, an explicit flush, or the buffer fills. If your program crashes between `print` and `close`, a writer with pending buffered data loses it. Always pair a writing open with a checked close.

Appending

```
open my $fh, ">>", $path or die "open $path: $!";
print $fh scalar(localtime), " event\n";
close $fh                                or die "close $path: $!";
```

- `>>` creates the file if needed and positions the write cursor at the end.
- On a POSIX kernel, each `write(2)` on an append-mode descriptor is atomic with respect to the file’s end-of-file marker. Two processes appending short lines to the same log therefore interleave line-by-line, not byte-by-byte. Do not rely on that for lines longer than `PIPE_BUF` bytes.

Closing as a real step, not a formality

A failed `close` on a write handle typically means the kernel refused to flush - out of space, I/O error, network filesystem trouble. Treat it exactly like a failed `print`:

```
close $fh or die "close $path: $!";
```

Ignoring it is how “the file looked fine during testing” becomes “production wrote half a log entry and moved on.”

What about read-write?

"<" opens a file for both reading and writing, without truncating. It is a real mode and it works, but it is also the source of most “my file came out corrupted” bug reports. If you need random-access reads with in-place overwrites, reach for `sysopen`, `sysread`, and `syswrite` so the buffering layer does not guess at seek positions on your behalf.

For almost every job labelled “edit the file”, the right answer is write a new file, `fsync`, rename over the old one. That is the pattern atomic editors use.

Next

With reading, writing, and appending covered, the next question is what the bytes *mean*. See [Encoding and layers](#).

Encoding and layers

A filehandle in Perl is not a raw pipe to bytes - it is a stack of PerlIO layers. Each layer transforms data on the way in or out: translating between characters and octets, converting newlines, applying compression, buffering. This page is about the two or three layers you will touch in everyday code.

Why this matters

When you read a text file, the bytes on disk are not the characters your program manipulates. UTF-8 encodes the character ä as two bytes (0xC3 0xA4). If Perl reads those bytes as if they were already characters, you get two garbage characters - classic “mojibake”. If you write characters without saying how they should be encoded, the reverse happens.

The layer stack is where you tell Perl how to translate. A correct stack for a UTF-8 text file is one line:

```
open my $fh, "<:encoding(UTF-8)", $path or die "open $path: $!";
```

The two places layers can live

Layers attach to a handle in two ways:

1. As part of the mode string at open time, combined with the read/write direction:

```
open my $fh, "<:encoding(UTF-8)", $path or die;
open my $fh, ">:encoding(UTF-8)", $path or die;
open my $fh, ">>:encoding(UTF-8)", $path or die;
```

2. After opening, via `binmode`:

```
open my $fh, "<", $path      or die;
binmode $fh, ":encoding(UTF-8)" or die "binmode: $!";
```

The two forms are equivalent for freshly opened handles. `binmode` is what you reach for on a handle you did not open - `STDIN`, `STDOUT`, `STDERR`, and handles passed in from elsewhere.

The layers you actually need

Four cover ninety-nine percent of cases:

- `:encoding(UTF-8)` - characters in Perl-land, UTF-8 bytes on disk. The name of the encoding is anything Perl's `Encode` module understands: `"UTF-8"`, `"ISO-8859-15"`, `"Windows-1252"`, `"Shift_JIS"`, and so on.
- `:utf8` - a looser sibling of `:encoding(UTF-8)` that skips validation on input. It is faster and more dangerous: malformed bytes become malformed strings that misbehave later. Prefer `:encoding(UTF-8)` unless you have measured and proven the validation cost matters.
- `:raw` - bytes through, no translation. Equivalent to plain `binmode` with no second argument. Use it for binary formats, for anything you intend to hash or compare byte-exactly, and for data you are about to hand to a format-specific parser.
- `:crlf` - converts `\r\n` to `\n` on read and back on write. On Linux this is a no-op by default; you add it explicitly when reading files written on Windows and you want `\n`-terminated lines in memory.

Layers compose in writing order:

```
open my $fh, "<:raw:encoding(UTF-8)", $path or die;
```

Reads as: strip to raw bytes first, then decode them as UTF-8. The `:raw` prefix is common when you want a predictable baseline to layer on top of, independent of whatever the open pragma set as the default.

A sensible default for text programs

If a program deals with UTF-8 text throughout, the cleanest move is to declare that at the top:

```
use open qw< :std :encoding(UTF-8) >;
```

- `:std` applies the given layer to `STDIN`, `STDOUT`, `STDERR`.
- `:encoding(UTF-8)` becomes the default layer for every subsequent open call inside the file.

After that pragma, bare three-argument opens decode UTF-8 automatically:

```
use open qw< :std :encoding(UTF-8) >;

open my $fh, "<", $path or die "open $path: $!";
while (my $line = <$fh>) {
```

(continues on next page)

(continued from previous page)

```
}  
    # $line is a character string, already decoded
```

Do not use a bare "<" without having either set a default encoding or explicitly planned for raw bytes. Reading without an announced encoding gives you the “Latin-1 superset” that looks right for ASCII and breaks on the first non-ASCII byte.

Binary files

For binary data, declare raw mode up front:

```
open my $in, "<:raw", $in_path or die "open $in_path: $!";  
open my $out, ">:raw", $out_path or die "open $out_path: $!";  
  
my $buf;  
while (my $n = read $in, $buf, 65536) {  
    print $out $buf or die "write: $!";  
}  
close $in or die;  
close $out or die "close $out_path: $!";
```

`read` is the buffered fixed-size reader; `readline` is the line-oriented one. For binary data you almost always want `read`.

The explicit `:raw` on the output is the belt-and-braces move: if someone added `use open ':encoding(UTF-8)'` to the module three years from now, your binary writer would start mangling bytes. The layer on the handle overrides the pragma.

Changing a handle's layers after open

`binmode` without a layer argument strips all encoding layers, reducing the handle to raw bytes. With a layer argument, it pushes the given layer onto the stack:

```
binmode STDOUT;                # raw bytes  
binmode STDOUT, ":encoding(UTF-8)"; # decode strings as UTF-8  
binmode STDIN, ":encoding(MacRoman)";
```

Call `binmode` before any data moves through the handle. Pushing a layer mid-stream is technically possible but the transitional byte handling is brittle.

Next

The file-on-disk case is covered. Next: the same `open` call, but with a running program on the other end. See *Pipes*.

Pipes

The same `open` that attaches a handle to a file can attach it to a child process. Two modes select the direction:

- `"- |"` - you *read* the child's standard output.

- `"| -"` - you *write* to the child's standard input.

In both cases the child runs concurrently with your Perl program, and its exit status becomes available in `$?` when the handle is closed.

Reading from a command

```
open my $fh, "|-", "sort", "-u", glob("unsorted/*.txt")
    or die "pipe: $!";

while (my $line = <$fh>) {
    # ... process one sorted unique line at a time ...
}

close $fh or die pipe_close_diag($?, $!);
```

What happens:

- Perl forks. In the child, `stdout` is connected to the write end of a pipe, and the command is `execed`.
- In the parent, `$fh` reads from the other end. As far as your code is concerned, it is a normal read handle - `readline`, `read`, `<$fh>`, and so on all work.
- When you finish reading, `close` waits for the child to exit and records its status in `$?`.

Writing to a command

```
open my $fh, "|-", "gzip", "-9", "-c"
    or die "pipe: $!";

for my $line (@records) {
    print $fh $line, "\n";
}

close $fh or die pipe_close_diag($?, $!);
```

Symmetry: the child's `stdin` is connected to the read end of the pipe; your parent writes to the other end; `close` waits for the child to exit.

One behaviour that often surprises people: `close` on a write-pipe flushes everything buffered, shuts the pipe's write end, and *only then* waits for the child. If the child is slow, `close` blocks until the child is done. That is almost always what you want.

List form versus string form

Both `open` calls above used the **list form**:

```
open my $fh, "|-", "sort", "-u", glob("unsorted/*.txt") or die;
```

Perl recognises that the third and subsequent arguments form a command and its argv. It does **not** involve a shell:

- No word splitting, no glob expansion, no redirection, no variable substitution.
- A filename containing a space or a ; is just a filename.
- There is no shell to mis-quote against, so the class of bugs known as “shell injection” is structurally impossible.

The alternative is the **string form**:

```
open my $fh, "-|", "sort -u unsorted/*.txt" or die;
```

Perl passes the single string to `/bin/sh -c`. The shell does word splitting, glob expansion, redirection, everything. That is what makes this form useful - you can write `"cat -n > numbered.txt"` on the command side and the shell sets up the `>` for you - and what makes it dangerous when any piece of the string came from outside your program.

Rule of thumb: **list form by default, string form only when a shell feature is the point**. When a filename glob is the reason to use a shell, consider pre-expanding the glob with Perl’s own glob built-in and staying in list form, as the example at the top of this page did.

Decoding \$?

After a pipe is closed, `$?` is set the same way `wait(2)` returns: a 16-bit value with the low byte carrying signal information and the high byte carrying the exit code. In Perl terms:

- `$? == 0` - the child exited cleanly with status 0. Success.
- `$? == -1` - close could not even wait for the child, because something went wrong before `fork/exec`. `#!` has detail.
- `$? & 0x7F` - signal number if the child died on a signal. `$? & 0x80` (the high bit of the low byte) is set if the child dumped core.
- `$? >> 8` - exit status if the child exited normally.

A diagnostic helper pulls these apart:

```
sub pipe_close_diag {  
    my ($status, $err) = @_;  
    return "pipe close: $err"           if $status == -1;  
    my $signal = $status & 0x7F;  
    return "child died on signal $signal" if $signal;  
    my $exit    = $status >> 8;  
    return "child exited with status $exit" if $exit;  
    return "child exited cleanly";  
}
```

Pair that with every pipe close:

```
close $fh or die pipe_close_diag($?, $!);
```

Not checking the close on a pipe is the same mistake as not checking it on a write handle - except louder, because a child process failed silently in the background.

One command, one pipe

open with `-|` or `|-` gives you a pipe in *one* direction. There is no standard way to get both directions from a single open call, because two-way pipes to a co-process require careful buffering and deadlock avoidance that the open shape cannot express. When you need that, reach for `IPC::Open2` or `IPC::Open3` - built for the job, with the handshake caveats documented in their own POD.

Next

The third kind of thing open can attach a handle to is neither a file nor a process - it is a string in memory. See *In-memory handles*.

In-memory handles

The third thing open can attach a handle to is a plain scalar variable. The scalar is used as the “file”: reads pull bytes out of it, writes push bytes into it. Nothing touches the filesystem, nothing forks.

This is the cleanest way to make a function that normally writes to a filehandle also produce an in-memory string. Any code that accepts “a filehandle” continues to work.

The signature

```
open my $fh, MODE, \ $scalar or die;
```

The third argument is a **reference** to a scalar. That reference is what distinguishes an in-memory handle from a pathname - a plain string `"$scalar"` would be a filename.

Modes work as they do for files:

- `"<"` - read from the contents of the scalar.
- `">"` - write into the scalar, clobbering any existing content.
- `">>"` - append to the scalar.

Writing into a buffer

```
my $buf = "";
open my $fh, ">", \ $buf or die "open buffer: $!";

print $fh "hello, ";
print $fh "world\n";
printf $fh "%d items\n", 42;

close $fh or die "close buffer: $!";

print $buf;           # hello, world\n42 items\n
```

After close, `$buf` holds everything that was printed. Before close, the data may still be sitting in PerlIO’s buffer - treat it as not-yet-final, just as you would with a real file.

The typical use is capturing output from a function that expects to print to a handle:

```
sub render {
    my ($fh, @rows) = @_;
    for my $row (@rows) {
        print $fh join("\t", @$row), "\n";
    }
}

my $buf = "";
open my $fh, ">", \ $buf or die;
render($fh, @rows);
close $fh or die;
```

No temporary file, no tmpfile, no cleanup.

Reading from a buffer

```
my $source = "one\ntwo\nthree\n";
open my $fh, "<", \ $source or die "open buffer: $!";

while (my $line = <$fh>) {
    chomp $line;
    # ... work with $line ...
}

close $fh or die;
```

This turns any string-processing task that “would be nicer with a filehandle” into one. Test data in particular reads cleanly:

```
my $fixture = <<'END';
name,count
apples,3
pears,7
END

open my $fh, "<", \ $fixture or die;
parse_csv($fh);
close $fh or die;
```

`parse_csv` sees a filehandle, does not know it is a string in disguise, and the test is entirely self-contained.

Layers work the same way

In-memory handles accept PerlIO layers:

```
my $buf = "";
open my $fh, ">:encoding(UTF-8)", \ $buf or die;
print $fh "café\n";
close $fh or die;
```

(continues on next page)

(continued from previous page)

```
# $buf now contains the UTF-8 bytes: c a f 0xC3 0xA9 0x0A
```

Without the layer, `$buf` would contain the character string `"café\n"`. With the layer, the write step encodes to bytes on the way into the buffer. Which one you want depends on whether the buffer is destined for something that expects characters (another Perl function) or bytes (a socket, a hash function, a file on disk).

When not to use this

- **When you want capture with `fork` and `exec`.** In-memory handles are pure Perl; a child process cannot see them. If you want to capture the output of an external command, use a read-pipe as described in [Pipes](#).
- **For bulk data that would not fit in RAM.** The scalar grows as you write. A one-gigabyte log is better served by a real tempfile.
- **For seekable random-access editing.** `seek` works on in-memory handles, but the semantics around growing the buffer past its current end are subtle. For random access, use a real file.

Next

Every example on every page in this chapter has been shouting or `die "...: $!"`. It is time to earn that idiom. See [Handling errors](#).

Handling errors

Every I/O operation can fail. The filesystem can be full, the permissions can be wrong, the file can be missing, the network mount can vanish mid-read. Perl does not raise exceptions for these by default - it returns a false value and sets `$!`. The programmer's job is to notice.

The `or die` idiom

```
open my $fh, "<", $path or die "open $path: $!";
```

The form has three loadbearing parts:

- **`or`, not `||`.** Precedence matters: `open(...) || die` would group the `||` against the full argument list of `open`. Using `or` binds lowest of all the operators involved and always does what it reads as.
- **The filename in the message.** `"$!"` alone says "Permission denied" with no context. `"open $path: $!"` says what file, in what step. When logs are all you have, context is everything.
- **`$!`, not `$@`.** `$!` is the system error - `errno` with a human-readable face. `$@` is the exception from the most recent `eval`. Mixing them up at 3am has happened to everyone.

Every open, every close, every print, every read, every binmode in production code has a check. There is no “it obviously cannot fail here”. The one time it does, you will want the diagnostic.

Read \$! immediately or not at all

\$! reflects the **most recent** system call. A naive sequence:

```
open my $fh, "<", $path;
my $line = <$fh>;                                # might also set $!
warn "something failed: $!" unless $line;
```

If the open failed, \$! is already overwritten by the time <\$fh> runs on the undefined handle. Check the return value of each call right where you call it.

Pattern: check the close as carefully as the open

```
open my $fh, ">", $path or die "open $path: $!";
print $fh @records      or die "write $path: $!";
close $fh               or die "close $path: $!";
```

On a write handle, close is where the kernel actually commits the buffered data. If the disk is full, print may have succeeded into memory and close is where you find out. The close check is not a stylistic flourish; it is the one that catches the half-written log entry.

For pipes, close additionally waits for the child and sets \$? . See [Pipes](#) for the diagnostic wrapper that decomposes \$? into signal and exit-status components.

autodie - errors become exceptions

For code where every line is a ... or die ... chain, the autodie pragma is worth the import:

```
use autodie qw(:all);

open my $fh, "<", $path;
while (my $line = <$fh>) {
    # ...
}
close $fh;
```

With autodie, built-ins like open, close, print, read, chdir, unlink, and a long list of siblings throw a structured exception on failure instead of returning false. The caller catches them with eval { ... } or a block-form try, and the message names the call and the path automatically.

Two practical notes:

- autodie is **lexical**. It applies only inside the block where it is use-d. A module you call from an autodie block does not pick up the pragma.

- `autodie` does not change the return convention for the **success** case. `open` still returns a true value; `$fh` is still the handle. You can delete your `or die` suffixes and everything else keeps working.

What the Perl exception looks like

Whether you die manually or let `autodie` do it, the exception travels up the stack until something catches it. Catching is `eval` plus a check of `$@`:

```
my $fh = eval {
    open my $h, "<", $path or die "open $path: $!\n";
    $h;
};
if ($@) {
    warn "could not read $path: $@";
    # ... fall back to a default, or return an error, or re-die ...
}
```

Two habits that save time:

- **Terminate die strings with `\n` when you want the raw message.** Without a trailing newline, Perl appends " at FILE line N" to the die string. Sometimes that is what you want; often it is noise.
- **Prefer structured objects for non-trivial errors.** A `bless { path => ..., errno => $!, step => "open" }, 'IOError'` costs three lines and gives the caller something to introspect instead of a string to regex.

A minimal checklist

Every open-and-process block should look, structurally, like this:

```
open my $fh, MODE, $path or die "open $path: $!";
binmode $fh, $layer      or die "binmode $path: $!"
                        if $layer_not_already_in_mode;

# ... do I/O; each op checked in-place ...

close $fh                or die "close $path: $!";
```

If the code deviates - a missing check, a `||` where an `or` belongs, a `$@` where `$!` belongs - that is where the bug is hiding.

Where to go next

This chapter covered the everyday `open`. When you need flag-level control over the system call itself - for example, `O_EXCL` to refuse to clobber, or `O_NONBLOCK` for a pipe you intend to select on - reach for `sysopen` and its unbuffered companions `sysread` and `syswrite`.

4.12.3 Reference cross-links

Every recipe here links into the per-built-in reference when a function's full behaviour matters:

- [open](#) - full argument shape, all modes
- [close](#) - closing and checking the result
- [binmode](#) - layers after the fact
- [readline](#) - the function behind <\$fh>
- [sysopen](#) - low-level open with flag bits
- [sysread](#) - unbuffered read
- [syswrite](#) - unbuffered write

4.13 References - a tutorial

A **reference** in Perl is a scalar that points at another value: an array, a hash, a subroutine, another scalar, even another reference. References are the single mechanism that turns Perl's three container types - scalars, arrays, hashes - into arbitrary nested data structures.

Without references you cannot build a hash of arrays, an array of hashes, a tree, a graph, a record with mixed fields, a callback table, or a closure. With them, you can. This tutorial walks you from the two reference-making operators, through the syntax for getting the underlying value back, up to anonymous constructors, code references, and the weak-reference trick that keeps cyclic structures from leaking.

4.13.1 Who this is for

You already write working Perl. You use arrays and hashes comfortably, but the moment the problem calls for a hash whose values are arrays - or an array of records, or a callback you can store in a table - the syntax stops being obvious. That gap is what references fill, and what this tutorial closes.

4.13.2 How this tutorial is organised

Each chapter introduces one concept and stands on its own. Read them in order the first time; jump to the relevant chapter later.

Basics - taking references and using them

There are only two ways to **make** a reference to something that already has a name, and only two ways to **use** a reference once you have it. This chapter covers both. Anonymous constructors (`[...]`, `{...}`) are covered in *Anonymous references*.

Making a reference with backslash

Put a backslash in front of any variable, and you get a scalar that refers to it:

```
my @array = (1, 2, 3);
my %hash  = (APR => 4, AUG => 8);
my $n     = 42;

my $aref  = \@array;    # reference to the array
my $href  = \%hash;     # reference to the hash
my $sref  = \$n;        # reference to the scalar
```

A reference is itself a scalar. You can copy it, pass it, store it in another data structure, without touching the thing it points at:

```
my $copy   = $aref;      # two scalars, one underlying array
my @slots;
$slots[3]  = $href;      # hash reference stored as an array element
my $pulled = $slots[3];
```

Both `$aref` and `$copy` point at the **same** array. Changing the array through one is visible through the other. To actually copy the contents, see *Anonymous references*.

What a reference prints as

Stringify a reference and you get a type tag plus an address:

```
print "$aref\n";        # ARRAY(0x55d3c1a02b40)
print "$href\n";        # HASH(0x55d3c1a02be0)
```

If you ever see output like that by accident, you printed the reference where you meant to print the contents.

Using a reference - the curly-brace form

Wherever you could have written a variable name, you can put the reference in curly braces instead. The rule is mechanical:

Operation on @array	Same operation via \$aref
@array	@{\$aref}
scalar @array	scalar @{\$aref}
\$array[3]	\${\$aref}[3]
\$array[3] = 17	\${\$aref}[3] = 17
for my \$x (@array) { ... }	for my \$x (@{\$aref}) { ... }

Hashes follow the same pattern:

Operation on %hash	Same operation via \$href
%hash	%{\$href}
keys %hash	keys %{\$href}
\$hash{red}	\${\$href}{red}
\$hash{red} = 17	\${\$href}{red} = 17

And scalars:

```
my $x    = 10;
my $ref  = \$x;
print ${$ref};           # 10
${$ref} = 99;
print $x;                 # 99
```

The curly braces can usually be dropped when what's inside is a plain scalar variable: `@$aref` means the same as `@{$aref}`, and `$$sref` means the same as `${$sref}`. Start with the curly form until the short form stops looking ambiguous; `$$hash{key}` and `${$hash}{key}` both work, but `$$table{$country}[0]` is easier to misread than `${$table}{$country}[0]`.

Using a reference - the arrow form

The most common thing you do with a reference is fetch one element out of the array or hash it points at. The curly form works (`${$aref}[3]`) but is hard to read. The arrow form is the abbreviation:

```
$aref->[3]           # same as ${$aref}[3]
$href->{red}          # same as ${$href}{red}
```

The arrow is not cosmetic. It is the thing that distinguishes two different meanings:

```
$aref->[3]           # fourth element of the array $aref points at
$aref[3]             # fourth element of the unrelated array @aref
```

`$aref` and `@aref` are separate variables, as unrelated as `$x` and `@x`. Forgetting the arrow silently fetches from a different array that probably does not exist. Running under `use strict` turns the mistake into a compile-time error instead of a silent wrong answer at runtime.

Asking what a reference points at

The `ref` built-in returns a string describing the referent's kind:

```
ref $aref            # ARRAY
ref $href            # HASH
ref $sref            # SCALAR
ref \&printer        # CODE
ref $n               # empty string - $n isn't a reference
```

Combine with boolean context to test “is this a reference?”:

```

if (ref $maybe) {
    # it's a reference of some kind
}
if (ref $maybe eq 'ARRAY') {
    # it's specifically an array reference
}

```

See [ref](#) for the full list of return strings (including CODE, GLOB, REF, Regexp, and the package name when the reference has been blessed into a class).

Two common beginner mistakes

- **Treating a reference like the underlying container.** `push $aref, 1` does not push into the array; it's a type error. You must dereference: `push @{$aref}, 1` or (equivalently) `push @$aref, 1`.
- **Dropping the arrow between a variable and its subscript.** `$aref[3]` is not the same as `$aref->[3]`. The first reads from `@aref`, the second reads through the reference.

Where to go next

- *Arrays of arrays* - the two-dimensional case, and the arrow-between-subscripts rule.
- *Anonymous references* - `[...]` and `{...}` for building structures without named intermediates.

Arrays of arrays

Perl arrays hold scalars, and scalars cannot directly contain other arrays. To build a two-dimensional array - a matrix, a grid, a table of rows - you store **references** to inner arrays as the outer array's elements. This chapter shows how.

Building a matrix

The most direct form uses anonymous array constructors (`[...]`, covered in detail in *Anonymous references*) so you don't have to name every inner array:

```

my @m = (
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
);

```

`@m` has three elements. Each element is a scalar holding an array reference. There are four arrays in play: the outer `@m` and three anonymous inner arrays.

Reading and writing elements

`$m[1]` is the reference to the second row. To reach the third element of that row, apply the arrow:

```
$m[1]->[2]          # 6
$m[0]->[0]          # 1
$m[2]->[2] = 99;    # overwrite the bottom-right cell
```

Between **two subscripts**, the arrow is optional. This rule is what makes multidimensional access readable:

```
$m[1][2]            # same as $m[1]->[2]
$m[2][2] = 99;      # same as $m[2]->[2] = 99
```

The arrow remains **required** between a scalar variable and its first subscript: `$aref->[0]` cannot be shortened to `$aref[0]` because that would mean “first element of `@aref`”. The shortcut applies only **inside** a chain of subscripts.

For three dimensions it’s the same idea:

```
$cube[2][3][5]      # readable
${${$cube[2]}[3]}[5] # the same expression fully spelled out
```

The second form is what you’d have to write without the arrow shortcut. Nobody does that.

Iterating over a matrix

To walk rows, loop over the outer array; each iteration gives you an array reference. To walk cells, dereference that reference inside the loop:

```
for my $row (@m) {
    for my $cell (@{$row}) {
        print "$cell ";
    }
    print "\n";
}
```

If you also need the indexes, use a counted loop on the outer array and on the dereferenced inner:

```
for my $i (0 .. $#m) {
    for my $j (0 .. ${$m[$i]}) {
        print "m[$i][$j] = $m[$i][$j]\n";
    }
}
```

`${$m[$i]}` is the last-index operator applied to the dereferenced inner array. It’s the `$array` form with `{m[$i]}` standing in for the array name.

Growing rows and columns

Append to a row by pushing into the inner array via a reference:

```
push @{$m[0]}, 10;      # row 0 is now (1, 2, 3, 10)
```

Append a whole row to the matrix by pushing an array reference onto the outer array:

```
push @m, [10, 11, 12]; # @m now has four rows
```

Both operations use `push` - once on a dereferenced inner array, once on the outer array directly.

Autovivification

Assigning to `$m[$i][$j]` works even if `$m[$i]` doesn't exist yet:

```
my @grid;
$grid[5][7] = 'X';
```

This creates `@grid` with six empty slots and one reference in slot 5. That reference points at a fresh anonymous array whose eighth slot holds 'X'. Perl generates both the outer slot and the inner array automatically. This is called **autovivification** and is the reason you almost never have to write `$m[$i] = []` before using it.

Autovivification fires on *assignment* and on *dereferencing in lvalue context*. Plain reads do **not** autovivify:

```
my @grid;
my $x = $grid[5][7]; # $x is undef; no inner array created
exists $grid[5];     # false
```

Use `defined` to tell the “never existed” case from the “exists and holds undef” case, and `exists` to check the slot itself without autovivifying.

Common mistake: sharing a single inner array

This looks like three rows but is actually one row repeated:

```
my @inner = (0, 0, 0);
my @bad   = (\@inner, \@inner, \@inner);

$bad[0][1] = 9;
print "$bad[2][1]\n"; # 9 - they're all the same array
```

To get three independent rows, construct each one separately. The anonymous-constructor form handles this by always allocating fresh:

```
my @good = ([0, 0, 0], [0, 0, 0], [0, 0, 0]);
$good[0][1] = 9;
print "$good[2][1]\n"; # 0
```

A loop over `map` gives the same result for variable row lengths:

```
my @grid = map { [ (0) x 10 ] } 1 .. 10; # 10×10 of zeros
```

Common mistake: storing the count instead of the elements

Assigning an array to a single scalar slot evaluates the array in **scalar context**, which yields its element *count*, not its contents:

```
for my $i (1 .. 10) {  
    my @array = somefunc($i);  
    $AoA[$i] = @array;          # WRONG - stores the count  
}
```

After this loop every `$AoA[$i]` is a number. To store the elements, wrap them in an anonymous-array constructor so you get a reference to a fresh copy:

```
$AoA[$i] = [ @array ];          # right - a new array, copied
```

If you genuinely wanted the count, say so with `scalar` and a name that reflects it:

```
$counts[$i] = scalar @array;    # explicit - no surprise later
```

Common mistake: one reused array behind every reference

The repeated-array trap from the previous section also appears when the shared array is reused across loop iterations:

```
our @array;  
for my $i (1 .. 10) {  
    @array = somefunc($i);  
    $AoA[$i] = \@array;        # WRONG - same array every time  
}
```

Every `\@array` is a reference to the *same* array, so all ten slots end up pointing at whatever `somefunc` returned last. Again, `[ @array ]` fixes it by copying.

Declaring the array with `my` **inside** the loop is safe, however:

```
for my $i (1 .. 10) {  
    my @array = somefunc($i);  
    $AoA[$i] = \@array;        # fine - fresh array each iteration  
}
```

`my @array` builds a new array on every pass, so each `\@array` refers to a distinct one. The reference outlives the loop body because the array it points at stays alive as long as the reference does. The constructor form `[ @array ]` is still the clearer choice; reach for the `\@array` form only when you understand why it works here.

Where to go next

- *Hashes and mixes* - hashes of arrays and arrays of hashes, the two shapes that cover most real-world records.
- *Anonymous references* - the `[...]` / `{...}` constructors used above, in detail.

Hashes and mixed structures

A hash's values are scalars, so a value cannot literally **be** an array or another hash. It can, however, be a **reference** to one - and that is enough to build every record shape real programs need: hashes of arrays, arrays of hashes, hashes of hashes, and nested combinations.

This chapter walks the three most common shapes. The patterns compose: once you have a hash of arrays and an array of hashes, you have “rows of records with multi-valued fields” for free.

Hash of arrays - grouping values under a key

The motivating example: you read lines of the form `city, country` and want to collect, for each country, the list of its cities.

```
Chicago, USA
Frankfurt, Germany
Berlin, Germany
Washington, USA
Helsinki, Finland
New York, USA
```

The natural shape is a hash whose keys are country names and whose values are references to arrays of cities:

```
my %cities_by_country;
while (<>) {
    chomp;
    my ($city, $country) = split /, /;
    push @{$cities_by_country{$country}}, $city;
}

for my $country (sort keys %cities_by_country) {
    my @cities = sort @{$cities_by_country{$country}};
    print "$country: ", join(', ', @cities), ".\n";
}
```

Two things to notice:

- `push @{$cities_by_country{$country}}, $city` works on the very first iteration for every new country. The hash entry does not exist yet, so Perl creates an anonymous array for it, stores a reference, and pushes the city in. That's autovivification again
 - and why you almost never need `$h{$k} = [] unless exists $h{$k}` as a separate line.

- The dereference `@{$cities_by_country{$country}}` is mechanical: “the array that this hash value points at.” Wherever you’d have written `@cities` if it were a named array, you write the curly form instead.

The output is:

```
Finland: Helsinki.
Germany: Berlin, Frankfurt.
USA: Chicago, New York, Washington.
```

Array of hashes - a table of records

When each item in a list carries several named fields, store each record as a hash reference and put the references in an array:

```
my @people = (
    { name => 'Ada',    born => 1815, field => 'computing' },
    { name => 'Hedy',   born => 1914, field => 'signals'   },
    { name => 'Grace',  born => 1906, field => 'compilers' },
);
```

`$people[0]` is a hash reference; `$people[0]->{name}` or (with the arrow-between-subscripts shortcut) `$people[0]{name}` is 'Ada'.

Typical operations:

```
# Sort by a field
my @by_year = sort { $a->{born} <=> $b->{born} } @people;

# Select rows
my @early = grep { $_->{born} < 1900 } @people;

# Project one field out
my @names = map { $_->{name} } @people;

# Mutate a record in place
$people[1]{field} = 'radio';
```

Inside `sort`, `grep`, and `map` the record is a hash reference, so you reach its fields through the arrow. `$a` and `$b` inside a sort comparator are hash references here, not hashes.

Hash of hashes - lookup tables keyed by structured values

When each record is keyed by a unique identifier, a hash of hashes is usually nicer than an array of hashes:

```
my %people = (
    ada => { born => 1815, field => 'computing' },
    hedy => { born => 1914, field => 'signals'   },
    grace => { born => 1906, field => 'compilers' },
);
```

(continues on next page)

(continued from previous page)

```
print $people{ada}{born};          # 1815
$people{grace}{field} = 'COBOL';  # mutate one record
```

Iterate in a definite order by sorting the keys:

```
for my $key (sort keys %people) {
    my $rec = $people{$key};
    print "$key: born $rec->{born}, $rec->{field}\n";
}
```

Binding the inner reference to a named lexical (\$rec) is a habit worth forming - it makes the code read top-to-bottom instead of repeating \$people{\$key}{field} five times.

Mixed - records with multi-valued fields

Put it all together: each record is a hash, and one of its fields is a reference to an array.

```
my %library = (
    'The Road' => {
        author => 'Cormac McCarthy',
        year   => 2006,
        genres => ['fiction', 'post-apocalyptic'],
    },
    'Thinking in Systems' => {
        author => 'Donella Meadows',
        year   => 2008,
        genres => ['non-fiction', 'systems'],
    },
);

for my $title (sort keys %library) {
    my $rec = $library{$title};
    my $genres = join ' ', ' ', @{$rec->{genres}};
    print "$title ($rec->{year}, $rec->{author}): $genres\n";
}
```

@{\$rec->{genres}} is the same pattern as before: reach into the hash, get the array reference, dereference it with @{...}.

A field can just as easily hold a **code reference** or a nested **lookup hash**. Nothing about the access changes; you follow the same arrow into whatever each field points at:

```
my $rec = {
    name      => 'lookup',
    CALLBACK => sub { uc $_[0] },
    LOOKUP    => { red => '#f00', green => '#0f0' },
};

$rec->{CALLBACK}->('hi');    # call the code ref - 'HI'
$rec->{LOOKUP}{red};         # nested hash lookup - '#f00'
```

`$rec->{CALLBACK}` is a code reference, so the trailing `->($arg)` calls it. `$rec->{LOOKUP}` is a hash reference, so `{red}` subscripts through it (the arrow between the two subscripts is elided).

Copying an array of refs copies the array, not the referents

When a record holds an array of references, copying that array with `[ @list ]` makes a *new* array whose elements are the *same* references. The copy and the original share their referents, so a mutation through one path is visible through every path that holds those same element refs:

```
my %TV = (
    simpsons => {
        kids => [ { name => 'Bart', age => 10 },
                  { name => 'Lisa', age => 8 } ],
    },
);

my @also = @{ $TV{simpsons}{kids} };    # array copied; hashes shared

$TV{simpsons}{kids}[0]{age}++;          # mutate through one path
print $also[0]{age};                    # 11 - the other path sees it.
→too
```

`@also` is a separate array, so pushing onto it does not grow the original. But its elements are the very same hash references, so bumping `age` through `$TV{simpsons}{kids}[0]` changes the hash that `$also[0]` also points at. To get independent records you must copy one level deeper, building fresh hashes rather than reusing their references.

Checking what you have

Before acting on a nested slot it's often useful to know whether the slot is actually there. Use `exists` rather than `defined` when you want to avoid autovivifying empty intermediate structures:

```
if (exists $library{$title} and exists $library{$title}{genres}) {
    push @{$library{$title}{genres}}, 'essential';
}
```

Reading `$library{$title}{genres}` in lvalue context - the left side of an assignment, or as the argument of `push` - autovivifies missing levels. Reading it in rvalue context does not. The `exists` checks above keep the scan read-only.

To test the type of a reference without pulling values out of it, use `ref`:

```
ref $library{'The Road'}          # HASH
ref $library{'The Road'}{genres}  # ARRAY
```

Where to go next

- *Anonymous references* - the `[...]` / `{...}` constructors used throughout this chapter, and when a named variable is the better choice.
- *Subroutine references* - the remaining reference flavour, needed for callback tables and dispatch.

Anonymous references - `[...]` and `{...}`

You don't always want to give an inner array or hash a name before you reference it. Perl provides two constructors that build a new container and hand you a reference to it in a single expression:

- `[...]` builds a fresh anonymous **array** and returns a reference to it.
- `{...}` builds a fresh anonymous **hash** and returns a reference to it.

These are the primary way nested data structures are assembled.

Array: `[...]`

```
my $aref = [1, 'foo', undef, 13];
```

The right-hand side evaluates the list `(1, 'foo', undef, 13)`, allocates a new array, copies those four elements in, and yields a scalar holding a reference to that array. This is **exactly** equivalent to the two-step form with a named intermediate:

```
my @array = (1, 'foo', undef, 13);
my $aref = \@array;
```

...except that no `@array` variable leaks into the surrounding scope. The anonymous array has no name; only the reference exists, and the array stays alive as long as the reference does.

Empty forms are useful as initial values:

```
my $aref = [];           # reference to a new empty array
push @{$aref}, 1, 2;    # now [1, 2]
```

Hash: `{...}`

```
my $href = { APR => 4, AUG => 8 };
```

Same idea for hashes. The expression is a list of key/value pairs (the `=>` is just a comma that auto-quotes its left side), a new hash is allocated, the pairs are inserted, and you get a reference back:

```
my %hash = (APR => 4, AUG => 8);
my $href = \%hash;
```

is the two-step equivalent. And as with arrays, the empty form is a common starting point:

```
my $href = {};  
$href->{red} = 17;
```

`{...}` vs. **block** - a parser corner

A bare `{...}` at statement position is parsed as a **block**, not as a hash constructor:

```
{ foo => 1 };           # block, with `foo => 1` as its expression
```

In most positions - on the right of `=`, inside a list, as an argument to a function - Perl figures out the hash-constructor reading automatically. If you ever need to force the hash-constructor interpretation, put `+` in front:

```
my $x = +{ foo => 1 }; # unambiguous: always a hash reference
```

You'll rarely need this. Inside `return +{...}` from a subroutine and in a few other "statement-like" positions, it's the fix.

Nesting - constructors inside constructors

Because each constructor yields a scalar, and scalars go anywhere, you compose structures by nesting:

```
my $tree = {  
    name      => 'root',  
    children => [  
        { name => 'left',  children => [] },  
        { name => 'right', children => [  
            { name => 'r1', children => [] },  
        ] },  
    ],  
};  
  
print $tree->{children}[1]{children}[0]{name}; # r1
```

Every `[...]` and `{...}` allocates fresh. Unlike the "shared-inner-array" footgun from *Arrays of arrays*, nested constructors never share storage by accident.

Named intermediates vs. anonymous constructors

Both forms work. Pick based on readability:

- **Anonymous constructor** when the container is single-purpose and its only role is "be the value of this field". Nested data literals, one-off lookup tables, callback-table entries.
- **Named intermediate** when you're building the container incrementally, or when you want to use array/hash operators on it directly without repeating `@{...} / %{...}`:


```
my @row;
for my $x (@source) {
    push @row, transform($x);
}
$grid[$i] = \@row;
```

Copying vs. sharing

Anonymous constructors always allocate; they never share. But assigning one reference to another scalar does not duplicate anything:

```
my $a = [1, 2, 3];
my $b = $a;           # $a and $b refer to the SAME array
push @{$b}, 4;
print "@{$a}";        # 1 2 3 4
```

To get a shallow copy of the underlying container, dereference inside a fresh constructor:

```
my $copy_eref = [ @{$a} ];    # new array, same elements
my $copy_href = { %{$h} };    # new hash, same keys/values
```

The new reference is independent of the original. The **values** inside are still shared - if those values are themselves references, the two structures still point at common subtrees. That's shallow copying. For deep copying, reach for a module like `Storable` (`dclone`).

Where to go next

- *Subroutine references* - the third flavour of anonymous thing, using `sub { ... }` instead of `[ ... ]` or `{ ... }`.
- *Weak references* - needed when a nested structure points back at itself.

Subroutine references

A subroutine reference is a scalar that points at a piece of code. Once you have one, you can store it in a variable, pass it to another subroutine as a callback, keep a table of them for dispatch, or call it. Code references are what make dispatch tables, callbacks, iterators, and closures possible in Perl.

Two ways to get a code reference

`\&name` - take a reference to a named subroutine:

```
sub greet { print "hello $_[0]\n" }

my $cref = \&greet;
$cref->('world');    # hello world
```

The `&` sigil is what marks `greet` as a subroutine. The backslash in front takes its reference. Without the `&`, you'd be calling the subroutine and taking a reference to the result.

sub { ... } - an **anonymous** subroutine. It evaluates to a reference to a fresh, nameless piece of code:

```
my $cref = sub { print "hello $_[0]\n" };
$cref->('world');
```

Anonymous subs are the direct analogue of `[ ... ]` and `{ ... }` from the previous chapter: a constructor that gives you back a reference to something new and nameless.

Calling through a reference

`$cref->(@args)` is the one you reach for:

```
$cref->('world');
$cref->($x, $y);
$cref->();           # call with no arguments
```

There is also an older syntax, `&{$cref}(...)`, parallel to the `@{$saref}` / `%{$shref}` form. It's still valid but the arrow form is standard everywhere modern.

Inside the called subroutine, arguments arrive in `@_` as usual:

```
my $max = sub {
    my ($a, $b) = @_;
    $a > $b ? $a : $b;
};
print $max->(3, 7);    # 7
```

Callbacks - passing code to another routine

Any subroutine that takes a code reference can accept either a named-subroutine reference or an anonymous one:

```
sub each_line {
    my ($filename, $cb) = @_;
    open my $fh, '<', $filename or die "$filename: $!";
    while (my $line = <$fh>) {
        $cb->($line);
    }
}

# Pass a named sub:
sub print_upper { print uc $_[0] }
each_line('notes.txt', \&print_upper);

# Or an anonymous one, inline:
each_line('notes.txt', sub { print uc $_[0] });
```

Inline anonymous subs are the idiomatic form when the callback body is short and specific to the call site.

Dispatch tables

A hash of code references replaces long if/elsif chains:

```
my %dispatch = (
    help => sub { print "usage: tool [cmd]\n" },
    list => sub { print "$_\n" for @items },
    quit => sub { exit 0 },
);

my $cmd = shift @ARGV;
if (my $action = $dispatch{$cmd}) {
    $action->();
} else {
    die "unknown command: $cmd\n";
}
```

Each value in %dispatch is a code reference. Looking up by command name and calling through the reference replaces branching with a single lookup. Adding a new command is a single hash entry.

Closures - code that captures lexicals

An anonymous subroutine that references a lexical (my) variable from its surrounding scope **captures** that variable. The variable survives for as long as the subroutine reference does, and each call sees the current value:

```
sub make_counter {
    my $n = 0;
    return sub { ++$n };
}

my $a = make_counter();
my $b = make_counter();
print $a->();      # 1
print $a->();      # 2
print $b->();      # 1   - independent from $a
print $a->();      # 3
```

Each call to make_counter creates a **new** lexical \$n and a **new** anonymous subroutine that captures that \$n. The subroutine returned to \$a and the one returned to \$b close over distinct variables, so their counters don't interfere.

This is how Perl expresses most of what other languages call “objects with private state” without any class machinery: the closed-over lexicals **are** the private state.

A closure captures by reference, not by copy. A loop like this captures the **same** loop variable:

```
my @subs;
for my $i (1 .. 3) {
    push @subs, sub { print "i=$i\n" };
}
```

(continues on next page)

(continued from previous page)

```
}  
$_->() for @subs;      # i=1 / i=2 / i=3
```

...because my `$i` in `for my $i` is re-my'd on every iteration, so each closure captures a *different* `$i`. The common mistake is using a `for`-loop variable that isn't my'd freshly:

```
my $i;                # outer  
my @subs;  
for $i (1 .. 3) {  
    push @subs, sub { print "i=$i\n" };  
}  
$_->() for @subs;      # i=3 / i=3 / i=3 - all share the outer $i
```

The fix is `for my $i (...)` - always prefer the inline `my`.

Introspection

`ref` on a code reference returns the string `CODE`:

```
ref $cref              # CODE  
ref \&greet            # CODE
```

`defined` works as usual to tell “no value” from “any value, including a code reference”:

```
if (defined $dispatch{$cmd}) { ... }
```

Where to go next

- *Weak references* - relevant when a structure of code references holds references back to objects that own it (event emitters, observer chains).

Weak references

Perl tracks object lifetime with **reference counting**. Every value knows how many references point at it; when the count drops to zero, the value is freed. This works well for tree-shaped data, where references flow one way, parent to child. It fails for **cycles**, where a chain of references eventually points back at something earlier in the chain. The cycle's members keep each other alive even when nothing outside the cycle refers to them any more.

A **weak reference** is a reference that does not increment the referent's reference count. It lets you point at something without keeping it alive. Weak references are Perl's standard answer to reference cycles.

How a cycle leaks

Consider a doubly-linked list node that holds both `next` and `prev` references:

```

sub new_node {
    my ($value) = @_;
    return { value => $value, next => undef, prev => undef };
}

my $a = new_node('A');
my $b = new_node('B');
$a->{next} = $b;
$b->{prev} = $a;

```

\$a and \$b each have a reference count of 2:

- \$a is referenced by the lexical \$a **and** by \$b->{prev}.
- \$b is referenced by the lexical \$b **and** by \$a->{next}.

When the lexicals go out of scope, their contribution drops:

- \$a drops to count 1 (only \$b->{prev} left).
- \$b drops to count 1 (only \$a->{next} left).

Neither reaches zero. Both nodes are unreachable from anywhere in your program, yet they keep each other alive for the life of the interpreter. That's a leak.

Fixing the leak with weaken

Scalar::Util::weaken turns an existing reference into a weak one. The reference still works for dereferencing; it just doesn't count toward keeping the target alive.

The standard pattern: whichever direction you consider “back-pointer” or “parent pointer”, make **that** one weak.

```

use Scalar::Util qw(weaken);

my $a = new_node('A');
my $b = new_node('B');
$a->{next} = $b;
$b->{prev} = $a;
weaken $b->{prev};           # the back-pointer

```

Now \$a's reference count stays at 1 (only the lexical \$a contributes; \$b->{prev} is weak and does not count). When \$a goes out of scope, \$a is freed - and because freeing \$a drops \$b->{next}, \$b is freed too. The whole structure collapses.

When a weak reference becomes undef

If the referent is freed while a weak reference still exists, the weak reference **automatically becomes undef**. You never see a dangling pointer:

```

use Scalar::Util qw(weaken);

my $target = { name => 'alive' };

```

(continues on next page)

(continued from previous page)

```
my $weak = $target;
weaken $weak;

print defined $weak ? "yes\n" : "no\n";    # yes
undef $target;                             # last strong ref gone
print defined $weak ? "yes\n" : "no\n";    # no
```

defined \$weak is the standard way to check whether the target is still there. See [defined](#).

Where weak references belong

The rule: use a weak reference for the “non-owning” pointer in any pair of references that would otherwise cycle.

- **Doubly-linked list.** next owns; prev is weak.
- **Parent/child tree with parent pointers.** children owns; parent on each child is weak.
- **Observer pattern.** The subject holds weak references to its observers; observers hold strong references to the subject. When an observer drops its reference, it goes away; the subject’s entry for it turns to undef and can be swept.
- **Caches that shouldn’t keep entries alive.** A cache whose values are weak references does not extend the lifetime of the cached objects.

isweak - inspecting a reference

Scalar::Util::isweak tells you whether a given scalar holds a weak reference:

```
use Scalar::Util qw(weaken isweak);

my $x = { k => 1 };
my $y = $x;
print isweak($y) ? "weak\n" : "strong\n";    # strong
weaken $y;
print isweak($y) ? "weak\n" : "strong\n";    # weak
```

Useful mostly when debugging a lifetime problem. Production code rarely branches on isweak.

Common mistakes

- **Weakening the wrong end.** Weakening the “forward” pointer (the one that owns the successor) frees the successor immediately, before you ever use it. Weak goes on the **back-pointer**.
- **Weakening a lexical.** weaken \$ref where \$ref is a fresh lexical referencing \$target makes the lexical weak. If nothing else holds a strong reference to \$target, it vanishes at the next statement and \$ref becomes undef. Weak references belong in data-structure slots, not typically in local lexicals.

- **Assuming weak references are automatic.** Perl does **not** detect cycles and weaken them for you. You have to opt in.

Beyond Scalar::Util

Scalar::Util ships with Perl; it is always available. For more elaborate needs - WeakRef, Hash::Util::FieldHash - reach for CPAN. For ordinary code, weaken is all you need.

Where to go next

You’ve finished the references tutorial. Good next reads:

- The [ref](#) reference page - the full list of strings it returns, including blessed-object cases.
- [bless](#) - how a plain hash reference becomes an object of a specific class.
- The object-orientation tutorial, once you’re ready for classes built on top of the references you now understand.
- **Basics** - the `\` operator, scalar/array/hash dereference with `$$x`, `@$x`, `%%x`, curly-brace disambiguation, and the `->` arrow for element access.
- **Arrays of arrays** - two-dimensional data, iteration patterns, the arrow-between-subscripts rule that makes `$m[i][j]` read as a matrix.
- **Hashes and mixes** - hashes of arrays, arrays of hashes, and the common record-table shape.
- **Anonymous references** - `[...]` and `{...}` as inline constructors, when to choose them over named variables.
- **Subroutine references** - `&name`, anonymous sub `{...}`, calling with `->(...)`, and how closures capture lexicals.
- **Weak references** - `Scalar::Util::weaken` and why cyclic structures need it.

4.13.3 Related reference pages

- [ref](#) - ask a value what kind of reference it is.
- [defined](#) - the standard way to check “does this reference exist yet?” after autovivification.
- [bless](#) - the step that turns a plain reference into an object.
- [References \(data-types reference\)](#)
 - the terse one-page reference that complements this tutorial.

4.14 Unicode - a tutorial

Modern programs carry accented letters, currency symbols, CJK ideographs, and emoji through their input and output. Perl handles Unicode well, but only if you are explicit

about where text is text and where bytes are bytes. This tutorial is the shortest path to that discipline.

These pages do not teach Unicode itself. You are assumed to know that a character set numbers characters, that an encoding maps those numbers to bytes, and that UTF-8 is one such encoding among several. What follows is the Perl-specific habit you need on top.

4.14.1 The one idea

A Perl string is either a **text string** - a sequence of Unicode code points, where `length` counts characters - or a **binary string** - a sequence of bytes, where `length` counts bytes. The same variable type holds both, but the meaning is different, and the boundary between them is where encoding and decoding happen.

Every bug in this area comes from losing track of which kind of string you have. Every fix comes from making the boundary explicit.

```
my $text = "caf\x{e9}";           # 4 characters: c a f é
my $bytes = "caf\xc3\xa9";        # 5 bytes:      c a f 0xc3 0xa9
length $text;                     # 4
length $bytes;                    # 5
```

Both strings “contain” the word *café*. Only one of them can go through a regex that expects characters; only the other can go down a byte-oriented socket.

4.14.2 The pipeline

A program that handles text correctly follows one shape:

1. **Decode** everything coming in. Bytes on the outside become characters on the inside.
2. **Process** in characters. Regex, length, case conversion, the lot.
3. **Encode** everything going out. Characters on the inside become bytes on the outside.

Step 2 is where Perl is already comfortable. Steps 1 and 3 are where the discipline lives. This tutorial walks through both.

4.14.3 Chapters

Strings and encodings

A Perl string holds a sequence of integers. What those integers *mean* is up to you. In a text string they are Unicode code points; in a binary string they are bytes. Perl does not stamp “this is text” on a scalar - you carry that knowledge with the data, and the boundary between the two worlds is drawn by encode and decode.

Text in source code: use utf8

By default, Perl reads your source file as a sequence of bytes. A string literal containing é is a two-byte string "\xc3\xa9", not the one-character string "\x{e9}". That is almost never what you want when you typed é in an editor set to UTF-8.

use utf8 tells the parser that the source file is UTF-8 and that literals should be decoded as they are read:

```
use utf8;
my $greeting = "café";
length $greeting;                # 4 - characters
```

Without use utf8, the same line produces:

```
my $greeting = "café";
length $greeting;                # 5 - bytes
```

Rule of thumb: if your source file contains any non-ASCII character, write use utf8 at the top and save the file as UTF-8. The pragma only governs the source file; it has nothing to do with I/O.

The internal representation

Perl keeps text strings in a representation of its own choosing. You do not need to know what it is, and you must not depend on it. The only guarantee is:

- A text string is a sequence of code points. length counts them, substr indexes them, chr and ord map between characters and their code points.
- A binary string is a sequence of bytes. The same functions now count and index bytes.

The utf8 pragma controls the *source*. There is also a utf8:: namespace of functions for probing and forcing the internal flag - they exist, they are occasionally useful, and in normal application code you should not need them. Stick to encode and decode.

The Encode module

The standard way to convert between text and bytes is the Encode module:

```
use Encode qw(encode decode);

my $bytes = encode("UTF-8", $text);    # text → bytes
my $text  = decode("UTF-8", $bytes);   # bytes → text
```

The first argument names the encoding. UTF-8 is the obvious default for new code, but Encode supports the ISO-8859-* family, every common Windows code page, Shift-JIS, EUC-JP, EUC-KR, GB2312, GBK, Big5, and dozens more.

Decoding is only reliable if you know which encoding the bytes are in. Nothing in the bytes themselves tells you - that information travels in a Content-Type header, a file's BOM, a protocol's metadata field, or a convention you have agreed with the producer. If you truly do not know, guess UTF-8 first, then fall back on ISO-8859-1 which at least never fails (every byte is a valid code point in that encoding).

Lossy encodings

Not every Unicode character survives every encoding. Converting "caf\x{e9}" to ISO-8859-1 works - é has code point 0xe9 which that encoding can hold. Converting an em-dash \x{2014} to ISO-8859-1 does not - the character has no place in the target, and encode replaces it with a substitution marker by default.

```
use Encode qw(encode);
my $s = "em\x{2014}dash";
my $b = encode("ISO-8859-1", $s);           # "em?dash"
```

Pass a third argument to encode to change that behaviour - see the Encode module documentation for the full list of Encode::FB_* constants. For most application code, the default substitution is the right compromise.

A complete round-trip

Putting decoding, processing, and encoding together:

```
use utf8;
use Encode qw(encode decode);

# bytes arrive from somewhere - HTTP, a file, a socket
my $input_bytes = "caf\xc3\xa9\n";

# step 1: decode to text
my $text = decode("UTF-8", $input_bytes); # "café\n"

# step 2: process as text
chomp $text;
my $shouty = uc $text;                    # "CAFÉ"

# step 3: encode for output
my $output_bytes = encode("UTF-8", $shouty . "\n");
```

The text layer is short - one uc call - but that is the layer where `length`, regex, case conversion, and `chomp` all work in the units the program means.

When you want I/O to do this automatically

Doing decode on every read and encode on every write quickly becomes tedious. The next chapter - *I/O* - shows how to push the conversion into the filehandle itself with the `:encoding` layer, so ordinary `print` and `readline` already speak text.

I/O

The filehandle is the boundary between your program's text world and the outside world of bytes. Getting Unicode right means setting that boundary up correctly at `open` time (or with `binmode`), so that ordinary `print` and `readline` already speak the right encoding.

PerlIO layers

Every filehandle carries a stack of *layers* - small pieces of machinery that transform bytes as they cross the boundary. The relevant ones for Unicode are:

- `:raw` - no transformation. Bytes in, bytes out.
- `:utf8` - check that bytes read form valid UTF-8 and mark the string as text. Trust the user not to produce invalid UTF-8 on writes.
- `:encoding(ENCODING)` - decode on read, encode on write, for any named encoding. Rejects malformed input by default.
- `:crlf` - translate `"\r\n"` to `"\n"` on read, `"\n"` to `"\r\n"` on write. On Linux this is inactive; on Windows it is the default for text handles.

`:encoding(UTF-8)` is what you almost always want. Prefer it over the bare `:utf8` layer - the `:encoding(...)` form validates, `:utf8` does not.

Opening a file with an encoding

The three-argument form of `open` takes a mode that can include layers:

```
open my $in, "<:encoding(UTF-8)", "input.txt" or die "open: $!";
open my $out, ">:encoding(UTF-8)", "output.txt" or die "open: $!";

while (my $line = <$in>) {
    chomp $line;                                # characters, not bytes
    print $out uc $line, "\n";                  # encoded on the way out
}
```

Every `readline` from `$in` returns a text string. Every `print` to `$out` is encoded to UTF-8 on the wire. The body of the loop sees only characters - no decode, no encode, no counting bytes by mistake.

Retrofitting with binmode

If the filehandle is already open - for example the standard streams, or a handle handed to you by another module - attach the layer with `binmode`:

```
binmode STDIN, ":encoding(UTF-8)";
binmode STDOUT, ":encoding(UTF-8)";
binmode STDERR, ":encoding(UTF-8)";
```

Do this once, at the top of the program, before any I/O on those handles. After that the standard streams behave the same as a file opened with an encoding layer.

The use open pragma

Writing the same `:encoding(UTF-8)` layer on every `open` is noisy. The `open` pragma sets the default layers for all subsequent opens in the enclosing scope:

```
use open ":std", ":encoding(UTF-8)";
```

The `:std` tag also re-layers STDIN, STDOUT, and STDERR on the spot, so you get the effect of three `binmode` calls and a default for every later `open` in one line.

This is the one-liner at the top of a modern Unicode-aware script:

```
use utf8;
use open ":std", ":encoding(UTF-8)";
```

- `use utf8` - literals in this source file are UTF-8.
- `use open ":std", ":encoding(UTF-8)"` - stdin/stdout/stderr and every later `open` default to UTF-8 on the wire.

You can also push I/O to UTF-8 from outside the program with the `PERL_UNICODE` environment variable. The letters I, O, E select stdin, stdout and stderr individually, S selects all three at once the way the `use open ":std"` pragma does, and A decodes `@ARGV`, so `PERL_UNICODE=SA` is roughly the command-line equivalent of the two-line preamble above. To decode arguments inside the program instead, write `@ARGV = map { decode('UTF-8', $_, 1) } @ARGV`.

The matching `-C` command-line switch (`-CSA`) is not yet recognised by `ppperl`, which is a discrepancy from stock `perl`; use the `PERL_UNICODE` variable or the `use open` pragma until it is implemented.

Binary I/O alongside text I/O

Some handles carry bytes, not text - image files, compressed streams, network protocols that frame their own encoding. Open those `:raw` and never attach an encoding:

```
open my $img, "<:raw", "photo.jpg" or die "open: $!";
```

Within one program, some handles can be text (`:encoding(UTF-8)`) and others binary (`:raw`). Keep the labels straight: a binary handle gives you byte strings, a text handle gives you text strings, and `$!` is the same either way.

Interactive input

A terminal does not tell you what encoding it is typing in. On modern Linux it is almost always UTF-8; on Windows the default console code page is still commonly CP1252 or CP437. The usual assumption for cross-platform scripts is UTF-8, with `binmode` applied at startup:

```
binmode STDIN, ":encoding(UTF-8)";
```

If your program must support legacy terminals, decode based on the `LANG` or `LC_CTYPE` environment variable, or accept a `--encoding` flag. There is no portable auto-detection.

Checking layers

To see which layers are on a handle, use `PerlIO::get_layers`:

```
use PerlIO;
my @layers = PerlIO::get_layers(\*STDOUT);
```

(continues on next page)

(continued from previous page)

```
print "@layers\n";           # e.g. unix perlIO
↳encoding(utf-8-strict)
```

Useful when a program is silently producing double-encoded output and you want to find out which handle has the surprise layer on it.

Regex and properties

Once your strings are decoded, regex works on characters - and Perl's regex engine knows about Unicode. `\w` matches every Unicode word character, case-insensitive matching folds across scripts, and `\p{...}` gives you access to the full Unicode character database. This chapter covers the habits you need to make that work reliably.

Characters, not bytes

On a text string, every regex metacharacter speaks in characters:

```
use utf8;
my $s = "café";           # 4 characters
$s =~ /^.{4}$/;           # matches
$s =~ /^.{5}$/;           # does not
```

On a binary string, the same metacharacters count bytes. That is usually not what you want - if you find yourself writing a regex against bytes that hold text, decode first.

`\w`, `\d`, `\s` and the friends

In default (Unicode) mode, these shorthands match their Unicode equivalents:

- `\w` - letters, digits, and underscore from every script.
- `\d` - decimal digits from every script (not just 0–9).
- `\s` - whitespace of every kind, including U+00A0 no-break space.

```
use utf8;
"café" =~ /\w+$/;         # matches
"café" =~ /^[a-z]+$/;     # does not - é is outside
"\x{0660}\x{0661}\x{0662}" =~ /\d+$/; # matches Arabic-Indic
↳digits
```

When you want the old ASCII-only meaning, add the `/a` modifier - see *Modifiers* below.

The `\p{...}` property classes

`\p{PROPERTY}` matches any character with a named Unicode property. The property names are drawn from the Unicode character database; the commonly useful ones are:

- `\p{Letter}`, shorthand `\p{L}` - any letter.
- `\p{Lu}`, `\p{Ll}`, `\p{Lt}` - upper, lower, title case letters.
- `\p{Number}`, `\p{N}`, `\p{Nd}` - all numbers, decimal digits.

- `\p{Punct}`, `\p{P}` - punctuation.
- `\p{Space}`, `\p{Zs}` - whitespace, space separators.
- `\p{ASCII}` - the 128 ASCII code points.
- `\p{Script=Greek}` - every character that belongs to the Greek script. Any script name from the Unicode data works here.
- `\p{Block=Cyrillic}` - every character in the Cyrillic block. (Script and block are different: *script* is the writing system a character belongs to; *block* is the code-point range it lives in.)

`\P{...}` is the negation - any character without that property.

```
use utf8;
my $s = "Γειά σου, κόσμε";
my @greek_letters = $s =~ /(\p{Script=Greek})/g;
scalar @greek_letters;                # 12
```

The full catalogue of properties ships with Perl; `perldoc perluniprops` enumerates every name the engine accepts.

Modifiers: `/a`, `/u`, `/l`, `/aa`

Four modifiers change how character classes interpret themselves:

- `/u` - **Unicode mode**. `\w` matches Unicode word characters, `\d` matches Unicode decimal digits, case folding uses Unicode rules. This is the default on text strings.
- `/a` - **ASCII mode**. `\w` becomes `[A-Za-z0-9_]`, `\d` becomes `[0-9]`, `\s` becomes `[\t\n\r\f]`. Case folding is still Unicode - `/foo/ai` still matches `FÖO` through `Ö` if you wrote `foo` - which is almost never what you want when you reached for `/a`.
- `/aa` - **strict ASCII**. `/a` plus case folding is restricted to ASCII. `/foo/aai` matches `F00` but not `FÖO`. Use this when you want an identifier match against a known-ASCII specification (HTTP header names, command keywords).
- `/l` - **locale mode**. Defers to the current POSIX locale. Almost never what you want in a Unicode-aware program; mentioned only so you can recognise it when another codebase uses it.

```
use utf8;
"café" =~ /\w+/;                # matches whole word
↳(Unicode)
"café" =~ /\w+/a;                # matches "caf" (ASCII
↳only)
"F00" =~ /foo/ai;                # matches (case fold
↳anywhere)
"FÖO" =~ /foo/aa;                # does not - strict ASCII
```

Rule of thumb: default mode is right for text; switch to `/aa` when you are parsing an ASCII-specified protocol and want to reject smuggled non-ASCII characters.

Case-insensitive matching

`/i` folds both sides of the match using Unicode case-folding tables. This handles the obvious European pairs (é/É, ß/SS), and the less obvious ones (İ/i, fi/fi).

```
use utf8;
"Ångström" =~ /ångström/i;           # matches
"STRASSE"  =~ /straße/i;             # matches - ß folds to SS
```

For literal-byte case folding (the old ASCII meaning of `/i`), use `/iaa`.

Named code points

`\N{...}` names a Unicode character, either by its official name or by its code-point number:

```
use charnames ();
"\N{LATIN SMALL LETTER E WITH ACUTE}" eq "\x{e9}"; # true
"\N{U+00E9}" eq "\x{e9}";                          # the same
↳ character
```

`\N{U+HEX}` works without any pragma; the long names go through the `charnames` module, loaded automatically inside a `\N{...}` escape (or explicitly with `use charnames qw(:full)`). In a pattern, `\N{...}` is useful when you want the source to document which character you mean without relying on the reader to recognise a hex escape:

```
"café" =~ /caf\N{LATIN SMALL LETTER E WITH ACUTE}/; # matches
```

Splitting and joining

`split` and `substr` both count in the units of the string they operate on. On a text string, the count is characters:

```
use utf8;
my $s = "a,é,b,Ω";
my @parts = split /,/, $s;           # ("a", "é", "b", "Ω")
substr($s, 2, 1);                    # "é"
```

On a binary string, the same operations count bytes. This is the most common subtle bug - an old codebase that decoded input in one function but indexed the result in another, built before the decode step was introduced.

Pitfalls

Most Unicode bugs in Perl programs show up as one of a handful of familiar symptoms. This chapter names each one, shows what produces it, and points to the fix.

“Wide character in print”

The warning:

```
Wide character in print at script.pl line N.
```

means you sent a text string to a filehandle that is not configured for text. The output gets written anyway - Perl encodes it as UTF-8 on the fly - but the warning is your notice that the configuration is wrong.

The fix is to attach an encoding layer to the handle:

```
binmode STDOUT, ":encoding(UTF-8)";
```

or, globally for the standard streams:

```
use open ":std", ":encoding(UTF-8)";
```

Do not silence the warning with `no warnings 'utf8'`. The warning is correct; it is telling you about a real missing configuration.

Double encoding: *café* becomes *café*©

Symptom: text looks almost right, but every non-ASCII character has been replaced by a pair of funny-looking bytes. *café* becomes *café*©, *Straße* becomes *Straße*©.

Cause: you encoded to UTF-8, then encoded again. The second stage read the already-encoded bytes as if they were text, treated each byte as a Latin-1 code point, and encoded those to UTF-8. Two passes through UTF-8 encoding is the classic double-encode.

Common producers:

- A filehandle with `:encoding(UTF-8)` that receives an already encoded byte string. Fix: send text to the handle, or drop the layer.
- An HTTP response whose body is encoded in the application, and encoded again by a middleware that assumed the body was text.
- A database client that decodes on read, in a codebase that was written before the client learned to decode.

Diagnosis: look at the bytes on the wire. If a single non-ASCII character takes four or more bytes when it should take two, you have been through UTF-8 twice.

Missing decode: *café* stays *café* but indexes are wrong

Symptom: prose looks correct on the terminal, but `length` gives too many, `substr` cuts in the middle of a character, a regex like `/^\.{5}$/` unexpectedly matches a four-character word.

Cause: you never decoded. The scalar holds UTF-8 bytes, the terminal happens to also render them as UTF-8, and the visual output hides the fact that Perl sees them as bytes.

Fix: decode on the way in, either with `binmode` on the handle or `Encode::decode` on the string:


```
use Encode qw(decode);
my $text = decode("UTF-8", $raw_bytes);
```

Heuristic: if length \$s is unexpectedly larger than the number of visible characters, you are looking at bytes.

Mojibake: unknown encoding

Symptom: non-ASCII looks like random line-noise - café as café, as café, as café, as café, as a box character. Different visual, different cause each time.

Cause: you decoded with the wrong encoding, or did not decode at all and the terminal guessed wrong.

There is no universal fix - you must know what encoding the source actually used. A few reliable anchors:

- Web: the Content-Type header's charset= parameter, or the HTML <meta charset> element for files served without one.
- Files from legacy Windows applications: usually CP1252 (Western European) or a regional CP1250/CP1251/CP1253.
- Files from classic Mac: MacRoman. Rare but survives in old archives.
- Email bodies: the Content-Type charset of the MIME part.

If nothing tells you, try UTF-8 first; if that fails with a decode error, try ISO-8859-1 (which cannot fail but may produce nonsense).

Mixing text and bytes in concatenation

Symptom: a string that concatenates a decoded text fragment with a raw byte fragment produces garbage after the boundary.

```
use utf8;
my $label = "café";           # text
my $raw   = "\xc3\xa9_ok";   # bytes that also look like
↪text
print $label, $raw;          # "Wide character" or
↪mojibake
```

Cause: the text half goes through whatever encoding layer is on the handle; the byte half is reinterpreted as code points by that same layer, with predictable confusion.

Fix: pick a side. Either decode the byte fragment first, or encode the text fragment to match. Never concatenate the two kinds of string without a conscious choice about which side you are on.

sprintf "%s" is not a conversion

sprintf does not decode or encode; it formats. Passing a byte string through %s yields a byte string; passing a text string yields a text string. If you need to change encoding, call encode or decode explicitly.

Locale-dependent case folding

Symptom: `lc` or `uc` produces different results on different machines, or fails to round-trip through a regex.

Cause: the program is running under the `/l` locale-mode regex modifier, or `uc` / `lc` have been redirected through a POSIX locale with `use locale`.

Fix: do not use `use locale` for text processing. Unicode case folding is the right answer for text; POSIX locales are a remnant of byte-oriented C I/O. The only modern use of `use locale` is interacting with C libraries that consult `LC_CTYPE`.

Identifier vs data

Source code that contains non-ASCII characters needs `use utf8`. A program that processes non-ASCII data needs `binmode` or `:encoding(UTF-8)` layers on its handles. These are separate configurations, and both may be needed.

A short checklist for a new Unicode-aware script:

```
use strict;
use warnings;
use utf8;                                     # source is UTF-8
use open ":std", ":encoding(UTF-8)";         # stdio + default I/O
```

With those four lines at the top, string literals, standard I/O, and any later `open` call already speak text. Remaining Unicode work is narrow and local - decoding a byte buffer from a socket, or attaching `:raw` to a handle that must carry binary data.

Recipes

The conceptual chapters cover the discipline; this page is the quick-reference appendix. Each recipe is a short, self-contained answer to a common Unicode task - the kind of thing the classic Perl Unicode cookbook reaches for a CPAN module to do, but which pperl's builtins and regex engine handle directly.

Every example here assumes your strings are already text - decoded on the way in, as the *Strings and encodings* chapter describes. The operations below all speak characters, so a binary string fed to them will count bytes and surprise you.

Case-insensitive sort and comparison with `fc`

The `fc` builtin returns the *foldcase* of a string - the same Unicode case folding the `/i` pattern modifier uses. Two strings are equal under case folding exactly when `fc` returns the same result for both, which makes `fc` the right tool for case-insensitive sorting and comparison.

```
use v5.36;      # or: use feature 'fc';
use utf8;

# sort case-insensitively
my @sorted = sort { fc($a) cmp fc($b) } @list;
```

(continues on next page)

(continued from previous page)

```
# both are true:
fc("tschüß") eq fc("TSCHÜSS");
fc("Σῖουφος") eq fc("ΣΙΣΥΦΟΣ");
```

Folding handles the cases that simple `lc` cannot: the German sharp `s` folds to `ss`, and Greek final sigma `ς` folds together with medial `σ`. Reach for `fc` rather than `lc` whenever the comparison is meant to ignore case in the full Unicode sense.

This is case folding only. It does not ignore accents, and it does not impose a locale's alphabetic order - for that you would need a collation library such as `Unicode::Collate`, which is not yet usable in `ppperl` (see the end of this page).

Normalising line breaks with `\R`

`\R` matches a single Unicode linebreak: the two-character CRLF grapheme, or any one of the vertical whitespace characters (LF, CR, form feed, vertical tab, next line, line separator, paragraph separator). It is the portable way to deal with text files that arrive from different operating systems.

```
my $text = "line one\r\nline two\rline three\n";
$text =~ s/\R/\n/g;           # normalise every linebreak to \n
```

After the substitution, `$text` uses a single `\n` everywhere regardless of how the input encoded its line endings. Use `\R` in place of an explicit `\r?\n` alternation: it covers the cases that the hand-written pattern forgets.

Matching a grapheme cluster with `\X`

A user-visible character - a base letter together with any combining marks - is a *grapheme cluster*. The `.` metacharacter matches one code point, which may be only part of such a cluster; `\X` matches one whole grapheme cluster.

```
use utf8;

# "é" written as e + COMBINING ACUTE ACCENT is two code points,
# but one grapheme.
my $str = "café\x{301}";      # c a f e + combining acute
$str =~ /^.{5}$/;             # matches - five code points
$str =~ /^X{4}$/;             # matches - four graphemes

# find a vowel plus any combining diacritics
$str =~ / (?=[aeiou]) \X /xi;
```

`\X` is the right unit whenever “character” means “what the reader sees” rather than “code point”. The recipes below build on it.

First N graphemes, reverse, and length by grapheme

These three tasks all reduce to applying `\X` instead of indexing by code point. They work correctly no matter which normalization form the string is in.

Grab the first five graphemes:

```
my ($first_five) = $str =~ /^ ( \X{5} ) /x;
```

Reverse a string by grapheme. Reversing by code point would detach a combining mark from its base letter and corrupt the text; collecting graphemes first keeps each cluster intact:

```
$str = join "", reverse $str =~ /\X/g;
```

Count the length of a string in graphemes. The word *brûlée* is six graphemes but may be up to eight code points, so `length` and a grapheme count can differ:

```
my $count = 0;
$count++ while $str =~ /\X/g;
```

Fine-tuning UTF-8 warnings

Perl splits UTF-8 warnings into three subclasses, so you can silence one kind of malformed-Unicode warning without silencing the others. `ppperl` recognises all three category names:

```
no warnings "nonchar";      # the forbidden non-characters
no warnings "surrogate";    # UTF-16 surrogate code points
no warnings "non_unicode";  # code points above 0x10_FFFF
```

Disable only the subclass you have a deliberate reason to allow - for example `no warnings "non_unicode"` in code that manipulates code points beyond the Unicode range on purpose - and leave the rest in force so that genuine encoding faults still surface.

Decoding @ARGV

Command-line arguments arrive as bytes. If a user passes a non-ASCII argument, you must decode it before treating it as text, exactly as you would decode any other input. The portable form maps decode over `@ARGV`:

```
use Encode qw(decode);
@ARGV = map { decode('UTF-8', $_, 1) } @ARGV;
```

After this line, each element of `@ARGV` is a text string and behaves under `length`, regex, and case operations in characters rather than bytes. The third argument 1 (the `Encode::FB_CROAK` check) makes a malformed argument a fatal error rather than a silent substitution, which is usually what you want for input you cannot re-request.

Custom property by code-point range

A `\p{...}` property can name a subroutine you define. When the regex engine meets an unknown property `\p{In_Foo}`, it calls `In_Foo`, which returns a list of code-point ranges (one `START\textEND` pair per line, in hexadecimal); the property then matches any character in those ranges.

```
sub In_Greek { return "0370\t03FF\n0780\t07BF\n" }

"\x{3b1}" =~ /\p{In_Greek}/;    # matches - alpha is in 0370..03FF
```

This is the portable way to define an ad-hoc character class once and reuse it by name across patterns, without spelling out the range in every regex.

Modules pending in pperl

A handful of recipes in the classic Perl Unicode cookbook reach for the Unicode helper modules that Perl provides as compiled extensions: `Unicode::Normalize` (NFC, NFD, NFKC, NFKD normalization), `Unicode::Collate` and `Unicode::Collate::Locale` (alphabetic and locale-sensitive sorting, accent-insensitive comparison), `Unicode::GCString` (grapheme-aware substr and print-column width), `Unicode::UCD` (programmatic character category and numeric-value lookup), and `Unicode::Line-Break` (line-breaking by Unicode rules).

These modules are not available in pperl yet: loading one today fails with dynamic loading not available in this perl. Until they are, the regex `\X` family above covers grapheme-aware length, reversal, and fixed-count extraction (though not print-column width), and there is no substitute here for normalization or locale collation. Named characters and user-defined properties, by contrast, work today, as the recipes above show.

- **Strings and encodings** - use `utf8`, the `Encode` module, what the internal representation is and is not.
- **I/O** - opening files with an encoding, standard streams, `binmode` and the `:encoding` layer.
- **Regex and properties** - matching Unicode in patterns, `\p{...}` property classes, the `/a /u /l /aa` modifiers.
- **Pitfalls** - “Wide character in print”, double encoding, locale interactions, and how to recognise each one from its symptom.
- **Recipes** - quick-reference one-liners for case-insensitive sorting with `fc`, normalising line breaks with `\R`, and working by grapheme cluster with `\X`.

4.14.4 Conventions

- Expected output appears as an inline `# ...` comment or directly below the code block.
- Code points are written with `\x{...}` hex escapes so the printed text matches what you see.
- Examples assume a UTF-8-capable terminal. If yours is not, output containing non-ASCII characters will look different, but the program is still correct.

4.15 Boolean Logic for Perl Programmers - a tutorial

Almost every line of working Perl is shaped by boolean logic. `open ... or die, $h{$k} != [], unless ($a == $b && $c eq $d), /^(yes|no)$/, $flags & 0x4` - they are all the same handful of operations applied in different domains. This tutorial walks through those operations from the bottom up: what truth means in Perl, what the operators actually return, all sixteen binary truth functions and how to spell each one in Perl, the two laws that turn convoluted conditions into simple ones, and the three places that logic shows up most: bitwise arithmetic, regular expressions, and control flow.

4.15.1 Who this is for

You already write working Perl. You use `&&`, `||`, `if`, `unless` without thinking - but when a condition grows past three or four clauses you stop trusting it, and you reach for parentheses you suspect are not needed. This tutorial closes that gap. After reading it, you will know exactly which logical operation a given expression performs, why two seemingly different forms are the same expression, and how to reduce a tangled `unless` to a clean `if` with confidence rather than by trial and error.

4.15.2 Scope

Classical two-valued boolean logic - true and false, full stop. Perl's `undef` introduces a third value (the *unknown*) and a whole family of three-valued idioms (`defined`, `//`, `//=`). Those are mentioned where they intersect with two-valued logic, but a proper treatment of three-valued logic would double the size of this tutorial without finishing it; that is a separate guide.

4.15.3 How this tutorial is organised

Each chapter is one concept. Read them in order on the first pass; jump to the relevant chapter later when you are looking up a fact.

Truthiness

Before logic, the values logic operates on. Perl does not have a dedicated boolean type. Every scalar is either *true* or *false* when used in boolean context, and the rule is short:

A scalar is **false** if it is one of the four values `undef`, the integer `0`, the string `"`, or the string `"0"`. Everything else is **true**.

Four false values, every other scalar - including the strings `"00"`, `"0.0"`, `"0E0"`, and `"false"` - is true.

```
if (0)           { ... } # false -- the integer zero
if (0.0)         { ... } # false -- numerically zero
if ("")          { ... } # false -- empty string
if ("0")         { ... } # false -- the string "0"
if (undef)       { ... } # false -- undefined value

if ("00")        { ... } # TRUE  -- non-empty, not the string "0"
if ("0.0")       { ... } # TRUE  -- non-empty, not the string "0"
```

(continues on next page)

(continued from previous page)

```

if ("0 but true") { ... } # TRUE -- non-empty, not the string "0"
if (" ")          { ... }  # TRUE  -- single space is non-empty
if ("false")      { ... }  # TRUE  -- it's just a 5-character string

```

The "0 but true" idiom is not a joke. Some Perl APIs return a value that must compare equal to zero numerically (so an arithmetic test does the right thing) but be true in a boolean context. The classical case is `ioctl()` - success returning literally zero would test false. The string "0 but true" is numerically zero (with a warning suppressed by historical agreement) and boolean-true because it is not in the four-element false set.

```

my $rc = ioctl($fh, $cmd, $buf);
$rc == 0      and warn "numeric zero";    # would fire on success
$rc          or warn "boolean false";    # FIRES if API returned 0
                                           # but NOT if "0 but true"

```

Two flavours of comparison

Perl has parallel families of comparison operators because it has parallel notions of equality. Numbers compare numerically; strings compare lexically. The wrong family on the wrong type is one of the most common bugs in beginner Perl.

Test	Numeric	String
equal	<code>==</code>	<code>eq</code>
not equal	<code>!=</code>	<code>ne</code>
less than	<code><</code>	<code>lt</code>
greater than	<code>></code>	<code>gt</code>
less or equal	<code><=</code>	<code>le</code>
greater or equal	<code>>=</code>	<code>ge</code>
three-way (sort)	<code><=></code>	<code>cmp</code>

`==` on strings always succeeds for any two non-numeric strings - both convert to 0, and `0 == 0` is true:

```

"foo" == "bar";    # TRUE  -- both stringify to numeric 0
"foo" eq "bar";    # FALSE -- the actual question

"10" eq 10;        # TRUE  -- string "10" equals numeric 10
↳stringified
"10" == 10;        # TRUE  -- 10 == 10 numerically
"10" lt "9";       # TRUE  -- lexical: "1" < "9"
"10" < "9";        # FALSE -- numeric: 10 > 9

```

`<=>` and `cmp` return -1, 0, or +1 - the standard sort comparator shape. They are not boolean operators in the strict sense, but they appear inside boolean contexts often enough that you should know them.

Why this matters for logic

Every boolean operator in Perl - `&&`, `||`, `//`, `!`, `xor`, `and`, `or`, `not` - uses the four-value false set above as its input rule. Whenever a chapter below says “if `$x` is false”, that is exactly what is meant: `$x` is `undef`, `0`, `""`, or `"0"`. Whenever it says “if `$x` is true”, `$x` is anything else.

Two consequences worth keeping in mind:

1. **A scalar can be true without being a boolean true value.** Strings, references, blessed objects, large numbers - all true. When you ask “is this true?” you are not asking “does this equal 1?” - you are asking “is this not in the false set?”.
2. **The result of a boolean operator is not always 1 or 0.** The next chapter covers this in detail. Briefly: `&&` and `||` return one of their *operands*, not a normalised boolean. This is a feature, not a quirk, and many Perl idioms depend on it.

See also

- [perlop - Numeric comparison](#) and [String comparison](#) - the reference pages for `==/eq` and the rest of the comparison families introduced above.

Operators

Perl spells the boolean operators twice, in two different precedence classes. The C-style symbolic forms (`&&`, `||`, `!`) bind tightly; the word forms (`and`, `or`, `not`) bind very loosely. There is also `//` for the *defined-or* idiom, `xor` for exclusive-or, and the ternary `?:` which is logical selection in disguise.

The full inventory

Operator	Reads as	Returns	Precedence
<code>!</code>	not	normalised 1 or empty string	high
<code>&&</code>	and	left if false, else right (the <i>operand</i>)	high
<code> </code>	or	left if true, else right (the <i>operand</i>)	high
<code>//</code>	defined-or	left if defined, else right (the <i>operand</i>)	high
<code>xor</code>	exclusive or	normalised 1 or empty string	very low
<code>not</code>	not	normalised 1 or empty string	very low
<code>and</code>	and	left if false, else right (the <i>operand</i>)	very low
<code>or</code>	or	left if true, else right (the <i>operand</i>)	very low
<code>?:</code>	ternary	middle if condition true, else right (operand)	low

Two things in this table do most of the work:

- **`&&` and `||` return an operand, not a boolean.** This is the property that makes Perl’s idioms work the way they do. See below.
- **The symbolic and word forms differ in precedence, not meaning.** `&&` and `and` compute the same logical function. They differ in how tightly they bind to other operators around them. That is why you see `or die` after function calls but `||` inside expressions - the chapter on the `or die` idiom explains.

Short-circuit and the operand-return rule

&& and | | evaluate left-to-right and stop as soon as the result is determined.

- A && B - if A is false, return A without evaluating B. Otherwise return B.
- A || B - if A is true, return A without evaluating B. Otherwise return B.

Crucially, the **return value is the operand itself**, not a normalised true/false. This is what makes patterns like `$config{port} || 8080` work: if `$config{port}` is set (truthy), the expression is `$config{port}`'s value; if not, it is `8080`.

```
my $port = $config{port} || 8080;
my $name = $user_input || "anonymous";
my $list = shift || [];
```

The same property powers or die:

```
open my $fh, '<', $path or die "open $path: $!";  
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^  
# if open returns true,   the right-hand operand  
# the whole expression   is never evaluated  
# is true and we move on
```

open returns true on success, undef on failure. On success the short-circuit stops on the left and die never runs. On failure the right-hand operand evaluates, which is the die call.

// - defined-or

|| treats 0, "", and "0" the same as undef. Sometimes that is wrong: a configured 0 is a real value, not a missing one. // short-circuits on **definedness** instead of truthiness:

- `A // B` - if A is defined, return A. Otherwise return B.

```
my $port      = $config{port} || 8080;      # 0 means "use default" - w
↪wrong
my $port      = $config{port} // 8080;      # 0 means "0", undef means
↪"use default"
my $verbose   = $opt{verbose} // 0;          # 0 is a real choice
```

The compound assignment forms `| |=` and `||=` save you from repeating the variable on the left:

```
$config{port} ||= 8080; # set only if currently undef
$cache{$key} ||= compute($key); # set if currently false
```

xor and the absence of a high-precedence form

There is no `^^` in Perl. `xor` exists only as the very-low-precedence word form, and unlike `and/or` it does *not* short-circuit (it cannot - both sides have to be evaluated to determine the result). It returns a normalised 1 or "", not an operand.

```
if ($admin xor $guest) { ... } # exactly one of the two
```

For xor on values rather than on truthiness, the bitwise ^ applies to integers - covered in *the applications chapter*.

! and not

Both negate. ! is the symbolic high-precedence form; not is the word low-precedence form. Both return a normalised value: 1 for “the operand was false”, empty string "" for “the operand was true”.

```
my $missing = ! $config{port}; # 1 if port is unset/0/"/"/"0"
return if not @items;         # cleaner reading than `if !
    ↳@items`
```

The ternary ? :

COND ? X : Y is logical selection: if COND is true the value is X, otherwise Y. It is not just sugar for an if/else because it is an *expression* - it has a value, can be assigned, returned, passed as an argument.

```
my $label = $count == 1 ? "1 item" : "$count items";
return $ok ? \%result : undef;
```

Beware nesting: \$a ? \$b : \$c ? \$d : \$e parses as \$a ? \$b : (\$c ? \$d : \$e) - chained ?: is right-associative, which is usually what you want for an if/elsif/else cascade but is worth knowing rather than guessing.

Why the word forms exist: precedence, not style

|| binds more tightly than =. or binds less tightly than = (or ,, or anything else of conventional weight). That is the entire reason the word forms exist.

```
my $fh = open $h, '<', $path || die "no $path: $!";
#
# parses as: open $h, '<', ($path || die "no $path: $!")
# which is: open the file named "either $path or the die-message"
# which is: catastrophe.

my $fh = open $h, '<', $path or die "no $path: $!";
# parses as: ($fh = open $h, '<', $path) or die "no $path: $!"
# which does what you meant.
```

Rule of thumb:

- Inside an expression, where you want the value to flow into a variable or another operator: use &&, ||, !, //.
- After a statement, where you want a side-effect (die, warn, return) to fire conditionally: use or, and, not.

What you should remember from this chapter

- `&&` and `||` are not predicates that return 1 or 0 - they return one of their operands. This is the foundation of `||`, `||=`, `or die`, and most short defaulting idioms.
- `//` is `||` for definedness, not truthiness. Use it when `0` is a legitimate value.
- The word forms exist to bind looser than assignment, not for style.
- `xor` exists, has no symbolic form, does not short-circuit.

See also

- [perlop - Logical](#) - the reference companion, with the precedence-row context and PetaPerl-specific notes.
- [perlop - Precedence](#) - the full table; rows 5, 15, 16, 22, 23, 24 are the operators on this page.
- [perlop - Ternary](#) - `?:` as logical selection.

Truth tables

A binary boolean function takes two boolean inputs and returns one boolean output. There are exactly **sixteen** such functions - that is $2^{(2^2)}$: each of the four input combinations $(0,0)$ $(0,1)$ $(1,0)$ $(1,1)$ is independently mapped to one of two outputs, so $2 \times 2 \times 2 \times 2 = 16$. No more, no fewer.

The five with their own Perl operator (`&&`, `||`, `xor`, plus the two trivial ones - always-true and always-false) are the ones everyone knows. The other eleven are quietly there too, expressed as small compositions. This chapter lists all sixteen, names them, and shows how to write each one in Perl.

The convention

Each function is fully described by its output for the four input rows, written top-to-bottom in this order:

a	b	→	output
0	0	→	bit 1
0	1	→	bit 2
1	0	→	bit 3
1	1	→	bit 4

Reading those four output bits as a 4-bit number gives the function its index 0..15. So AND is 0001 (only the last row is true), OR is 0111, XOR is 0110. Three quarters of the table is just looking up the right row.

All sixteen

#	Truth 00 01 10 11	Name	Symbol	Perl di- rect	Perl idiomatic
0	0 0 0 0	false (contra- diction)	$\perp$	0 / ""	-
1	0 0 0 1	AND	$\wedge$	<code>\$a && \$b</code>	<code>\$a and \$b</code>
2	0 0 1 0	a AND NOT b	$a \wedge \neg b$	-	<code>\$a && !\$b</code>
3	0 0 1 1	a (left projec- tion)	a	<code>\$a</code>	-
4	0 1 0 0	NOT a AND b	$\neg a \wedge b$	-	<code>!\$a && \$b</code>
5	0 1 0 1	b (right pro- jection)	b	<code>\$b</code>	-
6	0 1 1 0	XOR	$\oplus$	<code>\$a xor \$b</code>	<code>!\$a != !\$b / (\$a?1:0) != (\$b?1:0)</code>
7	0 1 1 1	OR	$\vee$	<code>\$a \$b</code>	<code>\$a or \$b</code>
8	1 0 0 0	NOR	$\downarrow$ (Peirce)	-	<code>!(\$a \$b) / !\$a && !\$b</code>
9	1 0 0 1	XNOR / equiv- alence	$\leftrightarrow$	-	<code>!\$a == !\$b / (\$a?1:0) == (\$b?1:0)</code>
10	1 0 1 0	NOT b	$\neg b$	<code>!\$b</code>	<code>not \$b</code>
11	1 0 1 1	b implies a	$b \rightarrow a$	-	<code>!\$b \$a</code>
12	1 1 0 0	NOT a	$\neg a$	<code>!\$a</code>	<code>not \$a</code>
13	1 1 0 1	a implies b	$a \rightarrow b$	-	<code>!\$a \$b</code>
14	1 1 1 0	NAND	$\uparrow$ (Shef- fer)	-	<code>!(\$a && \$b) / !\$a !\$b</code>
15	1 1 1 1	true (tautol- ogy)	$\top$	1	-

Five rows have a direct Perl operator (&&, ||, xor, !, plus the two constants). The other eleven are short compositions of those. There is no fundamental reason Perl spells some directly and not others - it is historical, inherited from C. Every column on the right is something you can type today.

Two things are worth pulling out of the table.

Implication: $\rightarrow$

Implication is the operator nobody thinks they use, and everyone uses constantly. $a \rightarrow b$ is read “a implies b” and is true in every case **except** when a is true and b is false. That is it.

a	b	$a \rightarrow b$
0	0	1
0	1	1
1	0	0
1	1	1

The Perl spelling is `!$a || $b`. This identity -

$$a \rightarrow b \equiv \neg a \vee b$$

- is so common that it deserves a name; and the name is *material implication*. Once you see it once, you spot it everywhere:

```
# "if the user is admin, the request must be authenticated"
# admin → authenticated
die unless !$user->is_admin || $req->is_authenticated;

# read aloud: "either not admin, or authenticated, or both"
```

The `unless ... !X || Y` shape is awkward. The next chapter on De Morgan's laws shows how to flip it into something readable.

A second useful identity: implication is **the negation of XOR only when written one specific way**. Don't memorise that - memorise this instead, which is the actual lookup table for flipping between common shapes:

Goal	One way	Equivalent way
$a \rightarrow b$	$\neg a \vee b$	$\neg(a \wedge \neg b)$
$a \leftrightarrow b$ (XNOR)	$(a \wedge b) \vee (\neg a \wedge \neg b)$	$\neg(a \oplus b)$
$a \oplus b$ (XOR)	$(a \wedge \neg b) \vee (\neg a \wedge b)$	$\neg(a \leftrightarrow b)$
$\neg a$	$a \uparrow a$ (NAND self)	$a \oplus 1$

The last column is a teaser for the chapter on functional completeness.

Equivalence and XOR are duals

XOR is true exactly when *a* and *b* differ. Equivalence (XNOR) is true exactly when they agree. They are negations of each other, and that is exactly what the truth-table rows say (row 6 is 0 1 1 0, row 9 is 1 0 0 1 - bitwise negation).

In Perl the cleanest spelling for both is to canonicalise each operand to 0 or 1 first and then compare with `==` or `!=`:

```
my $a01 = $a ? 1 : 0;
my $b01 = $b ? 1 : 0;

my $xor  = $a01 != $b01;  # true if they differ
my $xnor = $a01 == $b01;  # true if they agree
```

Why the canonicalisation? Because `$a` and `$b` may be any truthy values - the strings "yes" and 5 are both true but not equal. Comparing them directly with `==` or `eq` answers a different question. Boolean XOR asks “do they have the same *truthiness*?”, not “are they equal?”.

A short alternative without the temporaries is `!$a != !$b` - each `!` returns the canonical 1 or "", and then `!=` compares those. It works but reads sideways; the explicit `? 1 : 0` form is more readable.

Reading the table backwards

You will sometimes know the truth pattern you want and need to find the operator. Example: “I want a function that is true unless both `a` and `b` are false.” Read the rows: `a=0,b=0 → 0`, `a=0,b=1 → 1`, `a=1,b=0 → 1`, `a=1,b=1 → 1`. That is 0 1 1 1 - row 7 - OR.

This sounds trivial here but is exactly how you debug a tangled condition: write down its truth table, read off the four bits, find the row, and you have the canonical name and the simplest spelling.

What you should remember from this chapter

- There are exactly sixteen binary truth functions; every boolean expression in any language computes one of them.
- Five of the sixteen have a dedicated Perl operator. The other eleven are short compositions and the table above is the lookup.
- Implication `a → b` is `¬a ∨ b`. It shows up constantly under the surface of `unless` and `if` clauses.
- XOR and XNOR are negations of each other. The cleanest Perl spelling for both starts by canonicalising operands to 0 or 1.

De Morgan’s laws

Two identities, due to Augustus De Morgan, written for over a century in every textbook of formal logic, and the single most useful pair of equations a working programmer can know.

$$\neg(\mathbf{a} \wedge \mathbf{b}) \equiv \neg\mathbf{a} \vee \neg\mathbf{b}$$

$$\neg(\mathbf{a} \vee \mathbf{b}) \equiv \neg\mathbf{a} \wedge \neg\mathbf{b}$$

In words: pushing a not through a parenthesised expression flips every operator inside ($\wedge$ becomes $\vee$, $\vee$ becomes $\wedge$) and negates each operand. Equivalently: pulling a not out flips every operator and removes the negations on the operands.

Why this matters in Perl

`unless` is the canonical case. `unless (X)` is `if (!X)` - and the moment `X` is itself a compound, you have a leading not over a parenthesised expression: exactly the De Morgan setup.

```
unless ($a == $x && $b == $y) { ... }
```

The condition under the `unless` is `!($a == $x && $b == $y)`. Apply De Morgan: this is `!($a == $x) || !($b == $y)`. Each `!=` is the negation of the corresponding `==`, so the clean form is:

```
if ($a != $x || $b != $y) { ... }
```

Three transformations happened, each mechanical:

1. `unless` flipped to `if` (the outer not).
2. `&&` flipped to `||` (De Morgan, inside).
3. `==` flipped to `!=` on each operand (the negations De Morgan placed on the operands cancel into the opposite comparison).

This is *not* a stylistic preference. Every programmer who has debugged a wrong condition has stared at `unless (! ... && ...)` for a minute too long; the flat `if` reads in one pass.

A worked example

A real shape, slightly tangled:

```
unless ($x !~ /\d+$/ && $y !~ /[a-z]+$/) {
    print "at least one of $x and $y looks valid\n";
}
```

This says: *unless* `$x` is non-numeric and `$y` is non-alphabetic, print. Five negations stacked across two lines. Walk through it:

The condition under the `unless` is

```
!( $x !~ /\d+$/ && $y !~ /[a-z]+$/ )
```

De Morgan flips the `&&` to `||` and negates each side:

```
!($x !~ /\d+$/) || !($y !~ /[a-z]+$/)
```

`!($x !~ ...)` is just `$x =~ ...` (double negation of a regex match). Same for the right side:

```
$x =~ /\d+$/ || $y =~ /[a-z]+$/
```

So the cleaned-up version is:

```
if ($x =~ /\d+$/ || $y =~ /[a-z]+$/) {
    print "at least one of $x and $y looks valid\n";
}
```

Five negations down to zero. The condition now reads exactly the way the comment said it: *if* `$x` is numeric or `$y` is alphabetic.

The same trick for OR

The second form of the law is the symmetric case:

```
unless ($a == 0 || $b == 0) { divide($a, $b) }
```

becomes

```
if ($a != 0 && $b != 0) { divide($a, $b) }
```

- same mechanical steps, only this time the inner operator is `||` flipping to `&&`.

NAND, NOR - same identity, read the other way

The NAND row in the truth table from the previous chapter has two entries in the “Perl idiomatic” column:

```
14  NAND    !($a && $b)    /    !$a || !$b
```

These are equal *because of De Morgan’s first law*. NAND is exactly the negation of AND, and pushing the negation in flips the operator and negates the operands. The NOR row works the same way for the second law:

```
8   NOR     !($a || $b)    /    !$a && !$b
```

So De Morgan is not a separate fact you have to remember alongside NAND and NOR - it is the identity that turns one spelling into the other.

Mechanics, in one paragraph

Negation distributes over $\wedge$ and $\vee$ by flipping the connective and negating each operand. Negation also self-cancels: $\neg\neg x \equiv x$. The two rules together let you push a not arbitrarily deep, or pull it arbitrarily out, of any expression built from $\wedge$, $\vee$, and $\neg$. That is the whole content of De Morgan’s laws - every “convoluted unless” you ever rewrite is one application of those two rules plus double-negation cancellation.

Practical procedure

When you stare at a condition you don’t like:

1. If the outer keyword is `unless`, prepend a `not` mentally and make it `if`.
2. If the leading `not` is over a parenthesised `&&` or `||`, distribute it: flip the connective, negate each operand.
3. If a subexpression is `!(x == y)`, replace with `x != y`. Same for `!~` $\leftrightarrow$ `=~`, `!defined` $\leftrightarrow$ `defined`, etc.
4. Stop when there are no more leading negations to push or no more double-negations to cancel.

In practice you will do all four steps in your head in a single pass. The point of the procedure is that there is one - you are not searching for the right form, you are deriving it.

What you should remember from this chapter

- Two identities: pushing $\neg$ through $\wedge$ flips it to $\vee$ (and vice versa) and negates each operand.
- Most “ugly unless” is one mechanical De Morgan flip away from a clean `if`.
- NAND and NOR are not extra facts; they are De Morgan’s laws read as a pair of equivalent spellings.

See also

- *perlop - Logical* - the operator-reference companion. The De Morgan transforms here apply to every Perl expression that uses `&&`, `||`, `!`, and the `unless` keyword.

Functional completeness

A set of boolean operators is *functionally complete* when every one of the sixteen binary truth functions can be built from operators in that set, by composition. The classical complete sets you already know are $\{\wedge, \vee, \neg\}$ and $\{\rightarrow, \perp\}$. The surprising - and useful - result is that **a single operator suffices**, but only two of the sixteen binary functions have this property: NAND and NOR.

NAND is named after Henry M. Sheffer (1913) and written $\uparrow$; NOR is named after Charles Sanders Peirce (who described it earlier) and written $\downarrow$. They are the two *Sheffer strokes*.

Building everything from NAND

NAND alone gives you all sixteen functions. The constructions below show the four most useful - NOT, AND, OR, XOR - in terms of $\uparrow$. Think of `a  $\uparrow$  b` as `!($a && $b)`.

Target	In terms of NAND	Why
$\neg a$	<code>a $\uparrow$ a</code>	<code>!(a $\wedge$ a)</code> is <code>!a</code>
$a \wedge b$	<code>(a $\uparrow$ b) $\uparrow$ (a $\uparrow$ b)</code>	NAND then negate the result, i.e. <code>!!(a $\wedge$ b)</code> $\equiv a \wedge b$
$a \vee b$	<code>(a $\uparrow$ a) $\uparrow$ (b $\uparrow$ b)</code>	De Morgan: <code>a $\vee$ b $\equiv \neg(\neg a \wedge \neg b) \equiv \neg a \uparrow \neg b$</code>
$a \oplus b$	<code>(a $\uparrow$ (a $\uparrow$ b)) $\uparrow$ (b $\uparrow$ (a $\uparrow$ b))</code>	XOR is the AND of <code>(a $\vee$ b)</code> and <code>!(a $\wedge$ b)</code> - left as exercise

The same constructions exist with NOR; they are the De Morgan duals.

In Perl

You will not write your code this way in production - `(a  $\uparrow$  a)  $\uparrow$  (b  $\uparrow$  b)` is a worse spelling of OR than `||` is. But it is worth seeing the constructions actually run, because the abstract claim (“everything reduces to one operator”) becomes concrete when you type it.

```

sub nand { !($_[0] && $_[1]) }           # the only primitive

sub not_a { nand($_[0], $_[0]) }         # ¬a
sub and_  { my $n = nand($_[0], $_[1]); nand($n, $n) } # a ∧ b
sub or_   { nand(nand($_[0], $_[0]), nand($_[1], $_[1])) } # a ∨ b
sub xor_  {
    my ($a, $b) = @_;
    my $n = nand($a, $b);
    nand(nand($a, $n), nand($b, $n));
}

# Verify across all four input combinations:
for my $a (0, 1) {
    for my $b (0, 1) {
        printf "a=%d b=%d  not=%d and=%d or=%d xor=%d\n",
            $a, $b,
            not_a($a)           ? 1 : 0,
            and_($a, $b)       ? 1 : 0,
            or_($a, $b)        ? 1 : 0,
            xor_($a, $b)       ? 1 : 0;
    }
}

```

Output:

```

a=0 b=0  not=1 and=0 or=0 xor=0
a=0 b=1  not=1 and=0 or=1 xor=1
a=1 b=0  not=0 and=0 or=1 xor=1
a=1 b=1  not=0 and=1 or=1 xor=0

```

Each row matches the truth table from the previous chapter.

Why this matters

Three things, in order of practical-to-deep:

1. **It explains the bestiary of identities.** NAND, NOR, and the eleven non-direct rows of the truth-table chapter are not a zoo of separate facts. They are all expressible from a single operator; the eleven non-direct spellings in Perl are just readable shortcuts for short NAND or NOR compositions.
2. **It is how silicon actually works.** Static-CMOS logic produces NAND and NOR more cheaply than AND and OR - a two-input AND gate is built as a NAND followed by an inverter. Every other gate on a chip is a composition of NANDs (or NORs). The pattern repeats in solver and constraint-engine internals: SAT solvers normalise to CNF precisely because OR-of-(literals) and AND-of-(clauses) is the minimal vocabulary they need.
3. **It teaches the discipline of canonical form.** Every boolean expression has many equivalent shapes, but only a few are *canonical*. CNF (conjunctive normal form: an AND of ORs of possibly-negated literals) and DNF (disjunctive normal form: an OR of ANDs) are the two most common. Reducing a tangle to CNF or DNF is a pure me-

chanical procedure based on De Morgan plus distributivity, and once your expression is in normal form you can compare it with another normal-form expression to test equivalence - without truth tables, without testing.

You will not implement a SAT solver next week. But the next time you find yourself writing `if ((!$a && $b) || (!$c && $b) || ($a && !$c))` and wondering whether the previous version was equivalent, you have a tool: distribute, factor, normalise. Two expressions that reduce to the same DNF compute the same function.

What you should remember from this chapter

- NAND and NOR are each individually sufficient to express every boolean function. No other two-input operator has this property.
- The constructions are short and you can verify them in Perl in ten lines.
- The point is not to write code in NAND, but to know that all sixteen binary functions live on one foundation. The apparently disparate rows of the truth-table chapter are one family.

Applications

The previous chapters were about boolean logic in the abstract. This one is about where it shows up most often in working Perl: bitwise arithmetic, regular expressions, and the short-circuit idioms that take the place of explicit conditionals.

Bitwise: logic on integer bits

The bitwise operators apply boolean operations to each pair of bits in two integers in parallel. A 32-bit integer is, from the operators' point of view, thirty-two parallel one-bit values.

Op	Reads as	Per-bit rule
<code>&</code>	bitwise AND	each output bit = $a \wedge b$
<code> </code>	bitwise OR	each output bit = $a \vee b$
<code>^</code>	bitwise XOR	each output bit = $a \oplus b$
<code>~</code>	bitwise NOT	each output bit = $\neg a$
<code><<</code>	left shift	shift bits left, zero-fill from right
<code>>></code>	right shift	shift bits right

The same boolean operators - $\wedge$, $\vee$, $\oplus$, $\neg$ - appear here in purely numeric form. That is not a coincidence; it is the definition.

Setting, clearing, toggling, testing a flag

The four basic flag operations on a single bit:

```
use constant FLAG_VERBOSE => 0x01;
use constant FLAG_DRY_RUN => 0x02;
use constant FLAG_RECURSE => 0x04;
```

(continues on next page)

(continued from previous page)

```

use constant FLAG_FORCE    => 0x08;

my $flags = 0;

$flags |= FLAG_VERBOSE;      # SET    -- OR with the bit
$flags |= FLAG_DRY_RUN;      # SET another
$flags &= ~FLAG_DRY_RUN;      # CLEAR  -- AND with the inverted bit
$flags ^= FLAG_VERBOSE;      # TOGGLE -- XOR with the bit
my $on = $flags & FLAG_RECURSE; # TEST  -- AND, then test truthiness

```

Each of these is a one-bit application of a boolean operator. Set is $\text{bit} \vee \text{flag}$, clear is $\text{bit} \wedge \neg \text{flag}$, toggle is $\text{bit} \oplus \text{flag}$, test is $\text{bit} \wedge \text{flag}$.

XOR-swap: swapping without a temporary

XOR has two properties that combine into a memorable trick: $a \oplus a = 0$, and $a \oplus b \oplus b = a$. Apply them in sequence and you can swap two integers without a third variable:

```

my ($a, $b) = (0xFEED, 0xBEEF);

$a ^= $b;      # a := a ⊕ b
$b ^= $a;      # b := b ⊕ (a ⊕ b) = a
$a ^= $b;      # a := (a ⊕ b) ⊕ a = b

print "a=$a b=$b\n"; # a=48879 b=65261 (0xBEEF, 0xFEED)

```

The trick is not actually useful in Perl - $(\$a, \$b) = (\$b, \$a)$ is faster, clearer, and works on any scalar including strings and references. But it shows up in two places that matter:

- **Embedded code without a free register.** A microcontroller with three values to juggle in two registers reaches for this.
- **Interview folklore.** Knowing it exists and *why* it works (XOR is its own inverse) is worth the thirty seconds it takes to read.

The reason it works is exactly the boolean identity from the truth-table chapter: $x \oplus y \oplus y = x$. Each of the three lines above is one application of that identity.

Common bit tricks

A handful of patterns you will see in performance-sensitive code:

```

$x & ($x - 1)      # $x with its lowest set bit cleared
($x & ($x - 1)) == 0 # true when $x is a power of two (and non-
→zero)
$x | -$x           # signed: zero iff $x was zero, non-zero
→otherwise
($x >> 31) & 1     # the sign bit, on a 32-bit signed integer

```

(continues on next page)

(continued from previous page)

```
1 << $n          # the integer with only bit $n set
$x & (1 << $n)   # is bit $n set in $x?
```

Each of these is an exercise in tracking what the bits do - pure boolean reasoning applied 32 (or 64) times in parallel.

Regular expressions: logic on sets of strings

A regex matches a *set* of strings. The empty set, the singleton {"foo"}, the infinite set "anything that begins with a digit" - all are sets, and the regex denotes one of them.

Once you see a regex as a set, the boolean operations have geometric meaning:

Boolean operation	On sets	In regex syntax
OR ($\vee$)	union	alternation: <code>foo bar</code>
AND ($\wedge$)	intersection	lookahead pair: <code>(?=.{4})(?=\d+\$)</code>
NOT ($\neg$)	complement	negative lookahead: <code>(?!foo)</code>
AND-NOT ($a \wedge \neg b$)	difference	<code>(?=A)(?!B)A</code>

Two specifics worth pulling out.

Alternation is OR over sets

```
$s =~ /yes|no|maybe/;
```

Matches the union of three singleton sets. There is nothing more to it; | in regex syntax is precisely the boolean $\vee$.

Inside a character class the meaning is the same:

```
$s =~ /[abc]/;      # union of {"a"}, {"b"}, {"c"}
$s =~ /^[^abc]/;    # complement: anything NOT in {"a","b","c"}
```

`[^...]` is the regex syntax for $\neg$ applied to a character set.

Lookarounds are AND and NOT

A regex without lookahead consumes characters as it matches; the two ends of an alternation `A|B` cannot both match the same span (only one branch wins). To express *both* *A* and *B* at the same position, you need lookahead: a zero-width assertion that demands a property without consuming.

```
# matches only if the rest of the string is BOTH digits AND ≤ 4
↪ chars
$s =~ /^(?=\d+$(?=.{1,4}$))/;
#           \_____/ \_____/
#           A         B           intersection: A ∧ B
```

(?=...) is positive lookahead; (?!...) is negative lookahead (the boolean NOT). Combining them gives you the full set algebra on regex predicates:

```
# "starts with a digit but is not the literal '0':  
# (digit) ^ ~(literal "0")  
$s =~ /^(?=\d)(?!0$)/;
```

Why the framing helps

Once you read regex this way, the readability of complicated patterns improves dramatically. A regex with two lookaheads is not “two regexes glued together” - it is the intersection of two sets. A negative lookahead followed by a positive match is not “first reject, then match” - it is set difference. The boolean algebra you already know is the algebra of regexes; only the notation changes.

Control flow: short-circuit logic as conditional execution

The chapter on operators introduced the operand-return rule for &&, ||, and //. That rule, plus precedence, generates a small family of idioms that replace explicit if/else for short conditional logic:

```
# defaulting  
my $port = $cfg{port} // 8080;  
  
# guard with side-effect  
open my $fh, '<', $path or die "open $path: $!";  
  
# lazy initialisation (set if currently false)  
$cache{$key} ||= compute($key);  
  
# lazy initialisation (set if currently undef - a real `0` is kept)  
$cfg{retries} //= 3;  
  
# guard chain: do step 2 only if step 1 succeeded  
my $rc = step_one() && step_two();  
  
# logical selection inside an expression  
my $label = $count == 1 ? "1 item" : "$count items";
```

Each of these is a boolean expression chosen for its *side-effects*

- the short-circuit decides whether the right operand even runs. `open ... or die` works exactly because or does not evaluate its right side when the left is true.

Two pieces of advice.

Use // and //= when 0 or "" is a valid value. This single change has prevented more bugs than any other modern Perl idiom; || defaulting on a configured zero is one of the classic ways to lose data.

Reach for ?: when you would otherwise build an if/else/scalar-assign triplet. The ternary keeps the expression as an expression and the assignment in one line:

```
# verbose
my $kind;
if ($n == 1) { $kind = 'singular' }
else        { $kind = 'plural'   }

# idiomatic
my $kind = $n == 1 ? 'singular' : 'plural';
```

Don't nest `?:` more than two levels deep. Past two, the chained form is harder to read than the `if/elsif/else`. The ternary is for *choosing*; sequential decisions belong in a block.

What you should remember from this chapter

- Bitwise `&`, `|`, `^`, `~` are the boolean operators applied bit-by-bit in parallel; `flag set/clear/toggle/test` are the four one-bit applications.
- A regex denotes a set of strings; `|` is union, `[^...]` is complement, `(?=)` and `(?!)` make intersection and difference.
- `&&`, `||`, `//`, `?:` give you most of conditional control flow without writing `if`. Use `//` when `0` is real; use `?:` for short selection inside an expression.

See also

- [perlop - Bitwise](#) - the reference companion to the bitwise section, with the full shift semantics and the suffixed `&.`, `|.`, `^.`, `~.` byte-string forms.
- [perlop - Binding](#) - `=~` and `!~`, the operators that connect strings to regex matching.
- [Regular expressions guide](#)
 - the full regex language reference.
- [perlop - Logical](#) - the short-circuit operators behind the control-flow idioms.

4.15.4 What you will be able to do after reading

- Look at any boolean expression in Perl source and name the truth function it computes.
- Convert between the four equivalent shapes of an implication ($a \rightarrow b$, $!a \mid\mid b$, $!(a \ \&\& \ !b)$, the contrapositive) without having to think.
- Reduce an `unless` with two negated subclauses to a flat `if` in one mechanical step.
- Recognise when a regex alternation is a logical OR over sets and when a lookahead pair is a logical AND, and use that framing to read complicated regexes.
- Use the bitwise operators to set, clear, toggle and test individual flags without consulting a reference.
- Pick the right short-circuit idiom (`||`, `//`, `||=`, `//=`, or `die`) for the situation, knowing why each one works.

4.16 Interprocess communication - a tutorial

Two programs that need to talk have a handful of channels to choose from: a pipe, a named FIFO on disk, a socket over the network, a socket pair in memory, or a slab of System V shared memory. This tutorial is a cookbook. Each page hands you a complete, runnable program - a producer and a consumer, or a client and a server - that you can copy, run, and adapt.

The mechanics of each channel - what `socket` returns, how `fork` splits a process, what `%SIG` does - are already documented in the per-built-in reference and the signals chapter. This tutorial does not restate them. It assembles them into working IPC programs and links each call back to its reference page.

PetaPerl runs only on Linux, so everything here is POSIX. There is no Win32 IPC, no named-pipe-on-Windows alternative, no `Win32::Process`. The recipes use the system calls a Linux kernel actually provides.

4.16.1 Choosing a channel

Pick the lightest channel that carries what you need:

- **A child running a command** - you want the output of sort or the input of gzip. Use a *pipe open*, or `IPC::Open2` / `IPC::Open3` when you need both directions. See *Subprocesses and co-processes*.
- **Two related processes, one machine** - a parent and a child it just forked, sharing a byte stream. Use *pipe* or *socketpair*. See *Subprocesses* and *UNIX-domain and UDP sockets*.
- **Unrelated processes, one machine** - two programs started independently that rendezvous on a known path. Use a *FIFO* or a UNIX-domain socket. See *FIFOs* and *UNIX-domain and UDP sockets*.
- **Processes on different machines** - a network client and server. Use a TCP socket. See *TCP sockets*.
- **A datagram, fire-and-forget** - no connection, message boundaries preserved. Use a UDP socket. See *UNIX-domain and UDP sockets*.
- **A shared counter or buffer with no copying** - System V semaphores and shared memory. Use *semget* and *shmget*. See *System V IPC*.

4.16.2 How this tutorial is organised

Each page stands on its own. Read the one that answers your question.

Subprocesses and co-processes

The most common IPC task is the simplest: run an external command and talk to it. This page covers the spectrum, from a one-way pipe open to a two-way co-process - a child you both write to and read from.

The mechanics of a one-direction pipe open ("`- |`" and "`| -`") are covered in full on the *Pipes* page. This page picks up where that one stops: when one direction is not enough, and when you need a raw *pipe* and *fork* of your own.

One direction: the pipe open

To read a command's output, or to feed a command its input, a pipe open is all you need:

```
open my $fh, "-|", "sort", "-u", "names.txt" or die "pipe: $!";
while (my $line = <$fh>) { ... }
close $fh;
```

That is the whole story for one-way work, and it is told on the [Pipes](#) page - including the list-versus-string form, why list form sidesteps the shell, and how to decode the child's exit status. Start there if one direction is enough.

Both directions: IPC::Open2

When you need to write to a command *and* read its reply - a filter you drive interactively, a co-process - one pipe is not enough. IPC::Open2 opens two: one into the child's stdin, one out of its stdout.

```
use IPC::Open2;

my $pid = open2(my $from_child, my $to_child, "tr", "a-z", "A-Z");

print $to_child "hello\n";
close $to_child;                # signal end-of-input to the
↳child

my $reply = <$from_child>;
print "got: $reply";            # got: HELLO

waitpid($pid, 0);
```

open2 returns the child's process id. The argument order is the trap worth memorising: **reader handle first, writer handle second**, then the command. open2(OUT, IN, ..) - the handle you read *from* comes before the handle you write *to*.

The deadlock you must design around

A co-process can deadlock, and IPC::Open2 does nothing to prevent it. The shape of the trap:

- You write a large block to the child.
- The child reads some, produces output, and that output fills the pipe back to you.
- The child blocks writing to a full pipe; you block writing to a child that has stopped reading. Neither moves.

The recipe above sidesteps it by closing \$to_child before reading - sending the whole input, then signalling end-of-file, then draining the reply. That works when the input fits comfortably in the pipe buffer. For unbounded streaming in both directions at once, you need non-blocking I/O or a select loop; the two-pipe shape alone is not enough.

Capturing standard error too: IPC::Open3

IPC::Open2 joins you to stdin and stdout. When you also need the child's stderr on its own handle, IPC::Open3 adds a third:

```

use IPC::Open3;
use Symbol qw(gensym);

my $err = gensym;                # a fresh anonymous handle for
    ↪ stderr
my $pid = open3(my $to_child, my $from_child, $err, "sort");

print $to_child "banana\napple\ncherry\n";
close $to_child;

my @sorted = <$from_child>;      # ("apple\n", "banana\n",
    ↪ "cherry\n")
print @sorted;                  # apple / banana / cherry, one
    ↪ per line

waitpid($pid, 0);

```

Two differences from open2 to note:

- **The argument order flips.** open3 takes the writer handle first, then the reader, then the error handle: open3(IN, OUT, ERR, ...). This is the opposite of open2's reader-first order - a long-standing inconsistency that bites everyone once.
- **The error handle needs gensym.** Pass a fresh anonymous handle from Symbol::gensym for stderr; a bareword or an undefined lexical will not collect it cleanly.

Rolling your own: pipe and fork

When the command is not an external program but Perl code you want to run in a separate process, build the channel yourself. A raw pipe gives a connected reader/writer pair before the fork; after fork each side keeps the end it needs and closes the other:

```

pipe(my $reader, my $writer) or die "pipe: $!";

my $kid = fork() // die "fork: $!";

if ($kid == 0) {
    # Child writes, then exits.
    close $reader;                # child has no use for the read
    ↪ end
    $writer->autoflush(1);
    print $writer "down-the-pipe\n";
    close $writer;
    exit 0;
}

```

(continues on next page)

(continued from previous page)

```
# Parent reads.
close $writer;                # parent has no use for the
    ↪ write end
my $line = <$reader>;
print "parent read: $line";    # parent read: down-the-pipe
close $reader;

waitpid($kid, 0);
```

The discipline is the same as the forking server on the *TCP sockets* page: **each side closes the end it does not use**. If the parent keeps the write end open, the child's close never produces an end-of-file on the read end, and the parent's read blocks forever.

Explicit `fork()` creates a separate operating-system process; PetaPerl's auto-parallelism is an in-process thread pool. A forked child runs its own auto-parallel numeric reductions normally within itself, and the parent's and child's parallelism do not interfere - they are independent processes with independent thread pools. For running work across CPU cores *inside* one process rather than across separate processes, see the *Concurrent Execution* guide.

Reference cross-links

- [open](#) - the one-way pipe open (" - |", " | -")
- [pipe](#) - a raw connected handle pair
- [fork](#) - split into two processes
- [waitpid](#) - reap the child and recover its status
- [exec](#) - replace the child image with a command

Next

For two unrelated programs that meet on a filesystem path rather than through a shared parent, see *FIFOs*.

FIFOs

A FIFO - a *named pipe* - is a pipe with a name on the filesystem. Two programs that share no parent, started independently from different shells, can still talk: one opens the path for writing, the other opens it for reading, and bytes flow between them exactly as through an anonymous [pipe](#). The path is a rendezvous point; no data is ever stored in the file itself.

This is the channel to reach for when the processes are *unrelated* - when a plain pipe is impossible because there was never a common fork to inherit the descriptor from.

Creating the FIFO

A FIFO is created once, as a special file, with `POSIX::mkfifo`. After that it is opened like any other path:

```
use POSIX qw(mkfifo);

my $path = "/tmp/demo.fifo";
mkfifo($path, 0600) or die "mkfifo $path: $!";
```

The second argument is the permission mode, the same `0600` / `0644` you would pass to `chmod` - it decides who may open the FIFO. The call fails if the path already exists, so a long-running setup typically unlinks a stale FIFO first.

Producer and consumer

The defining behaviour of a FIFO is that **open blocks until both ends are present**. A reader opening the path waits until some writer opens it too, and vice versa. That rendezvous is the feature: it synchronises two programs that never agreed on timing.

This single program demonstrates both ends by forking, but the two halves would work just as well as two separate scripts run from two terminals:

```
use POSIX qw(mkfifo);

my $path = "/tmp/demo.fifo";
unlink $path;           # clear any stale FIFO
mkfifo($path, 0600) or die "mkfifo $path: $!";

my $kid = fork() // die "fork: $!";

if ($kid == 0) {
    # Producer: open for writing (blocks until a reader appears).
    open my $w, ">", $path or die "producer open: $!";
    $w->autoflush(1);
    print $w "from-producer\n";
    close $w;
    exit 0;
}

# Consumer: open for reading (blocks until a writer appears).
open my $r, "<", $path or die "consumer open: $!";
my $line = <$r>;
print "consumer read: $line";      # consumer read: from-producer
close $r;

waitpid($kid, 0);
unlink $path;                     # remove the FIFO when finished
```

The two open calls are the synchronisation. Whichever process reaches its open first parks there until the other arrives; then both proceed together. Once open, the read and write handles behave like the two ends of any pipe - line reads, `readline`, buffered

`print`, all unchanged.

Two practical notes:

- **Autoflush on the writer.** As with every pipe and socket, a buffered write may never reach the reader until the buffer fills or the handle closes. Set `autoflush(1)` if the reader expects each line as it is produced.
- **A FIFO persists on disk.** `mkfifo` leaves a special file behind that outlives both processes. Remove it with `unlink` when the conversation is over, the same way you would clean up a lock file.

Reference cross-links

- `open` - open the FIFO path for reading or writing
- `pipe` - the anonymous-pipe relative for related processes
- `unlink` - remove the FIFO when finished
- `readline` - the function behind `<$fh>`

Next

When two processes need a connection rather than a one-way path - request and reply, both directions at once - reach for a socket. See *UNIX-domain and UDP sockets* and *TCP sockets*.

TCP sockets

A TCP socket carries a reliable, ordered byte stream between two processes that may be on different machines. This page builds the two halves of a TCP conversation from the raw built-ins: a **client** that connects and exchanges a line, and a **forking server** that accepts connections and handles each in its own child.

The high-level convenience module `IO::Socket::INET` is not available in PetaPerl, so every recipe here uses the built-in `socket` layer directly with address helpers from the `Socket` module. That is no loss: the built-in form is what `IO::Socket::INET` wraps, and seeing it in the open makes the handshake legible.

The address helpers

A socket address is a packed C struct `sockaddr_in`. The `Socket` module builds and takes apart that structure for you:

```
use Socket;  
my $packed = sockaddr_in($port, $ip);    # build  
my ($port, $ip) = sockaddr_in($packed);  # take apart
```

`$ip` is itself a packed four-byte address. `inet_aton` turns a dotted-quad string into one; `inet_ntoa` turns it back. The constant `INADDR_LOOPBACK` is the packed form of `127.0.0.1`, and `INADDR_ANY` means “every local interface”.

```
use Socket;
my $ip = inet_aton("127.0.0.1");
print inet_ntoa($ip), "\n";           # 127.0.0.1
```

A TCP client

The client creates a socket, connects it to a server address, and then reads and writes the handle like any other stream.

```
use Socket;

my $host = "127.0.0.1";
my $port = 8080;

socket(my $sock, PF_INET, SOCK_STREAM, getprotobyname("tcp"))
    or die "socket: $!";

my $addr = sockaddr_in($port, inet_aton($host));
connect($sock, $addr) or die "connect: $!";

$sock->autoflush(1);                # send each line as it is
    ↪printed

print $sock "ping\n";
my $reply = <$sock>;
print "server said: $reply";        # server said: pong: ping

close $sock;
```

The one line that catches everyone is `$sock->autoflush(1)`. Without it, `print $sock "ping\n"` sits in Perl's output buffer, the server never sees it, and both sides block forever waiting on each other. On any socket where you write a request and then read a reply, set autoflush before the first write.

A forking server

A server binds to a port, listens, and then loops accepting connections. The classic structure handles each client in a forked child so one slow client cannot stall the next:

```
use Socket;
use POSIX qw(WNOHANG);                # for the non-blocking reaper

my $port = 8080;
my $backlog = 128;                    # pending-connection queue
    ↪depth

socket(my $server, PF_INET, SOCK_STREAM, getprotobyname("tcp"))
    or die "socket: $!";

# Let the address be reused immediately after a restart.
```

(continues on next page)

(continued from previous page)

```

setsockopt($server, SOL_SOCKET, SO_REUSEADDR, 1)
    or die "setsockopt: $!";

bind($server, sockaddr_in($port, INADDR_ANY)) or die "bind: $!";
listen($server, $backlog)                    or die "listen: $!";

# Reap finished children so they do not become zombies.
$SIG{CHLD} = sub { 1 while waitpid(-1, WNOHANG) > 0 };

print "listening on port $port\n";

while (accept(my $client, $server)) {
    my $pid = fork();
    die "fork: $!" unless defined $pid;

    if ($pid == 0) {
        # Child: serve this one connection, then exit.
        close $server;                # child does not need the
↪listener
        $client->autoflush(1);

        while (my $line = <$client>) {
            print $client "pong: $line";
        }
        close $client;
        exit 0;
    }

    # Parent: hand the connection to the child and move on.
    close $client;                    # parent does not need this
↪client
}

```

Three details make this server correct rather than merely working:

- **SO_REUSEADDR before bind.** After a server exits, the kernel keeps the port in TIME_WAIT for a while. Without this option a restart fails with “Address already in use”. Set it and the restart binds immediately.
- **The \$SIG{CHLD} reaper.** Every forked child becomes a zombie when it exits until the parent calls `waitpid`. The handler reaps all finished children non-blockingly with `WNOHANG`, so the accept loop never pauses to wait.
- **Both sides close the descriptor they do not use.** The child closes the listening socket; the parent closes the accepted client socket. If either skips its `close`, the underlying file descriptor stays open in the wrong process and the connection never sees EOF.

Running both halves together

For a self-contained demonstration, fork the server into the background and run the client in the parent. The server binds to an ephemeral port (port 0) and reports the port the kernel assigned:

```
use Socket;

# --- server child on an ephemeral port ---
socket(my $server, PF_INET, SOCK_STREAM, getprotobyname("tcp"))
    or die "socket: $!";
setsockopt($server, SOL_SOCKET, SO_REUSEADDR, 1) or die "setsockopt:
↪$!";
bind($server, sockaddr_in(0, INADDR_LOOPBACK))    or die "bind: $!";
listen($server, 128)                              or die "listen: $!
↪";
my ($port) = sockaddr_in(getsockname($server));

my $kid = fork() // die "fork: $!";
if ($kid == 0) {
    accept(my $client, $server) or die "accept: $!";
    $client->autoflush(1);
    my $line = <$client>;
    print $client "pong: $line";
    close $client;
    exit 0;
}

# --- client in the parent ---
socket(my $sock, PF_INET, SOCK_STREAM, getprotobyname("tcp")) or
↪die;
connect($sock, sockaddr_in($port, INADDR_LOOPBACK)) or die "connect:
↪$!";
$sock->autoflush(1);
print $sock "ping\n";
print "client got: ", scalar <$sock>;    # client got: pong: ping
close $sock;

waitpid($kid, 0);
```

`getsockname($server)` reports the local address the socket is bound to. Unpacking it with `sockaddr_in` recovers the port the kernel chose, which the client then connects to - the standard way to write a test that does not hard-code a port.

Reference cross-links

Each call has a per-built-in reference page with the full argument shape and return value:

- [socket](#) - create the endpoint
- [bind](#) - attach it to a local address

- `listen` - mark it as accepting connections
- `accept` - take the next pending connection
- `connect` - reach out to a server
- `setsockopt` - set `SO_REUSEADDR` and friends
- `getsockname` - discover the bound port
- `fork` and `waitpid` - spawn and reap the per-client child
- `%SIG` - the CHLD reaper handler

Next

For local-only stream sockets without the network stack, and for connectionless datagrams, see *UNIX-domain and UDP sockets*.

UNIX-domain and UDP sockets

Not every socket reaches across a network. This page covers two lighter-weight relatives of the TCP socket from the *previous page*: a **UNIX-domain socket pair** for two related processes on one machine, and a **UDP socket** for connectionless datagrams.

Both use the same `socket` family of built-ins and the same `Socket` address helpers. The high-level `IO::Socket::*` wrappers are not available in PetaPerl, so the recipes stay at the built-in layer.

A socket pair for parent and child

`socketpair` creates two connected sockets in one call, with no addresses, no bind, and no connect. It is the ideal channel between a parent and a child it is about to fork: full-duplex (both ends can read and write), local, and already connected the moment it returns.

```
use Socket;

socketpair(my $child_end, my $parent_end, AF_UNIX, SOCK_STREAM, PF_
↳ UNSPEC)
    or die "socketpair: $!";

my $kid = fork() // die "fork: $!";

if ($kid == 0) {
    # Child keeps its end, closes the parent's.
    close $parent_end;
    $child_end->autoflush(1);

    print $child_end "child-says-hi\n";
    my $back = <$child_end>;
    print "child got: $back";          # child got: ack
    exit 0;
}
```

(continues on next page)

(continued from previous page)

```
# Parent keeps its end, closes the child's.
close $child_end;
$parent_end->autoflush(1);

my $msg = <$parent_end>;
print "parent got: $msg";           # parent got: child-says-hi
print $parent_end "ack\n";

waitpid($kid, 0);
```

Why reach for `socketpair` over a plain `pipe`? A pipe is one-directional - the reader reads, the writer writes, and that is fixed at creation. A socket pair is bidirectional: each end both reads and writes, so a request-and-reply exchange needs one `socketpair` call instead of two pipes wired in opposite directions.

The autoflush rule from TCP applies unchanged: set it on each end before the first write, or a buffered request never reaches the other side and both processes block.

A UDP datagram exchange

UDP is connectionless. There is no connect, no accept, no stream - just discrete messages, each addressed to a destination and each preserving its own boundaries. A receiver binds to a port; a sender fires a datagram at it with `send`; the receiver collects it with `recv`, which also reports who sent it.

```
use Socket;

# --- receiver on an ephemeral port ---
socket(my $server, PF_INET, SOCK_DGRAM, getprotobyname("udp"))
    or die "socket: $!";
bind($server, sockaddr_in(0, INADDR_LOOPBACK)) or die "bind: $!";
my ($port) = sockaddr_in(getsockname($server));

my $kid = fork() // die "fork: $!";
if ($kid == 0) {
    # Sender fires one datagram at the receiver's port.
    socket(my $client, PF_INET, SOCK_DGRAM, getprotobyname("udp"))
        or die;
    my $to = sockaddr_in($port, INADDR_LOOPBACK);
    send($client, "hello-udp", 0, $to) or die "send: $!";
    exit 0;
}

# recv fills $buf and returns the sender's packed address.
my $from = recv($server, my $buf, 65535, 0);
my ($sender_port, $sender_ip) = sockaddr_in($from);
print "got '$buf' from ", inet_ntoa($sender_ip), "\n";
                                # got 'hello-udp' from 127.0.0.1
→1
```

(continues on next page)

(continued from previous page)

```
waitpid($kid, 0);
```

What UDP gives you and what it does not:

- **Message boundaries are preserved.** One send is one recv. A recv returns exactly one datagram, never two concatenated and never half of one. The 65535 is the maximum length you are willing to accept; a shorter datagram returns its real length.
- **There is no connect and no handshake.** The sender does not open a connection; it addresses each datagram with send's fourth argument. recv reports the sender's address so a reply can go back the same way.
- **Delivery is not guaranteed.** UDP datagrams can be dropped, duplicated, or re-ordered by the network. Over the loopback interface, as in this recipe, that effectively never happens; over a real network you must handle loss yourself if it matters.

Reference cross-links

- [socketpair](#) - two connected local sockets
- [socket](#) - create a datagram or stream endpoint
- [bind](#) - attach the receiver to a local port
- [send](#) - fire a datagram at an address
- [recv](#) - collect a datagram and its sender
- [getsockname](#) - discover the bound port

Next

For shared memory and semaphores - channels the kernel owns rather than streams between processes - see [System V IPC](#).

System V IPC

System V IPC gives two unrelated processes a shared resource the kernel owns: a **semaphore set** for coordination, or a **shared memory segment** for passing bytes with no copying through a pipe. Each resource is identified by an integer key and survives until a process explicitly removes it.

The convenience modules `IPC::SysV`, `IPC::Semaphore`, and `IPC::SharedMem` are not available in PetaPerl, so the recipes here use the raw [semget](#) / [shmget](#) built-in family with the flag values written as plain numbers. Those built-ins are the layer the modules wrap; using them directly costs only a few named constants you supply yourself.

The flag constants you need

A handful of small integers control how these calls behave. They are the standard Linux values:

```
use constant {
    IPC_PRIVATE => 0,          # "give me a fresh, private key"
    IPC_CREAT  => 0o1000,     # create the resource if absent
    IPC_EXCL   => 0o2000,     # fail if it already exists
    IPC_RMID   => 0,          # control command: remove the resource
};
```

The low nine bits of the flag word are ordinary permission bits, the same `0600` / `0644` you would pass to `chmod`. `IPC_CREAT | 0600` means “create it, owner read/write”.

A shared-memory hand-off

Shared memory is the fastest channel: one process writes bytes into a segment, another reads them straight out, with no kernel copy in between. This program creates a segment, writes a payload, forks, and the child reads the same bytes back:

```
use constant {
    IPC_PRIVATE => 0,
    IPC_CREAT  => 0o1000,
    IPC_RMID   => 0,
};

my $size = 1024;

my $id = shmget(IPC_PRIVATE, $size, IPC_CREAT | 0600);
defined $id or die "shmget: $!";

my $payload = "shared-payload";
shmwrite($id, $payload, 0, length $payload) or die "shmwrite: $!";

my $kid = fork() // die "fork: $!";
if ($kid == 0) {
    # Child reads the same segment the parent wrote.
    shmread($id, my $buf, 0, length $payload) or die "shmread: $!";
    print "child read: $buf\n";          # child read: shared-
    ↪payload
    exit 0;
}

waitpid($kid, 0);

# The segment outlives the processes - remove it explicitly.
shmctl($id, IPC_RMID, 0) or warn "shmctl: $!";
```

Two points decide whether this code leaks:

- **shmread / shmwrite take an explicit length.** They do not stop at a NUL or a new-

line; you tell them exactly how many bytes to move, starting at an offset. Pass length \$payload so reader and writer agree on the size.

- **IPC_RMID is not automatic.** A shared segment persists in the kernel after every process attached to it has exited. You can see orphaned segments with `ipcs -m` at the shell. Always pair a `shmget` with a matching `shmctl` IPC_RMID.

A semaphore for coordination

A semaphore set is a small array of counters the kernel guards. The classic use is a lock: a process decrements the counter to enter a critical section and increments it to leave. This recipe creates a one-element set, initialises it, and removes it:

```
use constant {
    IPC_PRIVATE => 0,
    IPC_CREAT  => 0o1000,
    IPC_RMID   => 0,
    SETVAL     => 16,      # semctl command: set one value
    GETVAL     => 12,      # semctl command: read one value
};

# A set with one semaphore.
my $sem = semget(IPC_PRIVATE, 1, IPC_CREAT | 0600);
defined $sem or die "semget: $!";

# Initialise semaphore 0 to the value 1 (one token available).
semctl($sem, 0, SETVAL, 1) or die "semctl SETVAL: $!";

my $value = semctl($sem, 0, GETVAL, 0);
print "semaphore value: $value\n";      # semaphore value: 1

# Remove the set when finished.
semctl($sem, 0, IPC_RMID, 0) or warn "semctl IPC_RMID: $!";
```

The wait/signal operations themselves go through `semop`, which takes a packed string of (sem_num, sem_op, sem_flg) triples built with `pack`. Decrementing by one waits until a token is free; incrementing by one releases it. As with shared memory, a semaphore set the kernel owns must be removed with IPC_RMID or it lingers - visible under `ipcs -s`.

Reference cross-links

- `shmget` - create or open a shared segment
- `shmread` / `shmwrite` - move bytes in and out
- `shmctl` - control, including IPC_RMID
- `semget` - create or open a semaphore set
- `semop` - the wait/signal operations
- `semctl` - set, read, and remove
- `pack` - build the semop operation string

Next

Shared memory and semaphores are the heavyweight channels. For the lighter, stream-oriented ones, see *Subprocesses and co-processes* and *TCP sockets*.

- **Subprocesses and co-processes** - running a command and talking to it, in one or both directions, with the pipe open and with `IPC::Open2` / `IPC::Open3`.
- **FIFOs** - a named pipe on the filesystem that lets two unrelated programs meet on a path.
- **TCP sockets** - a complete client and a forking echo server over the loopback interface.
- **UNIX-domain and UDP sockets** - local stream sockets via `socketpair`, and connectionless UDP datagrams.
- **System V IPC** - semaphores and shared memory with the raw `semget` / `shmget` built-ins.

4.16.3 What this tutorial does not cover

- **Signals as a coordination mechanism beyond the basics.** Sending a signal with `kill` and catching it through `%SIG` is covered where each recipe needs it; the full signal-disposition table lives in the `%SIG` reference.
- **In-process parallelism.** Running work across CPU cores inside a single program is auto-parallelization, a different subject. See the *Concurrent Execution* guide.
- **High-level networking modules.** This tutorial stays at the built-in socket layer so the mechanics are visible. The recipes are the foundation a higher-level wrapper would build on.

REFERENCE

Reference for pperl, a faithful Rust reimplementation of perl5 5.42. Two parts: the **Perl language** reference (variables, functions, operators, regex, data types, subroutines) and the **Module** reference for every built-in native module.

5.1 Perl language

- *Variables*
- *Functions*
- *Operators*
- *Data types*
- *Subroutines*
- *Locale handling*
- *Diagnostics*
- *Security and taint mode*

5.1.1 Special variables

Perl ships with a fixed set of built-in variables that are populated, read, or honoured by the language itself. Some hold state from the last operation (the regex match variables, \$!, \$@); some configure how built-ins behave (\$/, \$\\, \$,); some expose the running interpreter and its environment (\$^V, \$^O, %ENV, @INC); a few are not really variables at all but hooks into runtime services (%SIG).

PetaPerl implements every special variable that Perl 5.42 documents, with identical semantics. Where the runtime does something noteworthy of its own - a JIT fast path that reads a cached cell, an Arc-backed COW string for \$_ - that is an implementation detail; user-visible behaviour matches perl5.

This reference is split into one page per family. Variables that share context (you almost never set \$, without thinking about \$\\ and \$|) live on the same page; variables that don't (process IDs and regex captures) don't.

Choose a family

Default variables

Two variables hold the implicit operands of Perl. `$_` is the default scalar that countless built-ins read when called without an argument. `@_` is the argument array of every subroutine. Together they account for most of the “where did that value come from?” surprises in Perl code.

`$_` - the default scalar

`$_` is the *topic*: the value being talked about right now. A huge swath of Perl’s built-ins consult it when called with no explicit argument:

```
for my $line (<$fh>) {  
    chomp;                                # chomp $line - but $line was    
    ↪ aliased to $_  
    print if /\bERROR\b/;                # print $_ if $_ =~ /\bERROR\b/  
}
```

The full list of built-ins that default to `$_` includes `chomp`, `chop`, `lc`, `lcfirst`, `length`, `pos`, `print`, `printf`, `quotemeta`, `say`, `split` (default pattern), `study`, `uc`, `ucfirst`, `unlink`, the `m//` and `s//` operators, the `tr//` operator, and the `<>/-X` filetest operators. Their detail pages name `$_` as the default; this page is the canonical description.

Aliasing inside for and while

The `for` (and `foreach`) loop **aliases** `$_` to each element of the list, in turn:

```
my @items = (1, 2, 3);  
for (@items) {  
    $_ *= 2;                                # mutates @items in place  
}  
# @items is now (2, 4, 6)
```

Modifying `$_` inside the loop modifies the list element itself. This is identical behaviour to `map`, `grep`, `any`, `all` - and `first` and `reduce` from `List::Util`.

`while (<$fh>)` does **not** alias - it reads each line into `$_` as a fresh copy. You cannot push back into the file by assigning to `$_` in a `while (<$fh>)` loop.

Localising `$_`

A subroutine that wants its own `$_` without disturbing the caller’s must local-ise it:

```
sub categorise {  
    local $_ = shift;                        # decouple from caller's $_  
    return 'small' if /\^d{1,3}$/;  
    return 'medium' if /\^d{4,6}$/;  
    return 'large';  
}
```

(continues on next page)

(continued from previous page)

```
my @sizes;
for ('5', '12345', '999999999') {
    push @sizes, categorise($_); # caller's $_ stays the loop alias
}
```

Without the `local`, calling `categorise()` with `shift` would work - but any `=~`, `/...`, `/`, or `chomp` inside it would silently read or write the loop alias, breaking the caller's iteration.

`$_` as an lvalue

`$_` is a real scalar variable; you can assign to it, take a reference to it, and pass it as an lvalue:

```
$_ = 'hello';
chomp;                                # nothing to chomp; $_ unchanged
s/l/L/g;                              # $_ is now 'heLLo'

my $ref = \$_;                        # \$_ is the alias's address
$$ref = 'goodbye';                   # mutates whatever $_ is currently
→bound to
```

Beware that the second `$$ref` will write to **whatever `$_` is right now** - which inside a `for` loop is the current list element. References to `$_` taken outside a loop and used inside one rarely do what the author meant.

Lexical `$_`

our `$_` and my `$_` were experimental features that have been removed. `$_` is always the package-global; use `local $_` for isolation.

`@_` - subroutine arguments

Inside a subroutine, `@_` is the array of arguments passed to that call. Each element is **aliased** to the caller's expression - `@_` is not a copy:

```
sub double {
    $_[0] *= 2;                        # mutates the caller's variable
}

my $x = 5;
double($x);
# $x is now 10
```

This is why most subs start with `my (... ) = @_`; - the unpacking creates copies, after which the routine cannot accidentally mutate caller state:

```
sub greet {
    my ($name, $greeting) = @_; # decoupled from caller
```

(continues on next page)

(continued from previous page)

```
$greeting //= 'Hello';
return "$greeting, $name!";
}
```

The aliasing is occasionally useful - it is how `chomp(@lines)` mutates every element of the caller's array - but in everyday code the unpacked-copy idiom is the rule.

wantarray and the calling context

A sub that wants to behave differently depending on whether it was called in scalar, list, or void context inspects `wantarray`:

```
sub items {
    return wantarray ? (1, 2, 3) : 3;
}

my @list = items();           # (1, 2, 3)
my $cnt  = items();           # 3
items();                       # void - return value discarded
```

`@_` and `wantarray` are the two pieces of the calling-convention picture. `@_` says *what* was passed; `wantarray` says *what is expected back*.

@_ after shift, pop, unshift, push

Modifying `@_` itself (not its elements) has no aliasing effect on the caller - it's a normal local array variable. `shift @_` removes the first argument, leaving the rest:

```
sub method_call {
    my $self = shift;           # standard OO pattern
    my %args = @_;              # rest of args as a hash
    ...
}
```

`shift` with no arguments inside a subroutine defaults to `@_`, which is why you see `my $self = shift`; everywhere.

\$_ = shift - wait, what?

A common deceptive idiom:

```
sub trim {
    $_ = shift;                 # - set caller's $_ - DO NOT DO_
    ↪THIS
    s/^\s+//;
    s/\s+$//;
    return $_;
}
```

That `$_ = shift` writes to the package-global `$_`, which is *the caller's loop topic* if the caller is inside a `for`. The fix is local-isation, as above:

```
sub trim {
    local $_ = shift;
    s/^\s+//;
    s/\s+$//;
    return $_;
}
```

See also

- `map`, `grep` - the most common consumers of `$_` aliasing.
- `shift`, `pop` - default to `@_` inside a subroutine.
- `wantarray` - the companion of `@_` in the calling-convention picture.
- `local` - the safe way to set `$_` inside a sub that the caller might be iterating with.
- *Regex binding* - `=~` is what you use when you don't want the implicit `$_` target.

Process identity

Six variables expose what the operating system thinks of this process: its name, its PID, and its real-and-effective user and group IDs. They are most useful in scripts that need to act based on *who is running them*, drop privileges after a privileged operation, or distinguish parent from child after a `fork`.

Variable	What it holds	Mutable
<code>\$0</code>	Program name (and <code>ps</code> display)	yes
<code>\$\$</code>	Current process PID	no (read-only)
<code>\$<</code>	Real UID	yes (privileged)
<code>\$></code>	Effective UID	yes (privileged)
<code>\$(</code>	Real GID (plus supplementaries)	yes (privileged)
<code>\$)</code>	Effective GID (plus supplementaries)	yes (privileged)

`$0` - program name

The name of the program being executed, as passed in `argv[0]`:

```
print "Started by: $0\n";           # e.g. "/usr/local/bin/myscript.pl"
```

Assigning to `$0` changes how the process appears in `ps` output and similar listings - handy for long-running scripts that fork and want each child to be identifiable:

```
$0 = "myserver: worker pid $$";    # appears in `ps` as that string
```

The change writes into the area of process memory that `ps` reads on Linux. Older Unix variants and Windows have different rules; PetaPerl follows `perl5`'s behaviour exactly.

Reading `$0` always returns the current value, including any modifications you have made.

`$0` is **not** the program name as the *script* sees it - for that, see `@ARGV` (which has `$0` as the companion at index `[-1]` of the original `argv`, not as `@ARGV[0]`).

`$$` - process ID

Read-only. The PID of the current process.

```
print "Running as PID $$\n";

open my $log, '>>', "/var/log/myapp.log" or die $!;
print $log "[$$] starting...\n";
```

After fork, the child sees a different `$$` from the parent - this is the standard way to tell which side of the fork you are on:

```
my $pid = fork;
defined $pid or die "fork: $!";
if ($pid == 0) {
    # child
    print "child: my pid is $$\n";
    exit 0;
}
# parent
print "parent: my pid is $$, child is $pid\n";
waitpid $pid, 0;
```

The English alias `$PROCESS_ID` (and the redundant `$PID`) work the same as `$$`.

To get the *parent's* PID, use `getppid`.

Real vs effective IDs - what they mean

Every Unix process has *two* user IDs and *two* group IDs:

- The **real** UID/GID is who you are - usually the user who invoked the program.
- The **effective** UID/GID is what the kernel checks for permission decisions - typically the same as real, but different for *setuid* and *setgid* programs.

When you run a *setuid-root* binary, the kernel sets the effective UID to root (the file's owner) but leaves the real UID as your own. The program can then act as root for the privileged operation, and either *drop privileges* by setting effective back to real, or *check who really invoked it* by reading the real UID.

`$<` - real UID

```
print "Invoked by user with UID $<\n";
```

`$<` is the real UID. It almost never changes during a process's lifetime; the cases where it can are very limited (only root can change it, and only via specific syscalls).

Mnemonic: *the UID you came **from**.*

\$> - effective UID

```
print "Privileges currently effective: UID $>\n";
```

\$> is the effective UID - the one that matters for permission checks. In a `setuid` program, this is initially the file owner's UID; in an ordinary program, it equals \$<.

Mnemonic: *the UID you went **to**.*

The classic `setuid` pattern

A `setuid`-root helper that needs to do *one* privileged operation and then drop privileges so the rest of the script runs as the real user:

```
#!/usr/bin/perl
use strict;
use warnings;

# Privileged work first - we still have effective root.
open my $secret, '<', '/etc/some-restricted-file'
    or die "open: $!";
my $config = do { local $/; <$secret> };
close $secret;

# Drop privileges. From now on, both real and effective UID
# are the invoking user's, so any subsequent open/system call
# is checked against the real user.
$> = $<;                                # set effective UID to real UID
$) = $(;                                # set effective GID to real GID

# Now run the rest of the script as the real user.
do_user_work($config);
```

Permanent privilege drop **must** set effective before real (or both at once via `POSIX::setuid()`); on most systems setting real first will lock you out of restoring effective. Check \$! after each assignment - the kernel rejects unauthorised changes:

```
$> = $<;
die "could not drop privileges: $!" if $> != $<;
```

Swapping real and effective

When you want to *temporarily* switch the effective ID for a single operation and then restore it:

```
my $saved_uid = $>;
$> = $<;                                # become real user
my $ok = open(my $fh, '<', $user_path);
```

(continues on next page)

(continued from previous page)

```
$> = $saved_uid;           # back to effective
$ok or die "open $user_path: $!";
```

This works only on systems supporting `setreuid()` - most modern Unixes, including Linux. `POSIX::seteuid()` from the `POSIX` module is the explicit and portable spelling.

`$ (` - real GID (with supplementary groups)

`$ (` returns the real GID followed by every *supplementary group* the process belongs to, all space-separated:

```
print "Real GID list: $(\n";      # e.g. "1000 1000 4 27 100 999"
                                #       primary + supplementaries
my @groups = split ' ', $ ( ;
my $primary = $groups[0];
```

The leading number is the primary GID; the rest are supplementary groups (from `/etc/group`). Assigning to `$ (` sets the GID; the syntax accepts a single number:

```
$ ( = 100;                      # become primary GID 100
```

`$ )` - effective GID (with supplementary groups)

Same shape as `$ (`, but for the effective GID:

```
print "Effective GID list: $ )\n";

# Drop both real and effective GID to a specific gid:
$ ( = $ ) = "$gid $gid";
```

Permission checks use the effective GID and supplementaries; the real GID list is informational.

When you actually use these

Day-to-day Perl code does not touch UIDs and GIDs. The realistic use cases are:

1. **Setuid scripts** that need to drop privileges. Standard idiom shown above.
2. **Daemon startup** that runs as root to bind a privileged port, then drops privileges before serving requests:

```
socket(my $srv, ...) or die $!;
bind($srv, sockaddr_in(80, INADDR_ANY)) or die $!;
listen($srv, SOMAXCONN) or die $!;

# Now drop to nobody:nogroup before accepting connections.
my (undef, undef, $u, $g) = getpwnam('nobody');
$ ) = "$g $g";
$ ( = $g;
```

(continues on next page)

(continued from previous page)

```
$> = $u;
$< = $u;
```

3. **Privilege check** in a script that should refuse to run as root: `die "do not run me as root\n" if $< == 0.`

Tainted-mode interaction

Under `-T`, Perl notices when the real and effective IDs differ (i.e. the script is `setuid`) and turns on extra checks: certain operations refuse tainted arguments, and the search path for `system()` and `exec()` is constrained. The relevance here is that whether `-T` is on is decided based on `$<` vs `$>` at startup.

See also

- `getppid` - the *parent*'s PID, paired with `$$`.
- `fork` - after which `$$` differs in parent and child.
- `getpwuid`, `getgrgid` - turn the numeric IDs into user and group names.
- `POSIX::setuid` - the portable, explicit way to change UIDs together.
- *Error variables* - `#!` after an `$< / $>` assignment tells you why the kernel refused.

Error state

Four variables report errors from four different layers. They are not interchangeable and they are populated by entirely different mechanisms; misusing one for another is the most common cause of “why doesn’t my error handling work” bugs in Perl code.

Variable	Source	Lifetime
<code>#!</code>	C library <code>errno</code> from the last syscall	Set on syscall failure; meaningless otherwise
<code>\$@</code>	The exception caught by the last <code>eval</code>	Set on <code>eval</code> exit (empty string on success)
<code>\$?</code>	Exit status of the last child process	Set by every <code>wait/pipe-close/system</code>
<code>\$^E</code>	OS-extended error info	Same as <code>#!</code> on Unix; richer on VMS/Win32

The single best thing you can do for your error handling is to internalise *which variable answers which question*:

- “Did my open/read/connect/syscall fail?” → check the return value first, then `#!`.
- “Did my `eval` block catch an exception?” → check `$@`.
- “What was the exit code of the program I just ran?” → decode `$?`.
- “Am I on Windows or VMS and need OS-specific error text?” → also look at `$^E`.

\$! - the system error

\$! is the Perl interface to the C library's `errno`. It has a *dual nature*: read it in numeric context to get the integer `errno`, in string context to get the corresponding error string:

```
open(my $fh, '<', '/no/such/file') or do {  
    print "errno = ", $! + 0, "\n";           # numeric: 2  
    print "errstr = $!\n";                   # string: "No such file  
→or directory"  
};
```

Crucially, \$! is **only meaningful immediately after a failure**. The C `errno` is not cleared on success, so a later read might see a stale value from any earlier system call. The reliable shape:

```
if (open my $fh, '<', $path) {  
    # success - $! is meaningless here  
    process($fh);  
} else {  
    # ONLY here is $! meaningful  
    die "open $path: $!";  
}
```

Writing the failure on the same line as the operation that caused it is the safest spelling - it leaves no opportunity for an intervening Perl operation to clobber `errno`.

%! - symbolic `errno` checks

%! is a hash whose keys are `errno` symbol names like `ENOENT`, `EACCES`, `EAGAIN`. A key tests true exactly when \$! currently equals that `errno`:

```
unless (open my $fh, '<', $path) {  
    if ($!{ENOENT}) {  
        return [];           # missing file → empty list  
    }  
    if ($!{EACCES}) {  
        die "permission denied: $path";  
    }  
    die "open $path: $!";     # anything else: re-raise  
}
```

This is portable - the numeric values of `errno` symbols differ across operating systems, but %! keys are always the symbol names. Behind the scenes, %! is provided by the `Errno` module, which is loaded on demand the first time you use the hash.

Assigning to \$!

Writing to \$! sets `errno` in the C library - useful for faking up an error message that downstream code will see:


```

$! = 13;                                # EACCES
print "$!\n";                            # "Permission denied"

# Slightly more idiomatic - set errno by symbol:
use Errno qw(:POSIX);
$! = EACCES;
die "synthetic permission failure: $!";

```

\$@ - the exception variable

Set by `eval` when the block (or string) it ran threw an exception. It contains either the value passed to `die` (which is most often a string, but can be any reference) or the parsing/runtime error message from the interpreter.

```

eval {
    die "no port configured\n" unless defined $port;
    open(my $sock, '<', "/dev/tcp/$host/$port") or die "connect: $!";
    1;
};
if ($@) {
    warn "could not open $host:$port - $@";
    return;
}

```

Successful evaluation sets `$@` to the empty string `''`. The `if ($@)` check is therefore reliable: empty string is false, any non-empty value is true.

The classic `$@` clobbering bug

There is a hazard here. Anything that runs *between* the `eval` exit and your `if ($@)` check can clobber `$@`. The most common offender is an object's `DESTROY` method - when objects go out of scope at the close of the `eval` block, their `DESTROY` runs, and if `DESTROY` itself does any `eval`, your exception is gone:

```

eval {
    my $obj = MyClass->new;
    $obj->might_die;
    1;
};
# If MyClass::DESTROY does an eval, $@ is now empty.
warn "got: $@";                    # may print nothing!

```

The historically-recommended fix is to capture `$@` immediately, or to use the `Try::Tiny` module, which handles this and several related pitfalls:

```

my $err;
{
    local $@;                    # save and isolate
    eval {

```

(continues on next page)

(continued from previous page)

```
        risky_op();
        1;
    } or $err = $@ || 'unknown error';
}
if ($err) {
    warn "got: $err";
}
```

The Perl 5.34+ native `try/catch` feature is the cleaner modern spelling - it does not use `$@` at all and is not affected by the destructor problem:

```
use feature 'try';
try {
    risky_op();
} catch ($err) {
    warn "got: $err";
}
```

PetaPerl supports `try/catch` natively. New code should prefer it.

eval and \$!

`eval` does **not** preserve `$!`. If your `eval` block did a syscall that failed and then died, `$!` is whatever the *last* operation set it to - which inside a destructor or `local $SIG{__DIE__}` handler can be anything. If your error message needs `$!`, capture it inside the `eval`:

```
eval {
    open(my $fh, '<', $path) or die "open $path: $!\n";
    ...
};
```

The `\n` at the end is conventional - it suppresses the “at line N” suffix that Perl appends to die messages, which is right for user-facing errors.

\$? - the child error

After a `system`, `waitpid`, `wait`, backtick command, or pipe close, `$?` contains the 16-bit `wait()` status of the child process. It is **not** simply the exit code:

```
system($cmd, @args);
my $status = $?;

my $exit_code = $status >> 8;           # exit value passed to exit()
    ↳ in the child
my $signal     = $status & 0x7F;       # signal that killed the child,
    ↳ or 0
my $coredump   = $status & 0x80;       # set if the child dumped core

if ($status == -1) {
```

(continues on next page)

(continued from previous page)

```

    die "system: $!";                # could not even fork/exec
} elsif ($signal) {
    die "$cmd died with signal $signal";
} elsif ($exit_code) {
    die "$cmd failed (exit $exit_code)";
}

```

The `>> 8` shift is the most-cited Perl gotcha there is. Only the upper byte is the exit code; the lower byte holds the signal-and-coredump flags. Forgetting the shift produces values multiplied by 256.

Modern Perl provides the `POSIX` module's `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG`, `WIFEXITED` macros, which are the portable named accessors:

```

use POSIX ':sys_wait_h';
if (WIFEXITED($?)) { my $code = WEXITSTATUS($?); ... }
if (WIFSIGNALED($?)) { my $sig = WTERMSIG($?); ... }

```

`$?` is **not** preserved across other operations. If you need the value, capture it into a local variable on the next line - otherwise the next backtick or `system` will overwrite it.

`$?` in END blocks

Inside an `END` block, `$?` contains the intended exit code. Assigning to it overrides the script's exit status:

```

END {
    $? = 0 if $? == 255 && $shutdown_was_clean;
}

```

`$^E` - the extended OS error

On Unix, `$^E` is exactly the same as `$!`. On Windows, it contains the `GetLastError()` value, which is sometimes more informative than `errno` (which Win32 maps a coarse subset onto). On VMS, it is the native VMS status code.

PetaPerl is Linux-only. `$^E` and `$!` are interchangeable here; the variable exists for portable code that may also run on non-Unix perls.

Putting them together - a worked example

```

sub copy_file {
    my ($src, $dst) = @_;

    open my $in, '<', $src or die "open $src for reading: $!";
    open my $out, '>', $dst or die "open $dst for writing: $!";

    eval {
        local $/ = \65536;                # block reads, scoped to this eval

```

(continues on next page)

(continued from previous page)

```
while (defined(my $chunk = <$in>)) {
    print {$out} $chunk or die "write $dst: $!";
}
close $out                      or die "close $dst: $!";
1;
} or do {
    my $err = $@ || 'unknown error';
    unlink $dst;                # clean up the partial file
    close $in;                  # may set $! itself
    die "copy_file($src → $dst): $err";
};

close $in or warn "close $src: $!";
return 1;
}
```

Three error variables in five paragraphs. `$!` reports each syscall failure on the line that triggered it. `$@` collects any exception that crossed the eval boundary. `$?` is not in this example because there is no child process - but if the function shelled out to `cp`, you would decode `$?` in the same shape as the [system](#) page.

See also

- [die](#) - populates `$@`. The usual way to raise an exception from your own code.
- [eval](#) - catches exceptions; sets `$@`.
- [system](#), [waitpid](#), [wait](#) - populate `$?`.
- [try/catch](#) - the modern alternative to `eval/$@` that side-steps the clobbering hazard.
- [Errno](#) - the source of `%! keys`; provides named `errno` constants.
- `Try::Tiny` - robust eval wrapper that preserves `$@` correctly.
- `%SIG` - `$SIG{__DIE__}` and `$SIG{__WARN__}` are the hooks that fire when `die` and `warn` run; they can rewrite `$@`.
- [Logical operators](#) - the `or die` idiom that drives almost all `$!` reporting.

Filehandle and printing state

Seven variables that drive Perl's I/O. They are *globals* - setting `$/` in one routine changes how every other routine in the same process reads from a filehandle. The discipline that keeps that from being a disaster is local-isation: every change to a variable on this page should be inside a local block unless the program genuinely wants the new value to be permanent.

Variable	Reads as	Default	Used by
<code>\$/</code>	input record separator	<code>"\n"</code>	<code><\$fh></code> , <code>readline</code>
<code>\$\</code>	output record separator	<code>undef</code>	<code>print</code> , <code>printf</code> , <code>say</code> (forced <code>\n</code>)
<code>\$,</code>	output field separator	<code>undef</code>	<code>print</code> , <code>printf</code> (between args)
<code>\$"</code>	list separator	<code>" "</code>	array interpolation <code>"@arr"</code>
<code>`\$</code>	<code>`</code>	output autoflush	<code>0</code>
<code>\$.</code>	current line number	<code>0</code>	last filehandle read
<code>\$;</code>	subscript separator	<code>"\034"</code>	<code>\$h{\$x, \$y}</code> multidim emulation

`$/` - the input record separator

Controls what `<$fh>` and `readline` treat as the end of one record. The default `"\n"` means *line-by-line* reading; setting it to other values produces other reading modes.

Line mode (the default)

```
while (my $line = <$fh>) {
    chomp $line;                # strips the terminator $/
    process($line);
}
```

`chomp` removes whatever `$/` currently holds - when you change `$/`, `chomp` follows it.

Slurp mode

`undef $/` makes `<>` read the entire file in one go:

```
my $whole_file = do {
    local $/;
    <$fh>;
};
```

The `local` is essential. Without it, every other piece of code that subsequently reads from a filehandle in the same process will silently slurp.

Paragraph mode

`local $/ = ""`; reads paragraph-at-a-time, splitting on runs of two or more newlines:

```
{
    local $/ = "";
    while (my $para = <$fh>) {
        # $para ends with exactly two newlines (or one at EOF)
        process_paragraph($para);
    }
}
```

(continues on next page)

(continued from previous page)

```
}  
}
```

This is awk's RS="" behaviour. Leading blank lines at the start of the file are silently skipped.

Fixed-record mode

local \$/ = \N reads N bytes per call:

```
local $/ = \4096;                # binary 4-KiB chunks  
while (defined(my $chunk = <$fh>)) {  
    process($chunk);  
}
```

The reference is to a number, not the number itself. This is the right shape for reading binary data, network packets, or doing bulk copy without scanning for \n.

Multi-character terminators

```
local $/ = "\r\n";                # CRLF mode  
local $/ = "\nEND-OF-RECORD\n";  # custom marker
```

The terminator is a literal string, not a regex.

\$\ - the output record separator

Appended to every `print` call. Default is undef (nothing appended). Setting it changes every subsequent print:

```
{  
    local $\ = "\n";                # all prints get a trailing newline  
    print "first";  
    print "second";  
}  
# Output: "first\nsecond\n"
```

This is the natural fit for `-l` on the command line - `-l` sets `$\` to `\n`. In script code, `local $\ = "\n"` inside a tight reporting loop is occasionally worth it; otherwise `say` (which forces `\n` regardless of `$\`) is the cleaner spelling.

\$, - the output field separator

Inserted between `print`'s arguments. Default is undef:

```
print "a", "b", "c";                # "abc"  
  
{  
    local $, = ", ";
```

(continues on next page)

(continued from previous page)

```

    print "a", "b", "c";          # "a, b, c"
}

{
    local $, = "\t";
    print @items;                # tab-separated record
}

```

`$,` is per-print, not per-line - it does not separate *successive* print calls. Combine with `$\` for full TSV-style output:

```

local ($,, $\) = ("\t", "\n");
print 'col1', 'col2', 'col3';    # "col1\tcol2\tcol3\n"

```

`$"` - the list separator

Joins array elements when an array is interpolated into a double-quoted string. Default is " " (single space):

```

my @arr = (1, 2, 3);
print "items: @arr\n";          # "items: 1 2 3"

{
    local $" = ', ';
    print "items: @arr\n";      # "items: 1, 2, 3"
}

{
    local $" = "\n ";
    print "items:\n @arr\n";    # "items:\n 1\n 2\n 3"
}

```

`$"` is *only* consulted on interpolation. `print @arr` (no quotes) uses `$,`, not `$"`.

`$|` - output autoflush

Forces a flush after every write on the **currently selected** filehandle:

```

$| = 1;                          # autoflush STDOUT
print "Loading..."; sleep 1;     # appears immediately, not at exit

```

The “currently selected” qualification matters. By default, `select` returns `STDOUT`, so `$| = 1` autoflushes `STDOUT`. To autoflush a different handle:

```

my $old = select($log_fh);
$| = 1;
select($old);                    # restore selection

```

The slightly nicer modern spelling uses `IO::Handle`:

```
$log_fh->autoflush(1);
```

\$| is read-as-boolean: print \$| shows 1 or 0.

\$. - current line number

The line number of the last <>-style read. Each filehandle has its own counter; reading from a different handle aliases \$. to that handle's count:

```
while (<$fh>) {  
    print "$.: $_";           # numbered listing  
}  
  
# After the loop, $. is the total number of lines read.
```

\$. is reset when the filehandle is *closed*. Reopening (without an explicit close in between) does **not** reset.

For multi-file iteration via <>, lines are numbered cumulatively across files unless you explicitly close ARGV at each file boundary - see [eof](#) for the standard close ARGV if eof; idiom.

\$_ - the subscript separator

Used to fake multi-dimensional hashes - a deeply legacy feature preserved for backward compatibility:

```
$_{$x, $y} = $value;           # equivalent to $_{"$x\034$y"}
```

The default value is "\034" (FS, “file separator”, an obscure ASCII control byte). Real multi-dimensional structures should use nested hash refs (\$_{\$x}{\$y}); \$_ exists because Perl 4 had no proper references.

When you actually localise these

The most common shape, by far:

```
sub slurp_text {  
    my ($path) = @_;  
    open my $fh, '<', $path or die "open $path: $!";  
    local $/;           # slurp inside this sub only  
    return <$fh>;  
}  
  
sub paragraphs_of {  
    my ($path) = @_;  
    open my $fh, '<', $path or die "open $path: $!";  
    local $/ = "";  
    return <$fh>;       # list context: list of paragraphs  
}
```

(continues on next page)

(continued from previous page)

```

sub csv_print {
    my (@row) = @_;
    local ($/, $\\, $,) = (",", "\\n");
    print @row;
}

```

The cost of forgetting `local` is invisible at write time and catastrophic at use time - a subroutine that “just sets `$/`” can break unrelated code reading from completely different filehandles. Treat `$/`, `$\\`, `,$`, `,$` as *global mode flags*: always set them with `local`.

See also

- `open`, `readline`, `<$fh>` - the consumers of `$/`.
- `print`, `printf`, `say` - the consumers of `$\\`, `,$`.
- `chomp` - removes whatever `$/` currently holds; pairs with `$/` changes.
- `eof` - the close ARGV if eof idiom that makes `$.` count per-file across `<>`.
- `select` - selects the filehandle that `$|` and the unprefix `$/`, `$\\` apply to.
- `IO::Handle` - provides `autoflush`, `output_record_separator`, etc. as method calls.
- `local` - the safe way to change any variable on this page.
- *Interpolation* - how `$"` shapes array expansion inside double-quoted strings.

Regex match variables

After every successful pattern match, Perl populates a fixed set of variables with information about *what* matched and *where*. These are read-only - you observe them, you do not write to them. The “successful” qualification is critical: an unsuccessful match leaves the variables holding *whatever the previous successful match in the same dynamic scope set them to*. Always gate your access with the boolean result of the match itself.

Variable	Holds
<code>\$1..\$N</code>	Text captured by the <i>N</i> th capturing group
<code>\$&</code>	The entire matched substring
<code>\$`</code>	The string preceding the match
<code>\$'</code>	The string following the match
<code>\$+</code>	Text captured by the highest-numbered group
<code>^N</code>	Text captured by the <i>most recently closed</i> group
<code>@-</code>	Start offsets: <code>\$-[0]</code> = match, <code>\$-[N]</code> = <code>\$N</code> start
<code>@+</code>	End offsets: <code>\$+[0]</code> = match end, <code>\$+[N]</code> = <code>\$N</code> end
<code>%+</code>	Hash of named captures: <code>\$+{name}</code> = <code>(?<name>...)</code>
<code>%-</code>	Hash of <i>all</i> captures by name (arrayref values)
<code>@{^CAPTURE}</code>	Array of captures: <code>\${^CAPTURE}[0]</code> = <code>\$1</code> , etc.
<code>\${^MATCH}</code>	Same as <code>\$&</code> , populated only with the <code>/p</code> flag
<code>\${^PREMATCH}</code>	Same as <code>\$`</code> , only with <code>/p</code>
<code>\${^POSTMATCH}</code>	Same as <code>\$'</code> , only with <code>/p</code>

The basic pattern

```
if ( "Mr. Smith, age 47" =~ /(\w+)\s+(\w+),\s+age\s+(\d+)/ ) {
    print "title: $1\n";           # Mr
    print "name:  $2\n";           # Smith
    print "age:   $3\n";           # 47
    print "match: $&\n";           # the full match
}
```

The `if` is what makes the access safe. Without it, on a *non-matching* string, `$1/$&` would still hold values from some earlier successful match - there is no “match failed → clear” rule.

Numbered captures - `$1..$N`

Each `(...)` capturing group fills one variable. The numbering is left-to-right by *opening* parenthesis:

```
"alpha-beta=42" =~ /^( \w+ ) - ( \w+ ) = ( \d+ ) $ / ;
print "$1 / $2 / $3\n";           # alpha / beta / 42
```

Non-capturing groups `(?:...)` and lookarounds `(?=...)/(?!...)` do **not** consume a number; they are invisible to the count.

For `s///` substitutions, `$1..$N` are visible inside the replacement (along with the substitution-only `$1`-style backreferences in single-quoted replacements):

```
my $s = "John Smith";
$s =~ s/(\w+)\s+(\w+)/$2, $1/;    # "Smith, John"
```

Named captures - `(?<name>...)` and `%+`

Named captures are clearer than counting parentheses, especially in patterns with many groups:

```
my $log = "2024-03-15 14:32:01 ERROR connection refused";
if ( $log =~ /^(?<date>\d{4}-\d{2}-\d{2})
               \s+
               (?<time>\d{2}:\d{2}:\d{2})
               \s+
               (?<level>\w+)
               \s+
               (?<msg>.*$)/x ) {
    print "[${level}] ${date} ${time}: ${msg}\n";
}
```

`%+` is the named-capture hash. The numbered forms still work (named captures are also assigned numbers in the order they appear), so `$1` would be `${date}` here.

`%-` is similar but its values are array references, holding *every* capture under that name (relevant when using branch-reset groups `(?|...|...)` or when a name is reused across alternation branches):

```
"abc" =~ /(?!(<x>a)|(<x>b)|(<x>c))/;
print "%-{"x"} has @{"$-{"x"}}\n";      # captured value(s) under name 'x'
```

Most code only ever reads %+. %- is for the corner cases.

Match boundaries - @- and @+

The match's start and end offsets in the matched string:

```
"hello world" =~ /(\w+)\s+(\w+)/;
print "match started at $-[0], ended at $+[0]\n";      # 0 .. 11
print "group 1: $-[1] .. $+[1]\n";                    # 0 .. 5
print "group 2: $-[2] .. $+[2]\n";                    # 6 .. 11
```

$\$-[N]$ and $\$+[N]$ give the same information as `substr($var, $-[N], $+[N] - $-[N])` would extract - they are how you reconstruct *positions*, not values, after a match.

The classic use: replacing matched substrings while preserving the original (without the `s///` operator):

```
my $s = "the quick brown fox";
$s =~ /quick (brown)/;
my $before = substr($s, 0, $-[0]);
my $after  = substr($s, $+[0]);
my $g1     = substr($s, $-[1], $+[1] - $-[1]);
print "$before|FOUND $g1|$after\n";
```

$\$&$, $\$`$, $\$'$ - match, prematch, postmatch

```
"hello world" =~ /wo\w+/;
print "before: '$`'\n";      # 'hello '
print "match: '$&'\n";      # 'world'
print "after: '$\'''\n";     # ''
```

These three are Perl's oldest match variables and historically the most expensive - see *Performance: the $\$&$ story* below.

$\$+$ is the text captured by the *highest-numbered* group that participated in the match:

```
"abc" =~ /(a)(b)?(c)?/;      # $1='a', $2='b', $3='c', $+='c'
"a"    =~ /(a)(b)?(c)?/;      # $1='a', $2=undef, $3=undef, $+='a'
→ ''
```

$\$^N$ is similar but holds the *most-recently-closed* group's text - useful inside complex patterns that need the value of “the group that just finished”:

```
my $s = "tag:value";
$s =~ /(\w+):(\w+) (?{ $cb = $^N })/;
# $cb is the text most recently captured (here: "value")
```

Performance: the \$& story

Historical caveat that you will find in old code and in books:

Don't mention \$&, \$`, or \$' *anywhere* in your code, including in modules you require - they cause every successful match to copy the whole matched string, slowing the program down.

This was true through Perl 5.10. From Perl 5.18 onward, the runtime tracks *which* of the three variables your code actually mentions and only copies what is needed. From Perl 5.20 a copy-on-write scheme makes them effectively free.

PetaPerl follows the modern behaviour: \$& and friends are safe to use anywhere. The /p flag and \${^MATCH} / \${^PREMATCH} / \${^POSTMATCH} exist for the era between 5.10 and 5.20 when /p-gated copying was a useful optimisation. There is no reason to write new code with them.

Scoping - they are dynamically scoped

The match variables are not lexicals; they behave like dynamically-scoped variables. Every successful match localises a *global match state* to the current dynamic scope. Crucially:

- An *unsuccessful* match does **not** clear the variables.
- A successful match *inside an inner block* overrides them, but only within that block - when the inner block exits, the outer scope's match state is restored.

```
"alpha" =~ /(\w+)/;           # $1 = 'alpha'
{
    "1234" =~ /(\d+)/;         # $1 = '1234' (inside this block)
    print "inner: $1\n";       # 1234
}
print "outer: $1\n";          # alpha - restored
```

This is occasionally surprising: a function you call from inside a regex-handling block does not pollute your \$1 unless that function does its own successful match in the *same* dynamic scope (which is rare unless they are running in the same lexical block).

\${^LAST_SUCCESSFUL_PATTERN}

A read-only reference to the regex that produced the current match state - useful for diagnostics:

```
"hello" =~ /(\w+)/;
print "last pattern was: ${^LAST_SUCCESSFUL_PATTERN}\n";
```

When matches fail - pos

The position on a string after a /g-anchored match is held *not* in one of the variables on this page, but in `pos`:

```
my $s = "1 2 3 4";
while ($s =~ /(\d+)/g) {
    print "matched $1 at ", pos($s) - length($1), "\n";
}
# After the loop, pos($s) is undef.
```

pos is per-string, settable, and is what \G anchors to.

@{^CAPTURE} - captures as an array

The numbered captures, also exposed as a zero-indexed array:

```
"alpha=42" =~ /(\w+)= (\d+)/;
print "name = ${^CAPTURE}[0]\n"; # 'alpha' (same as $1)
print "val  = ${^CAPTURE}[1]\n"; # '42'      (same as $2)
print "n    = scalar @{^CAPTURE}\n"; # 2
```

This is occasionally easier to iterate than \$1, \$2, ..., but most code uses the numbered or named forms directly.

See also

- `m//`, `s///` - the operators that populate every variable on this page.
- `qr//` - compiles a pattern; the resulting regex object can later be matched against and produce these same captures.
- `pos` - the per-string offset for `/g` and `\G`.
- *Regex binding* - `=~`, the operator that decides which string the match runs against.
- *Regular expressions guide*
 - the regex language itself.
- *Groups and captures*
 - the chapter on the capture variables.

Interpreter introspection

Variables that expose the running Perl interpreter and its configuration. Most code never touches them; the cases where they matter are version checks, OS-conditional logic, profiler hooks, and in-place editing.

Variable	Holds
<code>\$]</code>	Perl version as a numeric string (5.042000)
<code>^V</code>	Perl version as a version object (v5.42.0)
<code>^O</code>	Operating system name (linux, darwin, ...)
<code>^X</code>	Path to the running Perl/PetaPerl executable
<code>^T</code>	Process basetime (Unix epoch seconds)
<code>^W</code>	Warnings master switch (boolean)
<code>^I</code>	In-place edit extension (set by <code>-i</code>)
<code>^P</code>	Debugger flag bitmask
<code>^R</code>	Last regex code-block result
<code>^S</code>	Exception-handling state (in eval? undef/0/1)
<code>^{^GLOBAL_PHASE}</code>	Current interpreter phase (START, RUN, END, ...)

`$]` - numeric version

A floating-point representation of the interpreter version. Format: 5.PPPSSS where PPP is the major and SSS the patch:

```
print "$]\n";                # e.g. "5.042000"
warn "old perl" if "$]" < 5.038;
warn "new perl" if "$]" >= 5.040;
```

The `"$]"` stringification before comparison is recommended - it sidesteps subtle floating-point representation issues that can make `$] == 5.042` mysteriously fail.

The English alias `$OLD_PERL_VERSION` is documented but discouraged; `$]` itself is fine.

`^V` - version object

The interpreter version as a [version](#) object, suitable for reading and for richer comparison:

```
print "$^V\n";                # "v5.42.0"
warn "need 5.40+" if $^V lt v5.40.0;
```

The literal `v5.40.0` is a v-string - Perl's native syntax for version constants. Use `$^V` when comparing to v-string literals; use `$]` when comparing to bare numerics.

In PetaPerl, `$^V` reports the underlying Perl-compatibility level (5.42), not the PetaPerl release version.

`^O` - operating system name

```
print "$^O\n";                # "linux"
if ($^O eq 'MSWin32') {
    use_windows_path_separator();
}
```

PetaPerl is Linux-only; `^O` is always `"linux"`. The variable exists for source-compatibility with portable code that branches on it. The full list of values

upstream Perl can produce includes linux, darwin, freebsd, openbsd, MSWin32, cygwin, solaris, and several historical entries.

`$^X` - executable path

The path to the interpreter currently running, suitable for re-invoking:

```
print "$^X\n";           # e.g. "/usr/local/bin/ppperl"

# Rerun the script under the same interpreter, with extra args:
exec $^X, $0, @ARGV, '--debug';
```

`$^X` is what makes `exec $^X, $0, ...` portable - you do not need to know whether the script was started via `ppperl`, `/usr/bin/env perl`, or an absolute path.

`$^T` - basetime

The Unix epoch second at which the program started running:

```
my $uptime = time() - $^T;
print "running for $uptime seconds\n";
```

The classic use is the `-M` and `-A` filetests, which by default report file ages *relative to* `$^T` rather than to `time()`. If you set `$^T = time()` periodically, those filetests recompute against the new reference point - useful in long-running daemons that need to ignore files older than “since I started this batch”.

`$^W` - the warnings switch

The master warnings flag, set by the `-w` command-line switch:

```
$^W = 1;           # all warnings on
$^W = 0;           # all warnings off
```

`$^W` is the Perl-1-era global flag. The modern lexical `use warnings` pragma is the recommended way to control warnings, and operates independently - a `use warnings` block stays on regardless of `$^W`, and a `no warnings` block stays off regardless of `$^W`. The variable mostly matters in code that pre-dates the lexical pragma or that wants to toggle the global default at runtime.

`${^WARNING_BITS}` is the lexical bitmap that `use warnings` reads and writes. Its format is internal; do not rely on specific bit positions.

`$^I` - in-place edit extension

Set by the `-i` command-line switch. When `$^I` is defined, reading from `<>` (the `while (<>)` form) re-opens each file for both read and write and redirects subsequent `print` to the rewritten file:

```
# perl -i.bak -ne 's/\b\w\b/new/g; print' *.txt
# Equivalent in script form:
```

(continues on next page)

(continued from previous page)

```
local $^I = '.bak';
while (<>) {
    s/\bold\b/new/g;
    print;
}
```

`$^I = ''` (empty string) means “rewrite in place without a backup”. `$^I = '.bak'` saves the original as `<filename>.bak`. Setting `$^I = undef` turns in-place editing off.

`$^P` - debugger flag bitmask

Read by the Perl debugger; if non-zero, `DB::DB` is called before every statement. Manipulated by `perl -d` and the `Devel::*` family of debugger and profiler modules.

User code rarely sets `$^P`. The flag bits are documented in upstream `perlvar`; the most commonly seen value is `0x73f` (full debugging).

`$^R` - regex code-block result

The result of the last `(?{...})` or `(?(N)...)` regex code assertion. Lives only as long as the surrounding match:

```
"abc123" =~ /(?(<word>\w+?)(\d+)(?{ length($1) + length($2) }))/;
print "$^R\n";                                     # 6 - the code block's last
↪expression
```

PetaPerl supports `(?{ })` regex code blocks; see the *performance chapter* of the regex guide for the language details.

`$^S` - exception-handling state

Reports whether the interpreter is currently inside an `eval`. It holds one of three values:

<code>\$^S</code>	Meaning
<code>undef</code>	Parsing a module, an <code>eval</code> , or the main program
<code>1 (true)</code>	Executing inside an <code>eval</code> or <code>try</code> block
<code>0 (false)</code>	Anywhere else

The `undef` state is the one that matters: it can occur inside a `$SIG{__DIE__}` or `$SIG{__WARN__}` handler, where the interpreter is compiling rather than running. A handler that needs to behave differently during compilation tests for it:

```
local $SIG{__DIE__} = sub {
    return if $^S;                # real runtime exception - let it
↪propagate
    warn "compile-time failure: $_[0]";
};
```


The legacy long name (use English) is `$EXCEPTIONS_BEING_CAUGHT`, which is slightly misleading: the `undef` value does not tell you whether exceptions are being caught, because compiling the main program does not catch them.

`${^GLOBAL_PHASE}` - current interpreter phase

The current life-cycle phase of the Perl interpreter:

Phase	When
CONSTRUCT	Inside the C-level interpreter constructor
START	BEGIN blocks, <code>use</code> , <code>require</code> , top-level compile
CHECK	After compilation, before INIT
INIT	INIT blocks
RUN	Main program execution
END	END blocks
DESTRUCT	Global destruction

```
END {
    if (${^GLOBAL_PHASE} eq 'END') {
        log_shutdown();           # we are still in user END
    }
}

# Inside a destructor, distinguish "object cleanup during normal
# scope exit" from "object cleanup during interpreter shutdown":
sub DESTROY {
    my $self = shift;
    if (${^GLOBAL_PHASE} eq 'DESTRUCT') {
        # global destruction - order is undefined, file handles
        # may already be torn down - be conservative
        return;
    }
    $self->{handle}->flush;
}
```

The “don’t run during global destruction” guard inside `DESTROY` is the variable’s classic use case. During `DESTRUCT`, the order of object teardown is not defined; an object’s references may already be invalid.

See also

- `use VERSION` - the conventional way to require a minimum Perl version, replacing `if "$]" < ....`
- `require VERSION` - same as `use VERSION` but at runtime.
- `Config` - the full Perl build configuration, much richer than `${^O}` and `$]`.
- `use warnings` - the lexical pragma that supersedes `${^W}` for new code.
- *Command-line switches* - documents `-i` (`${^I}`), `-w` (`${^W}`), `-d` (`${^P}`).

Program input - @ARGV, %ENV, @INC

Three globals carry input *into* the program: command-line arguments, environment variables, and the module search path. They share enough context that decisions about one usually inform the others - @ARGV interacts with \$0, %ENV's Perl-relevant keys influence @INC, and @INC's ordering question is mirrored by similar concerns for @ARGV consumers and shell PATH.

Variable	Holds
@ARGV	Command-line arguments (excluding script name)
\$0	Script name (paired with @ARGV)
%ENV	Process environment
@INC	Module search path (used by use/require)
%INC	Already-loaded modules
\$INC	Current @INC index during a hook (5.37.7+)

@ARGV and its relationship to \$0

After Perl starts, the *script name* is in \$0 and the *script's arguments* are in @ARGV:

```
$ pperl myscript.pl -v --output /tmp/x foo bar
```

```
print "$0\n";           # myscript.pl
print "@ARGV\n";        # -v --output /tmp/x foo bar
print scalar(@ARGV), " args\n"; # 5 args
```

\$ARGV[0] is the *first script argument*, **not** the script name itself. This differs from C's argv (where argv[0] is the program name); Perl exposes the program name separately via \$0.

Standard argument handling

Hand-written arg parsing is fine for trivial scripts:

```
my %opt;
while (@ARGV && $ARGV[0] =~ /^-/ ) {
    my $flag = shift @ARGV;
    last if $flag eq '--';
    if ($flag eq '-v') { $opt{verbose} = 1 }
    elsif ($flag eq '--output') { $opt{output} = shift @ARGV }
    else { die "unknown flag: $flag\n" }
}
my @files = @ARGV;
```

For anything beyond toy scripts, the Getopt::Long module handles every standard pattern (long options, abbreviation, mandatory arguments, repeated flags, -- separator, automatic help):

```

use Getopt::Long;
my %opt;
GetOptions(
    'verbose|v'    => \%opt{verbose},
    'output|o=s'   => \%opt{output},
    'help|h'       => \%opt{help},
) or die "bad options; try --help\n";
my @files = @ARGV;          # what's left after option_
    ↪ processing

```

GetOptions mutates @ARGV in place - when it returns, @ARGV contains only the non-option arguments.

@ARGV and the diamond operator

The <> (diamond) operator iterates lines from the files named in @ARGV, falling back to STDIN if @ARGV is empty:

```

while (<>) {
    # one line from one of the @ARGV files (or STDIN)
    chomp;
    process($_);
}

```

This is the default contract of every awk-style filter program. <> opens each file in turn, sets \$ARGV to its name during reading, and closes it at EOF.

-i and the other - switches (-n, -p, -l, -a) all build their behaviour on this @ARGV → <> pattern.

Modifying @ARGV before <>

You can pre-process @ARGV to alter what <> reads:

```

# Add an extra file to the front of the argv list:
unshift @ARGV, 'header.txt';

# Process compressed files transparently by rewriting argv:
@ARGV = map { /\.gz$/ ? "gunzip -c $_|" : $_ } @ARGV;

while (<>) {
    process($_);
}

```

The cmd | form turns the argument into a pipe-from-command open; this is a long-standing Perl idiom for transparent decompression in awk-style scripts.

%ENV - the process environment

%ENV is the hash of environment variables inherited from the parent process (typically the shell). Reading is straightforward:

```
my $home    = $ENV{HOME};
my $term    = $ENV{TERM} // 'dumb';
my $path    = $ENV{PATH};
```

Writing changes the environment seen by *child processes* that this Perl script subsequently spawns:

```
$ENV{LC_ALL} = 'C';           # POSIX locale for child commands
$ENV{TZ}     = 'UTC';
$ENV{LANG}   = 'C';
delete $ENV{LC_NUMERIC};     # remove a key entirely

system('date');              # child sees the modified ENV
```

The change does *not* propagate back to the parent - the shell that started the script keeps its original environment. If you need the parent shell to pick up new values, write them to a file or have the shell eval your output (the standard dotenv pattern).

Stringification of values

As of Perl 5.18, %ENV values are always stringified at assignment time. Storing a reference no longer round-trips:

```
my $arr = [1, 2, 3];
$ENV{DATA} = $arr;
print $ENV{DATA};           # "ARRAY(0x...)" - stringified
```

Environment variables are a string-typed interface to the operating system; preserving structure across a fork/exec boundary would not work anyway. Marshal anything non-trivial as JSON or similar before storing.

Perl-specific environment variables

A small set of environment variables are honoured by Perl itself, *not* by user scripts:

Variable	Effect
PERL5LIB	Colon-separated paths prepended to @INC (subject to taint rules)
PERLLIB	Same, older variable; used only if PERL5LIB is unset
PERL5OPT	Switches applied as if on the command line (subset only)
PERLDB_OPTS	Options for the debugger
PERL_UNICODE	Default UTF-8 layer settings (mirrors -C)
PERL_USE_UNSAFE_INC	Re-add . to @INC (removed by default since 5.26 - strongly discouraged)
PERL_HASH_SEED	Per-run hash randomisation seed
PERL_SIGNALS	unsafe to opt out of deferred signal handling
HOME	Read by ~ glob expansion; not Perl-specific but Perl-relevant
PATH	Searched by system/exec for unqualified commands

In tainted mode (-T), PERL5LIB, PERL5OPT, and PERLLIB are ignored to prevent privilege escalation. See the [command-line switches guide](#) for the full security model.

delete \$ENV{KEY} versus \$ENV{KEY} = undef

The two are *not* equivalent:

```
delete $ENV{KEY};           # KEY is not in the environment
$ENV{KEY} = undef;         # KEY is in the environment, value
                             ↪ ""
```

A child process started after delete will not see KEY at all

- `getenv("KEY")` returns NULL. After undef assignment, the child sees KEY="" (empty string). Code that distinguishes “unset” from “set to empty” cares about this difference.

@INC - the module search path

When `use` and `require` look up a module, they walk @INC left to right and stop at the first match:

```
print "$_\n" for @INC;
# /usr/local/lib/perl5/site_perl/5.42.0/x86_64-linux
# /usr/local/lib/perl5/site_perl/5.42.0
# /usr/local/lib/perl5/5.42.0/x86_64-linux
# /usr/local/lib/perl5/5.42.0
# (no trailing "." since Perl 5.26)
```

@INC is initialised at startup from a combination of:

1. Compiled-in defaults (the build-time `installprivlib`, `installsitelib`, ...).
2. `-I /path` switches on the command line, *prepended* in order.
3. The PERL5LIB (or PERLLIB) environment variable, also prepended.

After startup it is just an ordinary array. *Order matters*: a match in an early element wins.

use lib versus unshift @INC versus push @INC

Three ways to add a path. They are not equivalent:

```
use lib '/my/lib';           # unshift at compile-time
unshift @INC, '/my/lib';     # unshift at runtime
push @INC, '/my/lib';        # push at runtime
```

The differences:

- **use lib** runs at *compile time*. It happens before any use statement that follows it on the page. This is what you almost always want when you’re saying “this script needs a custom library directory”:

```
use lib '/opt/myapp/lib';
use MyApp::Module;           # found because lib was
↪unshifted first
```

Internally `use lib '/path'` does `BEGIN { unshift @INC, '/path' }`, so the new path is searched *first* - a copy of a module in `/opt/myapp/lib` shadows one in the system `@INC`.

- **unshift @INC, '/path'** at the top level has *runtime* semantics. It works for require calls that happen after the unshift, but a use statement on a later line of the same file *will not see it* (because use is itself compile-time):

```
unshift @INC, '/my/lib';
use Some::Module;           # NOT found in /my/lib -
↪too early
```

This is a frequent gotcha. `BEGIN { unshift @INC, '/my/lib' }` fixes it; `use lib` is the cleaner spelling of exactly that.

- **push @INC, '/path'** appends. The new path is searched *last*. This is the right choice when you want your library to be a fallback - used only if no other location has the module
 - for example, providing a built-in version of a CPAN module that the user might have installed on the system.

The rule of thumb: *always* use `use lib` for “I want my code’s module path to take effect at compile time”, which is almost every case.

@INC hooks

@INC may contain not only paths but also code references and blessed objects. When require encounters one of these, it calls it as a hook - passing the requested module path - and the hook returns either an open filehandle or a list of source code, generators, etc. This is how mechanisms like `Module::Pluggable`, `PAR`, and `Test::MockModule` intercept module loading.

The full hook protocol is documented under [require](#). Most users never write one; you read about them when debugging “why does this module not load from where I expect” problems.

%INC - the loaded-modules cache

After a module is successfully loaded, %INC records it. The key is the path Perl was asked to find (e.g. Foo/Bar.pm); the value is the absolute path that satisfied the request:

```
use Data::Dumper;
print "Data::Dumper loaded from $INC{'Data/Dumper.pm'}\n";
# /usr/local/lib/perl5/site_perl/5.42.0/Data/Dumper.pm

# Show every loaded module:
for my $key (sort keys %INC) {
    print "$key → $INC{$key}\n";
}
```

require checks %INC before walking @INC - a second `require Foo`; is a no-op because Foo.pm is already in the cache. To force a re-require:

```
delete $INC{'Foo.pm'};
require Foo;           # actually re-runs Foo.pm
```

This is the standard idiom for testing reload behaviour. Be aware that re-running a module file does **not** un-define what the first run created - packages, subs, and globals stay around.

\$INC - the index inside an @INC hook

Available since Perl 5.37.7. When an @INC hook is being called, \$INC is set to the index of the hook in @INC. After the hook returns, the iterator advances based on `$INC + 1` - so the hook can rewrite @INC and steer the search to a specific position afterwards. This is an advanced feature; everyday code never touches it.

See also

- [use, require](#) - the consumers of @INC/%INC.
- [use lib](#) - the canonical compile-time spelling for prepending to @INC.
- [Getopt::Long](#) - the standard parser for @ARGV.
- [readline](#) - the function form of the diamond <> operator that drives @ARGV → STDIN filtering.
- [-i, -n, -p, -a](#) - command-line switches built on @ARGV.
- [Process variables](#) - \$0 is the script name; \$\$ is the PID; pair these with @ARGV for self-restart and logging.
- [Command-line one-liners](#) - the tutorial showing every @ARGV/<>/switch combination in context.

Signal handling - %SIG

%SIG is the hash where Perl exposes signal handlers. Setting \$SIG{NAME} installs a handler for signal NAME; deleting it restores the default action. Two pseudo-signals - `__WARN__` and `__DIE__` - are not real OS signals at all but interpreter hooks into `warn` and `die`.

The signal table

Real OS signals come from the POSIX and `signal.h` lists. The common ones - keys you will actually use:

Key	Signal	Default action	When you handle it
INT	SIGINT	terminate	Ctrl-C: graceful shutdown
TERM	SIGTERM	terminate	kill <pid>: graceful shutdown
HUP	SIGHUP	terminate	terminal hangup; daemons reload config
QUIT	SIGQUIT	terminate + core	Ctrl-: like INT but with a core dump
PIPE	SIGPIPE	terminate	wrote to a closed pipe; usually ignored
CHLD	SIGCHLD	ignore	a child process died (waitpid here)
USR1	SIGUSR1	terminate	application-defined event
USR2	SIGUSR2	terminate	application-defined event
ALRM	SIGALRM	terminate	alarm() timer expired
WINCH	SIGWINCH	ignore	terminal window resized
IO	SIGIO	terminate	async I/O ready

The full list - including signals you cannot catch (KILL, STOP) and platform-specific signals - comes from the `POSIX` module, which exports the numeric constants (SIGINT, SIGTERM, ...) and the WIFEXITED-family wait macros.

Installing a handler

Three handler-value forms are accepted:

```
$SIG{INT} = \&handler;      # code reference - preferred
$SIG{INT} = sub { ... };     # anonymous sub - also fine
$SIG{INT} = 'IGNORE';        # ignore the signal
$SIG{INT} = 'DEFAULT';        # restore default action
$SIG{INT} = '';               # same as DEFAULT
$SIG{INT} = undef;            # same as DEFAULT
```

Don't use a *bareword* (`$SIG{INT} = 'handler'`) - that references a *name* in the `main::` package, which Perl looks up each time the signal arrives. If you rename handler, the bareword form silently keeps targeting the old name.

The handler receives one argument - the signal name as a string:

```
sub handler {
    my ($signame) = @_;
    print "received SIG$signame\n";
}
```

(continues on next page)

(continued from previous page)

```
$SIG{INT}  = \&handler;
$SIG{TERM} = \&handler;
```

Safe signals - the deferred-delivery rule

Since Perl 5.8, signal delivery is **deferred**. When a signal arrives, the OS sets a flag in the interpreter; the actual Perl handler runs only at the next *safe point* - typically between op-code executions. This means:

- A handler **cannot** interrupt a read, open, connect, or any other syscall mid-call. Slow syscalls return early with EINTR in \$! and the handler runs after.
- A handler **cannot** interrupt a regex match in progress. A long-running regex on a malicious input will not be cut short by SIGINT.
- The handler runs in regular Perl context. It can call any Perl code, allocate memory, throw exceptions - none of the re-entrancy hazards of C signal handlers.

This is the right trade-off for almost every program. If you genuinely need pre-5.8 *unsafe* signal handling - handlers that run synchronously in the signal context, with all the re-entrancy hazards - set the PERL_SIGNALS=unsafe environment variable before starting Perl. Do not do this casually.

PetaPerl implements deferred-delivery signal handling.

The graceful-shutdown pattern

The standard shape: a flag set by the handler, checked by the main loop:

```
my $shutting_down = 0;
$SIG{INT}  = sub { $shutting_down = 1 };
$SIG{TERM} = sub { $shutting_down = 1 };

while (my $job = $queue->next) {
    last if $shutting_down;      # check at a safe point
    process($job);
}

if ($shutting_down) {
    log_and_close();
    exit 0;
}
```

The main loop polls the flag at a place where it is safe to unwind cleanly. The handler does *only* “set the flag” - no logging, no I/O - minimising the time it spends running.

For applications that block on a long syscall (accept, read, select), the loop has to be structured to retry on EINTR:

```
my $shutting_down = 0;
$SIG{INT}  = sub { $shutting_down = 1 };
```

(continues on next page)

(continued from previous page)

```

while (!$shutting_down) {
    my $client = accept($srv, $sock);
    unless (defined $client) {
        next if ${EINTR};           # signal woke us up - re-check_
    }
    die "accept: $!";
    handle($client);
}

```

`${EINTR}` (see [error variables](#)) tells you the syscall exited because a signal arrived. The flag was set by the handler; the main loop sees the wake-up, re-checks `$shutting_down`, and exits cleanly.

Reaping child processes - SIGCHLD

When a child process exits, the parent gets SIGCHLD. The handler must call `waitpid` to *reap* the zombie:

```

$SIG{CHLD} = sub {
    while ((my $kid = waitpid(-1, WNOHANG)) > 0) {
        my $status = $?;
        log_child_death($kid, $status);
    }
};

```

The while loop with WNOHANG (from `POSIX :sys_wait_h`) handles the case where two children exit close together - only one SIGCHLD is delivered (it is not queued), so the handler must drain *all* pending exits each time it runs.

`$SIG{CHLD} = 'IGNORE'` is a special case: it does not just ignore the signal, it tells the kernel to auto-reap children. Use this when your script forks children whose exit status you do not care about.

SIGPIPE - the closed-pipe case

When you print to a pipe whose reader has gone away, the kernel sends SIGPIPE. The default action is to terminate the process - which is rarely what you want. Most filter programs should ignore SIGPIPE and check the print return value instead:

```

local $SIG{PIPE} = 'IGNORE';

print {$child_pipe} $data
    or die "write to pipe: $!"; # $! will be EPIPE if the reader_
    closed

```

With the handler ignored, the syscall returns failure with EPIPE rather than killing the program.

Timeouts via SIGALRM

Combined with `alarm` and an `eval` block, SIGALRM provides a portable timeout for any blocking operation:

```

sub with_timeout {
    my ($seconds, $code) = @_;
    my $result;
    eval {
        local $SIG{ALRM} = sub { die "TIMEOUT\n" };
        alarm $seconds;
        $result = $code->();
        alarm 0;                # cancel the timer
        1;
    } or do {
        die "$@" unless $@ eq "TIMEOUT\n";
        return undef;
    };
    return $result;
}

my $line = with_timeout(5, sub { <$slow_socket> });

```

The pattern:

1. `local $SIG{ALRM}` - handler is in scope only for this call.
2. `alarm $seconds` - schedule the signal.
3. Perform the blocking work.
4. `alarm 0` - cancel the timer if work succeeded.
5. Inside the handler, `die "TIMEOUT\n"` - the trailing `\n` suppresses the “at line N” suffix and makes the error value easy to test against.

__WARN__ and __DIE__ - interpreter hooks

These two keys in %SIG are *not* OS signals. They are called when the interpreter is about to emit a warning (`warn`) or throw a fatal exception (`die`).

\$SIG{__WARN__} - warning interception

```

my @warnings;
{
    local $SIG{__WARN__} = sub {
        my ($msg) = @_;
        push @warnings, $msg;
    };
    do_thing_that_might_warn();
}
# Now @warnings holds every warning that fired inside the block,
# without anything having printed to STDERR.

```

A `__WARN__` handler that does nothing silences the warning entirely. A handler that calls `die` upgrades warnings into exceptions:

```
local $SIG{__WARN__} = sub { die $_[0] };
eval { something_warny() };      # warns become catchable
```

The lexical pragma `use warnings FATAL => 'all'` is a more targeted way to do this - it makes warnings fatal only in the lexical scope where the pragma is active. Use it instead of `__WARN__` for “treat warnings as errors” requirements.

`$SIG{__DIE__}` - die interception

A `__DIE__` handler runs when `die` is about to throw - both for uncaught exceptions and inside `eval`. Inside an `eval`, the handler runs *before* the exception is caught, which is the classic source of confusion: a global `__DIE__` handler that logs every “death” will fire on every `eval { ... }` that calls `die` for ordinary control flow.

```
$SIG{__DIE__} = sub { print STDERR "dying: $_[0]" };
eval { die "expected error" };      # the handler still runs!
```

If you want a global death logger, gate it on `^S`:

```
$SIG{__DIE__} = sub {
    return if ^S;                  # inside eval - let it be caught
    print STDERR "uncaught: $_[0]";
};
```

Realistically, **avoid `$SIG{__DIE__}` for new code**. The pragma `use warnings` and the *error variables* cover what you need without the surprise. The modern `try/catch` feature handles exceptions explicitly without the action-at-a-distance of `$SIG{__DIE__}`.

Listing available signals

```
use Config;
my @signals = split / /, $Config{sig_name};
print "available signals: @signals\n";
# HUP INT QUIT ILL TRAP ABRT BUS FPE KILL ...
```

Or, for the numeric→name mapping:

```
use Config;
my @names = split / /, $Config{sig_name};
my @nums = split / /, $Config{sig_num};
my %sig_num = map { $names[$_] => $nums[$_] } 0..$#names;
my %sig_name = map { $nums[$_] => $names[$_] } 0..$#names;
```

This is what most signal-listing utilities do internally.

See also

- [POSIX](#) - the canonical source of SIG* numeric constants and the WIFEXITED/WEXITSTATUS macros for decoding child status in a SIGCHLD handler.
- [kill](#) - the partner of %SIG: send a signal to a process (typically to test your own handler).
- [alarm](#) - schedule a SIGALRM; the basis of timeout patterns.
- [waitpid](#) - reap a child after SIGCHLD.
- [fork](#) - produces the children whose deaths SIGCHLD reports.
- [Error variables](#) - \$! after EINTR, \$? after waitpid, \$@ after die from inside a handler.
- [Interpreter introspection](#) - \$^S for the __DIE__ “are we inside eval” check.
- [try/catch](#) - the modern alternative to \$SIG{__DIE__}-based exception handling.
- [Default variables](#) - \$_, @_. The implicit operand of most built-ins; the argument array of every sub.
- [Process identity](#) - \$0, \$\$, \$<, \$>, \$(, \$). Program name, PID, real and effective UIDs/GIDs.
- [Error state](#) - \$!, \$@, \$?, \$^E. The four error variables, each reporting from a different layer (C library, eval/die, child process, OS-extended).
- [Filehandle and printing state](#) - \$/, \$\\, \$,, \$", \$|, \$., \$;. Input record separator, output separators, autoflush, line counter, subscript joiner.
- [Regex match variables](#) - \$&, \$`, \$', \$1..\$N, %+, %-, @+, @-. The result of the last successful match; the performance story behind \$& and friends.
- [Interpreter introspection](#) - \$], \$^V, \$^O, \$^X, \$^T, \$^W, \$^I, \$^P. Version, OS, executable path, basetime, warnings, in-place edit, debugger flags.
- [Program input](#) - @ARGV, %ENV, @INC, %INC, \$0. How the script gets its arguments, environment, and module search path; the use lib vs unshift @INC story.
- [Signal handling](#) - %SIG, \$SIG{__WARN__}, \$SIG{__DIE__}. The signals you can catch, the safe-signals rule, and the standard graceful-shutdown shape.

Variable namespaces

Each special variable lives in **one** specific namespace. Most are in `main::`; some (@F, @ISA) are package-local. Every variable’s documentation says which.

The naming convention is a punctuation choice and a sigil:

- `$<digit>` - regex capture variables.
- `$<symbol>` - single-character variables (`$_`, `$!`, `$@`, `$/`, ...).
- `$^<letter>` - control-letter variables (`$^V`, `$^W`, `$^X`). Spelled with a literal caret followed by an uppercase letter.
- `${^NAME}` - extended-name variables (`${^GLOBAL_PHASE}`, `${^CAPTURE}`). The braces and caret are required.

- `$NAME` after `use English` - the named alias for almost every punctuation variable. `$ERRNO` is `$!`, `$EVAL_ERROR` is `$@`, etc.

The English module

```
use English;                # all aliases, plus $PREMATCH/$MATCH/  
                             ↳$POSTMATCH  
use English qw( -no_match_vars );    # skip $`, $&, $' aliases
```

English provides readable aliases for the punctuation variables: `$EVAL_ERROR` for `$@`, `$INPUT_RECORD_SEPARATOR` for `$/`, and so on. The aliases are real Perl variables that share storage with the originals - assigning to one writes the other.

The `-no_match_vars` import is recommended in code that doesn't explicitly need `$&`, `$``, or `$'`. See [regex match variables](#) for the historical performance reason and why it no longer matters on modern perls.

This page lists punctuation forms throughout because most code in the wild is written that way. The English aliases are noted alongside on the family pages where they are useful.

Localising special variables

Almost every special variable that controls a built-in's behaviour (`$/`, `$\`, `$,`, `$"`, `$|`, `$_`) is global - set it in one place and every other piece of code in the same process sees the change. The standard idiom for "I need a different value just for this block" is `local`:

```
sub slurp {  
    my ($path) = @_;  
    open my $fh, '<', $path or die "open $path: $!";  
    local $/;                # slurp mode, only inside this  
    ↳sub  
    return <$fh>;  
}
```

Without the `local`, the caller's `$/` would be permanently clobbered. `local` saves the current value, sets the new one, and restores the old value when the enclosing scope exits - even on an exception. This applies to every special variable on the family pages, except the read-only ones (regex match variables, `$0`'s formal argument, the version variables).

See also

- [perlop](#) - every operator that reads or writes a special variable as part of its definition (regex binding, printing, range, comparison).
- [perlfunc](#) - built-in functions, almost all of which read a default from one of these variables when called with no argument (`chomp`, `chop`, `print`, `lc`, `length`, `<>`, ...).
- [Regular expressions guide](#)
 - the regex language behind the match variables.
- [Boolean Logic for Perl Programmers](#)

- the conceptual companion for the `||||` defaulting idioms that recur throughout this reference.

5.1.2 Built-in Functions

Alphabetical index of every Perl 5 built-in. The content mirrors `perldoc perlfunc` with pperl-specific divergences flagged inline on each detail page.

For an alternate browsing view grouped by problem domain see [Built-in Functions by Category](#).

A

- **abs** — Return the absolute value of a number.
- **accept** — Accept an incoming connection on a listening socket.
- **alarm** — Schedule a SIGALRM to be delivered to the current process after a whole number of wallclock seconds.
- **all** — Test whether BLOCK returns true for **every** element of LIST.
- **any** — Return true if BLOCK yields true for at least one element of LIST.
- **atan2** — Arc tangent of Y/X in the range $-\pi$ to π .

B

- **bind** — Attach a local address to a socket.
- **binmode** — Set the I/O layer stack on a filehandle - typically to make it deliver raw bytes, or to attach a character encoding.
- **bless** — Mark the thing a reference points at as an object in a package.
- **break** — Break out of a given block.

C

- **caller** — Return information about the subroutine, `eval`, or `require` that called the currently executing code.
- **catch** — Handle an exception thrown by a preceding `try` block.
- **chdir** — Change the process's current working directory.
- **chmod** — Change the permission bits of a list of files.
- **chomp** — Strip the trailing input record separator from a string in place.
- **chop** — Remove the last character from a string and return it.
- **chown** — Change the owner and group of a list of files.
- **chr** — Return the character whose codepoint is the given number.
- **chroot** — Change the root directory of the current process and every child it later spawns.
- **class** — Declare a namespace that behaves as a native object class.

- **close** — Close a filehandle, flush its buffers, and release the underlying file descriptor.
- **closedir** — Close a directory handle opened by `opendir`.
- **connect** — Initiate a connection from a socket to a remote address.
- **continue** — Attach a block to a loop that runs after every iteration, just before the condition is re-tested.
- **cos** — Return the cosine of a number given in radians.
- **crypt** — One-way passwd(5)-style hashing of a plaintext string with a salt.

D

- **dbmclose** — Break the binding between a DBM file and a hash previously attached by `dbmopen`.
- **dbmopen** — Bind a DBM file on disk to a hash so hash reads and writes become lookups and stores in the database.
- **defer** — Schedule a block to run when the enclosing scope exits, for any reason.
- **defined** — Test whether a value, variable, or subroutine is defined.
- **delete** — Remove the named key-value pair(s) from a hash, or element(s) from an array, and return what was removed.
- **die** — Raise an exception.
- **do** — Execute a block of code or run a Perl source file as if it were part of the current program.
- **dump** — Cause the running program to produce a core dump immediately.

E

- **each** — Step through a hash or array one entry at a time.
- **endgrent** — Close the group database after iterating it.
- **endhostent** — Close the hosts database after iteration.
- **endnetent** — Close the networks database after iteration.
- **endprotoent** — Close the protocols database after iteration.
- **endpwent** — Close the passwd-database iterator opened by `getpwent` or `setpwent`.
- **endservent** — Close the services database after a walk with `getservent`.
- **eof** — Test a filehandle for end-of-file.
- **eval** — Run a piece of Perl code with any fatal error trapped instead of killing the program.
- **evalbytes** — Compile and run a string of Perl source, forcing the source to be interpreted as **bytes** rather than characters.
- **exec** — Abandon this program and run another in the same process.

- **exists** — Test whether a hash or array element, or a named subroutine, is present - without creating it and without caring what it holds.
- **exit** — Terminate the program with a status value.
- **exp** — Raise *e* to a power.

F

- **fc** — Return the Unicode casefolded form of a string for case-insensitive comparison.
- **fcntl** — Perform a `fcntl(2)` file-control operation on a filehandle.
- **field** — Declare a per-instance variable inside a `class` block.
- **fileno** — Return the OS-level file descriptor number behind a filehandle.
- **finally** — Run cleanup code on the way out of a `try/catch`, whether the body succeeded, threw, or jumped away.
- **flock** — Place an advisory lock on an open file.
- **fork** — Create a new process running the same program at the same point.
- **format** — Declare a picture-based report template for use by `write`.
- **formline** — Format a list of values into a picture string and append the result to the format accumulator `$^A`.

G

- **getc** — Read the next single character from a filehandle.
- **getgrent** — Read the next entry from the system group database.
- **getgrgid** — Look up a group record by numeric group ID.
- **getgrnam** — Look up a Unix group by name and return its `/etc/group` record.
- **gethostbyaddr** — Look up a host record by its packed IP address.
- **gethostbyname** — Look up a host record by DNS name.
- **gethostent** — Fetch the next entry from the host database.
- **getlogin** — Return the login name of the user associated with the controlling terminal.
- **getnetbyaddr** — Look up a network record by its numeric network address.
- **getnetbyname** — Look up a network record by name.
- **getnetent** — Read the next entry from the networks database.
- **getpeername** — Return the address of the remote end of a connected socket.
- **getpggrp** — Return the POSIX process-group ID that a process belongs to.
- **getppid** — Return the process ID of the current process's parent.
- **getpriority** — Return the current scheduling nice value of a process, a process group, or a user.

- **getprotobyname** — Look up an IP protocol by its textual name and return its database entry.
- **getprotobynumber** — Look up a network protocol entry by its assigned protocol number.
- **getprotoent** — Fetch the next entry from the protocols database.
- **getpwent** — Return the next entry from the system password database.
- **getpwnam** — Look up a user's passwd record by login name.
- **getpwuid** — Look up a user's passwd record by numeric UID.
- **getservbyname** — Look up a network service by its textual name and protocol.
- **getservbyport** — Look up a network service by its numeric port and protocol.
- **getservent** — Read the next entry from the system services database.
- **getsockname** — Return the local address of a connected or bound socket.
- **getsockopt** — Read one socket option out of the kernel as an opaque packed string.
- **glob** — Expand a shell-style filename pattern into the list of matching paths.
- **gmtime** — Convert an epoch time to a broken-down UTC time, either as a 9-element list or as a `ctime(3)`-style string.
- **goto** — Transfer execution elsewhere in the program without returning.
- **grep** — Filter a list to the elements where the block or expression is true.

H

- **hex** — Interpret a string as a hexadecimal numeral and return its numeric value.

I

- **import** — Populate the caller's namespace with names a module chooses to export.
- **index** — Find the position of a substring inside a string.
- **int** — Return the integer portion of a number, truncating toward zero.
- **ioctl** — Perform a device-control `ioctl(2)` system call on a filehandle.
- **isa** — Test whether an object is an instance of a class or of any subclass derived from it.

J

- **join** — Concatenate a list of strings with a separator between each adjacent pair, and return the single resulting string.

K

- **keys** — List all keys of a hash, or all indices of an array.
- **kill** — Send a signal to a list of processes.

L

- **last** — Exit a loop immediately, skipping the rest of the body and the `continue` block.
- **lc** — Return a lowercased copy of a string.
- **lcfirst** — Return a copy of a string with its first character lowercased.
- **length** — Return the number of characters in a string.
- **link** — Create a hard link from NEWFILE to the existing OLDFILE.
- **listen** — Mark a socket as passive so it can accept incoming connections.
- **local** — Save the current value of a package variable and restore it when the enclosing scope exits.
- **localtime** — Convert an epoch time to a broken-down calendar time in the local time zone.
- **lock** — Place an advisory lock on a shared variable, array, hash, or subroutine until the lock goes out of scope.
- **log** — Return the natural logarithm (base *e*) of a number.
- **lstat** — Return the 13-element status list for a path **without** following a symbolic link.

M

- **m//** — Search a string for a pattern and report whether - and what - it matched.
- **map** — Apply a block or expression to each element of a list and return the flattened results.
- **method** — Declare a named instance method inside a `class` block.
- **mkdir** — Create a single directory on the filesystem.
- **msgctl** — Perform a control operation on a System V IPC message queue.
- **msgget** — Create or look up a System V IPC message queue and return its id.
- **msgrcv** — Receive a message from a System V IPC message queue.
- **msgsnd** — Send a message to a System V IPC message queue.
- **my** — Declare one or more lexically scoped variables.

N

- **next** — Start the next iteration of the enclosing loop immediately.
- **no** — The compile-time inverse of **use** - call a module's `unimport` method to turn off what `use` turned on.

O

- **oct** — Interpret a string as a number written in octal, hexadecimal, or binary, and return the resulting integer.
- **open** — Associate a filehandle with a file, a command, or an in-memory scalar.
- **opendir** — Open a directory for reading.
- **ord** — Return the Unicode code point of the first character of a string.
- **our** — Declare a lexically scoped alias to a package variable.

P

- **pack** — Convert a list of Perl values into a binary string according to a template.
- **package** — Declare the compile-time namespace for the declarations that follow.
- **pipe** — Open a pair of connected filehandles - one for reading, one for writing.
- **pop** — Remove and return the last element of an array.
- **pos** — Report or set where the next `/g` regex match will resume in a string.
- **print** — Write a list of values to a filehandle.
- **printf** — Write a formatted string to a filehandle.
- **prototype** — Return the prototype string of a subroutine, or **undef** if it has none.
- **push** — Append one or more values to the end of an array.

Q

- **q//** — Single-quoted string literal with a choice of delimiter.
- **qq//** — Build a double-quoted, interpolated string with a delimiter you pick.
- **qr//** — Compile a pattern once and hand back a reusable regex object.
- **quotemeta** — Return a copy of a string with every regex-significant character backslash-escaped, so the result can be interpolated into a pattern and match its own literal content.
- **qw//** — Build a list of barewords by splitting a delimited string on whitespace, without quoting or comma-separating each word.
- **qx//** — Run a shell command and capture its standard output.

R

- **rand** — Return a pseudo-random floating-point number in the half-open range `[0, EXPR)`.
- **read** — Read a fixed amount of buffered input from a filehandle into a scalar.
- **readdir** — Read the next entry, or all remaining entries, from a directory handle opened by `opendir`.
- **readline** — Read one or more records from a filehandle.
- **readlink** — Return the target path a symbolic link points to.
- **readpipe** — Run a shell command and return its standard output.
- **recv** — Read an incoming message from a socket into a scalar.
- **redo** — Restart the current iteration of a loop without re-testing the condition and without running the `continue` block.
- **ref** — Return a string describing what a reference points to.
- **rename** — Change the name of a file.
- **require** — Load a Perl source file at runtime, or demand a minimum Perl version.
- **reset** — Clear every package variable whose name begins with one of a set of letters, and re-arm one-shot `m?pattern?` matches.
- **return** — Leave the current subroutine, `eval`, `do FILE`, `sort` block, or regex eval block, yielding a value to the caller.
- **reverse** — Reverse a list - or, in scalar context, reverse the characters of a string.
- **rewinddir** — Reset a directory handle to the start of its listing.
- **rindex** — Find the position of the **last** occurrence of a substring inside a string.
- **rmdir** — Remove an empty directory.

S

- **s///** — Search a string for a pattern and replace every match with a replacement.
- **say** — Print a list of values followed by a newline.
- **scalar** — Force EXPR to be evaluated in scalar context and return its value.
- **seek** — Reposition a filehandle for random-access reads or writes.
- **seekdir** — Restore a directory handle to a position previously captured by `telldir`.
- **select** — Either set the default output filehandle, or call the `select(2)` syscall for I/O multiplexing. Same name, two unrelated jobs - disambiguated by argument count.
- **semctl** — Perform a control operation on a System V semaphore set.
- **semget** — Create or look up a System V semaphore set and return its identifier.
- **semop** — Perform one or more System V semaphore operations atomically.

- **send** — Send a message on a socket.
- **setgrent** — Rewind the group-database iterator back to the first entry.
- **sethostent** — Open or rewind the hosts database for iteration.
- **setnetent** — Open or rewind the networks database for iteration.
- **setpgrp** — Set the process group of a process.
- **setpriority** — Set the scheduling priority (nice value) of a process, process group, or user.
- **setprotoent** — Open the protocols database and prepare it for sequential reads.
- **setpwent** — Rewind the password-database iterator so the next **getpwent** call returns the first entry again.
- **setservent** — Rewind the services database and optionally keep it open across lookups.
- **setsockopt** — Set a kernel-level option on an open socket.
- **shift** — Remove and return the first element of an array.
- **shmctl** — Control or query a System V shared memory segment.
- **shmget** — Create or look up a System V shared memory segment and return its identifier.
- **shmread** — Copy bytes out of a System V shared memory segment into a Perl scalar.
- **shmwrite** — Copy bytes into a System V shared memory segment.
- **shutdown** — Shut down one direction of a socket connection, or both.
- **sin** — Return the sine of a number given in radians.
- **sleep** — Pause the process for a number of whole seconds.
- **socket** — Create a socket filehandle.
- **socketpair** — Create an unnamed, connected pair of sockets that talk to each other.
- **sort** — Sort a list, returning the sorted list.
- **splice** — Remove and/or replace a slice of an array in place, and return the removed elements.
- **split** — Cut a string into a list of fields using a regex separator.
- **sprintf** — Build a formatted string from a format template and a list of values.
- **sqrt** — Return the non-negative square root of **EXPR**.
- **srand** — Seed the pseudo-random number generator.
- **stat** — Get a file's status information.
- **state** — Declare a lexically scoped variable whose value persists across calls to its enclosing subroutine.
- **study** — A no-op retained for source compatibility with older Perl code.
- **sub** — Declare or define a subroutine.

- **substr** — Extract, replace, or alias a contiguous slice of a string.
- **symlink** — Create a symbolic link at NEWFILE that points at the string OLDFILE.
- **syscall** — Invoke a raw system call by its kernel number, passing the remaining arguments as `int` or as a pointer to a string buffer.
- **sysopen** — Open a file the low-level way, passing an integer MODE bitmask straight through to the underlying `open(2)` system call.
- **sysread** — Read raw bytes from a filehandle by calling the underlying `read(2)` system call.
- **sysseek** — Reposition a filehandle at the system level, bypassing PerlIO buffering.
- **system** — Run a separate program and wait for it to finish.
- **syswrite** — Write bytes to a filehandle with the raw `write(2)` system call, bypassing Perl's buffered I/O.

T

- **tell** — Return the current byte position of a filehandle.
- **telldir** — Return the current read position of a directory handle as an opaque token.
- **tie** — Bind a variable to a class so that every access to the variable dispatches through that class's methods.
- **tied** — Return the object backing a tied variable.
- **time** — Return the current wall-clock time as an integer number of seconds since the system epoch.
- **times** — Report CPU time consumed by this process and its terminated children.
- **tr///** — Character-by-character substitution. `tr` scans a string and replaces every occurrence of a character from SEARCHLIST with the positionally corresponding character from REPLACEMENTLIST, returning the count of characters it touched.
- **truncate** — Shorten (or extend) a file to an exact byte length.
- **try** — Run a block and divert any exception it throws into a `catch` block, with an optional `finally` block that always runs on the way out.

U

- **uc** — Return an uppercased copy of a string.
- **ucfirst** — Return a copy of a string with its first character titlecased.
- **umask** — Set or read the process file-creation mode mask.
- **undef** — The undefined value, and the operator that produces it.
- **unlink** — Remove one or more directory entries that name files.
- **unpack** — Extract typed values from a binary or fixed-width string according to a template.

- **unshift** — Prepend one or more values to the start of an array and return the array's new length.
- **untie** — Break the binding between a variable and its tied class.
- **use** — Load a module at compile time and import its symbols into the current package.
- **utime** — Set access and modification times on a list of files.

V

- **values** — List every value of a hash (or every element of an array).
- **vec** — Read or write a fixed-width slot inside a string treated as a packed bit vector.

W

- **wait** — Block until any child process exits and reap it.
- **waitpid** — Wait for a specific child process to terminate and reap it.
- **wantarray** — Report the calling context of the currently executing subroutine.
- **warn** — Emit a warning to STDERR.
- **write** — Render one record to a filehandle through its associated **format**.

Y

- **y///** — Transliterate characters in a string. **y///** is a synonym for **tr///** - the two are identical in every respect.

__ (double-underscored)

- **__CLASS__** — Return the class name of the instance currently being acted on.
- **__FILE__** — The name of the source file the token is compiled in.
- **__LINE__** — A compile-time token that evaluates to the line number at which it appears in the source.
- **__PACKAGE__** — Return the name of the package currently in effect at the point where the token appears.
- **__SUB__** — Return a reference to the currently executing subroutine.

Built-in Functions by Category

The same built-ins as in *the alphabetical index*, grouped here by problem domain. A function may appear under more than one category: categories are a tag set, not a tree. Every link points at the same detail page regardless of which category the reader arrived through.

SCALARS and strings

- **chomp** — Strip the trailing input record separator from a string in place.
- **chop** — Remove the last character from a string and return it.
- **chr** — Return the character whose codepoint is the given number.
- **crypt** — One-way passwd(5)-style hashing of a plaintext string with a salt.
- **fc** — Return the Unicode casefolded form of a string for case-insensitive comparison.
- **hex** — Interpret a string as a hexadecimal numeral and return its numeric value.
- **index** — Find the position of a substring inside a string.
- **lc** — Return a lowercased copy of a string.
- **lcfirst** — Return a copy of a string with its first character lowercased.
- **length** — Return the number of characters in a string.
- **oct** — Interpret a string as a number written in octal, hexadecimal, or binary, and return the resulting integer.
- **ord** — Return the Unicode code point of the first character of a string.
- **pack** — Convert a list of Perl values into a binary string according to a template.
- **q//** — Single-quoted string literal with a choice of delimiter.
- **qq//** — Build a double-quoted, interpolated string with a delimiter you pick.
- **reverse** — Reverse a list - or, in scalar context, reverse the characters of a string.
- **rindex** — Find the position of the **last** occurrence of a substring inside a string.
- **sprintf** — Build a formatted string from a format template and a list of values.
- **substr** — Extract, replace, or alias a contiguous slice of a string.
- **tr///** — Character-by-character substitution. **tr** scans a string and replaces every occurrence of a character from SEARCHLIST with the positionally corresponding character from REPLACEMENTLIST, returning the count of characters it touched.
- **uc** — Return an uppercased copy of a string.
- **ucfirst** — Return a copy of a string with its first character titlecased.
- **y///** — Transliterate characters in a string. **y///** is a synonym for **tr///** - the two are identical in every respect.

Regular expressions and pattern matching

- **m//** — Search a string for a pattern and report whether - and what - it matched.
- **pos** — Report or set where the next /g regex match will resume in a string.
- **qr//** — Compile a pattern once and hand back a reusable regex object.
- **quotemeta** — Return a copy of a string with every regex-significant character backslash-escaped, so the result can be interpolated into a pattern and match its own literal content.

- **s///** — Search a string for a pattern and replace every match with a replacement.
- **split** — Cut a string into a list of fields using a regex separator.
- **study** — A no-op retained for source compatibility with older Perl code.

Numeric functions

- **abs** — Return the absolute value of a number.
- **atan2** — Arc tangent of Y/X in the range $-\pi$ to π .
- **cos** — Return the cosine of a number given in radians.
- **exp** — Raise e to a power.
- **hex** — Interpret a string as a hexadecimal numeral and return its numeric value.
- **int** — Return the integer portion of a number, truncating toward zero.
- **log** — Return the natural logarithm (base e) of a number.
- **oct** — Interpret a string as a number written in octal, hexadecimal, or binary, and return the resulting integer.
- **rand** — Return a pseudo-random floating-point number in the half-open range $[0, \text{EXPR})$.
- **sin** — Return the sine of a number given in radians.
- **sqrt** — Return the non-negative square root of EXPR .
- **srand** — Seed the pseudo-random number generator.

Arrays

- **each** — Step through a hash or array one entry at a time.
- **keys** — List all keys of a hash, or all indices of an array.
- **pop** — Remove and return the last element of an array.
- **push** — Append one or more values to the end of an array.
- **shift** — Remove and return the first element of an array.
- **splice** — Remove and/or replace a slice of an array in place, and return the removed elements.
- **unshift** — Prepend one or more values to the start of an array and return the array's new length.
- **values** — List every value of a hash (or every element of an array).

Lists

- **all** — Test whether **BLOCK** returns true for **every** element of **LIST**.
- **any** — Return true if **BLOCK** yields true for at least one element of **LIST**.
- **grep** — Filter a list to the elements where the block or expression is true.

- **join** — Concatenate a list of strings with a separator between each adjacent pair, and return the single resulting string.
- **map** — Apply a block or expression to each element of a list and return the flattened results.
- **qw//** — Build a list of barewords by splitting a delimited string on whitespace, without quoting or comma-separating each word.
- **reverse** — Reverse a list - or, in scalar context, reverse the characters of a string.
- **sort** — Sort a list, returning the sorted list.
- **unpack** — Extract typed values from a binary or fixed-width string according to a template.

Hashes

- **delete** — Remove the named key-value pair(s) from a hash, or element(s) from an array, and return what was removed.
- **each** — Step through a hash or array one entry at a time.
- **exists** — Test whether a hash or array element, or a named subroutine, is present - without creating it and without caring what it holds.
- **keys** — List all keys of a hash, or all indices of an array.
- **values** — List every value of a hash (or every element of an array).

I/O

- **binmode** — Set the I/O layer stack on a filehandle - typically to make it deliver raw bytes, or to attach a character encoding.
- **close** — Close a filehandle, flush its buffers, and release the underlying file descriptor.
- **closedir** — Close a directory handle opened by `opendir`.
- **dbmclose** — Break the binding between a DBM file and a hash previously attached by `dbmopen`.
- **dbmopen** — Bind a DBM file on disk to a hash so hash reads and writes become lookups and stores in the database.
- **die** — Raise an exception.
- **eof** — Test a filehandle for end-of-file.
- **fileno** — Return the OS-level file descriptor number behind a filehandle.
- **flock** — Place an advisory lock on an open file.
- **format** — Declare a picture-based report template for use by `write`.
- **getc** — Read the next single character from a filehandle.
- **print** — Write a list of values to a filehandle.
- **printf** — Write a formatted string to a filehandle.

- **read** — Read a fixed amount of buffered input from a filehandle into a scalar.
- **readdir** — Read the next entry, or all remaining entries, from a directory handle opened by **opendir**.
- **readline** — Read one or more records from a filehandle.
- **rewinddir** — Reset a directory handle to the start of its listing.
- **say** — Print a list of values followed by a newline.
- **seek** — Reposition a filehandle for random-access reads or writes.
- **seekdir** — Restore a directory handle to a position previously captured by **telldir**.
- **select** — Either set the default output filehandle, or call the `select(2)` syscall for I/O multiplexing. Same name, two unrelated jobs - disambiguated by argument count.
- **syscall** — Invoke a raw system call by its kernel number, passing the remaining arguments as `int` or as a pointer to a string buffer.
- **sysread** — Read raw bytes from a filehandle by calling the underlying `read(2)` system call.
- **sysseek** — Reposition a filehandle at the system level, bypassing PerlIO buffering.
- **syswrite** — Write bytes to a filehandle with the raw `write(2)` system call, bypassing Perl's buffered I/O.
- **tell** — Return the current byte position of a filehandle.
- **telldir** — Return the current read position of a directory handle as an opaque token.
- **truncate** — Shorten (or extend) a file to an exact byte length.
- **warn** — Emit a warning to STDERR.
- **write** — Render one record to a filehandle through its associated **format**.

Fixed-length data

- **pack** — Convert a list of Perl values into a binary string according to a template.
- **read** — Read a fixed amount of buffered input from a filehandle into a scalar.
- **syscall** — Invoke a raw system call by its kernel number, passing the remaining arguments as `int` or as a pointer to a string buffer.
- **sysread** — Read raw bytes from a filehandle by calling the underlying `read(2)` system call.
- **sysseek** — Reposition a filehandle at the system level, bypassing PerlIO buffering.
- **syswrite** — Write bytes to a filehandle with the raw `write(2)` system call, bypassing Perl's buffered I/O.
- **unpack** — Extract typed values from a binary or fixed-width string according to a template.
- **vec** — Read or write a fixed-width slot inside a string treated as a packed bit vector.

Filehandles, files, directories

- **chdir** — Change the process's current working directory.
- **chmod** — Change the permission bits of a list of files.
- **chown** — Change the owner and group of a list of files.
- **chroot** — Change the root directory of the current process and every child it later spawns.
- **fcntl** — Perform a `fcntl(2)` file-control operation on a filehandle.
- **glob** — Expand a shell-style filename pattern into the list of matching paths.
- **ioctl** — Perform a device-control `ioctl(2)` system call on a filehandle.
- **link** — Create a hard link from `NEWFILE` to the existing `OLDFILE`.
- **lstat** — Return the 13-element status list for a path **without** following a symbolic link.
- **mkdir** — Create a single directory on the filesystem.
- **open** — Associate a filehandle with a file, a command, or an in-memory scalar.
- **opendir** — Open a directory for reading.
- **readlink** — Return the target path a symbolic link points to.
- **rename** — Change the name of a file.
- **rmdir** — Remove an empty directory.
- **select** — Either set the default output filehandle, or call the `select(2)` syscall for I/O multiplexing. Same name, two unrelated jobs - disambiguated by argument count.
- **stat** — Get a file's status information.
- **symlink** — Create a symbolic link at `NEWFILE` that points at the string `OLDFILE`.
- **sysopen** — Open a file the low-level way, passing an integer `MODE` bitmask straight through to the underlying `open(2)` system call.
- **umask** — Set or read the process file-creation mode mask.
- **unlink** — Remove one or more directory entries that name files.
- **utime** — Set access and modification times on a list of files.

Control flow

- **__FILE__** — The name of the source file the token is compiled in.
- **__LINE__** — A compile-time token that evaluates to the line number at which it appears in the source.
- **__PACKAGE__** — Return the name of the package currently in effect at the point where the token appears.
- **__SUB__** — Return a reference to the currently executing subroutine.
- **break** — Break out of a given block.

- **caller** — Return information about the subroutine, `eval`, or `require` that called the currently executing code.
- **catch** — Handle an exception thrown by a preceding `try` block.
- **continue** — Attach a block to a loop that runs after every iteration, just before the condition is re-tested.
- **defer** — Schedule a block to run when the enclosing scope exits, for any reason.
- **die** — Raise an exception.
- **do** — Execute a block of code or run a Perl source file as if it were part of the current program.
- **dump** — Cause the running program to produce a core dump immediately.
- **eval** — Run a piece of Perl code with any fatal error trapped instead of killing the program.
- **evalbytes** — Compile and run a string of Perl source, forcing the source to be interpreted as `bytes` rather than characters.
- **exit** — Terminate the program with a status value.
- **finally** — Run cleanup code on the way out of a `try/catch`, whether the body succeeded, threw, or jumped away.
- **goto** — Transfer execution elsewhere in the program without returning.
- **last** — Exit a loop immediately, skipping the rest of the body and the `continue` block.
- **method** — Declare a named instance method inside a `class` block.
- **next** — Start the next iteration of the enclosing loop immediately.
- **redo** — Restart the current iteration of a loop without re-testing the condition and without running the `continue` block.
- **return** — Leave the current subroutine, `eval`, `do FILE`, `sort` block, or regex `eval` block, yielding a value to the caller.
- **sub** — Declare or define a subroutine.
- **try** — Run a block and divert any exception it throws into a `catch` block, with an optional `finally` block that always runs on the way out.
- **wantarray** — Report the calling context of the currently executing subroutine.

Scoping

- **caller** — Return information about the subroutine, `eval`, or `require` that called the currently executing code.
- **class** — Declare a namespace that behaves as a native object class.
- **field** — Declare a per-instance variable inside a `class` block.
- **import** — Populate the caller's namespace with names a module chooses to export.
- **local** — Save the current value of a package variable and restore it when the enclosing scope exits.

- **my** — Declare one or more lexically scoped variables.
- **our** — Declare a lexically scoped alias to a package variable.
- **package** — Declare the compile-time namespace for the declarations that follow.
- **state** — Declare a lexically scoped variable whose value persists across calls to its enclosing subroutine.
- **use** — Load a module at compile time and import its symbols into the current package.

Misc

- **defined** — Test whether a value, variable, or subroutine is defined.
- **formline** — Format a list of values into a picture string and append the result to the format accumulator `$^A`.
- **lock** — Place an advisory lock on a shared variable, array, hash, or subroutine until the lock goes out of scope.
- **prototype** — Return the prototype string of a subroutine, or **undef** if it has none.
- **reset** — Clear every package variable whose name begins with one of a set of letters, and re-arm one-shot `m?pattern?` matches.
- **scalar** — Force `EXPR` to be evaluated in scalar context and return its value.
- **undef** — The undefined value, and the operator that produces it.

Processes

- **alarm** — Schedule a `SIGALRM` to be delivered to the current process after a whole number of wallclock seconds.
- **exec** — Abandon this program and run another in the same process.
- **fork** — Create a new process running the same program at the same point.
- **getpgrp** — Return the POSIX process-group ID that a process belongs to.
- **getppid** — Return the process ID of the current process's parent.
- **getpriority** — Return the current scheduling nice value of a process, a process group, or a user.
- **kill** — Send a signal to a list of processes.
- **pipe** — Open a pair of connected filehandles - one for reading, one for writing.
- **qx//** — Run a shell command and capture its standard output.
- **readpipe** — Run a shell command and return its standard output.
- **setpgrp** — Set the process group of a process.
- **setpriority** — Set the scheduling priority (nice value) of a process, process group, or user.
- **sleep** — Pause the process for a number of whole seconds.

- **system** — Run a separate program and wait for it to finish.
- **times** — Report CPU time consumed by this process and its terminated children.
- **wait** — Block until any child process exits and reap it.
- **waitpid** — Wait for a specific child process to terminate and reap it.

Modules

- **do** — Execute a block of code or run a Perl source file as if it were part of the current program.
- **import** — Populate the caller's namespace with names a module chooses to export.
- **no** — The compile-time inverse of **use** - call a module's `unimport` method to turn off what `use` turned on.
- **package** — Declare the compile-time namespace for the declarations that follow.
- **require** — Load a Perl source file at runtime, or demand a minimum Perl version.
- **use** — Load a module at compile time and import its symbols into the current package.

Classes and OO

- **__CLASS__** — Return the class name of the instance currently being acted on.
- **bless** — Mark the thing a reference points at as an object in a package.
- **class** — Declare a namespace that behaves as a native object class.
- **dbmclose** — Break the binding between a DBM file and a hash previously attached by `dbmopen`.
- **dbmopen** — Bind a DBM file on disk to a hash so hash reads and writes become lookups and stores in the database.
- **field** — Declare a per-instance variable inside a `class` block.
- **isa** — Test whether an object is an instance of a class or of any subclass derived from it.
- **method** — Declare a named instance method inside a `class` block.
- **package** — Declare the compile-time namespace for the declarations that follow.
- **ref** — Return a string describing what a reference points to.
- **tie** — Bind a variable to a class so that every access to the variable dispatches through that class's methods.
- **tied** — Return the object backing a tied variable.
- **untie** — Break the binding between a variable and its tied class.
- **use** — Load a module at compile time and import its symbols into the current package.

Sockets

- **accept** — Accept an incoming connection on a listening socket.
- **bind** — Attach a local address to a socket.
- **connect** — Initiate a connection from a socket to a remote address.
- **getpeername** — Return the address of the remote end of a connected socket.
- **getsockname** — Return the local address of a connected or bound socket.
- **getsockopt** — Read one socket option out of the kernel as an opaque packed string.
- **listen** — Mark a socket as passive so it can accept incoming connections.
- **recv** — Read an incoming message from a socket into a scalar.
- **send** — Send a message on a socket.
- **setsockopt** — Set a kernel-level option on an open socket.
- **shutdown** — Shut down one direction of a socket connection, or both.
- **socket** — Create a socket filehandle.
- **socketpair** — Create an unnamed, connected pair of sockets that talk to each other.

SysV IPC

- **msgctl** — Perform a control operation on a System V IPC message queue.
- **msgget** — Create or look up a System V IPC message queue and return its id.
- **msgrcv** — Receive a message from a System V IPC message queue.
- **msgsnd** — Send a message to a System V IPC message queue.
- **semctl** — Perform a control operation on a System V semaphore set.
- **semget** — Create or look up a System V semaphore set and return its identifier.
- **semop** — Perform one or more System V semaphore operations atomically.
- **shmctl** — Control or query a System V shared memory segment.
- **shmget** — Create or look up a System V shared memory segment and return its identifier.
- **shmread** — Copy bytes out of a System V shared memory segment into a Perl scalar.
- **shmwrite** — Copy bytes into a System V shared memory segment.

User and group info

- **endgrent** — Close the group database after iterating it.
- **endpwent** — Close the passwd-database iterator opened by **getpwent** or **setpwent**.
- **getgrent** — Read the next entry from the system group database.
- **getgrgid** — Look up a group record by numeric group ID.

- **getgrnam** — Look up a Unix group by name and return its `/etc/group` record.
- **getlogin** — Return the login name of the user associated with the controlling terminal.
- **getpwent** — Return the next entry from the system password database.
- **getpwnam** — Look up a user's passwd record by login name.
- **getpwuid** — Look up a user's passwd record by numeric UID.
- **setgrent** — Rewind the group-database iterator back to the first entry.
- **setpwent** — Rewind the password-database iterator so the next **getpwent** call returns the first entry again.

Network info

- **endhostent** — Close the hosts database after iteration.
- **endnetent** — Close the networks database after iteration.
- **endprotoent** — Close the protocols database after iteration.
- **endservent** — Close the services database after a walk with **getservent**.
- **gethostbyaddr** — Look up a host record by its packed IP address.
- **gethostbyname** — Look up a host record by DNS name.
- **gethostent** — Fetch the next entry from the host database.
- **getnetbyaddr** — Look up a network record by its numeric network address.
- **getnetbyname** — Look up a network record by name.
- **getnetent** — Read the next entry from the networks database.
- **getprotobyname** — Look up an IP protocol by its textual name and return its database entry.
- **getprotobynumber** — Look up a network protocol entry by its assigned protocol number.
- **getprotoent** — Fetch the next entry from the protocols database.
- **getservbyname** — Look up a network service by its textual name and protocol.
- **getservbyport** — Look up a network service by its numeric port and protocol.
- **getservent** — Read the next entry from the system services database.
- **sethostent** — Open or rewind the hosts database for iteration.
- **setnetent** — Open or rewind the networks database for iteration.
- **setprotoent** — Open the protocols database and prepare it for sequential reads.
- **setservent** — Rewind the services database and optionally keep it open across lookups.

Time

- **gmtime** — Convert an epoch time to a broken-down UTC time, either as a 9-element list or as a `ctime(3)`-style string.
- **localtime** — Convert an epoch time to a broken-down calendar time in the local time zone.
- **time** — Return the current wall-clock time as an integer number of seconds since the system epoch.
- **times** — Report CPU time consumed by this process and its terminated children.

Classes and OO

CLASS

Return the class name of the instance currently being acted on.

`__CLASS__` is a compile-time-recognised token that evaluates at runtime to the class name of the invoking instance. Inside a **method** body or a field initializer expression (introduced by `use feature 'class'`), it gives the **actual** class of the object under construction or in use - which may be a subclass of the package the code textually lives in. Outside of class-feature code it is essentially equivalent to `ref($self)`, but unlike `ref` it is usable in a field initializer, where `$self` is not yet bound.

Synopsis

```
use feature 'class';

field $x = __CLASS__->DEFAULT_X;      # in a field initializer
method m { return __CLASS__; }      # in a method body
```

What you get back

A plain string: the name of the class of the invoking instance. The value is the same kind of string `ref` returns for a blessed reference. It is **not** a reference; call methods on it as you would on any class name.

```
my $name = __CLASS__;                # e.g. "DifferentCustomField"
my $obj  = __CLASS__->new(...);      # class-method dispatch
```

When it differs from `__PACKAGE__`

`__PACKAGE__` is resolved at compile time and names the package the token textually appears in. `__CLASS__` is resolved at runtime and names the class of the *instance* - which, for a method or field initializer inherited by a subclass, is the subclass.

```
use feature 'class';

class Base {
```

(continues on next page)

(continued from previous page)

```

    field $f = __CLASS__->default_f;      # evaluated per instance
    sub default_f { 10 }
    method class_name { __CLASS__ }
    method pkg_name   { __PACKAGE__ }
}

class Derived :isa(Base) {
    sub default_f { 20 }
}

my $obj = Derived->new;
$obj->class_name;           # "Derived"
$obj->pkg_name;             # "Base"
# $f was initialised to 20, not 10, because __CLASS__ dispatched
# through the Derived package.

```

In a non-inherited context (no subclassing involved), `__CLASS__` and `__PACKAGE__` yield the same string.

Examples

Class method invoked from a field initializer - the canonical use case. Without `__CLASS__`, a subclass cannot override the default:

```

use feature 'class';

class WithCustomField {
    use constant DEFAULT_X => 10;
    field $x = __CLASS__->DEFAULT_X;
}

class DifferentCustomField :isa(WithCustomField) {
    sub DEFAULT_X { rand > 0.5 ? 20 : 30 }
}

my $obj = DifferentCustomField->new;    # $x is 20 or 30, not 10

```

Inside a method body, equivalent to `ref $self`:

```

class Logger {
    method log ($msg) {
        print STDERR "[", __CLASS__, "] $msg\n";
    }
}

```

Constructing a fresh instance of the same class as the invoking object - useful for clone-like factory methods that must respect subclasses:

```

class Node {
    field $label :param;
}

```

(continues on next page)

(continued from previous page)

```

method spawn ($child_label) {
    return __CLASS__->new(label => $child_label);
}

```

Comparison with `__PACKAGE__` for diagnostics that want to pin the exact dispatch target:

```

class Thing {
    method describe {
        return sprintf "defined in %s, dispatched as %s",
            __PACKAGE__, __CLASS__;
    }
}

```

Edge cases

- **Only valid under use feature 'class'** inside a method body, a field initializer, or another class-feature construct (for example ADJUST blocks). Using it outside such a context is a compile-time error.
- **No parentheses, no arguments.** `__CLASS__` is a bareword token, not a function call. `__CLASS__( )` is a syntax error.
- **Not the same as `ref $self`** during field initialization - `$self` is not yet bound while fields are being set, so `ref $self` is unavailable. `__CLASS__` is the only way to reach the instance's class from inside a field initializer.
- **Not `ref` for a class method call.** When called as a class method (no instance in hand), `__CLASS__` yields the class the method was invoked on, not the package the method was defined in. This mirrors normal method dispatch.
- **String, not reference.** Method calls via `__CLASS__->...` are class-method dispatches, not instance-method dispatches. The method will not see a blessed reference as its invocant.
- **The `class` feature is marked experimental** in Perl 5.42; it emits a `experimental::class` warning unless silenced with `no warnings 'experimental::class'`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `__PACKAGE__` - compile-time package name of the surrounding code; use this when you want the textual package, not the runtime class
- `ref` - runtime class name of a blessed reference; use this when you already hold an instance and are not inside a field initializer

- `bless` - associate a reference with a class name; the low-level primitive that `__CLASS__`-style dispatch ultimately resolves against
- `isa` - test whether the current class is a subclass of a given name; pairs naturally with `__CLASS__` for runtime checks
- `sub` - declares subroutines, including the method form that makes `__CLASS__` meaningful

Control flow

FILE

The name of the source file the token is compiled in.

`__FILE__` is a compile-time token, not a runtime function. At the point the compiler reads it, it is replaced with a string literal holding the filename the compiler is currently reading from. It takes no arguments, is never called with parens, and has no runtime cost - by the time your program runs, `__FILE__` has already become a plain string constant.

Its one job is to let code identify the file it lives in, for error messages, log lines, test diagnostics, and similar reporting.

Synopsis

`__FILE__`

What you get back

A string: the filename as the compiler knew it when it read this token. For a script invoked as `perl /tmp/run.pl` the value is `/tmp/run.pl`; for a module loaded through `require` or `use` it is the path `require` resolved against `@INC` (typically an absolute path). For code fed on standard input or through `-e` it is `-` or `-e` respectively.

Because substitution happens at compile time, the value is fixed for each textual occurrence of the token. Two `__FILE__` tokens in different files produce different strings; two in the same file produce the same string.

Global state it touches

None at runtime. At compile time the value is taken from the compiler's current filename, which is the same value `caller` reports in list context as element `[1]` and which `die` / `warn` use to build their default `"at FILE line LINE."` suffix.

That filename can be **rewritten** by a `#line` directive embedded in the source - see *Edge cases* below. Once rewritten, `__FILE__` tokens compiled after the directive report the new name.

Examples

Identify the running script in a log line:

```
warn "started from ", __FILE__, "\n";
# started from /tmp/run.pl
```

Attach source location to a custom error without `die`'s default suffix:

```
sub fail {
    my ($msg) = @_;
    die sprintf("%s at %s line %d\n", $msg, __FILE__, __LINE__);
}
```

Build a path relative to the current module, so resource files ship alongside the code that loads them:

```
use File::Basename qw(dirname);
use File::Spec;
my $data = File::Spec->catfile(dirname(__FILE__), "data.txt");
```

Use `__FILE__` inside a string passed to `eval` to keep error reports pointing at the original source rather than at the anonymous (`eval N`) buffer:

```
my $code = qq[\n#line ] . __LINE__ . qq[ ] . __FILE__ . qq[\n]
    . qq[die "boom"];
eval $code;
# boom at <current file> line <current line>.
```

Distinguish “run as a script” from “loaded as a module” - the common test is whether the file was invoked directly:

```
__PACKAGE__->run(@ARGV) if __FILE__ eq $0;
```

Edge cases

- **Compile-time, not runtime.** `__FILE__` is a token the parser replaces with a string literal. It is not a function you can take a reference to; `\&__FILE__` is a syntax error. Stringify it, store it, or pass it - but you cannot defer its evaluation.
- **Stable per occurrence.** Calling the same subroutine from two different files does not change the `__FILE__` the subroutine body sees; that value was baked in when the subroutine was compiled. If you need the caller's filename, use `caller`.
- **#line directives rewrite the value.** A comment of the form `# line N "NAME"` at column zero resets both the line number and (optionally) the filename the compiler reports. Every `__FILE__` token compiled after such a directive picks up the new name. This is the mechanism code generators and template engines use to make errors point at the original `.tt` / `.pod` / `.xs` source instead of the generated `.pl`.

```
# line 1 "config.tt"
my $x = __FILE__;      # "config.tt"
```

Full syntax of the directive is specified in `perlsyn` under *Plain Old Comments (Not!)*.

- **Modules.** Inside a module, `__FILE__` is the path `require` used to load the file - usually absolute and under an `@INC` root. Do not parse it as `Some/Module.pm`; use `dirname(__FILE__)` if you need the directory, not string surgery.
- **Special command-line sources.** Code from `-e` reports `-e`; code from `-E` reports `-e` as well; code piped on stdin reports `-`. String `eval` without a `#line` directive reports `(eval N)` where `N` is the eval counter.
- **One-argument `open` and its `__FILE__` lookalike.** There is no `__FILE__` handle; the token has nothing to do with filehandles. `open(__FILE__)` does not open the current source file - it tries to open a file literally named after the token's expanded string, which is rarely what you want.
- **Not modifiable at runtime.** Assigning to `__FILE__` is a syntax error. The mechanism for changing what `__FILE__` reports is the `#line` directive at compile time; there is no runtime equivalent.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `__LINE__` - companion compile-time token for the current line number; the two are almost always used together
- `__PACKAGE__` - compile-time token for the current package name, same family of special tokens
- `caller` - runtime lookup of the caller's filename, line, and package; use this when you need the caller's source location rather than your own
- `die` - appends at `FILE line LINE`. automatically; reach for `__FILE__` explicitly only when you suppress that suffix with a trailing newline
- `perlsyn` - the *Plain Old Comments (Not!)* section specifies the `#line` directive that rewrites `__FILE__`

Control flow

LINE

A compile-time token that evaluates to the line number at which it appears in the source.

`__LINE__` is not a function. It is a special token the parser replaces, at compile time, with an integer literal - the line number of the file position where the token sits. It is the standard way to stamp a diagnostic, a log line, or a debug print with the exact location that produced it, without paying for a `caller` lookup at runtime.

Synopsis

```
__LINE__
```

What you get back

An integer: the 1-based line number of the token within the current compilation unit. The value is a plain number, usable anywhere a number is - in arithmetic, in string interpolation, as a hash key, as an argument to `die` or `warn`.

```
print __LINE__, "\n";      # prints the line number of this line
my $where = "line " . __LINE__;
```

Because the substitution happens at parse time, every occurrence of `__LINE__` is effectively a different constant. Two copies on two different lines produce two different numbers; a single `__LINE__` inside a loop produces the same number on every iteration.

Compile-time, not run-time

The value is frozen when the source is compiled. A common confusion:

```
sub where { return __LINE__ }      # returns the line where `__LINE__`
↪                                     # was written, i.e. the `sub` body
↪ 's                                 # line - NOT the caller's line
```

To capture the *caller's* line number, use `caller`:

```
sub where { my (undef, undef, $line) = caller; return $line }
```

Inside `eval` of a string, `__LINE__` numbers the lines of the string being compiled (starting at 1 for the first line of the string), not the lines of the surrounding file.

Altering the reported line number

The line number `__LINE__` reports is the line number the lexer believes it is on. A `#line` directive resets that belief:

```
#line 1000 "virtual.pl"
print __LINE__, "\n";      # prints 1001
```

`#line NNN` on its own sets only the line number; `#line NNN "NAME"` also changes what `__FILE__` reports. Code generators, template engines, and preprocessors use this to make diagnostics point back at the original source rather than the generated file. See *Plain Old Comments (Not!)* in the `perlsyn` section of the reference for the precise syntax and scope rules.

Examples

Stamp a diagnostic with its origin:

```
die "bad input at ", __FILE__, " line ", __LINE__, "\n"
    unless defined $x;
```

die with a message that does **not** end in a newline already appends " at FILE line LINE.\n". Use the explicit form above only when you want to control the formatting, or when you've suppressed the automatic suffix with a trailing newline.

Log with file-and-line prefix:

```
sub logf {
    my ($file, $line) = (__FILE__, __LINE__);
    warn "[$file:$line] @_ \n";
}
```

This logs the line where `__LINE__` was *written* - inside `logf` - which is rarely what you want. The correct pattern uses `caller`:

```
sub logf {
    my (undef, $file, $line) = caller;
    warn "[$file:$line] @_ \n";
}
```

Use `__LINE__` directly when you are marking *this* piece of code, not the caller:

```
print STDERR "checkpoint at line ", __LINE__, "\n";
```

Relocate diagnostics with `#line`:

```
#line 42 "user-input.pl"
die "syntax error\n";           # dies reporting user-input.pl line 42
```

Edge cases

- **Not a function, not callable.** `__LINE__()` is a syntax error. The token takes no arguments and has no parentheses form.
- **Not interpolated inside double-quoted strings.** "see line `__LINE__`" prints the literal string see line `__LINE__`. Concatenate instead: "see line " . `__LINE__`.
- **Not a valid hash-key bareword in the usual short form.** Inside `{ ... }` hash-key context, `$h{__LINE__}` works because the token is parsed as an expression, but `$h{ __LINE__ }` also works and is clearer. `$h{__LINE__ => 1}` parses `__LINE__` as a fat-comma bareword (i.e. the string "`__LINE__`"), which is almost never what you want - write `(__LINE__, 1)` or `(__LINE__() => 1)` if the literal value is wanted.
- **Inside string eval,** `__LINE__` counts from line 1 of the evaluated string. Prepend your own `#line` directive to the string to make diagnostics point at the enclosing file:

```
my $code = qq{#line @(__LINE__+1)} "@(__FILE__)"\n} . $user_
  ↪code;
eval $code;
```

- **Inside a here-doc body**, `__LINE__` evaluates to the line of the *opening* `<<` on the line where the here-doc is referenced, not lines inside the body - here-doc bodies are data, not code.
- **Heredoc and line directives**. `#line` inside a here-doc body has no effect; it is just text. `#line` only works as a beginning-of-line directive in code context.
- **Negative or zero line numbers**. `#line 0` is legal and makes the next physical line report as line 1 (the directive itself is line 0, the line after is line 1). Negative numbers are accepted but produce surprising diagnostics; don't.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `__FILE__` - the companion token for the current source file name; the two are almost always used together
- `__PACKAGE__` - compile-time token for the current package name, same family of special tokens
- `caller` - runtime equivalent when you need the *caller's* line number, not the line of the call-site itself
- `die` - auto-appends `at FILE line LINE` when the message has no trailing new-line, removing the need for explicit `__LINE__`
- `warn` - same auto-location behavior as `die` for non-fatal diagnostics

Control flow

PACKAGE

Return the name of the package currently in effect at the point where the token appears.

`__PACKAGE__` is a compile-time literal: the parser replaces it with a string constant naming whichever package the most recent `package` statement put into effect for this lexical scope. It does not look anything up at runtime, does not follow inheritance, and does not care what class the calling code is using - it is simply the name of the package that owns the source text it sits in.

Synopsis

```
__PACKAGE__
__PACKAGE__->some_method(@args)
$__PACKAGE__::VERSION           # does NOT work - see Edge cases
```

What you get back

A plain string containing the fully qualified package name (e.g. "My::Thing", "main"), interned at compile time. The value is a constant for the duration of the enclosing scope - it cannot change at runtime and does not depend on how the code is called.

Outside any explicit `package` declaration, the value is "main".

Global state it touches

None. `__PACKAGE__` is resolved at compile time from the lexical package in effect at its source location; it reads no interpreter state when executed and has no side effects.

Why use a token instead of writing the name

Three reasons, in the order you hit them in practice:

1. **The file gets renamed or the package gets renamed and nothing else has to change.** Every `__PACKAGE__` updates automatically; every hand-typed "My::Thing" has to be found and edited.
2. **It is the only portable way to write “this class” inside a package block form,** where the package ends at the closing brace and an inner helper cannot assume the file name matches.
3. **It composes with class methods cleanly:** `__PACKAGE__->new(...)` in a constructor helper calls the right class whether the file is loaded directly or copied into a subclass that has not yet redefined the helper.

For the runtime equivalent - “what class is `$self` actually blessed into?” - use `ref` or, in use feature 'class' code, `__CLASS__`. They are not the same thing; see *Edge cases*.

Examples

Read the current package inside a module:

```
package My::Thing;
say __PACKAGE__;           # My::Thing
```

Use `__PACKAGE__` to reach a package variable without hard-coding the name - robust against renames:

```
package My::Thing;
our $VERSION = '1.23';

sub version_string {
    no strict 'refs';
    return ${ __PACKAGE__ . '::VERSION' };
}
```

Constructor helper that stays correct when copy-pasted into a subclass or when the package is switched with the block form:

```
package My::Widget {
    sub new {
        my ($class, %args) = @_;
        bless { %args, kind => __PACKAGE__ }, $class;
    }
}
# kind is always "My::Widget", even if someone calls
# My::Widget::SubClass->new and forgets to override `new`.
```

Debug trace that prints where a log line came from, without having to duplicate the package name in every message:

```
package My::Worker;
use v5.36;

sub log_debug {
    my ($msg) = @_;
    warn __PACKAGE__ . ": $msg\n";
}
```

Cross-package resolution with the block form - each `__PACKAGE__` sees whichever package wraps it:

```
package Outer;
say __PACKAGE__;           # Outer

package Inner {
    say __PACKAGE__;       # Inner
}

say __PACKAGE__;           # Outer again
```

Edge cases

- **Constant at compile time, not a variable.** `__PACKAGE__` is not a scalar; `$__PACKAGE__` and `${__PACKAGE__}` do not exist. It interpolates in double-quoted strings as the string it resolves to:

```
package Foo;
say "in @{$[__PACKAGE__]}";    # in Foo
say "in $__PACKAGE__";        # prints "in " - $__PACKAGE__ is
↪undef
```

The `@{[ ... ]}` idiom is the shortest way to interpolate it.

- **Symbolic reference to a package variable** requires string concatenation, not interpolation of a sigil:

```
no strict 'refs';
my $v = ${ __PACKAGE__ . '::-VERSION' }; # works
```

(continues on next page)

(continued from previous page)

```
my $v = ${"__PACKAGE__::VERSION"};      # WRONG - literal_
↪package
```

- **Not the same as `ref $self`.** `__PACKAGE__` is where the source was written; `ref $self` is what the object was actually blessed into. A method inherited from a base class returns the base class from `__PACKAGE__` but the subclass from `ref $self`:

```
package Base;
sub who { (__PACKAGE__, ref $_[0]) }

package Child;
our @ISA = ('Base');

say join " / ", Child->who;      # Base / Child
```

When you want the class the method was invoked on, use `ref` on the invocant, `__CLASS__` inside a `class/method` block, or take `$_[0]` / `shift` as the class name for a class method.

- **Block-form scoping.** Inside `package NAME { ... }`, `__PACKAGE__` is `NAME`; after the closing brace the previous value is restored. See the nested example above.
- **`eval` `STRING` captures its compile-time package.** The string is compiled in the caller's package, so `__PACKAGE__` inside `eval "..."` is the package that called `eval`:

```
package Foo;
eval 'say __PACKAGE__';          # Foo
```

- **Before any package statement** (top of a script, top of a one-liner), `__PACKAGE__` is `"main"`.
- **`__PACKAGE__` does not mean “current class”.** Underuse feature `'class'`, the class name and the package name agree on the declaration line but can differ at a call site invoked on a subclass. Inside a `method`, prefer `__CLASS__` when you want the class of the receiver.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `package` - the declaration that sets what `__PACKAGE__` returns; every non-main value comes from one
- `__CLASS__` - the runtime counterpart for code inside a `class` block: returns the class of the invoking instance, which may be a subclass
- `ref` - ask an object what class it was blessed into; the right tool when you have `$self` and want its actual class

- `caller` - the calling package at runtime; use when you need the caller's package, not your own
- `bless` - the primitive that turns a reference into an object of a given package; typically paired with `__PACKAGE__->new( . . . )` style constructors
- `our` - declares a lexical alias to a package variable in the current compile-time package (the same package `__PACKAGE__` names)

Control flow

SUB

Return a reference to the currently executing subroutine.

`__SUB__` is the self-reference token: inside a sub, it evaluates to a code reference pointing at that same sub, without the sub having to know its own name. Outside any sub, it is `undef`. The common reason to reach for it is anonymous recursion - a sub stored in a lexical or passed around as a callback needs a way to call itself that does not depend on a package-qualified name, and `__SUB__` is it.

Synopsis

```
__SUB__
__SUB__->(@args)
goto __SUB__
```

Takes no arguments and has no parentheses form - it is a compile-time token, not a function call.

What you get back

A code reference (the same thing `\&name` would give you for a named sub), or `undef` when evaluated outside any subroutine. The reference is to the **runtime** sub, including any closure over lexicals captured when the sub was created, so calling it recursively sees the same captured environment.

```
my $fact = sub {
    my $n = shift;
    $n < 2 ? 1 : $n * __SUB__->($n - 1);
};

print $fact->(5);           # 120
```

The call form is an ordinary coderef invocation: `__SUB__->(@args)` or `&{__SUB__}(@args)`. No arguments are passed implicitly; recursion must pass them on.

Why not just use the sub's name

Named subs can already recurse by name - `factorial(...)` inside `sub factorial` works. `__SUB__` earns its place in three situations where a name is unavailable, unreliable, or expensive:

- **Anonymous subs.** An anonymous sub assigned to a lexical has no stable package-qualified name. A common pre-5.16 workaround was capturing a reference to `\&name` or using Y-combinator tricks; `__SUB__` replaces both.
- **Subs passed as callbacks.** A sub handed to another API (sort comparator, event handler, iterator step) may be called back before it has a name visible at the recursion site.
- **Tail-call recursion via `goto`.** `goto __SUB__` reuses the current call frame and jumps to the top of the same sub with whatever `@_` you set up, which matters for deep recursion that would otherwise blow the stack.

Global state it touches

`__SUB__` reads the currently executing call frame - the same frame `caller` inspects. It does not read or modify any special variables.

Examples

Anonymous factorial, the textbook case:

```
my $fact = sub {
    my $n = shift;
    $n < 2 ? 1 : $n * __SUB__->($n - 1);
};
print $fact->(6);           # 720
```

Tail-recursive sum with `goto __SUB__` - no stack growth per iteration, because `goto` reuses the current frame:

```
my $sum = sub {
    my ($acc, @rest) = @_;
    return $acc unless @rest;
    @_ = ($acc + shift @rest, @rest);
    goto __SUB__;
};
print $sum->(0, 1..1_000_000); # 5000000500000
```

Walking a nested data structure without naming the walker:

```
my $count_leaves = sub {
    my $node = shift;
    return 1 unless ref $node eq 'ARRAY';
    my $n = 0;
    $n += __SUB__->($_) for @$node;
    return $n;
};
```

(continues on next page)

(continued from previous page)

```
};
print $count_leaves->([1, [2, 3], [[4], 5]]);    # 5
```

Outside any sub, `__SUB__` is `undef`:

```
print defined(__SUB__) ? "yes" : "no";           # no
```

Captured, not resolved-on-call. `__SUB__` is evaluated each time it runs, in the frame it runs in - so in a nested sub it refers to the **inner** sub:

```
sub outer {
    my $inner = sub { __SUB__ };    # inner's own coderef
    return $inner->();
}
my $ref = outer();
print $ref == $ref;                # 1 - stable coderef
```

Edge cases

- **Requires the `current_sub` feature.** `__SUB__` is gated behind use feature `'current_sub'`, enabled by default under use `v5.16` or any later version bundle. Code targeting older syntax levels must opt in explicitly, or prefix the token with `CORE::` as `CORE::__SUB__`.
- **No parentheses form.** `__SUB__()` is a syntax error - the token is not a function, it's a keyword that evaluates to a coderef. Call the result with `__SUB__->()` or `&{__SUB__}()`.
- **Outside any sub**, including at file scope and inside `BEGIN`, `UNITCHECK`, `CHECK`, `INIT`, and `END` blocks, `__SUB__` returns `undef`. Calling it (`__SUB__->()`) in those positions dies with `Can't use an undefined value as a subroutine reference`.
- **Inside `eval` blocks**, `__SUB__` refers to the **enclosing sub**, not the `eval` - an `eval { ... }` does not create a new sub frame for this purpose. Inside `eval EXPR` the same rule applies: the token resolves to the outer sub.
- **Inside a regex code block** (`/(?{...})/` or `/({...})/`), the behaviour of `__SUB__` is subject to change - upstream documents this as unstable. Do not rely on what it returns there.
- **Identity across calls.** Two evaluations of `__SUB__` inside the same sub yield the same coderef (`==` compares true). Across different closures made from the same sub `{ ... }`, each closure sees its own coderef - one per captured environment.
- **`goto __SUB__` vs `__SUB__->()`.** The `goto` form reuses the current call frame and replaces `@_` with whatever you assigned before the `goto`; it does not grow the call stack. The coderef call form makes a fresh frame every time, so it hits the recursion depth limit on deeply recursive inputs.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `sub` - defines the subroutine whose reference `__SUB__` hands back; the first place to look when anonymous recursion is the tool for the job
- `goto` - pairs with `__SUB__` for tail-call recursion via `goto __SUB__`, the idiom that avoids stack growth
- `caller` - inspect the same call frame from the outside (package, file, line) instead of grabbing a reference to it
- `ref` - classify what `__SUB__` returned; always "CODE" inside a sub, empty string outside
- `return` - the usual way to exit the sub whose reference `__SUB__` is; relevant when choosing between recursion and tail-call via `goto __SUB__`

Numeric functions

abs

Return the absolute value of a number.

`abs` takes a single scalar, forces it to numeric context, and returns its magnitude - the value with any sign stripped. Non-numeric strings are converted using the same rules as arithmetic (leading whitespace and an optional sign, followed by digits; the rest is discarded, with a warning under `use warnings`). If `VALUE` is omitted, `abs` operates on `$_`.

Synopsis

```
abs VALUE
abs
abs($x)
```

What you get back

A non-negative number. The result is an integer if the input was an integer within the platform's integer range, otherwise a floating-point number. `abs` never changes its argument - the result is a fresh scalar, so `abs $x` leaves `$x` alone.

Global state it touches

With no argument, `abs` reads `$_`. It neither writes nor reads any other special variable. Under `use warnings`, a non-numeric argument triggers the standard Argument "... isn't numeric" warning via the normal numeric-conversion path.

Examples

Strip the sign from an integer:

```
print abs(-7);           # 7
print abs(7);            # 7
```

Floating-point magnitude - the result keeps the full precision of the input:

```
print abs(-3.14159);     # 3.14159
print abs(-1e-10);       # 1e-10
```

Use the default argument form inside a loop over `$_`:

```
for (-3, -1, 0, 2, 4) {
    print abs, "\n";      # 3, 1, 0, 2, 4 on separate lines
}
```

Distance between two points on a number line:

```
sub distance { abs($_[0] - $_[1]) }
print distance(10, 3);    # 7
print distance(3, 10);    # 7
```

String that looks like a negative number - `abs` parses the leading numeric prefix and strips the sign:

```
print abs("-42");         # 42
print abs(" -3.5xyz");    # 3.5 (warning under use warnings)
```

Edge cases

- **`abs(undef)`** returns 0. Under `use warnings` it emits `Use of uninitialized value in abs.`
- **Non-numeric string:** `abs("hello")` returns 0 and warns under `use warnings` that the argument isn't numeric.
- **Integer overflow boundary:** `abs` of the most-negative platform integer (PERL_INT_MIN, e.g. -2^{63} on 64-bit builds) promotes to a floating-point result, because the positive counterpart does not fit in a signed integer:

```
use Config;
my $min = -(2 ** ($Config{ivsize} * 8 - 1));
print abs($min);           # 9.22337203685478e+18 on 64-bit
```

- **IEEE 754 special values:** `abs(-0.0)` returns 0 (a positive zero). `abs("-Inf")` returns Inf. `abs("NaN")` returns NaN - NaN has no sign that `abs` can meaningfully remove.
- **Complex numbers and big integers** are not handled by the built-in. `use Math::Complex` overloads `abs` to compute the modulus; and provide their own sign-stripping on blessed objects via operator overloading.

- **No in-place form:** `abs` always returns a new scalar. To replace a variable's value with its magnitude, assign back:

```
$x = abs $x;
```

- **Parser note:** the unary form `abs $x` binds tighter than `,` but looser than most arithmetic operators, so `abs $x + 1` parses as `abs($x) + 1`, not `abs($x + 1)`. Parenthesise when in doubt:

```
print abs $x + 1;           # abs($x) + 1
print abs($x + 1);         # abs of the sum
```

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `sqrt` - square root; like `abs` it is a pure numeric unary with a default-to-`$_` form
- `int` - truncate toward zero; pairs with `abs` when you want the integer magnitude of a float
- `sprintf` - formatted numeric output; use `%d` or `%g` together with `abs` to render unsigned magnitudes
- `$_` - the default argument when `abs` is called without one

Sockets

accept

Accept an incoming connection on a listening socket.

`accept` is the server side of the connection handshake. It waits on `GENERICSOCKET` - a socket that has already been created with `socket`, bound with `bind`, and placed in the listening state with `listen` - until a client connects, then installs a brand-new connected socket in `NEWSOCKET` and returns the client's packed address. `GENERICSOCKET` stays in the listening state, ready for the next call. Behaviour mirrors the POSIX `accept(2)` system call.

Synopsis

```
accept NEWSOCKET, GENERICSOCKET
```

What you get back

The client's address in the packed form used by the address family of `GENERICSOCKET` - the same shape you'd pass to `connect` or unpack with `Socket:::unpack_sockaddr_in` for IPv4, or with `Socket:::unpack_sockaddr_in6/unpack_sockaddr_un` for IPv6 and Unix-domain sockets respectively. On failure returns a false value (the empty string) and sets `$_!` to the underlying `errno`.

NEWSOCKET is populated as a side effect - it is a plain filehandle argument, not a return value. A bareword like `CLIENT` autovivifies a typeglob; a lexical filehandle like `my $client` is populated in place:

```
my $client;
accept($client, $server) or die "accept: $!";
# $client is now a readable/writable handle to the connected peer
```

Global state it touches

- `$!` - set on failure to the underlying `errno` (`EINTR`, `EAGAIN/EWOULDBLOCK` on non-blocking listeners, `ECONNABORTED`, `EMFILE`, `ENFILE`).
- `$^F` - the system maximum file descriptor. On systems that support the close-on-exec flag, `accept` sets `FD_CLOEXEC` on the new descriptor when its number is greater than `$^F` (default 2, covering the standard streams). Raise `$^F` before `accept` if you want the accepted socket to survive `exec`.

Examples

Minimal TCP echo-style accept loop:

```
use Socket;

socket(my $server, PF_INET, SOCK_STREAM, getprotobyname("tcp"))
    or die "socket: $!";
bind($server, sockaddr_in(8080, INADDR_ANY)) or die "bind: $!";
listen($server, SOMAXCONN)                  or die "listen: $!";

while (my $peer = accept(my $client, $server)) {
    my ($port, $iaddr) = sockaddr_in($peer);
    print $client "hello ", inet_ntoa($iaddr), ":$port\n";
    close $client;
}
die "accept: $!"; # loop exits only on error
```

Unpack the returned address for logging:

```
use Socket;

my $peer = accept(my $client, $server)
    or die "accept: $!";
my ($port, $iaddr) = sockaddr_in($peer);
printf "connection from %s port %d\n", inet_ntoa($iaddr), $port;
```

Non-blocking `accept`. Put `$server` in non-blocking mode first; a pending-connection-less call then returns false with `$!` set to `EAGAIN` or `EWOULDBLOCK`:

```
use Errno qw(EAGAIN EWOULDBLOCK);
use Fcntl qw(F_GETFL F_SETFL O_NONBLOCK);
```

(continues on next page)

(continued from previous page)

```
my $flags = fcntl($server, F_GETFL, 0);
fcntl($server, F_SETFL, $flags | O_NONBLOCK);

my $peer = accept(my $client, $server);
if (!$peer) {
    if ($! == EAGAIN || $! == EWOULDBLOCK) {
        # no pending connection - try again later
    } else {
        die "accept: $!";
    }
}
```

Preserve the accepted socket across `exec` by raising `^F` so the runtime does not set `FD_CLOEXEC`:

```
local ^F = 10_000;
accept(my $client, $server) or die "accept: $!";
exec "/usr/libexec/handler", fileno($client);
```

Edge cases

- **GENERICSOCKET not listening** - `accept` fails with `$!` set to `EINVAL`. Call `listen` on the server socket first.
- **NEWSOCKET already open** - the existing handle is silently closed before the new descriptor is installed, exactly as for `open` and `socket`. If the old handle was the only reference to a buffered stream, any unflushed data is lost. Close explicitly beforehand when that matters.
- **Signal during the call** - a signal delivered while `accept` is blocked makes the underlying syscall return `EINTR`. pperl does not auto-restart: `accept` returns false and leaves `$!` set to `EINTR`. Wrap in a retry loop if your signal handlers do not terminate the process.
- **Descriptor exhaustion** - `EMFILE` (process limit) or `ENFILE` (system limit) on a busy server is the common reason `accept` starts failing under load. Treat them as transient; close idle clients and retry.
- **Address-family mismatch of the packed return value** - the packed buffer follows the family of `GENERICSOCKET`. Unpacking a Unix-domain address with `Socket::unpack_sockaddr_in` produces garbage. Track the family yourself or use `Socket::sockaddr_family` to dispatch.
- **Parentheses are optional, commas are not** - `accept $c, $s` and `accept($c, $s)` both work; `accept $c $s` is a syntax error.
- **List vs scalar context** - the packed-address return is a byte string; list context does **not** expand it into `(port, addr)`. Call `sockaddr_in` / equivalent to `unpack`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `socket` - create the server socket that accept waits on
- `bind` - attach the server socket to a local address before `listen` and accept
- `listen` - mandatory step between `bind` and accept; sets the backlog depth
- `connect` - the client-side counterpart; accept returns the address a connect call supplied
- `getpeername` - recover the same packed address later from NEWSOCKET itself
- `Socket` - the constants (PF_INET, SOCK_STREAM, INADDR_ANY) and packers (sockaddr_in, unpack_sockaddr_in) that turn the packed return value into something usable
- `$^F` - threshold that controls whether accept sets FD_CLOEXEC on the new descriptor

Processes

alarm

Schedule a SIGALRM to be delivered to the current process after a whole number of wall-clock seconds.

`alarm` arms a single per-process countdown timer. When the timer expires, the kernel delivers SIGALRM; what happens next is entirely up to the installed signal handler in `%SIG`. The timer runs in wallclock time, not CPU time, so a sleeping or blocked process still receives the signal on schedule (subject to scheduling jitter).

Synopsis

```
alarm SECONDS
alarm
```

With no argument, the value in `$_` is used.

What you get back

The number of seconds remaining on the **previous** timer, or 0 if no timer was pending. This lets you save and restore a caller's alarm across a nested timed operation:

```
my $prev = alarm 10;
# ... do something that must finish in 10s ...
alarm $prev;                # restore caller's timer
```

`alarm 0` cancels any pending timer and returns the seconds that were left on it.

Global state it touches

- `%SIG` - the ALRM slot names the handler invoked when the timer fires. Without a handler, the default disposition of SIGALRM is to terminate the process.
- `$_` - read when alarm is called with no argument.
- `$@` - carries the die message out of the canonical `eval { alarm ...; ...; alarm 0 }` idiom.
- `$!` - a blocking syscall interrupted by SIGALRM sets `$!` to `EINTR`, **but** Perl restarts many syscalls automatically on some systems, so you cannot rely on seeing `EINTR`. Use `eval` / `die` instead (see below).

The canonical timeout idiom

Use `alarm` with `eval` and `die` to put a wall-clock cap on any block of code, including blocking I/O:

```
eval {
    local %SIG{ALRM} = sub { die "timeout\n" };    # \n matters
    alarm $timeout;
    my $nread = sysread $sock, $buf, 4096;
    alarm 0;                                       # cancel on_
    ↪ success
};
if ($@) {
    die $@ unless $@ eq "timeout\n";             # rethrow others
    # ... handle timeout ...
}
```

Three details are load-bearing:

- `local` scopes the handler to the `eval` block so it does not leak to the caller.
- The `\n` in `"timeout\n"` suppresses the file/line suffix that `die` would otherwise append, making the exact-match test in `$@` reliable.
- `alarm 0` **inside** the `eval` cancels the timer on the success path. Without it, a timer left armed can fire during unrelated code after the `eval` returns.

Examples

Simple wallclock deadline:

```
$SIG{ALRM} = sub { die "timeout\n" };
eval {
    alarm 5;
    my $line = <$slow_pipe>;                # may block indefinitely
    alarm 0;
};
warn "gave up waiting\n" if $@ eq "timeout\n";
```

Nested timers - save and restore:


```
my $saved = alarm 30;           # remember caller's timer
do_something();
alarm $saved;                   # put it back
```

Cancel a pending timer:

```
my $left = alarm 0;            # 0 if none was pending
```

Periodic tick without `Time::HiRes`:

```
$SIG{ALRM} = sub {
    print "tick\n";
    alarm 1;                       # re-arm from inside the handler
};
alarm 1;
sleep 10;                          # do not mix sleep+alarm in real_
↪code
```

Sub-second delay - use `Time::HiRes`:

```
use Time::HiRes qw(alarm);
alarm 0.25;                       # 250 ms
```

Edge cases

- **Integer seconds only.** The core alarm truncates fractional arguments to whole seconds. For finer granularity, load `Time::HiRes`, which exports a replacement alarm that accepts floats.
- **Scheduling jitter.** The signal may arrive up to roughly one second early or late because of how the kernel counts seconds and schedules your process. Treat alarm `N` as a *not-before-N-minus-1* bound, not a precise deadline.
- **One timer per process.** Each alarm call replaces the previous timer; there is no queue. The return value is your only record of what was overwritten.
- **Default disposition is fatal.** With no handler installed for `ALRM`, timer expiry terminates the process with `Alarm clock`. Install a handler *before* arming the timer.
- **Do not mix with `sleep`.** On many systems `sleep` is implemented on top of alarm, so arming one clobbers the other. Pick one per block.
- **EINTR is unreliable.** Perl installs signal handlers with `SA_RESTART` on many platforms, so a `SIGALRM` that interrupts a blocking syscall transparently restarts the call instead of returning with `$!` set to `EINTR`. The portable way to enforce a deadline is `eval / die` from the handler, as shown above.
- **Signals are delivered between Perl ops.** Inside a tight pure-Perl loop the handler runs at the next safe point, not instantly; a CPU-bound XS call can defer delivery further.
- **fork clears the timer.** Child processes start with no pending alarm regardless of what the parent had armed.

- **exec clears the timer** on Linux; the replacement process image starts with no pending SIGALRM.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `sleep` - wallclock delay; often implemented on top of alarm, so the two do not compose
- `die` - raise the exception from the ALRM handler that `eval` catches to implement a timeout
- `eval` - catches the die from the handler and lets the program recover from a timeout
- `%SIG` - where the ALRM handler is installed; the handler decides what the timer *means*
- `select` - the four-argument form accepts a floating-point timeout and is an alternative to alarm for I/O deadlines
- `Time::HiRes` - drop-in alarm replacement that accepts sub-second floats via `setitimer(2)`

Lists

all

Test whether BLOCK returns true for **every** element of LIST.

`all` walks LIST left to right, aliasing `$_` to each element in turn and evaluating BLOCK with that element. It returns a true value if every evaluation yields truth, and a false value as soon as any evaluation yields falsity. As soon as one element makes the block false, `all` stops and returns - it does not evaluate the block for the remaining elements.

This is the language-level `all` keyword, enabled by the `keyword_all` feature. It is distinct from `List::Util::all` - see *Edge cases* below.

Synopsis

```
use feature 'keyword_all';
no warnings 'experimental::keyword_all';

all { /\w+/ } @words;
all { defined $_ } @list;           # canonical "all defined" check
all { $_ > 0 } @numbers;
```

What you get back

A boolean. True if every element passes, false otherwise. The exact true/false values are unspecified - treat the result as a condition, not a datum to propagate.

Empty list returns true. This is *vacuous truth*: the statement “every element satisfies BLOCK” is trivially true when there are no elements to check. It is the single most common source of surprise with `all`:

```
all { $_ > 0 } ();                # TRUE - no elements, nothing to
→fail
```

Guard with an explicit size check when an empty list should be treated as failure:

```
if (@numbers && all { $_ > 0 } @numbers) { ... }
```

Global state it touches

- `$_` - aliased to each element of `LIST` in turn for the duration of the block. Because this is an alias and not a copy, assigning to `$_` inside the block mutates the underlying list element. The previous value of `$_` is restored when `all` returns.

Examples

Check that every value in a list is defined - the canonical use:

```
my @row = get_record();
die "missing field" unless all { defined $_ } @row;
```

Check that every number is positive:

```
my @prices = (1.99, 4.50, 0.99);
say "ok" if all { $_ > 0 } @prices;           # ok
```

Validate a record where every field must match its own predicate. Combine multiple tests inside the block rather than chaining `all` calls:

```
my @fields = ($name, $age, $email);
my $ok = all {
    defined $_ && length $_ > 0
} @fields;
```

Combine `all` with `any` for an exclusive check - at least one match, but not every element the same:

```
if (any { $_ eq 'admin' } @roles
    && !all { $_ eq 'admin' } @roles) {
    say "mixed roles, including admin";
}
```

Short-circuit behaviour: the block runs only until the first failure. Useful for predicates with side effects or cost:

```
all { expensive_check($_) } @candidates;    # stops at first failure
```

Edge cases

- **Empty list returns true.** Vacuous truth. Guard with `@list && all { ... }` `@list` when emptiness should fail.
- **Short-circuits on first false.** The block is not called for any element after the first failure. Do not rely on side effects in the block running for every element - use a `for` loop if you need that.
- **`$_` is aliased, not copied.** Modifying `$_` inside the block modifies the list element:

```
my @xs = (1, 2, 3);
all { $_++; $_ > 0 } @xs;
# @xs is now (2, 3, 4)
```

- **Experimental under 5.42.** The feature emits a compile-time warning in the `experimental::keyword_all` category unless that category is silenced:

```
no warnings 'experimental::keyword_all';
```

- **Not the same as `List::Util::all`.** The built-in is a language keyword enabled by the `keyword_all` feature and parses BLOCK as a true block (no leading `sub`, no trailing comma). `List::Util::all` is a regular subroutine imported from a module and takes a code reference / block prototype argument. For most practical purposes they behave the same on non-empty lists, but:
 - The keyword form is a parser-level construct and can be optimised more aggressively.
 - The two share neither implementation nor warning category.
 - Code that needs to run on pre-5.42 perls should use `List::Util::all`.
- **Equivalent `grep` form.** `all { COND } @xs` is semantically equivalent to `@xs == grep { COND } @xs` - but `grep` always evaluates the block for every element, so it lacks `all`'s short-circuit behaviour.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `any` - true if the block returns true for **at least one** element; the mirror operator to `all`
- `grep` - evaluates the block for every element and returns matches; scalar `@list == grep { ... } @list` is `all` without the short-circuit
- `defined` - the predicate inside the canonical `all { defined $_ } @list` check
- `List::Util::all` - module-level equivalent available on perls older than 5.42; also provides `none` and `notall` which have no language-level counterpart

- `List::Util::any` - module-level counterpart to the `any` keyword, alongside `none` and `notall` for the opposite polarities

Lists

`any`

Return true if BLOCK yields true for at least one element of LIST.

`any` runs BLOCK once per element of LIST, aliasing `$_` to the current element, and returns a true value the moment the block yields true. If no element makes the block true - including the case where LIST is empty - `any` returns false. It never runs the block more times than it has to: the first true result decides the answer and the remaining elements are not examined.

`any` is not unconditionally available. It is a feature-gated core keyword introduced in Perl 5.42; enable it with:

```
use feature 'keyword_any';
```

The long-standing `List::Util::any`, available since at least Perl 5.20, implements the same semantics as a subroutine. The two are interchangeable for everyday code - pick the core keyword when you want the slightly lower call overhead and an extra stack frame fewer in profiles, or `List::Util::any` when you also need other `List::Util` exports or have to support Perls older than 5.42.

Synopsis

```
use feature 'keyword_any';
any BLOCK LIST
```

What you get back

A boolean: a true value if any iteration of BLOCK yielded true, a false value otherwise. The exact shape of the true/false value is unspecified - treat the return as boolean, do not rely on getting 1 and "" specifically.

An empty LIST returns false. This is the mathematically correct answer (the existential quantifier over an empty domain is false) and matches `List::Util::any`.

Global state it touches

- `$_` is aliased to each element of LIST in turn while BLOCK runs. The aliasing is the same kind `map` and `grep` use: modifying `$_` inside the block modifies the list element itself. Use a named lexical via `my $x = $_` in the first line of the block if you need a safe copy.

Examples

Test whether any line in a list matches a pattern:

```
use feature 'keyword_any';
my @lines = ("foo\n", "bar ERROR baz\n", "quux\n");
if (any { /ERROR/ } @lines) {
    warn "log contains an error line\n";
}
```

Numeric threshold check:

```
use feature 'keyword_any';
my @nums = (1, 4, 7, 12, 3);
print "over limit\n" if any { $_ > 10 } @nums;    # over limit
```

Command-line flag detection, without pulling in a module:

```
use feature 'keyword_any';
if (any { $_ eq '--verbose' } @ARGV) {
    $verbose = 1;
}
```

any paired with `all` as a sanity check - every record has an id, and at least one is flagged active:

```
use feature qw(keyword_any keyword_all);
my @records = (
    { id => 1, active => 0 },
    { id => 2, active => 1 },
);
die "malformed" unless all { defined $_->{id} } @records;
die "none active" unless any { $_->{active} } @records;
```

Using the `List::Util` version when you cannot depend on Perl 5.42:

```
use List::Util qw(any);
print "found\n" if any { $_ eq 'needle' } @haystack;
```

Edge cases

- **Short-circuits on the first true result.** Side effects in BLOCK run only for elements examined before (and including) the first true element. Do not use any to drive a loop that must touch every element - use `for` or `map` for that.
- **Empty LIST returns false.** `any { anything } ()` is false without ever running the block.
- **`$_` aliasing is identical to `map` and `grep`.** Writing to `$_` mutates the source list element. The idiom `any { my $x = $_; ... }` makes the copy explicit.
- **`return` inside BLOCK returns from the enclosing sub,** not from any. To terminate the scan early with a specific truth value, let the block evaluate to that value - any already short-circuits.

- **Core keyword vs `List::Util::any`.** Both implement the same semantics. The core keyword avoids one Perl-level call frame per invocation of any itself, which shows up in tight profiling; the block itself still runs per-element either way. Mixing both in the same file is legal but confusing; pick one per file.
- **`experimental::keyword_any` warning.** Because the keyword is still flagged experimental, enabling it emits a warning in the `experimental::keyword_any` category on first use. Silence it with no warnings `'experimental::keyword_any'`; once you have decided to depend on the keyword.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `all` - the universal-quantifier counterpart; true iff BLOCK yields true for **every** element
- `grep` - collects every matching element; `scalar grep { ... } @list` has the same truth value as `any { ... } @list` but runs the block for every element
- `List::Util::any` - pre-5.42 subroutine form with identical semantics
- `List::Util::first` - returns the first element for which the block is true, rather than a boolean; use this when you need the element itself
- `map` - same `$_-` aliasing contract, but always visits every element and collects results

Numeric functions

atan2

Arc tangent of Y/X in the range $-\pi$ to π .

`atan2` is the two-argument arctangent - the inverse of tangent that keeps enough information about its inputs to return the correct quadrant. Unlike a single-argument `atan`, which collapses the two signs of Y and X into one ratio and cannot distinguish the upper half-plane from the lower, `atan2` looks at both arguments and returns an angle covering the full circle: $(-\pi, \pi]$ radians. It is the canonical way in Perl to convert Cartesian coordinates (X, Y) into the polar angle θ .

Synopsis

```
atan2 Y, X
```

What you get back

A floating-point number, the angle in radians between the positive x-axis and the point (X, Y) , measured counter-clockwise. Range: $-\pi < \text{result} \leq \pi$. Sign follows Y : positive Y gives a positive angle (upper half-plane), negative Y gives a negative angle (lower half-plane). To convert to degrees multiply by $180 / \pi$:

```
my $deg = atan2($y, $x) * 180 / 3.14159265358979;
```

Examples

Compute π to machine precision. The angle from the origin to $(0, 1)$ is exactly $\pi/2$, so `atan2(1, 0) * 2` gives π :

```
my $pi = atan2(1, 0) * 2;
printf "%.15f\n", $pi;           # 3.141592653589793
```

Quadrant handling. The four cardinal directions land on exact multiples of $\pi/2$, with $(-1, 0)$ at the boundary $+\pi$, not $-\pi$:

```
print atan2( 1,  0), "\n";      # 1.5707963267949    (+ $\pi/2$ , up)
print atan2( 0,  1), "\n";      # 0                    (right)
print atan2(-1,  0), "\n";      # -1.5707963267949    (- $\pi/2$ , down)
print atan2( 0, -1), "\n";      # 3.14159265358979    (+ $\pi$ , left)
```

Polar conversion. Given a point (X, Y) , recover radius and angle:

```
my ($x, $y) = (3, 4);
my $r      = sqrt($x*$x + $y*$y);    # 5
my $theta  = atan2($y, $x);          # 0.927295218001612 rad  $\approx 53.13^\circ$ 
```

Tangent via the standard identity. Perl has no built-in `tan`, but `sin` and `cos` give it directly - and `atan2` inverts the result unambiguously:

```
sub tan { sin($_[0]) / cos($_[0]) }
my $angle = 0.7;
print atan2(tan($angle), 1), "\n";   # 0.7    (recovers the input)
```

Full-circle sweep. Iterating Y around the unit circle shows the range covers $(-\pi, \pi]$:

```
for my $deg (0, 45, 90, 135, 180, -135, -90, -45) {
    my $rad = $deg * 3.14159265358979 / 180;
    printf "%4d° -> %+.4f rad\n",
        $deg, atan2(sin($rad), cos($rad));
}
```

Edge cases

- **Both arguments zero:** `atan2(0, 0)` is mathematically undefined. The value Perl returns is whatever the platform's C `atan2(3)` returns for this case - on glibc and musl that is 0, and POSIX permits any value in $[-\pi, \pi]$. Do not rely on a specific result; check for `($x == 0 && $y == 0)` before calling if zero is a real possibility in your inputs.
- **Signed zeros:** on IEEE-754 platforms `atan2(+0, -0)` returns $+\pi$ and `atan2(-0, -0)` returns $-\pi$. This only matters in code that deliberately tracks zero signs; most Perl code never distinguishes them.

- **Infinities:** `atan2(Inf, Inf) ==  $\pi/4$` , `atan2(Inf, -Inf) ==  $3\pi/4$` , `atan2(-Inf, Inf) ==  $-\pi/4$` , `atan2(-Inf, -Inf) ==  $-3\pi/4$` . The direction “toward ($\pm\text{Inf}$, $\pm\text{Inf}$)” is still a well-defined angle bisecting the appropriate quadrant.
- **NaN inputs:** if either Y or X is NaN, the result is NaN. Propagates through any further arithmetic - guard with a `Scalar::Util::looks_like_number` check on inputs from untrusted sources.
- **Non-numeric arguments:** strings are coerced to numbers by the usual Perl rules. `atan2("3", "4")` works; `atan2("abc", 1)` is equivalent to `atan2(0, 1)` and emits an `Argument "abc" isn't numeric` warning under `use warnings`.
- **Argument order is Y, X, not X, Y:** this matches C's `atan2(3)`, FORTRAN's `ATAN2`, and every other language's two-argument arctangent. Reversing the arguments gives the complement angle - a subtle bug that tests in only one quadrant will miss.
- **atan2 is not atan:** there is no one-argument form. `atan2($ratio)` is a syntax error at the call site. For the single-argument arctangent, use `Math::Trig::atan` or `POSIX::atan`, or write it as `atan2($ratio, 1)`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `sin` - sine; paired with `cos` to reconstruct an angle's coordinates after `atan2` has extracted it
- `cos` - cosine; the X component when converting a radius and angle back to Cartesian coordinates
- `sqrt` - square root; used with `atan2` in the Cartesian $\rightarrow$ polar conversion (`$r`, `$theta`) = (`sqrt($x**2+$y**2)`, `atan2($y,$x)`)
- - ships `tan`, `atan`, `asin`, `acos`, and the hyperbolic family; also exports `pi` as a constant so you do not have to write `atan2(1,0)*2`

Sockets

bind

Attach a local address to a socket.

`bind` is the Perl wrapper around the POSIX `bind(2)` system call. It associates the already-opened `SOCKET` filehandle with the local address `NAME`, so that subsequent `listen` / `accept` calls (for a server) or `connect` / `send` calls (for a client with a fixed local address) use that address as their source. `NAME` is a **packed** address structure in the form the socket's address family expects - the job of packing is yours, typically via `Socket` helpers such as ``sockaddr_in`` or ``sockaddr_un``.

Synopsis

```
bind SOCKET, NAME
```

What you get back

A true value on success, false on failure (with `$!` set to the underlying errno: `EADDRINUSE`, `EACCES`, `EADDRNOTAVAIL`, etc.). Always check the return value - unlike `print`, a failed bind is a setup-time error that has to abort the program:

```
bind $sock, $addr
    or die "bind failed: $!";
```

Global state it touches

None directly. `bind` sets `$!` on failure; that is the only interpreter global involved. The socket itself must already be open - typically produced by `socket` or `socketpair` - and `bind` mutates its kernel-side state, not any Perl-level variable.

Examples

Bind a TCP server socket to port 9000 on all local interfaces and start listening. `INADDR_ANY` selects “any local address”; the packed form comes from ``sockaddr_in``:

```
use Socket;

socket(my $srv, PF_INET, SOCK_STREAM, getprotobyname("tcp"))
    or die "socket: $!";
bind($srv, sockaddr_in(9000, INADDR_ANY))
    or die "bind: $!";
listen($srv, SOMAXCONN)
    or die "listen: $!";
```

Bind a client socket to a specific local address before connecting - used when the remote peer filters by source IP, or when a host has multiple interfaces and you need outgoing traffic on a specific one:

```
use Socket;

socket(my $cli, PF_INET, SOCK_STREAM, getprotobyname("tcp"))
    or die "socket: $!";
bind($cli, sockaddr_in(0, inet_aton("192.0.2.17")))
    or die "bind: $!";           # port 0 = kernel picks one
connect($cli, sockaddr_in(443, inet_aton("93.184.216.34")))
    or die "connect: $!";
```

Bind a UNIX-domain listener to a filesystem path. The path must not already exist - `bind` creates the socket node and leaves it behind when the process exits, so clean it up yourself:

```
use Socket;

my $path = "/tmp/myapp.sock";
unlink $path;                # clear stale socket from prior run
socket(my $srv, PF_UNIX, SOCK_STREAM, 0)
    or die "socket: $!";
bind($srv, sockaddr_un($path))
    or die "bind $path: $!";
listen($srv, 5) or die "listen: $!";
```

Bind a UDP socket to a fixed local port so replies from the peer come back to a predictable address:

```
use Socket;

socket(my $udp, PF_INET, SOCK_DGRAM, getprotobyname("udp"))
    or die "socket: $!";
bind($udp, sockaddr_in(5300, INADDR_ANY))
    or die "bind: $!";
```

Rebinding a previously-released port in quick succession requires `SO_REUSEADDR`; otherwise bind fails with `EADDRINUSE` during the `TIME_WAIT` window:

```
use Socket;

setsockopt($srv, SOL_SOCKET, SO_REUSEADDR, pack("l", 1))
    or die "setsockopt: $!";
bind($srv, sockaddr_in(9000, INADDR_ANY))
    or die "bind: $!";
```

Edge cases

- **SOCKET must already be open.** `bind` does not create the socket - call `socket` first. A closed or never-opened filehandle fails with `EBADF`.
- **NAME is a packed byte string, not a human-readable address.** Pass the output of ``sockaddr_in``, ``sockaddr_in6``, or ``sockaddr_un`` from `Socket`. Passing a plain string like `"127.0.0.1:9000"` produces `EINVAL` or worse - the kernel reads the bytes as a struct `sockaddr`.
- **Port 0 means “let the kernel choose”.** After a successful bind with port 0, retrieve the actual port with `getsockname` and ``sockaddr_in`` in unpack mode.
- **Ports below 1024** require elevated privileges on Unix - bind fails with `EACCES` for an unprivileged process.
- **EADDRINUSE** is the most common failure. It fires when another process holds the address, when a prior socket on the same address is still in `TIME_WAIT`, or when you call `bind` twice on the same socket. `SO_REUSEADDR` (see `setsockopt`) addresses the `TIME_WAIT` case; the others are genuine conflicts.
- **UNIX-domain sockets leave a filesystem entry** that outlives the process. Remove it before a fresh bind (`unlink $path`) and on clean shutdown.

- **Binding after `connect` is an error.** Once a socket has been connected, its local address is fixed. Call `bind` first if you need a specific source address.
- **IPv6 sockets** use ``sockaddr_in6`` and `PF_INET6`; mixing families between the socket and the packed address fails with `EAFNOSUPPORT`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `socket` - create the socket filehandle that `bind` attaches to; must be called first
- `connect` - the client-side counterpart; sets the remote address instead of the local one
- `listen` - mark a bound socket as willing to accept incoming connections
- `accept` - pull the next pending connection off a bound, listening socket
- `getsockname` - recover the actual address `bind` chose, especially after binding to port 0
- `setsockopt` - set `SO_REUSEADDR` and friends *before* `bind` to control address reuse
- `Socket` - packs and unpacks the `NAME` argument; source of ``sockaddr_in``, ``sockaddr_un``, `INADDR_ANY`, `PF_INET`, and related constants

I/O

binmode

Set the I/O layer stack on a filehandle - typically to make it deliver raw bytes, or to attach a character encoding.

`binmode` reconfigures `FILEHANDLE` so that subsequent reads and writes go through a specific PerlIO layer stack. In the one-argument form it puts the handle into raw binary mode (no CRLF translation, no character decoding). In the two-argument form it pushes (or, for a handful of pseudo-layers, replaces or pops) the named I/O layers onto the handle's stack. Call it after `open` and before any I/O on the handle, except for `:encoding`, which may be applied mid-stream.

Synopsis

```
binmode FILEHANDLE
binmode FILEHANDLE, LAYER
```

What you get back

1 on success, `undef` on failure, with `$!` set. Failure is rare in practice - passing a malformed layer string or an unknown layer name is the usual cause. The return value is worth checking when the layer argument is user-supplied:

```
binmode $fh, ":encoding($enc)"
    or die "bad encoding '$enc': $!";
```

If FILEHANDLE is an expression rather than a bareword or simple scalar, its value is taken as the name of the filehandle, matching the filehandle-resolution rules of `open` and `print`.

Global state it touches

`binmode` itself reads no special variables, but the layers it installs change how later I/O on the handle interacts with them. The layer stack governs how `$/` (input record separator) and `$\` (output record separator) are matched and emitted against the external byte stream - on platforms where text mode translates `\n` to a multi-byte sequence, the translation happens inside the layer, not in the variable.

PerlIO layers

A layer is a string beginning with `:` that names a filter in the PerlIO stack - see [PerlIO](#) for the full catalogue. Multiple layers may be given in one call by concatenating them:

```
binmode $fh, ":raw:utf8";
```

The order is bottom-up: `:raw` is applied first (closest to the OS file descriptor), then `:utf8` sits on top of it. Reads traverse the stack from bottom to top; writes traverse it from top to bottom.

The commonly used layers:

- `:raw` - strip the handle down to byte-for-byte I/O. Turns off CRLF translation, removes any character-encoding layers, and marks the handle as bytes. Despite occasional claims to the contrary, `:raw` is **not** simply the inverse of `:crlf`; it also disables any other layer that would alter the binary nature of the stream. This is what the one-argument form installs.
- `:crlf` - translate `\r\n` sequences to `\n` on input and `\n` to `\r\n` on output. This is the default on Windows / DOS-family systems for text-mode handles; applying it explicitly forces CRLF translation on any platform.
- `:utf8` - mark data on the handle as UTF-8. No validation on input: bytes are accepted and flagged as characters. On output, characters are encoded to UTF-8 bytes. Fast, but trusts the source.
- `:encoding(NAME)` - decode bytes to characters on input and encode characters to bytes on output, using the named encoding (UTF-8, iso-8859-1, shiftjis, etc.). Input that is not valid for the encoding triggers a warning and substitution. `:encoding` implicitly pushes `:utf8` on top of itself because Perl works on UTF-8 internally. See [PerlIO:encoding](#) for details and tuning options.
- `:bytes` - the inverse of `:utf8`; mark data on the handle as bytes, disabling any character-semantics flag that a higher layer set.

Two pseudo-layers control stack shape rather than filtering:

- `:pop` - remove the topmost layer from the handle.

- `:push` - used with `:via(...)` to stack a custom layer module.

`:raw` in the two-argument form acts as a reset: it pops layers until the handle is in its most minimal state, rather than pushing a new layer on top.

Examples

The one-argument form, for reading a binary file portably:

```
open my $img, "<", "photo.jpg" or die $!;
binmode $img;
my $bytes = do { local $/; <$img> };
```

Read a UTF-8 text file with validation:

```
open my $fh, "<", "notes.txt" or die $!;
binmode $fh, ":encoding(UTF-8)";
while (my $line = <$fh>) {
    # $line contains decoded characters, not bytes
}
```

Write Windows-style line endings on any platform:

```
open my $out, ">", "dos.txt" or die $!;
binmode $out, ":crlf";
print $out "one\ntwo\nthree\n";    # written as one\r\ntwo\r\nthree\r\n
```

Strip a previously-applied encoding layer and drop back to bytes:

```
binmode $fh, ":pop";                # remove topmost layer
binmode $fh, ":raw";                # or: flatten to bare bytes
```

Validate a user-supplied encoding before committing to it:

```
my $enc = $ENV{INPUT_ENCODING} // "UTF-8";
binmode $fh, ":encoding($enc)"
    or die "unsupported encoding '$enc': $!";
```

Edge cases

- **Call after open, before I/O.** `binmode` on a handle that has already been read from or written to produces defined but implementation-dependent results; most layers flush pending buffers on install, which may lose data that crossed the layer boundary. `:encoding` is the documented exception - it may be pushed mid-stream without a flush.
- **One-argument form is `:raw`, not “do nothing”.** On Unix the observable effect is often nil because handles are already in byte-oriented mode, but on Windows it disables CRLF translation. Write portable code as if `binmode $fh` is always meaningful.

- **CRLF on Windows / DOS.** Without binmode, the C runtime on those platforms converts `\r\n` to `\n` on input and the reverse on output. For binary formats (images, compressed data, serialised structures) this silently corrupts the stream. binmode is mandatory there.
- **Ctrl-Z as end-of-file.** On Microsoft-family systems, text-mode handles treat `\cZ` (byte 0x1A) as end-of-file. Binary data that happens to contain that byte will appear truncated unless binmode is used.
- **:utf8 is trust-only.** An input handle with `:utf8` but no `:encoding(UTF-8)` will happily hand you invalid UTF-8 flagged as characters. Use `:encoding(UTF-8)` when the source is untrusted.
- **:encoding pulls in :utf8.** After `binmode $fh, ":encoding(shiftjis)"` the handle has both `:encoding(shiftjis)` and `:utf8` in its stack - perl's internal form is UTF-8 regardless of the wire encoding.
- **Filehandle expressions.** `binmode $handles[0], ":utf8"` works; any expression that yields a filehandle value is accepted as the first argument and is evaluated to name the handle.
- **Affects every I/O primitive on the handle.** The layer stack governs `read`, `sys-read`, `syswrite`, `seek`, `tell`, `print`, and `readline` alike. In particular, `sysread` and `syswrite` on a `:utf8`-marked handle still go through the layer and may return partial characters.
- **Interaction with the open pragma.** use `open ":encoding(UTF-8)"` installs a default layer set for subsequently-opened handles. Explicit binmode overrides that default for a given handle.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `open` - create the filehandle in the first place; accepts layers directly in the mode string (`<:encoding(UTF-8)`) to skip a follow-up binmode call
- `Encode` - encode and decode strings in memory, independent of any filehandle
- `PerlIO` - full layer catalogue, including less common layers like `:scalar`, `:via`, and `:mmap`
- `read` / `sysread` - input primitives whose byte / character semantics depend on the layer stack
- `print` / `syswrite` - output primitives whose encoding and line-ending behaviour depend on the layer stack
- `$/` - input record separator, matched against the post-layer byte stream
- `$\` - output record separator, written through the layer stack

Classes and OO

bless

Mark the thing a reference points at as an object in a package.

`bless` associates the referent of `REF` with the package named by `CLASSNAME`, so that method calls through `REF` dispatch against that package's subroutines (and its `@ISA` ancestors). The reference itself does not change; what changes is the referent's class tag. `bless` returns `REF`, which is why it is almost always the last statement in a constructor.

Synopsis

```
bless REF, CLASSNAME
bless REF
```

What you get back

The same reference that was passed in as the first argument, now carrying the class tag of `CLASSNAME`. Returning the reference lets you tail-call `bless` from a constructor:

```
sub new {
    my ($class, %args) = @_;
    return bless { %args }, $class;
}
```

From that point on, `ref` on the returned reference yields the class name rather than the raw reference type (HASH, ARRAY, CODE, ...).

Forms

Two calling forms, one strongly preferred:

- **Two-argument - always use this.**

```
bless $ref, $class;
```

The referent becomes an object in the package named by `$class`. Using the first argument of the constructor (`$_[0]`, conventionally `$class`) keeps the constructor inheritable: a subclass calling `Parent::new('Child', ...)` gets back an object blessed into `Child`, not into `Parent`.

- **Two-argument with empty string - bless into main.**

```
bless $ref, "";
```

An empty `CLASSNAME` is treated as the main package. Rarely useful outside of deliberate tricks.

- **One-argument - discouraged.**

```
bless $ref;
```

The referent is blessed into the *current* package (the package the `bless` call is compiled in). This breaks inheritance: a subclass that inherits a one-argument `bless`

constructor still ends up with objects in the parent's package, not the caller's. Use the two-argument form unless you have a concrete reason not to, and document the reason.

Examples

A minimal hash-based class:

```
package Point;

sub new {
    my ($class, $x, $y) = @_;
    return bless { x => $x, y => $y }, $class;
}

sub x { $_[0]->{x} }
sub y { $_[0]->{y} }

package main;

my $p = Point->new(3, 4);
print ref $p, "\n";          # Point
print $p->x, ", ", $p->y;    # 3,4
```

Inheritable constructor - the same new works for a subclass without modification:

```
package Shape;
sub new {
    my ($class, %args) = @_;
    return bless { %args }, $class;
}

package Circle;
our @ISA = ('Shape');
sub area {
    my $self = shift;
    return 3.14159 * $self->{radius} ** 2;
}

package main;
my $c = Circle->new(radius => 5);
print ref $c, "\n";          # Circle
print $c->area, "\n";        # 78.53975
```

Array-based objects work the same way - any reference type can be blessed:

```
package Pair;
sub new {
    my ($class, $a, $b) = @_;
    return bless [$a, $b], $class;
}
```

(continues on next page)

(continued from previous page)

```

sub first { $_[0]->[0] }
sub second { $_[0]->[1] }

my $p = Pair->new("left", "right");
print $p->first, "/", $p->second;  # left/right

```

Re-blessing an existing object changes its class tag in place:

```

my $obj = bless {}, 'OldClass';
bless $obj, 'NewClass';
print ref $obj;           # NewClass

```

bless returns the reference, which is why chaining into method calls on the same line works:

```

Logger->new(file => $path)->info("started");
#                                     ^ called on the blessed ref

```

Edge cases

- **The reference is what gets tagged, not the variable.** Copies of the reference share the class tag because they all point to the same referent:

```

my $a = bless {}, 'Foo';
my $b = $a;
print ref $b;           # Foo

```

- **Re-blessing is permitted and immediate.** A reference can be blessed as many times as you like; each call overwrites the previous class tag. There is no “unbless” built-in - rebless into a neutral package if you need to strip objecthood from a referent you still hold references to.
- **CLASSNAME of "" means main.** Explicit and rarely wanted; usually a sign that a computed class name came out empty by accident.
- **Avoid bless \$ref, "0".** The class name "0" is a false value, so code that checks `if (ref $obj) { ... }` will misread the object as “not a reference”. Pick any non-false class name.
- **Class-name casing convention.** Use mixed-case (`My::Package`). ALL-CAPS names are reserved for Perl built-in types (`SCALAR`, `ARRAY`, `HASH`, `CODE`, ...) and all-lowercase names are reserved for pragmas (`strict`, `warnings`, `utf8`). Blessing into either namespace invites confusion with language-level machinery.
- **Non-reference first argument is a fatal error.** `bless 42, 'Foo'` dies with `Can't bless non-reference value`. The first argument must be a reference.
- **One-argument bless picks the compile-time package.** It uses the package the `bless` call is *lexically* inside - not the caller’s package, not the package of `$class`. That is exactly why it breaks inheritance and why the two-argument form exists.
- **Weak references and blessing interact as expected.** `Scalar::Util::weaken` on a blessed reference does not unbless it; the weakened slot still sees the class tag,

and `ref` works until the referent is destroyed.

- **DESTROY fires on the blessed referent.** When the last reference to a blessed referent goes away, Perl calls DESTROY in the reference's class (following `@ISA`). Changing the class with a late bless changes which DESTROY runs.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `ref` - read the class tag back off a blessed reference, or the reference type (HASH, ARRAY, ...) for an unblessed one
- `package` - declare the package a bless call or a subroutine definition compiles into; the target of one-argument bless
- `class` - the experimental class-syntax built-in that replaces the hand-rolled `package + bless` constructor pattern with declarative syntax
- `tie` - a different form of per-variable magic; unrelated to bless despite the superficial “attach behaviour to a value” framing
- `@ISA` - the inheritance chain consulted when dispatching a method call on a blessed reference
- DESTROY - the destructor method Perl calls on the blessed referent when its last reference goes away; declared as a subroutine inside the class's package

Control flow

break

Break out of a given block.

`break` terminates the enclosing given block immediately and transfers control to the statement following it - the given analogue of what `last` does for loops. It takes no arguments, returns no value, and applies only to the innermost given; it cannot target an outer block by label.

Synopsis

```
break
```

Availability - read this first

`break` is a keyword of the switch feature, together with `given`, `when`, `default`, and (inside a `when`) `continue`. In Perl 5.42 the switch feature is **disabled by default** and is not part of any current feature bundle: use `v5.36` through use `v5.42` do **not** turn it on. The feature still exists and can be used, but only under explicit opt-in:

```
use feature 'switch';
no warnings 'experimental::smartmatch';
```

Without that opt-in, the bareword `break` is parsed as an ordinary subroutine call and fails at compile time with Bareword `"break"` not allowed while `"strict subs"` in use, or at runtime with Undefined subroutine `&main::break` called. The fully-qualified form `CORE::break` works from any scope without enabling the feature and is how you invoke it in code that otherwise keeps `switch` off.

`given/when` rely on `smartmatch (~~)`, which is itself experimental and warns on every use unless `experimental::smartmatch` warnings are suppressed. New code should **not** reach for `given/when/break`; see [Replacements](#) below.

Behaviour

Inside a `given` block, `break` unwinds to the statement immediately after the block. It is equivalent to what falls through the bottom of a `when` branch implicitly - `when` clauses `break` at the end of their body unless the body ends in `continue`, so an explicit `break` is mostly used for early exit from a long `when` body:

```
use feature 'switch';
no warnings 'experimental::smartmatch';

given ($cmd) {
    when ('quit') {
        cleanup();
        break if $fast_exit;           # skip the logging below
        log_exit();
    }
    when ('help') { show_help() }
    default      { warn "unknown: $cmd" }
}
```

Outside a `given` block, `break` raises `Can't "break" outside a given block at runtime`.

Replacements

For new code, prefer one of these forms. They are plain Perl, need no feature opt-in, and have no deprecation cloud over them:

- **if / elsif / else chain** - direct replacement for most `given/when` usage:

```
if ($cmd eq 'quit') { cleanup(); log_exit() }
elsif ($cmd eq 'help') { show_help() }
else { warn "unknown: $cmd" }
```

- **Dispatch table** - a hash of coderefs keyed by the input, with a default branch. Scales better than a long `elsif` ladder and makes the set of handled cases data rather than control flow:

```
my %dispatch = (
    quit => sub { cleanup(); log_exit() },
    help => \&show_help,
```

(continues on next page)

(continued from previous page)

```
);
($dispatch{$cmd} // sub { warn "unknown: $cmd" }->());
```

- **Labelled loop with `last`** - when you specifically want the “exit this block early” shape that `break` provided, wrap the body in a bare block and `last` out of it:

```
CMD: {
    if ($cmd eq 'quit') { cleanup(); last CMD if $fast_exit;
    log_exit(); last CMD }
    if ($cmd eq 'help') { show_help(); last CMD }
    warn "unknown: $cmd";
}
```

Examples

Explicit feature opt-in, early-exit from a when body:

```
use feature 'switch';
no warnings 'experimental::smartmatch';

given ($n) {
    when ($_ < 0) { warn "negative"; break }    # skip the log below
    when (0)      { warn "zero" }
    default      { warn "positive: $n" }
}
# control resumes here after break
```

Using `CORE::break` without enabling the feature globally:

```
sub classify {
    my ($n) = @_;
    given ($n) {
        when ($_ < 0) { return "neg"; CORE::break }
        when (0)      { return "zero" }
        default      { return "pos" }
    }
}
```

Edge cases

- **No argument form.** `break` takes nothing. `break LABEL` is a syntax error - unlike `last/next`, `break` has no way to target an outer block.
- **Outside given.** Calling `break` (or `CORE::break`) when no `given` block is on the call stack raises Can't "break" outside a given block.
- **return wins.** Inside a subroutine, `return` unwinds the whole sub regardless of any enclosing `given`. `break` only unwinds to the end of the `given`; control continues in the surrounding sub.

- **Implicit break at end of when.** A when clause's body ends with an implicit break unless the body is continue, so an explicit break at the end of a when body is redundant. continue suppresses the implicit break and lets control fall through to the next when.
- **Not a loop keyword.** break does not interact with for, while, foreach, do, or bare blocks. For those, use `last`.
- **Experimental warnings.** Running given/when code without no warnings 'experimental::smartmatch' prints the warning once per smartmatch site, which in a busy given block can be every iteration.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `last` - the loop counterpart; exits the innermost enclosing for / while / foreach / bare block, and unlike break can target an outer block by label
- `next` - start the next iteration of a loop rather than exiting it; useful when break is the wrong shape because you want continuation, not termination
- `return` - unwinds out of the enclosing subroutine entirely, not just the given block; use when the given is a dispatcher inside a sub and the answer is ready

Control flow · Scoping

caller

Return information about the subroutine, `eval`, or `require` that called the currently executing code.

`caller` walks the call stack. With no argument it returns a short three-element record identifying *who called us*. With a numeric argument `EXPR` it walks `EXPR` frames further back and returns a longer eleven-element record - the form the debugger, stack-trace modules, and `warn/die` machinery use to decorate diagnostics with file-and-line context. XS frames are invisible to `caller`; the next pure-Perl frame surfaces in their place.

Synopsis

```
caller
caller EXPR
```

What you get back

In scalar context, the caller's package name, or `undef` if there is no caller (i.e. we are at the top level of the file, not inside a `sub`, `eval`, or `require`):

```
my $pkg = caller;
```

In list context with no argument, three elements - the immediate caller's package, file, and line number:

```
# 0      1      2
my ($package, $filename, $line) = caller;
```

In list context with a numeric argument, eleven elements describing the frame EXPR levels up. 0 is the immediate caller, 1 is its caller, and so on:

```
# 0      1      2      3      4
my ($package, $filename, $line, $subroutine, $hasargs,
# 5      6      7      8      9      10
    $wantarray, $evaltext, $is_require, $hints, $bitmask, $hinthash)
= caller($i);
```

The eleven fields, in order:

- `$package`, `$filename`, `$line` - location of the *call site* inside that frame's caller. Same three values the short form returns.
- `$subroutine` - fully qualified name of the sub the frame belongs to, e.g. "Foo::bar". For an `eval` frame it is the literal string "(eval)". If the sub has been removed from the symbol table since the call, it is "(unknown)".
- `$hasargs` - true if a fresh `@_` was set up for the frame. False for `eval` and `require` frames.
- `$wantarray` - the context the frame was called in, as `wantarray` would have reported it: true for list, defined-but-false ("") for scalar, `undef` for void.
- `$evaltext` - for an `eval` EXPR frame, the source text that was compiled. `Undef` for `eval` BLOCK and for non-`eval` frames.
- `$is_require` - true if the frame was created by `require` or `use` (each `use` creates a `require` frame nested inside an `eval` EXPR frame).
- `$hints` - the value of `%^H` at the point the frame's code was compiled. Internal; layout is not stable across Perl versions.
- `$bitmask` - the value of `$_{^WARNING_BITS}` at the same point. Also internal.
- `$hinthash` - a reference to the `%^H` hash captured at compile time, or `undef` if it was empty. Do **not** modify it - the reference points at the live optree.

Global state it touches

- `@DB:::args` - set as a side effect when `caller` is called with an argument *from inside the DB package*. The caller's argument list is aliased into this array for debugger use. See *Edge cases* below - this is best-effort and has sharp corners.
- `%^H`, `$_{^WARNING_BITS}`, `%^H` - read out of the target frame's compiled state to populate `$hints`, `$bitmask`, `$hinthash`.

Examples

Find the immediate caller's package and report to the right log:

```
sub log_for_caller {
    my $pkg = caller;
    warn "[$pkg] @_\\n";
}
```

Produce a `warn`-style location tag. `caller` returns filename and line of the call site one frame up, which is almost always the information you actually want in a diagnostic:

```
sub note {
    my (undef, $file, $line) = caller;
    print STDERR "note at $file line $line: @_\\n";
}
```

Walk the full stack. Iterate `caller($i)` until it returns an empty list:

```
sub stacktrace {
    my @frames;
    for (my $i = 0; my @f = caller($i); $i++) {
        push @frames, \@f;
    }
    return @frames;
}
```

Detect whether the current sub was called in list, scalar, or void context. This is `wantarray` at the current level, but for somebody else's frame you reach through `caller`:

```
sub describe_parent_context {
    my $wa = (caller(1))[5];
    return !defined $wa ? "void"
           : $wa        ? "list"
           :              "scalar";
}
```

Tell a `require/use` frame apart from an ordinary `eval`:

```
sub called_from_require {
    my @f = caller(1);
    return @f && $f[7];           # field 7 is $is_require
}
```

Render a single frame the way the default `die` / `Carp` formatter does - "at FILE line LINE":

```
sub location_tag {
    my $depth = shift // 0;
    my (undef, $file, $line) = caller($depth);
    return defined $file ? "at $file line $line" : "at top level";
}
```


Edge cases

- **Scalar context, no caller:** at the top level of a script, scalar `caller` returns `undef`, and list-context `caller` returns the empty list. A common idiom for “am I being `required` as a module, or run as a script?” is `caller() ? ... : ...`.
- **XS frames are skipped.** `caller` only reports pure-Perl frames. If a pure-Perl sub was invoked by an XS sub which was invoked by another pure-Perl sub, `caller(0)` from the inner sub reports the outer pure-Perl sub - the XS layer is invisible. Stack depths measured with `caller` therefore do not include XS dispatchers.
- **eval frames have no args.** `$hasargs` is false for `eval` and `require` frames, and `$subroutine` is the literal `"(eval)"` - not the package of the surrounding code.
- **eval BLOCK vs eval EXPR.** Both give `$subroutine` eq `"(eval)"`. Only `eval EXPR` fills `$evaltext`; `eval BLOCK` leaves it `undef`.
- **use creates two frames.** A `use Foo` statement is compiled as `BEGIN { require Foo; Foo->import; }`, and `require` itself is implemented as an `eval EXPR`. Walking the stack through a `use` line therefore sees a `require`-flagged `eval` frame inside an ordinary `eval` frame. Check `$is_require` to distinguish.
- **(unknown) subroutines.** If the sub named in a frame has been deleted from the symbol table (symbol-table surgery, `delete $Foo::{\bar}`) between the call and the caller, `$subroutine` reads `"(unknown)"`.
- **Loops and try blocks are not frames.** `while`, `for`, `foreach`, and `try` do not create call frames. `caller` only sees subroutine calls and `evals`.
- **Optimiser-elided frames.** The optimiser may remove tail-call and similar frames before `caller` can observe them. For `caller(N)` with `N > 1`, the frame you get is not guaranteed to be the frame you expected. Do not encode stack depth as a load-bearing contract.
- **@DB:::args is best-effort.** It is populated only when `caller` is called with an argument from inside package `DB`. The values are aliases into the target frame’s `@_`, not copies - so:
 - Mutations the frame has already made to `@_` are visible; original call-time values are not preserved.
 - `shift @_` inside the frame is usually undone in `@DB:::args`, but `pop @_`, `splice`, or having taken a reference to `@_` all break that.
 - Elements may have been freed and reused for other values by the time you read them.
 - Stale state from a previous `caller` call may still be in `@DB:::args` if the current call didn’t trigger a refresh. Use it for backtraces and debugger display; do not rely on it for program logic.
- **\$hynthash is live.** The hash reference in field 10 points at the actual `%^H` snapshot baked into the optree. Modifying its contents mutates compiler state. Read it; don’t write it.
- **Filename and line can be rewritten.** `#line` directives and anything else the “Plain Old Comments (Not!)” mechanism recognises (see `perlsyn`) change the values `caller` reports, just as they change `__FILE__` and `__LINE__`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `die` - raises an exception; its default message uses the same file-and-line information caller exposes
- `eval` - creates frames that caller tags with `$subroutine` eq `"(eval)"` and may fill `$evaltext` / `$is_require`
- `wantarray` - reports the current frame's calling context; the same value appears as field 5 of `caller(N)` for the frame N levels up
- `^H` - compile-time hint bits; surfaced as field 8
- `%H` - compile-time hint hash; surfaced as field 10
- `#{^WARNING_BITS}` - warnings bitmask; surfaced as field 9

Control flow

catch

Handle an exception thrown by a preceding `try` block.

`catch` is the second half of the `try/catch` construct. It is not a standalone statement: it follows a `try` block, declares a lexical VAR to receive the exception value, and runs its BLOCK only when the `try` block died. If the `try` block completed normally the `catch` block is skipped. Control then resumes at the statement following the whole construct (or the `finally` block, if one is present).

This syntax must be enabled with use feature 'try'. `try` and `catch` are non-experimental as of Perl 5.40.

Synopsis

```
try BLOCK catch (VAR) BLOCK
try BLOCK catch (VAR) BLOCK finally BLOCK
```

What you get back

`catch` is not a function call and has no return in the expression sense. Like other control-flow constructs, the value of the whole `try/catch` is the value of whichever block ran last - useful inside a `do` block or as the tail expression of a subroutine:

```
my $value = do {
    try {
        fetch_thing(@args);
    }
    catch ($e) {
        warn "fetch failed: $e";
    }
}
```

(continues on next page)

(continued from previous page)

```

    $DEFAULT;
}
};

```

If `try` ran to completion, the construct yields the `try` block's last value and `VAR` is never bound. If `try` died, the construct yields the catch block's last value.

The VAR declaration

The parentheses after `catch` are mandatory and must contain exactly one scalar variable declaration. It behaves like a `my` declaration

- the `my` keyword is implicit, as in subroutine signatures:

```

catch ($e) { ... }      # $e is a new lexical
catch (my $e) { ... }   # syntax error - 'my' is already implied

```

`VAR` is scoped to the `catch` block only. It is not visible in the preceding `try` block, in any following `finally` block, or after the construct. Its value is whatever the `try` block threw - a string, a list that stringified into one, or a reference, typically a blessed exception object. See `die` for how the exception value is produced.

Unlike `eval`, `catch` does **not** read or clear `$@`; the exception travels through `VAR` directly. `$@` is untouched by the construct.

Global state it touches

`catch` itself touches no special variables. The `try/catch` construct as a whole leaves `$@` unchanged - a deliberate divergence from `eval` intended to stop callers from conflating the two idioms. If you need `$@`-style inspection, use `eval` instead.

Examples

Catch a simple string exception and report it:

```

use feature 'try';

try {
    risky();
}
catch ($e) {
    warn "risky failed: $e";
}

```

Inspect a structured exception object:

```

try {
    process($record);
}
catch ($e) {

```

(continues on next page)

(continued from previous page)

```
if (ref($e) && $e->isa('My::NotFound')) {  
    return;                # missing is fine  
}  
die $e;                    # re-throw everything else  
}
```

Re-throw unchanged with `die`. Because `catch` does not touch `$@`, you re-throw the variable you bound, not `$@`:

```
try {  
    step();  
}  
catch ($e) {  
    log_error($e);  
    die $e;                # propagates to the next_  
→ handler  
}
```

Produce a value from the construct without `return`. Using `return` inside `try` or `catch` returns from the **enclosing subroutine**, not from the construct - see *Edge cases*:

```
sub safe_parse {  
    my ($text) = @_;  
    try {  
        parse($text);  
    }  
    catch ($e) {  
        warn "parse error: $e";  
        undef;              # value of the catch block  
    }  
}
```

A catch block that itself dies - the new exception propagates outward to the next enclosing handler (`try/catch`, `eval`, or program exit):

```
try {  
    open_db();  
}  
catch ($e) {  
    cleanup();  
    die "open_db failed: $e";    # thrown from inside catch  
}
```

Edge cases

- **Cannot stand alone.** `catch (VAR) { ... }` without a preceding `try` block is a syntax error. `catch` is parsed as part of the `try` construct, not as an independent keyword.
- **Exactly one scalar in parentheses.** The parentheses are mandatory, must declare

exactly one scalar, and the `my` keyword is implied. `catch { ... }` without parentheses is a syntax error.

- **VAR is block-scoped.** It exists only inside the `catch` block. A `finally` block that follows does **not** see it.
- **Falsy exceptions still trigger the handler.** `try` distinguishes “died” from “completed normally” by a flag set at unwind time, not by truthiness of the exception value. `die 0`, `die ""`, and `die undef` all enter the `catch` block; VAR receives the value as thrown (with `die`’s usual coercion of `undef` to “Died at ...”).
- **`$@` is not set.** Code that reaches for `$@` inside a `catch` block is reading the caller’s `$@`, not the exception just caught. Use VAR.
- **A `catch` block may die.** Exceptions raised from inside `catch` are **not** re-caught by the same construct. They propagate to the next enclosing handler. This is the mechanism for re-throw (`die $e`) and for reporting cleanup failures from the handler itself.
- **`return` inside `catch` returns from the enclosing subroutine,** not from the `catch` block. This differs from `eval` BLOCK, where `return` would leave only the `eval`. To yield a value *from* the construct, use the block’s final expression instead of `return`.
- **Loop controls (`last`, `next`, `redo`) inside `catch`** target an enclosing loop, not the construct. The `try/catch` construct is not itself a loop.
- **`caller()` does not see the `catch` frame.** Because `catch` does not intercept `return`, it is invisible to `caller`, like `while` or `foreach` are.
- **Pairs only with the immediately preceding `try`.** You cannot have two `catch` blocks, nor place statements between `try` and `catch`. The optional `finally` block, if present, must come after `catch`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `try` - the block whose failure this `catch` handles; always paired with exactly one `catch`
- `finally` - optional third block that runs regardless of whether the `try` block died; cannot `return` or use loop controls
- `die` - produces the exception value that VAR receives; use it inside `catch` to re-throw
- `eval` - older exception-handling idiom using `$@`; reach for `try/catch` instead in new code for clearer scoping and no `$@` footguns
- `$@` - left untouched by `try/catch`; inspect VAR instead

Filehandles, files, directories

chdir

Change the process's current working directory.

`chdir` is the Perl wrapper around the `chdir(2)` and `-` where the kernel supports it - `fchdir(2)` system calls. It changes the directory that relative paths are resolved against for the current process and, on success, persists that change for every subsequent file operation in the program.

Synopsis

```
chdir EXPR
chdir FILEHANDLE
chdir DIRHANDLE
chdir
```

What you get back

1 on success, 0 on failure. On failure `$!` is set to the underlying `errno` so you can distinguish “no such directory” (`ENOENT`) from “permission denied” (`EACCES`) from “not a directory” (`ENOTDIR`):

```
chdir $path
or die "chdir $path failed: $!";
```

The return is always a plain boolean - there is no separate “changed to nowhere because no argument was given” value. The no-argument form either succeeds by changing to the home directory or fails like any other call.

Global state it touches

- `%ENV` - read when `chdir` is called with no argument. The lookup order is `$ENV{HOME}` first, then `$ENV{LOGDIR}`. If neither is set, `chdir` does nothing and returns 0.
- `$!` - set on failure to the `errno` reported by the kernel.

`chdir` does **not** update `$ENV{PWD}`. Shells maintain `PWD` themselves; a Perl program that cares about `PWD` for child processes has to assign to `$ENV{PWD}` after a successful `chdir`.

Argument forms

- **`chdir EXPR`** - `EXPR` is stringified and passed to `chdir(2)` as a pathname. Relative paths are resolved against the current working directory at the moment of the call.
- **`chdir FILEHANDLE`** and **`chdir DIRHANDLE`** - on Linux (and any system with `fchdir(2)`) the kernel changes directory to the one the handle refers to. `FILEHANDLE` here means a filehandle opened on a directory, or a directory handle from `opendir`. Using the handle form avoids a TOCTOU race between resolving a pathname and changing into it.

- **chdir** (no argument) - changes to \$ENV{HOME}, falling back to \$ENV{LOGDIR} if HOME is unset. If both are unset the call fails and \$! is left untouched; check the return value, not \$!.

Examples

Change into a directory and bail out if it does not exist:

```
chdir "/var/log"
  or die "cannot enter /var/log: $!";
```

Run a block of work in a different directory and return to where you started:

```
use Cwd qw(getcwd);
my $origin = getcwd;
chdir $work_dir or die "chdir $work_dir: $!";
# ... do work relative to $work_dir ...
chdir $origin or die "chdir $origin: $!";
```

Change via a directory handle (race-free on systems with fchdir(2)):

```
opendir my $dh, $path or die "opendir $path: $!";
chdir $dh or die "fchdir $path: $!";
```

Go home:

```
chdir or die "no HOME/LOGDIR in environment";
```

Probe a list of candidates, stopping at the first that works:

```
for my $d ("/srv/app", "/opt/app", "/usr/local/app") {
    last if chdir $d;
}
```

Edge cases

- **Empty string or undef:** `chdir ""` and `chdir undef` both fail and set \$! to ENOENT. They do **not** fall back to \$HOME
 - the no-argument form is triggered by *absence* of an argument, not by an argument that happens to be empty.
- **Relative paths on a thread with its own cwd:** the kernel's current working directory is per-process, not per-thread, on Linux. Two threads calling `chdir` concurrently race.
- **Symlink to a directory:** `chdir` follows symlinks, landing in the target. Use `readlink` or `Cwd::abs_path` first if you need to know the real destination.
- **Deleted directory:** if the directory was removed after you entered it, relative operations from inside fail with ENOENT and most shells show the path as (deleted). You can still `chdir` out with an absolute path.

- **Handle form on systems without `fchdir(2)`:** raises an exception. Linux always supports it; the restriction is a portability concern for code that also runs on older systems.
- **Untrusted input:** passing a user-supplied path to `chdir` without validation lets the caller redirect all subsequent relative-path operations in the program. Canonicalise with `Cwd: :abs_path` and check against an allow-list before the call if the input crosses a trust boundary.
- **After `fork`:** the child inherits the parent's cwd. A `chdir` in the child does not affect the parent.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `mkdir` - create a directory; pair with `chdir` when building a workspace and stepping into it
- `rmdir` - remove a directory; do not `rmdir` the cwd, `chdir` out first
- `opendir` - open a directory handle usable as the argument to `chdir` for race-free directory changes
- `chroot` - stronger confinement than `chdir`; changes the filesystem root for the process, not just the cwd
- `$ENV{HOME}` - consulted by the no-argument form of `chdir` before falling back to `$ENV{LOGDIR}`
- `$!` - carries the `errno` when `chdir` returns 0

Filehandles, files, directories

chmod

Change the permission bits of a list of files.

`chmod` sets each file in `LIST` to the permission `MODE`. `MODE` is a **numeric** value - the low twelve bits are the usual Unix permission triad plus the `setuid`, `setgid`, and sticky bits, exactly as accepted by `chmod(2)`. Both plain pathnames and open filehandles are accepted in `LIST`. Returns the number of files for which the underlying system call succeeded.

Synopsis

```
chmod MODE, LIST
```

What you get back

The number of files successfully changed, as an integer. Zero means every call failed; a value less than scalar `@files` means a partial failure. `chmod` does **not** throw on a per-file failure - it keeps going and tallies the successes.

`$!` reflects the error from the **last** failing file only. On a mixed success/failure run there is no built-in way to learn which files failed; if you need per-file error reporting, call `chmod` once per file and check the return value each time:

```
for my $f (@files) {
    chmod 0644, $f
    or warn "chmod $f: $!";
}
```

Writing MODE correctly

MODE is an integer. The three ways to spell it:

- **Octal literal with leading zero:** `0644`, `0755`, `0600`. This is the conventional form and matches the digits you would type into `chmod(1)`.
- **Octal literal with `0o` prefix:** `0o644`, `0o755`. Accepted since Perl 5.34; equivalent to the leading-zero form and easier to read as “octal” at a glance.
- **Symbolic constants from `Fcntl`:** use `Fcntl qw(:mode); S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH` evaluates to `0755`. Useful when a permission set is built up from pieces.

A string argument is almost always a bug:

```
chmod "0644", "foo";      # WRONG - stringifies as decimal 644 (octal
    ↪1204),
                           #           grants mode --w---r-T
chmod 644, "foo";         # WRONG - decimal 644 is octal 1204, same
    ↪result
chmod 0644, "foo";        # right
chmod 0o644, "foo";       # right (Perl 5.34+)
```

If the mode genuinely arrives as a string (configuration file, CLI argument), convert it with `oct` first:

```
my $mode = "0644";        # string from config
chmod oct($mode), "foo";  # oct("0644") == 0644
```

Global state it touches

Sets `$!` on any failure to match the last failing `chmod(2)` / `fchmod(2)` call. Reads no interpreter globals.

Examples

Change one file, check for success:

```
chmod 0755, "install.sh"
    or die "chmod install.sh: $!";
```

Change many files in a single call, capture the count:

```
my $cnt = chmod 0644, "a.txt", "b.txt", "c.txt";
print "updated $cnt of 3 files\n";
```

With an array of pathnames:

```
my @executables = glob "bin/*";
chmod 0755, @executables;
```

Preserve existing bits, add owner write. Reads the mode with `stat`, masks to the permission bits, ORs in the new bit, and writes it back:

```
open my $fh, "<", "foo" or die $!;
my $perm = (stat $fh)[2] & 07777;
chmod $perm | 0200, $fh;
```

Symbolic form via `Fcntl`:

```
use Fcntl qw(:mode);
chmod S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH, @executables;
# identical to: chmod 0755, @executables;
```

Per-file error reporting (needed to learn *which* file failed on a multi-file call):

```
my @failed;
for my $f (@files) {
    push @failed, $f unless chmod 0644, $f;
}
warn "chmod failed on: @failed\n" if @failed;
```

Edge cases

- **Empty list:** `chmod 0644` returns 0. The mode is evaluated but no system call is made.
- **String modes:** `chmod "0644", $f` and `chmod "+x", $f` are both wrong. `chmod` never parses symbolic mode strings - unlike `chmod(1)`, the built-in takes a number, not a specification.
- **Filehandles vs barewords:** a bareword in `LIST` is a filename, not a filehandle. Pass a filehandle as a glob or glob reference:

```
chmod 0644, *FH;           # filehandle
chmod 0644, \*FH;          # filehandle
chmod 0644, $fh;           # filehandle (lexical)
chmod 0644, "FH";          # filename "FH" - almost certainly a
↪ bug
```

On systems without `fchmod(2)`, passing a filehandle raises an exception. Linux supports `fchmod(2)`, so on the platforms pperl targets this always works.

- **Setuid / setgid / sticky bits:** the high three bits of the mode word (04000, 02000, 01000) set setuid, setgid, and the sticky bit respectively. 0o4755 makes a setuid executable; 0o2775 a setgid directory.

- **Symlinks:** `chmod` follows symlinks. Changing permissions on the symlink itself is not portable and is not exposed by this built-in
 - use `lchmod` from a module if the platform supports it.
- **Permission to change permission:** only the file's owner (and root) may call `chmod(2)` on a file. Failure sets `$!` to `EPERM`.
- **Non-existent paths:** `chmod` on a missing path sets `$!` to `ENOENT` and contributes 0 to the success count. It does not warn and does not die.
- **umask does not apply:** `umask` masks the mode of *newly created* files from `open`, `sysopen`, `mkdir`, and similar. `chmod` sets the mode literally - no masking.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `chown` - the sibling call for changing owner/group; same list-and-return-count shape
- `stat` - read the current mode so you can modify it rather than replace it wholesale
- `umask` - default permission mask applied to *new* files; unrelated to `chmod`, which sets bits literally
- `open` / `sysopen` - create files; the mode argument to `sysopen` is masked by `umask` at creation time and can then be tightened or relaxed with `chmod`
- `oct` - parse "0644" into the integer 0644 when the mode arrives as a string
- `$!` - system error from the last failing file; only meaningful for single-file calls or immediately after a failing per-file call in a loop

SCALARs and strings

chomp

Strip the trailing input record separator from a string in place.

`chomp` removes any trailing substring that matches the current value of `$/` from its argument and returns the total number of characters removed across every argument it touched. It mutates its argument; the return value is a count, not the chomped string. If no argument is given, it operates on `$_`. `chomp` is the safer counterpart to `chop`: it only removes the separator if it is actually there, and it respects `$/` so it does the right thing on files that end without a final newline or that use a non-newline record terminator.

Synopsis

```
chomp VARIABLE
chomp( LIST )
chomp
```

What you get back

The total number of characters removed, as an integer. Usually 0 (no separator was present) or 1 (a single "\n" was removed under the default \$/). When \$/ is a multi-character string, a successful chomp returns its length. When chomping a list, the count is summed across every element.

The chomped value lives in the variable you passed; chomp is called for effect, not for its return value:

```
my $line = "hello\n";
my $n = chomp $line;           # $line is "hello", $n is 1
```

Global state it touches

- **\$/ - input record separator.** chomp reads \$/ on every call and strips a trailing occurrence of its current value. Changing \$/ changes what chomp removes; this is usually what you want when you set \$/ to read custom-delimited records.
- **\$_** - consumed as the default argument when chomp is called with no argument. chomp; inside while (<>) { ... } is the canonical idiom.
- Hash iterator - chomping a hash resets its [each](#) iterator (see *Edge cases*).

Examples

The everyday use - strip the trailing newline from input:

```
while (my $line = <$fh>) {
    chomp $line;
    # process $line without the trailing "\n"
}
```

The no-argument form operates on \$_:

```
while (<>) {
    chomp;           # removes "\n" from $_
    my @fields = split /:/;
}
```

Chomp an lvalue assignment - parentheses are **required** so the argument to chomp is the whole assignment, not just the variable:

```
chomp(my $cwd = `pwd`);           # strip "\n" from command output
chomp(my $answer = <STDIN>);      # strip "\n" from user input
```

Chomping a list - every element is chomped and the total count is returned:

```
my @lines = ("a\n", "b\n", "no-nl");
my $n = chomp @lines;           # @lines = ("a", "b", "no-nl"),
↪ $n = 2
```

Custom record separator - chomp follows whatever \$/ is set to:

```
local $/ = "\r\n";
my $line = "GET / HTTP/1.0\r\n";
chomp $line;                                # removes "\r\n", returns 2
```

Paragraph mode - `$/ = ""` makes `chomp` remove **all** trailing newlines from the string:

```
local $/ = "";
my $para = "line one\nline two\n\n\n";
chomp $para;                                # $para is "line one\nline two"
```

Edge cases

- **No separator present:** a no-op. Returns `0` and leaves the argument unchanged. This is the whole point of `chomp` over `chop`
 - safe to call on strings that may or may not end in `$/`.
- **undef argument:** chomping an undefined scalar leaves it undefined and returns `0`. Under `use warnings` this emits a uninitialized warning.
- **Hash argument:** `chomp %h` chomps every value (not the keys) and **resets the hash's each iterator**. A `while (my ($k, $v) = each %h)` loop in progress over `%h` will be disrupted.
- **Slurp mode (`$/ = undef`) and fixed-length record mode (`$/` set to a reference to an integer):** `chomp` removes nothing and returns `0`. There is no notion of a trailing separator in these modes.
- **Parenthesisation trap on assignment:** without parens, `chomp` binds tighter than `=`:

```
chomp $cwd = `pwd`;                        # parsed as (chomp $cwd) = `
↪ `pwd`;
chomp( $cwd = `pwd` );                    # what you meant
```

Same trap with a bare comma list:

```
chomp $x, $y;                             # parsed as chomp($x), $y;
chomp( $x, $y );                          # chomps both
```

- **Any lvalue works**, not just plain scalars. Array and hash elements, slices, dereferences, and assignment expressions are all legal targets.
- **Constants and read-only values:** chomping a read-only string croaks with `Modification of a read-only value attempted`. String literals are read-only.
- **Multi-byte `$/`:** `chomp` compares bytes of the string against bytes of `$/`. If the string and `$/` disagree on encoding (one has the UTF-8 flag set, the other does not), the match is done on the internal byte representation and may surprise you. Keep both sides in the same encoding.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `chop` - unconditionally remove the **last character** and return it; older, blunter tool superseded by `chomp` for input cleanup
- `split` - often the next step after `chomp`, splits the now-trimmed record into fields
- `readline` / `<>` - reads records terminated by `$/`; `chomp` undoes the terminator that `readline` left in place
- `each` - the hash iterator that `chomp %hash` resets
- `$/` - input record separator; `chomp` reads this on every call
- `$_` - default argument when `chomp` is called with no explicit target

SCALARS and strings

chop

Remove the last character from a string and return it.

`chop` shortens its argument in place by exactly one character and returns the character that was removed. It does not scan the string and does not copy it - it simply moves the end. If `VARIABLE` is omitted, `chop` operates on `$_`. If `VARIABLE` is a hash, `chop` chops every value of the hash (not the keys) and resets the hash's `each` iterator as a side effect. Any lvalue is a valid target, including an assignment expression.

Synopsis

```
chop VARIABLE
chop( LIST )
chop
```

What you get back

The character that was removed, as a string. If the target was empty or undefined, the empty string is returned and the target is left unchanged. When `chop` is applied to a list, every element is chopped but only the character removed from the **last** element is returned.

```
my $last = chop $line;      # one character, or "" if $line was empty
```

Global state it touches

Reads `$_` when called with no argument. No other special variables are consulted - in particular, `chop` is **not** affected by `$/` (the input record separator), which is `chomp`'s concern.

Examples

Chop a scalar:

```
my $s = "hello";
my $c = chop $s;           # $s is now "hell", $c is "o"
```

Chop `$_` when no argument is given - idiomatic inside a read loop that wants the terminator gone regardless of what it is:

```
while (<$fh>) {
    chop;                 # drops whatever character ends the line
    push @rows, $_;
}
```

Chop a list. Every element loses its last character; the return value is the character removed from the **last** element only:

```
my @lines = ("foo\n", "bar\n", "baz!");
my $last = chop @lines;    # @lines is ("foo","bar","baz"), $last
                             ↪is "!"
```

Chop a hash. Values are chopped, keys are not, and the `each` iterator is reset:

```
my %h = (a => "one\n", b => "two\n");
chop %h;           # %h is (a => "one", b => "two")
```

Chop an assignment - chop works on any lvalue, and an assignment returns one:

```
chop(my $cwd = `pwd`);    # strip the trailing newline from pwd's
                             ↪output
```

Keep all but the last character without modifying the original - not a job for chop, which mutates. Use `substr` with a negative length instead:

```
my $head = substr($s, 0, -1); # $s is unchanged
```

Edge cases

- **Empty or undefined target:** returns the empty string "" and leaves the target alone. chop on `undef` does *not* raise a warning by itself; it simply produces "".
- **Read-only target:** raises Modification of a read-only value attempted. This includes string literals and values flagged read-only via `Internals::SvREAD-ONLY`.
- **UTF-8 strings:** chop removes one **character**, not one byte. A multi-byte character at the end of the string is removed whole and returned as a UTF-8 string. The target's UTF-8 flag is preserved on what remains.
- **List return value:** only the character chopped from the last element is returned; the others are discarded. If you need every removed character, chop the elements individually in a loop.

- **One character vs. a line ending:** `chop` unconditionally removes **one** character, whatever it is - including spaces, digits, or letters. To conditionally remove a line terminator (what `$/` says one looks like), use `chomp`.
- **Parser parenthesisation:** `chop($x = $y)` chops the result of the assignment, which is `$x`. `chop $x, $y` is parsed as `chop($x), $y` - the comma does not extend `chop`'s argument list without parentheses. Write `chop($x, $y)` when you mean the list form.
- **Hash iteration reset:** because `chop %h` walks the values via the hash iterator, any in-progress `each` loop over `%h` is invalidated.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `chomp` - conditional cousin: removes a trailing `$/`, not an unconditional last character; returns the number of characters removed rather than the character itself
- `substr` - non-destructive alternative; `substr($s, 0, -1)` returns all but the last character without mutating `$s`
- `s///` - `$s =~ s/.\z//s` is the regex equivalent and the one `chop` is explicitly faster than, because `chop` neither scans nor copies
- `lc`, `uc` - other in-place(ish) string transforms that accept any lvalue
- `$_` - the implicit target when `chop` is called with no argument
- `$/` - **not** consulted by `chop`; documented here only to flag the contrast with `chomp`

Filehandles, files, directories

chown

Change the owner and group of a list of files.

`chown` takes a numeric user id, a numeric group id, and a list of files, and reassigns ownership on every file that the calling process is allowed to touch. The first two elements of `LIST` are the new UID and GID, in that order; every element after that is a filename (or, on systems that support `fchown(2)`, a filehandle). A value of `-1` in either of the id slots means “leave that id unchanged” - useful when you want to change only the group, or only the owner.

Synopsis

```
chown UID, GID, LIST
chown $uid, $gid, $file
chown $uid, $gid, @files
chown -1, $gid, @files           # change group only
```


What you get back

The number of files whose ownership was successfully changed, as a plain integer. Not a boolean - a call over ten files that succeeds on seven returns 7. Check against the list length when you need all-or-nothing semantics:

```
my @files = ('a', 'b', 'c');
my $ok = chown $uid, $gid, @files;
$ok == @files
    or die "chown failed on ", scalar(@files) - $ok, " files: $!";
```

When any individual file fails, `$!` holds the `errno` from the **last** failing call. A return value less than `scalar @files` is the reliable “something went wrong” signal; `$!` tells you what the last failure was, not every failure.

Global state it touches

- `$!` - set to the `errno` of the last underlying `chown(2)` call that failed. Unchanged when every file succeeds.

Numeric ids only

`chown` does **not** accept user or group names. Pass the string “root” where a uid is expected and you get a numeric conversion of 0 (with an Argument ... isn't numeric warning under `use warnings`) - almost never what you want. Translate names to numbers with `getpwnam` and `getgrnam`:

```
my $uid = getpwnam('alice') // die "no such user";
my $gid = getgrnam('staff') // die "no such group";
chown $uid, $gid, @files;
```

The list-context return of `getpwnam` gives the full `passwd` entry; in scalar context it gives just the uid, which is usually what `chown` wants.

Permission rules

On Linux and every other POSIX system, only the superuser can give a file away to a different owner. An unprivileged user can change the **group** of a file they own, but only to one of their own secondary groups. Trying to do more returns failure with `$!` set to `EPERM`.

```
chown 0, 0, 'passwd.new'
    or warn "need root to chown: $!";
```

The restriction is kernel-enforced, not Perl-enforced - there is no pre-check that would save you a failed system call. Rely on the return value.

Examples

Change both owner and group on a single file:

```
chown $uid, $gid, '/var/log/app.log'
    or die "chown failed: $!";
```

Change the group only, leaving the owner alone:

```
chown -1, $gid, @files;
```

Translate names to numeric ids before calling:

```
my ($login, $pass, $uid, $gid) = getpwnam('www-data')
    or die "www-data not in passwd file";
chown $uid, $gid, glob('/var/www/*');
```

Count how many files were actually reassigned:

```
my @targets = glob('/srv/data/*.dat');
my $changed = chown $uid, $gid, @targets;
warn "only $changed of ", scalar(@targets), " files changed: $!"
    if $changed != @targets;
```

Chown by filehandle on a system that supports `fchown(2)`. The filehandle must be a glob or a glob reference - a bareword is treated as a filename:

```
open my $fh, '>', '/tmp/out' or die $!;
chown $uid, $gid, $fh;
```

Edge cases

- **Empty file list:** `chown $uid, $gid;` - no files to act on, returns 0, does not touch \$!.
- **-1 in id position:** standard POSIX sentinel meaning “leave this id unchanged.” Works on every supported platform.
- **String id:** a non-numeric string silently becomes 0 (root / wheel) after the usual numeric conversion. Under use warnings you get an `Argument isn't numeric` warning, but there is no hard error. Always pass an integer.
- **Non-existent file:** counted as a failure, does not abort the call. Remaining files still get processed.
- **Symlink target:** `chown` follows symlinks and changes the pointed-to file. To `chown` the link itself, use `lchown(2)` via `syscall` or a module that exposes it - core Perl does not.
- **Filehandle form:** on systems without `fchown(2)` (rare in practice on Linux), passing a filehandle raises an exception. Barewords in the file list are always treated as filenames - use `\*FH` or a `my $fh` to mean “this handle.”
- **Give-away restriction:** on POSIX systems with `_PC_CHOWN_RESTRICTED` set for the target path, only root can change the owner at all. Probe with:

```
use POSIX qw(pathconf _PC_CHOWN_RESTRICTED);
my $restricted = pathconf($path, _PC_CHOWN_RESTRICTED);
```

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `chmod` - change file permission bits; takes the same “mode, then file list” shape and returns the same number-of-files-changed count
- `stat` - read the current owner, group, and mode of a file before deciding what to change
- `umask` - set the default permission mask for files created by this process; complements `chown` for newly created files
- `getpwnam` - translate a user name to the numeric uid `chown` requires
- `getgrnam` - translate a group name to the numeric gid `chown` requires
- `$!` - holds the `errno` of the last failing per-file `chown(2)` when the return count is below the list length

SCALARs and strings

chr

Return the character whose codepoint is the given number.

`chr` takes a non-negative integer and returns a one-character string holding the character at that codepoint in the current character set. `chr(65)` is "A" in both ASCII and Unicode; `chr(0x263a)` is the Unicode smiley face ☺. If `NUMBER` is omitted, `chr` operates on `$_`. The inverse operation is `ord`.

Synopsis

```
chr NUMBER
chr
chr($n)
```

What you get back

A string of length one character. With Unicode semantics - the default for any codepoint above `0x7f` - that character may be encoded internally as several bytes of UTF-8, so `length` reports 1 while `bytes::length` can report up to four. For codepoints in the range `0..127` the result is a single-byte ASCII string and the two lengths agree.

Global state it touches

With no argument, `chr` reads `$_`. It writes nothing. The `bytes` pragma changes what `chr` does with out-of-range or negative input (see *Edge cases*); no other pragma or special variable affects it.

Examples

ASCII character from its codepoint:

```
print chr(65);           # A
print chr(97), "\n";     # a
```

Build a string from a list of codepoints with `map` and `join`:

```
my @code = (72, 101, 108, 108, 111);
print join("", map { chr } @code), "\n";  # Hello
```

Unicode codepoint above the ASCII range - the result is one character, four bytes in its internal UTF-8 form:

```
use utf8;
my $smiley = chr(0x263a);
print length($smiley);           # 1
use bytes; print length($smiley); # 3
```

Round-trip with `ord` - for any single character `$c`, the identity `chr(ord($c)) eq $c` holds:

```
my $c = "Z";
print chr(ord($c)) eq $c ? "yes" : "no";  # yes
```

Default-argument form inside a loop over `$_`:

```
for (65, 66, 67) {
    print chr;           # prints ABC
}
```

Negative input yields the Unicode replacement character U+FFFD:

```
print chr(-1) eq "\{fffd}" ? "repl" : "?"; # repl
```

Edge cases

- **Fractional argument:** truncated toward zero before lookup. `chr(65.9)` returns "A".
- **Non-numeric string:** converted by the normal numeric-coercion rules, with a warning under `use warnings`. `chr("65abc")` returns "A".
- **`chr(undef)`** returns `chr(0)`, the NUL character `"\0"`, and warns under `use warnings` about use of an uninitialized value.

- **Negative values:** outside the bytes pragma, negative input returns the Unicode replacement character "\{fffd}". Under use bytes, the low eight bits of the integer value are used: `chr(-1)` yields the byte "\xff".
- **Values above 0x10ffff** (the maximum Unicode codepoint) still return a single-character string under default semantics, but emit a Unicode non-character is illegal for interchange warning for the non-character ranges, and a Unicode surrogate U+... warning for the surrogate range 0xd800..0xdfff when the string is later encoded.
- **The 128..255 range:** characters in this range are by default **not** stored as UTF-8 internally, for backward compatibility with byte strings. They compare equal to their single-byte form and to the same Unicode codepoint; the internal representation only matters when inspecting with `bytes::length` or interacting with PerlIO layers.
- **chr never croaks** on any numeric input. Errors surface only as warnings, never exceptions.
- **Not an lvalue:** `chr` returns a fresh scalar. `chr(65) = "B"` is a syntax error.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `ord` - inverse of `chr`: codepoint of the first character of a string
- `sprintf` - `sprintf "%c", $n` is equivalent to `chr $n`; use it to splice a character into a larger format at once
- `pack` - build a multi-character string from a list of codepoints in one call (`pack "U*", @code` for Unicode, `pack "C*", @code` for bytes)
- `hex` - parse a hex string into the integer you then pass to `chr`; pairs with `chr` when decoding U+XXXX notation
- `use bytes` - pragma that switches `chr` to byte-oriented semantics for values outside 0..0x7f
- `perlunicode` - background on how Perl stores and compares Unicode strings

[Filehandles](#), [files](#), [directories](#)

chroot

Change the root directory of the current process and every child it later spawns.

After a successful `chroot`, any pathname that begins with `/` is resolved relative to the directory named by `FILENAME` instead of the real filesystem root. The process loses the ability to name files outside that subtree through absolute paths. The kernel enforces this for the current process and for every descendant - there is no per-thread scope and no way to undo it from inside the jail.

`chroot` does **not** change the current working directory. If the working directory was outside the new root when the call succeeded, it stays outside, and relative paths can still

escape. Always follow a successful `chroot` with `chdir` to `/` (which now means inside the jail) before doing anything else.

The call is restricted to the superuser. An unprivileged process gets `EPERM` and `$!` is set accordingly.

Synopsis

```
chroot FILENAME
chroot
```

What you get back

1 on success, 0 on failure. On failure `$!` holds the `errno` from the underlying `chroot(2)` system call - commonly `EPERM` (not root), `ENOENT` (path does not exist), or `ENOTDIR` (path is not a directory).

Always check the return value. A silent failure leaves the process running outside the intended jail, which is usually worse than the code path that made the call in the first place.

```
chroot "/var/empty"
    or die "chroot: $!";
chdir "/"
    or die "chdir after chroot: $!";
```

Global state it touches

- `$_` - when called with no argument, `chroot` uses the value of `$_` as the path. Inside a `while (<>)` loop this is rarely what you want; pass the path explicitly.
- `$!` - set to the underlying `errno` on failure.

Examples

The canonical safe sequence - change root, then change working directory to the new root, then drop privileges:

```
chroot "/srv/jail"    or die "chroot: $!";
chdir "/"             or die "chdir: $!";
$) = $(= 65534;      # drop to 'nobody' gids
$> = $< = 65534;     # drop to 'nobody' uids
```

Open files and load shared libraries **before** `chroot`, not after. The new root typically has neither `/etc/resolv.conf` nor the dynamic linker's search paths:

```
open my $log, ">>", "/var/log/app.log" or die $!;
my $cfg = read_config("/etc/app.conf");

chroot "/srv/jail" or die "chroot: $!";
chdir "/"         or die "chdir: $!";
```

(continues on next page)

(continued from previous page)

```
# $log and $cfg are still usable inside the jail.
```

Checking failure modes separately - distinguishing “not root” from “bad path” matters for diagnostics:

```
use Errno qw(EPERM ENOENT ENOTDIR);

unless (chroot $path) {
    die "chroot: need root privileges\n"      if $! == EPERM;
    die "chroot: $path does not exist\n"      if $! == ENOENT;
    die "chroot: $path is not a directory\n"  if $! == ENOTDIR;
    die "chroot: $!";
}
```

Default-argument form reads the path from \$_:

```
$_ = "/srv/jail";
chroot or die "chroot: $!";
```

Edge cases

- **Not root:** fails with EPERM. There is no capability-based relaxation in stock Linux unless the process holds CAP_SYS_CHROOT.
- **Relative path:** chroot "jail" resolves against the current working directory at the moment of the call. Prefer absolute paths so the behavior does not depend on the caller's cwd.
- **Symlinks in FILENAME:** resolved before the chroot takes effect. Symlinks *inside* the new root that point to absolute paths will resolve relative to the new root, not the real root - this is the intended jailing behavior, not a bug.
- **Open file descriptors survive:** any handle opened before the call remains usable afterward. This is how well-behaved daemons keep logs, config, and listening sockets across the transition.
- **No escape from inside:** a process cannot chroot its way out. The only way to leave a chroot is for a privileged process *outside* it to reap and replace it.
- **Working directory outside the new root:** the kernel allows the call, but relative paths and `..` traversal from there can still reach files outside the jail. The mandatory chdir "/" after chroot closes that gap.
- **Threads:** chroot on Linux is per-process (strictly, per fs struct). Under use threads all interpreter threads share the same root after the call - there is no per-thread jail.
- **Does not affect \$0, %ENV, or open sockets.** Only path resolution changes.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `chdir` - the mandatory follow-up after a successful `chroot`; without it the working directory may still point outside the new root
- `fork` - children inherit the `chroot`; combine with `chroot` to run untrusted code in a restricted subtree
- `exec` - runs the replacement program inside the current root, so the target binary and its dynamic libraries must exist inside the jail
- `open` - opening files by absolute path after `chroot` looks up names inside the jail; pre-open anything you need from outside before the call
- `$_` - the default path when `chroot` is called with no argument
- `$!` - holds the `errno` from `chroot(2)` on failure

Scoping · Classes and OO

class

Declare a namespace that behaves as a native object class.

`class` introduces a new package just like `package`, but marks it as a *class*: the runtime attaches a constructor named `new`, enables the `field` and `method` keywords inside its body, and supports single inheritance through the `:isa` attribute. It is the core-language answer to hand-rolled `bless`-based OO - the same problem, without the boilerplate.

Note

Experimental The `class` feature (codename *Corinna*) is experimental in Perl 5.42 and may still change in incompatible ways. Enable it with `use feature 'class'` and silence the notice with `no warnings 'experimental::class'`.

Synopsis

```

class NAMESPACE;                                # rest-of-scope form
class NAMESPACE VERSION;
class NAMESPACE :ATTRIBUTES;
class NAMESPACE BLOCK                            # block form
class NAMESPACE VERSION BLOCK
class NAMESPACE VERSION :ATTRIBUTES BLOCK

```

Minimal working class:

```

use v5.38;
use feature 'class';
no warnings 'experimental::class';

```

(continues on next page)

(continued from previous page)

```

class Point {
    field $x :param;
    field $y :param;

    method distance_from_origin {
        return sqrt($x ** 2 + $y ** 2);
    }
}

my $p = Point->new(x => 3, y => 4);
say $p->distance_from_origin;           # 5

```

What you get back

class is a declaration, not an expression - it does not return a value you can assign. What you get is:

- A new package in the symbol table, reachable as any other package.
- A generated new constructor. **Do not write your own new.** The generated one honours :param field attributes and runs every ADJUST block in declaration order.
- \$self implicitly bound inside every method body.
- __CLASS__ usable inside field initialisers and methods to name the class currently being constructed (which may be a subclass).

Forms

class accepts four syntactic shapes. The split between **block** and **statement** forms mirrors package exactly:

- **class NAME BLOCK** - the class definition is the contents of the block. Scope ends with the closing brace.
- **class NAME;** - the rest of the current scope (or file) up to the next class or package statement is the class body. Useful for single-class files.
- **class NAME VERSION ...** - the version is declared like package Foo 1.234. The version must precede any attributes.
- **class NAME :ATTRIBUTES ...** - one or more class attributes, separated by whitespace, each starting with ::. The only currently defined class attribute is :isa.

```

class My::Thing 1.02 :isa(My::Base) {
    # body
}

```

Fields, methods, ADJUST

Inside a class body three new keywords are in scope:

- `field` declares per-instance storage. Fields behave like lexicals inside methods but each instance has its own slot. Attributes `:param`, `:reader`, and `:writer` generate constructor binding and accessors.
- `method` declares a subroutine that auto-binds `$self` and that can read and write the enclosing fields by name. Methods act as if use feature 'signatures' were in effect; `$self` does not appear in the signature.
- `ADJUST BLOCK` runs after field initialisers, during construction, with `$self` already bound. Use it for post-construction validation and for derived fields.

```
class Counter {  
  field $value :param = 0;  
  field $step  :param = 1;  
  
  ADJUST {  
    die "step must be positive" if $step <= 0;  
  }  
  
  method tick { $value += $step }  
  method peek { $value }  
}
```

Fields are **always lexical to the class** - they are not inherited. Subclasses that need access to a parent's state do so through methods, not through fields.

Inheritance: `:isa`

A class inherits from at most one other class:

```
class Shape { ... }  
  
class Circle :isa(Shape) {  
  field $r :param;  
  method area { 3.14159265 * $r * $r }  
}
```

Only single inheritance is supported. The `:isa` target is loaded automatically if it is not already in memory - equivalent to use `MODULE ( )` before the class declaration. A minimum version can be required the same way use `MODULE VERSION` does:

```
class Circle :isa(Shape 2.000) { ... }
```

`:isa` only inherits **methods**. Fields are per-class private storage; a subclass has no implicit view into a parent's fields.

Global state it touches

- **%INC** - :isa(Parent) loads Parent via require on first use and records it here, just like `use`.
- **@ISA** - populated for the class from the :isa attribute, so existing isa / UNIVERSAL / SUPER :: introspection keeps working.
- **The symbol table** - the class name becomes a stash in %:: exactly as `package` would create one.

Examples

Plain class with a required parameter and a derived field:

```
class User {
  field $name :param;
  field $greeting;

  ADJUST { $greeting = "Hello, $name" }

  method greet { say $greeting }
}

User->new(name => 'Ada')->greet;           # Hello, Ada
```

Default parameter values - = applies when the caller omitted the parameter; //= also applies when the value was undef; ||= also applies when the value was false:

```
class Window {
  field $title :param           = 'untitled';
  field $width :param          //= 640;
  field $shown :param          ||= 1;
}
```

Single inheritance with a method override:

```
class Animal {
  field $name :param;
  method speak { "some sound" }
  method introduce { say $name, " says ", $self->speak }
}

class Dog :isa(Animal) {
  method speak { "woof" }
}

Dog->new(name => 'Rex')->introduce;       # Rex says woof
```

Private method via lexical my method, invoked through ->&:

```
class Safe {
  field $secret :param;

  my method check ($n) { $n == $secret }

  method unlock ($n) {
    return $self->&check($n) ? "open" : "denied";
  }
}
```

Using `__CLASS__` so a base-class field initialiser picks up the subclass's override:

```
class Base {
  sub DEFAULT_X { 10 }
  field $x = __CLASS__->DEFAULT_X;
  method x { $x }
}

class Tuned :isa(Base) {
  sub DEFAULT_X { 99 }
}

say Tuned->new->x; # 99
```

Rest-of-file statement form:

```
use v5.38;
use feature 'class';
class Config;

field $path :param;
method path { $path }
# File ends here; whole remainder is Config's body.
```

Edge cases

- **Do not define new.** The generated constructor is the only supported entry point. A user-written `sub new { ... }` inside a class body is a compile-time error.
- **No multiple inheritance.** `:isa(A, B)` is a syntax error. Multiple `:isa` attributes on one class are also rejected.
- **No @ISA manipulation.** Treat `@ISA` as read-only for classes declared with `class`. Pushing onto it at runtime is undefined.
- **package cannot re-open a class.** Once a namespace has been declared with `class`, subsequent `package` statements targeting the same name are a compile-time error, and vice versa.
- **Fields are not lexicals** in the usual sense. You cannot refer to a field from outside the class body, and you cannot `our` or `local` it. Only methods and `ADJUST` blocks see fields.

- **\$self** is read-only inside a method. Assigning to it is a runtime error.
- **Field initialisers run before \$self exists.** Inside field `$x = EXPR`, you can use `__CLASS__` and previously declared fields, but not `$self`. Anything that needs the fully built instance belongs in an ADJUST block.
- **No field access before declaration.** Within a field's initialiser expression, only fields declared *earlier* in the class body are visible.
- **Method resolution.** Dispatch still goes through the usual MRO (`UNIVERSAL::isa`, `SUPER::`), so `$self->SUPER::foo(@args)` works the way it does for `bless`-based classes.
- **Statement form scope.** `class Foo;` consumes the rest of the enclosing block - typically the rest of the file. A second `class` or `package` terminates it; there is no way to “close” a statement-form class early.
- **Experimental warning.** Every class declaration emits a class is experimental warning under the default warnings. Use `no warnings 'experimental::class'` once the feature is in use for real.

Differences from upstream

Fully compatible with upstream Perl 5.42. The feature is flagged experimental in upstream; `pperl` mirrors that status and emits the same warning category (`experimental::class`).

See also

- `field` - declares per-instance storage within a class body; supports `:param`, `:reader`, `:writer`
- `method` - declares a subroutine with implicit `$self` and direct access to the enclosing class's fields
- `package` - the non-class namespace declaration; `class` is the OO-aware sibling with the same syntactic forms
- `bless` - the pre-5.38 way to attach a class to a reference; still supported, and inter-operates with `class`-built objects for `isa/can` checks
- `isa` - operator for testing class membership; works for both `class`-declared and `bless`-based objects
- `use` - how `:isa(Parent)` loads the parent class under the hood

I/O

close

Close a filehandle, flush its buffers, and release the underlying file descriptor.

`close` finishes whatever I/O was pending on `FILEHANDLE`, tears down the PerlIO layer stack, and asks the kernel to close the descriptor. The return value distinguishes a clean shutdown from one where either a buffered write failed on the way out, a PerlIO layer reported an error, or - for pipes - the child process exited non-zero. Without an argument, `close` operates on the currently selected filehandle (see `select`).

Synopsis

```
close FILEHANDLE
close EXPR
close
```

What you get back

Truthy (1) on a clean shutdown; false otherwise. A false return means one of the following went wrong:

- A pending buffered write could not be flushed (disk full, broken pipe, closed socket).
- A PerlIO layer (e.g. `:encoding(...)`, `:gzip` from a CPAN layer) reported an error.
- A pre-existing sticky error was already recorded on the handle.
- The handle came from a piped `open` and the child exited non-zero (or another syscall involved in tearing the pipe down failed).

When `close` returns false, `$!` holds the error from the failing operation and is safe to read. For pipe handles, if the only problem was a non-zero child exit, `$!` is set to `0` and the child's status shows up in `$?` (see below).

Always check the return value on writable handles: the final buffered block is flushed inside `close`, so `print` returning true earlier does **not** guarantee the data reached disk.

```
close $fh
or die "close $path failed: $!";
```

Global state it touches

- `$!` - set to the failing `errno` when `close` returns false, including the flush error on the final buffered write. Cleared to `0` on a pipe close where the only problem was a non-zero child exit.
- `$?` - for handles from a piped `open`, set to the wait status of the child process: low 8 bits are the signal, high 8 bits are the exit code (same encoding as `system`).
- `$_{^CHILD_ERROR_NATIVE}` - for pipe handles, set to the raw OS-native status value (before Perl's portable encoding).
- `$.` - an **explicit** close on an input handle resets the current line number. An implicit close via reopening the same handle does not.
- Currently selected filehandle (via `select`) - close with no argument closes it.

Examples

Close a regular file and check for a flush failure:

```
open my $fh, ">", "out.txt" or die $!;
print {$fh} @records;
close $fh or die "flush to out.txt failed: $!";
```

Close a pipe opened for writing and inspect the child's exit status. `close` waits for the child, then populates `$?`:

```
open(my $sort, "|-", "sort", ">", "foo")
    or die "can't start sort: $!";
# ... print records to $sort ...
close $sort
    or warn $! ? "error closing sort pipe: $!"
               : "sort exited with status " . ($? >> 8);
```

Close a pipe opened for reading, then collect the command's exit status. Exit code is the top 8 bits of `$?`:

```
open(my $who, "-|", "who") or die $!;
my @lines = <$who>;
close $who;
my $exit = $? >> 8;
```

Bare `close` closes whatever handle `select` most recently made current, then restores STDOUT as the selected handle:

```
open my $log, ">>", "app.log" or die $!;
my $old = select $log;
print "entry\n";
close;                                # closes $log (the selected handle)
select $old;
```

Closing the read end of a pipe early makes the writer receive SIGPIPE. If the writer has no handler, it will be killed - drain the pipe first if you need the writer to exit cleanly:

```
open my $gen, "-|", "some-producer" or die $!;
while (<$gen>) { last if /^END/ }
1 while <$gen>;                      # drain so the child finishes cleanly
close $gen or warn "producer failed: $!";
```

Edge cases

- **Bare close** - with no argument, closes the currently selected filehandle. In a freshly started program that is STDOUT; after a `select($fh)` it is `$fh`. Rarely what you want outside tightly scoped code.
- **Reopening closes for you** - `open my $fh, ...` on a handle that is already open will close the old descriptor first. The implicit close does **not** reset `$.`; an explicit close does. Use an explicit close when line-number state matters.
- **Scope exit closes lexical handles** - when the last reference to a lexical filehandle goes out of scope, Perl closes it. If that implicit close fails (flush error, child exit non-zero), Perl emits a Warning: unable to close filehandle %s properly warning. Call `close` explicitly at a point where you can check the return value and silence that warning.
- **Indirect handles** - FILEHANDLE may be any expression producing a usable filehandle: a bareword (`close STDERR`), a scalar (`close $fh`), a glob ref (`close \*FH`),

or an autovivified handle from `open my $fh, ...`

- **Pipe close waits** - closing a pipe handle blocks until the child exits, so `$?` is populated by the time `close` returns. Under threads, if the same pipe handle is still open in another thread, `close` returns true immediately without reaping the child.
- **Pipe close with non-zero exit** - `close` returns false and sets `$!` to `0` so that `$!? "$!" : "exit $?"` discriminates between an OS-level failure and an ordinary non-zero exit.
- **Closed or invalid handle** - closing a handle that is not open sets `$!` to "Bad file descriptor" and returns false. Under `use warnings`, a `close()` on unopened filehandle warning is emitted.
- **STDIN / STDOUT / STDERR** - can be closed like any other handle. The descriptor is freed and will be reused by the next `open` on the standard handle, which is how the classic `open STDOUT, ">", "log"` redirect works.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `open` - counterpart; an `open` on an already-open handle does an implicit `close` first (but does not reset `$.`)
- `print` - final buffered `print` is flushed inside `close`, so the write's real success is only known after `close`
- `fileno` - descriptor number of a handle; becomes invalid once `close` succeeds
- `binmode` - layer stack set by `binmode` is torn down during `close`, and its teardown can itself fail
- `$?` - child exit status populated by `close` on a pipe handle
- `$!` - error from the failing `close` operation

I/O

closedir

Close a directory handle opened by `opendir`.

`closedir` releases the operating-system directory stream associated with `DIRHANDLE` and frees the slot so the handle can be reused. It is the counterpart to `opendir`, and the only supported way to close a directory handle - `close` operates on file handles and does **not** accept a directory handle, because file handles and directory handles live in separate namespaces.

Synopsis

```
closedir DIRHANDLE
closedir $dh
```

What you get back

1 on success, `undef` on failure (with `$!` set to the error from the underlying `closedir(3)` call). The return value is worth checking on networked or FUSE-backed filesystems where the close may surface a deferred I/O error.

```
closedir $dh
    or warn "closedir $path failed: $!";
```

Closing a directory handle that is already closed, or one that was never successfully opened, returns `undef` and - under use warnings - emits a `closedir()` attempted on invalid dirhandle warning.

Lexical directory handles and auto-close

A directory handle stored in a lexical scalar - the form produced by `opendir my $dh, $path` - is closed automatically when the scalar goes out of scope (or is reassigned, or the program exits). Explicit `closedir` is then a matter of timing, not correctness:

```
sub entries {
    my ($path) = @_;
    opendir my $dh, $path or die "opendir $path: $!";
    return readdir $dh;
    # $dh is closed here as it leaves scope
}
```

Call `closedir` explicitly when you need the release to happen at a known point - before forking, before renaming the directory on platforms that hold it open, before checking the return value, or simply when the enclosing scope is long and holds many other resources.

Bareword directory handles (package globals, e.g. `opendir DH, $path`) are **not** auto-closed. They persist until the program ends or `closedir DH` is called.

Global state it touches

`closedir` does not read or write any of Perl's documented special variables. It sets `$!` on failure. It does **not** affect `$_`, the selected filehandle, or any of the output-separator globals.

Examples

Basic open/read/close cycle:

```
opendir my $dh, "." or die "opendir: $!";
my @names = readdir $dh;
closedir $dh;
```

Check the return value - useful on network filesystems:

```
opendir my $dh, "/mnt/nfs/project" or die "opendir: $!";
my @entries = readdir $dh;
closedir $dh
    or die "closedir failed: $!";
```

Reset to the start with `rewinddir` instead of `closedir` + `opendir` when you want to re-read the same directory:

```
opendir my $dh, $path or die $!;
my @first = readdir $dh;
rewinddir $dh;
my @again = readdir $dh;           # same set as @first
closedir $dh;
```

Bareword handle - must be closed explicitly:

```
opendir DH, "/etc" or die $!;
while (my $name = readdir DH) {
    last if $name eq "hosts";
}
closedir DH;                       # not optional for barewords
```

Using `close` on a directory handle does **not** work:

```
opendir my $dh, "." or die $!;
close $dh;                         # wrong - does not close the
    ↪dir
closedir $dh;                       # correct
```

Edge cases

- **close on a directory handle:** file handles and directory handles are distinct types. `close` on a directory handle is either a no-op or an error depending on the handle form; always use `closedir` to close what `opendir` opened.
- **Double close:** calling `closedir` twice on the same handle returns `undef` on the second call and sets `$!`. Under `use warnings` a `closedir() attempted on invalid dirhandle` warning is emitted.
- **Closed or never-opened handle:** same behaviour as double close - `undef` return and a warning under `use warnings`.
- **Pending `readdir` state:** `closedir` discards any unread entries. If you want to restart iteration, use `rewinddir` instead of `close` + `reopen`.
- **`seekdir` / `telldir` positions:** positions obtained from `telldir` are invalidated by `closedir`. They are only meaningful for the lifetime of a single open directory

stream.

- **Scope exit during exception:** lexical directory handles are still closed when the scope unwinds via `die`, so explicit `closedir` in cleanup code is usually redundant.
- **Fork:** after `fork`, both parent and child share the directory stream's kernel state; each should call `closedir` in whichever process no longer needs it.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `opendir` - open a directory handle; `closedir` is its counterpart
- `readdir` - read the next entry (or all entries in list context) from an open directory handle
- `rewinddir` - reset the directory stream to the beginning without closing it
- `seekdir` - reposition the directory stream to a value previously returned by `telldir`
- `telldir` - return the current position within an open directory stream
- `close` - close a **file** handle; does not work on directory handles

Sockets

connect

Initiate a connection from a socket to a remote address.

`connect` is the client-side half of the socket handshake. Given a `SOCKET` that was created with `socket` and a packed address `NAME` in the form the socket's address family expects, it asks the kernel to establish a connection to that peer. It is a thin wrapper over the `connect(2)` system call and inherits every one of that call's blocking, non-blocking, and error semantics.

Synopsis

```
connect SOCKET, NAME
```

What you get back

Truthy (1) on success, false on failure with `$!` set to the system error. Always check the return value - there is no useful default behaviour when a connection attempt fails:

```
connect($sock, $peer)
    or die "connect to $host:$port failed: $!";
```

Global state it touches

- `$!` is set to the kernel's `errno` on failure. Interpret it against the `Errno` constants, not by string value - the stringified message is locale-dependent.
- The socket handle itself carries the connection state; no other special variables are involved.

Building NAME: it's a packed struct, not a string

The second argument is **not** a "host:port" string. It's a packed `sockaddr` in the exact byte layout the kernel expects for the socket's address family. Build it with the helpers from `Socket`:

- `sockaddr_in($port, $inaddr)` for IPv4 - `$inaddr` from `inet_aton` or `inet_pton`.
- `sockaddr_in6($port, $in6addr)` for IPv6 - `$in6addr` from `inet_pton(AF_INET6, ...)`.
- `sockaddr_un($path)` for Unix-domain sockets.

Hand-packing with `pack` works, but there is no reason to do so when `Socket` already exports the right builders.

Examples

A plain IPv4 TCP client:

```
use Socket;

socket(my $sock, PF_INET, SOCK_STREAM, getprotobyname("tcp"))
    or die "socket: $!";

my $addr = sockaddr_in(80, inet_aton("93.184.216.34"));
connect($sock, $addr)
    or die "connect: $!";

print $sock "GET / HTTP/1.0\r\nHost: example.com\r\n\r\n";
```

Modern address-family-agnostic client via `getaddrinfo`:

```
use Socket qw(getaddrinfo SOCK_STREAM);

my ($err, @res) = getaddrinfo("example.com", "80",
                              { socktype => SOCK_STREAM });
die "getaddrinfo: $err" if $err;

my $sock;
for my $ai (@res) {
    socket($sock, $ai->{family}, $ai->{socktype}, $ai->{protocol})
        or next;
    last if connect($sock, $ai->{addr});
}
```

(continues on next page)

(continued from previous page)

```

    close $sock;
    undef $sock;
}
defined $sock or die "could not connect to any address";

```

A Unix-domain client:

```

use Socket;

socket(my $sock, PF_UNIX, SOCK_STREAM, 0) or die "socket: $!";
connect($sock, sockaddr_un("/tmp/my.sock"))
    or die "connect: $!";

```

Non-blocking connect with a timeout via `select` - the idiomatic way to bound a connect attempt in pure Perl:

```

use Socket;
use Fcntl qw(F_GETFL F_SETFL O_NONBLOCK);
use Errno qw(EINPROGRESS);

socket(my $sock, PF_INET, SOCK_STREAM, 0) or die "socket: $!";
my $flags = fcntl($sock, F_GETFL, 0);
fcntl($sock, F_SETFL, $flags | O_NONBLOCK);

my $ok = connect($sock, sockaddr_in(80, inet_aton("93.184.216.34
↪")));
if (!$ok && $! == EINPROGRESS) {
    my $w = '';
    vec($w, fileno($sock), 1) = 1;
    my $n = select(undef, $w, undef, 5.0); # 5-second timeout
    $n > 0 or die "connect timed out";
    # Check SO_ERROR to see whether the async connect succeeded:
    my $err = unpack("i", getsockopt($sock, SOL_SOCKET, SO_ERROR));
    $err == 0 or die "connect: " . do { local $! = $err; "$!" };
}

```

Edge cases

- **Address-family mismatch.** If NAME was packed for a different family than the socket was created with (e.g. `sockaddr_in` against a `PF_INET6` socket), `connect` fails with `EAFNOSUPPORT` or `EINVAL`. The packed bytes carry the family; the kernel checks.
- **Already connected.** Calling `connect` on a stream socket that is already connected fails with `EISCONN`.
- **Connectionless sockets.** On `SOCK_DGRAM` / `SOCK_RAW` sockets, `connect` does not exchange packets; it records a default peer so that subsequent `send` / `recv` calls can omit the address. It can be called again with a new NAME, or cleared by calling with an `AF_UNSPEC` `sockaddr`.

- **Non-blocking sockets.** If the socket is in non-blocking mode, a TCP connect typically returns false with `$!` set to `EINPROGRESS`. The connection is **not yet established**. Wait for the socket to become writable (via `select` or your event loop), then read `SO_ERROR` with `getsockopt` to obtain the final status. A zero `SO_ERROR` means the connection is up; anything else is the real failure code.
- **Signals interrupting the call.** A blocking connect can return false with `$!` set to `EINTR` if a signal handler ran. The socket is still connecting in the background; you must wait on writability and check `SO_ERROR` as in the non-blocking case - **do not** retry the connect call, that yields `EALREADY`.
- **Unbound source address.** If `bind` was not called first, the kernel picks a source address and ephemeral port automatically. The common case is to skip `bind` on the client.
- **Closed or unopened socket.** Fails with `EBADF`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `socket` - create the endpoint that `connect` operates on
- `bind` - pin the local address before connecting; rarely needed on a client
- `accept` - the server-side counterpart to `connect`
- `getpeername` - recover the peer address of a connected socket
- `shutdown` - close one direction of a connected socket without releasing the descriptor
- `Socket` - packers (`sockaddr_in`, `sockaddr_un`, ...) and constants (`PF_INET`, `SOCK_STREAM`, ...) used to build arguments
- `Errno` - symbolic constants for comparing `$!` against `EINPROGRESS`, `EISCONN`, `ECONNREFUSED`, ...

Control flow

continue

Attach a block to a loop that runs after every iteration, just before the condition is re-tested.

`continue` is not a built-in function - it is a flow-control keyword. In its one remaining sense it introduces the trailing block of a `while`, `until`, or `foreach` loop. The block runs at the end of each iteration **and** whenever `next` transfers control out of the main body, but **not** when `last` or `redo` do. It plays the role that the third clause of a C `for (init; cond; step)` loop plays in C.

Synopsis

```

while (EXPR) { BODY } continue { STEP }
until (EXPR) { BODY } continue { STEP }
for VAR (LIST) { BODY } continue { STEP }
foreach VAR (LIST) { BODY } continue { STEP }
LABEL: while (EXPR) { BODY } continue { STEP }

```

What you get back

`continue` is a statement, not an expression. It has no return value and cannot appear in value position. The loop itself yields whatever its context demands (nothing in void context, the last evaluated expression under some constructs); the `continue` block does not contribute to that.

Execution model

For a loop with an attached `continue` block, each iteration runs in this order:

1. Evaluate the condition (or take the next list element, for `foreach`). If the loop should not run again, exit.
2. Run the main body.
3. Run the `continue` block.
4. Go back to step 1.

The three loop-control keywords interact with the `continue` block differently:

- `next` - jumps straight to step 3. The `continue` block **is** executed, then the condition is re-checked. This is the point of having a `continue` block: state that must advance on every iteration (a counter, a line number, a one-shot match reset) still advances when the body short-circuits via `next`.
- `last` - leaves the loop entirely. The `continue` block is **not** executed.
- `redo` - restarts step 2 without re-checking the condition and without running the `continue` block. Use it when the body needs another pass over the current iteration's data (for example, after splicing in more input).

Omitting the `continue` block is equivalent to an empty one.

Examples

A counter that advances even when the body skips:

```

my $n = 0;
for my $line (@lines) {
    next if $line =~ /^s*#/;    # skip comments
    process($line);
} continue {
    $n++;                      # still counts skipped lines
}
print "scanned $n lines\n";

```

Equivalence to a C-style for loop - the two forms below are the same, because Perl rewrites the C-style header as a while with a continue block:

```
for (my $i = 1; $i < 10; $i++) { say $i }

my $i = 1;
while ($i < 10) {
    say $i;
} continue {
    $i++;
}
```

Resetting one-shot matches and line counters while streaming files with <>:

```
while (<>) {
    m?(fred)?    && s//WILMA/;
    m?(barney)?  && s//BETTY/;
} continue {
    print "$ARGV $.: $_";
    close ARGV if eof;           # reset $.
    reset if eof;               # reset ?pat?
}
```

The interaction with `redo` - note that `continue` runs on the next path but not on the `redo` path:

```
while (<>) {
    chomp;
    if (s/\\$//) {
        $_ .= <>;
        redo unless eof;        # merge continuation, re-enter body
    }                           # continue block skipped on redo
    process($_);
} continue {
    $seen++;                    # counts completed records only
}
```

Edge cases

- **No trailing block** - bare `continue` (without a following `{ ... }`) is a parse error in Perl 5.42. It used to be a valid statement inside the `switch` feature's `given / when` construct, where it fell through to the next `when`. That feature was removed in 5.42; `continue` now only has meaning as a trailing loop block.
- **last skips the continue block.** If the block holds cleanup you need on every path out, put the cleanup after the loop or in a scope guard, not in `continue`.
- **redo skips the continue block.** If the `continue` block advances a counter, a `redo` iteration will not tick it - that is usually what you want (the iteration is not yet finished) but it can surprise code that relies on the counter matching the number of body entries.

- **do { ... } while** is **not** a real loop - `last`, `next`, and `redo` do not work in it, and a trailing `continue` block is a syntax error.
- **Labeled loops** - `LABEL:` attaches to the `while` / `for` / `foreach` keyword, not to `continue`. `next LABEL` and `last LABEL` target the loop as a whole; the `continue` block is part of that loop.
- **A block by itself** is a one-shot loop, so a bare `{ ... } continue { ... }` parses and runs the `continue` block once after the body - rarely useful, but legal.
- **Foreach with aliasing** - inside a `foreach` loop the loop variable aliases the current list element. The `continue` block still runs with that alias in scope, so you can inspect or mutate the element there just as in the body.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `next` - jump to the `continue` block (if any) and re-test the condition; the one loop-control keyword that **does** run the `continue` block
- `last` - leave the loop entirely; **skips** the `continue` block
- `redo` - restart the body without re-testing the condition; **skips** the `continue` block
- `return` - leave the enclosing subroutine; also skips the `continue` block because it leaves the loop
- `break` - companion keyword of the removed `switch` feature; like the bare `given` / `when` form of `continue`, no longer available in 5.42

Numeric functions

cos

Return the cosine of a number given in radians.

`cos` is the standard trigonometric cosine: it takes an angle measured in **radians** and returns the ratio of the adjacent side to the hypotenuse of the corresponding right triangle. If `EXPR` is omitted, `cos` operates on `$_`. The result is always a floating-point number in the closed interval `[-1, 1]`.

Synopsis

```
cos EXPR
cos
```

What you get back

A double-precision float in `[-1, 1]`. Perl forwards the call to the platform's `C cos(3)` routine, so the precision and rounding behaviour match your `libm`. Special values follow IEEE 754: `cos(0)` is exactly 1, `cos("inf")` and `cos("nan")` are both NaN.

Radians, not degrees

The single most common mistake with `cos` is feeding it degrees. A full turn is $2 * \pi$ radians, not 360. Convert once and keep the converted value:

```
use POSIX qw(acos);
my $pi = acos(-1);                #  $\pi$  to full double precision

sub deg2rad { $_[0] * $pi / 180 }

print cos( deg2rad(60) ), "\n";    # 0.5
print cos( 60 ), "\n";            # -0.952412980415 - wrong if you
↪meant degrees
```

exports `deg2rad` and `rad2deg` if you would rather not write the conversion yourself.

Examples

Walk the unit circle in eight steps and print (`cos  $\theta$` , `sin  $\theta$`):

```
use POSIX qw(acos);
my $pi = acos(-1);

for my $k (0 .. 7) {
    my $theta = $k * $pi / 4;
    printf "%.3f %+.3f %+.3f\n",
        $theta, cos($theta), sin($theta);
}
```

Compare `cos` against its Taylor series around zero. For small `$x` the four-term expansion is already accurate to ten decimal places - a useful sanity check when porting numeric code:

```
sub cos_taylor {
    my $x = shift;
    my $x2 = $x * $x;
    1 - $x2/2 + $x2*$x2/24 - $x2*$x2*$x2/720;
}

my $x = 0.3;
printf "libm:  %.15f\n", cos($x);
printf "taylor: %.15f\n", cos_taylor($x);
```

Default argument. Inside a `while (<>)` loop `cos` without an argument reads the current line - almost always a bug, but the language allows it:

```
$_ = 0;
print cos, "\n";                # 1
```

Inverse cosine via the identity from `perlfunc`:

```
sub acos { atan2( sqrt(1 - $_[0] * $_[0]), $_[0] ) }

print acos(0.5), "\n";           # 1.0471975511966 (=  $\pi/3$ )
```

Using for the full set of inverse and hyperbolic functions rather than rolling your own:

```
use Math::Trig;
print acos(0.5), "\n";           # same result, no hand-written
↪ formula
```

Edge cases

- **Non-numeric arguments coerce to 0.** `cos("hello")` returns 1 because the string is numified to 0 first. Under `use warnings` this produces an `Argument "hello" isn't numeric in cos warning`. Validate input with `looks_like_number` from `Scalar::Util` if the source is untrusted.
- **Very large arguments lose precision.** `cos` first reduces its argument modulo 2π , and for $|x|$ on the order of $1e16$ the nearest representable double differs from x by more than π . The result is still a number in $[-1, 1]$, but it is essentially random. Reduce the angle yourself before the call if you are accumulating a phase over many iterations.
- **undef coerces to 0** and returns 1, with the usual uninitialized warning under `use warnings`.
- **IEEE specials:** `cos("inf")`, `cos("-inf")`, and `cos("nan")` all return NaN. Compare with `$result != $result` (the only value not equal to itself) to detect NaN, or use `POSIX::isnan`.
- **No complex-number support** in the built-in. For complex arguments use `cplx`, which overloads `cos` via operator overloading:

```
use Math::Complex;
my $z = cplx(1, 2);
print cos($z), "\n";           # 2.03272300701967-3.
↪ 0518977991518i
```

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `sin` - sine of an angle in radians; the companion function most often used alongside `cos`
- `atan2` - two-argument arctangent, the usual building block for inverse trig (`acos`, `asin`) in pure Perl
- `sqrt` - square root; appears in the classical `acos` identity `acos(x) = atan2(sqrt(1 - x*x), x)`

- - `acos`, `asin`, `atan`, hyperbolic variants, `deg2rad/rad2deg`, and great-circle helpers
- - overloads `cos` (and the rest of the trig family) for complex arguments
- POSIX - exposes `acos` directly, plus `isnan/isinf` for detecting the non-finite results discussed above

SCALARs and strings

crypt

One-way `passwd(5)`-style hashing of a plaintext string with a salt.

`crypt` is a thin wrapper over the system C library's `crypt(3)` function. It turns PLAINTEXT and SALT into a short string - a *digest* - and returns it. The same PLAINTEXT and SALT always produce the same digest; small changes in either produce a completely different digest. There is no inverse function: you cannot recover PLAINTEXT from a digest.

Warning

`crypt` is **not a general-purpose encryption function**, despite the name. It cannot be undone. It is also **not a modern password hash** on most traditional configurations - the historical Unix scheme considers only the first eight bytes of PLAINTEXT and produces a 13-byte digest over a 12-bit salt, which is unsuitable for protecting passwords against contemporary attackers. What `crypt` is good for is verifying a user-supplied string against an already-stored digest produced by the same system's `crypt`. For anything else - encryption, key derivation, or hashing large data - use a real cryptographic library. See *See also* below.

Synopsis

```
crypt PLAINTEXT, SALT
```

Both arguments are strings. The return value is a string digest.

What you get back

A string whose exact shape depends on the host's `crypt(3)` implementation. Traditionally it is 13 bytes: two bytes of salt followed by 11 bytes from the set `[./0-9A-Za-z]`. Modern `glibc`, `musl`, and BSD `libc`s accept extended salts that select stronger schemes (MD5, SHA-256, SHA-512, `bcrypt`, `yescrypt`) and return longer digests in the `$id$salt$hash passwd-file` format.

Treat the digest as opaque. Do not parse it, do not assume a length, and do not assume which bytes of SALT the implementation consumed. The correct way to verify a plaintext against an existing digest is to pass the digest itself back in as the salt:

```
if (crypt($plain, $digest) eq $digest) { ... }
```

The salt used to create the digest is embedded in the digest, so this re-runs the same scheme with the same parameters on the candidate plaintext. Code written this way works with the classical DES scheme and with every modern extension a given `crypt(3)` exposes, without having to know which one produced the stored digest.

Global state it touches

`crypt` does not read or write any Perl special variable. On Unicode input it may emit a fatal `Wide character in crypt` error (see *Edge cases*).

Examples

Verify a password against a stored passwd-style digest:

```
my $stored = (getpwuid($<))[1];      # encrypted field from passwd

print "Password: ";
chomp(my $word = <STDIN>);

if (crypt($word, $stored) eq $stored) {
    print "ok\n";
} else {
    die "Sorry...\n";
}
```

Create a fresh digest for a new password using a random two-character salt from the classical alphabet. The set is only a recommendation; the exact characters accepted in a salt are whatever the host `crypt(3)` allows, and Perl imposes no restriction of its own:

```
my @chars = ('.', '/', 0..9, 'A'..'Z', 'a'..'z');
my $salt  = $chars[rand 64] . $chars[rand 64];
my $hash  = crypt($plaintext, $salt);
```

Request a specific modern scheme by using an extended salt (glibc and compatible libcs). The `$id$` prefix selects the algorithm - `$1$` is MD5-crypt, `$5$` is SHA-256-crypt, `$6$` is SHA-512-crypt, `$2b$` is bcrypt where supported:

```
my $hash = crypt($plaintext, '$6$rounds=100000$' . $random_salt);
```

Re-verify using the stored digest as the salt - scheme-agnostic:

```
my $candidate = crypt($plaintext, $stored_digest);
my $match     = $candidate eq $stored_digest;
```

Constant-time comparison of the two digests is still the caller's responsibility; plain `eq` leaks timing information. If the threat model cares, compare byte-by-byte with a loop that does not short-circuit, or use a comparison routine from a crypto module.

Edge cases

- **First-8-byte truncation on classical DES.** On traditional Unix `crypt(3)`, only the first eight bytes of PLAINTEXT affect the digest: `crypt("password123", $s)` and `crypt("password456", $s)` collide. Extended schemes (`$1$, $5$, $6$, $2b$, ...`) do not have this limit, but the classical scheme is what you get from an unprefixd two-byte salt.
- **Salt-length assumptions are wrong.** Do not assume `substr($digest, 0, 2)` is the salt. Extended digests carry the salt in a different position and length. Re-use the full digest as the salt argument (`crypt($plain, $digest)`) and let `crypt(3)` parse it.
- **Unicode input.** If PLAINTEXT or SALT contains characters above codepoint 255, Perl attempts to downgrade a copy of the string to bytes before calling `crypt(3)`. If every character fits in a byte, the downgrade succeeds silently. If any character does not, `crypt` dies with `Wide character in crypt`. Encode explicitly (e.g. `Encode::encode_utf8`) if your plaintext might contain wide characters.
- **Unsupported scheme returns failure.** Passing a `$id$` prefix the host `crypt(3)` does not understand typically returns `undef` or a short string starting with `*0` or `*1`. Always check the return shape before using it.
- **`crypt(3)` may be absent or restricted.** On a small number of hosts the library function has been removed or replaced with a stub for export-control reasons. `crypt` in Perl is only as capable as the underlying libc call; if the libc call is a stub, so is this.
- **Not suitable for hashing large data.** `crypt` ignores most of its input on the classical scheme and is slow per call on the stretched schemes. It is the wrong tool for checksumming files or building content-addressable storage.

Differences from upstream

Fully compatible with upstream Perl 5.42. `ppperl` dispatches to the host `crypt(3)` the same way `perl5` does, so the accepted salt formats and the exact digest shape match whatever the system library produces - including the availability of `$1$/$5$/$6$/$2b$` extended schemes.

See also

- `Digest::MD5` - MD5-128 over arbitrary byte strings; use for fingerprinting data, not for password storage
- `Digest::SHA` - SHA-1/2 family digests; use for integrity checks and as a building block for HMAC and key derivation
- `Crypt::Bcrypt`, `Crypt::Argon2`, `Crypt::Passphrase` - CPAN modules for modern password hashing; prefer these over `crypt` for any new password-verification code
- `Crypt::Eksblowfish`, `Crypt::ScryptKDF` - CPAN modules for `bcrypt` and `scrypt`-based key derivation
- `getpwuid` - fetch the encrypted-password field from the local `passwd` database; its seventh field is the digest you pass as SALT when verifying

- `chomp` - remove the trailing newline from a password read with `<STDIN>` before passing it to `crypt`

I/O · Classes and OO

dbmclose

Break the binding between a DBM file and a hash previously attached by `dbmopen`.

`dbmclose` is the teardown counterpart to `dbmopen`. It flushes any pending writes, releases the DBM library's handles, and leaves `HASH` as an ordinary, untied hash you can continue to use or discard. In modern code `dbmclose` has been largely superseded by `untie` - the two accomplish the same thing, but `untie` works for every tied variable, not just DBM-bound hashes. Prefer `untie` unless you are maintaining existing `dbmopen/dbmclose` pairs and want them to read symmetrically.

Synopsis

```
dbmclose HASH
dbmclose %hash
```

What you get back

True on success, false on failure. Failure modes are rare - the common one is passing a hash that was never tied to a DBM file in the first place, in which case `dbmclose` returns false and sets `$!`. Most programs do not check the return value; the interesting errors (disk full, quota exceeded) surface on the writes that preceded the close, not on `dbmclose` itself. Check it when the DBM file is on a filesystem where a delayed write failure at close time is plausible:

```
dbmclose(%cache)
    or warn "dbmclose failed: $!";
```

Global state it touches

`dbmclose` reads and mutates the tie state of `HASH`. It does not touch any of the usual I/O-related special variables (`$,`, `$\`, `$|`, the selected filehandle); DBM access does not flow through the `PerlIO` layers that `print` and `friends` use. On failure, `$!` is set by the underlying DBM library.

Examples

Open a DBM file, iterate it, close it:

```
dbmopen(%HIST, '/usr/lib/news/history', 0666)
    or die "dbmopen: $!";
while (my ($key, $val) = each %HIST) {
    print $key, ' = ', unpack('L', $val), "\n";
}
dbmclose(%HIST);
```

Equivalent using the modern `tie/untie` pair - this is what new code should look like:

```
use AnyDBM_File;
use Fcntl;
tie(my %HIST, 'AnyDBM_File', '/usr/lib/news/history', O_RDWR, 0666)
    or die "tie: $!";
# ... use %HIST ...
untie %HIST;
```

Guard against closing an unbound hash:

```
my %h;
dbmclose(%h)
    or warn "not tied to a DBM file: $!";    # warns
```

Pair with an `eval` block to localise any cleanup errors without aborting the surrounding code:

```
eval {
    dbmclose(%cache) or die "dbmclose: $!";
    1;
} or warn $@;
```

Edge cases

- **Hash not tied to a DBM file:** returns false, sets `$!`. No exception is raised. The hash is left unchanged.
- **Hash tied via `tie`, not `dbmopen`:** `dbmclose` still works and is equivalent to `untie` for that hash, because `dbmopen` is itself implemented as a `tie` under the hood. Mixing the two APIs is legal but confusing - pick one per hash and stay with it.
- **References still held to the tied object:** like `untie`, `dbmclose` severs the binding between the hash and the DBM file, but if other code still holds a reference to the underlying tied object the DBM library may not actually release its handles until that reference is dropped. The symptom is a DBM file that stays locked or open past the `dbmclose` call. A `use warnings` program receives the `untie attempted while N inner references still exist` warning in that case.
- **Calling `dbmclose` twice on the same hash:** the first call succeeds, the second returns false. Idempotent-style code should either track whether the hash is currently tied or ignore the failure.
- **Hash is local'd or goes out of scope without `dbmclose`:** the binding is broken implicitly when the hash is destroyed, and the DBM library closes the file as part of that teardown. Explicit `dbmclose` is preferable because it makes write-flush failures visible at a known point in the program.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `dbmopen` - the opening half of the pair; every `dbmclose` matches a prior `dbmopen` on the same hash
- `untie` - the modern, general-purpose replacement; works for every tied variable, not just DBM-bound hashes
- `tie` - the modern counterpart to `dbmopen`, used with `AnyDBM_File`, `DB_File`, `GDBM_File`, `NDBM_File`, or `SDBM_File` to bind a hash to a DBM file
- `close` - unrelated, but the analogous operation for filehandles: needed when you care about write-flush failures surfacing at a known point

I/O · Classes and OO

dbmopen

Bind a DBM file on disk to a hash so hash reads and writes become lookups and stores in the database.

`dbmopen` is the old-style way to tie a hash to a `dbm`, `ndbm`, `sdbm`, `gdbm`, or Berkeley DB file. It has been largely superseded by `tie` with an explicit DBM backend module (`DB_File`, `GDBM_File`, `SDBM_File`, `NDBM_File`, `AnyDBM_File`), which gives you control over which implementation is used and over per-backend options. Reach for `dbmopen` when you are reading legacy code or when the one-liner brevity matters more than backend choice.

Unlike `open`, the first argument is **not** a filehandle - it is the hash that will be bound. `HASH` is written with the % sigil.

Synopsis

```
dbmopen %HASH, $DBNAME, $MASK
dbmopen(%HASH, $DBNAME, 0)           # open existing only; never create
dbmclose %HASH                       # or: untie %HASH
```

What you get back

A true value on success, a false value on failure. On failure `$!` is set to the system error that prevented the bind. The common failure cases are “database does not exist and `MASK` is 0” and permission errors on the backing files.

```
dbmopen(%cache, $path, 0)
    or die "cache $path missing or unreadable: $!";
```

After a successful call, `%HASH` behaves like any other hash: `$HASH{key} = $value` writes to the database, `$HASH{key}` reads from it, `delete` removes an entry, `exists`

tests membership, and `each / keys / values` iterate. The data lives on disk; the hash is only a view.

Arguments

- **HASH** - the hash variable to bind. Must be written with `%`. Any existing contents are discarded. The hash stays bound until `dbmclose` or `untie` is called on it, or until it goes out of scope.
- **DBNAME** - the database pathname **without** the backend's file extension. Traditional `dbm` / `sdbm` stores use two files named ``DBNAME.dir`` and ``DBNAME.pag``; `gdbm` and Berkeley DB use a single file. Passing `"/var/lib/app/cache"` opens `/var/lib/app/cache.dir + /var/lib/app/cache.pag` under `sdbm`, or `/var/lib/app/cache` under `gdbm`.
- **MASK** - creation mode for any files that have to be created, in the same form as the third argument to `chmod` (typically an octal literal like `0666` or `0644`). The active `umask` is applied to `MASK`, so real on-disk permissions are `MASK & ~umask`. **A MASK of 0 suppresses creation:** if the database does not already exist, `dbmopen` returns false and `$!` is set.

Global state it touches

- `$!` - set on failure (database missing, permission denied, backing store corrupt).
- The active `umask` - masks `MASK` when new backing files are created.

Examples

Open a history database, iterate with `each`, then close:

```
dbmopen(%HIST, '/usr/lib/news/history', 0666)
    or die "open history: $!";
while (my ($key, $val) = each %HIST) {
    print $key, ' = ', unpack('L', $val), "\n";
}
dbmclose %HIST;
```

Open read-only, refusing to create the database:

```
dbmopen(%cache, "$ENV{HOME}/.app/cache", 0)
    or die "no cache at $ENV{HOME}/.app/cache: $!";
my $hit = %cache{$key};
dbmclose %cache;
```

Pick a specific backend by loading its module before calling `dbmopen`. `AnyDBM_File` consults `@AnyDBM_File::ISA` to decide which implementation `dbmopen` uses:

```
use DB_File;
dbmopen(%NS_Hist, "$ENV{HOME}/.netscape/history.db", 0644)
    or die "open netscape history: $!";
```

Write-protected database - assignment silently fails (or croaks depending on the backend); guard with `eval` when you need to probe:

```
dbmopen(%db, $path, 0) or die $!;
my $writable = eval { $db{__probe__} = 1; delete $db{__probe__}; 1 }
↪;
warn "read-only\n" unless $writable;
```

Large database - avoid `keys` / `values`, which materialise the whole list in memory. Use `each` instead:

```
dbmopen(%big, $path, 0) or die $!;
while (my ($k, $v) = each %big) {
    process($k, $v);
}
dbmclose %big;
```

Edge cases

- **Hash sigil is required.** `dbmopen %h, $name, 0666` - writing `$h` or `\%h` does not work. The first slot is the hash's name as a hash, not a reference.
- **MASK = 0 is the “open existing only” idiom.** A plausible-looking `0666` silently creates a fresh empty database if the file is missing, which is almost never what a reader wants on a cache lookup path.
- **Extensions are managed by the backend.** Do not pass `"cache.dir"` or `"cache.pag"` - pass `"cache"`. `sdbm` appends `.dir / .pag`; `gdbm` writes to the bare name.
- **One open per process for older DBM.** If the backend is the traditional `dbm(3)` library (rare today), a process may only have a single DBM open at a time. `sdbm` and `gdbm` do not have this limit.
- **Value size limits.** `sdbm` caps keys + values at roughly 1008 bytes per record; over-size writes are truncated or rejected. `gdbm` and Berkeley DB have much higher limits. When in doubt, use `tie` with `DB_File` and check the backend's own limits.
- **Numeric values are bytes on disk.** Storing `$h{k} = 42` writes the string `"42"`, not an integer. Pack explicitly with `pack` / `unpack` when the field needs a fixed binary layout (as in the news-history example above).
- **Closing is optional but recommended.** The bind is released when the hash goes out of scope, but explicit `dbmclose` or `untie` flushes pending writes and surfaces close-time errors.
- **Re-binding an already-bound hash.** Calling `dbmopen` on a hash that is already bound closes the previous binding first.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `dbmclose` - release a dbmopen binding and flush pending writes
- `tie` - the modern replacement; pair with `DB_File`, `GDBM_File`, `SDBM_File`, or `AnyDBM_File` to pick a backend explicitly
- `untie` - release a binding made with either `dbmopen` or `tie`
- `each` - stream entries without materialising the whole database in memory
- `umask` - masks the MASK argument when backing files are created
- `open` - the ordinary file-open built-in, which takes a filehandle and not a hash

Control flow

defer

Schedule a block to run when the enclosing scope exits, for any reason.

`defer` is a flow-control keyword, not a function. Writing `defer { ... }` inside a scope records the block for later execution; when control leaves the enclosing block - by falling off the end, by `return`, by `die`, by `goto`, or by a loop-control statement (`next`, `last`, `redo`) - the stored block runs on the way out. It is the standard idiom for pairing acquisition with release (open/close, lock/unlock, push/pop) without relying on object destructors or wrapping every path in `eval`.

The feature is **experimental** and must be enabled. Either opt in explicitly or pick up the bundle:

```
use feature 'defer';    # explicit
use v5.36;              # bundle - enables defer among others
```

Under `use warnings 'experimental::defer'` (or plain `use warnings`) the first use emits an experimental warning. Silence it with `no warnings 'experimental::defer'` once you have decided to rely on the feature.

Synopsis

```
defer BLOCK
```

`defer` takes a brace-delimited block and nothing else. There is no expression form, no filehandle form, and no return value - the statement exists for its side effect on scope exit.

What you get back

`defer` is a statement, not an expression. It contributes no value to its surrounding context. The block's own return value is discarded when it runs at scope exit.

When the deferred block runs

The deferred block is invoked **once**, when control leaves the directly enclosing block, regardless of how control leaves:

- normal fallthrough off the end of the block,
- explicit `return` from the containing subroutine,
- an exception thrown by `die` or propagated from a called sub,
- `goto` out of the block,
- loop control via `next`, `last`, or `redo`.

If the flow of execution never reaches the `defer` statement itself, the block is **not** enqueued and does not run. This is the direct opposite of an `END` phaser, which the compiler enqueues unconditionally:

```
use feature 'defer';

{
    defer { say "This will run"; }
    return;
    defer { say "This will not"; }    # never reached, never
    ↪scheduled
}
```

LIFO ordering

Multiple `defer` blocks in the same scope run in **last-in, first-out** order - the most recently scheduled block runs first. This matches the natural nesting of resource acquisition: the last thing acquired is the first thing released.

```
use feature 'defer';

{
    defer { say "1"; }
    defer { say "2"; }
    defer { say "3"; }
}
# prints 3, then 2, then 1
```

Global state it touches

None directly. The deferred block sees interpreter globals (`$_`, `$!`, `$@`, selected filehandle, etc.) with whatever values they hold **at the moment the block runs**, not at the moment it was scheduled. A `defer` that wants to preserve `$!` or `$@` across cleanup must save and restore them itself - otherwise a system call inside the block will clobber them and the caller of the enclosing sub will see the overwritten values.

```
defer {
    local ($!, $@);
```

(continues on next page)

(continued from previous page)

```
    close $fh;           # clobbers $! on failure - local'd copy_
    ↪absorbs it
}
```

Examples

Pair an open with its close, guaranteed to run even on die:

```
use feature 'defer';

sub read_config {
    open my $fh, "<", "config.ini" or die "open: $!";
    defer { close $fh; }

    while (<$fh>) { ... }
    # close $fh runs here on fallthrough, on return, and on die.
}
```

Unlock a shared resource on every exit path:

```
use feature 'defer';

sub critical_section {
    $lock->acquire;
    defer { $lock->release; }

    do_work();           # release runs on fallthrough
    return if $shortcut; # release runs before the return completes
    die "boom" if $bad;  # release runs, then the exception_
    ↪propagates
}
```

LIFO cleanup of nested resources - the outer resource is released last, which is usually what you want:

```
use feature 'defer';

sub copy_file {
    open my $in, "<", $src or die "open $src: $!";
    defer { close $in; }

    open my $out, ">", $dst or die "open $dst: $!";
    defer { close $out; }

    print {$out} $_ while <$in>;
    # close $out runs first, then close $in.
}
```

Temporarily adjust interpreter state and restore it on exit without using `local`:

```

use feature 'defer';

sub with_separator {
    my $saved = $,;
    $, = " | ";
    defer { $, = $saved; }

    print @_, "\n";
}

```

Edge cases

- **Conditional scheduling.** A defer statement only schedules its block when control actually reaches it. An early `return` or `die` that happens before the defer line leaves nothing to run. Place the defer immediately after the paired acquisition so they cannot become separated by an intervening failure.
- **Once scheduled, always runs.** There is no mechanism to un-schedule a defer block. If the acquisition that the block releases has failed, either do not schedule the defer (keep it after the success check) or make the block idempotent so a redundant release is harmless.
- **No control-flow escape from the block.** A defer block may throw an exception, but it may **not** `return` from the enclosing sub, `goto` a label outside itself, or run `next`, `last`, or `redo` for a loop that encloses the defer. Those constructs are fine *inside* the defer when they act on loops **contained entirely within** the block's own body:

```

defer {
    foreach (1 .. 5) { last if $_ == 3; }    # permitted
}

foreach (6 .. 10) {
    defer { last if $_ == 8; }                # forbidden
}

```

- **Exception during exception unwind.** If the defer block is running because an exception is propagating and the block itself throws, the language does not specify the resulting exception - only that the caller will see one. Do not rely on which exception wins. Wrap risky cleanup in `eval` if you need both signals preserved.
- **Exceptions propagate normally.** An exception thrown by a defer block that is running for any other reason (normal exit, return, loop control) propagates to the caller exactly like any other exception. A failing close in cleanup can therefore turn a successful-looking function into a caller-visible die.
- **\$@ and \$! inside the block.** On the exception-driven exit path, `$@` holds the in-flight exception while the defer block runs. Reading it is fine; replacing it changes what the caller sees after the unwind completes.
- **Scope granularity.** `defer` binds to the **immediately** enclosing block, not to the enclosing sub. A defer inside `if (...) { defer { ... } }` runs when the `if`

block exits, not when the sub returns. Put the `defer` at subroutine scope if you want subroutine-exit semantics.

- **Experimental warning.** Under `use warnings` the first occurrence warns. Opt in deliberately with `no warnings 'experimental::defer'` once you have committed to the feature.
- **Not available without the feature guard.** Without `use feature 'defer'` (or a bundle that enables it), `defer { ... }` is parsed as a call to a subroutine named `defer` followed by a block - almost certainly not what you meant, and usually a compile-time error.

Differences from upstream

Fully compatible with upstream Perl 5.42. The feature remains marked experimental in pperl to mirror upstream's classification; behaviour (scheduling on statement execution, LIFO order, exception propagation, restrictions on control-flow escape) matches Perl 5.42.

See also

- `eval` - the established tool for turning exceptions into return values; `defer` complements it by guaranteeing cleanup without an explicit `eval` wrapper on every path
- `die` - one of the exit paths that triggers deferred blocks; read alongside `defer` when designing cleanup
- `return` - another exit path; a `defer` scheduled before a `return` runs before the caller resumes
- `local` - the other idiom for scoped state restoration; pick `local` for dynamic scoping of a single variable, `defer` for arbitrary cleanup actions
- `$@` - current exception; visible inside a `defer` block running on the exception-unwind path
- `$!` - system `errno`; easily clobbered by cleanup syscalls, `local`-ise inside the deferred block if the caller needs it preserved

Misc

defined

Test whether a value, variable, or subroutine is defined.

`defined` returns true when `EXPR` holds any value other than the undefined value `undef`, and false when it holds `undef`. It is the one and only way to tell `undef` apart from the other false values (`0`, `" "`, `"0"`), which a plain boolean test cannot.

Synopsis

```
defined EXPR
defined
defined &SUBROUTINE
```

What you get back

A boolean: true if EXPR is defined, false if it is `undef`.

With no argument, defined operates on `$_`.

The result is context-sensitive only in the usual boolean sense; defined does not propagate list context into EXPR.

Why it exists, and why a plain boolean is not enough

Many operations return `undef` to signal failure, end of file, a system error, an uninitialised variable, or some other exceptional condition. `defined` lets you distinguish that sentinel from a legitimate result. A simple boolean test treats `undef`, `0`, `"`, and `"0"` the same - all four are false - so you cannot use truthiness to tell “nothing was returned” apart from “zero was returned”:

```
my $n = some_count();
if ($n)      { ... }    # false for 0 AND for undef
if (defined $n) { ... }  # false only for undef
```

Note that `undef` is itself a valid scalar value. Its presence does not *necessarily* mean something went wrong: `pop @empty` returns `undef`, but so does `pop @array` when the element popped happened itself to be `undef`. `defined` tells you the value is `undef`; it does not tell you why.

Default argument: `$_`

With no argument, defined tests `$_`:

```
while (<$fh>) {
    next unless defined;           # same as: defined $_
    chomp;
    ...
}
```

defined &subroutine

`defined &func` asks whether subroutine `func` has ever been defined. A forward declaration (`sub func;`) does **not** make it defined; an actual body does:

```
sub greet { "hello" }
print defined &greet ? "yes\n" : "no\n";    # yes
print defined &missing ? "yes\n" : "no\n";  # no
```

A subroutine that is not defined may still be *callable*: the package's AUTOLOAD method can bring it into existence on first call. `defined &func` reflects the current state of the symbol table, not what a future AUTOLOAD might produce. See `perlsub`.

The return value is unaffected by forward declarations.

Hash elements: `defined` vs `exists`

On a hash element, `defined` asks about the **value**, not the key:

```
my %h = (a => 1, b => undef);

print exists $h{a} ? 1 : 0, "\n";    # 1
print defined $h{a} ? 1 : 0, "\n";    # 1

print exists $h{b} ? 1 : 0, "\n";    # 1    - key is present
print defined $h{b} ? 1 : 0, "\n";    # 0    - value is undef

print exists $h{c} ? 1 : 0, "\n";    # 0    - key not present
print defined $h{c} ? 1 : 0, "\n";    # 0    - also autovivifies? no,
↳ rvalue is safe
```

Use `exists` to ask “is this key in the hash?” and `defined` to ask “does this slot hold a usable value?”. The two differ whenever a value was explicitly assigned `undef` or left `undef` by construction (e.g. `$h{b} = undef`, or `@h{qw(a b c)} = (1)`).

The same distinction applies to array elements: `exists $a[7]` reports whether the array is long enough and the slot has been assigned, while `defined $a[7]` reports whether that slot currently holds a non-`undef` value.

Examples

The common `pop`-until-empty idiom - stops only when the array is exhausted, not on a legitimate falsy element:

```
while (defined(my $val = pop @ary)) {
    process($val);
}
```

Check a system call for failure:

```
my $target = readlink $sym;
die "can't readlink $sym: $!" unless defined $target;
```

Call a coderef only if one was supplied:

```
sub dispatch {
    my ($cb, @args) = @_;
    return defined &$cb ? $cb->(@args) : die "no callback";
}
```

Provide a default without clobbering a legitimate `0` or `""`:

```
$debugging = 0 unless defined $debugging;
```

The modern form is the defined-or operator `//`, which does exactly this test inline:

```
$debugging //= 0;
my $name = $user // "anonymous";
```

Regex capture: `$1` being defined tells you the capture group participated in the match, even if it matched the empty string:

```
"ab" =~ /a(.*)b/;
print defined $1 ? "matched '$1'\n" : "no match\n";  # matched ''
```

Aggregates: `defined @array`, `defined %hash` are no longer supported

Historically `defined @array` and `defined %hash` reported whether memory for the aggregate had ever been allocated - an implementation detail that was rarely what callers actually wanted. Modern Perl rejects these forms. Test the aggregate's size instead:

```
if (@array) { print "array has elements\n" }
if (%hash) { print "hash has members\n" }
```

In boolean context an array evaluates to its element count and a hash evaluates to a true value iff it has any keys. That is the test you almost always wanted.

Testing *an element* of an aggregate is still fine and is the usual use: `defined $array[$i]`, `defined $hash{$key}`.

Overuse

`defined` is frequently overused. Zero and the empty string are defined values. A regex match that captures zero characters still sets `$1` to a defined empty string - the match did not fail; it matched something that happened to be of length zero. When a function returns `undef`, it is admitting it cannot give you a meaningful answer. Reach for `defined` when you are questioning the integrity of a value. For “is it zero?” or “is it empty?” a plain comparison to `0` or `""` is what you want.

Edge cases

- **Bare `defined`** with no argument tests `$_`. Inside `map`, `grep`, or a `while (<$fh)` loop this is the idiomatic form.
- **`defined(&func)` vs `defined(&{$ref})`**: both work. The second dereferences a code reference and asks whether the referenced slot currently holds a defined subroutine body.
- **`defined $hash{$key}` does not autovivify the key** - `defined` is an rvalue context and leaves the hash unchanged. Contrast with `$hash{$key}{inner} = 1`, which does autovivify.

- **Chained hash/array access:** defined `$h{a}{b}` autovivifies `$h{a}` into an empty hash ref in order to perform the inner lookup, even though the outer defined reads the result. This is a consequence of how the dereference chain is evaluated, not of defined itself. To probe nested structure without autovivifying, check each level with `exists` first.
- **defined `&`, defined `$!`:** after a successful match these are defined; after a failed match they retain whatever value they had from the previous successful match in the same scope. `defined` alone is not a safe “did the match succeed” test - check the match operator’s return value instead.
- **defined is not an lvalue.** defined `$x = $y` is a syntax error; parenthesise the assignment: `defined($x = $y)`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `undef` - the value defined tests against, and the operator that produces or clears it
- `exists` - ask whether a hash key or array index is present, independent of whether its value is `undef`
- `ref` - when the question is not “is it defined” but “what kind of reference is this”
- `// and //=` - defined-or operator; the modern inline form of `defined($x) ? $x : $default`
- `$_` - the default argument when `defined` is called with no expression

Hashes

delete

Remove the named key-value pair(s) from a hash, or element(s) from an array, and return what was removed.

`delete` reaches into an aggregate and removes the specified entries outright - not just by setting them to `undef`. After `delete`, `exists` on the same key or index returns false, `keys` no longer lists it, and iterating the hash with `each` skips it. The argument must be an element or slice of a hash or array; its final operation selects what to remove, and anything leading up to that operation (dereferences, method calls) is just navigation.

Synopsis

```
delete $hash{KEY}           # scalar key
delete @hash{LIST}          # hash slice: values of the listed keys
delete %hash{LIST}          # key/value hash slice (5.20+)
delete $array[INDEX]        # array element
delete @array[LIST]         # array slice
```

What you get back

In list context, `delete` returns the value or values removed, in argument order. A slice argument whose key or index did not exist yields `undef` at the matching position, so the length of the returned list matches the length of the key/index list.

In scalar context, `delete` returns the value of the last element removed, or `undef` if the key/index was not present.

The 5.20+ key/value hash-slice form `delete %hash{LIST}` returns a list of alternating key/value pairs - two elements for every deleted entry - which makes it the idiomatic way to pluck a named set of entries and keep both sides.

```
my %h = (foo => 11, bar => 22, baz => 33);
my $v = delete $h{foo};           # 11
my @v = delete @h{qw(bar quux)};  # (22, undef)
my %kv = delete %h{qw(baz quux)}; # (baz => 33, quux => undef)
```

Global state it touches

- **%ENV** - `delete $ENV{VAR}` unsets the environment variable for the current process and its future children. The effect is real, not just the Perl-side hash; system calls see the change.
- **tied aggregates** - `delete` dispatches to the `DELETE` method of the tying package. The return value is whatever the implementation chooses to hand back; some tied hashes return the deleted value, some do not.
- **\$_** - unaffected. `delete` does not default to anything; `EXPR` is mandatory.

Examples

Remove one key, inspect what was there:

```
my %cfg = (host => "db1", port => 5432, user => "alice");
my $old_host = delete $cfg{host};
# $old_host is "db1"; exists $cfg{host} is false
```

Free memory by removing many entries in one call (hash slice):

```
delete @cache{@stale_keys};
```

Pluck a set of named entries and keep the pairs (5.20+ kv-slice):

```
my %h = (a => 1, b => 2, c => 3);
my %taken = delete %h{qw(a c)};
# %taken is (a => 1, c => 3); %h is (b => 2)
```

Delete through a nested structure. Leading navigation can be arbitrary - only the final operation decides what gets removed:

```
my $tree = { users => { alice => { roles => [qw(admin ro)] } } };
delete $tree->{users}{alice}{roles}[0];
```

(continues on next page)

(continued from previous page)

```
# roles is now (undef, "ro")
delete $tree->{users}{alice};
# the whole user record is gone
```

Avoid autovivification on the read side while still removing on the write side. `delete` never vivifies the intermediate hashes it walks through:

```
delete $h{outer}{inner};    # does not create $h{outer} if absent
```

Compare with `splice` when you want indices to renumber. `delete` on an array element leaves a hole; `splice` closes the hole:

```
my @a = (10, 20, 30, 40);
delete $a[1];
# @a is (10, undef, 30, 40); $#a is still 3

my @b = (10, 20, 30, 40);
splice(@b, 1, 1);
# @b is (10, 30, 40); $#b is 2
```

Edge cases

- **delete vs assigning `undef`:** `$h{k} = undef` leaves `k` in the hash with an undefined value; `exists $h{k}` is still true. `delete $h{k}` removes the key entirely.
- **Missing key in scalar context:** `delete $h{nope}` returns `undef`. That's indistinguishable from deleting a key whose value was `undef`; use `exists` first if the distinction matters.
- **each + delete inside the loop:** safe when you delete only the current key (the key just returned by `each`). Deleting *other* keys during iteration can skip entries or revisit them - Perl's hash iterator is not stable across structural change. The safe rewrite is to collect keys first:

```
for my $k (grep { $h{$_} < 0 } keys %h) {
    delete $h{$k};
}
```

- **Trailing array deletions shrink the array:** if the deleted elements are at the top end, `$#array` drops to the highest index still testing true under `exists`. Holes in the middle stay as holes.
- **delete on an array is discouraged:** the concept of “missing index” is not coherent for arrays - indices are positional, not named. Leave array shrinkage to `shift`, `pop`, or `splice`. WARNING in upstream POD restated: reach for an array delete only when you know you want a hole.
- **delete local \$h{k}:** removes the entry for the enclosing block and reinstates it on scope exit. Useful for temporarily hiding configuration keys from code called inside the block without losing the original value:

```
{  
    local $ENV{LANG};          # still present, but undef  
    delete local $ENV{PATH};  # gone for this block only  
    run_thing();  
} # both restored to original values here
```

- **Tied hash, tied array:** delete calls DELETE; the return value is whatever the tie implementation supplies. A tied DBM hash deletes the entry on disk.
- **Readonly target:** delete on a readonly element croaks with Modification of a read-only value attempted.
- **The expression must end in an element or slice:** delete $\$h\{k\} + 1$ is a compile-time error; delete has lower precedence than arithmetic and expects an lvalue-ish aggregate lookup, not a computed value.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `exists` - the query counterpart; use it before delete when you need to tell “was never there” apart from “was there with value `undef`”
- `each` - hash iterator; safe to combine with delete only when you delete the current key
- `keys` - snapshot of hash keys; the usual way to build a deletion list that won’t interact with iterator state
- `values` - companion to `keys` when you want the deleted-value set without the keys
- `splice` - the right tool when you want to remove array elements *and* renumber remaining indices
- `undef` - clears a whole aggregate (`undef %h`, `undef @a`) or a single scalar; the customary way to empty a hash or array wholesale rather than deleting every key

I/O · Control flow

die

Raise an exception.

`die` throws an exception built from `LIST`. If the exception happens inside an `eval`, the `eval` terminates with `undef` and `$@` holds the exception value. If nothing catches it, Perl prints the message to `STDERR` and the interpreter exits with a non-zero status. The exception can be a plain string, a list that stringifies into one, or a reference - typically a blessed object carrying structured error state.

Synopsis

```
die LIST
die $message
die $exception_object
die                                     # re-raise / propagate current $@
```

What you get back

`die` does not return. Control transfers to the nearest dynamically enclosing `eval` (which then returns `undef` in scalar context or the empty list in list context), or - if there is no enclosing `eval` - the interpreter unwinds, runs `END` and `DESTROY` handlers, and exits.

The exception value itself lands in `$@`:

- **String exceptions** - the stringified message, including any appended at `FILE line N.\n` (see below).
- **Reference exceptions** - the reference itself, unchanged. This is the idiomatic way to throw a blessed exception object.

On uncaught exit, the exit code is derived from `$!` and `$?`:

```
exit $! if $!;           # errno
exit $? >> 8 if $? >> 8; # child exit status
exit 255;                # last resort
```

Rely only on the exit code being non-zero; the exact value is a best-effort encoding of whatever global error state was live at the moment of the throw.

Global state it touches

- `$@` - written with the exception on unwind into an `eval`, read by the no-argument form of `die` to propagate an existing exception, and read by the uncaught-exception printer.
- `$!` - read when deriving an exit code for an uncaught exception, and commonly interpolated into the message itself (`die "open $f: $!"`).
- `$?` - read when deriving an exit code if `$!` is zero. `END` and `DESTROY` handlers can still mutate it before the process actually exits.
- `$.` - the current input line number, appended to string exceptions that lack a trailing newline.
- `$SIG{__DIE__}` - if set, the handler runs *before* the exception is propagated, with the exception as its argument. The handler can replace the exception by calling `die` again with a different value.
- `$_^S` - readable by `$SIG{__DIE__}` handlers to tell whether the die is happening inside an `eval`.

The trailing-newline rule

Whether `die` appends location information depends on one character at the end of the message:

- **Message ends in `"\n"`** - used verbatim. Nothing is appended. Use this form when the message is already a complete, user-facing diagnostic.
- **Message does not end in `"\n"`** - Perl appends " at FILE line N", and if a file is being read, ", <HANDLE> line M", then a final "." and newline. Use this form during development, or whenever locating the `die` call in the source is more useful than a tidy message.

```
die "bad config\n";           # "bad config\n"
die "bad config";            # "bad config at script.pl line_
↪42.\n"
```

A common idiom is to append " , stopped" to make the location suffix read naturally:

```
die "/etc/games is no good, stopped";
# /etc/games is no good, stopped at canasta line 123.
```

Reference exceptions never get location information appended - stringifying an object to inspect it would defeat the point of throwing an object. If you want file and line inside a structured exception, capture them at construction time (`__FILE__`, `__LINE__`, or `caller`).

Reference-valued exceptions

Passing a reference - typically a blessed object - lets `$@` carry structured state instead of a string you later have to parse:

```
package MyApp::Error;
sub new { my ($c, %a) = @_; bless { %a }, $c }
sub code { $_[0]{code} }
sub where { $_[0]{where} }

eval {
    die MyApp::Error->new(code => 'EPARSE', where => 'line 12');
};
if (my $err = $@) {
    if (ref($err) && $err->isa('MyApp::Error')) {
        warn "parse error at ", $err->where;
    }
    else {
        die $err; # not ours - re-raise
    }
}
```

Overload stringification on exception classes so that an uncaught exception still prints a useful message:

```
use overload '""' => sub { "[MyApp::Error ".$_[0]->code."]\n" };
```

The stringified form should end in a newline and be non-empty, matching the treatment of string exceptions.

Because `$@` is a global, copy it into a lexical before inspecting it - any method call, `ref`, or `isa` you run during handling can itself `eval` internally and clobber `$@`:

```
if (my $err = $@) { ... }      # safe
if ($@) { $@->method; ... }    # unsafe: method may reset $@
```

Propagation: die with no argument (or empty string)

When `LIST` is empty or stringifies to `"`, `die` does not throw a fresh exception - it re-raises whatever is in `$@`:

- If `$@` holds a **string**, `die` appends `"\t...propagated at FILE line N.\n"` and re-throws. This is the canonical way to let an exception continue up the call stack after a partial-match check:

```
eval { risky() };
die unless $@ =~ /Expected exception/;
```

- If `$@` holds an **object reference with a PROPAGATE method**, that method is invoked as `$@ = eval { $@->PROPAGATE(__FILE__, __LINE__) }`, and the return value replaces `$@` before the re-throw. Implement `PROPAGATE` on an exception class when you want the exception to record the chain of re-raises.
- If `$@` holds an object reference with **no PROPAGATE**, the reference is re-thrown unchanged.
- If `$@` is also empty, the string "Died" is used.

The `$SIG{__DIE__}` hook

Setting `$SIG{__DIE__}` installs a handler that runs when `die` fires, with the exception as its argument. The handler can inspect or replace the exception by calling `die` again with a different value. It runs **even inside eval**, which is the usual reason to want it out of the way:

```
local $SIG{__DIE__} = sub {
    die @_ if $^S;          # inside eval: do nothing special
    log_fatal(@_);          # outside eval: log before we exit
};
```

`$^S` is `undef` during compilation, true inside an `eval`, and false at the top level - use it to gate handler work that only makes sense for uncaught exceptions.

Examples

Bail out with a clean message (note the trailing newline suppresses file/line appending):

```
chdir '/var/spool' or die "can't cd to spool: $!\n";
```

Die without a newline during development so the location is appended:

```
die "unexpected empty record";
# unexpected empty record at parse.pl line 87.
```

Catch a string exception:

```
eval {
    parse($input);
};
if ($@) {
    warn "parse failed: $@";      # $@ already ends in "\n" if the
                                # die'd message did
}
```

Throw and catch a blessed exception object:

```
eval {
    die MyApp::Error->new(code => 'EIO', detail => $!);
};
if (my $err = $@) {
    if (ref($err) && $err->isa('MyApp::Error')) {
        handle($err);
    }
    else {
        die $err;                # re-raise anything unexpected
    }
}
```

Conditional propagation - catch one class, let the rest through:

```
eval { work() };
die unless $@ =~ /^recoverable:/;
recover();
```

Edge cases

- **List with two or more items** is stringified and concatenated to form the message: `die "read failed on ", $file, ": $!"`. The newline rule applies to the concatenated result.
- **Empty list** (`die;` or `die "";`) triggers the propagation path described above, not a fresh exception.
- **Reference with trailing newline inside the stringified form** is irrelevant: location is never appended to a reference, regardless of what its `""` overload produces.

- **die inside DESTROY during global destruction** can still fire `$SIG{__DIE__}` but may not propagate as expected - the interpreter is already unwinding. Keep destructors defensive.
- **Uncaught die during END block** runs remaining END blocks but sets exit status per the `$/` rule above.
- **Nested eval and \$SIG{__DIE__}**: the handler runs for *every* die, including those caught by an inner `eval`. Guard with `^S` if the handler has side effects.
- **Re-throwing the current \$@**: prefer bare `die`; over `die $@`; . The bare form triggers the propagation machinery (. . . propagated suffix, PROPAGATE hook); `die $@` does not - it treats `$@` as a fresh exception value.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `eval` - the only construct that catches a die; its block form traps exceptions into `$@`
- `warn` - same formatting rules (trailing-newline suppresses location), but writes to STDERR and does not unwind
- `caller` - capture file and line at throw time when building a structured exception object
- `exit` - use when you want a specific exit code without the exception-handling machinery
- `$@` - where the exception lands after an `eval` catches it, and the source the no-arg `die` re-raises from
- `$SIG{__DIE__}` - pre-throw hook that can inspect or rewrite the exception
- `^S` - tells a `$SIG{__DIE__}` handler whether it is running inside an `eval`

Control flow · Modules

do

Execute a block of code or run a Perl source file as if it were part of the current program.

`do` wears two unrelated hats that share only a keyword. `do BLOCK` groups statements into a single expression and returns the value of the last one - a syntactic tool, not a function. `do EXPR` treats the string as a filename and compiles-and-runs the file at runtime. Read whichever section matches the form you see; mixing them up is the most common source of confusion around this keyword.

Synopsis

```
do BLOCK                # group statements, yield last value
do EXPR                 # load and run a Perl file
do { ... } while CONDITION;  # post-test loop idiom
```

What you get back

- `do BLOCK` - the value of the last expression evaluated in the block, in whatever context the surrounding expression imposes.
- `do EXPR` - the value of the last expression evaluated in the file on success; `undef` on failure, with `$@` set to the compile error or `$!` set to the I/O error.

do BLOCK - group statements into one expression

`do BLOCK` is not a function. It is a syntactic construct that turns a brace-delimited sequence of statements into a single term whose value is the value of the last statement. Reach for it when the language wants an expression but you have several statements to run:

```
my $x = do {
    my $tmp = fetch_raw();
    $tmp =~ s/\s+\z//;
    lc $tmp;
};                                     # $x = cleaned, lower-cased value
```

When `do BLOCK` is modified by a trailing `while` or `until`, the block runs **once before** the condition is tested - this is the standard post-test loop idiom:

```
do {
    $line = <$fh>;
} while defined $line && $line !~ /^END\b/;
```

On any other statement, `while` and `until` as statement modifiers test the condition first. `do` is the exception.

next, last, redo do not work inside do BLOCK

A `do BLOCK` is **not** a loop, even when trailed by `while` or `until`. Loop-control keywords do not see it:

```
do {
    last if $done;           # WRONG - not a loop
} while $more;
```

If you need loop control, wrap the body in a bare block, which is a loopable scope:

```
{
    last if $done;
    redo if $retry;
}
```

Or use an explicit `while` loop.

return inside a bare do BLOCK

`return` in a `do BLOCK` returns from the **enclosing subroutine**, not from the block itself. That is occasionally what you want and occasionally a bug - either way, know which one you are writing:

```
sub classify {
    my $n = shift;
    my $kind = do {
        return "zero" if $n == 0;    # returns from classify(), not
→the do
        $n > 0 ? "pos" : "neg";
    };
    "$kind ($n)";
}
```

To yield a value from the block without returning from the sub, just let the last expression be the value - that is the whole point of `do BLOCK`.

do EXPR - load and run a Perl source file

`do EXPR` evaluates `EXPR`, treats the result as a filename, reads the file, compiles it, and runs it in the current package. It is similar to, but not quite the same as,

```
eval `cat stat.pl`;
```

The `do` form runs no external process, keeps the original filename for error messages, and **cannot see lexicals in the enclosing scope** - which `eval STRING` can. Like `eval STRING`, the file is reparsed on every call, so `do FILE` inside a loop is almost always wrong.

Path resolution and @INC

The filename is interpreted like this:

- Absolute paths (`/foo/stat.pl`), paths beginning with `./`, and paths beginning with `../` are used as-is.
- Any other relative path is searched along `@INC`, and on success the found path is recorded in `%INC` under the key you passed.

```
do '/etc/myapp/stat.pl';           # exact file
do './stat.pl';                   # exact file, current directory
do 'stat.pl';                     # search @INC
do 'MyApp/config.pl';             # search @INC
```

Since Perl 5.26 `.` is not in `@INC` by default, so a bare `do 'stat.pl'` no longer falls back to the current directory. If the file is not found, Perl emits the hint:

```
do "stat.pl" failed, '.' is no longer in @INC;
did you mean do "./stat.pl"?
```

Pass `./stat.pl` when you mean the current directory.

Return value and error reporting

`do` `EXPR` has three outcomes you must distinguish:

Outcome	Return	<code>\$@</code>	<code>\$!</code>
File compiled and ran	value of last expression	<code>" "</code>	unchanged
File read but did not compile	<code>undef</code>	compile error	may also be set
File could not be read	<code>undef</code>	<code>" "</code>	I/O error

Because compilation failure can incidentally set `$!` as well, **always check `$@` first**:

```

my $return;
for my $file ("/etc/myapp.rc", "$ENV{HOME}/.myapprc") {
    unless ($return = do $file) {
        warn "couldn't parse $file: $@" if $@;
        warn "couldn't do $file: $!" unless defined $return;
        warn "couldn't run $file" unless $return;
    }
}

```

The three `warn`s correspond to the three failure shapes above plus the fourth case - the file ran but its last expression was false.

Why not use `do` `FILE` for modules

For loading library code, use `require` or `use` instead of `do` `EXPR`. `require` caches through `%INC` (so a file is loaded at most once), raises an exception on failure instead of returning `undef`, and integrates with `@INC` hooks. `use` adds compile-time loading and import handling on top. `do` `EXPR` exists for configuration files and ad-hoc script inclusion - not for libraries.

Examples

Group statements into an expression:

```

my $config = do {
    open my $fh, "<", $path or die $!;
    local $/;
    <$fh>;
};
# $config = whole file as one string

```

Post-test loop with `do ... while`:

```

my $n;
do {
    print "enter a positive number: ";
    chomp($n = <STDIN>);
} until defined $n && $n =~ /\A\d+\z/ && $n > 0;

```

Read a Perl-syntax configuration file, with full three-way error handling:

```
my $cfg = do '/etc/myapp/config.pl';
if (!defined $cfg) {
    die $@ ? "config parse error: $@"
          : "config read error: $!";
}
die "config did not return a true value" unless $cfg;
```

Conditional value selection where each branch is several statements:

```
my $greeting = do {
    if ($user) {
        my $name = $user->display_name;
        "hello, $name";
    }
    else {
        "hello, stranger";
    }
};
```

Temporary localisation that must span several statements and still yield a value:

```
my $dump = do {
    local $Data::Dumper::Sortkeys = 1;
    local $Data::Dumper::Indent   = 1;
    Data::Dumper::Dumper(\%state);
};
```

Global state it touches

- `@INC` - searched when `do EXPR` is called with a relative path other than `./...` or `../...`
- `%INC` - updated with the resolved path when `do EXPR` succeeds on an `@INC` search.
- `$@` - set by `do EXPR` on compile failure; cleared on success.
- `$!` - set by `do EXPR` on I/O failure (file could not be read).
- `$0`, `__FILE__`, `__LINE__` - the currently compiling filename is set to the file being loaded while `do EXPR` parses it, so warnings and `die` messages point at the right file.

`do BLOCK` touches no globals; it is a pure syntactic construct.

Edge cases

- **`do BLOCK` is not a loop.** `next`, `last`, and `redo` inside a `do ... while` do not apply to the block. Use a bare block or a real `while` loop.
- **`return` in `do BLOCK` returns from the enclosing `sub`.** Not from the block. There is no way to return a value from a `do BLOCK` other than making it the last expression evaluated.

- **Statement modifiers are pre-test everywhere else.** `STATEMENT while COND` tests first; `do BLOCK while COND` tests after the first run. This is a deliberate language exception.
- **Relative paths without `./` search @INC.** Since 5.26, `.` is not in @INC, so `do 'stat.pl'` does not mean “current directory file” any more; pass `./stat.pl` if that was the intent.
- **do EXPR cannot see enclosing lexicals.** The file is parsed as if at the top level. If you need access to lexicals, use `eval STRING` - with the usual taint and injection caveats.
- **The file is re-parsed every call.** `do EXPR` is not cached via %INC the way `require` is. Calling it in a loop re-reads and re-compiles the file each iteration.
- **Check `$@` before `$!`.** A compilation failure can also set `$!` as a side effect; the compile error in `$@` is the one you actually want to report.
- **A file that compiles but returns a false value is ambiguous.** `unless ($ret = do $file) { ... }` fires for both “file failed” and “file ran and returned 0”. If your config file is allowed to end in a false value, use `defined` and `$@` explicitly instead of truth-testing the return.
- **do with no argument is a syntax error.** Both forms require a term: a block or an expression.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `eval` - `eval BLOCK` traps exceptions without reading a file; `eval STRING` parses arbitrary code and **can** see enclosing lexicals, unlike `do EXPR`
- `require` - load a library file with %INC caching and automatic exception on failure; the right tool for modules
- `return` - note that `return` inside `do BLOCK` returns from the enclosing subroutine, not from the block
- `@INC` - module search path consulted by `do EXPR` on relative filenames
- `%INC` - records the resolved path after a successful @INC-searched `do EXPR`
- `$@` - compile-time error from `do EXPR`
- `$!` - I/O error from `do EXPR` when the file cannot be read

Control flow

dump

Cause the running program to produce a core dump immediately.

`dump` is a pre-historic Perl operator whose original purpose was to freeze a running interpreter to a core file so that the (separately supplied) `undump` utility could turn that core file back into an executable. The resulting binary, when run, would restart the program

by jumping to LABEL via `goto`, skipping the startup cost of re-reading and re-compiling the source. That workflow has been dead for decades: `undump` was never shipped with Perl, modern operating systems make core-to-executable conversion essentially impossible, and the whole operator is now a historical curiosity. As of Perl 5.30 it must be spelled `CORE::dump()` to discourage accidental use.

There is almost no situation in modern Perl where `dump` is the right answer. If you reached for it because you want a stack trace, use `Carp::confess`. If you want to exit abnormally, use `die` or `exit`. If you want a snapshot of the program state, use a check-pointing tool at the OS level.

Synopsis

```
CORE::dump();  
CORE::dump LABEL;  
CORE::dump EXPR;
```

What you get back

Nothing. `dump` does not return: control leaves the Perl interpreter the way `abort(3)` leaves a C program. The process is terminated by the kernel's core-dump handling (SIGABRT-style), with the usual side effects: an exit status reflecting the signal, a possible core file in the current directory if the user's `RLIMIT_CORE` and the kernel's `core_pattern` allow it, and no orderly shutdown of the Perl runtime. `END` blocks do **not** run. `DESTROY` methods do **not** run. Buffered output on any filehandle is lost.

Global state it touches

None directly. `dump` bypasses the normal end-of-program machinery entirely - it does not read or write special variables, does not flush buffers, and does not run destructors. Whatever state the interpreter happens to be in at the moment of the call is what lands in the core file.

Examples

Trigger a core dump from a running program (requires `CORE::` as of 5.30):

```
CORE::dump();
```

With a label - originally, the `undump`-produced binary would resume execution here. With `undump` gone this is purely decorative; the process still just aborts:

```
RESTART:  
    # ... initialisation ...  
CORE::dump RESTART;
```

Runtime-computed label name (the `dump EXPR` form, available since Perl 5.18):

```
my $label = pick_entry_point();  
CORE::dump $label;
```

What you almost certainly want instead. For a post-mortem with a stack trace:

```
use Carp;
Carp::confess("unreachable state: $state");
```

For a clean abnormal exit with a status code:

```
die "fatal: $reason\n";      # exit via $@, runs END / DESTROY
exit 1;                     # plain exit, runs END / DESTROY
```

Edge cases

- **Spelling since 5.30:** a bare `dump` or `dump LABEL` at statement level is a compile-time error unless written as `CORE::dump` or `CORE::dump(...)`. This is deliberate - the core team made the operator hard to reach by accident.
- **LABEL is not resolved at call time:** historically the label was looked up only after `undump` reincarnated the program, so a typo would not be caught at dump-time. Today the point is moot - the program never resumes, so the label is pure documentation.
- **dump EXPR:** the expression is evaluated in scalar context and must yield a label name. It has no effect on the outcome, for the same reason.
- **Parser quirks:** `dump` has the same precedence as assignment and is exempt from the looks-like-a-function rule. `dump ("foo") . "bar"` parses with `"bar"` as part of the argument, not as a post-concatenation. In practice this only matters if you are writing `dump` at all, which you should not be.
- **Open files are lost:** any filehandle open at the moment of `dump` is gone when (in the historical workflow) the reincarnated program starts. The POD warning about “possible resulting confusion by Perl” dates from the era when someone might actually try this; today the warning is academic because nothing resumes.
- **Core files may not be produced:** whether a core file actually lands on disk depends on `ulimit -c`, the kernel’s `kernel.core_pattern`, filesystem permissions, and whether the process is running under a supervisor that captures or discards core files. `dump` succeeding in aborting the process is not the same as a core file being available afterwards.
- **Portability:** on systems without meaningful core-dump support the operator reduces to “abort the process”; see `perlport` for historical notes.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `exit` - terminate the program normally, running `END` blocks and destructors; this is what you almost certainly want instead of `dump`
- `die` - raise an exception; with no handler it terminates the program but still runs `END` blocks and flushes output

- `goto` - the control-transfer operator whose LABEL form dump was historically meant to cooperate with after reincarnation
- `Carp::confess` - stack-trace-on-exit, the modern answer for “I want to see program state at the point of failure”
- `warn` - emit a diagnostic without terminating; combine with a stack-trace module when you want information without aborting
- `perlrun` - documents the historical `-u` command-line switch that performs a dump immediately after compilation

[Arrays](#) · [Hashes](#)

each

Step through a hash or array one entry at a time.

`each` advances a per-container iterator and hands back the next entry. For a hash that entry is a (key, value) pair; for an array it is an (index, value) pair. When the iterator has walked every entry, the next call returns the empty list (or `undef` in scalar context) and then rewinds, so a fresh `while (each ...)` loop starts from the beginning. Hash order is randomised per process and per hash; array order is strictly lowest index first.

Synopsis

```
while (my ($k, $v) = each %h) { ... }
while (my ($i, $v) = each @a) { ... }
while (each %h) { ... }           # sets $_ to the key
my $k = each %h;                 # scalar context: key/index only
```

What you get back

- **List context:** a two-element list (`$key`, `$value`) for a hash, (`$index`, `$value`) for an array. On exhaustion, the empty list.
- **Scalar context:** the key (for a hash) or the index (for an array). On exhaustion, `undef`.
- **Bare `each %h` in a `while/for` condition:** tests *definedness* of the returned key/index, not truth, and assigns the key to `$_`. This is what makes `while (each %h) { print "$_\n" }` work even when a key is the empty string or `0`.

After exhaustion the iterator is implicitly reset, so the call *after* the one that returned empty restarts iteration from the top.

Global state it touches

Every hash and every array carries **its own internal iterator**. `each`, `keys`, and `values` all advance and share that one iterator - they are three views of the same cursor, not three independent loops. This is the single most important fact about `each`:

- Calling `keys %h` or `values %h` (in any context, even as a one-off count like `scalar keys %h`) **resets** the hash's iterator. A `while (each %h)` loop interrupted by `scalar keys %h` in the body will restart from the top on the next iteration - or worse, loop forever.
- **Referencing a hash in list context resets its iterator too** (e.g. `my @copy = %h`, or passing `%h` to a sub). Array iterators are *not* reset by list-context reference.
- Two loops iterating the same `%h` share one cursor. Nesting `while (each %h) { ... while (each %h) { ... } ... }` visits each entry at most once total, not $n * m$ times.
- An anonymous hash/array in the condition (`each %{ +{ a => 1 } }`) builds a *new* container every iteration, so the iterator is always fresh - the loop never terminates.
- The bare-`each` form assigns to `$_`; if you rely on `$_` elsewhere in the loop body, save it with `local`.

Every entry to a `while (each ...)` loop picks up whatever iterator position the container happens to be at. There is no way to isolate a loop's iterator from other code that touches the same container. If the loop body calls a function that might read the same hash, the safe pattern is a `foreach` loop over `keys` instead.

Examples

Basic hash iteration:

```
my %colour = (red => "#f00", green => "#0f0", blue => "#00f");
while (my ($name, $hex) = each %colour) {
    print "$name = $hex\n";
}
# order is unspecified
```

Array iteration (Perl 5.12+):

```
my @letters = ('a' .. 'e');
while (my ($i, $c) = each @letters) {
    print "$i $c\n";
}
# [0] a
# [1] b
# [2] c
# [3] d
# [4] e
```

Printing a hash in sorted order - `each` does **not** sort, so reach for `keys` plus `sort` instead:

```
for my $k (sort keys %colour) {
    print "$k = $colour{$k}\n";
}
```

Bare `each` in a `while` condition, which sets `$_` to the key every iteration (Perl 5.18+):

```
while (each %ENV) {  
    print "$_=$ENV{$_}\n";  
}
```

Safe deletion during iteration - you may delete the key *just returned* by each without disturbing the cursor:

```
while (my ($key, $value) = each %hash) {  
    delete $hash{$key} if $value < 0;    # safe  
}
```

Iterating a hash with a foreach loop when anything in the body might touch %hash (the robust pattern - snapshot the keys first):

```
for my $key (keys %hash) {  
    my $value = $hash{$key};  
    # body is free to read %hash, call subs that read it, even  
    # mutate it cautiously - our iteration list is fixed already.  
}
```

Multi-value foreach over a hash (Perl 5.36+), which makes each unnecessary for most loops and sidesteps the shared-iterator trap:

```
foreach my ($key, $value) (%hash) {  
    print "$key => $value\n";  
}
```

Edge cases

- **Iterator not reset between re-entries:** a while (each %h) loop that exits early via `last`, `return`, or `die` leaves the iterator mid-walk. The next while (each %h) on the same hash picks up where the first left off, not at the start. Call `keys %h` in void context - `keys %h; -` to force a reset before the new loop.
- **keys/values resets the iterator:** any call to `keys` or `values` on the same hash - including scalar `keys %h` inside the loop body - rewinds the cursor.
- **List-context reference resets a hash's iterator, not an array's:** `my @flat = %h` rewinds %h's cursor; the equivalent `my @copy = @a` does not touch @a's iterator.
- **Delete-during-iteration is unspecified** except for the entry most recently returned. Deleting any other key may cause entries to be visited twice or skipped entirely.
- **Add-during-iteration is unspecified** - inserting a new key mid loop may or may not be visited, and may disturb the order of entries not yet seen.
- **Anonymous container in the condition:** `each %{ +{ a => 1 } }` or `each @[ 1, 2, 3 ]` constructs a fresh container each time the condition is evaluated, so the iterator is always new and the loop never terminates.
- **Tied hashes and arrays** delegate to FIRSTKEY/NEXTKEY (hashes) or FETCH-SIZE/FETCH (arrays), so iteration order is whatever the tied class chooses and none of the guarantees above necessarily hold.

- **Definedness vs truth in the loop condition:** `while (each %h)` and `while (my $k = each %h)` both test *definedness* of the expression, so a key of `"0"` or `" "` does not terminate the loop. This is special-cased for `each` specifically.
- **Scalar expressions:** `each $href` on a scalar holding a reference was an experimental feature in 5.14 and was **removed in 5.24**. Dereference explicitly: `each %$href`, `each @$aref`.
- **Shared state across subs:** passing `%h` to a sub that calls `each %h`, `keys %h`, or `values %h` on it perturbs your caller's iterator. If you publish a hash, assume callers may break your loop; prefer `foreach (keys ...)` for robustness.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `keys` - snapshot of all keys (hash) or indices (array); **resets the iterator** `each` uses
- `values` - snapshot of all values; **resets the iterator** `each` uses
- `delete` - safe for the entry just returned by `each`, unsafe for any other during iteration
- `exists` - test key/index membership without perturbing the iterator
- `for` / `foreach` - the safer loop when the body might touch the container; combine with `keys` to snapshot first
- `sort` - pair with `keys` when you need a defined iteration order; `each` never sorts

User and group info

endgrent

Close the group database after iterating it.

`endgrent` is the cleanup half of the group-database iteration trio `setgrent` / `getgrent` / `endgrent`. It tells the C library that you are done walking `/etc/group` (or the NSS source behind it), releasing any file descriptors or cached state the library opened on your behalf. After `endgrent`, the next `getgrent` call starts a fresh iteration from the top.

Synopsis

```
endgrent;  
endgrent();
```

No arguments, no meaningful return value.

What you get back

The return value is unspecified and must not be relied on. `endgrent` is called for its side effect - closing the group database - not for a result. Treat it as a void call.

```
endgrent;                                # correct
my $ok = endgrent;                        # wrong: $ok is not a success_
→flag
```

If your platform does not implement `endgrent`, the call croaks with The `endgrent` function is unimplemented at runtime. This is effectively a compile-once check - if the function works on your system at all, it will continue to.

Global state it touches

`endgrent` operates purely on the libc-internal group-iteration state that `setgrent` and `getgrent` share. It reads and writes no Perl special variables. It does not alter `$!`: a failure path that sets `errno` in the underlying C call is not surfaced to Perl.

The iteration state is **per-process**, not per-thread. In a threaded program, one thread's `endgrent` closes the iteration for every thread

- see *Edge cases*.

Examples

The canonical iterate-and-close pattern. Always pair a `setgrent` / `getgrent` loop with `endgrent` so long-running programs do not leak the file descriptor the C library keeps open:

```
setgrent;
while (my @g = getgrent) {
    my ($name, $passwd, $gid, $members) = @g;
    print "$name ($gid): $members\n";
}
endgrent;
```

Dropping into `endgrent` mid-iteration is legal - it simply abandons whatever cursor position the library was holding:

```
setgrent;
while (my @g = getgrent) {
    last if $g[0] eq 'wheel';           # found what we wanted
}
endgrent;                               # release the handle anyway
```

Restarting iteration. `endgrent` followed by `setgrent` is the portable way to rewind:

```
setgrent;
my @first_pass = collect_groups();
endgrent;
```

(continues on next page)

(continued from previous page)

```
setgrent;
my @second_pass = collect_groups(); # starts over from the top
endgrent;
```

Defensive cleanup in a END block for programs that might die mid-walk:

```
setgrent;
END { endgrent }
while (my @g = getgrent) { ... } # any die still triggers END
```

Edge cases

- **Calling endgrent without a prior setgrent or getgrent** is harmless. The C library silently does nothing if no iteration is in progress.
- **Calling it twice in a row** is also harmless - the second call is a no-op.
- **Parser ambiguity.** endgrent takes no argument and has the precedence of a named unary operator. A bareword expression immediately to its right will be parsed as an argument and rejected:

```
endgrent or die;           # WRONG: parses as _
↪endgrent(or die)
endgrent() or die;        # right: empty parens _
↪disambiguate
```

In practice there is nothing to check the return value against, so this trap rarely bites - but the same rule applies to any chained expression.

- **Threads share the iteration.** endgrent from one thread ends iteration for all threads of the same process, because the underlying state is process-global. Do not interleave getgrent calls across threads expecting each to see every record.
- **NSS and caching.** On systems where the group database is served through NSS (nss_ldap, nss_sss, etc.), endgrent releases per-module caches that the NSS backend may hold. Long-lived daemons that walk the group database periodically should call endgrent after each walk to avoid accumulating backend state.
- **No effect on name-lookup calls.** getgrnam and getgrgid are independent of the iteration cursor. endgrent does not invalidate their results or cached data, and you do not need to call endgrent before or after them.
- **Unimplemented on some platforms.** Perl croaks The endgrent function is unimplemented if the host libc has no endgrent(3). All modern Linux, BSD, and macOS systems provide it.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `setgrent` - open or rewind the group database before walking it; always the prologue to an iteration that ends with `endgrent`
- `getgrent` - pull the next record from the iteration `endgrent` closes
- `getgrnam` - look up one group by name, independent of the iteration cursor; no `endgrent` needed
- `getgrgid` - same as `getgrnam` but keyed on numeric gid
- `endpwent` - the parallel cleanup call for the passwd database; the two iterations are independent and both need closing if you opened both

Network info

endhostent

Close the hosts database after iteration.

`endhostent` releases any resources held by the hosts database after a walk driven by `gethostent`. It pairs with `sethostent`, which opens or rewinds the database, and is a direct wrapper around the C library call of the same name. Calling it when no iteration is in progress is harmless.

Synopsis

```
endhostent;  
endhostent();
```

`endhostent` takes no arguments. Parentheses are optional.

What you get back

No useful return value. The call is made for its side effect - tearing down whatever per-process state the platform's `gethostent(3)` keeps between iterations (file descriptors on traditional `/etc/hosts` iteration, resolver sockets on NSS-backed systems).

Global state it touches

- The hosts-database iteration cursor shared by `gethostent`, `sethostent`, `gethostbyname`, and `gethostbyaddr`. After `endhostent`, the next `gethostent` call restarts from the beginning (typically after an implicit `sethostent`).
- `$?` carries the C-library `h_errno` value on systems where host lookups surface it. Most implementations of `endhostent` itself do not touch `h_errno`, but a subsequent `gethostent` can.

This state is per-process, not per-thread. Two threads iterating hosts concurrently will interleave records and neither will see the full set

- pair `sethostent` / `gethostent` / `endhostent` within one thread.

Examples

Close the database after a full walk:

```
sethostent(1);                                # 1 = keep connection open
while (my @h = gethostent()) {
    print "$h[0]\n";                          # canonical hostname
}
endhostent();
```

Always pair endhostent with sethostent in a subroutine so the iteration state does not leak to the caller:

```
sub list_hosts {
    sethostent(0);
    my @names;
    while (my ($name) = gethostent()) {
        push @names, $name;
    }
    endhostent();
    return @names;
}
```

Defensive cleanup - endhostent is safe even when no iteration is in progress, so it is fine to call unconditionally on an error path:

```
eval {
    sethostent(1);
    process_hosts();
    1;
} or do {
    my $err = $@;
    endhostent();
    die $err;
};
endhostent();
```

Using the by-name accessor object interface from Net::hostent does not change the rule - the underlying iterator is the same one endhostent closes:

```
use Net::hostent;
sethostent(1);
while (my $h = gethostent()) {
    print $h->name, "\n";
}
endhostent();
```

Edge cases

- **No-op when nothing is open:** calling `endhostent` without a prior `sethostent` or `gethostent` is defined and does nothing observable.
- **Interleaving with name / address lookups:** a `gethostbyname` or `gethostbyaddr` call in the middle of an iteration may reset the cursor on some platforms (any implementation that internally calls `sethostent(0)` for single-shot lookups). Do not mix the two styles against the same database without re-calling `sethostent` afterwards.
- **Platforms without a hosts iterator:** on systems whose resolver does not support enumeration (most modern NSS setups querying DNS only), `gethostent` returns the empty list immediately. `endhostent` is still callable and still a no-op.
- **Thread safety:** the host iterator is process-global. In a multi-threaded program, guard the `sethostent` / `gethostent` / `endhostent` trio with your own mutex.
- **Prototype / parsing:** `endhostent` has no arguments, so there is no parser-ambiguity trap. Writing `endhostent LIST` is a syntax error rather than being silently interpreted as a call with arguments.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `sethostent` - open or rewind the hosts database; the opening half of the iteration pair
- `gethostent` - read one record from the open database; returns the empty list when the iteration is exhausted
- `gethostbyname` - single-shot lookup by name; does not require `sethostent` / `endhostent`
- `gethostbyaddr` - single-shot lookup by address
- `endnetent` - same pattern for the networks database
- `endservent` - same pattern for the services database
- `$?` - receives `h_errno` from failed host lookups on platforms where the C library exposes it

Network info

`endnetent`

Close the networks database after iteration.

`endnetent` releases the resources held by the networks database - typically the `/etc/networks` file or its NSS-backed equivalent - that were opened implicitly by the first call to `getnetent`, `getnetbyname`, or `getnetbyaddr`, or explicitly by `setnetent`. After `endnetent` the iteration cursor is gone; a subsequent `getnetent` starts a fresh sweep

from the top of the database. It is a thin wrapper over the system C library's `endnetent(3)`.

Synopsis

```
endnetent;
endnetent();
```

What you get back

A system-defined true value. The C function returns `void`; Perl surfaces a placeholder so the call can stand as a statement. Do not compare the return value against anything - there is nothing useful in it.

Global state it touches

`endnetent` operates on the process-wide networks-database cursor owned by `libc`. That cursor is shared with `getnetent`, `getnetbyname`, `getnetbyaddr`, and `setnetent`; a call here invalidates the iteration state those functions depend on. No Perl special variable is touched.

Examples

Paired with `setnetent` and `getnetent` for a complete sweep of the networks database:

```
setnetent(1);                # stayopen = 1
while (my @net = getnetent()) {
    my ($name, $aliases, $addrtype, $net) = @net;
    print "$name\t$net\n";
}
endnetent();                 # release the cursor
```

Close after the handful of lookups you actually needed, so the database file does not stay open for the rest of the program:

```
setnetent(0);
my @loopback = getnetbyname("loopback");
my @link      = getnetbyname("link-local");
endnetent();
```

Safe to call when the database was never opened - `libc` treats it as a no-op:

```
endnetent();                # harmless on a fresh process
```

Edge cases

- **No arguments, list-operator precedence.** `endnetent` takes no arguments but parses as a list operator, so a stray term after it is swallowed: `endnetent $x` is `endnetent($x)` and produces a “Too many arguments” compile-time error. Write `endnetent;` or `endnetent();`.

- **Repeated calls.** Calling `endnetent` twice in a row is a no-op on the second call on every libc we target. Do not rely on it raising an error.
- **Interleaving with other DB iterators.** The networks cursor is independent of the cursors used by `getpwent`, `getgrent`, `gethostent`, `getprotoent`, and `getservent`. `endnetent` does not close those - call the matching `end*ent` for each iterator you opened.
- **Forking between `setnetent` and `endnetent`.** The cursor lives in the parent's libc state; a child inherits it at fork time but further `getnetent` calls in either process will desynchronise. Close and re-open in the child.
- **NSS backends.** On glibc the networks database routes through NSS, so the meaning of “close” depends on the configured service (files, nis, ldap, ...). User-visible behaviour - iteration restarts on the next `getnetent` - is the same; the cost of `endnetent` is not.
- **Thread safety.** The libc networks cursor is a single process-global resource. `ppperl` does not serialize access to it; a program that iterates the networks database from multiple threads must do its own locking.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `setnetent` - open the networks database and choose whether it stays open across individual lookups
- `getnetent` - read the next entry from the cursor that `endnetent` closes
- `getnetbyname` - single lookup by network name; implicitly opens the database but does not close it
- `getnetbyaddr` - single lookup by numeric address, same implicit-open, no-implicit-close rule
- `endhostent` - the hosts-database analogue; same pattern, independent cursor
- `endservent` - the services-database analogue for completeness of the `end*ent` family

Network info

`endprotoent`

Close the protocols database after iteration.

`endprotoent` is the terminator for the `getprotoent` / `setprotoent` / `endprotoent` iteration triad that walks the system protocols database (typically `/etc/protocols`). It releases whatever handle or state the system C library kept open for the iteration. You call it once the sequential walk is done; `getprotobyname` and `getprotobynumber` do not need it - they open and close the database on each call.

Takes no arguments, returns nothing useful. This is a thin wrapper over the system `endprotoent(3)` call.

Synopsis

```
endprotoent
```

What you get back

No meaningful return value. Calling it in list or scalar context yields the empty list / undef-equivalent; nothing you'd want to check. Its value is the side effect of closing the protocols database.

Examples

Walk every protocol, then close the database:

```

setprotoent(1);                                # 1 = keep the file open
↪ between calls
while (my @p = getprotoent()) {
    my ($name, $aliases, $proto) = @p;
    print "$proto\t$name\n";
}
endprotoent();

```

Short-circuit mid-walk - endprotoent still closes cleanly:

```

setprotoent(0);
while (my @p = getprotoent()) {
    last if $p[0] eq 'tcp';
}
endprotoent();

```

Paired with a direct lookup, endprotoent is unnecessary - name and number lookups manage their own state:

```

my @tcp = getprotobyname('tcp');                # no endprotoent needed

```

Edge cases

- **Calling without a prior setprotoent / getprotoent:** harmless. The system call is idempotent; closing an already-closed database is a no-op on every supported platform.
- **No arguments accepted:** endprotoent \$x is a syntax error - the built-in has arity zero.
- **Not reentrant:** the protocols database is a single global resource inside the C library. Two concurrent iterations in the same process step on each other's state. Use per-thread or per-subprocess walks if you need parallelism.
- **Platform quirks:** behaviour is whatever the host system's endprotoent(3) does. See perlport for the list of platforms where the network-database routines diverge or are absent entirely.

- **Stubbed on platforms without `/etc/protocols`:** on systems where the C library has no protocols database, the call is a no-op rather than an error.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `setprotoent` - open the protocols database and optionally keep it open across `getprotoent` calls; pair with `endprotoent` to bracket a walk
- `getprotoent` - read the next entry during an iteration; drives the loop that `endprotoent` terminates
- `getprotobyname` - single-shot lookup by protocol name; self-contained, no `endprotoent` needed
- `getprotobynumber` - single-shot lookup by protocol number; same independence from the iteration triad
- `endservent` - the analogous terminator for the services database; same idiom, different table
- `endhostent` - the analogous terminator for the hosts database

User and group info

endpwent

Close the passwd-database iterator opened by `getpwent` or `setpwent`.

`endpwent` is the companion teardown call for a passwd-file walk. After iterating the password database with repeated `getpwent` calls, `endpwent` releases whatever per-process state the C library allocated to hold the iteration cursor (an open file descriptor on `/etc/passwd`, a cached buffer for NSS, a connection to `nscd` / `sssd`, an LDAP session, depending on the host's name-service configuration). It takes no arguments and wraps the system `endpwent(3)` call directly.

Synopsis

```
endpwent
```

What you get back

A single boolean value. The return is rarely checked - on every platform PetaPerl supports, `endpwent` cannot meaningfully fail, and real code treats it as a void call.

Global state it touches

`endpwent` mutates no Perl-visible variable. The state it clears is entirely inside the C library: the per-process passwd iterator, shared with `getpwent` and `setpwent` (and, on some systems, with `getpwnam` / `getpwuid` when those happen to reuse the same underlying stream). Calling `endpwent` in one thread affects every other thread in the same process - the iterator is per-process, not per-thread.

Examples

Standard iterate-then-close idiom:

```
setpwent;
while (my @ent = getpwent) {
    my ($name, undef, $uid) = @ent;
    print "$uid\t$name\n";
}
endpwent;
```

Ensure the iterator is closed even if the loop exits early:

```
setpwent;
eval {
    while (my @ent = getpwent) {
        my ($name, undef, $uid, $gid) = @ent;
        die "found it\n" if $uid == $target;
    }
};
endpwent;      # always run, regardless of how the loop ended
```

Pair with a `local` guard so nested code cannot leave the iterator half-walked:

```
sub each_user {
    my $cb = shift;
    setpwent;
    my $guard = Guard::guard { endpwent }; # or a manual eval-and-
    ↪ rethrow
    while (my @ent = getpwent) { $cb->(@ent) }
}
```

`endpwent` with no prior iteration is harmless:

```
endpwent;      # no-op on a fresh ↵
↪ process
```

Edge cases

- **No arguments, no parentheses needed.** `endpwent` is a named unary operator with zero arguments; write it bare. `endpwent ( )` is accepted but adds nothing.
- **Idempotent.** Calling `endpwent` twice in a row, or calling it before any `getpwent` / `setpwent`, is defined as a successful no-op on every supported platform.

- **Does not invalidate records you already fetched.** Arrays and hashes returned by earlier `getpwent` / `getpwnam` / `getpwuid` calls are plain Perl data and remain valid after `endpwent` - only the iterator cursor is closed.
- **Thread safety.** The iterator is process-wide. If two threads are walking the passwd database simultaneously, `endpwent` closes the shared cursor for both - neither thread will get the full record set. Use a lock around the whole `setpwent` / `loop` / `endpwent` block when iterating from threads.
- **Name-service backends.** On hosts using LDAP, NIS, `sssd`, or any other non-flat-file backend, `endpwent` may release a network handle. A long-running daemon that iterates the passwd database periodically should call `endpwent` at the end of each pass rather than leaking one handle per pass.
- **Not a forced cache flush.** `endpwent` closes the iterator; it does **not** flush the C library's `getpwnam` / `getpwuid` caches. A subsequent `getpwnam` may still return a cached record.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `setpwent` - rewind or open the passwd-database iterator before a walk; pairs with `endpwent` as the open/close bracket
- `getpwent` - fetch the next record from the iterator that `endpwent` closes
- `getpwnam` - direct lookup by login name, independent of the iterator in most implementations
- `getpwuid` - direct lookup by numeric UID, same independence note as `getpwnam`
- `endgrent` - analogous teardown for the group-database iterator; the two are almost always called together

Network info

endservent

Close the services database after a walk with `getservent`.

`endservent` is the teardown half of the services-database iteration triplet. After `setservent` opens the database (typically `/etc/services` on Linux) and a sequence of `getservent` calls walks it entry by entry, `endservent` releases the underlying handle and any buffered state the C library is holding. It wraps the POSIX `endservent(3)` call one-to-one.

Synopsis

```
endservent;  
endservent();
```

What you get back

Nothing useful. The function is called for its side effect; it does not return a meaningful value. Do not test the return value.

Global state it touches

- The system-wide services-database iterator opened by `setservent` or implicitly by the first `getservent` / `getservbyname` / `getservbyport` call. `endservent` closes that iterator.
- `$!` is **not** meaningfully set by this call; the POSIX contract for `endservent(3)` defines no error return.

Examples

Paired with `setservent` and `getservent` to enumerate every entry in `/etc/services`:

```
setservent 0;
while (my @e = getservent) {
    last unless @e;
    my ($name, $aliases, $port, $proto) = @e;
    print "$name/$proto = $port\n";
}
endservent;
```

Close a walk early. The database was opened implicitly by the first `getservent`; `endservent` is still the right way to release it:

```
while (my @e = getservent) {
    last if $e[0] eq 'http';
}
endservent;
```

Defensive cleanup around a block that may leave the iterator open. Harmless when no iteration is in progress:

```
eval {
    setservent 1;
    scan_services();
    1;
} or do {
    endservent;           # always release the handle
    die $@;
};
endservent;
```

Edge cases

- **Calling without a preceding `setservent` or `getservent`** is harmless. The underlying C call tolerates a no-op close.
- **Calling twice** is harmless for the same reason. Repeated `endservent` does not report an error.
- **Does not affect the name and port lookup functions** - `getservbyname` and `getservbyport` open and close their own short-lived handles and are unaffected by the iterator state that `endservent` manages.
- **No prototype, no arguments.** `endservent $x` is a compile-time error. Use `endservent;` or `endservent();`.
- **Not portable beyond Unix-like systems.** On platforms where the C library lacks `endservent(3)`, the call croaks with `endservent not implemented`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `setservent` - open the services database and, with a non-zero `STAYOPEN`, keep the handle across subsequent lookups
- `getservent` - read the next entry from the open services database; pair every iteration loop with `endservent`
- `getservbyname` - one-shot lookup by service name; does not interact with the iterator `endservent` closes
- `getservbyport` - one-shot lookup by port number; same independence from the iterator
- `endprotoent` - the protocols-database analogue, same shape and same rules
- `endhostent` - the hosts-database analogue

I/O

`eof`

Test a filehandle for end-of-file.

`eof` reports whether the next read on a filehandle will return end-of-file. The three call forms look similar but mean different things: `eof FILEHANDLE` asks about a specific handle, bare `eof` asks about the handle most recently read from, and `eof()` with empty parentheses asks about the virtual ARGV stream fed by `<>`. The forms are not interchangeable - picking the wrong one is the classic bug with this built-in.

Synopsis

```
eof FILEHANDLE
eof()
eof
```

What you get back

1 if the next read on the handle would return end-of-file, or if the handle is not open. 0 otherwise. `eof` never blocks waiting for more input on a regular file, but on a pipe or terminal it may have to read and push back one character to decide - see *Edge cases*.

Global state it touches

- `ARGV` - the magical filehandle backing the `<>` operator. `eof()` examines and may advance it; `eof` inside a `while (<>)` loop queries it indirectly.
- `@ARGV` - the list of file names `<>` iterates over. `eof()` called before `<>` has been used triggers `@ARGV` inspection and, if empty, arranges for `STDIN` to be read.
- `$.` - the current input line number, attached to the handle `eof` queries. Not written by `eof` itself, but relevant in the idioms below where `close ARGV` resets it at file boundaries.

The three forms, spelled out

`eof FILEHANDLE` - asks about that one handle. `FILEHANDLE` may be a bareword, a scalar holding a handle, or an expression evaluating to one. Returns true for a closed or never-opened handle.

`eof` (no parentheses, no argument) - asks about the *last file read*. Inside a `while (<>)` { ... } loop this means “the current `ARGV` file” - the physical file currently being streamed, not the whole `<>` sequence. `eof` becomes true at every file boundary, once per input file.

`eof()` (empty parentheses) - asks about the `<>` stream as a whole. It becomes true only at the end of the *last* file in `@ARGV`. If `<>` has not been used yet, `eof()` will examine `@ARGV`, fall back to `STDIN` when `@ARGV` is empty, and may actually open the first file to answer the question.

Bare `eof` and `eof()` look almost identical and do completely different things. The parentheses matter.

Examples

Check a specific handle before the next read:

```
open my $fh, "<", "data.txt" or die $!;
while (not eof $fh) {
    my $line = <$fh>;
    # process $line
```

(continues on next page)

(continued from previous page)

```
}  
close $fh;
```

The classic while (<>) idiom, resetting \$. at each file boundary. Bare eof (no parens) is what you want here - it fires once per input file:

```
while (<>) {  
    next if /\s*#/;  
    print "$.\t$_";  
} continue {  
    close ARGV if eof;           # not eof() - resets $. per file  
}
```

Insert a marker before the last line of the last file. eof() with parentheses catches the end of the *entire* <> stream, not each file:

```
while (<>) {  
    if (eof()) {  
        print "-----\n";  
    }  
    print;  
    last if eof();              # needed when reading from a terminal  
}
```

Guard against a closed handle. eof on a closed or never-opened handle returns true without warning:

```
my $fh;                                # never opened  
if (eof $fh) {  
    warn "handle not ready";  
}
```

Edge cases

- **eof before any read returns false, not true.** Bare eof with no prior read has no “last file read” to refer to and Perl treats it as false. Writing if (eof) { ... } at the top of a program does not mean “if there is no input” - use eof() for that.
- **eof() before <> is used has side effects.** It examines @ARGV, and if @ARGV is empty it arranges for <> to read from STDIN. A stray eof() at program start can therefore silently bind the program to STDIN.
- **eof() after <> has exhausted its input** assumes you are starting another @ARGV pass. If @ARGV is empty at that point, the next <> reads STDIN. This is usually not what you want; set @ARGV explicitly or test the bare-eof form instead.
- **eof on a terminal consumes a character.** The implementation reads one byte and ungetc's it to tell whether more input exists. On terminals the pushback may not survive after end-of-file is reached - do not call eof FH on a terminal after you have already hit EOF there, or the handle can lose its end-of-file condition.

- **Closed or never-opened handle** - `eof FILEHANDLE` returns true. There is no way to distinguish “closed” from “at EOF” using `eof` alone; check `fileno` or track the open state yourself if that matters.
- **`eof (ARGV)` equals bare `eof` inside `while (<>)`**. Both query the current ARGV file. Prefer bare `eof` for brevity and to make the parenthesised `eof()` form visually distinct.
- **You rarely need `eof` at all**. The input operators return `undef` at end-of-file, so `while (my $line = <$fh>) { ... }` and `while (<>) { ... }` handle termination without any `eof` test. Reach for `eof` only when you need to act *at* the boundary - last line, per-file reset, final-file marker.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `readline` - read a line from a handle; returns `undef` at end-of-file, which replaces most uses of `eof`
- `read` - block read of a fixed byte count; returns `0` at end-of-file
- `getc` - read a single character; returns `undef` at end-of-file
- `close` - close a handle; the common partner of `eof` in the `close ARGV if eof` per-file-reset idiom
- `ARGV` - the magical handle fed by `<>` that `eof` and `eof()` query

Control flow

eval

Run a piece of Perl code with any fatal error trapped instead of killing the program.

`eval` is Perl’s exception-catching mechanism and its runtime code-loader in one. It comes in two real forms: **block `eval`** (`eval { ... }`), which catches exceptions thrown by already-compiled code, and **string `eval`** (`eval EXPR`), which compiles and runs a string of Perl source at runtime. Either way, a `die` that would otherwise terminate the program becomes an assignment to `$@` and a normal return.

Synopsis

```
eval BLOCK           # catch exceptions in compiled code
eval EXPR            # compile and run source at runtime
eval                # short for: eval $_
```

What you get back

The value of the last expression evaluated inside the mini-program, or the value handed to an explicit `return`. The evaluation happens in the same context as the surrounding `eval` - void, scalar, or list - so `wantarray` inside an `eval` reports the caller's context, not "some eval context."

If anything inside raises an exception (`die`, a division by zero, a string-eval syntax error, any other trappable fatal), `eval` returns `undef` in scalar context and the empty list in list context, and the exception lands in `$@`:

```
my $n = eval { risky() };    # undef on failure, $@ holds the error
if ($@) { ... }
```

On success, `$@` is set to the empty string. Every completed `eval` touches `$@` - there is no "leave it alone" option. A control-flow operator that bypasses the normal exit (`last`, `goto` out of the block) can leave `$@` unchanged, which is a corner case rather than a feature to rely on.

Global state it touches

- `$@` - written on every normal exit of the `eval`: the thrown exception on failure, the empty string on success. Read to determine whether the `eval` caught something.
- `$^S` - reflects whether Perl is currently inside an `eval` (or `try`). `undef` during parsing, true while executing inside an `eval/try` block, false at the top level. Read primarily by `$SIG{__DIE__}` handlers that want to act differently for caught vs. uncaught exceptions.
- `$SIG{__DIE__}` - fires on **every** `die`, including those that an enclosing `eval` is about to catch. Use `local $SIG{__DIE__}` inside an `eval` to suppress a globally installed hook for the duration of the trap.
- `$SIG{__WARN__}` - unaffected by `eval`. Warnings emitted inside the block still go through the normal warning path; `eval` does not redirect them into `$@`.
- `$_` - the source of the no-argument form (`eval` with nothing after it is exactly `eval $_`).

Block eval vs. string eval

The two forms look similar but do very different things.

- **Block eval** (`eval { ... }`) is parsed and compiled together with the surrounding program. At runtime it is just a marker that says "catch any `die` thrown from inside this block." Because the code is already compiled, block `eval` is cheap and safe - the compiler already checked it for syntax errors. This is the form to reach for whenever the goal is exception handling.
- **String eval** (`eval EXPR`) evaluates `EXPR` in scalar context, then parses and executes the resulting string as if it were a block in the lexical scope of the surrounding code. The parse happens every time the `eval` is reached. A syntax error in the string becomes a runtime `$@`, not a compile-time failure of the enclosing program. This is the form for loading code decided at runtime - optional features,

user-supplied expressions, dynamically constructed subs - and it carries the costs and hazards that implies.

Both forms run in the lexical scope of the surrounding code: outer `my` variables are visible, and package variable assignments, subroutine definitions, and format definitions made inside the `eval` persist after it returns.

The `$@` contract

After an `eval` completes normally, `$@` is always one of:

- The empty string - the block ran to completion with no exception.
- A string ending in a newline - a die "message\n" or a string exception the callee already terminated.
- A string with appended location (... at FILE line N.\n) - a die "message" without a trailing newline, or a runtime error Perl itself raised.
- A reference, typically a blessed exception object - `die` called with a reference.

Check `$@` immediately after the `eval` and nowhere else. Any method call, `ref` test, `isa` check, or regex match done during handling can itself invoke an `eval` internally (for instance, during autoloading or overload resolution) and overwrite `$@` before you finish inspecting it. Copy into a lexical first:

```
eval { work() };
if (my $err = $@) {           # safe snapshot
    if (ref $err && $err->isa('MyApp::Error')) { handle($err) }
    else { die $err }
}
```

Nested `eval` and local `$@`

The outer `eval` sees whatever `$@` holds when the outer block finishes - not what it held partway through. If inner code has its own `eval`, that inner `eval` sets `$@` (to the empty string on success), which can clobber an error the outer code wanted to preserve. The guard is `local $@` around the inner work:

```
eval {
    # ... some work that might fail ...
    {
        local $@;           # shield outer $@
        eval { optional_step() };
        # ignore whatever optional_step did to $@
    }
    die "still needed\n" if $condition;
};
warn "outer caught: $@" if $@;
```

Before Perl 5.14, `$@` was assigned before localized variables were restored, so code that wanted to filter errors had to stage the value through a temporary. On 5.14+ (which 5.42 inherits) the order is fixed and the temporary is unnecessary.

String eval security

String eval runs whatever is in the expression. If any part of that string came from untrusted input - a file, a network peer, a user form - the eval hands the attacker the full Perl interpreter. Rules of thumb:

- Never pass user input through eval `EXPR` unmodified. “It’s only a number” is not a safeguard - `"1; system 'rm -rf /'"` is also only a string.
- If runtime-chosen code is genuinely needed, constrain the input against a strict allow-list *before* interpolation, and prefer data-driven dispatch (a hash of coderefs) over generating source.
- Floating-point values interpolated into source are fragile. Under `use locale` the decimal separator may not be a dot, and values like "NaN" or "Infinity" stringify to letters that the parser reads as barewords, not numbers.
- Source filters activated inside a string eval leak out into the surrounding file scope (unless `evalbytes` is used). This is a long-standing quirk, not a feature.

Block eval has none of these problems - the code was compiled with the rest of the program.

BEGIN inside string eval

`BEGIN { ... }` blocks embedded in a string eval execute immediately, before the rest of the eval’d code runs, matching their behaviour in a normal compilation unit. Set `${^MAX_NESTED_EVAL_BEGIN_BLOCKS}` to 0 to make any embedded BEGIN throw instead:

```
local ${^MAX_NESTED_EVAL_BEGIN_BLOCKS} = 0;
eval $untrusted;                        # any BEGIN inside $untrusted dies
```

This does not make a string eval safe - it only removes one specific attack on compile-time side effects.

try/catch

Under `use feature 'try'` (stable since 5.40, available in 5.42), the `try/catch/finally` construct provides dedicated exception-handling syntax:

```
use feature 'try';

try {
    risky();
}
catch ($e) {
    warn "caught: $e";
}
finally {
    cleanup();
}
```

Differences from `eval { ... }; if ($@) { ... }`:

- The caught value is bound to a fresh lexical `$e` (or whatever name is given), not to `$@`. No need for the `my $err = $@` snapshot dance.
- `$@` is not set as a side effect - the `try` block has no global after-effect on it.
- `try` is not itself a loop, but control-flow operators behave analogously to `eval BLOCK`: `return` inside `try` returns from the enclosing sub, and `next/last/redo` target an outer loop.
- `finally` runs on every exit path and cannot itself use `return/next/last/redo`. It is marked experimental in 5.42 and emits a warning under `experimental::try`.

Reach for `try/catch` in new code. The `eval-plus-$@` idiom remains correct and is still the right tool whenever the surrounding code must run on older Perls or must not require the feature guard.

Examples

Trap a runtime error and keep going:

```
my $q = eval { $x / $y };
if ($@) { warn "division failed: $@"; $q = 0 }
```

Detect whether a feature is available without aborting at startup:

```
my $have_socket = eval { socket(my $s, PF_INET, SOCK_STREAM, 0); 1 }
→;
die "no sockets on this build" unless $have_socket;
```

Load a module whose absence should be tolerated:

```
my $json = eval { require JSON::XS; JSON::XS->new } //
do { require JSON::PP; JSON::PP->new };
```

Catch a blessed exception, re-throw anything else:

```
eval {
    parse($input);
};
if (my $err = $@) {
    if (ref($err) && $err->isa('MyApp::ParseError')) {
        report($err);
    }
    else {
        die $err;
    }
}
```

Nested `eval` with `local $@` so an inner optional step does not erase an outer diagnostic:

```
eval {
    do_main_work();
    { local $@; eval { maybe_cleanup() } } # ignore cleanup errors
```

(continues on next page)

(continued from previous page)

```
die "main still failed\n" if $something_else_wrong;
};
warn $@ if $@;
```

Suppress a globally installed `$SIG{__DIE__}` hook for the duration of a private trap:

```
eval {
    local $SIG{__DIE__};          # don't trigger hooks while we_
    ↪probe
    $answer = $x / $y;
};
warn $@ if $@;
```

A `$SIG{__DIE__}` hook that logs only uncaught exceptions, using `^S` to tell the two cases apart:

```
$SIG{__DIE__} = sub {
    return if ^S;                # inside eval/try - let it propagate
    log_fatal($_);               # top level - record before exit
};
```

Runtime-built dispatch (string eval used deliberately, input is not user-supplied):

```
for my $op (qw(add sub mul div)) {
    eval "sub ${op}_of { $_[0] "
        . { add=>'+', sub=>'-', mul=>'*', div=> '/' }->{$op}
        . " $_[1] }";
    die $@ if $@;
}
```

Edge cases

- **No argument** - `eval`; is `eval $_`. Rarely what you want; write `eval { ... }` or `eval EXPR` explicitly.
- **Final semicolon** - may be omitted from the end of `EXPR` or `BLOCK`. `eval "1+2"` and `eval "1+2;"` are equivalent.
- **eval BLOCK is not a loop** - `next`, `last`, and `redo` cannot leave or restart the block. They look for an enclosing loop and will croak if there is none.
- **return inside eval BLOCK** returns from the *enclosing subroutine*, not from the `eval` - the `eval` block is not a function body. To produce a value from the block, put the value as the final expression.
- **Warnings are not trapped**. `warn` inside an `eval` still writes to `STDERR`. Route them into `$@` only if you set `$SIG{__WARN__}` to do so yourself.
- **Control flow out of eval** - a `goto`, `last`, or `next` that jumps out of the block skips the normal `eval` exit and therefore may leave `$@` untouched. Do not build logic that depends on this.

- **Quotes on EXPR - `eval $x` vs. `eval "$x"`** behave identically: both run the code contained in `$x`. The double quotes add visual noise, not semantics.
- **`eval '$x'` vs. `eval { $x }`** - both return the value of `$x`. The block form is compiled with the surrounding program; prefer it unless the code genuinely needs to be constructed at runtime.
- **`eval ' ' inside the DB package`** - does not see the usual surrounding lexical scope but the scope of the first non-DB caller. Only debugger authors hit this.
- **XS load failures** - some binary-interface problems during `require` of an XS module are fatal even inside `eval`, unless `$ENV{PERL_DL_NONLAZY}` is set. Version skew between the running interpreter and a pre-built `.so` is the common case. `pperl` does not load XS; the caveat applies when `eval'd` code delegates loading to a classic `perl5`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `die` - the counterpart: throws the exception that `eval` catches. The trailing-newline rule on the message shapes what lands in `$@`.
- `do` - `do FILE` reads and evaluates a file the way `string eval` reads and evaluates a string; useful for configuration files written in Perl.
- `return` - inside `eval BLOCK` returns from the enclosing sub, not from the block; use a trailing expression to hand a value back from the `eval` itself.
- `require` - commonly wrapped in `eval` to probe for optional modules without aborting at compile time.
- `evalbytes` - string `eval` that treats its argument as a byte string and allows source filters to work normally; use when byte-level semantics matter.
- `$@` - where a caught exception lands, cleared to the empty string on successful `eval`. The error family page also covers the classic `$@`-clobbering hazard.
- `$_^S` - tells `$SIG{__DIE__}` and `$SIG{__WARN__}` handlers whether an `eval` or `try` is currently active.
- `$SIG{__DIE__}` - pre-throw hook; fires even for exceptions an enclosing `eval` is about to catch, which is the usual reason to `local`-ize it for the duration of a trap.

Control flow

`evalbytes`

Compile and run a string of Perl source, forcing the source to be interpreted as **bytes** rather than characters.

`evalbytes` is the byte-oriented twin of string `eval`. It takes an EXPR (or `$_` when EXPR is omitted), treats the string as a raw byte stream, and parses it as Perl source. The parse is insulated from the caller's use of `utf8` pragma: whatever encoding rules the string itself

declares win. Reach for it when the code you are about to compile was read from disk, the wire, or a build step, and you need the parser to see exactly the bytes you handed it.

Synopsis

```
evalbytes EXPR
evalbytes
CORE::evalbytes EXPR      # when the feature is not enabled
```

What you get back

In scalar context, the value of the last expression evaluated; in list context, a list. On a compile or run-time error, `undef` (or an empty list) and `$@` is set to the error message; `$@` is the empty string on success. Return shape matches string `eval` exactly - the only difference is how the source text itself is decoded.

```
my $result = evalbytes $source;
die $@ if $@;
```

How it differs from string eval

Two concrete behaviours distinguish `evalbytes` from `eval EXPR`:

- **The source is parsed as bytes.** Every octet in the string is a separate input character to the lexer, regardless of whether the string is internally marked as `utf8`. A multi-byte UTF-8 sequence in the source shows up to the parser as the individual bytes, not as the decoded code point.
- **Code points above 255 are a hard error.** If the string contains any character whose ordinal exceeds `0xFF`, it cannot be a byte stream and the compile fails with `Wide character in eval` stored in `$@`. Downgrade with `utf8::encode` first if the source currently holds decoded text.

`use utf8` and `no utf8` **inside** the evaluated code have their usual effect: the pragma switches on once the parser reaches it and affects the remainder of that compilation. It is the source's outer framing that is fixed to bytes, not what the source itself declares.

Enabling the name without `CORE::`

`evalbytes` is feature-gated. To call it as a bareword:

```
use feature 'evalbytes';
```

A `use v5.16` (or higher) declaration enables the `evalbytes` feature automatically, along with the rest of the `:5.16` feature bundle. Without either, the unqualified name is not recognised and you must write `CORE::evalbytes` explicitly - useful inside generated code or one-liners that do not carry a version declaration.

```
use v5.16;
my $n = evalbytes '1 + 2';      # 3
```

```
# no feature, no version bundle:
my $n = CORE::evalbytes '1 + 2'; # 3
```

Global state it touches

- `$@` - cleared before the compile, then set to the error message on failure or the empty string on success. Same contract as `eval`.
- `$_` - read as the source text when `EXPR` is omitted.

Source filters activated *inside* the evaluated code apply to the code itself, exactly as they would inside a string `eval`. The filter sees the same bytes the parser sees.

Examples

Evaluate a literal byte string - no surprises, no encoding choices to make:

```
use feature 'evalbytes';
my $n = evalbytes 'my $x = 40; $x + 2'; # 42
```

Evaluate source read from a file that is known to be UTF-8 encoded but whose declared pragma inside the source is what should govern parsing:

```
use feature 'evalbytes';
open my $fh, '<:raw', 'script.pl' or die $!;
my $src = do { local $/; <$fh> };
my @values = evalbytes $src;
die "compile failed: $@" if $@;
```

Reading with `:raw` keeps `$src` a byte string. `evalbytes` then feeds those bytes to the parser verbatim; a `use utf8;` line at the top of the script will do its own work once the parser reaches it.

Compare with string `eval` when the caller has `use utf8`:

```
use utf8;
use feature 'evalbytes';

my $src = "length('caf\xc3\xa9')"; # bytes for 'café' in UTF-8

print eval $src;          # 3 (characters: c, a, f, é)
print "\n";
print evalbytes $src;     # 5 (bytes:      c, a, f, 0xC3, 0xA9)
print "\n";
```

Under `use utf8` the caller's own literal `"length('caf\xc3\xa9')"` is a **character** string to `eval`, so `length` inside the evaluated code sees one `é` character. `evalbytes` forces the same source to be parsed as the byte sequence `c a f 0xC3 0xA9`, so `length` reports the byte count.

Catching a wide-character error:

```
use feature 'evalbytes';
my $src = "print \x{2603}";           # contains U+2603 SNOWMAN
evalbytes $src;
warn "rejected: $@" if $@;           # "Wide character in eval ..."
```

Omitting EXPR - source comes from \$_:

```
use feature 'evalbytes';
for ('1+1', '2*3', '10-7') {
    my $r = evalbytes;
    print "$_ = $r\n";
}
```

Edge cases

- **Wide character in source:** any code point above 0xFF aborts the compile and leaves Wide character in eval in \$@. Run `utf8::encode` on the source (or read the file in `:raw`) before calling.
- **use utf8 is local to the evaluated code.** It affects string literals and identifiers inside the evaluated text once the parser reaches the pragma line. It does **not** change how `evalbytes` frames the outer source - that framing is always “bytes”.
- **Feature not enabled:** without `use feature 'evalbytes'` and without a `use v5.16+` bundle, the bareword is unrecognised. Write `CORE::evalbytes` to bypass the feature gate.
- **evalbytes with no argument and \$_ undef:** compiles an empty string, which succeeds and returns `undef` in scalar context, the empty list in list context, with \$@ empty.
- **Source filters** inside the evaluated code see the byte stream. A filter that expects decoded characters will misbehave; write filters for `evalbytes` in terms of bytes.
- **Lexicals and package state:** same rules as string `eval`. The evaluated code shares the caller’s current package and sees the caller’s in-scope lexicals; its own `my` variables vanish when the eval finishes.
- **die inside evalbytes:** caught the same way as for `eval`. The thrown value lands in \$@; execution continues after the `evalbytes` call.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `eval` - the character-mode counterpart; parses EXPR according to the caller’s Unicode framing, so `use utf8` in the caller changes how the source is interpreted
- `die` - throw an exception from within evaluated code; sets \$@ just like any other run-time error

- `do` - compile and run a file; takes its source from disk rather than a string, and is not affected by the caller's `utf8` framing
- `require` - load and compile a file once; the standard tool when the source lives on disk and should only run on first reference
- `utf8` - the pragma controlling character vs byte semantics of string literals; `evalbytes` exists precisely to pin the outer framing to bytes regardless of this pragma
- `$@` - the error slot populated on `evalbytes` failure, cleared on success

Processes

`exec`

Abandon this program and run another in the same process.

`exec` replaces the current process image with another program. It does **not** return on success - execution continues in the new program, with the same process id, open file-handles subject to the `close-on-exec` flag, and whatever environment was in effect. It returns false (and sets `$!`) only when the target could not be launched *and* it was executed directly rather than through the shell. Reach for `system` when you want the child to run and control to come back.

Synopsis

```
exec LIST
exec PROGRAM LIST
exec { PROGRAM } LIST
```

What you get back

Nothing, on success - the process has been replaced and this Perl interpreter is gone. On failure `exec` returns false and leaves the reason in `$!`. Failure is only detectable for the direct-exec path; when the argument is handed to the shell, a missing program is reported by the shell itself and `exec` still counts as having “succeeded” in launching `/bin/sh`.

```
exec '/usr/bin/vi', $file
    or die "couldn't exec vi: $!";
```

Because `exec` normally does not return, Perl emits a warning under `use warnings` if it appears in void context followed by another statement that is not `die`, `warn`, or `exit`. Either handle the failure inline with `or die`, or wrap the call in a block to silence the warning:

```
exec('foo') or die "couldn't exec foo: $!";
{ exec('foo') }; die "couldn't exec foo: $!";
```

Shell or no shell - the single-string rule

exec picks between two launch mechanisms based on the shape of LIST:

- **Two or more arguments** → exec calls `execvp(3)` directly with the list. No shell is involved, no metacharacter interpretation, no word splitting.
- **Exactly one argument** → exec scans that single string for shell metacharacters. If any are found, the whole string is passed to `/bin/sh -c` for parsing. If none are found, the string is split on whitespace and handed to `execvp` directly.

The rule matters for both correctness and security. A one-argument string containing a space but no metacharacters is split into words and executed directly - which is often what you want. A one-argument string containing `|`, `>`, `$`, `;`, glob characters, or quotes goes through the shell, and the shell will happily expand wildcards, interpret variables, and split on `$IFS`.

```
exec '/bin/echo', 'Your arguments are:', @ARGV; # direct execvp
exec "sort $outfile | uniq";                    # pipeline → shell
exec "ls /tmp";                                # no metachars →
↳split + execvp
```

To **force** the no-shell path even with a single logical command, use the indirect-object form described next.

Indirect-object form - set `argv[0]`, skip the shell

Placing a block or scalar before LIST (with no comma) tells exec two things at once: run *this* program, and pass *those* arguments as argv. The first element of LIST becomes `argv[0]` - the name the child sees for itself - which need not match the program path.

```
exec { '/bin/echo' } 'echo', 'hi'; # argv[0] is "echo"
my $shell = '/bin/csh';
exec $shell '-sh';                 # login-shell convention:
↳argv[0] = "-sh"
exec { '/bin/csh' } '-sh';         # same, more direct
```

The indirect-object form always treats LIST as a multi-valued list, even if it has only one element. That disables the metacharacter scan entirely:

```
my @args = ('echo surprise');
exec @args; # one element → shell scan → /bin/sh -c
↳'echo surprise'
exec { $args[0] } @args; # always direct execvp, "echo surprise" is
↳argv[0]
```

In the second form there is no program on disk literally named `echo surprise`, so `execvp` fails, `exec` returns false, and `$!` holds the reason. That's the safe behaviour: the caller gets to decide what to do, rather than the shell running something unexpected.

Global state it touches

- `$!` - set on failure to the `errno` from the underlying `execvp(3)` call. Only meaningful when `exec` actually returns.
- `$?` - left unchanged by `exec` itself, but a failed shell-mediated attempt may leave the shell's exit status visible to a wrapping `system`.
- The process's open filehandles, environment (`%ENV`), current working directory, `umask`, signal dispositions, and resource limits all carry into the replacement program - `exec` replaces the program, not the process.

Examples

Run a program with an explicit argument list, direct `execvp`:

```
exec '/usr/bin/git', 'status', '--short'
or die "exec git: $!";
```

Hand a command line to the shell deliberately, using metacharacters:

```
exec "find . -name '*.tmp' -print0 | xargs -0 rm"
or die "exec pipeline: $!";
```

Set a distinctive `argv[0]` so the child's `ps` line identifies its role:

```
exec { '/usr/local/bin/worker' } 'worker[queue=email]', @flags
or die "exec worker: $!";
```

Fork-and-exec - the classic pattern for launching a child without replacing yourself:

```
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    exec '/usr/bin/gzip', '-9', $file
    or die "exec gzip: $!";
}
waitpid $pid, 0;
```

Guarantee the no-shell path when the argument list was assembled from data:

```
my @cmd = build_command();          # may contain one element or
many
exec { $cmd[0] } @cmd
or die "exec $cmd[0]: $!";
```

Edge cases

- **Never returns on success.** Any code after a bare `exec` is reachable only on failure. Under use `warnings`, a following statement other than `die`, `warn`, or `exit` triggers a warning - Perl's hint that you probably meant `system`.
- **Single-argument trap.** `exec $cmd` where `$cmd` came from user input is a shell-injection vector if `$cmd` contains metacharacters. Use the indirect-object form or pass a pre-split list.

- **Output buffering.** Perl attempts to flush handles opened for output before handing control to the new program, but this is not guaranteed on every platform. If you have pending output on an unflushed handle, set `$|` on the selected handle - or call the `autoflush` method on each `IO::Handle` - before `exec`.
- **END blocks and DESTROY.** `exec` does **not** run END blocks, and it does **not** invoke DESTROY on your objects. The process image is replaced; Perl-level cleanup never happens. Anything that needs a clean shutdown (temp files, lockfile removal, buffered log flushes) must happen before `exec`.
- **Close-on-exec.** Filehandles with the `FD_CLOEXEC` flag set are closed by the kernel during `execvp`; others stay open. Control this with `fcntl` or the `$_F` system filehandle threshold.
- **Shell-mediated failure is invisible.** When `exec` routes through `/bin/sh -c`, a missing target program is reported by the shell and `exec` itself has already been replaced by `/bin/sh`. Your Perl code will not see a failure return.
- **One-arg list with no metacharacters** is *still* split on whitespace and passed to `execvp` - so `exec "ls /tmp"` is a direct `exec` of `/bin/ls` with one argument, not a shell call.
- **Empty LIST** is a runtime error; there is no program to run.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `system` - run a program and wait for it; what you almost certainly wanted if `exec` is at the top of the file
- `fork` - pair with `exec` in the child to launch a program without replacing the parent
- `wait` - collect a child produced by `fork` + `exec`
- `die` - terminate the current program with an error; the idiomatic partner of `exec ... or die`
- `$_!` - error reason set when direct-`exec` fails
- `$|` - autoflush flag; set before `exec` so no buffered output is lost when the process image is replaced

Hashes

exists

Test whether a hash or array element, or a named subroutine, is present - without creating it and without caring what it holds.

`exists` asks a structural question: *is there an entry at this location?* It does not ask what the entry's value is. An entry whose value is `undef` still exists. An entry that has never been assigned does not - looking it up with a plain read would return `undef`, but `exists` distinguishes the two cases.

Synopsis

```
exists $hash{KEY}
exists $array[INDEX]
exists &subroutine
exists $ref->{A}{B}           # autovivifies $ref->{A}!
```

What you get back

A boolean: 1 (a true scalar) if the element or subroutine is present, the empty string "" (a false scalar) if not. The result is always a plain scalar - there is no list-context variant.

```
my %h = (a => undef);
exists $h{a};           # 1    (key exists)
defined $h{a};          # ""   (value is undef)
exists $h{b};           # ""   (key does not exist)
```

exists is not defined

The two answer different questions and they disagree exactly when an existing entry holds `undef`:

State	exists	defined	Plain read
Never assigned	false	false	undef
Assigned <code>undef</code>	true	false	undef
Assigned any other value	true	true	that value

Use `exists` when you need to know whether a key has been placed in the structure - e.g. an option hash where a bare key means “the caller opted in, value doesn’t matter.” Use `defined` when you need to know whether a value is usable.

Examples

Hash-key test. Classic option-hash idiom - presence, not truth:

```
sub configure {
    my %opt = @_;
    my $verbose = exists $opt{verbose};    # caller passed verbose
    => anything
    my $color   = $opt{color} // "auto";   # caller-supplied or
    default
    ...
}
```

Distinguishing “missing” from “present but undef”:

```
my %cache = (user_42 => undef);           # cached: "we looked,
nothing there"
```

(continues on next page)

(continued from previous page)

```

if (exists $cache{user_42}) {
    # hit - even though the value is undef, we already did the
    ↳lookup
} else {
    # miss - have to query the database
}

```

Sparse-array test. Deleting an array element leaves a hole that exists reports as false:

```

my @a = (10, 20, 30);
delete $a[1];
exists $a[0];           # 1
exists $a[1];           # ""    (hole)
exists $a[2];           # 1
exists $a[99];          # ""    (past the end)

```

Subroutine existence. True once the sub has been declared, even if the body is a forward declaration with no definition yet:

```

sub later;                # forward declaration
exists &later;            # 1
defined &later;           # ""    (no body yet)

sub later { 42 }
defined &later;           # 1

```

Nested structure - the **safe** way. Check each level before descending, because the arrow-chain form has a side effect (see Edge cases):

```

if (exists $tree{users}
    && exists $tree{users}{$id}
    && exists $tree{users}{$id}{email}) {
    send_mail($tree{users}{$id}{email});
}

```

Edge cases

- **Autovivification of intermediate levels.** This is the single most important pitfall. `exists $ref->{A}{B}{C}` tests the innermost key, but evaluating the expression autovivifies every **intermediate** hash - `$ref->{A}` and `$ref->{A}{B}` spring into existence as empty hashes even when the exists test returns false:

```

my %h;
exists $h{a}{b}{c};      # returns "" (false)
exists $h{a};            # now returns 1 - $h{a} was
↳created!
exists $h{a}{b};         # likewise - now an empty hash

```

The innermost element is the only one that does not autovivify. The same happens through references: `exists $ref->{A}` on an undef `$ref` vivifies `$ref` into

a hash reference. If you need to probe a deep structure without mutating it, walk it one level at a time with `exists` at each step, or use a dedicated helper like `[Data::Diver]` semantics (a chained `eval` / no autovivification pragma scope).

- **exists on a key with value `undef` is true.** This is the defining difference from `defined`. `delete` removes both the key and the value; assigning `undef` keeps the key.
- **Array bounds.** `exists $a[$i]` is false when `$i` is past the end of the array **or** when the slot has been `deleted` mid-array. Negative indices count from the end: `exists $a[-1]` tests the last element if the array is non-empty.
- **Array exists is discouraged upstream.** The notion of deleting an array element is not conceptually clean - `delete $a[1]` leaves a hole rather than shifting. Most code that reaches for `exists $a[$i]` is better served by a bounds check (`$i <= $#a`) or by using a hash.
- **exists &sub() is a syntax error.** The argument must name a subroutine, not call one. `exists &sub` asks “is `sub` declared?”; `exists &sub()` would mean “does the return value of calling `sub` exist?”, which is meaningless.
- **AUTOLOAD does not count.** `exists &sub` is false for a sub that would spring into existence via AUTOLOAD on first call. A failing `exists` therefore does not guarantee a call would fail.
- **Tied variables.** For tied hashes and arrays, `exists` calls the `EXISTS` method on the tied object. The tie implementation decides what “existing” means - a DBM tie checks the on-disk record, a lazy tie might materialise the entry. See `perltie`.
- **The argument must be an lvalue expression whose final operation is a key or index lookup, or a subroutine name.** `exists func()`, `exists $scalar`, `exists @array`, `exists %hash` are all errors. Only element access and `&sub` are accepted.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `defined` - asks whether the value is usable, not whether the key is present; differs from `exists` only when the stored value is `undef`
- `delete` - removes an entry so that `exists` on it becomes false; the counterpart to `exists`
- `each` - iterates key/value pairs of a hash; only visits keys for which `exists` is true
- `keys` - returns the list of existing keys; `exists $h{$k}` is equivalent to asking whether `$k` is among keys `%h`, but `O(1)` instead of `O(n)`
- `ref` - classifies what a reference points to; useful before `exists` on `$ref->{...}` if you are not sure `$ref` is a hash reference
- `scalar` - force scalar context; note that `exists` is already scalar-only and does not need it

Control flow

exit

Terminate the program with a status value.

`exit` evaluates `EXPR`, runs end-of-program cleanup (END blocks and object destructors), and then exits the process with that value as its status code. With no argument it exits 0 - success.

Synopsis

```
exit EXPR
exit
```

What you get back

Nothing. `exit` does not return to its caller. It is a statement in the control-flow sense even though the grammar lets you write it inside an expression. Code after `exit` in the same block is unreachable:

```
exit 1;
print "never\n";           # dead code
```

Status values and the 8-bit truncation

The operating system stores an exit status in **8 bits**. Whatever integer you pass to `exit` is taken modulo 256 by the kernel before the parent process sees it. `exit 256` looks like `exit 0` to the shell. `exit -1` becomes 255. Pass only values in 0..255 and you avoid the surprise.

The only universally portable values are 0 for success and 1 for a generic error. Other numbers have conventions - the `sysexits.h` range 64..78 on Unix, specific codes expected by sendmail filters, and so on - but those are agreements between your program and a specific caller, not guarantees of the language. Exiting 69 (`EX_UNAVAILABLE`) tells a sendmail incoming-mail filter to defer the message; the same 69 from a script run by cron means nothing in particular.

```
exit 0;           # success
exit 1;           # generic failure
exit 2;           # conventional "misuse" (grep, diff)
exit 77;          # conventional "skip" (Automake test)
```

Cleanup: END blocks and destructors

`exit` is **not** an immediate syscall. Before the process goes away, Perl:

1. Runs every END block in **reverse order of compilation** (LIFO).
2. Calls DESTROY on any object whose refcount drops to zero as global destruction tears down pads, the symbol table, and package globals.

Both kinds of cleanup can observe **and change** `$?`. The value you passed to `exit` is placed in `$?` on entry to the `END/DESTROY` phase; whatever `$?` holds when the last cleanup returns is the status the process exits with.

```
END { $? = 0 if $? == 2; } # rewrite "misuse" to success

exit 2;                    # process actually exits 0
```

This is occasionally useful and regularly astonishing. If a module you did not write installs an `END` block that touches `$?`, your carefully chosen status code can arrive at the parent shell altered.

When you need a hard, unconditional exit that bypasses `END` and `DESTROY` entirely, call `POSIX::_exit`:

```
use POSIX ();
POSIX::_exit(3);           # no END, no DESTROY, status = 3
```

Reach for `_exit` in child processes after `fork` where the parent owns the cleanup, and in signal handlers where running arbitrary Perl code is unsafe.

exit is not die

`exit` is the wrong tool for reporting an error from a library or a subroutine. A caller higher up the stack may want to catch the failure, log it, and continue - an `eval` block or a `Try::Tiny` wrapper cannot recover from `exit`. The process is gone.

```
sub load_config {
    my ($path) = @_;
    open my $fh, "<", $path
        or die "open $path: $!"; # correct - trappable
    ...
}

# NOT:
# open my $fh, "<", $path or exit 1;
```

Use `die` for error conditions. Reserve `exit` for the top level of the program where you have decided, as the application, that the run is over.

exit inside eval

`exit` is **not** caught by `eval`. Unlike `die`, which unwinds to the nearest `eval` and sets `$@`, `exit` walks straight past every `eval` frame on the stack, runs `END/DESTROY`, and terminates.

```
eval { exit 5 };           # process exits 5; the eval block
print "after eval\n";      # never runs
```

If you are writing a plugin host or a test harness that must survive misbehaved user code calling `exit`, there is no language-level catch. The conventional workaround is to `fork`, let the child call `exit`, and inspect the child's status in the parent:

```
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    run_user_code();      # may call exit
    exit 0;
}
waitpid $pid, 0;
my $status = $? >> 8;
```

Examples

Early return from the program after a prompt:

```
my $ans = <STDIN>;
exit 0 if $ans =~ /^[Xx]/;
```

Exit with a status derived from a child process - extract the high byte of `$?` so the shell sees the same code the child produced:

```
system @cmd;
exit $? >> 8 if $?;
```

Rewrite a status in an END block - whatever `$?` holds at the end of cleanup is the exit status delivered to the parent:

```
END {
    $? = 0 if $? == 2 && $ENV{IGNORE_MISUSE};
}
```

Bypass cleanup in a forked child so the parent's destructors do not run twice:

```
use POSIX ();
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    exec @cmd;
    POSIX::_exit(127);      # exec failed; do NOT run parent's END
}
```

Edge cases

- **exit with no argument** is `exit 0`. Many scripts rely on falling off the end of the file to get an implicit 0; writing `exit`; makes the intent explicit.
- **Non-integer argument**: `EXPR` is evaluated in scalar context and converted to an integer. `exit "3x"` exits 3; `exit "hello"` exits 0 with a warning under `use warnings`.
- **Negative values** are taken modulo 256 by the kernel. `exit -1` appears to the shell as 255. Pass only 0..255 to avoid the coercion.
- **Values above 255** lose their high bits. `exit 256` looks like success; `exit 300` looks like `exit 44`. If you need to convey a larger value, write it to a file or to `STDERR` and `exit` with a small status code.

- **END blocks can cancel your status but not the exit itself.** An END block cannot stop the process from terminating - it can only change `$_`, and therefore the status seen by the parent.
- **Global destruction order is not the reverse of construction.** During the final sweep Perl walks package globals in an implementation-defined order. Do not rely on one DESTROY running before another; if ordering matters, tear the objects down explicitly before calling `exit`.
- **Inside a BEGIN block,** `exit` terminates the compilation phase and exits the process. END blocks already registered run; later BEGIN/END/code in the file does not.
- **Threads:** in a program with multiple Perl interpreter threads, `exit` terminates the whole process, not just the calling thread. Use `threads->exit` for per-thread exit.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `die` - raise a trappable exception; the right choice for reporting errors from a subroutine
- `warn` - emit a diagnostic without terminating; pairs with `die` the way `print` pairs with `exit`
- END blocks - compile-time-registered cleanup code run in LIFO order on any exit, including `exit` `EXPR`
- DESTROY methods - per-object cleanup called during global destruction after END blocks have run
- `$_` - child-process / final exit status; visible to and mutable by END blocks and destructors
- `POSIX::_exit` - immediate termination that bypasses END and DESTROY; use in forked children and signal handlers

Numeric functions

`exp`

Raise e to a power.

`exp` returns e (the base of natural logarithms, roughly 2.718281828459045) raised to the power of `EXPR`. It is the inverse of `log`: for any finite positive x , `exp(log( $x$ ))` reproduces x to within floating-point rounding. With no argument, `exp` operates on `$_`.

Synopsis

```
exp EXPR
exp
```

What you get back

A double-precision floating-point number. The result is always non-negative: `exp(0) == 1`, `exp` of any positive value is greater than 1, and `exp` of any negative value lies in the open interval $(0, 1)$.

On every Linux target pperl supports, the underlying type is IEEE 754 binary64, so the range of finite results is roughly $5e-324$ up to $1.7976931348623157e+308$. Arguments above about 709.78 overflow to `Inf`; arguments below about -745 underflow to `0.0`.

Examples

Get *e* itself, and verify the round-trip against `log`:

```
my $e = exp(1);           # 2.718281828459045
printf "%.15f\n", exp(log(42)); # 42.000000000000000
```

No-argument form reads `$_`:

```
for (1, 2, 3) {
    printf "exp(%d) = %.6f\n", $_, exp;
}
# exp(1) = 2.718282
# exp(2) = 7.389056
# exp(3) = 20.085537
```

Compose with other numeric built-ins. The hyperbolic sine, for example, has no dedicated operator but falls out of `exp`:

```
sub sinh { (exp($_[0]) - exp(-$_[0])) / 2 }
printf "%.6f\n", sinh(1);           # 1.175201
```

Overflow at the top of the IEEE 754 range:

```
print exp(709), "\n";           # 8.21840746949e+307
print exp(710), "\n";           # Inf
```

Underflow at the bottom:

```
print exp(-745), "\n";           # 5e-324 (smallest subnormal)
print exp(-746), "\n";           # 0
```

Edge cases

- **Very large positive argument:** Any argument greater than roughly 709.78 produces `Inf`. `Inf` propagates through subsequent arithmetic (`Inf + 1 == Inf`, `Inf - Inf` is `NaN`) and stringifies as `"Inf"`.
- **Very negative argument:** Arguments below roughly -745 underflow to `0.0`. Values in between render as denormal floats with progressively reduced precision.
- **NaN input:** `exp(NaN)` is `NaN`. `NaN` compares unequal to everything including itself - test with `Scalar::Util::looks_like_number` or `$x != $x` rather than `==`.

- **exp(0)** is exactly 1, not a rounded approximation.
- **Non-numeric argument:** Perl coerces the argument through its standard string-to-number rules before calling the C `exp(3)` routine. `exp("")` and `exp("abc")` are therefore `exp(0) == 1`, with an Argument "... isn't numeric" warning under `use warnings`. A leading-numeric string like `"3.5xyz"` becomes 3.5, again with a warning.
- **undef argument:** Coerced to 0, yielding 1, with an uninitialized warning under `use warnings`.
- **List context:** `exp` is a unary numeric function. It ignores list context and always returns a single scalar. `my @r = exp(2)` puts one element into `@r`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `log` - natural logarithm, the inverse of `exp`; `exp(log($x)) == $x` for positive `$x`
- `sqrt` - square root; `sqrt($x) == exp(log($x)/2)` for positive `$x`
- `sin` - sine, the other transcendental unary built-in
- `cos` - cosine; with `sin` and `exp` covers the standard transcendentals available without a module
- `abs` - absolute value, useful when feeding a signed quantity into `log` after exp-based manipulation
- `$_` - default argument when `exp` is called without an explicit operand

SCALARS and strings

fc

Return the Unicode casefolded form of a string for case-insensitive comparison.

`fc` is the string operation that powers case-insensitive string equality. It produces a form of the string where case distinctions have been erased, so two strings are considered case-insensitively equal when - and only when - their `fc` results are byte-for-byte identical. It is also the function behind the `\F` escape in double-quoted strings.

Synopsis

```
use feature 'fc';           # or: use v5.16;

fc EXPR
fc                           # operates on $_
"\F...\E"                   # same casefold, inside a double-quoted
↪string
```

`fc` is gated behind the `fc` feature. Enable it explicitly with `use feature 'fc'`, pull it in via a version bundle (use `v5.16` or newer, use `feature ':5.16'`), or call it fully qualified as `CORE::fc` without any pragma.

What you get back

A string containing the casefolded form of `EXPR`. The result is a fresh string; the argument is never modified. Length may change - casefolding `"\x{1E9E}"` (LATIN CAPITAL LETTER SHARP S) normally expands to `"ss"`, two characters from one.

Treat the return value as opaque: it is a key suitable for equality comparison, not for display. `fc("Hello")` is `"hello"` for ASCII, but in the general case the output is not something you'd show to a user.

Why not `lc` or `uc`?

Lowercasing and uppercasing are **not** reliable for case-insensitive comparison. Both of these are wrong:

```
lc($a) eq lc($b)           # Wrong
uc($a) eq uc($b)           # Also wrong
```

They fail on characters whose lower/upper mapping is not symmetric - most famously the German sharp S. `lc("\x{1E9E}")` is `"\x{1E9E}"` (no lowercase form), but `uc("ß")` is `"SS"`. Casefolding sidesteps this by mapping both sides into a dedicated equality form:

```
fc($a) eq fc($b)           # Right
```

The regex-based equivalent that was correct before `fc` existed:

```
$a =~ /\^\Q$b\E\z/i
```

`fc` is the direct, non-regex way to get the same answer.

Global state it touches

- `$_` - used as the argument when `EXPR` is omitted.
- `use locale` - inside a `use locale` scope, casefolding of characters crossing the 255/256 boundary is disabled (see *Edge cases* below for the `U+1E9E` rule).
- `use feature 'unicode_strings'` - affects `fc` the same way it affects `lc`: forces full Unicode semantics on byte strings that would otherwise be treated under legacy 8-bit rules.

Examples

Case-insensitive equality, the canonical use:

```
use feature 'fc';
fc("Hello") eq fc("HELLO");           # true
fc("café") eq fc("CAFÉ");              # true
```

The German sharp S - the textbook case where `lc` and `uc` both fail but `fc` succeeds:

```
use feature 'fc';
my $a = "straße";
my $b = "STRASSE";
fc($a) eq fc($b);           # true
lc($a) eq lc($b);           # false - "straße" ne
→ "strasse"
```

Using `fc` as a hash key for case-insensitive lookup:

```
use feature 'fc';
my %seen;
for my $word (@words) {
    $seen{ fc $word }++;      # groups "Foo", "F00", "foo"
}
```

No argument - operates on `$_`:

```
use feature 'fc';
for (@lines) {
    next unless fc eq "quit";  # matches "QUIT", "Quit", ..
→ .
    last;
}
```

Inside a double-quoted string via `\F` (same operation, inline):

```
use feature 'fc';
my $name = "Alice";
my $key = "\F$name\E";        # "alice" - same as fc(
→ $name)
```

Calling without the feature pragma, fully qualified:

```
my $folded = CORE::fc($input); # works in any scope
```

Edge cases

- **No argument:** `fc` with no argument folds `$_`.
- **undef argument:** stringifies to the empty string, which folds to the empty string. Emits an uninitialized warning under `use warnings`.
- **Feature not enabled:** `fc` `EXPR` without `use feature 'fc'` (or a `use v5.16+` bundle) is a compile-time error - the parser does not recognise `fc` as a keyword. `CORE::fc(EXPR)` always works.
- **Length can change:** full casefolds may expand one character to several. `fc("\x{1E9E}")` is `"ss"`. Never rely on `length(fc $s) == length($s)`.
- **Not a round-trip:** `fc` is one-way. There is no “uncasefold” operation; the original case is gone.

- **U+1E9E under use locale:** `fc` of *LATIN CAPITAL LETTER SHARP S* (U+1E9E) normally folds to "ss". Under `use locale` that mapping is suppressed because it crosses the 255/256 codepoint boundary, which locale rules do not handle cleanly. Instead `fc` returns "\x{17F}\x{17F}" (two *LATIN SMALL LETTER LONG S*). Since each *long s* itself folds to "s", two of them compare equal to a single U+1E9E folded outside the locale scope - so equality semantics are preserved even though the byte form differs.
- **Turkic and “simple” folds are not provided:** Perl implements only the *full, non-Turkic* form of casefolding. For the simple form or the Turkic variant, use `Unicode::UCD::casefold` or the CPAN module `Unicode::Casing`.
- **Not the same as NFKC or NFC:** `fc` erases case, not compatibility differences. `fc("ffi")` (*LATIN SMALL LIGATURE FFI*) is "ffi" after fold, not "ffi". Combine with `Unicode::Normalize` if you need both.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `lc` - lowercases a string; use for display, not for case-insensitive comparison
- `uc` - uppercases a string; same caveat as `lc`
- `lcfirst` - lowercases only the first character
- `ucfirst` - uppercases only the first character
- `index` - substring search; pair with `fc` on both operands for a case-insensitive variant
- `$_` - the default argument when `EXPR` is omitted

Filehandles, files, directories

fcntl

Perform a `fcntl(2)` file-control operation on a filehandle.

`fcntl` is the direct Perl binding for the POSIX `fcntl(2)` system call. It reads or changes per-descriptor state - non-blocking mode, close-on-exec, advisory record locks, descriptor flags - by invoking `FUNCTION` against `FILEHANDLE`, with `SCALAR` acting as either the input argument, the output buffer, or both, depending on which `FUNCTION` you pass.

The `FUNCTION` codes are not Perl-level literals; they are C preprocessor macros exposed by the `Fcntl` module. Load it first, or `FUNCTION` will be an undefined bareword:

```
use Fcntl;
```


Synopsis

```
use Fcntl;
fcntl FILEHANDLE, FUNCTION, SCALAR
my $flags = fcntl($fh, F_GETFL, 0);
fcntl($fh, F_SETFL, $flags | O_NONBLOCK);
```

What you get back

On success, the integer the kernel returned. When that integer is 0, Perl substitutes the dual-valued string "0 but true" - true in boolean context, numeric 0 in numeric context, and exempt from the Argument "... isn't numeric warning. This lets you test the result with `or die` even for operations whose success value is literally zero:

```
fcntl($fh, F_SETFD, FD_CLOEXEC)
  or die "F_SETFD: $!";
```

On failure, `fcntl` returns `undef` and sets `$!` to the relevant `errno`. You do **not** need to wrap the return in `defined` - the "0 but true" convention makes a plain truth test sufficient.

For FUNCTION codes that write back through SCALAR (such as `F_GETLK`), the updated bytes land in the scalar you passed; the return value still follows the rule above.

How SCALAR is used

SCALAR plays three different roles depending on FUNCTION:

- **Input integer.** For flag-setting calls like `F_SETFD` or `F_SETFL`, SCALAR is the integer bitmask to install.
- **Ignored placeholder.** For flag-getting calls like `F_GETFD` or `F_GETFL`, SCALAR is unused by the kernel; pass 0 by convention.
- **In/out struct buffer.** For record-locking calls like `F_GETLK`, `F_SETLK`, and `F_SETLKW`, SCALAR must hold a packed struct `flock` - build it with `pack` before the call, and for `F_GETLK` decode the returned bytes with `unpack` afterward.

When SCALAR is used as an output buffer, pre-size it to at least the length of the struct the kernel will write; an undersized scalar is grown automatically, but pre-sizing makes the intent obvious.

Global state it touches

- `$!` - set on failure to the `errno` from `fcntl(2)`.
- Descriptor-level flags on FILEHANDLE (e.g. `O_NONBLOCK`, `FD_CLOEXEC`) persist for the lifetime of the descriptor, survive `dup` and `fork` according to POSIX rules, and are **not** Perl-visible through any other variable. The only way to read them back is another `fcntl` call with `F_GETFL` / `F_GETFD`.

Examples

Make a socket non-blocking - the canonical use:

```
use Fcntl qw(F_GETFL F_SETFL O_NONBLOCK);

my $flags = fcntl($remote, F_GETFL, 0)
    or die "F_GETFL: $!";
fcntl($remote, F_SETFL, $flags | O_NONBLOCK)
    or die "F_SETFL: $!";
```

Set close-on-exec so the descriptor is not inherited by child processes across `exec`:

```
use Fcntl qw(F_SETFD FD_CLOEXEC);

fcntl($fh, F_SETFD, FD_CLOEXEC)
    or die "F_SETFD: $!";
```

Read the current descriptor flags and test a bit:

```
use Fcntl qw(F_GETFD FD_CLOEXEC);

my $fd_flags = fcntl($fh, F_GETFD, 0);
defined $fd_flags or die "F_GETFD: $!";
if ($fd_flags & FD_CLOEXEC) {
    # descriptor will close on exec
}
```

Place an advisory write lock on the whole file. The struct flock layout is platform-specific; on Linux `s s l l i` covers `l_type` `l_whence` `l_start` `l_len` `l_pid`:

```
use Fcntl qw(F_SETLK F_WRLCK SEEK_SET);

my $lock = pack('s s l l i', F_WRLCK, SEEK_SET, 0, 0, 0);
fcntl($fh, F_SETLK, $lock)
    or die "lock refused: $!";
```

Toggling off a single flag - mask out `O_NONBLOCK` to return a handle to blocking mode:

```
use Fcntl qw(F_GETFL F_SETFL O_NONBLOCK);

my $flags = fcntl($fh, F_GETFL, 0);
fcntl($fh, F_SETFL, $flags & ~O_NONBLOCK)
    or die "F_SETFL: $!";
```

Edge cases

- **Forgot use Fcntl:** `F_GETFL` and friends parse as barewords and evaluate to a warning-level nuisance under `use strict / use warnings`, or to the string `"F_GETFL"` otherwise - either way the call fails with `EINVAL`. Always load `Fcntl` first.

- **"0 but true" return:** do not compare the return with `== 0` as a failure test. Use a plain boolean (or `die`) or `defined` - the former is the idiom.
- **F_SETLK vs F_SETLKW:** `F_SETLK` returns `undef` and sets `$!` to `EAGAIN` / `EACCES` when another process holds a conflicting lock. `F_SETLKW` blocks until the lock is granted. Pick deliberately; mixing them up is the usual source of hangs.
- **F_GETLK output:** on success the `struct flock` bytes in `SCALAR` are overwritten with information about the conflicting lock (or `l_type == F_UNLCK` if the region is free). `unpack` back with the same template you packed with.
- **Not every system implements every FUNCTION:** `fcntl` raises an exception on a machine that does not implement `fcntl(2)` at all. On systems that do, unsupported `FUNCTION` codes fail with `EINVAL` via `$!`. The `Fcntl` module only exports constants the current platform actually defines.
- **fcntl vs ioctl:** both share the argument-processing and return-value convention, but they target different kernel interfaces. Use `fcntl` for descriptor flags and advisory locks; use `ioctl` for device-specific control codes.
- **fcntl vs flock:** `flock` uses whole-file BSD-style `flock(2)` locks, which on Linux interoperate with `fcntl` POSIX locks only in limited ways. If you need byte-range locking or POSIX semantics (lock released on *any* close of the file by the owning process), use `fcntl` with `F_SETLK`; otherwise `flock` is simpler.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `Fcntl` - the module that exports the `F_*` / `O_*` / `FD_*` / `SEEK_*` constants `fcntl` expects; always use it first
- `ioctl` - same calling and return-value convention as `fcntl`, but for device control codes rather than file-control flags
- `flock` - simpler whole-file advisory locks; reach for it when you don't need byte-range or POSIX semantics
- `open` - opens the filehandle in the first place; some flags `fcntl` adjusts (e.g. `O_NONBLOCK`, `O_APPEND`) are also settable at open time
- `sysopen` - low-level open that takes the same `O_*` flags from `Fcntl`; use it when you want the flags applied atomically rather than via a follow-up `fcntl(F_SETFL)`
- `pack` / `unpack` - required for building and decoding the `struct flock` buffer used by `F_GETLK` / `F_SETLK`

[Scoping](#) · [Classes and OO](#)

field

Declare a per-instance variable inside a `class` block.

`field` introduces a new variable whose storage is private to each object of the enclosing class. Every method and every `ADJUST` block of that class sees the field as if it were a lexical in scope at that point - but the value each sees is the value belonging to the *current* instance. Fields are the class feature's replacement for hash-slot bookkeeping on a blessed reference.

The feature is still marked **experimental** in Perl 5.42 (see *Differences from upstream*); enable it with `use feature 'class';` and silence the warning category with `no warnings 'experimental::class';` if you don't want the noise.

Synopsis

```
field $scalar;  
field $scalar = EXPR;  
field @array  : ATTRIBUTES;  
field %hash   : ATTRIBUTES = EXPR;
```

What you get back

`field` is a declaration, not an expression. It produces no value and cannot appear on the right-hand side of anything. Each `field` statement allocates one slot per instance in the class's layout; the slot is the field variable itself, bound to a per-object storage location on entry to any method or `ADJUST` block.

Scalars, arrays, and hashes are all allowed:

```
class Thing {  
    field $scalar = 42;  
    field @array  = qw(this is just an array);  
    field %hash   = (species => 'Martian', planet => 'Mars');  
}
```

Where field declarations are legal

Only directly inside a `class` block. `field` is not a general-purpose declarator - it is not legal inside a method body, an `ADJUST` block, a nested sub, or plain package code. The compiler rejects it anywhere else:

```
class C {  
    field $ok;                # OK  
    method m {  
        field $nope;          # compile error  
    }  
}
```

A field must be declared before it can be referred to by any method or initializer in the same class body.

Field initializers

If `a = EXPR` is present, the expression runs once per constructor call, after all fields declared earlier in the class body have been initialized. This matters when a later field's default depends on an earlier one:

```
class WithACounter {
  my $next_count = 1;
  field $count = $next_count++;
}
```

Inside an initializer, `$self` does **not** exist - the object is still being built. Use the `__CLASS__` token when the initializer needs the class name (for example, to call a class method that subclasses can override):

```
class WithCustomField {
  use constant DEFAULT_X => 10;
  field $x = __CLASS__->DEFAULT_X;
}

class DifferentCustomField :isa(WithCustomField) {
  sub DEFAULT_X { rand > 0.5 ? 20 : 30 }
}
```

An instance of `DifferentCustomField` will see `__CLASS__` resolve to its own name, so the initializer picks up the overridden `DEFAULT_X`.

Field attributes

Attributes after a colon control how the field participates in construction and whether accessor methods are generated for it.

:param

Take the field's value from a named argument to the constructor:

```
field $x :param;
field $y :param(the_y_value);
```

By default the parameter name is the field's name with the sigil stripped (`$x` $\rightarrow$ `x`). An explicit name in parentheses overrides that. Without a defaulting expression the parameter is **required**

- omitting it from `->new( . . . )` throws. With a defaulting expression the parameter is optional, and three default operators are available:
- `=` - default applies only when the caller omitted the parameter.
- `//=` - default also applies when the caller passed `undef`.
- `||=` - default also applies when the caller passed a false value.

```
class Point {  
  field $x :param = 0;           # default if omitted  
  field $y :param //= 0;        # default if omitted or undef  
  field $z :param ||= 0;        # default if omitted, undef, or 0  
}
```

:reader

Generate a zero-argument accessor method that returns the field. Without an explicit name the method is named after the field (sigil stripped):

```
field $s :reader;  
  
# equivalent to:  
field $s;  
method s () { return $s }
```

An explicit name is allowed:

```
field $x :reader(get_x);           # method get_x () { return $x }
```

Readers can be applied to array and hash fields too; in list context the method yields the contents, and in scalar context the element count - the usual context behaviour of the underlying variable:

```
field @users :reader;  
...  
scalar $instance->users;          # count of users
```

:writer

Generate a one-argument setter method that assigns its argument to the field and returns the invocant (to support chaining). The default method name is the field name (sigil stripped) prefixed with set_:

```
field $s :writer;  
  
# equivalent to:  
field $s;  
method set_s ($new) { $s = $new; return $self }
```

An explicit name is allowed:

```
field $x :writer(write_x);         # method write_x ($new) { ... }
```

:writer currently only works on scalar fields; applying it to an array or hash field is a compile-time fatal error. Build array or hash writers by hand.

Global state it touches

None directly. Field storage lives on the instance, and initializer expressions run in the constructor's scope - they can read any variable visible at the point of declaration, but field itself touches no documented special variable.

Examples

Minimal class with one field, initialized in ADJUST:

```

use feature 'class';
no warnings 'experimental::class';

class Greeter {
    field $greeting;

    ADJUST {
        $greeting = "Hello";
    }

    method say_to ($name) {
        say "$greeting, $name";
    }
}

Greeter->new->say_to("world");    # Hello, world

```

Constructor parameters via :param, with a mix of required and defaulted fields:

```

class Point {
    field $x :param;           # required
    field $y :param = 0;       # optional, defaults to 0

    method as_string { "($x, $y)" }
}

Point->new(x => 3)->as_string;    # (3, 0)
Point->new(x => 3, y => 4)->as_string; # (3, 4)

```

Auto-generated accessors cover the common boilerplate:

```

class Person {
    field $name :param :reader;
    field $age  :param :reader :writer;
}

my $p = Person->new(name => "Ada", age => 36);
$p->name;           # "Ada"
$p->age;            # 36
$p->set_age(37);    # returns $p

```

Array and hash fields with defaults:

```
class Bag {
  field @items = ();
  field %counts;

  method add ($item) {
    push @items, $item;
    $counts{$item}++;
    return $self;
  }

  method unique { keys %counts }
}
```

Initializer that depends on an earlier field:

```
class Rect {
  field $w :param;
  field $h :param;
  field $area = $w * $h;      # evaluated after $w, $h are set
}
```

Edge cases

- **Lexical visibility is method-scoped, not file-scoped.** A field is visible from any method or ADJUST block of the same class, regardless of source file order. It is *not* visible from plain sub definitions inside the class block - only from method.
- **Per-instance, not per-class.** Two instances of the same class have independent storage for every field. Assigning in one instance does not affect another. If you want class-wide state, use a file-scoped `my` variable outside the field list.
- **Sigil determines type.** `field $x` is a scalar slot, `field @a` an array slot, `field %h` a hash slot. Unlike hash-based objects, there is no way to retype a field at run-time; the slot's container type is fixed by the sigil at declaration.
- **\$self is unavailable in initializers.** The object is not yet constructed when the initializer runs. Use `__CLASS__` for the class name, and defer anything requiring the full object to ADJUST.
- **Initializer order is declaration order.** Later initializers may refer to earlier fields; the reverse is a reference to an uninitialized variable.
- **:writer on non-scalar fields is fatal at compile time.** The attribute is defined only for scalars in 5.42; applying it to `@arr` or `%h` aborts compilation of the class.
- **:param without a default is mandatory.** `Class->new()` with the parameter missing throws. Add `= EXPR, //= EXPR, or ||= EXPR` to make it optional.
- **No direct external access.** Without `:reader` / `:writer` (or a hand-written method) a field is unreachable from outside the class - that is the encapsulation the class feature provides.
- **Experimental warning.** Parsing a field declaration emits an experimental::class warning unless that category is suppressed; see *Differences from up-*

stream.

Differences from upstream

Fully compatible with upstream Perl 5.42. Upstream marks the `class` feature - and therefore `field` - as experimental; pperl tracks the same status and emits the same `experimental::class` warning. The known upstream bugs listed in `perlclass` (segfaults around in-file inheritance, refaliasing interactions, and leaky encapsulation) apply equally to pperl.

See also

- `class` - declares the class whose body is the only place `field` is legal
- `method` - the only subroutine form that sees `field` variables by name
- `my` - ordinary lexical declaration; use it for class-wide state that should *not* be per-instance
- `our` - package-global declaration; use it for genuinely global state a class wants to expose
- `bless` - the pre-5.38 way to build objects; `field` replaces the hash-slot bookkeeping that `bless`-based classes needed
- `ref` - check an object's class at runtime; still works on instances produced by a `class` constructor

I/O

fileno

Return the OS-level file descriptor number behind a filehandle.

`fileno` looks past Perl's buffered filehandle abstraction and gives you the raw integer descriptor the kernel uses. It is the bridge between Perl I/O and system calls that speak descriptors directly - `select`, `POSIX::dup2`, `fcntl`, or anything in `IO::Poll` / `IO::Select`. On a directory handle it asks the same question against `dirfd(3)`, which not every platform provides.

Synopsis

```
fileno FILEHANDLE
fileno DIRHANDLE
fileno EXPR           # EXPR evaluated as an indirect handle
```

What you get back

- An integer `>= 0` - the OS file descriptor - for a normal open handle backed by a real kernel object.
- `-1` when the handle is open but has **no** real descriptor. This is what you get for in-memory handles created by `open` with a scalar reference as the third argument (`open my $fh, ">", \ $buf`).

- `undef` when the handle is closed, never opened, or - on a directory handle - when the platform lacks `dirfd(3)`. In the `dirhandle` case `$!` is set.

The three return classes matter. Code that checks `fileno($fh)` for truth conflates “no such fd” with “fd number 0” (STDIN). Always compare explicitly:

```
my $fd = fileno $fh;
die "handle not open"      unless defined $fd;
die "handle has no real fd" if $fd == -1;
```

Global state it touches

- `$!` - set when `fileno` on a directory handle fails because the platform has no `dirfd(3)` equivalent.

`fileno` does not read or write any other I/O globals. It does not flush, it does not seek, it does not change the currently selected handle.

Examples

The common case - turn a Perl filehandle into the fd number a system call wants:

```
open my $fh, "<", "/etc/hostname" or die $!;
print fileno $fh, "\n";           # e.g. "3"
```

Standard handles always have fixed descriptors 0, 1, 2:

```
print fileno STDIN, "\n";         # 0
print fileno STDOUT, "\n";       # 1
print fileno STDERR, "\n";      # 2
```

Check whether two handles are duplicates of the same underlying descriptor (the idiom from the upstream POD, adjusted for the -1 case):

```
if (fileno($this) != -1 && fileno($this) == fileno($that)) {
    print "\$this and \$that are dups\n";
} elsif (fileno($this) != -1 && fileno($that) != -1) {
    print "\$this and \$that have different "
        . "underlying file descriptors\n";
} else {
    print "at least one handle has no real file descriptor\n";
}
```

In-memory handles have no kernel object, so `fileno` returns -1:

```
my $buf = "";
open my $mh, ">", \$buf or die $!;
print fileno $mh, "\n";           # -1
```

Build a descriptor bitmap for `select` - the textbook use for `fileno`:

```
my $rin = "";
vec($rin, fileno($sock), 1) = 1;
vec($rin, fileno(STDIN), 1) = 1;
my $n = select(my $rout = $rin, undef, undef, 5.0);
```

Directory handles, where the platform supports `dirfd(3)`:

```
opendir my $dh, "." or die $!;
my $dfd = fileno $dh;
if (defined $dfd) {
    # use with fstat, openat, fchdir, ...
} else {
    warn "no dirfd on this platform: $!";
}
```

Edge cases

- **Closed handle:** returns `undef`. No warning under `use warnings` - `fileno` is deliberately quiet so it can be used as a probe.
- **Never-opened bareword:** `fileno NOSUCH` returns `undef`; strict mode will refuse the bareword at compile time unless it's declared.
- **In-memory scalar handles:** return `-1`, not `undef`. Treating `-1` as “open but unusable for system calls” is the intended contract - the handle itself still works for `print`, `readline`, etc.
- **`fileno(-1)` trap:** an explicit `if (fileno $fh) { ... }` is a bug waiting to happen: `fd 0` is `STDIN`. Use `defined` and compare against `-1` explicitly.
- **Indirect filehandle via expression:** if `FILEHANDLE` is any expression other than a bareword or simple scalar, its value is taken as an **indirect handle name** (a string), not as a handle object. To pass a handle held in a data structure, dereference first:

```
fileno $handles[0];           # indirect: uses the string
fileno ${ \ $handles[0] };    # still wrong
fileno $handles[0]->fileno;    # wrong: calling a method
↪ on ""
my $fh = $handles[0]; fileno $fh; # right
```

- **Directory handle on platforms without `dirfd(3)`:** returns `undef` and sets `$!`. Linux has `dirfd(3)`, so on pperl's supported targets this path is not exercised.
- **Tied handles:** `fileno` calls the tie class's `FILENO` method if defined; otherwise returns `undef`. The tied class is responsible for returning a sensible value (`-1` for “no fd” is the convention).
- **Dup'd handles:** two handles opened with `open` from the same underlying fd (e.g. `open my $copy, ">&", $orig`) return the **new** descriptor, not the original - the `dup` created a fresh fd.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `open` - creates the filehandles that `fileno` inspects; the `>&` and `<&=` modes take descriptor numbers that round-trip through `fileno`
- `opendir` - source of directory handles for the `dirhandle` form of `fileno`
- `close` - after which `fileno` on the same handle returns `undef`
- `select` - the four-argument form consumes bitmaps built from `fileno` values
- `binmode` - works on the PerlIO layer above the `fd`; `fileno` reaches beneath it

Control flow

finally

Run cleanup code on the way out of a `try/catch`, whether the body succeeded, threw, or jumped away.

`finally` introduces a third block that can follow a `try/catch` pair. Its body runs exactly once, after the `try` body completes - however it completes. A normal fall-through, an exception caught by the `catch`, an exception that escapes because there is no matching `catch`, a return from the enclosing sub, a `goto`, a `last/next/redo` leaving the enclosing loop: every one of these routes through the `finally` block before control continues. The pattern is equivalent to scheduling a `defer` block at the top of the `try`, with a syntax that keeps the cleanup visually attached to the `try/catch` it belongs to.

Synopsis

```
use feature 'try';
no warnings 'experimental::try';
```

```
try    { BODY }
catch ($e) { HANDLE }
finally { CLEANUP }
```

What you get back

`finally` has no return value. The final expression of the block is evaluated for its side effects and then discarded - it does **not** contribute to the value of the enclosing `try/catch` expression, even if the whole construct is the last statement of a sub or `do` block being used to produce a value.

```
my $v = do {
    try    { compute() }
    catch ($e) { $DEFAULT }
    finally { log("done"); 999 }    # 999 is discarded
```

(continues on next page)

(continued from previous page)

```
};
# $v is either compute()'s value or $DEFAULT
```

Use `finally` for cleanup that has to happen, not for computing a result.

Feature status

`finally` is part of the `try` feature, but unlike `try` and `catch` (which became non-experimental in Perl 5.40) it remains experimental in Perl 5.42. Using it without silencing the warning produces:

```
finally is experimental at ... line N.
```

Enable the feature and silence the warning together:

```
use feature 'try';
no warnings 'experimental::try';
```

Because the warning category is `experimental::try` - the same one that `try/catch` used to emit - silencing it enables `finally` without re-enabling warnings on the now-stable parts of the feature you do want diagnostics for.

When the block runs

The contract is “on the way out, no matter how.” Concretely, `finally` runs after each of the following:

- **Normal completion of `try`.** The `try` body fell off the end or returned its last expression value; `finally` runs next, then the statement after the `try` construct.
- **Exception caught by `catch`.** The `try` body threw, the `catch` matched and handled it; `finally` runs after the `catch` body.
- **Exception not caught.** There is no `catch`, or the `catch` itself re-threw with `die`; `finally` still runs, then the exception continues propagating to the next dynamic `eval/try` outward.
- **return inside `try` or `catch`.** The enclosing sub is about to return; `finally` runs, then the return takes effect.
- **Loop control (`last`, `next`, `redo`) inside `try` or `catch`.** `finally` runs, then the loop control takes effect in the enclosing loop.
- **`goto &sub` or `goto LABEL` leaving the `try`.** `finally` runs before the jump completes.

The ordering matters: if the `catch` itself raises a new exception, `finally` still runs, and the new exception is the one that propagates (the original is replaced, just as with any `die` inside a handler).

Restrictions on the block body

A `finally` block is a cleanup point, not a control-flow redirect. Perl rejects any control-flow statement that would try to override how the construct is already unwinding:

- **return** inside `finally` is a syntax / compile-time error.
- **goto** inside `finally` is a syntax / compile-time error.
- **Loop controls** (`last`, `next`, `redo`) targeting a loop *outside* the `finally` block are rejected.

Inside a plain loop that is *contained* in the `finally` body, the loop controls apply to that inner loop as normal - the restriction is on leaving the `finally` block, not on looping inside it.

`die` is permitted. A `finally` block that throws replaces any in-flight exception from the `try/catch` with the new one. This is usually a bug (you lose the original failure), so guard cleanup code that might itself fail:

```
finally {  
    eval { $fh->close };           # swallow close errors during  
    ↪ cleanup  
}
```

Relationship to defer

`finally` BLOCK is “`defer` BLOCK placed at the top of the `try`, attached instead to the `try/catch` pair.” Both run on every exit path from their containing scope; both forbid the same control-flow statements in their body; both discard their return value.

The difference is scope attachment:

- `defer` attaches to the *innermost enclosing block* and fires on that block’s exit. It is the general-purpose cleanup primitive and works in any block.
- `finally` attaches to the `try/catch construct` and fires when that construct is done. It is syntactic sugar for the common case where the cleanup is specifically about what the `try` was doing.

```
# These two are behaviourally equivalent:
```

```
try {  
    my $fh = acquire();  
    defer { release($fh) }  
    work($fh);  
}  
catch ($e) { warn $e }  
  
try {  
    my $fh = acquire();  
    work($fh);  
}
```

(continues on next page)

(continued from previous page)

```
catch ($e) { warn $e }
finally { release_last() }      # if $fh were available here
```

Note one practical consequence: variables declared inside the try body go out of scope before finally runs. If the cleanup needs a resource allocated inside the try, either declare it in an outer scope, or use defer inside the try where the variable is live.

Examples

Always close a handle, even on error:

```
use feature 'try';
no warnings 'experimental::try';

open my $fh, '<', $path or die "open $path: $!";
try {
    process($fh);
}
catch ($e) {
    warn "processing failed: $e";
}
finally {
    close $fh;
}
```

Release a lock regardless of outcome:

```
my $lock = $mutex->lock;
try {
    critical_section();
}
finally {
    $lock->release;
}
```

Observability: log every exit path:

```
try {
    run($job);
}
catch ($e) {
    $job->mark_failed($e);
    die $e;                      # re-throw; finally still runs
}
finally {
    metrics->tick('job.exit', $job->id);
}
```

Value-producing try/catch, with finally for cleanup only:

```
my $result = do {
    try    { fetch($url) }
    catch ($e) { $CACHED }
    finally { close_connection() }
};
# $result is fetch() or $CACHED; finally's value is never used
```

Edge cases

- **catch can be omitted.** `try { ... } finally { ... }` is legal and runs finally on both normal and exceptional exit; the exception then propagates as if no try were there.
- **Nested try/finally:** finally blocks fire innermost-first as scopes unwind, matching defer semantics.
- **Exception inside finally:** replaces any in-flight exception. The original is lost unless you captured it in the catch first.
- **\$@ during finally:** not reliable for “what just happened.” If the finally needs to know whether it is running after success or failure, set a flag in the catch block and inspect it:

```
my $failed;
try { risky() }
catch ($e) { $failed = $e; die $e }
finally {
    if ($failed) { rollback() } else { commit() }
}
```

- **Global destruction:** finally blocks registered by a try that is still active when the interpreter starts tearing down run, but the same caveats as `die` inside DESTROY apply - keep the body defensive.
- **Parse requires the feature:** without use feature 'try', the word finally is not a keyword and parses as a bareword function call, which usually produces a confusing unrelated error. Always enable the feature before using the syntax.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `try` - introduces the block whose exit finally hooks; see *Try Catch Exception Handling* in `perlsyn`
- `catch` - the optional middle block that handles exceptions thrown by try; finally runs after it either way
- `defer` - the general-purpose “run on scope exit” block; finally is defer attached to a try/catch pair

- `die` - the only way to cause the catch branch to fire; a die inside finally replaces any in-flight exception
- `eval` - the older exception-catching mechanism; `try/ catch/finally` supersede the common `eval { ... }; if ($@)` idiom with cleaner scoping and proper control-flow semantics

I/O

flock

Place an advisory lock on an open file.

`flock` is Perl's portable file-locking primitive. It calls the underlying `flock(2)`, `fcntl(2)`, or `lockf(3)` - whichever the platform provides - on `FILEHANDLE` and blocks by default until the requested lock can be granted. Locks cover the **whole file**, never a byte range, and are **advisory**: they coordinate only among processes that also call `flock` on the same file. Code that ignores the lock can still read or modify the contents.

Synopsis

```
flock FILEHANDLE, OPERATION
```

What you get back

True on success, false on failure (with `$!` set). Always check the return value - with `LOCK_NB` a failure is the normal “someone else holds it” signal, not an error:

```
flock($fh, LOCK_EX | LOCK_NB)
    or die "already locked: $!";
```

On a platform that implements none of `flock(2)`, `fcntl(2)` locking, or `lockf(3)`, `flock` raises a fatal error rather than returning false. On Linux (ppperl's only supported platform) this never happens.

OPERATION values

`OPERATION` is one of the mutually exclusive request types, optionally bitwise-or'ed with the non-blocking flag. Traditional numeric values are 1, 2, 4, 8 but portable code imports the symbolic names from `Fcntl`, either individually or as a group via the `:flock` tag:

```
use Fcntl qw(:flock);
```

- `LOCK_SH` - shared lock. Multiple processes may hold a shared lock on the same file simultaneously. Used for readers.
- `LOCK_EX` - exclusive lock. Only one process may hold it; no other process may hold any lock on the file at the same time. Used for writers.
- `LOCK_UN` - release whatever lock this filehandle currently holds.
- `LOCK_NB` - bitwise-or with `LOCK_SH` or `LOCK_EX` to make the call non-blocking; `flock` returns false immediately if the lock cannot be granted, instead of waiting.

An flock call with a new mode atomically replaces an existing lock on the same file-handle (for example, upgrading LOCK_SH to LOCK_EX).

Global state it touches

- Sets \$! on failure.
- Flushes FILEHANDLE's output buffer before locking or unlocking, so locked-region data hits the OS before another process can see the lock state change.

Examples

Exclusive lock for a writer, blocking until granted:

```
use Fcntl qw(:flock);
open my $fh, ">>", "counter.log" or die $!;
flock($fh, LOCK_EX) or die "lock failed: $!";
print $fh "tick\n";
flock($fh, LOCK_UN) or die "unlock failed: $!";
close $fh;
```

Non-blocking attempt - skip the work if another process is already running:

```
use Fcntl qw(:flock);
open my $fh, ">", "/var/run/myjob.lock" or die $!;
flock($fh, LOCK_EX | LOCK_NB)
    or do { warn "another instance running\n"; exit 0 };
# ... work ...
# lock auto-released when $fh is closed / program exits
```

Classic mailbox appender. Note the seek to SEEK_END after locking - on very old systems >> append mode is not atomic, so the write position must be re-established once the lock is held:

```
use Fcntl qw(:flock SEEK_END);

sub lock_mbox { flock($_[0], LOCK_EX) or die "lock: $!" }
sub unlock_mbox { flock($_[0], LOCK_UN) or die "unlock: $!" }

open my $mbox, ">>", "/var/mail/$ENV{USER}" or die $!;
lock_mbox($mbox);
seek($mbox, 0, SEEK_END) or die "seek: $!";
print $mbox $msg, "\n\n";
unlock_mbox($mbox);
close $mbox;
```

Shared read lock while loading a config file, letting other readers proceed concurrently:

```
use Fcntl qw(:flock);
open my $fh, "<", "config.dat" or die $!;
flock($fh, LOCK_SH) or die "lock: $!";
```

(continues on next page)

(continued from previous page)

```
my @lines = <$fh>;
close $fh;                                # releases the lock
```

Upgrading a shared lock to exclusive - the second flock atomically swaps modes:

```
flock($fh, LOCK_SH) or die $!;
# ... inspect contents ...
flock($fh, LOCK_EX) or die $!; # now we intend to write
```

Edge cases

- **Locks are per-filehandle, not per-file.** Two independent opens of the same path each have their own lock slot. This is why locking your own file twice from the same process through different handles can deadlock or behave surprisingly on `fcntl(2)`-based systems.
- **Closing the filehandle releases the lock.** There is no separate “unlock” step required at program exit; `LOCK_UN` is useful only when you want to keep the handle open after.
- **LOCK_NB failure is not a croak.** It returns false and sets `$!` to `EWOULDBLOCK`. Treat it as an expected outcome, not an error.
- **Advisory only.** A process that never calls `flock` sees no locks. Use `flock` as a cooperation protocol among your own programs, not as a security mechanism.
- **Whole files only.** There is no byte-range locking through `flock`. For record-level locking use `fcntl` with `F_SETLK` / `F_SETLKW`.
- **NFS and other network filesystems.** Some older NFS implementations cannot honour `flock` across hosts; the call may silently become a no-op or return success without providing exclusion. Use `fcntl` locking if network coordination matters.
- **Fork inheritance.** When the platform uses real `flock(2)` (Linux does), locks are inherited across `fork`; the child shares them with the parent. On `fcntl(2)`-emulated platforms locks do not survive the fork, which makes writing locking servers considerably harder.
- **Buffer flushing.** Perl flushes `FILEHANDLE`’s output buffer on every `flock` call. Code that mixes buffered `print` with locking does not need an explicit flush before the lock boundary.
- **Interaction with dup-style opens.** An open of the form `open(my $A, ">>&=", $B)` shares the underlying file descriptor with `$B`, so `flock($A)` and `flock($B)` act on the same lock slot. A `>>& dup` gives each handle its own descriptor and its own lock slot.

Differences from upstream

Fully compatible with upstream Perl 5.42. `pperl` runs on Linux only, where `flock(2)` is always available, so the “platform has no locking primitive - fatal error” path documented upstream never triggers in practice.

See also

- `fcntl` - byte-range and advisory locking with finer control; use it when whole-file granularity is too coarse or when network filesystems require it
- `open` - produces the filehandle `flock` operates on; remember that the required open mode (< vs >) matters for `fcntl`-emulated `LOCK_SH` / `LOCK_EX`
- `close` - releases any `flock` still held on the handle
- `seek` - pair with `flock` when appending to files that other processes may have extended while you waited for the lock
- `fork` - lock inheritance across child processes depends on the underlying locking primitive
- `Fcntl` - source of the `LOCK_SH`, `LOCK_EX`, `LOCK_UN`, `LOCK_NB` constants and the `:flock` import tag

Processes

fork

Create a new process running the same program at the same point.

`fork` issues the `fork(2)` system call. After it returns, **two** processes are executing the same Perl program at the same statement - the original (parent) and a near-identical copy (child). The two are distinguished only by `fork`'s return value. File descriptors and, on some systems, the locks held on them are shared between parent and child; everything else is copied. On Linux the copy is cheap: data pages are shared copy-on-write, so the actual duplication happens only as each side writes to memory.

Synopsis

```
my $pid = fork;
```

What you get back

Three return values, one per role:

- **Parent:** the child's PID (a positive integer).
- **Child:** `0`.
- **Failure:** `undef`, with `$!` set to the `errno` from the underlying `fork(2)` (typically `EAGAIN` - resource limit hit - or `ENOMEM`).

The canonical dispatch is a three-way branch. Test for `undef` first, then for `0`, so the parent path is the fall-through:

```
my $pid = fork // die "fork failed: $!";
if ($pid == 0) {
    # child
    exec $cmd, @args;
    die "exec failed: $!";
}
```

(continues on next page)

(continued from previous page)

```
}
# parent continues here, with $pid = child's PID
```

The two-line “fork or die” idiom:

```
defined(my $pid = fork) or die "fork: $!";
```

Writing `fork or die` (without `defined`) is a bug: it treats the child’s legitimate `0` return as failure and dies in every child.

Flush before you fork

Perl attempts to flush all output handles before the `fork(2)` call, but you should not rely on that alone - any unflushed buffered data that does slip through is duplicated in both processes, so the same bytes get written twice. The canonical defence is to disable buffering on any handle you intend to use across a fork:

```
STDOUT->autoflush(1);           # or: $| = 1 while STDOUT is
↪selected
STDERR->autoflush(1);

print "about to fork\n";        # now actually on the wire
my $pid = fork // die "fork: $!";
```

Loading `IO::Handle` is not required for `autoflush` in modern Perl - the method is available on all filehandles. Setting `$|` on the currently selected handle has the same effect for that one handle.

Reaping children - or `$SIG{CHLD}`

Every child that terminates before its parent reaps it becomes a **zombie**: a kernel process-table entry held open so the parent can read the exit status. Ignore them and the process table fills up.

Two ways to keep it clean:

- Actively reap with `wait` or `waitpid`:

```
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    # ... child work ...
    exit 0;
}
my $reaped = waitpid $pid, 0;    # block until this child exits
my $status = $?;               # exit status of the child
```

- Tell the kernel you do not care:

```
$SIG{CHLD} = 'IGNORE';          # children auto-reaped, no
↪status
```

Useful for fire-and-forget workers. You lose the exit status - `wait` and `waitpid` will also no longer see the children.

For long-running parents that do care about exit status, install a reaper handler that drains all pending children without blocking:

```
$SIG{CHLD} = sub {
    while ((my $pid = waitpid -1, POSIX::WNOHANG) > 0) {
        # optionally inspect $? here
    }
};
```

See `perlipc` for fuller patterns.

What the child inherits

- **Open file descriptors:** shared. A `print` in either process advances the same underlying file offset. Close the descriptors the child does not need, and reopen handles the child should not share with the parent - particularly `STDIN/STDOUT/STDERR` when they are connected to a pipe or socket driving the parent's caller. A backgrounded CGI script that forks and exits without closing its inherited `STDOUT` will leave the HTTP client hanging, because the child still holds the socket open:

```
open STDIN, '<', '/dev/null' or die $!;
open STDOUT, '>', '/dev/null' or die $!;
open STDERR, '>', '/dev/null' or die $!;
```

- **Memory:** logically copied, physically copy-on-write. Mutating a large data structure in one side materialises only the touched pages.
- **Process ID:** changes. `$$` (also `$PID` / `$PROCESS_ID` under English) is re-read from the kernel on access after a fork, so both parent and child see their own PID.
- **Signal handlers, %ENV, working directory, umask, process group, controlling terminal:** all copied.
- **Pending alarms and timers:** not inherited by the child (per POSIX).
- **Locks held via flock:** shared with the parent - both processes hold the same lock on the same open file description. Locks acquired via `fcntl` record locks behave differently; consult `fcntl(2)`.

Examples

Classic fork/exec - launch an external command without going through the shell:

```
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    exec '/usr/bin/gzip', '-9', $file;
    die "exec: $!"; # only reached if exec fails
}
waitpid $pid, 0;
die "gzip failed: $" if $?;
```

Fan out N workers, then reap them all:

```
my @kids;
for my $i (1 .. 4) {
    my $pid = fork // die "fork: $!";
    if ($pid == 0) {
        do_work($i);
        exit 0;
    }
    push @kids, $pid;
}
waitpid $_, 0 for @kids;
```

Parent/child communication through a pipe. Open the pipe **before** forking so both sides inherit the two ends:

```
pipe(my $reader, my $writer) or die "pipe: $!";
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    close $reader;
    $writer->autoflush(1);
    print $writer "hello from $$\n";
    exit 0;
}
close $writer;
chomp(my $line = <$reader>);
waitpid $pid, 0;
print "got: $line\n";
```

Note the close on the unused end in each process - leave both ends open in both processes and the reader will never see EOF.

Detach a daemon-style child and let the kernel reap it:

```
$SIG{CHLD} = 'IGNORE';
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    # child: will be auto-reaped on exit
    run_background_task();
    exit 0;
}
# parent moves on, never calls wait
```

Edge cases

- **fork inside a thread or after DESTROY-sensitive code:** only the calling thread survives in the child; anything waiting on a mutex or joined thread in another thread is gone. Keep fork points before you spawn threads.
- **Buffered filehandles in the child:** any data still buffered at the moment of fork is present in both processes' buffers. When each side eventually flushes, the bytes are written twice. Autoflush before forking.

- **exit in the child runs END blocks and DESTROY-ers** written by the parent. That is rarely what you want - prefer `POSIX::_exit` in the child after an `exec` failure, or ensure END blocks check `$$` against the PID recorded at program start.
- **Error vs child confusion:** `fork` returns `0` legitimately in the child and `undef` on failure. Use `//` and `defined`, never `||` / truth tests.
- **EAGAIN:** transient - the kernel's per-user process limit is reached. Retry with a short sleep, or lower your concurrency.
- **Working directory and `chdir` in the child** do not affect the parent. The same is true of `%ENV`, `umask`, and signal handlers: the child has its own copy from the moment of `fork` on.
- **Seeks on shared descriptors are shared:** if parent and child both write to the same inherited `STDOUT`, their output interleaves at whatever the OS scheduler decides. Coordinate with explicit locking or give each child its own output handle.

Differences from upstream

Fully compatible with upstream Perl 5.42.

`pperl` targets Linux only, so `fork` is always a real `fork(2)`. Unlike traditional `perl5` on Windows - where `fork` is emulated with interpreter-level *pseudo-processes* sharing a single OS process, with the quirks documented in `perlfork` - the `pperl` `fork` creates an independent OS process with its own PID, address space, and process table entry. Code that assumed pseudo-process semantics (shared global state across “forked” sides) will not port to `pperl`.

See also

- `exec` - replace the current process image; the typical next call in a child right after `fork`
- `wait` - block until any child exits and return its PID and status in `$?`
- `waitpid` - reap a specific child, optionally non-blocking via `WNOHANG`
- `exit` - terminate the current process; use in the child to end it without falling through into the parent's code
- `pipe` - create a pair of connected filehandles before forking to give parent and child a communication channel
- `$$` - the current process ID, re-read after `fork` so each side sees its own PID
- `%SIG` - in particular `$SIG{CHLD}`, controlling whether children are auto-reaped or must be collected with `wait` / `waitpid`
- `perlipc` - fuller treatment of forking, signals, pipes, and reaping moribund children

I/O

format

Declare a picture-based report template for use by `write`.

`format` is a **declaration**, not an executable statement. It binds a named picture - a block of fixed-column output lines with embedded field markers - to the symbol table, where `write` later looks it up by name. Pictures do not interpolate variables; each picture line is followed by an argument line whose expressions fill the fields in order. `format` is a legacy feature of Perl; upstream notes its use is limited and points at `perlform` for the full field grammar.

Synopsis

```
format NAME =
FORMLIST
.

format =          # declares the "STDOUT" format
FORMLIST
.
```

A FORMLIST is a sequence of lines, each one of three kinds: a comment (# in column 1), a **picture line**, or an **argument line** supplying values for the picture line immediately above it. A single . in column 1 terminates the declaration.

What you get back

`format` is a declaration and has no runtime value. It installs the compiled picture into the current package's format table under NAME (or STDOUT if NAME is omitted). It is not called; it is triggered indirectly by `write`.

The pieces of a picture

Each picture line mixes literal text with **field markers**. A field begins with @ (regular field) or ^ (special field, used for fill mode and for blankable numeric fields) and extends through pad characters that set both the width and the justification:

- < - left-justified text
- > - right-justified text
- | - centered text
- # - right-justified numeric; a leading 0 pads with zeroes; . places the decimal point
- . . . - at the tail of a text field, renders three dots when the value was truncated
- @* - variable-width field that emits a multi-line scalar verbatim
- ^* - variable-width field that emits the next line of a scalar and consumes that portion of the source variable

Two suppression tokens may appear anywhere on a picture line and are not part of any field:

- `~` - if every field on this line is empty, the line is suppressed entirely; the `~` itself renders as a space otherwise
- `~~` - like `~`, and additionally repeats the line until every field on it is exhausted (every source scalar is empty or every [shift](#)-style expression returns undef)

`perlform.pod` gives the full grammar of picture lines including fill mode, multi-line fields, overflow (####), and locale-dependent decimal points.

The argument line

The line immediately below a picture line is the **argument line**: a comma-separated list of expressions, one per field, evaluated in list context before the picture is rendered. A single list-returning expression may fill multiple fields. The argument list may be broken across lines if and only if the first token on the first line is an opening brace:

```
format Report =
@<<<<<<<< @>>>>>>>
{ $name,
  $score }
.
```

Globals that control write

Output through `format` is driven entirely by per-filehandle globals. Each is attached to the **currently selected filehandle**; changing them for another handle requires a [select](#) dance (see below):

- `$~` - name of the current format used for each record (defaults to the filehandle's own name)
- `$^` - name of the current top-of-page format (defaults to the filehandle's name with `_TOP` appended)
- `$=` - page length in lines (default 60)
- `$-` - lines remaining on the current page; `write` decrements it and triggers the top-of-page format when it reaches zero
- `$%` - current page number, incremented automatically
- `$:` - characters that fill mode (`^`) may break on (default " \n- ")
- `$$L` - string emitted before each top-of-page except the first (default "\f")

Under use English these are `$FORMAT_NAME`, `$FORMAT_TOP_NAME`, `$FORMAT_LINES_PER_PAGE`, `$FORMAT_LINES_LEFT`, `$FORMAT_PAGE_NUMBER`, `$FORMAT_LINE_BREAK_CHARACTERS`, and `$FORMAT_FORMFEED`.

Because these are per-handle, setting them for a handle other than the currently selected one needs `select`:

```
my $ofh = select $fh;
$~ = 'Report';
$^ = 'Report_TOP';
select $ofh;
```

Top-of-form

When `write` decrements `$-` to zero, it emits `^L` (except on the very first page), resets `$-` to `=`, increments `$_`, and invokes the format named by `^`. The default top-of-page format name is the filehandle's name with `_TOP` appended, so declaring `STDOUT_TOP` suffices for writes through `STDOUT`.

Examples

A minimal report with a header:

```
format STDOUT_TOP =  
    Inventory  
Name                Qty      Price  
-----  
.   
format STDOUT =  
@<<<<<<<<<<< @>>   @##.##  
$name,             $qty, $price  
.  
  
for my $row (@rows) {  
    ($name, $qty, $price) = @$row;  
    write;  
}
```

Switching formats for a non-selected handle:

```
open my $fh, '>', 'report.txt' or die $!;
my $ofh = select $fh;
$~ = 'Report';
$^ = 'Report_TOP';
select $ofh;
write $fh;
```

Fill mode with `~~` to drain a multi-line scalar one line at a time:

[illegible]

The ^ fields consume \$note as they render; the ~~ line repeats until \$note is empty.

Edge cases

- **Lexical visibility:** a format sees a `my` variable only when declared inside that variable's scope. A top-level format cannot see `my` variables declared inside a subroutine; use package globals or declare the format inside the sub.

- **Quoted pictures:** if you build a picture with `eval` from a double-quoted string, `@name` looks like an array interpolation. Escape it or use single quotes around the template.
- **A lone `.` is the terminator:** a picture whose literal text would begin with `.` in column 1 must indent that line. This also bites format code sent through naive mail transports that cut at `.` on a line of its own.
- **Numeric overflow:** a `#`-field narrower than the formatted number renders as `####` rather than truncating silently.
- **Locale decimal point:** if `use locale` was in effect when the format was **declared**, `LC_NUMERIC` picks the decimal character. Toggling `use locale` at `write` time does not change the picture.
- **Control characters** inside a fixed-width text field are replaced by spaces to preserve column alignment.
- **Mixing `print` and `write`:** legal on the same handle, but `$-` only tracks lines emitted by `write`. Adjust `$-` manually if interleaved `print` output consumes page lines you care about.
- **No footer mechanism:** there is no `$FORMAT_BOTTOM_NAME`. Footers are done by hand - test `$-` before each `write` and emit the footer yourself when it reaches a threshold.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `write` - render one record through the current format into the selected filehandle; the only way a declared format produces output
- `select` - change which filehandle is current, required when setting `$~`, `$^`, `$=`, `$-`, `$_`, `$:`, or `$_L` for a handle other than `STDOUT`
- `$~` - current format name
- `$^` - current top-of-page format name
- `$=` - page length
- `$-` - lines left on the current page
- `$_` - current page number
- `$:` - fill-mode break characters
- `perlform.pod` - full picture-line grammar, fill-mode rules, `formline` / `$_A` internals, and worked report examples

Misc

formline

Format a list of values into a picture string and append the result to the format accumulator `$_^A`.

`formline` is the low-level engine behind Perl's `format` / `write` report mechanism. It takes a PICTURE string describing field widths and alignment, interpolates the values from LIST into those fields, and appends the resulting text to the global `$_^A` accumulator (also known as `$ACCUMULATOR` under `use English`). Every call to `write` flushes `$_^A` to the associated filehandle and clears it; outside of `write`, you manage `$_^A` yourself - read it for the formatted text, then reset it to `" "` before the next call.

PICTURE is the same mini-language that `format` uses: `@<<<<` for a left-justified text field, `@>>>>` for right-justified, `@| | |` for centered, `@####.##` for a right-justified numeric field with a decimal point, `@*` for an unbounded multi-line text field, and `^<<<` (and friends) for the fill-mode field that consumes its scalar argument incrementally. See `perldoc perlform` for the full picture vocabulary.

Synopsis

```
formline PICTURE, LIST
formline $picture, @values
```

What you get back

`formline` always returns 1. The useful output is the side effect: the formatted text is **appended** to `$_^A`. If you need the text as a value, read `$_^A` after the call and then clear it:

```
$_^A = "";
formline '@<<<<<<<< @>>>>', $name, $score;
my $line = $_^A;
$_^A = "";
```

Because the result goes to an accumulator, consecutive `formline` calls build up multi-line output without any intermediate join.

Global state it touches

- `$_^A` - the **format accumulator**. `formline` appends to it; it is never cleared automatically. Also known as `$ACCUMULATOR` under `use English`.
- `$:` - the set of characters on which fill-mode fields (`^*`, `^<<<`) are allowed to break. Defaults to `" \n "`. Also known as `$FORMAT_LINE_BREAK_CHARACTERS` under `use English`.
- `LC_NUMERIC` locale - if `use locale` is in effect when the picture is compiled, the decimal point character in numeric fields (`@####.##`) follows the locale.
- Fill-mode fields (`^*`) **mutate their scalar arguments in place**, chopping off the portion that was consumed so the next `formline` call picks up where the previous one left off. Pass a copy if you need the original preserved.

Edge cases

- **Quoting the picture:** inside double quotes, @ introduces an array interpolation. Write pictures as single-quoted strings or heredocs with a quoted terminator (<<'END') to keep @ literal. The classic trap: formline "@<<<<<<<", \$name interpolates @< before formline ever sees the string.
- **Newlines in the picture:** formline does not care how many newlines PICTURE contains. The ~ (suppress-if-all-empty) and ~~ (repeat-until-exhausted) tokens treat the whole picture as one logical line, so a record format with ~~ typically needs one formline per physical line. A **format** declaration splits a multi-line form into one formline call per output line; when calling formline directly, you make that split yourself.
- **Accumulator is not cleared:** the most common mistake is forgetting to reset \$^A between logical uses. Appending is the defined behaviour, not a bug. Use local \$^A = ""; inside helpers.
- **Field count vs. value count:** surplus values in LIST are silently ignored; missing values render as empty (regular field) or blanked (special ^ numeric field).
- **undef in a regular field** stringifies to the empty string under use warnings and triggers a uninitialized warning. A special field (^###) blanks the whole field on undef without warning.
- **Using a format scalar - capturing write output:** opening a filehandle onto a scalar reference is often easier than driving formline manually:

```
open my $fh, ">", \my $out or die $!;
format FH =
@<<<<<<< @>>>>
$name,      $score
.
$~ = "FH";
select((select($fh), $~ = "FH")[0]);
write $fh;
```

Reach for `formline` directly when you want the picture-to-string path without involving a filehandle or a named `format` declaration.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `write` - drives one or more `formline` calls from a named `format` and flushes `$$A` to a filehandle
- `format` - declaration syntax for the picture/value pairs that `write` feeds into `formline`
- `sprintf` - the other “format into a string” primitive; use `sprintf` for `printf`-style `%`-directives, `formline` for fixed-width report columns and fill-mode text blocks

- `$_A` - the accumulator formline writes into; read it for the result, clear it before the next logical use
- `$:` - characters that fill-mode fields (`^*`, `^<<<`) may break on

I/O

getc

Read the next single character from a filehandle.

`getc` returns one character of input from `FILEHANDLE`, advancing the handle's read position by that character. If `FILEHANDLE` is omitted, `getc` reads from `STDIN`. At end-of-file or on a read error, `getc` returns `undef` and - in the error case - sets `$!`. The unit is a **character**, not a byte: under a `:utf8` or `:encoding( . . . )` PerlIO layer a single call consumes however many bytes make up one decoded codepoint.

Synopsis

```
getc FILEHANDLE
getc
```

What you get back

A one-character string on success, or `undef` at end-of-file or on error. Distinguish the two by checking `$!`:

```
my $ch = getc $fh;
if (!defined $ch) {
    die "read error: $!" if $!;
    # otherwise clean EOF
}
```

Because `undef` collapses with the empty string `""` under stringy comparisons, always test with `defined`, never with `length` or a string match.

Global state it touches

- `$!` - set when `getc` returns `undef` because of a read error (not set for a clean end-of-file).
- The current input filehandle - the no-argument form reads from `STDIN`, not from the handle selected with `select`. `select` governs default output, not default input.
- The handle's PerlIO layer stack - determines whether the returned scalar is a byte or a decoded character, and whether CRLF translation is applied on the way in.

Examples

Read one character from `STDIN`:


```
my $ch =getc;
print "got: $ch\n" if defined $ch;
```

Read from an explicit filehandle and detect end-of-file:

```
open my $fh, "<", "input.txt" or die "open: $!";
while (defined(my $ch =getc $fh)) {
    print $ch;
}
close $fh;
```

Read characters, not bytes, from a UTF-8 file. Each call returns one decoded codepoint even when the underlying byte stream uses multi-byte sequences:

```
open my $fh, "<:encoding(UTF-8)", "utf8.txt" or die $!;
my $ch =getc $fh;          # e.g. "ä" as one character, even though
                           # it occupies two bytes on disk
```

Consume a yes/no prompt without waiting for a whole line - requires turning off terminal line buffering first (see *Edge cases*):

```
print "continue? [y/n] ";
system "stty", "-icanon", "eol", "\001";
my $key =getc(STDIN);
system "stty", "icanon", "eol", "^@";
print "\n";
exit unless defined $key and lc $key eq "y";
```

Edge cases

- **Terminal line buffering:** on an interactive STDIN, the terminal driver hands Perl nothing until the user presses Enter, so `getc` blocks until a full line is available and then returns only its first character. `getc` itself cannot bypass this. To read a single keypress, put the terminal into non-canonical (“cbreak”) mode first - shell out to `stty`, use POSIX: `:termios` via the POSIX module for a portable in-process solution, or use the CPAN module `Term::ReadKey` for a friendlier interface. Remember to restore the terminal mode on exit, including from signal handlers.
- **Bytes vs characters under I/O layers:** the return value is whatever the handle’s layer stack produces. A bytes-mode handle yields one byte per call; a `:utf8` or `:encoding(...)` handle yields one decoded character per call, consuming as many bytes off the stream as that character takes. Malformed input under a decoding layer triggers the layer’s usual warning or error rather than silently returning garbage.
- **End-of-file vs error:** both return `undef`. `$!` is cleared before the call is entered but only set when an actual read error occurred, so `defined $ch` or `$! and die` .. is the idiom.
- **Closed or unopened filehandle:** returns `undef` and sets `$!` to Bad file descriptor. Under use warnings a `getc()` on closed filehandle warning is emitted.

- **Efficiency:** `getc` is one-character-at-a-time, which crosses the PerlIO layer stack per call. For bulk scanning, `read`, `sysread`, or `readline` are substantially faster; reserve `getc` for interactive prompts and protocol parsers that genuinely need character-at-a-time control.
- **Default handle is STDIN, not ARGV:** unlike the diamond operator `<>`, `getc` without an argument does not walk `@ARGV` or fall back to the magic `ARGV` handle - it reads `STDIN` directly.
- **Wide terminal input:** reading a character from a terminal in raw mode still returns one codepoint only when the input layer decodes UTF-8. A paste containing a multi-byte character under a bytes-mode handle surfaces as several separate `getc` calls, one per byte.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `read` - read a fixed number of characters or bytes in one call, much faster than looping `getc`
- `readline` - read one whole record (line) at a time, governed by `$/`
- `sysread` - unbuffered read straight from the OS, bypassing the PerlIO layer stack
- `eof` - test whether the next `getc` / `readline` would return `undef` for end-of-file
- `$!` - `errno` set when `getc` returns `undef` because of a read error

User and group info

getgrent

Read the next entry from the system group database.

`getgrent` walks the host's group database one entry at a time, the way a plain loop over `/etc/group` would read it line by line, and returns a Perl-shaped view of each record. It is the iterator form of the group lookups - use it when you want every group, or every group matching some predicate. For single-entry lookup by name or gid, reach for `getgrnam` or `getgrgid` instead.

A first call to `getgrent` implicitly opens the database. Iteration continues across further calls until the database is exhausted, at which point `getgrent` returns the empty list (or `undef` in scalar context). Call `setgrent` to rewind to the start, or `endgrent` to close the database and free any resources the C library is holding.

Synopsis

```
my ($name, $passwd, $gid, $members) = getgrent;
my $name                             = getgrent;           # scalar
↪ context
```

What you get back

In **list context**, a four-element list - one entry per group:

Index	Field	Meaning
0	\$name	Group name (e.g. "wheel")
1	\$passwd	Group password, typically "x" on shadow-password hosts
2	\$gid	Numeric group id
3	\$members	Space-separated string of member login names

When iteration runs off the end, the return is the empty list. If you write `if (my @gr = getgrent)` the loop terminates cleanly on end-of-database.

In **scalar context**, `getgrent` returns just `$name`. When the database is exhausted, it returns `undef` - which is the signal most scripts test against to end a `while` loop.

`$members` is a plain string, not an array. Split it on whitespace when you need the list:

```
my @logins = split ' ', $members;
```

Global state it touches

`getgrent` keeps **per-process iteration state** inside the C library (conceptually a file position inside the group database). Two calls in the same process share that state; a nested loop that calls `getgrent` inside another `getgrent` loop will interleave records and miss most of them. Serialise the passes, or capture all entries into an array first:

```
my @all;
while (my @gr = getgrent) { push @all, [ @gr ] }
endgrent;
```

The iterator is not reset when a subroutine returns. If library code you do not control calls `getgrent`, it moves your position. Guard with `setgrent` when you need a clean start.

Examples

Walk every group and print the name and member count:

```
while (my ($name, undef, $gid, $members) = getgrent) {
    my $n = length $members ? scalar(split ' ', $members) : 0;
    printf "%-16s gid=%d members=%d\n", $name, $gid, $n;
}
endgrent;
```

Collect all groups a given login belongs to:

```
my $user = 'alice';
my @groups;
while (my ($name, undef, undef, $members) = getgrent) {
```

(continues on next page)

(continued from previous page)

```
    push @groups, $name
    if grep { $_ eq $user } split ' ', $members;
}
endgrent;
```

Scalar context - list every group name, one per line:

```
while (defined(my $name = getgrent)) {
    print $name, "\n";
}
endgrent;
```

Rewind and re-read, for example to build two indexes in one pass each:

```
setgrent;
my %by_name = map { $_->[0] => $_ }
               map { [ getgrent ] } 1 .. (); # collect...
# ...or just loop twice with an explicit setgrent before each pass.
setgrent;
while (my @gr = getgrent) { ... }
endgrent;
```

Convert a numeric gid from `stat` to a group name without a second lookup by caching the iterator output:

```
my %name_of;
while (my ($name, undef, $gid) = getgrent) {
    $name_of{$gid} = $name;
}
endgrent;

my $gid = (stat $path)[5];
print "group: $name_of{$gid}\n";
```

Edge cases

- **End of iteration vs. a real entry:** in list context, an exhausted database returns the empty list - which assigns as zero elements, so `my @gr = getgrent; @gr` or `last`; is the idiomatic termination test. In scalar context, test with `defined` rather than truthiness, since a legitimate group name could in principle be the string `"0"`.
- **No password field on modern Linux:** `$passwd` is almost always `"x"` (real hashes live in `/etc/gshadow`). Do not treat a non-`"x"` value as “password set” without consulting the shadow database.
- **Empty `$members`:** groups with no explicit members (users listed only by primary gid) have `$members` as the empty string. `split ' ', ""` returns an empty list - the idiomatic way to get members as an array without a special case.
- **Implicit primary-gid membership is not reported:** `$members` lists only the users explicitly named in the group’s `/etc/group` record. A user whose primary gid

matches this group but who is not also listed in the `members` field does **not** appear. Cross-check with `getpwent` if you need both kinds of membership.

- **Nested iteration clobbers position:** a second `getgrent` loop started before the first one finishes shares the same cursor. Use `setgrent` between passes, or snapshot the database into memory up front.
- **Taintedness:** the `$passwd` field is tainted under `-T`, same as for `getpwent` - treat it as untrusted input even though the usual value is `"x"`.
- **Process-wide, not thread-wide:** on systems without thread-safe variants the iteration state is per process, not per thread. Two threads walking `getgrent` at once will miss records.
- **endgrent is optional but polite:** leaving the database open to program exit works, but if you are done iterating, call `endgrent` so the C library can drop its buffers.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `getgrnam` - single lookup by group name; use when you already know the name and want just one record
- `getgrgid` - single lookup by numeric gid; the usual partner of `stat` when you need a group name for a file
- `setgrent` - rewind the iterator to the start of the group database before another full-scan pass
- `endgrent` - close the database and release any resources the C library holds open
- `getpwent` - the parallel iterator over the passwd database; pair with `getgrent` when resolving full user-to-group relationships
- `scalar` - force scalar context when you only need the group name and want to skip the other fields

User and group info

`getgrgid`

Look up a group record by numeric group ID.

`getgrgid` asks the system group database for the entry whose `gid` is `GID` and returns what it finds. It is the numeric counterpart of `getgrnam` and mirrors the C library call `getgrgid(3)`. In list context you get the unpacked record fields; in scalar context you get the group name.

Synopsis

```
my ( $name, $passwd, $gid, $members ) = getgrgid($gid);
my $name                               = getgrgid($gid);
```

What you get back

In **list context**, four values unpacked from struct group:

```
#      0      1      2      3
my ( $name, $passwd, $gid, $members ) = getgrgid($gid);
```

- `$name` - the group name.
- `$passwd` - the group password field. On modern systems this is typically "x" or empty; real hashes live in a shadow database.
- `$gid` - the numeric group ID (the same number you passed in, echoed back from the record).
- `$members` - a **single space-separated string** of member login names, not a list. Split it yourself when you need an array:

```
my @users = split / /, (getgrgid($gid))[3];
```

In **scalar context**, just the group name:

```
my $name = getgrgid(0);    # "root" on most Linux systems
```

If no group with the given GID exists, the function returns the empty list in list context and `undef` in scalar context. A system-level failure (e.g. NSS backend unreachable) is reported the same way; check `$!` if you need to distinguish.

Global state it touches

- `$!` - set on system-level failure of the underlying `getgrgid(3)` call. Not set when the GID simply has no matching entry; that is a normal “not found”, not an error.
- The process-wide group-database iteration cursor maintained by libc. `getgrgid` does **not** advance or reset the cursor used by `getgrent`, `setgrent`, and `endgrent`, but it shares the same underlying file descriptor on most implementations - interleaving direct lookups with iteration is portable in theory and surprising in practice.

Examples

Look up the primary group name for the current process:

```
my $group = getgrgid($());    # $( is the real GID
print "running as group: $group\n";
```

Resolve a file's group ID to a name, with a sensible fallback for orphaned GIDs (files owned by a group that no longer exists):

```
my $gid    = (stat $file)[5];
my $name   = getgrgid($gid) // "<unknown gid $gid>";
```

Unpack the full record and list the members:

```
if (my ($name, undef, $gid, $members) = getgrgid(100)) {
    my @users = split / /, $members;
    print "group $name ($gid) has ", scalar(@users), " members\n";
}
```

Guard against the “no such group” case explicitly - the empty list in boolean/list context is already falsy, so an `if` suffices:

```
unless (my @gr = getgrgid($gid)) {
    die "no group with gid $gid\n";
}
```

Pair with `getpwuid` to render a directory listing’s owner and group columns:

```
my @st    = stat $path                      or die "stat $path: $!";
my $user   = getpwuid($st[4]) // $st[4];
my $grp    = getgrgid($st[5]) // $st[5];
printf "%-8s %-8s %s\n", $user, $grp, $path;
```

Edge cases

- **Scalar vs list context matters.** `getgrgid($gid)` in scalar context yields only the group name; wrap in parentheses or assign to a list to get the full record:

```
my $name    = getgrgid(0);                # "root"
my @record  = getgrgid(0);                # four-element list
my $gid_out = (getgrgid(0))[2];           # the gid field
```

- **\$members is a string, not an array.** The whitespace-joined form is historical and matches perl5’s `pp_gpwent / pp_ggrent` output. Splitting on `/ /` (single space) is the canonical idiom; splitting on `/\s+/` is slightly more forgiving if a backend ever uses tabs.
- **Empty groups** return `$members` as the empty string. `split / /, ""` yields the empty list, which is what callers usually want.
- **Numeric coercion.** GID is passed to the C call as an integer. A non-numeric scalar is coerced: `"0root"` becomes `0`. Pass a clean integer.
- **GID namespace overlap.** Multiple group entries with the same gid are legal in `/etc/group` and in NSS. `getgrgid` returns **one** of them - which one is backend-defined. Do not rely on which name comes back when GIDs collide; use `getgrnam` for the reverse direction when the name matters.
- **NSS / `nsswitch.conf`.** The lookup follows the system’s name service switch - files, sss, ldap, etc. A successful lookup in a test harness on one host may fail on another with a different NSS configuration. This is a property of the C library, not of Perl.

- **Large groups.** Very large group-member lists (hundreds of members) can arrive truncated if the backend's internal buffer is smaller than the rendered line. This is a libc / NSS limitation; Perl faithfully surfaces whatever the system returned.
- **Threaded builds.** On systems that provide `getgrgid_r`, perl transparently calls the reentrant variant. Callers see no difference.
- **Taint.** Under `-T`, `$passwd` and `$members` come back tainted - the first because it may encode credentials, the second because administrators can change it. `$name` and `$gid` are not tainted.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- [getgrnam](#) - the inverse lookup: group record by name rather than by gid
- [getgrent](#) - iterate every group entry when you need to scan rather than look up
- [setgrent](#) and [endgrent](#) - reset and release the iteration cursor used by [getgrent](#)
- [getpwuid](#) - the passwd-side counterpart; pair with [getgrgid](#) to render owner/group columns
- `User::grent` - by-name accessor object that overrides [getgrgid](#) when you prefer `$gr->name` over positional unpacking

User and group info

getgrnam

Look up a Unix group by name and return its `/etc/group` record.

`getgrnam` wraps the C library's `getgrnam(3)`. Given a group name like `"wheel"` or `"staff"`, it returns the matching record from the system's group database - the name, the encrypted password placeholder, the numeric GID, and the list of member login names. Use it when you have a group name in hand and need the GID or the membership list; use [getgrgid](#) for the inverse lookup.

Synopsis

```
my ($name, $passwd, $gid, $members) = getgrnam NAME
my $gid = getgrnam NAME
```

What you get back

List context yields a four-element list:

```
my ($name, $passwd, $gid, $members) = getgrnam "wheel";
```


Index	Field	Meaning
0	\$name	canonical group name (same as NAME on hit)
1	\$passwd	group password field, typically "x" or empty
2	\$gid	numeric group ID
3	\$members	space-separated string of member login names

\$members is a single scalar, **not** an array. Split it yourself when you want a list:

```
my @logins = split /\s+/, (getgrnam("wheel"))[3];
```

Scalar context yields the numeric GID (the “other thing”, since the lookup was by name):

```
my $gid = getgrnam "wheel";
```

No such group: list context returns the empty list; scalar context returns `undef`. Test scalar returns for definedness rather than truth - GID 0 (root) is a valid hit that is also false:

```
defined(my $gid = getgrnam $group) or die "no group '$group'";
```

Global state it touches

None directly. `getgrnam` is a stateless lookup - unlike `getgrent`, it does not advance the group-database iterator, and `setgrent` / `endgrent` do not affect it. It does, however, share the underlying C library’s static result buffer with other `getgr*` calls on platforms without thread-safe variants; copy the fields you care about before calling another `getgr*` function.

Examples

Resolve a group name to its GID:

```
my $gid = getgrnam "wheel";
defined $gid or die "no 'wheel' group on this system";
print "wheel GID = $gid\n";
```

List the members of a group:

```
my @fields = getgrnam "staff"
    or die "no 'staff' group";
my @members = split /\s+/, $fields[3];
print "staff has ", scalar(@members), " members: @members\n";
```

Use a group name to set the effective GID (requires privilege):

```
my $gid = getgrnam "video"
    // die "no 'video' group";
$) = "$gid $gid";           # effective + supplementary
```

Check whether a user is listed as a group member. Note that the primary group (the user's GID in `/etc/passwd`) is **not** in `$members` - check both:

```
sub in_group {
    my ($user, $group) = @_;
    my @g = getgrnam $group or return 0;
    return 1 if grep { $_ eq $user } split /\s+/, $g[3];
    my @u = getpwnam $user or return 0;
    return $u[3] == $g[2];           # primary group match
}
```

Object-oriented form via `User::grent`, which overrides the built-in with a by-name record:

```
use User::grent;
my $g = getgrnam "wheel";
printf "wheel: gid=%d members=%s\n", $g->gid, "@{$g->members}";
```

Edge cases

- **Missing group:** list context returns `()`, scalar returns `undef`. A bare `if (getgrnam $g)` is fine in list context but wrong in scalar context when the GID could be `0`:

```
if (defined(my $gid = getgrnam $g)) { ... } # correct
if (my $gid = getgrnam $g)           { ... } # wrong for GID 0
```

- **Members is a string, not a list.** Forgetting this and writing scalar `@members = getgrnam $g` (or indexing `$members[0]`) silently gives wrong results. Always `split /\s+/` the fourth field.
- **Primary-group members are absent from the list.** A user whose `/etc/passwd` GID matches the group is a member in the kernel's eyes but will not appear in `$members`. Lookups that classify users by group must consult `getpwnam` too.
- **Password field is not a password.** On every modern Linux the second field is `"x"` (shadow redirection) or empty. It is never the plaintext or hash of a group password. Do not rely on its format.
- **Case sensitivity:** group names are case-sensitive on Linux. `getgrnam "Wheel"` does not find `wheel`.
- **Name with embedded whitespace or colon:** the group database format does not allow either, so such names cannot exist; the lookup simply fails.
- **NSS backends:** on systems configured with LDAP, SSSD, or other name-service switch modules, `getgrnam` consults those as well as `/etc/group`. Results therefore depend on `/etc/nsswitch.conf` and the configured backends, not on `/etc/group` alone.
- **Taint:** the fields come from outside the program, so under `-T` they are tainted and must be laundered before use in sensitive operations.
- **Thread safety:** `pperl` automatically substitutes the thread-safe `getgrnam_r` when the platform provides it. On platforms without it, concurrent `getgr*` calls across

threads can clobber each other's results.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `getgrgid` - inverse lookup by numeric GID; reach for this when you have the GID and want the group name or member list
- `getgrent` - iterate every group in the database when you do not know the name or GID in advance
- `setgrent` - rewind the `getgrent` iterator to the start of the group database
- `endgrent` - close the group-database file descriptor opened by iteration
- `getpwnam` - the parallel call for the user database; use alongside `getgrnam` when you need to resolve a user's primary group
- `User::grent` - object-oriented wrapper that overrides `getgrnam` to return a record object with named accessors (`name`, `passwd`, `gid`, `members`) instead of a positional list

Network info

gethostbyaddr

Look up a host record by its packed IP address.

`gethostbyaddr` performs a reverse DNS lookup: given a packed binary address (four bytes for IPv4, sixteen for IPv6) and the matching address family, it calls the system's C `gethostbyaddr(3)` routine and returns the host record. The typical use case - *"whose machine is this packet from?"* - wants the scalar-context form: a single hostname. The list-context form exposes the full struct `hostent`: canonical name, aliases, address family, address length, and every address the resolver knows for that host.

Synopsis

```
my $name = gethostbyaddr $packed_addr, $addrtype;
my ($name, $aliases, $addrtype, $length, @addrs) = gethostbyaddr
    ↪ $packed_addr, $addrtype;
```

What you get back

Scalar context - the canonical hostname as a plain string, or `undef` if the reverse lookup fails.

List context - a five-element list mirroring the C struct `hostent`:

- `$name` - the canonical hostname the resolver mapped the address to.
- `$aliases` - a space-separated string of alternate names. Single scalar, not a list.

- `$addrtype` - the address family of the returned addresses, typically `AF_INET` (2) or `AF_INET6` (10).
- `$length` - the length of each address in bytes (4 for IPv4, 16 for IPv6).
- `@addrs` - every address returned for the host, each a packed binary string of `$length` bytes. Often just one, but multi-homed hosts can yield several.

On lookup failure in list context, the return is the empty list.

Global state it touches

- `$?` - set to the C `h_errno` value on failure, when the platform exposes it. Test with `$? != 0` to distinguish `HOST_NOT_FOUND` from `TRY_AGAIN` and similar transient errors.
- `$!` - not meaningfully set by DNS failures. Reverse lookups fail via `h_errno`, not `errno`; do not check `$!` after a failed call.
- The host database iteration cursor shared with `gethostent`, `sethostent`, and `endhostent`. A call to `gethostbyaddr` may reset a cursor left open by `gethostent`.

Examples

Scalar form - map a dotted-quad to a hostname:

```
use Socket;
my $packed = inet_aton("127.0.0.1");
my $name    = gethostbyaddr($packed, AF_INET);
print defined $name ? "$name\n" : "no PTR record\n";
```

Round-trip - forward lookup, then reverse the first address:

```
use Socket;
my $packed = gethostbyname("www.perl.org");
my $name    = gethostbyaddr($packed, AF_INET);
print "$name\n";
```

Identify the peer on an accepted socket connection:

```
use Socket;
my $peer = getpeername($sock);
my ($port, $iaddr) = sockaddr_in($peer);
my $host = gethostbyaddr($iaddr, AF_INET);
printf "connection from %s:%d\n", $host // inet_ntoa($iaddr), $port;
```

List form - every name and address the resolver knows for the host:

```
use Socket;
my $packed = inet_aton("8.8.8.8");
my ($name, $aliases, $type, $len, @addrs) = gethostbyaddr($packed, AF_INET);
```

(continues on next page)

(continued from previous page)

```
print "canonical: $name\n";
print "aliases:  $_\n" for split ' ', $aliases;
print "address:  ", inet_ntoa($_), "\n" for @addrs;
```

By-name interface via the `Net::hostent` override - accessor methods instead of positional unpacking:

```
use Net::hostent;
use Socket;
my $h = gethostbyaddr(inet_aton("127.0.0.1"), AF_INET);
print $h->name, "\n" if $h;
```

Edge cases

- **ADDR is a packed binary string, not a dotted-quad.** Passing "127.0.0.1" directly does not work - Perl will call the resolver with nine raw bytes and get nothing useful back. Always go through `Socket::inet_aton`, `Socket::inet_pton`, or an equivalent packer first.
- **ADDRTYPE must match the address width.** `AF_INET` wants a four-byte ADDR; `AF_INET6` wants sixteen. A mismatch either returns `undef` or, on some platforms, reads past the buffer. Use the constant from `Socket`, not a hand-typed integer.
- **Check for `undef` on the scalar form.** A missing PTR record returns `undef`, not an empty string. Many addresses on the public internet have no reverse mapping at all.
- **@addrs may not include the address you queried.** The resolver returns the full record for the canonical host, which may contain additional addresses that happen to share the hostname.
- **Scalar context runs the full resolver call anyway.** The C routine always populates the whole hostent; Perl just discards the list. There is no performance advantage to the scalar form beyond clearer code.
- **Reverse of a forward lookup is not guaranteed to round-trip.** `gethostbyaddr(gethostbyname($h), AF_INET)` may return a different name from `$h` - reverse DNS maps to the canonical hostname, which often differs from the alias the forward query used.
- **IPv6 support is platform-dependent.** Not every libc's `gethostbyaddr(3)` handles `AF_INET6` cleanly. For portable dual-stack reverse resolution use `Socket::getnameinfo`.
- **Thread safety.** The underlying C routine keeps per-process static state for the host cursor. Concurrent calls from multiple threads may interleave results. Scripts that need thread-safe reverse resolution should use `Socket::getnameinfo`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `gethostbyname` - the forward lookup: hostname → packed address
- `gethostent` - iterate every entry in the host database instead of looking up one address
- `sethostent` - rewind the host iterator and control whether the file stays open across lookups
- `Socket::inet_aton` - build the packed ADDR argument from a dotted-quad string
- `Socket::inet_ntoa` - render a packed address from @addrs back to dotted-quad form
- `Socket::getnameinfo` - modern, protocol-agnostic reverse resolver that handles IPv6 and is thread-safe

Network info

gethostbyname

Look up a host record by DNS name.

`gethostbyname` resolves a hostname to its IP address(es) and canonical record by calling the system's `C gethostbyname(3)` routine. The typical use case - “*give me the packed IP of this host*” - wants the scalar-context form and nothing more. The list-context form exposes the full host record: canonical name, aliases, address family, address length, and every address on file.

Synopsis

```
my $packed_ip = gethostbyname $name;
my ($name, $aliases, $addrtype, $length, @addrs) = gethostbyname
    ↪ $name;
```

What you get back

Scalar context - the first IP address as a packed binary string (four bytes for IPv4, sixteen for IPv6), or `undef` if the lookup fails. Convert to a dotted-quad with `Socket::inet_ntoa`:

```
use Socket;
my $packed = gethostbyname("www.perl.org");
my $dotted = inet_ntoa($packed) if defined $packed;
```

List context - a five-element list mirroring the C struct `hostent`:

- `$name` - the canonical hostname.
- `$aliases` - a space-separated string of alternate names.

- `$addrtype` - the address family, typically `AF_INET` (2) or `AF_INET6` (10).
- `$length` - the length of each address in bytes (4 for IPv4, 16 for IPv6).
- `@addrs` - every address returned for the name, each a packed binary string of `$length` bytes.

On lookup failure in list context, the return is the empty list.

Global state it touches

- `$?` - set to the `C h_errno` value on failure, when the platform exposes it. Test with `$? != 0` to distinguish “no such host” (`HOST_NOT_FOUND`) from “try again” (`TRY_AGAIN`) and similar.
- The host database iteration cursor shared with `gethostent`, `sethostent`, and `endhostent`. A call to `gethostbyname` may reset a cursor left open by `gethostent`.

Examples

Scalar form - the common case:

```
use Socket;
my $packed = gethostbyname("localhost");
defined $packed or die "cannot resolve localhost: $!";
print inet_ntoa($packed), "\n";      # 127.0.0.1
```

List form - enumerate every address for a multi-homed host:

```
use Socket;
my ($name, $aliases, $type, $len, @addrs) = gethostbyname("www.
→example.com");
for my $addr (@addrs) {
    print inet_ntoa($addr), "\n";
}
```

Unpack an IPv4 address by hand, without `Socket`:

```
my $packed = gethostbyname("127.0.0.1");
my ($a, $b, $c, $d) = unpack("W4", $packed);
print "$a.$b.$c.$d\n";              # 127.0.0.1
```

Forward then reverse - resolve a name, then look the address back up:

```
use Socket;
my $packed = gethostbyname("www.perl.org");
my $back    = gethostbyaddr($packed, AF_INET);
print "$back\n";
```

By-name interface via the `Net::hostent` override - accessor methods instead of positional unpacking:


```
use Net::hostent;
my $h = gethostbyname("www.perl.org");
print $h->name, " ", join(" ", @{$h->aliases}), "\n";
```

Edge cases

- **Always call in scalar context when you want “the IP”.** The list-context form is never what a quick resolver wants and allocates four extra return values. The upstream docs flag this explicitly: *“Make sure gethostbyname is called in SCALAR context and that its return value is checked for definedness.”*
- **Check for `undef` on the scalar form.** A failed lookup returns `undef`, not an empty string. A literal `0.0.0.0` address, by contrast, is a valid four-byte packed string and stringifies to the zero byte sequence, not to `undef`.
- **@addrs may be empty** in list context even when the first four return values are defined - some resolvers produce a record with no addresses. Always iterate, don’t index `[0]` blindly.
- **Aliases are one scalar, not a list.** `$aliases` is a single space-separated string. Split on whitespace if you need a list.
- **Numeric IP strings are accepted.** Passing `"127.0.0.1"` returns the packed form of that address without a DNS round-trip - the resolver short-circuits numeric input.
- **IPv6.** `gethostbyname` queries only the IPv4 (AF_INET) address family. For dual-stack resolution use `Socket::getaddrinfo`, which returns records for every address family the host offers.
- **Thread safety.** The underlying C routine keeps per-process static state for the host cursor. Concurrent calls from multiple threads may interleave results. Scripts that need thread-safe resolution should use `Socket::getaddrinfo`.
- **`h_errno` vs `$!`.** DNS failures set `h_errno` (exposed via `$?`), not `errno` / `$!`. Checking `$!` after a failed `gethostbyname` is meaningless.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `gethostbyaddr` - the reverse lookup: packed address → host record
- `gethostent` - iterate every entry in the host database instead of looking up one name
- `sethostent` - rewind the host iterator and control whether the file stays open across lookups
- `Socket::inet_ntoa` - turn the packed scalar-context result into a dotted-quad string

- `Socket::inet_aton` - the pure name-to-packed helper when you don't need the full host record
- `unpack` - decode the packed address manually when `Socket` is not available

Network info

gethostent

Fetch the next entry from the host database.

`gethostent` walks the host database - on traditional Unix systems the `/etc/hosts` file, on modern systems whatever NSS source is configured

- returning one record per call until the database is exhausted. Use it when you want to enumerate every known host, not when you want to resolve a single name: `gethostbyname` is the right tool for a targeted lookup.

The iteration cursor is shared with `sethostent` (which rewinds it) and `endhostent` (which closes it). Between those two calls `gethostent` yields each record in order, then returns the empty list (or `undef` in scalar context) once the end of the database is reached.

Synopsis

```
my ($name, $aliases, $addrtype, $length, @addrs) = gethostent;  
my $name = gethostent;
```

What you get back

List context - a five-element list mirroring the C struct `hostent`:

- `$name` - the canonical hostname for this entry.
- `$aliases` - a space-separated string of alternate names.
- `$addrtype` - the address family, typically `AF_INET` (2) or `AF_INET6` (10).
- `$length` - the length in bytes of each address (4 for IPv4, 16 for IPv6).
- `@addrs` - every address recorded for this host, each a packed binary string of `$length` bytes.

At end-of-database the return is the empty list. A distinction worth keeping in mind: the upstream `perlfunc` note that a missing entry may return “*a single meaningless true value*” applies to the by-name and by-address lookups, not to `gethostent` - iteration past the end is a clean empty list.

Scalar context - the canonical `$name` only, or `undef` at end-of-database. Use list context whenever you need anything beyond the name; reconstructing the other fields is not possible after the fact.

Convert a packed address from `@addrs` with `Socket::inet_ntoa` for IPv4 or `unpack` directly:

```
use Socket;
print inet_ntoa($addrs[0]), "\n";
```

Global state it touches

- The **host database cursor**, shared with `sethostent`, `endhostent`, `gethostbyname`, and `gethostbyaddr`. A by-name or by-address lookup may reset or invalidate a cursor left mid-iteration by `gethostent`; don't interleave the two styles without a fresh `sethostent` between them.
- `$?` - set to the C `h_errno` value on failure, where the platform exposes it. A plain end-of-database is not a failure and leaves `$?` alone.
- `$!` - `gethostent` is a database iterator, not a syscall wrapper in the `open/read` sense. DNS and NSS errors surface through `h_errno` (via `$?`), not through `$!`.

Examples

Enumerate every host in the database:

```
use Socket;
sethostent(1);
while (my ($name, $aliases, $type, $len, @addrs) = gethostent) {
    my @dotted = map { inet_ntoa($_) } @addrs;
    print "$name @dotted\n";
}
endhostent;
```

Scalar context - list just the canonical names:

```
sethostent(0);
while (defined(my $name = gethostent)) {
    print "$name\n";
}
endhostent;
```

Build a name-to-address hash in one pass, then look up entries locally without further DNS calls:

```
use Socket;
my %ip_of;
sethostent(1);
while (my ($name, $aliases, undef, undef, @addrs) = gethostent) {
    next unless @addrs;
    $ip_of{$name} = inet_ntoa($addrs[0]);
    $ip_of{$_} = inet_ntoa($addrs[0]) for split ' ', $aliases;
}
endhostent;

print $ip_of{"localhost"} // "not in hosts DB", "\n";
```

By-record interface via the `Net::hostent` override - accessor methods instead of positional unpacking:

```
use Net::hostent;
while (my $h = gethostent) {
    printf "%-30s %s\n", $h->name, join(",", @{$h->aliases});
}
```

Edge cases

- **No implicit rewind.** `gethostent` does not open or rewind the database on its own. First call after program start works because the C library lazily opens the database, but once the end is reached further calls keep returning end-of-database until a `sethostent` rewinds the cursor. Always pair enumeration with an explicit `sethostent` / `endhostent`.
- **STAYOPEN argument to `sethostent`.** Passing a true value asks the library to keep the database file open across lookups, which matters for NSS sources that otherwise reconnect on every call. Passing `0` permits the library to close and reopen between calls.
- **End-of-database is the empty list, not `undef`.** In list context the loop condition `while (my @r = gethostent)` works because the empty list is false. In scalar context use `while (defined(my $name = gethostent))` - a hostname of `"0"` is theoretically valid and would falsely terminate a truthiness-based loop.
- **@`addrs` may be empty** even for a returned record on some NSS backends. Guard the address-unpacking step:

```
next unless @addrs;
```

- **Aliases are one scalar, not a list.** `$aliases` is a single space-separated string. Split on whitespace if you want a list.
- **NSS source mix.** On glibc-based systems the hosts database may draw from `/etc/hosts`, DNS, mDNS, and other NSS modules depending on `nsswitch.conf`. DNS-backed NSS modules typically do **not** enumerate - `gethostent` against a DNS-only hosts database returns the empty list immediately. On most real deployments only the `files` source contributes entries to iteration.
- **Interleaving with `by-name` / `by-address` lookups.** A call to `gethostbyname` or `gethostbyaddr` can reset the shared cursor. Finish the iteration first, or restart it with `sethostent` after the lookup.
- **Thread safety.** The underlying C routine keeps per-process static state for the iteration cursor. Concurrent calls from multiple threads interleave results unpredictably. Confine enumeration to a single thread.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `sethostent` - rewind the iterator and control whether the database stays open across calls
- `endhostent` - close the iterator and release the database handle
- `gethostbyname` - targeted lookup by name instead of full enumeration
- `gethostbyaddr` - targeted reverse lookup by packed address
- `getnetent` - same iteration pattern for the networks database
- `Socket::inet_ntoa` - turn a packed address from `@addrs` into a dotted-quad string

User and group info

getlogin

Return the login name of the user associated with the controlling terminal.

`getlogin` wraps the C library function of the same name. On Linux it consults the kernel's record of who owns the current controlling terminal - traditionally recorded in `/var/run/utmp` - and returns that name as a string. It answers the question “*who logged in on this tty?*”, which is **not** the same question as “*who is running this process?*”. A process that has changed its real or effective UID, or that has no controlling terminal at all (a daemon, a cron job, a container entrypoint), will usually see `getlogin` return the empty string or `undef`.

Synopsis

```
my $name = getlogin;
```

What you get back

A string - the login name recorded for the controlling terminal - or the empty string / `undef` when no such record is available. Treat both empty-string and undefined as “unknown” and fall back to a UID-based lookup:

```
my $login = getlogin || getpwuid($<) || "Kilroy";
```

Using `||` here is deliberate: the empty string and `undef` both mean “no answer” and both should trigger the fallback.

Global state it touches

- Reads `$<` indirectly only via the idiomatic `getpwuid` fallback above; `getlogin` itself takes no arguments and consults no Perl variables.
- Sets `$!` on the underlying system-call failure paths exposed by the C library.

Examples

The canonical who-is-this-user pattern, robust against detached sessions and UID changes:

```
my $login = getlogin || getpwuid($<) || "Kilroy";
print "hello, $login\n";
```

Bare call in a shell-attached script:

```
my $name = getlogin;
print defined $name && length $name
  ? "tty owner: $name\n"
  : "no controlling tty\n";
```

Contrast with `getpwuid`, which answers a different question

- “*who is the effective user running this code?*”:

```
my $tty_user = getlogin           // "";
my $proc_user = getpwuid($<);
print "tty=$tty_user proc=$proc_user\n";
# After `sudo -u bob script.pl`, tty_user is your login,
# proc_user is "bob".
```

Inside a daemon or a cron job there is no controlling terminal, so `getlogin` has nothing to report:

```
# running from cron
my $name = getlogin;
# $name is "" or undef; always pair with a getpwuid fallback
```

Edge cases

- **No controlling terminal:** daemons, systemd services, cron jobs, container entry-points, and anything started without a tty see the empty string or `undef`. This is the common case outside interactive shells - always provide a fallback.
- **sudo and su:** `getlogin` returns the **original** login name (the tty owner), not the target user. After `sudo -u bob script.pl` you get your own login, while `getpwuid` with `$<` returns bob. Pick the one that matches the question you are actually asking.
- **Not an authentication primitive:** the record `getlogin` reads can be forged or stale; a process that inherits a tty from an unrelated session can see a name that has nothing to do with its own credentials. Never gate privileged behaviour on the return value. Use `getpwuid` on `$<` / `$>` instead.
- **Empty string vs undef:** platforms differ on which they return when there is no record. The `||` fallback handles both; a defined-only check does not.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `getpwnuid` - resolves `$<` or `$>` to a name, and is the correct building block for “*who is running this process?*”
- `getpwnam` - the reverse lookup when you already have a login name and need the passwd entry
- `getpwent` - iterate the whole passwd database when you need more than one entry
- `$<` - real UID; the natural argument to `getpwnuid` in the `getlogin` fallback idiom
- `$>` - effective UID; use this instead of `$<` when you care about the identity under which privileged operations will actually run

Network info

getnetbyaddr

Look up a network record by its numeric network address.

`getnetbyaddr` translates a numeric network number and address family into a network database entry by calling the system’s C `getnetbyaddr(3)` routine. It is the reverse of `getnetbyname` and consults the same system network database (typically `/etc/networks` or an NSS backend). In scalar context it returns just the canonical network name; in list context it returns the full record.

Synopsis

```
my $name = getnetbyaddr $addr, $addrtype;  
my ($name, $aliases, $addrtype, $net) = getnetbyaddr $addr,  
    ↪ $addrtype;
```

What you get back

Scalar context - the canonical name of the network, or `undef` if no entry matches.

List context - a four-element list mirroring the C struct `netent`:

- `$name` - the canonical network name.
- `$aliases` - a space-separated string of alternate names.
- `$addrtype` - the address family of the record, typically `AF_INET` (2).
- `$net` - the network number as an unsigned integer, in host byte order.

On lookup failure in list context, the return is the empty list. If the entry exists but is empty, the return is a single meaningless true value per the shared `get*` family convention.

Arguments

- ADDR - the network number as an **integer in host byte order**, not a packed string. This is the native-integer form you get back from a prior `getnetbyname` call, or that you build with `Socket::inet_aton` followed by `unpack("N", ...)` to convert from network to host order.
- ADDRTYPE - the address family, almost always `AF_INET` from the `Socket` module.

Global state it touches

- The network database iteration cursor shared with `getnetent`, `setnetent`, and `endnetent`. A call to `getnetbyaddr` may reset a cursor left open by `getnetent`.
- `$!` - the C `errno` value on failure, where the platform sets it. Most `/etc/networks` lookups fail silently with `errno` untouched, so treat `undef` as the primary failure signal.

Examples

Look up the loopback network by its integer number:

```
use Socket;
my $name = getnetbyaddr(127 << 24, AF_INET);
print "$name\n";           # loopback
```

List context - the whole record:

```
use Socket;
my ($name, $aliases, $type, $net)
    = getnetbyaddr(127 << 24, AF_INET);
printf "%s (net=%08x)\n", $name, $net;
```

Round-trip from a dotted string, via `Socket`:

```
use Socket;
my $packed = inet_aton("127.0.0.0");      # packed, network order
my $net     = unpack("N", $packed);       # integer, host order
my $name    = getnetbyaddr($net, AF_INET);
```

By-name interface via the `Net::netent` override - accessor methods instead of positional unpacking:

```
use Net::netent;
my $n = getnetbyaddr(127 << 24, AF_INET);
print $n->name, "\n" if defined $n;
```

Forward then reverse - resolve a name, then look the address back up:

```
use Socket;
my $net = (getnetbyname("loopback"))[3];
my $back = getnetbyaddr($net, AF_INET);
print "$back\n";           # loopback
```

Edge cases

- **Integer, not packed string.** ADDR is a plain integer in host byte order. Passing the four-byte packed form that `gethostbyaddr` consumes here will silently look up the wrong number. This is the most common mistake with this function.
- **Classful only, on most systems.** The underlying `/etc/networks` database predates CIDR. Entries are classful network numbers (e.g. 127, 10, 192.168), and the system call is not useful for arbitrary CIDR subnets. For modern subnet reasoning use `Socket` or a CPAN module like `NetAddr::IP`.
- **Empty `/etc/networks`.** Many Linux distributions ship a nearly empty `/etc/networks`; expect most lookups to return `undef` on such a host. This is a configuration fact, not a bug in the call.
- **List-context empty database entry.** Per the shared `get*` family rule, an entry that exists but has no fields returns a single meaningless true value, not the five-element list. In practice no platform produces such an entry for networks; guard with defined and element count if you care.
- **ADDRTYPE other than `AF_INET`.** The C library accepts the argument but has no defined behavior for non-Internet families here. Pass `AF_INET` unless the target system documents otherwise.
- **Thread safety.** The underlying C routine keeps per-process static state for the network cursor. Concurrent calls from multiple threads may interleave results. Where the platform offers `getnetbyaddr_r`, Perl substitutes it automatically.
- **List-operator precedence.** Like the rest of the `get*` family, `getnetbyaddr` parses as a list operator. `getnetbyaddr $n, AF_INET or die` binds the `or die` to the whole argument list, which is usually what you want; add parens when in doubt.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `getnetbyname` - the forward lookup: canonical or alias name → network record
- `getnetent` - iterate every entry in the network database instead of looking up one address
- `setnetent` - rewind the network iterator and control whether the database stays open across lookups
- `endnetent` - close the network database when done iterating
- `gethostbyaddr` - the host-level analogue; takes a packed address, not an integer
- `Net::netent` - by-name wrapper that returns an object with `name` / `aliases` / `addrtype` / `net` accessors

Network info

getnetbyname

Look up a network record by name.

getnetbyname resolves a symbolic network name (as listed in the system's networks database, typically `/etc/networks`) to its numeric network address by calling the C `getnetbyname(3)` routine. The scalar-context form returns the network number; the list-context form exposes the full record: canonical name, aliases, address family, and network number.

Synopsis

```
my $net = getnetbyname $name;
my ($name, $aliases, $addrtype, $net) = getnetbyname $name;
```

What you get back

Scalar context - the network number as a plain integer, or `undef` if the name is not in the networks database. This is the host-byte-order network portion of an IPv4 address, not a packed string. For the class-C network `192.168.0`, the returned value is `12625920` (that is, `0xC0A800`).

List context - a four-element list mirroring the C struct `netent`:

- `$name` - the canonical network name.
- `$aliases` - a space-separated string of alternate names.
- `$addrtype` - the address family, typically `AF_INET` (2).
- `$net` - the network number, same integer as the scalar-context form.

On lookup failure in list context, the return is the empty list.

Global state it touches

- The networks database iteration cursor shared with `getnetent`, `setnetent`, and `endnetent`. A call to `getnetbyname` may reset a cursor left open by `getnetent`.

Examples

Scalar form - look up a named network and print it as dotted-quad:

```
my $net = getnetbyname("loopback");
defined $net or die "no networks entry for loopback";
printf "%d.%d.%d.%d\n",
    ($net >> 24) & 0xFF, ($net >> 16) & 0xFF,
    ($net >> 8) & 0xFF, $net & 0xFF;    # 127.0.0.0
```

List form - every field of the record:

```
my ($name, $aliases, $type, $net) = getnetbyname("loopback");
print "name=$name aliases=$aliases type=$type net=$net\n";
```

Round-trip against `getnetbyaddr` - name to number, then number back to name:

```
use Socket qw(AF_INET);
my $net = getnetbyname("loopback");
my $back = getnetbyaddr($net, AF_INET);
print "$back\n";                # loopback
```

By-name interface via the `Net::netent` override - accessor methods instead of positional unpacking:

```
use Net::netent;
my $n = getnetbyname("loopback");
print $n->name, " ", $n->net, "\n";
```

Guard against a missing entry - a name that is not in the database returns `undef`, not zero:

```
my $net = getnetbyname("no-such-network");
if (defined $net) {
    print "found: $net\n";
} else {
    print "not in /etc/networks\n";
}
```

Edge cases

- **Check for `undef`, not falsehood.** A network number of `0` is a legal (if unusual) record; treating the return value as a boolean will misread that case as failure. Always test with `defined`.
- **The returned number is host-byte-order, not a packed string.** Unlike `gethostbyname`, which returns a four-byte packed address, `getnetbyname` returns a plain integer. Do not pass it to `Socket::inet_ntoa` - that function expects a packed string and will produce garbage on an integer.
- **Aliases are one scalar, not a list.** `$aliases` in the list form is a single space-separated string. Split on whitespace if you need a list.
- **Networks are typically class-boundary numbers.** The entries in `/etc/networks` are traditionally stored as the network portion only
 - 127 for loopback, 10 for the private class-A, not full four-octet addresses. On modern CIDR-routed networks the utility of this database is limited; most deployments keep only a handful of legacy entries.
- **No DNS fallback.** Unlike `gethostbyname`, this call consults only the local networks database. There is no equivalent of DNS for network names.
- **Empty list vs. `undef`.** Scalar context returns `undef` on failure; list context returns the empty list. The two are not interchangeable - code written for one context will misbehave in the other.
- **Thread safety.** The underlying C routine keeps per-process static state for the networks cursor. Concurrent calls from multiple threads may interleave results.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `getnetbyaddr` - the reverse lookup: network number → network record
- `getnetent` - iterate every entry in the networks database instead of looking up one name
- `setnetent` - rewind the networks iterator and control whether the file stays open across lookups
- `endnetent` - close the networks database after an iteration
- `gethostbyname` - the host-level counterpart, for resolving a hostname to an IP address
- `Net::netent` - by-name object wrapper returning accessor methods instead of a positional list

Network info

`getnetent`

Read the next entry from the networks database.

`getnetent` wraps the C library routine of the same name. It walks the system networks database (usually `/etc/networks`, sometimes augmented by NIS or other resolver backends) one record at a time, returning the next entry on each call. Pair it with `setnetent` to rewind or control the connection, and `endnetent` to close it.

Synopsis

```
my ($name, $aliases, $addrtype, $net) = getnetent;
my $name                               = getnetent;    # scalar
↪ context
```

What you get back

In list context, a four-element list:

- `$name` - canonical network name (e.g. "loopback").
- `$aliases` - space-separated string of alternative names.
- `$addrtype` - address family, typically `AF_INET` (2).
- `$net` - network number as an integer in host byte order. Convert to dotted form with `Socket::inet_ntoa(pack("N", $net))`.

In scalar context, just `$name`.

When the iterator is exhausted (no more records), list context returns the empty list and scalar context returns `undef`. A missing entry during a direct lookup returns a single

meaningless true value per perlfunc convention, but since `getnetent` takes no key, the only terminal condition is “end of database.”

Global state it touches

`getnetent` reads from a **process-wide iterator** maintained by the C library. Every call advances that iterator; there is no per-scope state, and concurrent iteration from two places in the same process will interleave records. `setnetent` rewinds it and optionally asks the library to keep the backing connection open for the duration of iteration; `endnetent` closes it.

Examples

Walk the entire networks database:

```
setnetent(1);
while (my ($name, $aliases, $addrtype, $net) = getnetent) {
    printf "%-20s %s\n", $name, join(" ", split / /, $aliases);
}
endnetent();
```

Scalar context gives just the name - useful for a quick enumeration:

```
setnetent(1);
while (defined(my $name = getnetent)) {
    print "$name\n";
}
endnetent();
```

Decode the network number into dotted form:

```
use Socket;
setnetent(1);
while (my ($name, undef, undef, $net) = getnetent) {
    printf "%-20s %s\n", $name, inet_ntoa(pack "N", $net);
}
endnetent();
```

Object-oriented access through `Net::netent` - the by-name interface from the standard library replaces the positional return list with a blessed object:

```
use Net::netent;
while (my $n = getnetent()) {
    printf "%s => %d\n", $n->name, $n->net;
}
```

Edge cases

- **Iterator exhaustion:** list context returns `()`, scalar context returns `undef`. The loop forms above terminate naturally.

- **No `setnetent` before the loop:** the first call implicitly opens the database. Calling `setnetent(1)` is still recommended - without the “stay open” flag, some C libraries reopen the file on every record and performance collapses.
- **Multiple concurrent walks:** the iterator is process-global. Two loops over `getnetent` in the same process will split the records between them. Collect into an array first if both need the full set.
- **Unsupported on some platforms:** Windows has no `/etc/networks` equivalent and the underlying C routine is absent. Attempting to call `getnetent` there raises `The getnetent function is unimplemented`. On Linux, `glibc` provides the function but the database is often empty - most systems rely on DNS instead, which this interface does not consult.
- **Thread safety:** the C library’s iterator is not thread-safe. Perl may transparently substitute `getnetent_r` on systems that provide it, but do not assume this is universal. In a threaded program, serialise access through a single thread.
- **Tainting:** when tainting is enabled, all four return fields are tainted - they come from a file (or a remote resolver) outside the program’s control.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `setnetent` - rewind the iterator and optionally keep the backing connection open across calls
- `endnetent` - close the iterator when you’re done
- `getnetbyname` - look up one network by name instead of walking the whole database
- `getnetbyaddr` - look up one network by its address
- `gethostent` - the analogous walk over the hosts database, with the same `set/end` companion pair
- `Net::netent` - by-name object interface that replaces the positional return list

Sockets

`getpeername`

Return the address of the remote end of a connected socket.

`getpeername` asks the kernel who is on the other side of `SOCKET` and returns the answer as a packed `sockaddr` byte string - the same shape `connect` took and the same shape `accept` returned. You then hand that byte string to an unpacker from `Socket` (or `Socket::unpack_sockaddr_in`, `Socket::unpack_sockaddr_in6`, `Socket::unpack_sockaddr_un`) to pull out the port, address, or path.

This is the companion to `getsockname`, which returns the **local** end of the same socket.

Synopsis

```
use Socket;
my $peer = getpeername SOCKET;
my ($port, $iaddr) = unpack_sockaddr_in($peer);
```

What you get back

A packed sockaddr byte string on success, `undef` on failure (with `$!` set - typically `ENOTCONN` when the socket has no peer, or `EBADF` when `SOCKET` is not an open file descriptor).

The byte string is opaque. Its layout depends on the socket's address family; always decode it with an unpacker that matches the family you expect:

- `AF_INET` → `Socket::unpack_sockaddr_in` returns `($port, $iaddr)` where `$iaddr` is the 4-byte packed IPv4 address.
- `AF_INET6` → `Socket::unpack_sockaddr_in6` returns `($port, $iaddr, $scope_id, $flowinfo)`.
- `AF_UNIX` → `Socket::unpack_sockaddr_un` returns the peer's filesystem path.

If you do not know the family at the call site, ask `Socket::sockaddr_family` first:

```
my $peer = getpeername $sock;
my $family = sockaddr_family($peer);
```

Global state it touches

`getpeername` reads the file-descriptor state of `SOCKET`. It sets `$!` on failure and does not otherwise touch interpreter globals. It does not interact with `$_,` the selected filehandle, or output separators.

Examples

Look up the remote IPv4 peer of a client-side TCP socket and render it in dotted-quad form:

```
use Socket;
my $peer = getpeername($sock)
    or die "getpeername: $!";
my ($port, $iaddr) = unpack_sockaddr_in($peer);
printf "peer is %s:%d\n", inet_ntoa($iaddr), $port;
```

Server side: every socket returned by `accept` is already connected, and `accept` itself hands back the peer address. `getpeername` is useful later, when that packed address has not been kept around:

```
while (my $client = accept(my $conn, $listen)) {
    handle($conn);
}
```

(continues on next page)

(continued from previous page)

```

sub handle {
    my ($conn) = @_;
    my $peer = getpeername($conn) or return;
    my ($port, $iaddr) = unpack_sockaddr_in($peer);
    warn "request from ", inet_ntoa($iaddr), ":$port\n";
    # ...
}

```

Resolve the peer back to a hostname - note that this issues a blocking DNS lookup:

```

use Socket;
my $peer = getpeername($sock);
my ($port, $iaddr) = unpack_sockaddr_in($peer);
my $name = gethostbyaddr($iaddr, AF_INET);
print "connected to ", $name // inet_ntoa($iaddr), "\n";

```

Mixed IPv4 / IPv6 code - decode generically via the address family:

```

use Socket qw(sockaddr_family unpack_sockaddr_in unpack_sockaddr_in6
              AF_INET AF_INET6 inet_ntop);

my $peer = getpeername($sock) or die "getpeername: $!";
my $fam = sockaddr_family($peer);

if ($fam == AF_INET) {
    my ($port, $iaddr) = unpack_sockaddr_in($peer);
    printf "v4 %s:%d\n", inet_ntop(AF_INET, $iaddr), $port;
}
elsif ($fam == AF_INET6) {
    my ($port, $iaddr) = unpack_sockaddr_in6($peer);
    printf "v6 [%s]:%d\n", inet_ntop(AF_INET6, $iaddr), $port;
}

```

Probing whether a socket is connected at all - the ENOTCONN path is the well-defined way to ask:

```

if (defined getpeername($sock)) {
    # connected
} else {
    # $! == ENOTCONN for an unconnected socket
}

```

Edge cases

- **Unconnected socket:** a socket that has been `socket`'d but not yet `connect`'d or `accept`'d has no peer. `getpeername` returns `undef` and sets `$!` to `ENOTCONN`.
- **Non-socket filehandle:** passing a plain file or pipe yields `undef` with `$!` set to `ENOTSOCK`.

- **Closed or invalid filehandle:** returns `undef` with `$!` set to `EBADF`. Under use `warnings` a `getpeername()` on closed socket warning is emitted.
- **Datagram sockets:** `getpeername` only returns an address when the `SOCK_DGRAM` socket has been `connect`'d to a default peer; otherwise it fails with `ENOTCONN` even though the socket is usable via `recv` / `send`.
- **AF_UNIX with an unbound peer:** a Unix-domain socket whose peer never called `bind` returns a packed `sockaddr_un` with an empty path. `Socket::unpack_sockaddr_un` gives back the empty string.
- **After shutdown:** `getpeername` still succeeds as long as the descriptor is open; shutdown tears down data flow, not the kernel's record of the peer.
- **SOCKET is a parsed filehandle token:** like other socket built-ins, the argument is parsed as a filehandle, not evaluated as a general expression. `getpeername $handles[0]` is a syntax error; use a scalar holding the handle, or `*{...}` dereferencing.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `getsockname` - mirror call that returns the **local** end of the same socket; pairs with `getpeername` when logging both ends of a connection
- `accept` - already hands you the peer address for a freshly accepted server-side socket; use `getpeername` later when that address was not retained
- `connect` - the call that establishes the peer in the first place; the address you pass to `connect` is what `getpeername` will hand back afterwards
- `Socket` - provides the unpackers (`unpack_sockaddr_in`, `unpack_sockaddr_in6`, `unpack_sockaddr_un`) and `sockaddr_family` needed to interpret the returned byte string
- `gethostbyaddr` - turns the packed IPv4 address from the unpacker into a host-name
- `$!` - carries the reason on failure (`ENOTCONN`, `ENOTSOCK`, `EBADF`)

Processes

getpgrp

Return the POSIX process-group ID that a process belongs to.

Every Unix process is a member of exactly one process group, which in turn belongs to a session. Process groups are how the kernel addresses related processes as a unit - most visibly, they are the unit that receives a signal from the controlling terminal (`SIGINT` on `Ctrl-C`, `SIGTSTP` on `Ctrl-Z`). `getpgrp` asks the kernel for the group ID of a given process; with no argument or a PID of `0` it answers for the current process.

Synopsis

```
getpgrp
getpgrp PID
getpgrp 0
```

What you get back

A non-negative integer - the process-group ID in scalar context. Process-group IDs share the PID numbering space, and the group leader's PID equals the group ID, so the value returned by `getpgrp 0` is the PID of the current process's group leader.

On a system that does not implement `getpgrp(2)` at all, the call raises a fatal exception. That is vanishingly rare on modern Unix; Linux, the BSDs, macOS, and Solaris all support it.

Global state it touches

None. `getpgrp` reads kernel state for the named process and sets `$!` only on failure (which, on a POSIX system, means “no such process” for a non-zero PID that does not exist).

Examples

Get the current process's own group:

```
my $pgid = getpgrp;
print "my group is $pgid\n";
```

PID of `0` means “myself” and is the portable form - some very old systems do not accept any other value:

```
my $pgid = getpgrp 0;           # same as getpgrp with no args
```

Detect whether the current process is its own group leader - the usual test a daemon runs before calling `setpgrp` or POSIX: `setsid` to detach from its controlling terminal:

```
if (getpgrp(0) == $$) {
    print "already a group leader\n";
}
else {
    print "group leader is ", getpgrp(0), "\n";
}
```

Look up another process's group, e.g. a child just returned by `fork`:

```
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    # child
    exec @cmd;
```

(continues on next page)

(continued from previous page)

```
}  
print "child $pid is in group ", getpgrp($pid), "\n";
```

Send a signal to every member of a process group. `kill` with a negative PID targets the group, and `getpgrp` is how you discover the group ID to negate:

```
my $grp = getpgrp($child);  
kill 'TERM', -$grp;                # SIGTERM to the whole group
```

Edge cases

- **Omitted argument equals PID of 0:** `getpgrp` and `getpgrp 0` both report the current process's group. Prefer the explicit `0` form when portability to historical systems matters.
- **Non-existent PID:** returns `undef` and sets `$!` to "No such process" (ESRCH). Check `$!`, not the return value's truthiness - a legitimate group ID of `0` is falsy in Perl but is not an error condition (it is, on Linux, the idle/swapper group).
- **Not-permitted lookups:** on systems that restrict cross-session lookups, querying a PID in a different session can fail with `$!` set to "Operation not permitted" (EPERM). This is unusual on Linux, where any process may query any PID's group.
- **getpgrp vs. getpgid PID:** the BSD-style one-argument form was not portable to all Unixes historically; POSIX standardised both. On the platforms pperl runs on (Linux, all cranelift targets), both forms work.
- **Signed vs. unsigned:** the kernel's PID type is signed, so the returned value fits comfortably in a Perl integer. A return of `-1` is not used for success on any platform pperl targets; treat `-1` as the error sentinel only if you are porting code from a C codebase that checked it that way.
- **Not a shell alias for job control:** the shell's jobs and `%` notation operate on job tables the shell keeps, not on kernel process groups directly. `getpgrp` tells you the kernel truth, which can drift from a subshell's idea of the job tree.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- [setpgrp](#) - put a process into a new or existing group; the write counterpart to `getpgrp`
- [getppid](#) - parent process ID; often wanted alongside the group ID when walking process relationships
- [getpriority](#) - read scheduling priority, which also accepts a process-group selector (`PRIO_PGRP`)
- [fork](#) - the child inherits its parent's process group, so `getpgrp` on a fresh child reports the parent's group until a [setpgrp](#) call changes it

- `kill` - kill `SIGNAL`, `-$pgid` signals every member of the group; `getpgrp` is how you find `$pgid`
- `$$` - current PID; equal to `getpgrp(0)` when the process is its own group leader

Processes

getppid

Return the process ID of the current process's parent.

`getppid` is a thin wrapper over the POSIX `getppid(2)` system call. It takes no arguments and returns an integer - the PID of whatever process forked (or `execed` into) the running program. Combined with `$$` (the current process's own PID), it is the standard way for a child to identify the process that spawned it.

Synopsis

```
my $parent = getppid;
```

What you get back

A positive integer - the PID of the parent process. On Linux this is 1 (`init` / `systemd`) once the original parent has exited and the child has been reparented; there is no special “orphaned” return value, the reparenting is transparent. Never returns `undef` or `0` on a supported platform; the underlying syscall is specified to always succeed.

Global state it touches

None. `getppid` reads the kernel's process table for the current process; it does not touch any Perl special variable. Note that `$$` holds *this* process's PID, not the parent's - fetch the parent with `getppid` explicitly.

Examples

Identify the parent that launched this script:

```
printf "my pid is %d, my parent is %d\n", $$, getppid;
```

Detect orphaning - a common idiom for daemons that want to exit when their controlling process goes away:

```
while (1) {
    last if getppid == 1;          # parent died, we've been reparented
    to init
    do_work();
    sleep 1;
}
```

Child process reports back to its parent after fork:

```
my $pid = fork // die "fork failed: $!";
if ($pid == 0) {
    warn "child $$ spawned by $>, parent is " . getppid . "\n";
    exit 0;
}
waitpid $pid, 0;
```

Compare before and after fork to confirm the relationship:

```
my $before = $$;
my $pid = fork // die $!;
if ($pid == 0) {
    die "parent mismatch" unless getppid == $before;
    exit;
}
waitpid $pid, 0;
```

Edge cases

- **No arguments, no parentheses needed:** `getppid` is a niladic built-in. Write `getppid` or `getppid()`; both parse identically. Do not try to pass a PID argument - unlike `getpgrp`, there is no “parent of another process” form.
- **Orphan reparenting is invisible:** if the original parent exits, the kernel reparents the process (to `init/systemd` on Linux, or to the nearest ancestor with `PR_SET_CHILD_SUBREAPER` set). `getppid` silently starts returning the new parent’s PID. Code that caches the return value at startup and compares later is the standard way to detect this.
- **Race at startup:** calling `getppid` extremely early, before the kernel has finished setting up the process, is not a concern in user Perl code - by the time the interpreter is running, the parent PID is stable.
- **PID wraparound:** PIDs are recycled. A stored parent PID can, over long-running time, refer to a *different* process than the original parent even without reparenting. Use `getppid == 1` (or the equivalent subreaper PID), not equality against a saved value, when the intent is “has my parent died?”.
- **Threads:** since Perl 5.16 the Linux LinuxThreads workaround has been removed. `getppid` now always returns what the kernel reports for the current thread group.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `$$` - the current process’s own PID; pair with `getppid` to log both ends of the parent/child relationship
- `fork` - spawn a child process; the child’s `getppid` returns the forking process’s PID
- `waitpid` - reap a specific child by PID; the mirror operation from the parent side

- `wait` - reap any child; same parent-side operation, unspecified PID
- `getpgrp` - process *group* ID rather than parent PID; related but distinct kernel concept
- `kill` - send a signal to a PID, including one obtained from `getppid` (e.g. notifying the parent)

Processes

getpriority

Return the current scheduling nice value of a process, a process group, or a user.

`getpriority` is a thin wrapper around the `getpriority(2)` system call. Nice values run from `-20` (most favourable to the process) to `19` (least favourable); a freshly-spawned process inherits its parent's value, typically `0`. Only the superuser can *lower* a nice value; any user may raise their own.

Synopsis

```
getpriority WHICH, WHO
```

WHICH selects what WHO means:

- `PRIO_PROCESS` - WHO is a process ID (`0` means the current process).
- `PRIO_PGRP` - WHO is a process-group ID (`0` means the current group).
- `PRIO_USER` - WHO is a numeric user ID (`0` means the real UID of the calling process).

The three constants come from `POSIX`; they are not built-ins and are not exported by default.

```
use POSIX qw(PRIO_PROCESS PRIO_PGRP PRIO_USER);
```

What you get back

The current nice value as an integer in the range `-20 .. 19`.

There is no distinguished error return: `-1` is a perfectly valid nice value. To detect failure, clear `$!` before the call and inspect it afterwards:

```
$! = 0;
my $nice = getpriority PRIO_PROCESS, $pid;
die "getpriority: $!" if $!;
```

On a kernel that implements `getpriority(2)` but returns an error (typically `ESRCH` for a non-existent target or `EINVAL` for a bogus WHICH), the function returns `-1` with `$!` set. On a platform that does not implement the call at all, `getpriority` raises a fatal exception; catch it with `eval` if the script must run on such a host.

Global state it touches

- `$!` - set when the underlying `getpriority(2)` call fails. Must be cleared before the call to distinguish a legitimate `-1` result from an error.

Examples

Read the current process's own nice value:

```
use POSIX qw(PRIO_PROCESS);
my $nice = getpriority PRIO_PROCESS, 0;
print "nice = $nice\n";           # nice = 0
```

Check another process by PID:

```
use POSIX qw(PRIO_PROCESS);
my $nice = getpriority PRIO_PROCESS, $pid;
```

Read the priority of the calling process's process group:

```
use POSIX qw(PRIO_PGRP);
my $nice = getpriority PRIO_PGRP, 0;
```

Read the priority cap associated with a user ID. The value returned is the *lowest* (most favourable) nice value held by any of that user's processes:

```
use POSIX qw(PRIO_USER);
my $nice = getpriority PRIO_USER, $uid;
```

Safe read that distinguishes `-1-as-value` from `-1-as-error`:

```
use POSIX qw(PRIO_PROCESS);
$! = 0;
my $nice = getpriority PRIO_PROCESS, $pid;
if ($!) {
    warn "cannot read priority of $pid: $!";
} else {
    print "pid $pid: nice=$nice\n";
}
```

Edge cases

- **-1 is a valid return**, not an error indicator. Always clear `$!` before the call and test it after.
- **WHO == 0** is the “self” shorthand for each WHICH: current PID, current PGID, or real UID. Passing your own PID / PGID / UID explicitly gives the same result.
- **PRIO_USER with a UID that owns multiple processes** returns the numerically lowest nice value across those processes, reflecting the most favourable scheduling that user currently enjoys.
- **Missing target**. A PID, PGID, or UID with no matching process yields `-1` and sets `$!` to `ESRCH`.

- **Bad WHICH.** A value outside `PRIO_PROCESS / PRIO_PGRP / PRIO_USER` yields `-1` and sets `$!` to `EINVAL`.
- **No `getpriority(2)` on the host.** The call raises a fatal exception rather than returning a value. Guard with `eval` in portable code.
- **No POSIX import, no constants.** Writing `getpriority 0, 0` works by accident because `PRIO_PROCESS` happens to be `0` on Linux, but it is not portable. Always import and use the named constants.
- **Raising priority needs root.** `getpriority` itself is unprivileged; the matching `setpriority` call fails with `EACCES` when a non-root caller tries to *decrease* the nice value.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `setpriority` - write side of the same syscall pair; same WHICH constants, same caveats about `-1`
- `POSIX` - source of `PRIO_PROCESS`, `PRIO_PGRP`, and `PRIO_USER`; none of them are built-in
- `getpgrp` - process-group ID you would pass as `WHO` under `PRIO_PGRP`
- `fork` - child inherits the parent's nice value; read it back with `getpriority` after the fork if you need to confirm
- `$!` - the only channel for distinguishing a real `-1` priority from a failed call

Network info

getprotobyname

Look up an IP protocol by its textual name and return its database entry.

`getprotobyname` is the Perl wrapper around the system C library's `getprotobyname(3)`. Given a protocol name such as `"tcp"`, `"udp"`, or `"icmp"`, it returns the matching entry from the system's `/etc/protocols` database (or whatever backend `nsswitch.conf` directs to). The typical use is turning a human-readable protocol name into the numeric value that `socket` and `setsockopt` want.

Synopsis

```
getprotobyname NAME
my ($name, $aliases, $proto) = getprotobyname NAME;
my $proto                    = getprotobyname NAME;    # scalar
↪ context
```

What you get back

In list context, a four-element-looking list with three fields:

```
my ($name, $aliases, $proto) = getprotobyname("tcp");
# $name      => canonical name, e.g. "tcp"
# $aliases   => space-separated alias list, e.g. "TCP"
# $proto     => numeric protocol number, e.g. 6
```

In scalar context, just the numeric protocol number (\$proto). That is the common form, because the protocol number is the whole reason the lookup was performed:

```
my $tcp = getprotobyname("tcp");    # 6
```

If the protocol is not found, the return is an empty list in list context and `undef` in scalar context. Always check definedness before using the result - 0 is a valid protocol number ("ip"), so truth-testing alone is wrong.

```
defined(my $tcp = getprotobyname("tcp"))
    or die "tcp protocol not in /etc/protocols";
```

Global state it touches

- Shares the protocol-database iteration cursor with `getprotobynumber`, `getprotoent`, `setprotoent`, and `endprotoent`. A call to `getprotobyname` may reposition or reset that cursor depending on the system C library; do not interleave name lookups with a `getprotoent` iteration loop.
- On some platforms the underlying `getprotobyname(3)` sets `$_` / `errno` on failure. Portable code should not rely on it - treat an undefined return as “not found” and move on.

Examples

Turn a name into a protocol number for `socket`:

```
use Socket;
my $proto = getprotobyname("tcp")
    // die "tcp not configured";
socket(my $sock, PF_INET, SOCK_STREAM, $proto)
    or die "socket: $_";
```

List form, to see the canonical name and aliases a system uses:

```
my ($name, $aliases, $proto) = getprotobyname("TCP");
print "name=$name aliases=$aliases number=$proto\n";
# name=tcp aliases=TCP number=6
```

Defensive lookup with a fallback for the two protocols every script actually needs:

```
my %num_for = (tcp => 6, udp => 17, icmp => 1);
sub proto_num {
```

(continues on next page)

(continued from previous page)

```
my ($name) = @_;  
return getprotobyname($name) // $num_for{lc $name};  
}
```

Scalar context is required when the return value feeds an expression; list context would flatten into the surrounding list:

```
my @args = (SOCK_STREAM, scalar getprotobyname("tcp"));
```

Edge cases

- **Case sensitivity is system-dependent.** glibc's `getprotobyname(3)` matches the canonical name and every alias case-insensitively; other C libraries may not. If portability matters, pass the lowercase form ("tcp", not "TCP").
- **Protocol number 0 is valid** ("ip"). Use `defined(...)`, not boolean truth, when checking for success.
- **Not the same as a port number.** `getprotobyname("http")` returns `undef` - "http" is a service name, not a protocol. Use `getservbyname` for service lookups.
- **Empty alias string is normal.** Many entries have no aliases; the second return value is then the empty string, not `undef`.
- **Iteration cursor side-effects.** If your program is in the middle of a `getprotoent` loop, calling `getprotobyname` may reset the cursor and skip or repeat entries. Finish the iteration, or save state, before doing name lookups.
- **Thread safety.** The underlying C call is not thread-safe on every platform - it typically uses a per-process static buffer. Perl may use the reentrant `getprotobyname_r` form when available, but do not assume it.
- **Missing `/etc/protocols` entry.** On minimal containers the database may be absent, in which case every call returns `undef`. Build protocol tables into your code when the target environment is not guaranteed to ship a protocol database.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `getprotobynumber` - the inverse lookup; use when you have the numeric protocol and want its name
- `getprotoent` - iterate every entry in the protocol database one record at a time
- `setprotoent` / `endprotoent` - control the iteration cursor shared with this function
- `getservbyname` - the equivalent lookup for services (ports), not protocols; the two are often confused
- `socket` - the usual consumer of the returned protocol number

- `Net::protoent` - object-oriented wrapper that returns a blessed record with named accessors instead of a positional list

Network info

getprotobynumber

Look up a network protocol entry by its assigned protocol number.

`getprotobynumber` wraps the system `getprotobynumber(3)` call. Given a numeric protocol identifier - the same values named by constants like `IPPROTO_TCP` (6), `IPPROTO_UDP` (17), `IPPROTO_ICMP` (1) - it returns the matching entry from the system's protocol database (typically `/etc/protocols`). The common use is turning a protocol number pulled off the wire, out of a routing table, or returned by `getsockopt` into a human-readable name.

Synopsis

```
getprotobynumber NUMBER
```

What you get back

In list context, a three-element list:

```
my ($name, $aliases, $proto) = getprotobynumber($number);
```

- `$name` - canonical protocol name ("tcp", "udp", "icmp").
- `$aliases` - space-separated string of alternative names for the protocol, or the empty string if none.
- `$proto` - the protocol number you passed in, echoed back from the database entry.

In scalar context, you get `$name` alone. If the number is not in the database, the scalar return is `undef` and the list return is empty.

```
my $name = getprotobynumber(6);           # "tcp"
my @rec  = getprotobynumber(6);           # ("tcp", "", 6)
my $none = getprotobynumber(9999);        # undef
```

Global state it touches

Shares the protocol-database cursor with `getprotobynname`, `getprotoent`, `setprotoent`, and `endprotoent`. Calling `getprotobynumber` does not disturb an iteration in progress on most systems, but portable code that interleaves single lookups with `getprotoent` iteration should call `setprotoent` afterwards to reset the cursor.

On failure, `$!` is set to the underlying `errno`, though most implementations simply return “not found” without setting `errno`.

Examples

Resolve the canonical name of a protocol number:

```
my $name = getprotobynumber(6);
print "$name\n";                      # "tcp"
```

Full record with aliases:

```
my ($name, $aliases, $proto) = getprotobynumber(1);
print "$name ($proto): $aliases\n";    # e.g. "icmp (1): ICMP"
```

Reverse-lookup a protocol pulled out of a packet header:

```
sub protocol_label {
    my ($num) = @_;
    my $name = getprotobynumber($num);
    return defined $name ? $name : "proto-$num";
}
```

Round-trip against `getprotobyname` - useful as a sanity check that your system's protocol database knows the name you expect:

```
my $num = getprotobyname("tcp");      # 6
my $back = getprotobynumber($num);    # "tcp"
```

Pretty-print a `getsockopt` level. The `level` argument to `getsockopt` is a protocol number; `getprotobynumber` turns it back into a label for logging:

```
my ($level) = unpack("I", $packed_level);
printf "socket option at level %s\n", getprotobynumber($level) //
↳ $level;
```

Edge cases

- **List-operator precedence trap.** Even though `getprotobynumber` takes a single argument, it parses with list-operator precedence. The function name swallows everything to its right as one big list expression:

```
getprotobynumber $number eq 'icmp'    # WRONG - equivalent to:
getprotobynumber($number eq 'icmp')   # looks up protocol 0 or 1
getprotobynumber($number) eq 'icmp'   # what you meant
```

Always parenthesise the argument when comparing the result, or assign the lookup to a variable first.

- **Unknown number:** scalar return is `undef`; list return is the empty list. Guard with `defined`:

```
defined(my $name = getprotobynumber($n))
    or die "unknown protocol number $n";
```

- **Non-integer argument:** converted to an integer via the usual numeric coercion before the lookup. A string like "tcp" coerces to 0 and you get whatever entry (if any) is registered for protocol 0, which is usually "ip". For name-based lookup, use [getprotobyname](#).
- **Negative or out-of-range numbers:** no entry, returns empty / [undef](#). No warning is raised.
- **The `Net::protoent` object interface.** If you import `Net::protoent`, `getprotobynumber` is overridden to return a blessed object with named accessors (`$p->name`, `$p->aliases`, `$p->proto`). Handy when you do not want to remember positional field order.
- **Database lookup, not protocol stack introspection.** This function reads `/etc/protocols` (or the equivalent NSS source). It tells you what name the system assigns to a number; it does not tell you whether the kernel actually supports that protocol or whether a socket of that type can be opened.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- [getprotobyname](#) - the inverse: look up a protocol entry by name when you have a label like "tcp" and need the number
- [getprotoent](#) - iterate every entry in the protocol database, when you want to list or index them all
- [setprotoent](#) - reopen / rewind the protocol-database cursor; pair with [endprotoent](#) around an iteration
- [getservbyport](#) - the service-table analogue, turning a port number into a service name
- [getsockopt](#) - returns a protocol number as its `level` argument; feed it to `getprotobynumber` for readable logs
- [Socket](#) - exports the `IPPROTO_*` constants whose numeric values are what you pass in

Network info

[getprotoent](#)

Fetch the next entry from the protocols database.

`getprotoent` walks the system protocols database - on traditional Unix-likes that's `/etc/protocols`, on modern systems whatever the C library and NSS stack map to it - returning one record per call until the database is exhausted. It is the iterating leg of the [setprotoent](#) / `getprotoent` / [endprotoent](#) triad. Reach for [getprotobyname](#) or [getprotobynumber](#) when you want a single targeted lookup; `getprotoent` is for enumerating every protocol known to the system.

The iteration cursor is shared process-wide and owned by the C library. `setprotoent` rewinds it and optionally pins the backing descriptor open; `endprotoent` closes it. Between those two calls `getprotoent` yields each record in order, then returns the empty list (or `undef` in scalar context) once there is nothing left.

Synopsis

```
my ($name, $aliases, $proto) = getprotoent;
my $name = getprotoent;
```

What you get back

List context - a three-element list mirroring the C struct `protoent`:

- `$name` - the canonical protocol name, e.g. "tcp", "udp", "icmp".
- `$aliases` - a space-separated string of alternate names for the same protocol. Often empty.
- `$proto` - the numeric protocol number as used in IP headers and accepted by `socket` / `getsockopt`.

At end-of-database the return is the empty list. The upstream `perlfunc` note about “a single meaningless true value” for missing entries applies to the by-name and by-number lookups, not to `getprotoent` - running off the end of the iteration is a clean empty list.

Scalar context - the canonical `$name` only, or `undef` at end-of-database. Use list context whenever you need the number or the aliases; you cannot reconstruct them from the name after the fact.

Split the aliases into a list with a plain `split`:

```
my @alts = split ' ', $aliases;
```

Global state it touches

- The **protocols database cursor**, a per-process C-library-owned resource shared with `setprotoent`, `endprotoent`, `getprotobyname`, and `getprotobynumber`. A by-name or by-number lookup may reset the cursor mid-iteration on some platforms - see *Edge cases*.
- No Perl-level special variables are read or written. Unlike `gethostent`, the protocols routines have no `h_errno` equivalent, so `$?` is not touched. End of database is not a failure and does not set `$!`.

Examples

Enumerate every protocol the system knows about:

```
setprotoent(1);
while (my ($name, $aliases, $proto) = getprotoent) {
    print "$proto\t$name\t$aliases\n";
```

(continues on next page)

(continued from previous page)

```
}  
endprotoent;
```

Scalar context - list just the canonical names:

```
setprotoent(0);  
while (defined(my $name = getprotoent)) {  
    print "$name\n";  
}  
endprotoent;
```

Build a name-to-number map in one pass so later lookups don't hit the database at all:

```
my %proto_num;  
setprotoent(1);  
while (my ($name, $aliases, $num) = getprotoent) {  
    $proto_num{$name} = $num;  
    $proto_num{$_}    = $num for split ' ', $aliases;  
}  
endprotoent;  
  
print $proto_num{"tcp"}, "\n";           # 6
```

By-record interface via the `Net::protoent` override - accessor methods instead of positional unpacking:

```
use Net::protoent;  
while (my $p = getprotoent) {  
    printf "%-10s %3d\n", $p->name, $p->proto;  
}
```

Edge cases

- **No implicit rewind.** `getprotoent` does not open or rewind the database on its own. The first call after program start works because the C library lazily opens the file, but once the end is reached further calls keep returning end-of-database until a `setprotoent` rewinds the cursor. Always bracket enumeration with an explicit `setprotoent` and `endprotoent`.
- **End-of-database is the empty list, not `undef`.** In list context the loop condition `while (my @r = getprotoent)` works because the empty list is false. In scalar context use `while (defined(my $name = getprotoent))` - a protocol name of `"0"` is theoretically valid and would falsely terminate a truthiness-based loop.
- **Aliases are one scalar, not a list.** `$aliases` is a single space-separated string; split on whitespace if you want a list. An empty aliases field is the common case, not an error.
- **Interleaving with by-name / by-number lookups.** A call to `getprotobyname` or `getprotobynumber` during a walk can reset the shared cursor on some platforms. Finish the iteration first, or restart it with `setprotoent` after the lookup.

- **Not reentrant.** The database cursor is a single global resource inside the C library. Two concurrent iterations in the same process - across threads, signal handlers, or nested callbacks - corrupt each other's state. Confine each walk to one thread at a time.
- **Platforms without a protocols database.** On systems with no `/etc/protocols` equivalent, the call is a no-op that returns the empty list immediately rather than raising an error. See `perlport` for the full list of network-database quirks.
- **Tainting.** Records come from a file the kernel reads on your behalf and are not marked tainted. Treat them as untrusted anyway when the file is user-writable.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `setprotoent` - rewind the iterator and optionally pin the backing descriptor open for the full walk
- `endprotoent` - close the iterator and release the database handle once enumeration is done
- `getprotobyname` - targeted lookup by protocol name instead of full enumeration
- `getprotobynumber` - targeted lookup by numeric protocol number
- `getservent` - same iteration pattern for the services database, which references protocol names from this one
- `gethostent` - same idiom for the hosts database

User and group info

getpwent

Return the next entry from the system password database.

`getpwent` walks the password database one record at a time. The first call opens the database (typically `/etc/passwd`, possibly augmented by NSS with LDAP, SSSD, or similar); each subsequent call returns the next entry. When the iteration is exhausted the call returns an empty list in list context or `undef` in scalar context. Pair it with `setpwent` to rewind and `endpwent` to release any resources the database library is holding.

Synopsis

```
my @fields = getpwent;  
my $name   = getpwent;      # scalar context: login name only
```

What you get back

In list context, a 10-element list describing one account:

```
my ( $name, $passwd, $uid, $gid, $quota,
     $comment, $gcos, $dir, $shell, $expire ) = getpwent;
#      0         1         2         3         4
#      5         6         7         8         9
```

- `$name` - login name.
- `$passwd` - the encrypted password field. On systems with shadow passwords this is usually "x" or "*" unless the process is privileged; the real hash lives in `/etc/shadow`. Tainted under `-T`.
- `$uid`, `$gid` - numeric user and primary group ids.
- `$quota` - disk quota on systems that support it, otherwise an empty string. On some systems this slot carries `$change` or `$age` (password-aging data).
- `$comment` - administrative comment on systems that support it, otherwise empty. On some systems this slot carries `$class`.
- `$gcos` - the GECOS field, conventionally the user's real name. User-writable on many systems, therefore tainted under `-T`.
- `$dir` - home directory.
- `$shell` - login shell. Tainted under `-T`.
- `$expire` - account or password expiration on systems that support it, otherwise absent.

In scalar context, only the login name is returned. When the iteration is exhausted the scalar value is `undef` and the list value is empty.

Whether `$quota`, `$comment`, and `$expire` carry meaningful data on the current build depends on how Perl was configured; inspect `Config` values `d_pwquota`, `d_pwage`, `d_pwchange`, `d_pwcomment`, and `d_pwexpire` to find out.

Global state it touches

`getpwent` keeps its cursor in a **process-wide**, library-owned state block inside the C library's `pwd` routines. Consequences:

- Two independent loops over the database in the same process interfere with each other. Each `getpwent` advances the one shared cursor.
- `setpwent` rewinds that cursor to the beginning.
- `endpwent` closes the database and drops any resources (open file descriptors, NSS module state) the library was holding.
- Threads share the cursor too. `getpwent` is not safe to call concurrently from multiple threads; if two threads iterate in parallel, neither sees every record.

`getpwent` does not read or write any Perl special variables.

Examples

Dump every account name and home directory:

```
while ( my @pw = getpwent ) {
    print "$pw[0]\t$pw[7]\n";
}
endpwent;
```

Named-field destructuring reads more clearly than numeric indexing:

```
while ( my ( $name, undef, $uid, $gid, undef, undef, $gcos, $dir ) =
    ↪ getpwent ) {
    next if $uid < 1000;           # skip system accounts
    print "$name ($gcos) uid=$uid home=$dir\n";
}
endpwent;
```

Scalar context yields the login name, useful for enumerating users without unpacking the rest:

```
my @users;
while ( defined( my $name = getpwent ) ) {
    push @users, $name;
}
endpwent;
```

Rewind in the middle of a walk with `setpwent`:

```
my @first_pass = (getpwent)[0];
setpwent;                               # back to the top
while ( my @pw = getpwent ) { ... }
endpwent;
```

Collect all entries into a hash keyed by uid, then release the database:

```
my %by_uid;
while ( my @pw = getpwent ) {
    $by_uid{ $pw[2] } = \@pw;
}
endpwent;
```

Edge cases

- **End of iteration:** returns an empty list in list context and `undef` in scalar context. `while (my @pw = getpwent)` and `while (defined(my $name = getpwent))` are the idiomatic loop forms.
- **Forgetting `endpwent`:** leaks whatever file descriptors or NSS state the library opened until the process exits. Not a correctness bug, but noticeable in long-lived programs and in test harnesses.

- **Nested iteration is broken:** two loops in the same process share the library's single cursor. Collect one walk into a data structure before starting the next.
- **Shadow passwords:** on Linux and Solaris the real password hash is in `/etc/shadow`, readable only by root. Non-privileged processes see "x" or "*" in `$passwd` and must call the shadow-aware routines (outside of core Perl) to get the hash.
- **NSS surprises:** the database is whatever `/etc/nsswitch.conf` says it is. On a host bound to LDAP or SSSD, `getpwent` may return thousands of directory accounts, take a noticeable amount of time, and fail intermittently if the network is flaky. Consider looking accounts up by name with `getpwnam` when you know which one you want.
- **Tainting under -T:** `$passwd`, `$gcos`, and `$shell` come back tainted because users can influence them. Untaint with a deliberate regex match if you need to use them in sensitive operations.
- **Thread safety:** unlike `getpwnam` and `getpwuid`, there is no `getpwent_r` in POSIX that Perl can transparently substitute. Concurrent iteration from multiple threads is unsafe.
- **By-name interface:** `User::pwent` overrides `getpwent` with a version that returns an object exposing fields as methods (`$pw->name`, `$pw->uid`, ...). Use it when the numeric-slot unpacking gets unwieldy.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `getpwnam` - look up one account by login name; the right call when you know which user you want
- `getpwuid` - look up one account by numeric uid
- `setpwent` - rewind the `getpwent` cursor to the first entry
- `endpwent` - close the password database and release library resources; call it when the walk is done
- `getgrent` - the group-database counterpart, same iteration model
- `User::pwent` - by-name interface that returns an object with named accessors instead of a 10-element list

User and group info

getpwnam

Look up a user's `passwd` record by login name.

`getpwnam` wraps the system C library's `getpwnam(3)` call. Pass a login name like "root" or "alice"; you get back the `passwd` entry for that user, or an empty list / `undef` if no such user exists. The return shape depends on context: list context gives every field of the record; scalar context gives a single useful value - for `getpwnam` specifically, the numeric UID.

Synopsis

```
getpwnam NAME
```

What you get back

List context returns a 10-element list mirroring `struct passwd` augmented with the BSD extras:

```
my ($name, $passwd, $uid, $gid, $quota,
    $comment, $gcos, $dir, $shell, $expire) = getpwnam($login);
```

- `$name` - login name (same string you passed in)
- `$passwd` - encrypted password, or "x" / "*" on shadow-password systems. Tainted under -T.
- `$uid` - numeric user ID
- `$gid` - numeric primary group ID
- `$quota` - disk quota, or `$change / $age` on some systems. Empty string when unsupported.
- `$comment` - administrative comment, sometimes `$class`. Empty string when unsupported.
- `$gcos` - GECOS field (usually the real name plus other user info). Tainted under -T because users can edit it.
- `$dir` - home directory
- `$shell` - login shell. Tainted under -T.
- `$expire` - account / password expiration, when supported.

Scalar context returns just `$uid`. For `getpwnam` - a lookup *by name* - scalar context gives you “the other thing,” matching the rule upstream documents for all `getpw*` / `getgr*` / `gethost*` / `getnet*` / `getproto*` / `getserv*` functions.

On lookup failure both contexts return the empty list / `undef`. Always check the return before unpacking.

Global state it touches

- Sets `$!` on system-call failure (rare - “user not found” is not a failure, it’s an empty return).
- Under -T (taint mode), `$passwd`, `$gcos`, and `$shell` come back tainted. See `perlsec` for how to launder them.
- Shares the same underlying iterator state as `getpwent`, `setpwent`, and `endpwent`. A `getpwnam` call does **not** disturb an in-progress `getpwent` loop on glibc, but portable code should not assume that.

Examples

Basic name-to-UID lookup in scalar context:

```
my $uid = getpwnam("root");          # 0
```

Full record in list context, with a defensive check:

```
my @pw = getpwnam($user)
    or die "$user: not in passwd database";
my ($name, undef, $uid, $gid, undef, undef, $gcos, $home, $shell) =
    @pw;
print "$name ($uid) -> $home, shell $shell\n";
```

Assignment pattern from upstream - unpacking directly with or die:

```
my ($login, $pass, $uid, $gid) = getpwnam($user)
    or die "$user not in passwd file";
```

Comparing file ownership against a named user without hand-written field indexing, via the `User::pwent` override:

```
use User::pwent;
use File::stat;
my $is_theirs = (stat($filename)->uid == getpwnam($whoever)->uid);
```

`use User::pwent` replaces `getpwnam` with a version returning a blessed object with `->name`, `->uid`, `->gid`, `->dir`, `->shell` accessors - handy when you want one field and don't want to count commas. The two forms do not mix: inside one scope either use the override or the built-in, not both.

Fallback chain for “who am I running as”:

```
my $login = getlogin() || getpwuid($<) || "unknown";
```

`getpwuid` is the by-UID counterpart; `getlogin` reads the controlling-tty record and can lie on detached processes, so it's the first choice, not the only one.

Edge cases

- **No such user** returns the empty list in list context and `undef` in scalar context. This is not an error - `$!` is not set. Distinguish “user missing” from “lookup failed” by checking `$!` only when you want to report an unexpected failure, not when deciding whether the user exists.
- **\$passwd is rarely useful.** On modern Linux and BSD systems the real hash lives in `/etc/shadow`, readable only by root. You get `"x"` or `"*"` here. Use PAM or the `Authen::PAM` module for authentication rather than comparing against this field.
- **\$quota, \$comment, \$expire portability.** These fields are empty or meaningless on Linux. The upstream POD lists Config values (`d_pwquota`, `d_pwage`, `d_pwchange`, `d_pwcomment`, `d_pwexpire`) you can check if you need to know at runtime whether the platform populates them.

- **Tainted fields under -T.** \$passwd, \$gcos, \$shell are tainted because users can modify them (via chsh, chfn, or direct /etc/passwd edits in privileged contexts). Pass them through a regex capture to launder before using in a system call.
- **Return value is a “single meaningless true value”** in Boolean-ish scalar context *when the entry exists but you wanted list context*. The scalar-context documented return for getpwnam is the UID, which for root is 0 - a false value. `if (getpwnam("root")) { ... }` therefore fails for root. Use defined or list context:

```
if (defined getpwnam("root")) { ... } # correct
if (my @pw = getpwnam("root")) { ... } # also correct
```

- **Name is a string, not a UID.** `getpwnam(0)` stringifies 0 and looks up the literal user named "0", which almost never exists. Use `getpwuid` for numeric lookups.
- **Large directories (LDAP / NIS / SSSD).** `getpwnam` goes through NSS, so the cost scales with however the local resolver answers. A miss against a misconfigured LDAP server can block for seconds. Cache results in a hash if you are looking up many users in a tight loop.
- **Threads.** Upstream notes that `getpwnam_r` is used automatically where the platform provides it; `getpwent` iteration, by contrast, remains per-process. Not a concern under pperl (no ithreads), but worth knowing if you share code with perl5.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `getpwuid` - same record, looked up by numeric UID; pair with `getpwnam` when you need to go UID → name
- `getpwent` - iterate the whole passwd database; use when you need every user, not a specific one
- `endpwent` - close the passwd iterator after `getpwent`
- `getgrnam` - group-database counterpart; needed to resolve the \$gid field from this record back to a group name
- `getlogin` - cheap “who is on this tty” lookup; fall back to `getpwnam/getpwuid` when it returns empty
- `User::pwent` - object wrapper with named accessors; reach for it when you want `->uid` instead of counting list positions

User and group info

getpwuid

Look up a user’s passwd record by numeric UID.

`getpwuid` wraps the system C library’s `getpwuid(3)` call. Pass a numeric user ID like 0 or \$<; you get back the passwd entry for that user, or an empty list / `undef` if no such UID

exists. The return shape depends on context: list context gives every field of the record; scalar context gives a single useful value - for `getpwnid` specifically, the login name.

Synopsis

```
getpwnid UID
```

What you get back

List context returns a 10-element list mirroring `struct passwd` augmented with the BSD extras - the same shape `getpwnam` returns:

```
my ($name, $passwd, $uid, $gid, $quota,
    $comment, $gcos, $dir, $shell, $expire) = getpwnid($uid);
```

- `$name` - login name
- `$passwd` - encrypted password, or "x" / "*" on shadow-password systems. Tainted under -T.
- `$uid` - numeric user ID (same number you passed in)
- `$gid` - numeric primary group ID
- `$quota` - disk quota, or `$change / $age` on some systems. Empty string when unsupported.
- `$comment` - administrative comment, sometimes `$class`. Empty string when unsupported.
- `$gcos` - GECOS field (usually the real name plus other user info). Tainted under -T because users can edit it.
- `$dir` - home directory
- `$shell` - login shell. Tainted under -T.
- `$expire` - account / password expiration, when supported.

Scalar context returns just `$name` (the login name). For `getpwnid` - a lookup *by UID* - scalar context gives you “the other thing,” matching the rule upstream documents for all `getpw*` / `getgr*` / `gethost*` / `getnet*` / `getproto*` / `getserv*` functions.

On lookup failure both contexts return the empty list / `undef`. Always check the return before unpacking.

Global state it touches

- Sets `$!` on system-call failure (rare - “UID not found” is not a failure, it’s an empty return).
- Reads `$<` and `$>` implicitly only when you pass them in, but the common idiom `getpwnid($<)` is what turns the real-UID special variable into a useful record.
- Under -T (taint mode), `$passwd`, `$gcos`, and `$shell` come back tainted. See `perlsec` for how to launder them.

- Shares the same underlying iterator state as `getpwent`, `setpwent`, and `endpwent`. A `getpwuid` call does **not** disturb an in-progress `getpwent` loop on glibc, but portable code should not assume that.

Examples

Basic UID-to-name lookup in scalar context:

```
my $name = getpwuid(0);           # "root"
```

“Who am I running as” - the canonical `getpwuid` idiom, resolving the real UID to a login name:

```
my $me = getpwuid($<);           # e.g. "alice"
```

Pair with `getlogin` for a reliable fallback chain that survives detached processes and su sessions:

```
my $login = getlogin() || getpwuid($<) || "unknown";
```

`getlogin` reads the controlling-tty record and can lie (or return empty) when there is no tty; `getpwuid($<)` answers from the kernel’s recorded UID and always works.

Full record in list context, with a defensive check:

```
my @pw = getpwuid($uid)
    or die "uid $uid: not in passwd database";
my ($name, undef, undef, $gid, undef, undef, $gcos, $home, $shell) = @pw;
print "$name ($uid) -> $home, shell $shell\n";
```

Go UID → group name via the `$gid` field:

```
my $gid      = (getpwuid($<))[3];
my $groupnam = getgrgid($gid);
```

Object-style access via the `User::pwent` override, useful when you want named accessors instead of counting list positions:

```
use User::pwent;
my $pw = getpwuid($<);
printf "%s lives in %s, runs %s\n", $pw->name, $pw->dir, $pw->shell;
```

`use User::pwent` replaces `getpwuid` with a version returning a blessed object with `->name`, `->uid`, `->gid`, `->dir`, `->shell` accessors. The two forms do not mix: inside one scope either use the override or the built-in, not both.

Edge cases

- **No such UID** returns the empty list in list context and `undef` in scalar context. This is not an error - `$!` is not set. Distinguish “UID missing” from “lookup failed” by checking `$!` only when you want to report an unexpected failure, not when deciding whether the UID exists.

- **\$passwd is rarely useful.** On modern Linux and BSD systems the real hash lives in `/etc/shadow`, readable only by root. You get `"x"` or `"*"` here. Use PAM or the `Authen::PAM` module for authentication rather than comparing against this field.
- **\$quota, \$comment, \$expire portability.** These fields are empty or meaningless on Linux. The upstream POD lists Config values (`d_pwquota`, `d_pwage`, `d_pwchange`, `d_pwcomment`, `d_pwexpire`) you can check if you need to know at runtime whether the platform populates them.
- **Tainted fields under -T.** `$passwd`, `$gcos`, `$shell` are tainted because users can modify them (via `chsh`, `chfn`, or direct `/etc/passwd` edits in privileged contexts). Pass them through a regex capture to launder before using in a system call.
- **Scalar-return on UID 0 is never false.** Unlike `getpwnam`, whose scalar return for root is the UID 0 (a false value), `getpwuid`'s scalar return is the *name* "root" - always a true string for existing users. `if (getpwuid($uid)) { ... }` therefore works as you would expect, but reach for defined anyway if you care about the distinction between "missing" and "empty name":

```
if (defined getpwuid($uid))      { ... } # unambiguous
if (my @pw = getpwuid($uid))    { ... } # also correct
```

- **UID is a number, not a name.** `getpwuid("root")` stringifies "root" to 0 (with a warning under `use warnings`) and looks up UID 0. Use `getpwnam` for name lookups.
- **Negative / oversized UIDs.** The system call accepts whatever `uid_t` accepts on the platform (typically 32-bit unsigned). Perl converts negative values via two's-complement on the way in; the result is platform-specific and almost never what you want.
- **Large directories (LDAP / NIS / SSSD).** `getpwuid` goes through NSS, so the cost scales with however the local resolver answers. A miss against a misconfigured LDAP server can block for seconds. Cache results in a hash if you are resolving many UIDs in a tight loop (e.g. annotating `ls -l`-style output).
- **Threads.** Upstream notes that `getpwuid_r` is used automatically where the platform provides it; `getpwent` iteration, by contrast, remains per-process. Not a concern under `ppperl` (no `ithreads`), but worth knowing if you share code with `perl5`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `getpwnam` - same record, looked up by login name; pair with `getpwuid` when you need to go name → UID
- `getpwent` - iterate the whole `passwd` database; use when you need every user, not a specific one
- `endpwent` - close the `passwd` iterator after `getpwent`
- `getgrgid` - group-database counterpart; needed to resolve the `$gid` field from this record back to a group name

- `getlogin` - cheap “who is on this tty” lookup; fall back to `getpwuid($<)` when it returns empty
- `User::pwent` - object wrapper with named accessors; reach for it when you want `->name` instead of counting list positions

Network info

getservbyname

Look up a network service by its textual name and protocol.

`getservbyname` consults the system service database (typically `/etc/services`, or whatever `nsswitch.conf` routes the services source to) and returns the entry whose name or alias matches `NAME` under the transport protocol named by `PROTO`. It is the Perl wrapper around the C library’s `getservbyname(3)` call, and it is how you turn “https” into 443 without hardcoding the number.

Both arguments are strings. `NAME` is the service name or alias (“http”, “ssh”, “imaps”); `PROTO` is the protocol name (“tcp”, “udp”, “sctp”) - **not** a number, and **not** a `SOCK_STREAM`-style constant.

Synopsis

```
my $port = getservbyname NAME, PROTO; # scalar
my ($name, $aliases, $port, $proto) = getservbyname NAME, PROTO; # list
```

What you get back

In **scalar context**, the port number as a plain integer in host byte order - ready to feed to `pack` with “n” or to a `sockaddr_in` constructor. If no entry matches, the return is `undef`.

In **list context**, the four fields of the servent record:

- `$name` - canonical service name (the primary name, not whichever alias you looked up by).
- `$aliases` - space-separated string of alternate names.
- `$port` - port number, host byte order.
- `$proto` - canonical protocol name as returned by the resolver, normally equal to the `PROTO` you passed.

If no entry matches in list context the return is the empty list.

The port is **host byte order**. Network APIs expect network byte order, so pass it through `pack` with the “n” template - or let `Socket` do it via `sockaddr_in`.

Global state it touches

- Holds an open handle to the services database between calls when the database has been opened with `setservent` using a non-zero STAYOPEN. `endservent` closes it.
- Sets `$!` on lookup failures that stem from a system-level error (database file missing, NSS backend unreachable). A plain “no such service” is reported by returning `undef` / the empty list, not by setting `$!`.

Examples

Resolve a well-known service to its port:

```
my $port = getservbyname("https", "tcp");
print $port, "\n";                               # 443
```

Full record in list context:

```
my ($name, $aliases, $port, $proto) = getservbyname("http", "tcp");
print "$name ($aliases) -> $port/$proto\n";
                                     # www (www-http http) -> 80/
↪tcp                                     # (exact aliases depend on /
↪etc/services)
```

Build a `sockaddr_in` for `connect`, converting the host-order port to network order with `pack`:

```
use Socket;
my $port = getservbyname("smtp", "tcp")
    or die "smtp/tcp not in services db";
my $addr = sockaddr_in($port, inet_aton("mail.example.com"));
socket(my $s, AF_INET, SOCK_STREAM, getprotobyname("tcp")) or die $!
↪;
connect($s, $addr) or die "connect: $!";
```

Distinguish “not found” from “wrong protocol” - a service that exists under `tcp` but not `udp` returns `undef` when the protocol does not match:

```
defined(getservbyname("https", "tcp")) or die "expected 443";
defined(getservbyname("https", "udp")) or warn "no https/udp entry";
```

Look up by alias - aliases resolve the same as the canonical name, but the returned `$name` is always the canonical one:

```
my ($name) = getservbyname("www", "tcp");
print $name, "\n";                               # "http" on most systems
```

Edge cases

- **Unknown service:** returns `undef` in scalar context, the empty list in list context. `$!` is **not** set for a plain miss - only for underlying system errors.
- **Unknown protocol:** same as an unknown service. `getservbyname("ssh", "xyzzzy")` returns nothing; no separate error is raised.
- **Case sensitivity:** service and protocol names are compared case-insensitively by the system resolver on every platform we support; "HTTPS" and "https" give the same answer.
- **Port is host byte order:** passing it straight into a wire format is a bug. Use `pack("n", $port)` or `sockaddr_in($port, $addr)` - both handle the conversion.
- **NAME and PROTO are both required.** There is no one-arg form; omitting PROTO is a compile-time error.
- **Precedence trap:** like every named-unary-lookalike in this family, `getservbyname` is a list operator. Parenthesise when combining with operators:

```
getservbyname "http", "tcp" == 80      # WRONG: parses as
                                       # getservbyname("http", (
    ↪ "tcp" == 80))
getservbyname("http", "tcp") == 80    # right
```

- **Threads:** on systems with `getservbyname_r(3)`, Perl uses the reentrant variant transparently. On the handful of platforms that lack it, concurrent lookups from multiple threads can clobber each other's results.
- **By-name interface:** `Net::servent` (core) wraps this call and returns an object with named accessors, at the cost of one allocation per lookup.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `getservbyport` - inverse lookup: port + protocol → service record
- `getservent` - walk every entry in the services database, one record at a time
- `setservent` - rewind (or persistently open) the services database before a walk
- `endservent` - close the services database opened by `setservent`
- `getprotobyname` - companion lookup for the PROTO argument when you also need its numeric protocol number
- `Socket` - `sockaddr_in` and `pack_sockaddr_in` consume the host-order port returned here

Network info

getservbyport

Look up a network service by its numeric port and protocol.

getservbyport consults the system services database (/etc/services on Linux, plus whatever NSS sources are configured) and returns the record for the service that listens on PORT under protocol PROTO. PORT is an integer **in network byte order** - the value you would pass to `bind` or receive from `sockaddr_in`, not the decimal shown in /etc/services. PROTO is a lowercase string such as "tcp" or "udp". The call wraps the C library's `getservbyport(3)`.

Synopsis

```
getservbyport PORT, PROTO      # list context: full record
scalar getservbyport PORT, PROTO # scalar context: service name
```

What you get back

In **list context**, a five-element list describing the service:

```
my ($name, $aliases, $port, $proto) = getservbyport($port_n, "tcp");
```

Index	Name	Meaning
0	\$name	Canonical service name, e.g. "http"
1	\$aliases	Space-separated alias list, e.g. "www www-http"
2	\$port	Port number in network byte order
3	\$proto	Protocol name, e.g. "tcp"

If the entry does not exist the list is empty, so a list-context assignment leaves every target `undef`. A bare list-context call returns a single meaningless true value on miss - always assign to a list and test the first element, or call in scalar context.

In **scalar context**, you get the canonical service name (the same \$name as above), or `undef` if no service is registered on that port for that protocol:

```
my $name = getservbyport($port_n, "tcp")
or warn "port $port not registered for tcp";
```

Global state it touches

- `$!` is **not** reliably set on a miss - the C `getservbyport(3)` function has no standard `errno` contract for “not found”. Treat an empty list or `undef` return as the only trustworthy failure signal.
- Iteration state held by the services database (shared with `getservent`, `setservent`, and `endservent`) is process-global and not thread-safe.

Byte-order trap

PORT must be network byte order. The port column in `/etc/services` is a human decimal; the value the kernel and the C library use is the big-endian 16-bit encoding. Convert with `pack / unpack` or with the `Socket` helpers:

```
use Socket;

# Host decimal 80 -> network-order short:
my $port_n = pack("n", 80);           # two bytes, big-endian
my @rec     = getservbyport($port_n, "tcp");

# Or going the other way, a port extracted from sockaddr_in is
# already network order:
my ($port_n_from_peer, $iaddr) = sockaddr_in(getpeername($sock));
my $name = getservbyport($port_n_from_peer, "tcp");
```

Passing a host-order integer such as 80 works on little-endian hardware only by accident for low ports whose encoding happens to be non-zero in the low byte - most ports will simply miss. This is the single most common reason a `getservbyport` call “silently returns nothing”.

Examples

Identify the service on port 443 over TCP:

```
use Socket;
my $port_n = pack("n", 443);
my ($name) = getservbyport($port_n, "tcp");
print $name, "\n";           # https
```

Pair with `getpeername` to name an incoming connection’s port:

```
use Socket;
my ($peer_port_n, $peer_addr) = sockaddr_in(getpeername($client));
my $service = getservbyport($peer_port_n, "tcp") // "unknown";
print "client connected from service $service\n";
```

Full record dump - useful when you want the protocol or aliases:

```
use Socket;
my @rec = getservbyport(pack("n", 53), "udp");
if (@rec) {
    my ($name, $aliases, $port_n, $proto) = @rec;
    printf "%s (%s) on %d/%s\n",
        $name, $aliases, unpack("n", pack("n", $port_n)), $proto;
}
```

Scalar context, guard clause style:

```
my $svc = getservbyport(pack("n", $port), "tcp")
    or die "no TCP service registered on port $port";
```

Iterate every registered service (the underlying database is shared with `getservent`):

```
while (my ($n, $a, $p, $proto) = getservent()) {  
    next unless $proto eq "tcp";  
    # ...do something with $n / $p...  
}  
endservent();
```

Edge cases

- **Missing entry:** empty list in list context, `undef` in scalar context. A bare `if (getservbyport(...))` in list context is truthy even for a missing entry because of the “single meaningless true value” legacy return - always assign to a list or force scalar context.
- **Unknown protocol:** an unrecognised PROTO string (e.g. “sctp” on a host whose database does not include it) is treated by the C library as no-match. The call returns empty/undef, not an error.
- **Case of PROTO:** case-sensitive on most Linux libc implementations. Use lowercase “tcp” / “udp”; “TCP” will miss.
- **Host-order port:** passing 80 instead of `pack("n", 80)` is the classic bug - see *Byte-order trap* above.
- **Numeric vs string PORT:** Perl numifies PORT before passing to libc. A two-byte packed string works because `pack` with “n” produces bytes that numify deterministically on the supported platforms; if in doubt, `unpack("n", $packed)` first and pass the integer to keep the intent explicit.
- **Precedence with list operators:** like other named unary / list operators, `getservbyport` binds loosely. Parenthesise when the result feeds another operator:

```
getservbyport $port_n, "tcp" eq "https"      # WRONG  
(getservbyport($port_n, "tcp")) eq "https"  # correct
```

- **Threads:** the underlying lookup shares state with `getservent` / `setservent` / `endservent`. Two threads iterating concurrently will interleave records. Use thread-local wrappers when that matters.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `getservbyname` - inverse lookup, service name to port; reach for it when you know the name and want the number
- `getservent` - iterate every service record; use when you need a full sweep rather than a single lookup
- `setservent` - rewind the services iterator, optional STAYOPEN flag to keep the file open across fork

- `endservent` - close the services database after an iteration
- `getprotobyname` - the protocol-side companion lookup; often used right after `getservbyport` to translate the `$proto` field
- `pack` - build the network-order port value `getservbyport` expects as its first argument
- `Socket` - constants and `sockaddr_in` helpers that produce network-order ports straight from kernel structures

Network info

getservent

Read the next entry from the system services database.

`getservent` walks the `/etc/services` database one record at a time, yielding the same list of fields as `getservbyname` and `getservbyport` but without a lookup key. Each call advances an internal cursor; the cursor is opened on the first call, rewound by `set-servent`, and released by `endservent`. It mirrors the POSIX `getservent(3)` C library routine and inherits every portability quirk that implies.

Synopsis

```
my ($name, $aliases, $port, $proto) = getservent;
my $name = getservent;           # scalar context
```

What you get back

In list context, four fields matching the order documented for every `getserv*` variant:

- `$name` - canonical service name, e.g. "http".
- `$aliases` - space-separated string of alternate names, e.g. "www www-http". Not an array; split it yourself if you need one.
- `$port` - port number as a native (host-byte-order) integer, e.g. 80.
- `$proto` - transport protocol, typically "tcp" or "udp".

In scalar context, just `$name`. When the cursor reaches end of file, `getservent` returns the empty list in list context and `undef` in scalar context - the idiomatic `while (my @e = getservent)` loop relies on this.

To convert the aliases into a list:

```
my @alt = split / /, $aliases;
```

Global state it touches

- **Internal database cursor** for `/etc/services` - shared with `getservbyname`, `getservbyport`, `setservent`, and `endservent`. Any by-name or by-port lookup on some platforms rewinds or closes this cursor as a side effect; iterate in a tight loop or wrap the iteration in `setservent` / `endservent` to be safe.

- `$!` - set when the underlying C call reports an error.
- Not thread-safe. `getservent` keeps per-process, not per-thread state; two threads iterating concurrently will interleave records and neither will see the full list.

Examples

Walk the full services database and print each entry:

```
while (my ($name, $aliases, $port, $proto) = getservent) {  
    print "$name/$proto $port\n";  
}  
endservent;
```

Reset the cursor before iterating so earlier calls elsewhere in the program cannot leave it mid-stream:

```
setservent(0);  
while (my @e = getservent) {  
    # @e is ($name, $aliases, $port, $proto)  
}  
endservent;
```

Collect only TCP services into a hash keyed by port:

```
my %tcp;  
while (my ($name, undef, $port, $proto) = getservent) {  
    $tcp{$port} = $name if $proto eq 'tcp';  
}  
endservent;
```

Use the object-oriented by-name interface from `Net::servent`, which overrides the built-in and returns a blessed object with accessor methods:

```
use Net::servent;  
while (my $s = getservent) {  
    printf "%-20s %5d/%s\n", $s->name, $s->port, $s->proto;  
}  
}
```

Edge cases

- **End of file returns empty / undef, not an error.** A single-scalar false value in list context (`my @e = getservent` giving an empty list) is the terminator. Do not check `$!` to detect end of iteration.
- **Missing `/etc/services`:** on systems without the file, the first call returns the empty list immediately and `$!` is set. Nothing in the call signals the difference between an empty database and a missing one - inspect `$!` before assuming the database was simply empty.
- **Cursor sharing with by-name / by-port lookups:** calling `getservbyname` or `getservbyport` in the middle of a `getservent` loop may reset the cursor on some platforms. Either finish the walk first or re-open with `setservent`.

- **\$aliases is a string, not a list.** `split / /`, `$aliases` gives you the array form. Treating `$aliases` as a list directly silently gives you the length of a one-element list.
- **Threaded programs:** iteration is per-process. Coordinate access with a lock, or prefer `getservbyname` / `getservbyport` for point queries in threaded code.
- **Scalar context returns only the name.** If you need the port, call in list context even when you only care about one field.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `getservbyname` - look up a single service by its textual name instead of iterating
- `getservbyport` - look up a single service by its numeric port
- `setservent` - rewind the shared cursor before iterating so prior calls cannot leave it mid-stream
- `endservent` - release the cursor when the walk is done, freeing the underlying file descriptor
- `getprotoent` - same iterator pattern for `/etc/protocols`, typically paired when resolving a (port, proto) tuple
- `Net:::servent` - object-oriented wrapper whose accessors remove the need to remember field order

Sockets

getsockname

Return the local address of a connected or bound socket.

`getsockname` asks the kernel which address and port `SOCKET` is bound to on this end of the connection. The result is the same packed `sockaddr` structure the kernel hands to `accept` or that you would pass to `bind` - unpack it with the appropriate helper from `Socket` (for example `sockaddr_in` for IPv4, `sockaddr_in6` for IPv6). Useful when a host has multiple interfaces and you need to know which one a client reached you on, or when you asked the kernel for an ephemeral port via `bind` with port 0 and now want to know the port it picked.

Synopsis

```
getsockname SOCKET
```

What you get back

On success, the packed sockaddr for the local end of SOCKET as a byte string whose layout matches the socket's address family. On failure, `undef` with `$!` set (typically `ENOTSOCK` if `SOCKET` is not a socket, or `EBADF` if it is not an open filehandle).

The return value is **opaque bytes**. Do not inspect it directly - hand it to the family-appropriate unpacker:

```
use Socket;
my $packed = getsockname($sock) or die "getsockname: $!";
my ($port, $addr) = sockaddr_in($packed);      # IPv4
```

For address-family-agnostic code, call `sockaddr_family` on the result first and dispatch on the returned `AF_*` constant.

Global state it touches

None. `getsockname` is a pure system-call wrapper - it does not read or write `$_`, `$!` aside, or any other interpreter global.

Examples

Discover the ephemeral port the kernel assigned after `bind(..., 0)`:

```
use Socket;
socket(my $srv, AF_INET, SOCK_STREAM, 0) or die $!;
bind($srv, sockaddr_in(0, INADDR_ANY)) or die $!;
my ($port) = sockaddr_in(getsockname($srv));
print "listening on port $port\n";
```

Identify which local IP a connected client reached you on, on a multi-homed host:

```
use Socket;
my $mysockaddr = getsockname($sock);
my ($port, $myaddr) = sockaddr_in($mysockaddr);
printf "Connect to %s [%s]\n",
    scalar gethostbyaddr($myaddr, AF_INET),
    inet_ntoa($myaddr);
```

IPv6 variant - unpack with `sockaddr_in6`:

```
use Socket;
my ($port, $addr) = sockaddr_in6(getsockname($sock));
printf "local: [%s]:%d\n", inet_ntop(AF_INET6, $addr), $port;
```

Family-agnostic dispatch when the socket could be either IPv4 or IPv6:

```
use Socket qw(sockaddr_family sockaddr_in sockaddr_in6
              inet_ntop AF_INET AF_INET6);
my $name = getsockname($sock) or die "getsockname: $!";
if (sockaddr_family($name) == AF_INET) {
```

(continues on next page)

(continued from previous page)

```

    my ($port, $addr) = sockaddr_in($name);
    printf "%s:%d\n", inet_ntoa($addr), $port;
} else {
    my ($port, $addr) = sockaddr_in6($name);
    printf "[%s]:%d\n", inet_ntop(AF_INET6, $addr), $port;
}

```

Edge cases

- **Unbound socket:** A socket that has been created but never bound or connected typically returns a `sockaddr` with the wildcard address and port 0. It is not an error - the kernel simply reports “no local name assigned yet.”
- **Not a socket:** Passing an open filehandle that is not a socket (a plain file, a pipe) returns `undef` with `$!` set to `ENOTSOCK`.
- **Closed filehandle:** Returns `undef` with `$!` set to `EBADF`. Under `use warnings` a `getsockname()` on closed socket warning is emitted.
- **Unix-domain sockets:** The packed form is an `AF_UNIX` `sockaddr`; unpack with `unpack_sockaddr_un`. The path may be shorter than what you passed to `bind`, and on Linux it may begin with a NUL byte for abstract-namespace sockets.
- **Wildcard address with IPv4:** A socket bound to `INADDR_ANY` reports the wildcard address (0.0.0.0) here, not the specific interface address the peer reached. To learn the peer’s target address you must use `getsockname` *after* the kernel has routed the connection (post-accept), or consult `IP_PKTINFO` via `getsockopt`.
- **Call before bind/connect:** The behaviour is platform-specific but harmless. Linux returns a zeroed `sockaddr` of the socket’s family; some BSDs return `undef` with `EINVAL`. Rely on it only after `bind`, `connect`, or `accept` have completed.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `getpeername` - the mirror call: returns the **remote** end’s `sockaddr` for a connected socket
- `bind` - set the local address; `getsockname` reads back what the kernel actually bound, including ephemeral ports chosen for you
- `accept` - returns the remote `sockaddr` directly, so you rarely need `getsockname` on the accepted socket unless the listener bound to a wildcard and you want the concrete local address
- `socket` - creates the socket whose local name `getsockname` reports
- `Socket` - the module providing the unpackers (`sockaddr_in`, `sockaddr_in6`, `unpack_sockaddr_un`, `sockaddr_family`) you apply to the return value

- `getsockopt` - for per-packet destination-address information (IP_PKTINFO) when a wildcard-bound socket needs to know the concrete local address of an incoming datagram

Sockets

getsockopt

Read one socket option out of the kernel as an opaque packed string.

`getsockopt` asks the kernel for the current value of the option named `OPTNAME` at protocol level `LEVEL` on `SOCKET`. The kernel does not return a typed value - it returns a raw byte buffer whose layout is determined by the `(LEVEL, OPTNAME)` pair. You decode that buffer with `unpack`, usually with the `"i"` or `"I"` template, because most options are plain integers.

Synopsis

```
getsockopt SOCKET, LEVEL, OPTNAME
```

What you get back

A packed string holding the kernel's reply on success, or `undef` on failure with `$!` set to the system error. The string's contents are **opaque bytes** - its length and layout depend entirely on `(LEVEL, OPTNAME)`:

- Boolean and integer options (`SO_KEEPALIVE`, `SO_REUSEADDR`, `TCP_NODELAY`, ...) come back as a packed int. Decode with `unpack` using `"i"` or `"I"`.
- Timeouts (`SO_RCVTIMEO`, `SO_SNDTIMEO`) come back as a packed struct `timeval` - two longs on most platforms.
- Address-bound options (`SO_PEERCRED`, `IP_MULTICAST_IF`, linger structs, ...) have their own layouts; consult `getsockopt(2)` and the relevant protocol man pages for the exact struct.

Always check the return value before decoding - calling `unpack` on `undef` yields an empty result and hides the real error recorded in `$!`.

Global state it touches

- `$!` is set on failure to the `errno` value from the underlying `getsockopt(2)` call. It is not cleared on success, so check the return value, not `$!` directly.

Examples

Check whether Nagle's algorithm is currently disabled (`TCP_NODELAY` set) on a TCP socket:

```
use Socket qw(IPPROTO_TCP TCP_NODELAY);  
  
my $packed = getsockopt($sock, IPPROTO_TCP, TCP_NODELAY)
```

(continues on next page)

(continued from previous page)

```

    or die "getsockopt TCP_NODELAY: $!";
my $nodelay = unpack("I", $packed);
print "Nagle is ", $nodelay ? "off" : "on", "\n";

```

Same idea, but obtaining the TCP protocol number at runtime from the system's protocol database rather than a Socket constant:

```

use Socket qw(TCP_NODELAY);

defined(my $tcp = getprotobyname("tcp"))
    or die "no protocol entry for tcp";
my $packed = getsockopt($sock, $tcp, TCP_NODELAY)
    or die "getsockopt: $!";

```

Read `SO_ERROR` to drain and clear a pending asynchronous error on a non-blocking socket after connect or after select flags it writable:

```

use Socket qw(SOL_SOCKET SO_ERROR);

my $packed = getsockopt($sock, SOL_SOCKET, SO_ERROR)
    or die "getsockopt SO_ERROR: $!";
my $err = unpack("i", $packed);
if ($err) {
    $! = $err;
    die "asynchronous socket error: $!";
}

```

Read the socket type (`SOCK_STREAM`, `SOCK_DGRAM`, ...) of an already opened handle:

```

use Socket qw(SOL_SOCKET SO_TYPE SOCK_STREAM);

my $packed = getsockopt($sock, SOL_SOCKET, SO_TYPE)
    or die "getsockopt SO_TYPE: $!";
my $type = unpack("i", $packed);
print "stream socket\n" if $type == SOCK_STREAM;

```

Read a struct timeval option. `SO_RCVTIMEO` returns two native longs - seconds and microseconds:

```

use Socket qw(SOL_SOCKET SO_RCVTIMEO);

my $packed = getsockopt($sock, SOL_SOCKET, SO_RCVTIMEO)
    or die "getsockopt SO_RCVTIMEO: $!";
my ($sec, $usec) = unpack("l!l!", $packed);
printf "receive timeout: %d.%06d s\n", $sec, $usec;

```

Edge cases

- **SOCKET is not an open socket:** the call fails, returns `undef`, and sets `$!` to `ENOTSOCK` ("Socket operation on non-socket") or `EBADF` ("Bad file descriptor") for a closed handle.
- **Unknown LEVEL / OPTNAME:** the kernel rejects the request with `ENOPROTOOPT` ("Protocol not available"). There is no way to tell this apart from the success case except by checking the return value.
- **Truthy-return idiom:** the common `getsockopt(...)` or `die` works because a successful call returns a non-empty packed string. A zero-length reply would make the idiom misfire, but no kernel option returns a zero-length buffer in practice - every defined option has at least one byte of payload.
- **Integer decode width:** use `"i"` or `"I"` for int-sized options (`SO_KEEPALIVE`, `TCP_NODELAY`, `SO_ERROR`, ...); use `"l!"` or `"L!"` only when the option's documented layout genuinely uses a native long. Getting the width wrong silently returns the wrong number on LP64 platforms.
- **Do not hand-pack the reply:** `getsockopt` returns whatever the kernel wrote, including any padding. Re-pack only on round-trips via `setsockopt`, and then pass the buffer back verbatim rather than reconstructing it.
- **No constants without Socket:** `SOL_SOCKET`, `IPPROTO_TCP`, `TCP_NODELAY`, `SO_ERROR`, and friends are not built-in names. Import them from `Socket` with `use Socket qw(...)` or `use Socket qw(:all)`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `setsockopt` - the write half; sets the option whose value `getsockopt` reads back
- `Socket` - source of the `SOL_*`, `IPPROTO_*`, `SO_*`, `TCP_*`, `IP_*` constants you pass as `LEVEL` and `OPTNAME`
- `unpack` - required to decode the packed byte string the kernel hands back
- `getprotobyname` - run-time way to obtain the `LEVEL` protocol number when you do not want to pull in the `IPPROTO_*` constants from `Socket`
- `getsockname` - sibling introspection call; returns the local address bound to `SOCKET` rather than a single option
- `socket` - creates the socket that `getsockopt` then queries

Filehandles, files, directories

glob

Expand a shell-style filename pattern into the list of matching paths.

`glob` takes a pattern string such as `"*.c"` or `"src/**/*.pm"` and returns the filenames on disk that match, in the same style a Unix shell would produce. In list context it returns

every match at once; in scalar context it acts as a stateful iterator, returning one match per call and `undef` when the list is exhausted. With no argument, `glob` expands the pattern in `$_`.

Synopsis

```
glob EXPR
glob
<PATTERN>
```

What you get back

In list context, a (possibly empty) list of filenames. An empty list means the pattern did not match - it is **not** an error and does not set `$!`.

In scalar context, `glob` is an iterator: each call returns the next match, and after the last match one more call returns `undef`. The iterator state is per-call-site, so two `while (my $f = glob ...)` loops at different source locations do not interfere.

```
my @mp3 = glob "*.mp3";           # list context
while (my $f = glob "*.mp3") { }  # scalar context, iterator
```

Global state it touches

- `$_` - read when `glob` is called with no argument.
- `glob` itself does **not** set `$!` on “no matches”. A non-match is a legitimate empty list, not a failure.
- When a `glob` expression is used as the condition of a `while` or `for` loop, Perl implicitly assigns the result to `$_` and tests for **definedness**, not truth - so a filename of `"0"` still keeps the loop running.

The `<...>` shortcut and how Perl picks it

`<*.c>` is an alternative syntax for `glob("*.c")`. The angle-bracket form shares a reader with `<FH>` ([readline](#)); the parser decides which one you meant at **compile time** based on what sits between the angle brackets:

- Bareword that could be a filehandle (`<STDIN>`, `<FH>`) → [readline](#).
- A simple scalar (`<$fh>`) → [readline](#) on that handle.
- Anything that looks like a pattern (`<*.c>`, `<"*.txt">`, `<${dir}/*>`) → `glob`.

The named function `glob` is unambiguous and more searchable. Prefer it in new code; reach for `<...>` only for the shortest throwaway patterns.

```
my @txt = <"*.txt">;           # same as glob '"*.txt"'
my @txt = glob "*.txt";       # recommended
```


Whitespace splits the pattern

glob splits its argument on whitespace and treats each segment as a separate pattern. The results are concatenated:

```
my @sources = glob "*.c *.h";      # all .c OR .h files
my @all     = glob ".* *";         # every entry in cwd
```

To glob a filename that itself contains spaces, wrap it in an inner pair of quotes so the split sees one segment:

```
my @spacey = glob "'*e f*'" ;      # files with "e f" in the name
my @spacey = glob q("*e f*");      # same, q() for clarity
my @spacey = <"*e f*">;            # same via the shortcut form
```

Interpolating a variable into a spacey pattern:

```
my @spacey = glob "'*${var}e f*'" ;
my @spacey = glob qq("*${var}e f*");
```

If the whitespace-splitting rule gets in the way, use `File::Glob`'s `bsd_glob` directly - it takes the pattern verbatim.

Brace expansion without a wildcard

If the only special characters are non-empty braces, glob does not touch the filesystem at all: it just enumerates the Cartesian product of the alternatives. This is useful for generating string combinations:

```
my @pairs = glob "{apple,tomato,cherry}={green,yellow,red}";
# apple=green, apple=yellow, apple=red,
# tomato=green, tomato=yellow, tomato=red,
# cherry=green, cherry=yellow, cherry=red
```

File::Glob - the engine, and its options

glob is implemented on top of `File::Glob`'s `bsd_glob`. Importing the module's tags tunes behaviour for the whole compilation unit:

- `:nocase` - case-insensitive matching.
- `:case` - force case-sensitive matching (the default on most Unix filesystems).
- `:globally` - replace the core glob operator with `bsd_glob` so every `glob(...)` and `<...>` in the file picks up your options.

```
use File::Glob qw(:globally :nocase);
my @txt = glob "readme*";           # README, readme.txt, Readme.md
```

Call `bsd_glob` directly when you need flags per call, or when you want to bypass the whitespace-splitting rule:


```
use File::Glob qw(bsd_glob GLOB_TILDE GLOB_BRACE);
my @files = bsd_glob("~/docs/*.{md,rst}", GLOB_TILDE | GLOB_BRACE);
```

Examples

List every Perl source file in the current directory:

```
my @perl_files = glob "*.pl *.pm";
```

Iterate MP3 files one at a time - useful when the list might be large and you do not want to materialise it:

```
while (my $song = glob "*.mp3") {
    process($song);           # one filename per iteration
}
```

Recursive patterns are **not** built in; `*` does not cross `/`. Walk the tree with `File::Find` or combine `glob` with an explicit traversal:

```
my @all_c;
for my $dir (glob "src/*") {
    push @all_c, glob "$dir/*.c" if -d $dir;
}
```

Use the iterator form in a while condition - the result is assigned to `$_` and tested for definedness, so filenames like `"0"` do not terminate the loop:

```
while (<*.log>) {
    unlink $_ if -M $_ > 30;      # older than 30 days
}
```

Case-insensitive match for a whole file:

```
use File::Glob qw(:globally :nocase);
my @readmes = glob "readme*";
```

Edge cases

- **No matches:** returns the empty list. No warning, no `$!` set. If you need “pattern had to match”, check scalar `@files` yourself.
- **glob with no argument:** uses `$_`. Bare `glob` inside `while (<>)` is rarely what you want - the pattern will be whatever line of input you just read.
- **while (glob ...) assigns to \$_ and tests definedness:** a filename `"0"` is false under normal boolean rules, but the loop still runs because the condition is defined `$_`. This matches the rule for `readline` in the same position.
- **Wildcards and whitespace:** the split-on-whitespace rule means `glob "a b"` searches for `a` and `b` as two patterns, not for the literal string `"a b"`. Use inner quotes or `bsd_glob` when the filename legitimately contains spaces.

- **Hidden files:** a leading `.` is not matched by `*` - you need an explicit `.*` segment (`glob ".* *"`) to include dotfiles. This matches shell glob semantics, not regex semantics.
- **Tilde expansion:** `glob "~/foo"` works because `File::Glob` expands `~` by default. Writing it through `bsd_glob` without `GLOB_TILDE` in the flags, you get a literal `~`.
- **Interpolation inside `<...>`:** `<$var/*.c>` interpolates `$var` at runtime. Using the angle-bracket form does not escape you from Perl's normal double-quote rules.
- **Angle-bracket ambiguity:** `<FOO>` is a handle read, `<$foo>` is a handle read, `<*.c>` is a glob. `<$foo/*.c>` - one scalar plus pattern characters - is a glob. The resolution is compile-time; runtime changes to `$foo` do not re-decide it.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `opendir` - open a directory handle for finer-grained traversal than pattern matching gives you
- `readdir` - read entries from a directory handle one at a time; pair with `opendir` when you need every entry (including ones `glob` hides, like dotfiles) or file-type info
- `readline` - the other meaning of `<...>`; shares the angle-bracket syntax but reads lines from a filehandle
- `File::Glob` - the underlying implementation; use `bsd_glob` directly for per-call flags and to bypass the whitespace-splitting rule
- `$_` - default pattern source for bare `glob`, and the variable implicitly assigned when `glob` is used as a `while/for` loop condition

Time

gmtime

Convert an epoch time to a broken-down UTC time, either as a 9-element list or as a `ctime(3)`-style string.

`gmtime` is `localtime` expressed in the Greenwich time zone. It takes an integer number of seconds since the epoch - the kind of value `time` returns - and hands back either the individual calendar fields (in list context) or a ready-to-print summary string (in scalar context). No time-zone lookup, no TZ, no DST: the result is always UTC.

Synopsis

```
gmtime EXPR
gmtime
```

`gmtime` without an argument uses the current `time`.

What you get back

In **list context**, a nine-element list drawn straight from the C struct `tm`:

```
#      0      1      2      3      4      5      6      7      8
my ($sec, $min, $hour, $mday, $mon, $year, $yday, $isdst)
    = gmtime(time);
```

Index	Field	Range	Notes
0	\$sec	0..60	60 allows a positive leap second
1	\$min	0..59	
2	\$hour	0..23	
3	\$mday	1..31	1-based
4	\$mon	0..11	0-based - January is 0, December is 11
5	\$year		years since 1900 - add 1900 for AD
6	\$yday	0..6	Sunday is 0
7	\$yday	0..365	0 is January 1st; 365 only in leap years
8	\$isdst	0	always 0 for gmtime - UTC has no DST

In **scalar context**, a single `ctime(3)`-formatted English string:

```
my $s = gmtime;      # "Thu Oct 13 04:54:34 1994"
```

The scalar form is **always English and locale-independent**. For a locale-aware or differently-formatted string, build it from the list form with `POSIX::strftime`.

Global state it touches

None. `gmtime` does not read `$ENV{TZ}`, does not consult the interpreter's locale, and does not depend on or modify any of the Perl special variables. Given the same `EXPR`, it always returns the same result.

With no argument, it reads the current wall-clock time via `time`.

Examples

Current UTC broken down into its fields, with the month-0-index and year-since-1900 traps both handled:

```
my @abbr = qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec);
my ($sec, $min, $hour, $mday, $mon, $year) = gmtime;
printf "%s %d, %d %02d:%02d:%02d UTC\n",
    $abbr[$mon], $mday, $year + 1900, $hour, $min, $sec;
# e.g. "Apr 23, 2026 14:07:09 UTC"
```

The scalar form - a single fixed-format string, useful for logs:

```
my $stamp = gmtime;
print "[$stamp] startup\n";
# "[Thu Apr 23 14:07:09 2026] startup"
```

Format the list with POSIX: `strftime` when you want ISO-8601 or a locale-aware rendering:

```
use POSIX qw(strftime);
my $iso = strftime "%Y-%m-%dT%H:%M:%SZ", gmtime;
# e.g. "2026-04-23T14:07:09Z"
```

Convert a stored epoch (not the current time) - any integer seconds since 1970-01-01 UTC works:

```
my $epoch = 1_700_000_000;
my @t = gmtime($epoch);
printf "%04d-%02d-%02d\n", $t[5] + 1900, $t[4] + 1, $t[3];
# "2023-11-14"
```

`$isdst` is always 0 under `gmtime` - do not use it to probe DST. Call `localtime` if you need that information:

```
my $dst_here = (localtime)[8]; # 0 or 1 depending on local zone
my $dst_utc  = (gmtime)[8];   # always 0
```

Edge cases

- **Bare call uses current time:** `gmtime` with no argument is `gmtime(time)`. Inside interpolation or a list that might swallow the call, parenthesise: `gmtime()`.
- **List vs scalar context is load-bearing.** In list context you get nine numbers; in scalar context you get a string. Forcing scalar on a bare call inside a list:

```
print gmtime, "\n";           # nine digits, no spaces
print scalar gmtime, "\n";    # "Thu Apr 23 14:07:09 2026"
```

- **`$mon` is 0-indexed, `$year` is since-1900.** Two of the most common Perl bugs. For a full calendar year do `$year + 1900`; for a 1-indexed month do `$mon + 1`.
- **`$mday` is 1-indexed, `$yday` is 0-indexed.** Mirrors C struct `tm` exactly - inconsistent, but stable.
- **`$sec == 60` on a positive leap second.** Code that validates with `$sec <= 59` will reject legitimate values.
- **Negative epoch values** represent dates before 1970-01-01 UTC and are accepted on platforms whose `gmtime(3)` accepts them - which Linux does. `gmtime(-1)` returns 1969-12-31 23:59:59 UTC.
- **Fractional seconds are discarded.** `EXPR` is truncated toward zero before the conversion; `gmtime(1.9)` is `gmtime(1)`. For sub-second precision, keep your own fractional part and format it separately.
- **No time-zone abbreviation in the scalar form.** Unlike `localtime`'s `ctime(3)` output on some platforms, `gmtime`'s scalar form never includes a zone name. It is always the five-field "Wdy Mon DD HH:MM:SS YYYY" shape.
- **`$isdst` is always 0,** not undef and not -1. Do not test it for truth under `gmtime`; the answer is meaningless.

Alternatives and companions

- **Going the other way** - from broken-down fields back to an epoch - is the job of the `Time::Local` module. Use `timegm` (UTC) or `timelocal` (local zone) there:

```
use Time::Local qw(timegm);  
my $epoch = timegm($sec, $min, $hour, $mday, $mon, $year);
```

- **Object-oriented access by field name** is offered by `Time::Piece`, which overloads `localtime` and `gmtime` to return objects with `->year`, `->mon`, `->mday` etc. The field conventions there are adjusted - year is already the full year, mon is already 1-indexed - so code written against `Time::Piece` cannot be mixed casually with code written against the built-in list form.
- **Human-readable formatting** belongs to `POSIX::strftime`; feed it the list returned by `gmtime` directly.
- **Comprehensive date arithmetic, time zones, durations** - reach for `DateTime` on CPAN. `gmtime` is the minimum viable primitive; `DateTime` is the full library.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `localtime` - same conversion, but in the local time zone; `$isdst` there is meaningful
- `time` - produces the epoch value `gmtime` consumes
- `POSIX::strftime` - format the 9-element list into an arbitrary date string (ISO-8601, locale-aware, etc.)
- `Time::Local` - inverse operation: build an epoch from broken-down UTC or local fields
- `Time::Piece` - object wrapper around `gmtime` / `localtime` with named accessors

Control flow

goto

Transfer execution elsewhere in the program without returning.

`goto` has three unrelated forms that share only a keyword. The useful one in modern Perl is `goto &NAME` - the **tail-call** form, which replaces the current subroutine frame with a call to another sub while preserving `@_`. The `goto LABEL` and `goto EXPR` forms exist, are constrained, and are almost never the right tool.

Synopsis

```
goto LABEL
goto EXPR
goto &NAME
```

What you get back

goto does not return. Control leaves the goto statement and does not come back. The return value of the construct the goto eventually lands in is the value of the enclosing expression.

For goto &NAME, the return value is whatever the target subroutine returns, propagated to the original caller as if the target had been called directly.

The tail-call form: goto &NAME

This is the form that matters. It exits the current subroutine - undoing any `local` bindings - and immediately calls NAME with the current `@_`. The current frame is **replaced**, not stacked on top of.

Consequences:

- `caller` inside the target sub sees the **original caller**, not the sub that issued the goto. The intermediate frame is gone.
- `@_` is passed through as-is. Modifications to `@_` before the goto are visible to the target.
- `local` changes made in the departing sub are undone before the target runs, exactly as they would be on a normal return.
- The target does not have to know it was reached via goto &; from its perspective it was called normally.

The canonical use is AUTOLOAD dispatch: resolve the real sub, then goto it so stack traces, `caller`, and `wantarray` all behave as though the real sub had been called in the first place.

NAME need not be a literal name. It can be any expression that yields a code reference - a scalar holding a coderef, a block returning one, or a dereferenced slot.

```
goto &$coderef;
goto &{ $dispatch{$op} };
```

For recursive tail calls, goto `__SUB__` jumps to the currently executing sub without re-resolving its name - safe under renaming and anonymous subs.

The goto LABEL form

Finds the statement labeled LABEL within the current dynamic scope and resumes execution there. In practice this means it can jump out of blocks and out of subroutines, but it **cannot jump into** a construct that requires initialization - a subroutine body, a `foreach` loop, a `given` block, the parameter of a binary or list operator, or code the optimizer removed. Jumping into such a construct is a fatal error as of Perl 5.44.

Most of what `goto LABEL` could do is better expressed with labeled loop control (`last LABEL`, `next LABEL`, `redo LABEL`) or with exceptions (`die` / `eval`). Reach for those first.

The `goto EXPR` form

Evaluates `EXPR`. If it yields a code reference, behaves like `goto &NAME`. If it yields a string, that string is used as a label name resolved at runtime - a computed goto. The same restrictions as `goto LABEL` apply to the label case.

Two parser quirks to remember:

- `goto EXPR` is exempt from the “looks like a function” rule. Parentheses after `goto` do not necessarily delimit its argument: `goto( "NE" ) . "XT"` is `goto NEXT`, not `goto "NE"` followed by a useless concatenation.
- `goto` has the same precedence as assignment, unlike most named operators.

Examples

Tail-call dispatch from AUTOLOAD - the target sub sees the original caller, not AUTOLOAD:

```
sub AUTOLOAD {
    our $AUTOLOAD;
    my ($method) = $AUTOLOAD =~ /::(\w+)\z/;
    my $code = $self->can_really($method)
        or die "no such method: $method";
    goto &$code;           # @_ forwarded, caller() unchanged
}
```

Tail recursion without growing the call stack. `__SUB__` resolves to the currently running sub:

```
sub fact {
    my ($n, $acc) = @_;
    $acc //= 1;
    return $acc if $n <= 1;
    @_ = ($n - 1, $acc * $n);
    goto __SUB__;
}
```

Computed goto via `goto EXPR`. The label is chosen at runtime from a list:

```
goto ("FOO", "BAR", "GLARCH")[$i];
FOO:  handle_foo();  return;
BAR:  handle_bar();  return;
GLARCH: handle_glarch(); return;
```

Delegating to a coderef held in a dispatch table. The caller never sees the intermediate frame:

```
my %ops = (
    add => sub { $_[0] + $_[1] },
```

(continues on next page)

(continued from previous page)

```
mul => sub { $_[0] * $_[1] },
);

sub run_op {
    my $op = shift;
    goto &{$ops{$op}} // die "unknown op: $op" ;;
}
```

Modifying @_ before a tail call - changes are visible to the target:

```
sub wrap {
    unshift @_, "prefix";
    goto &real_handler;      # real_handler sees ("prefix", @orig_
    ↪args)
}
```

Edge cases

- **goto LABEL cannot enter a sub, foreach, or given.** As of Perl 5.44 this is a fatal error, not a warning. It also cannot jump into optimized-away code or into most operator operands - the one exception is the first operand of a binary operator (for =, the right-hand side).
- **goto LABEL cannot get out of a sort comparison block.** The comparator runs in a context that does not permit the jump.
- **goto &NAME discards local bindings** from the departing sub before the target runs. Anything the sub localized is restored as though the sub had returned normally.
- **caller is invisible past a goto &.** The departing frame is gone. This is the whole point of the form for AUTOLOAD - do not rely on seeing the intermediate sub in a stack trace.
- **@_ aliasing is preserved.** The target sub's @_ aliases the same scalars the departing sub's @_ did. Assigning to \$_[0] in the target still modifies the caller's argument.
- **goto &\$coderef requires a real coderef.** A string that looks like a sub name does not work in the & form - use goto EXPR (without the &) for that, or resolve the name to a coderef first with \&{\$name}.
- **goto has assignment precedence.** goto "A" . "B" is goto "AB", not (goto "A") . "B".

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `last` - exit the innermost or a named enclosing loop; the right tool for most “jump out of here” cases
- `next` - skip to the next iteration of a loop, with optional label
- `redo` - restart the current loop iteration without re-evaluating the condition
- `return` - leave the current sub with a value; compare with `goto &NAME` which leaves the sub by **replacing** it
- `caller` - inspect the call stack; note that `goto &` makes the departing frame invisible to it

Lists

grep

Filter a list to the elements where the block or expression is true.

`grep` walks `LIST` from left to right, evaluates `BLOCK` or `EXPR` once per element with `$_` aliased to that element, and returns the elements for which the test produced a true value. It is a list filter, not a loop - the list you get back is a subset of the list you passed in, in original order.

Synopsis

```
grep BLOCK LIST
grep EXPR, LIST
my @kept = grep { COND } @list;
my @match = grep /pattern/, @list;
```

The two forms are interchangeable in power; `BLOCK` is the general case, and `EXPR` is a convenience for one-expression tests. When `EXPR` is a bare regex (`/.../`, `m/.../`, or `qr/.../`), it is implicitly matched against `$_` - this is the idiom behind `grep /^#/`, `@lines`.

What you get back

- **List context:** the filtered list, in original order, with each returned element an alias into the source list (see *Global state* and *Edge cases*).
- **Scalar context:** the count of elements for which the test was true. Not a boolean - it is the size of what the list context would have produced, so `0` is false, any positive count is true.

```
my @keep = grep { $_ > 0 } @nums; # list context: filtered list
my $count = grep { $_ > 0 } @nums; # scalar context: count
```

Global state it touches

- `$_` is **aliased** to each element of `LIST` for the duration of one `BLOCK` or `EXPR` evaluation. On exit from `grep`, `$_` is restored to whatever it held before the call. Because the alias is to the real element - not a copy - assigning to `$_` inside the block mutates the source list in place. This is the same contract as `map` and as a `for` loop's index variable.
- The bare-regex form of `EXPR` reads and writes all the usual match-related magic variables (`$1..$9`, `$&`, `${^MATCH}`, `@+`, `@-`, `%+`, `%-`, `pos`) on each iteration, exactly as if you had written the match longhand.

Examples

Filter out undefined values:

```
my @vals      = (1, undef, 2, undef, 3);
my @defined = grep { defined $_ } @vals;
# @defined = (1, 2, 3)
```

Regex filter via the expression form - the bare regex matches against `$_`:

```
my @lines      = ("# comment", "code", "# another", "more code");
my @code       = grep !/^#/ , @lines;      # drop comments
my @comments   = grep /^#/ , @lines;      # keep comments
```

Custom test via the block form, when the condition is more than one expression:

```
my @users      = (
    { name => "alice", active => 1, age => 30 },
    { name => "bob",   active => 0, age => 25 },
    { name => "carol", active => 1, age => 17 },
);
my @adults_active = grep {
    $_->{active} && $_->{age} >= 18
} @users;
```

Count without building the list - use scalar context:

```
my @errors = ("ok", "fail", "ok", "fail", "fail");
my $nfail  = grep { $_ eq "fail" } @errors; # 3
if (grep { $_ eq "fail" } @errors) {        # boolean use
    warn "had failures";
}
```

Remove blank and whitespace-only lines:

```
my @clean = grep { /\S/ } @lines;
```

Filter the keys of a hash by some property of the value:

```
my %scores = (alice => 42, bob => 17, carol => 95);
my @passing = grep { $scores{$_} >= 50 } keys %scores;
```

Edge cases

- **`$_` is an alias, not a copy.** Modifying `$_` inside the block modifies the original list element. This is occasionally useful but more often a bug; write a read-only test:

```
my @bumped = grep { $_++; $_ > 10 } @nums;    # MUTATES @nums
```

If the elements are themselves not variables (e.g. literals, constants, or the result of a function call), assigning to `$_` is a runtime error - “Modification of a read-only value”.

- **Returned elements are aliases too.** Elements of the returned list share storage with the source list, just like the index variable of a `for` loop. Writing to them through the returned list propagates back:

```
my @arr = (1, 2, 3, 4);
my @even = grep { $_ % 2 == 0 } @arr;
$even[0] = 99;      # @arr is now (1, 99, 3, 4)
```

This is rarely wanted. Copy explicitly with `map { $_ } grep ...` or a fresh `my` when you need independent storage.

- **Expression form vs. block form - precedence.** `grep` `EXPR`, `LIST` uses a comma to separate the test from the list; `grep` `BLOCK` `LIST` does not. Mixing them trips you up:

```
grep /x/, @arr;      # EXPR form, correct
grep { /x/ } @arr;   # BLOCK form, correct
grep { /x/ }, @arr;  # WRONG: comma after block is a syntax
                    # error or silently wrong depending on
                    # context
```

- **`grep` in chained expressions.** `grep` takes a `LIST`, which means everything after the first argument is consumed as part of the list. Parenthesise when you want to follow `grep` with more terms:

```
my @out = (grep { /x/ } @a), @b;    # concatenates filtered @a
↪and @b
my @bad = grep { /x/ } @a, @b;     # filters BOTH @a and @b
↪together
```

- **Not a loop.** `grep` is an expression, not a control structure. `last`, `next`, and `redo` inside the block target the nearest enclosing *loop*, not the `grep` itself - they will exit or restart whatever `for/while` the `grep` sits inside, not the `grep`. Use an early `return` from a helper sub if you need to short-circuit, or switch to `List::Util` first when all you want is the first match.
- **Empty input is empty output.** `grep` over an empty list is an empty list in list context and `0` in scalar context. No special case needed.
- **Hash flattening.** `grep { ... } %h` walks the hash’s alternating key/value flattening; the block sees keys and values interleaved in `$_`. This is almost never what you want; filter the keys and rebuild instead:

```
my %filtered = map { $_ => $h{$_} }  
                grep { some_test($_, $h{$_}) } keys %h;
```

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `map` - same iteration contract with the same `$_` alias semantics, but returns what the block *produced* for each element rather than the element itself
- `sort` - reorder a list; often chained with `grep` as `sort { ... } grep { ... } @list`
- `any` / `all` - short-circuiting boolean tests over a list from `List::Util`; use these instead of `scalar grep` when you only need a yes/no answer and the list is large or the test is expensive
- `first` (from `List::Util`) - return the first element matching a block and stop; the right tool when `grep` is being used only to fetch one match
- `for` / `foreach` - the looping construct with the same `$_` aliasing rule; reach for it when you want side effects, not a filtered list
- `$_` - the default scalar that `grep` aliases on every iteration

[SCALARs and strings](#) · [Numeric functions](#)

hex

Interpret a string as a hexadecimal numeral and return its numeric value.

`hex` reads `EXPR` as a sequence of hex digits - optionally introduced by a `0x` or `x` prefix - and returns the integer those digits represent. If `EXPR` is omitted, `$_` is used. The companion operation, turning a number back into a hex string, is `sprintf` with the `"%x"` or `"%X"` format.

Synopsis

```
hex EXPR  
hex                                     # operates on $_
```

What you get back

A non-negative number. `hex` itself never returns a negative value: it has no notion of sign, it only decodes digits.

- If the result fits in a native integer (typically 64-bit unsigned on a 64-bit build), you get an integer.

- If the decoded value exceeds that range, hex falls back to an IEEE 754 double. A warning is emitted under `use warnings` and precision may be lost for values above 2^{53} .

To convert the other direction:

```
printf "%x", 255           # "ff"
printf "0x%08X", 4096      # "0x00001000"
```

Examples

Plain hex digits, with or without the `0x` prefix:

```
print hex '0xff';          # 255
print hex 'ff';            # 255
print hex 'DEADBEEF';      # 3735928559
```

Underscores between digits are permitted as visual separators, the same way numeric literals allow them:

```
print hex '1_0000';        # 65536
print hex 'cafe_f00d';     # 3405700109
```

Operating on `$_`:

```
for ('10', '20', 'ff') {
    print hex, "\n";        # 16, 32, 255
}
```

Round-tripping a value through hex:

```
my $n = 0xC0FFEE;
my $s = sprintf "%x", $n;    # "c0ffee"
print hex $s;                # 12648430
```

Values that overflow native integer width warn and return a float:

```
use warnings;
my $big = hex 'ffffffffffffffffffff'; # 19 'f's - too wide for u64
# "Integer overflow in hexadecimal number" warning
# $big is now an IEEE 754 double, with lost precision
```

Edge cases

- **Non-hex characters terminate parsing.** The first character that is not a hex digit, underscore, or recognised prefix ends the scan. Under `use warnings` this emits `Illegal hexadecimal digit '...' ignored`:

```
use warnings;
print hex '1f garbage';    # 31, with a warning
print hex '0xAZ';          # 10, with a warning on 'Z'
```

- **Leading whitespace is not skipped.** Unlike `oct`, `hex` treats whitespace as a non-hex character, so a leading space stops parsing before any digit is seen:

```
print hex ' ff';           # 0
```

- **Empty string and `undef` return 0.** An empty EXPR yields 0 with no warning. A bare `undef` yields 0 and emits the usual Use of uninitialized value warning under `use warnings`.
- **No sign handling.** A leading - or + is a non-hex character and terminates parsing before any digit, so `hex '-ff'` returns 0. To parse a signed hex string, strip the sign yourself:

```
my $s = '-ff';  
my $n = ($s =~ s/^-//) ? -hex($s) : hex($s); # -255
```

- **Prefix forms.** The accepted prefixes are `0x`, `0X`, `x`, and `X`. A lone 0 is not a prefix - it's a digit. There is no `#` or `$` prefix.
- **Case is irrelevant** for the digits a-f and for the prefix letter x.
- **Underscore placement.** A single underscore may precede each digit. Two consecutive underscores, a leading underscore before the prefix, or a trailing underscore terminate parsing as non-hex characters.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `oct` - decodes strings with auto-detected base: `0x...` as hex, `0b...` as binary, `0...` as octal, anything else as decimal. Use it when the input's base is not known in advance
- `sprintf` - the reverse direction; `"%x"` / `"%X"` format a number into a lowercase or uppercase hex string, `"%#x"` adds the `0x` prefix, `"%08x"` zero-pads to a fixed width
- `printf` - same `"%x"` formatting, written straight to a filehandle
- `pack` - templates `H` (high nibble first) and `h` (low nibble first) convert between a hex-digit string and packed binary bytes; use these when moving hex data to and from a binary protocol
- `unpack` - the inverse of `pack`; with `"H*"` it turns a byte string into a hex digit string in one call

Scoping · Modules

import

Populate the caller's namespace with names a module chooses to export.

`import` is **not** a built-in function. It is a class method a module defines so that `use` has somewhere to hand off the argument list. When you write `use Module LIST`, Perl loads the module and then calls `Module->import(LIST)` at compile time, in the package that

contains the use statement. Whatever that method does - install subroutines into the caller's stash, flip feature bits, set pragma flags - is the module's entire import contract.

Most modules never write `import` by hand; they inherit it from `Exporter` and declare `@EXPORT/@EXPORT_OK` instead. Pragmas (`strict`, `warnings`, `feature`) write their own `import` because what they install is lexical state, not subroutines.

Synopsis

```
Module->import;
Module->import(LIST);
Module->import(@symbols);
```

Inside a module, the definition looks like:

```
package My::Module;
sub import {
    my ($class, @args) = @_;
    # install names into caller's package
}
```

What you get back

The return value of `import` is discarded by `use`. A module that is called directly as a method inherits whatever convention the author chose - by convention, return nothing useful.

The visible effect is in the caller's package: after `use Module qw(foo bar)` returns, `foo` and `bar` are callable without a package prefix from the code that ran the `use`.

How use calls import

`use Module LIST` is, in effect:

```
BEGIN {
    require Module;
    Module->import(LIST);
}
```

Three consequences fall out of that equivalence:

- **Compile time.** `import` runs while the enclosing file is still being parsed. Any subroutine prototypes or lexical pragmas it installs are visible to the rest of the file from that point on. Anything done at runtime - opening files, reading configuration - happens earlier than a naive reading of the source suggests.
- **caller reflects the BEGIN block**, not the surrounding subroutine. An `import` method that inspects `caller(0)` sees the package that wrote `use Module`, which is exactly what it wants; it does **not** see the subroutine that contains the `use` statement.
- **Absent list vs empty list.** `use Module` with no list calls `Module->import` with no arguments - the module installs its defaults. `use Module ( )` with an **explicit**

empty list skips the import call entirely. This is the idiom for “load the file but do not touch my namespace.”

no Module LIST is the mirror image: it calls `Module->unimport(LIST)` instead of `import`. Pragmas use this to turn features off for the enclosing lexical scope.

Defining an import method

The common path is inheritance from `Exporter`:

```
package My::Utils;
use Exporter 'import';           # installs Exporter::import
our @EXPORT_OK = qw(slurp trim);
sub slurp { ... }
sub trim { ... }
1;
```

Callers then write:

```
use My::Utils qw(slurp trim);
slurp($path);
```

A hand-rolled import has the same shape as any method - it receives the class name as its first argument and the use argument list as the rest:

```
package My::Feature;
sub import {
    my ($class, @flags) = @_;
    my $caller = caller;
    no strict 'refs';
    for my $name (@flags) {
        *{ "${caller}::${name}" } = \&{ "${class}::${name}" };
    }
}
```

The caller’s package name comes from `caller`, not from `$class` - `$class` is whatever module the user named on the use line.

Examples

Default import - whatever the module’s @EXPORT list contains:

```
use List::Util;                 # imports nothing by default
use File::Basename;            # imports basename, dirname, ...
↪ ...
```

Explicit import list - only the names you ask for:

```
use List::Util qw(sum min max);
my $total = sum(1, 2, 3);      # 6
```

Empty list - load the file, touch nothing:


```
use POSIX ();                                # no POSIX::* names imported
my $pi = POSIX::acos(-1);                   # fully qualified call
```

Importing from a pragma - lexical effect in the enclosing block:

```
use strict;
use warnings;
{
    no warnings 'uninitialized';           # unimport in this block only
    print "x=", $x, "\n";
}
```

Calling import directly after a runtime `require`:

```
require My::Plugin;
My::Plugin->import(@config);               # deferred to runtime on _
↪purpose
```

This pattern loads the module only when a code path actually needs it, trading compile-time checking for startup speed.

Edge cases

- **No such thing as a built-in import.** `print import(@list)` calls the subroutine named `import` in the current package (if one exists); it is not a language-level operator. Parsing rules apply as for any user-defined subroutine.
- **Omitted list vs empty list** is the most common source of confusion. `use Mod` calls `import`; `use Mod ()` does not. The `()` is not a stylistic empty - it is a signal.
- **Version argument runs VERSION, not import.** `use Mod 1.23 LIST` calls `Mod->VERSION(1.23)` **before** `Mod->import(LIST)`. A bare `use Mod 1.23` is still a version check followed by a default import. To version-check and skip imports, write `use Mod 1.23 ()`.
- **No import defined at all.** If the module neither writes nor inherits an `import`, `use Module LIST` with a non-empty list does not error - the method call falls through silently as `UNIVERSAL::can('import')` returns false and `use` treats it as “nothing to do.” Non-empty `LIST` is then effectively ignored, which is almost never what the caller wanted.
- **import runs once per use statement**, not once per process. `use Mod qw(foo); use Mod qw(bar);` calls `import` twice, even though the file is loaded only once - `@INC / %INC` gate the file load, not the method call.
- **Symbol installation is not automatic.** Writing an `import` method that merely returns does nothing; you must assign into the caller’s stash (`*{"${caller}::name"} = \&sub`) or delegate to `Exporter`.
- **Pragmas typically install lexical state**, not package-level subroutines. Their `import` calls functions like `warnings::import_into` or manipulates `%^H`; this is why a pragma’s effect is scoped to the enclosing block rather than persisting in the package.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `use` - the compile-time loader that invokes `import`; understanding one requires the other
- `no` - the mirror of `use`; calls `unimport` with the same argument-passing rules
- `require` - loads the file without calling `import`; pair with an explicit `Module->import(...)` when you want to defer or skip the compile-time dispatch
- `package` - defines the namespace that an `import` method populates from the caller's point of view
- `caller` - how an `import` method discovers which package to install names into
- `bless` - unrelated to `import`, but often invoked in the same module files; both shape how a package presents itself to its users

SCALARs and strings

index

Find the position of a substring inside a string.

`index` scans `STR` left-to-right looking for the first occurrence of `SUBSTR` and returns the zero-based position where it starts. No regular-expression metacharacters, no case folding, no wildcards - `SUBSTR` is matched literally, character for character. When the search fails, `index` returns `-1`.

Synopsis

```
index STR, SUBSTR
index STR, SUBSTR, POSITION
```

What you get back

An integer. On a match, the zero-based offset of the first character of `SUBSTR` inside `STR`. On no match, `-1`. The sentinel is the idiomatic way to test:

```
if (index($line, $needle) >= 0) { ... } # found
if (index($line, $needle) == -1) { ... } # not found
```

`POSITION` and the return value use the **same** zero-based scale, so you can feed one back into the other to walk every occurrence:

```
my $pos = -1;
while (($pos = index($text, $needle, $pos + 1)) != -1) {
    push @hits, $pos;
}
```

How POSITION is interpreted

POSITION is the earliest offset the match is allowed to start at. The search still proceeds to the end of STR; POSITION does not bound the search, it only shifts where it begins.

- POSITION omitted or `undef` - search from offset 0.
- POSITION negative or otherwise before the start - treated as 0.
- POSITION past the end of STR - treated as the end, so the only way to match is if SUBSTR is the empty string (which matches at any offset, including the end).

An empty SUBSTR always matches, and matches at POSITION (clamped into range). This follows from the “first position where SUBSTR occurs” rule: the empty string occurs everywhere.

```
index("hello", "");      # 0
index("hello", "", 3);    # 3
index("hello", "", 99);   # 5    (clamped to end of string)
```

Examples

Find a single character or a whole word:

```
index("Perl is great", "P");    # 0
index("Perl is great", "g");    # 8
index("Perl is great", "great"); # 8
```

Report a miss:

```
index("Perl is great", "Z");    # -1
```

Skip past an earlier match with POSITION to find the second occurrence:

```
index("Perl is great", "e", 5); # 10
```

Walk every occurrence of a substring:

```
my $s = "abcabcabc";
my $p = -1;
while (($p = index($s, "bc", $p + 1)) != -1) {
    print "hit at $p\n";
}
# hit at 1
# hit at 4
# hit at 7
```

A common idiom - test for containment without building a regex:

```
if (index($path, "/tmp/") != -1) {
    warn "path touches /tmp";
}
```

Pairs naturally with `substr` to split on the first occurrence of a separator:

```
my $line = "key=value=with>equals";
my $eq   = index($line, "=");
my ($k, $v) = $eq >= 0
    ? (substr($line, 0, $eq), substr($line, $eq + 1))
    : ($line, undef);
```

Edge cases

- **Empty SUBSTR** matches at POSITION (clamped into STR). `index($s, "")` is 0; `index($s, "", $n)` is \$n capped at length \$s. Never -1.
- **Empty STR with non-empty SUBSTR** returns -1. Empty STR with empty SUBSTR returns 0.
- **Negative POSITION** is clamped to 0. `index` does **not** interpret negative offsets as “from the end” - that is `rindex`’s job, and even there the semantics differ.
- **POSITION past the end of STR** is clamped to length STR, so only an empty SUBSTR can match.
- **Undef arguments** stringify to "" and trigger an uninitialized warning under use warnings. `index(undef, "x")` is -1; `index("abc", undef)` is 0 (empty-substring rule).
- **Characters, not bytes.** `index` operates on the logical character sequence of the string. For a string of wide characters, the returned offset is a character offset, not a byte offset. If you need byte offsets, downgrade or encode the string first (use `bytes` for a lexical byte view, or `Encode:::encode_utf8` to work on an octet string).
- **Case sensitivity:** `index` is case-sensitive. Lowercase both arguments first if you want a case-insensitive search, or use `=~ /\Q$needle\E/i` and `@- / $- [0]` to recover the position.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `rindex` - same matching rules, scans from the right and returns the offset of the **last** occurrence at or before POSITION
- `substr` - extract the matched region once `index` has located it, or replace it in place
- `length` - upper bound for a valid POSITION; returns character length on the same scale `index` uses
- `pos` - position tracking for regex-based scanning; use together with `m/ /g` when you need captures rather than a raw offset
- `sprintf` - build the search string when SUBSTR is assembled from parts; `index` takes a literal, so prepare it first

Numeric functions

int

Return the integer portion of a number, truncating toward zero.

`int` takes a single scalar, forces it to numeric context, and drops its fractional part. The discard is always **toward zero** - not toward negative infinity - so `int(3.9)` is 3 and `int(-3.9)` is -3, not -4. Non-numeric strings are coerced with the usual Perl numeric-conversion rules (leading whitespace, optional sign, digits; the rest is discarded, with an Argument "... isn't numeric warning under use warnings). If `EXPR` is omitted, `int` operates on `$_`.

Synopsis

```
int EXPR
int
int($x)
```

What you get back

A number with no fractional part, same sign as the input. The result is an integer value if the truncated magnitude fits in the platform's native integer range (typically 64-bit); otherwise Perl keeps it as a floating-point value whose fractional part happens to be zero. Either way it compares and prints as an integer.

```
my $n = int(7.8);      # 7
my $m = int(-7.8);     # -7
```

`int` is not a rounding function. For rounding, see *See also*.

Examples

Positive truncation - the fractional part is discarded:

```
print int(3.9), "\n";      # 3
print int(3.1), "\n";      # 3
print int(3.0), "\n";      # 3
```

Negative truncation - the trap. `int` truncates toward zero, so `int(-3.9)` is -3, which is *larger* than -3.9. This is the opposite of a mathematical floor:

```
print int(-3.9), "\n";     # -3   (not -4)
print int(-3.1), "\n";     # -3
```

Works on `$_` when called with no arguments - handy inside `map` or `while (<>)`:

```
my @whole = map { int } @floats;  # truncate every element
```

Strings are coerced, keeping the leading numeric prefix:

```
print int("42.7 bottles"), "\n";  # 42
print int(" -5.5xyz"), "\n";      # -5
```

(continues on next page)

(continued from previous page)

```
print int("hello"), "\n";           # 0      (with a warning under_
↪ `use warnings`)
```

Division producing a fraction, then truncated - a common idiom for integer division of signed values, **with the caveat above about negatives**:

```
my $pages = int($total_items / $per_page); # floor only for non-
↪ negative $total_items
```

The classic floating-point surprise. $-6.725 / 0.025$ is *mathematically* -269 , but the IEEE-754 result is slightly greater than -269 (around -268.99999999999994). `int` truncates toward zero, giving -268 :

```
print int(-6.725 / 0.025), "\n";    # -268, not -269
```

This is not an `int` bug - every truncation/rounding function has to cope with inputs like this. See *Edge cases* below.

Edge cases

- **`int` on `$_`:** with no argument, operates on `$_`. Inside `map { int }` this is what you want; inside a larger expression it can surprise a reader who missed the implicit argument. Prefer `int($x)` when the source variable isn't obvious from context.
- **Already-integer input:** `int(5)` is 5. No conversion, no warning, no precision loss.
- **Very large floats lose precision before `int` sees them.** A double-precision float can represent every integer up to 2^{53} exactly; beyond that, consecutive representable values are spaced 2, then 4, then 8 apart. `int(1e20)` returns a float whose integer value is close to but not equal to 10^{20} . If you need exact large integers, keep them as integers throughout - don't route them through floating point.
- **Floating-point representation artefacts:** inputs like $-6.725 / 0.025$ (see *Examples*) don't land on the mathematical integer they "should." Use `sprintf "%.0f"` or `POSIX::floor/POSIX::ceil` if you need rounding behaviour that tolerates this.
- **Infinites and NaN:** `int('Inf')` returns `Inf`; `int('-Inf')` returns `-Inf`; `int('NaN')` returns `NaN`. `int` has no integer to truncate to, so it returns the special value unchanged. This matches every alternative (`sprintf "%d"`, `POSIX::floor`, `POSIX::ceil`) - none of them give you a finite integer here.
- **undef:** `int(undef)` returns 0 and emits a `Use of uninitialized value` warning under `use warnings`.
- **Non-numeric strings:** `int("abc")` returns 0 and emits `Argument "abc" isn't numeric` under `use warnings`. A string that starts with a numeric prefix ("42abc") yields the prefix's integer value (42) with the same warning - `int` inherits this from Perl's standard numeric coercion, not from any custom parsing.
- **Booleans and dualvars:** `int` reads the numeric half. `int(!!1)` is 1; `int(!!0)` is 0. For a dualvar whose string and numeric values disagree, `int` uses the numeric side.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `abs` - magnitude without sign; pair with `int` when you want the truncated magnitude regardless of sign
- `sprintf` - `sprintf "%d", $x` also truncates toward zero, but additionally formats the result as a string; use when you want the truncated value *and* a specific textual form
- `POSIX::floor` - round *toward negative infinity*; `floor(-3.9)` is -4, which is what most mathematical contexts (graphics, bucketing) actually want
- `POSIX::ceil` - round *toward positive infinity*; `ceil(-3.1)` is -3, `ceil(3.1)` is 4; the mirror of `floor`
- `POSIX::round` - round to the nearest integer, halves away from zero; use this when you want statistical rounding rather than truncation
- `sprintf` with `"%.0f"` - round to nearest integer, halves to even (banker's rounding) on most platforms; subtly different from `POSIX::round` at half-values

Filehandles, files, directories

ioctl

Perform a device-control `ioctl(2)` system call on a filehandle.

`ioctl` is the direct Perl binding for the POSIX `ioctl(2)` syscall - the catch-all kernel interface for device-specific operations that do not fit the regular read/write model: terminal window size, serial line parameters, socket device queries, tape positioning, and anything else a driver chooses to expose through a request code. It dispatches `FUNCTION` against `FILEHANDLE`, passing `SCALAR` as the third argument - either a small integer or a pointer to a byte buffer, depending on what the specific `FUNCTION` expects.

Unlike `fcntl`, whose request codes live in the `Fcntl` module, `ioctl` request codes are almost never exported as ready-made constants. The usual path is:

```
require "sys/ioctl.ph";
```

which pulls in the translated `<sys/ioctl.h>` header. If that file does not exist, or does not define the code you need, build it from the C headers with the `h2ph` tool shipped with Perl, or define the constant yourself from the kernel header.

Synopsis

```
require "sys/ioctl.ph";
ioctl FILEHANDLE, FUNCTION, SCALAR
ioctl($fh, $request, $buf)
my $rv = ioctl($fh, $request, $arg) || -1;
```

What you get back

On success, the integer the kernel returned. When that integer is 0, Perl substitutes the dual-valued string "0 but true" - true in boolean context, numeric 0 in numeric context, and exempt from the Argument "... isn't numeric" warning. This means a plain boolean test reliably distinguishes success from failure even for requests whose success value is literally zero:

```
ioctl($fh, $request, $buf) or die "ioctl: $!";
```

On failure, `ioctl` returns `undef` and sets `$!` to the relevant `errno`. The full mapping:

OS returns	Perl returns
-1	<code>undef</code>
0	string "0 but true"
any other value	that integer

When you need the raw kernel return - for request codes that encode information in the return value itself - use the idiom from `perlfunc`:

```
my $retval = ioctl($fh, $request, $arg) || -1;
printf "System returned %d\n", $retval;
```

How SCALAR is used

SCALAR is handed to the kernel as the third argument of the C `ioctl` call, and its role depends on the request code:

- **Pointer to a buffer.** The usual case. Most `ioctl` requests read from or write into a C struct whose layout is defined by the driver. Build the buffer with `pack` before the call and, for requests that write back, decode with `unpack` afterward. Pre-size the scalar to at least the struct's length - Perl will grow an undersized scalar automatically, but pre-sizing makes the intent obvious and avoids surprises if the driver writes more than you expected.
- **Small integer.** A minority of requests take a bare integer rather than a pointer. If SCALAR has no string value but does have a numeric value, Perl passes the number instead of a pointer. To guarantee this dispatch - even for a scalar that might have been stringified earlier - add 0 to it first:

```
my $n = 0 + $n;      # force numeric representation
ioctl($fh, $request, $n);
```

- **Ignored.** For request codes that take no third argument, pass 0 as a placeholder. The kernel ignores it; the convention keeps the call readable.

Global state it touches

- `$!` - set on failure to the `errno` returned by `ioctl(2)`.
- Device or descriptor state on `FILEHANDLE` - `ioctl` typically mutates kernel state associated with the underlying file descriptor or its device (terminal attributes, socket options, tape position). That state is not Perl-visible through any other variable; read it back with another `ioctl` call using the matching `TIOCG...` / `SIOCG...` request.

Examples

Query the terminal window size on Linux. `TIOCGWINSZ` writes a `struct winsize` (two unsigned short rows/cols plus two ignored fields) into the buffer:

```
require "sys/ioctl.ph";

my $winsize = "\0" x 8;
ioctl(STDOUT, TIOCGWINSZ(), $winsize)
    or die "TIOCGWINSZ: $!";
my ($rows, $cols) = unpack('S S', $winsize);
print "terminal is $rows rows by $cols cols\n";
```

Count bytes available for reading on a socket without consuming them (`FIONREAD` is widely supported across Unixes):

```
require "sys/ioctl.ph";

my $pending = pack('L', 0);
ioctl($sock, FIONREAD(), $pending)
    or die "FIONREAD: $!";
my $n = unpack('L', $pending);
print "$n bytes pending\n";
```

Put a socket into non-blocking mode through the `FIONBIO` request - an alternative to the `fcntl` / `F_SETFL` approach, and the form historically portable to some older systems:

```
require "sys/ioctl.ph";

my $on = pack('L', 1);
ioctl($sock, FIONBIO(), $on)
    or die "FIONBIO: $!";
```

Distinguishing “syscall returned 0” from “syscall failed” using the “`0` but `true`” convention:

```
my $rv = ioctl($fh, $request, $buf);
if (!defined $rv) {
    die "ioctl failed: $!";
} elsif ($rv eq "0 but true") {
    # kernel returned 0 - success with a zero value
} else {
```

(continues on next page)

(continued from previous page)

```
# kernel returned $rv
}
```

Rolling your own request code when `sys/ioctl.ph` is missing or incomplete. The encoding (`_IOR / _IOW / _IOWR`) is kernel-specific; on Linux the macro expands to a 32-bit value that you can precompute and hardcode:

```
use constant TIOCGWINSZ => 0x5413;    # Linux <asm-generic/ioctls.h>
my $winsize = "\0" x 8;
ioctl(STDOUT, TIOCGWINSZ, $winsize) or die $!;
```

Edge cases

- **Forgot the .ph require:** constants like `TIOCGWINSZ` parse as barewords with no defined value, triggering an `EINVAL` from the kernel. Either require `"sys/ioctl.ph"` or define the constant yourself with `use constant`.
- **"0 but true" return:** do not compare the return with `== 0` as a failure test. Use a plain boolean (`or die`) or `defined`. Comparing for equality to the literal string also works (`$rv eq "0 but true"`) but is rarely necessary.
- **Scalar/integer dispatch surprise:** a scalar that was numeric but got stringified - for instance via interpolation into a message - flips from integer-argument to buffer-pointer semantics on the next call. Force numeric representation with `0 + $x` when the request code expects an integer.
- **Struct layout is platform-specific.** `struct winsize`, `struct termios`, `struct ifreq`, and friends differ between kernels and even between architectures on the same kernel. Do not hardcode `pack` templates from memory; verify against the target system's header.
- **Undersized output buffer.** If the driver writes more bytes than `SCALAR` currently holds, the scalar is grown, but you have no hint in Perl that this happened. Pre-size to at least the struct's length with `"\0" x N` so that reads past the end of the buffer are driver bugs, not Perl bugs.
- **Not every system implements every request.** `ioctl` itself is POSIX, but specific request codes are driver-defined. An unsupported request fails with `ENOTTY` or `EINVAL`; an `ioctl` call on a system without `ioctl(2)` at all raises an exception.
- **ioctl vs fcntl:** identical argument-processing and return-value convention, different kernel interface. Use `fcntl` for descriptor flags and POSIX locks; use `ioctl` for device control codes.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `fcntl` - same calling and return-value convention; used for descriptor-level flags and advisory locks rather than device-specific control codes
- `pack` / `unpack` - required for building and decoding the struct buffers most `ioctl` requests exchange with the kernel
- `open` / `sysopen` - open the device or special file whose driver the `ioctl` request will talk to
- `binmode` - some `ioctl` requests (e.g. terminal mode changes) interact with PerlIO buffering; switch to raw mode when the driver and PerlIO disagree about line discipline
- `select` / `sysread` - the normal I/O path; reach for `ioctl` only when the operation you need is not expressible as a read or write

Classes and OO

isa

Test whether an object is an instance of a class or of any subclass derived from it.

`isa` is a **binary operator**, not a function. The left operand is an expression that should yield an object reference; the right operand names a class. It returns true when the left operand is a blessed reference whose class is - or inherits from - the class on the right. It returns false for anything else, including `undef`, unblessed references, and plain strings. It never throws.

Unlike the old `UNIVERSAL::isa` method, `isa` is safe to apply to arbitrary values: calling `$x->isa("Foo")` on a plain string that happens to name a class (`$x = "Foo"`) invokes the class method and returns true, which is almost never what the caller meant. `isa` as an operator only reports true for **blessed instances**.

Synopsis

```
use feature 'isa';           # or: use v5.36;

$obj isa Some::Class
$obj isa "Different::Class"
$obj isa $name_of_class
```

What you get back

1 for a blessed object whose class is, or derives from, `CLASS`. The defined empty string `"` (a false value that is numerically zero and exempt from uninitialized warnings) in every other case.

The operator never raises an exception: undefined, unreferenced, and misbranded values simply test false.

Feature gate

`isa` is available from Perl 5.31.6 and became non-experimental in Perl 5.36. It is a **lexical feature**, enabled by either of:

```
use feature 'isa';
use v5.36;           # enables the 5.36 feature bundle
```

Under no feature `'isa'` (or in a scope that never enabled it), the bareword `isa` reverts to being a normal identifier, and the expression is a syntax error.

The right-hand side

The right operand of `isa` is resolved to a class name at compile time when it is a bareword:

```
$obj isa Some::Class      # bareword - resolved at compile time
```

Any other expression is evaluated at runtime and must yield a string naming the class:

```
$obj isa "Different::Class" # string literal
$obj isa $class_name        # scalar variable
$obj isa (ref $other)       # any expression yielding a string
```

Barewords on the right side do **not** need to be declared, imported, or loaded; `isa` only consults the symbol table of `CLASS` to walk `@ISA`. A class that was never loaded simply has no parents, so `$obj isa Unknown::Class` just tests false.

Examples

Basic instance check:

```
use feature 'isa';

package Animal { }
package Dog    { our @ISA = ('Animal'); }

my $rex = bless {}, 'Dog';

$rex isa Dog;           # true
$rex isa Animal;        # true - via @ISA
$rex isa Cat;           # false
```

Gate a method call on the actual class:

```
sub describe {
    my ($thing) = @_;
    return "unknown" unless $thing isa Animal;
    return $thing->sound;
}
```

Safe on arbitrary values - no exception on non-references:

```

undef      isa Animal;      # false
"Animal"   isa Animal;      # false - a string is not an instance
[]         isa Animal;      # false - unblessed ref
{}         isa Animal;      # false - unblessed ref
\&main::x  isa Animal;      # false - unblessed code ref

```

Dispatch on a class name captured at runtime:

```

for my $class (qw(Dog Cat Fish)) {
    if ($rex isa $class) {
        say "rex is a $class";
    }
}

```

Why `isa` is preferred over `UNIVERSAL::isa` as a method:

```

my $x = "Dog";
$x->isa('Dog');           # TRUE - string "Dog" is treated as a
    ↪ class name
$x isa Dog;               # false - $x is not a blessed instance

```

The first form silently does the wrong thing on any string that happens to match a loaded class name. The operator form is type-aware and reports what the caller actually wanted to know.

Edge cases

- **Left operand is undef:** the operator returns false. No uninitialized warning is emitted - this is a deliberately quiet test.
- **Left operand is an unblessed reference:** false. `isa` requires a blessed reference; array refs, hash refs, and code refs that have never been passed through `bless` fail the test.
- **Left operand is a plain string:** false, even when the string matches a loaded class name. This is the behavioural split from `UNIVERSAL::isa` called as a method.
- **Right operand names an unknown class:** the operator does not autoload or require the module. It walks the class's `@ISA`; an unknown class has no `@ISA`, so the test is false.
- **Multiple inheritance:** `isa` honours whatever method resolution order is active in the class (`mro`, `c3` or the default depth-first search). The lookup is the same walk that method dispatch uses.
- **Reblessed objects:** the test reflects the current blessing. Calling `bless` again on a reference changes what `isa` reports.
- **Parser precedence:** `isa` is a non-associative named operator at the same precedence as the relational operators (`lt`, `gt`, `le`, `ge`). Chaining (`$a isa B isa C`) is a syntax error, as with the other relationals.
- **Feature not enabled:** without use feature `'isa'` (or use `v5.36`), the parser sees `isa` as an ordinary bareword and the expression fails to compile.

- **Right operand as a subroutine call:** write it in parens. `$obj isa some_class()` parses; `$obj isa some_class` treats `some_class` as the literal class name (bareword).

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `bless` - attach a class name to a reference so that `isa` (and method dispatch) has something to report
- `ref` - retrieve the class name of a blessed reference, or the underlying type for unblessed ones; complements `isa` when you need the name rather than a yes/no answer
- `class` - the native class declaration syntax (feature `class`, Perl 5.38+); objects built with `class` are ordinary blessed references and work with `isa` unchanged
- `UNIVERSAL::isa` - the historical method form. Works on every reference, but also returns true for plain strings that happen to name a class; use the operator in new code
- `mro` - controls the method resolution order that `isa` walks through `@ISA`
- `perlop` - full operator-precedence table; `isa` sits with the non-associative relational operators

Lists

join

Concatenate a list of strings with a separator between each adjacent pair, and return the single resulting string.

`join` takes a separator as its first argument and any number of values as the rest. It stringifies every value in `LIST`, glues them together with `EXPR` placed between each pair, and returns one scalar. The separator is inserted **between** items only - never at the start, never at the end. With zero or one items the separator never appears at all.

Synopsis

```
join $sep, @list
join "", @chars
join "\n", @lines
```

What you get back

A single string. The result is a fresh scalar; modifying it does not reach back into `LIST`. In scalar context you get that string; `join` is effectively always scalar because it only ever produces one value.

Examples

Comma-separated values, the canonical use:

```
my @cols = ("alice", 42, "ops");
my $row  = join ",", @cols;           # "alice,42,ops"
```

Build a file path with / as the separator:

```
my @parts = ("var", "log", "app.log");
my $path  = "/" . join("/", @parts); # "/var/log/app.log"
```

Join lines with a newline between each - note that the trailing line does **not** get a newline, because join puts the separator *between* items:

```
my @lines = ("first", "second", "third");
my $text  = join "\n", @lines;         # "first\nsecond\nthird"
```

No-separator concatenation. Use the empty string "", not `undef` - `undef` triggers an uninitialized warning under `use warnings` and stringifies to "" anyway:

```
my @chars = ("h", "e", "l", "l", "o");
my $word  = join "", @chars;          # "hello"
```

An empty LIST yields an empty string - this is safe to rely on and needs no guard:

```
my @empty;
my $s = join ",", @empty;             # ""
```

Edge cases

- **Empty LIST:** returns "". The separator is never used.
- **Single-item LIST:** returns that item stringified, with no separator anywhere: `join ",", ("only")` is "only".
- **Separator is stringified:** pass a number, a reference, an object with overloaded "" - join calls the same stringification every other Perl operator would. `join 0, 1, 2, 3` is "10203".
- **Multi-character separator:** EXPR can be any string, not just a single character. `join " ", "a", "b", "c"` gives "a, b, c".
- **undef items in LIST:** each `undef` stringifies to "" and emits an uninitialized warning under `use warnings`. Two adjacent `undef`s still get a separator between them: `join " ", (undef, undef)` is ", ".
- **join is not the inverse of split on a single character** when the separator you pass differs from the pattern used to split. `split` takes a *pattern*; join takes a literal string. Round-tripping `join "\t", split /\s+/, $s` loses the distinction between runs of whitespace and a single tab.
- **Not a pattern:** unlike `split`, the first argument to join is not a regex. `join /,/, @x` is a syntax surprise - the separator is always a string.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `split` - the inverse: take a string and a pattern, return a list of pieces
- `sprintf` - when you need per-item formatting, padding, or field widths that join can't express
- `pack` - when the output is a binary record, not a separator-delimited text string
- `map` - transform items before joining, e.g. `join ",", map { uc } @names`

[Arrays](#) · [Hashes](#)

keys

List all keys of a hash, or all indices of an array.

`keys` is the introspection primitive for associative and indexed containers. Given `%HASH` it returns the hash's keys; given `@ARRAY` (from Perl 5.12 onward) it returns the array's indices `0 .. $#ARRAY`. The list is what you iterate over, count, sort, or feed to another list-consuming operator. `keys` is also the idiomatic way to reset the iterator that `each` walks.

Synopsis

```
my @k = keys %h;
my @i = keys @a;
my $n = keys %h;           # scalar context: count
keys(%h) = 1000;          # preallocate buckets (hash only)
```

What you get back

In list context, a list of every key currently in `%HASH`, or every valid index of `@ARRAY`. In scalar context, the count - without building the list - so `scalar keys %h` is the cheap way to ask “how many entries?”

Hash keys are returned in an **arbitrary** order. The order is stable for a given unmodified hash - `keys`, `values`, and `each` all walk it in the same sequence - but it is not the insertion order, not sorted, and deliberately randomised per hash to defeat algorithmic complexity attacks (see `perlsec`). Any insertion or deletion may reshuffle. The one documented exception: deleting the key most recently returned by `each` or `keys` does not perturb the remaining order.

Array indices, by contrast, come back lowest-first: `(0, 1, 2, ..., $#ARRAY)`.

The returned hash keys are **copies**. Modifying an element of the returned list does not modify the hash. Compare `values`, which returns aliases.

Global state it touches

- **Iterator of %HASH / @ARRAY:** keys resets the internal iterator shared with `each` and `values`. This is the idiomatic reset - call `keys %h` in void context specifically to discard any partial `each` walk:

```
keys %h;                                # reset iterator, no list built
while (my ($k, $v) = each %h) { ... }
```

Without this reset, a prior `each` that exited early leaves the iterator mid-hash and the next `each` call resumes there.

Examples

Iterate all keys in an arbitrary order:

```
my %h = (apple => 1, pear => 2, plum => 3);
for my $k (keys %h) {
    print "$k => ${h{$k}}\n";
}
```

Iterate in a deterministic order by running the keys through `sort`:

```
for my $k (sort keys %h) {
    print "$k => ${h{$k}}\n";    # apple, pear, plum
}
```

Iterate in **insertion order** by tracking keys separately - hashes themselves do not remember insertion order:

```
my %h;
my @order;
for my $pair (["apple", 1], ["pear", 2], ["plum", 3]) {
    my ($k, $v) = @$pair;
    push @order, $k unless exists $h{$k};
    $h{$k} = $v;
}
for my $k (@order) {
    print "$k => ${h{$k}}\n";    # apple, pear, plum (insertion)
}
```

Count entries without materialising the list:

```
my $n = keys %h;                    # scalar context, O(1)
print "hash has $n entries\n";
```

Preallocate buckets when you know the hash will grow large. The assignment is a size hint, rounded up to the next power of two:

```
my %big;
keys(%big) = 10_000;                # allocates 16384 buckets
$big{$_} = 1 for 1 .. 10_000;      # no rehash churn during fill
```

Walk an array by index - useful when you need both the index and the element together:

```
my @a = ("zero", "one", "two");
for my $i (keys @a) {
    print "$i: $a[$i]\n";
}
```

Sort a hash by value, descending numeric:

```
for my $k (sort { $h{$b} <=> $h{$a} } keys %h) {
    printf "%4d %s\n", $h{$k}, $k;
}
```

Edge cases

- **Empty hash or array:** returns the empty list in list context, `0` in scalar context. No warning, no special case to code around.
- **require v5.12 for arrays:** `keys @array` is a syntax error on perls older than 5.12. Put `use v5.12;` at the top of files that use the array form if portability to ancient perls matters.
- **Hash order is not reproducible across runs:** Perl randomises hash seeding at startup. Never build a test, cache key, or file format that assumes a particular `keys %h` order. Sort if you need determinism.
- **Hash order is not stable across Perl versions:** the randomised order within a run is stable; the algorithm itself is not guaranteed between releases. Do not pickle keys output and expect it to round-trip on a different Perl.
- **Deleting during iteration:** safe only for the key most recently returned by `each` or `keys`. Deleting a different key may cause other keys to be visited twice or skipped.
- **Lvalue keys on arrays is a syntax error:** `keys(@a) = 100` is not valid. The bucket-preallocation form works on hashes only.
- **Lvalue keys cannot shrink:** `keys(%h) = 10` on a hash that already has 1024 buckets has no effect. `%h = ()` clears the contents but preserves the bucket count; `undef %h` frees the buckets.
- **Memory cost of the list form:** `keys %h` in list context materialises every key as a fresh SV. For a million-entry hash that is a million allocations plus the returned list. Iterate with `each` or `while (my ($k, $v) = each %h)` when you want streaming access without the memory spike.
- **Tied hashes:** dispatch through `FIRSTKEY` / `NEXTKEY`. Order and side effects are whatever the tie class provides. In particular, tied hashes may reshuffle on insertion in ways ordinary hashes do not, and a tied `keys` may be expensive if the backing store is remote.
- **Scalar context on a tied hash:** may or may not be $O(1)$. Ordinary hashes answer scalar `keys %h` in constant time; a tied class has to implement `SCALAR` explicitly to avoid a full walk.
- **Scalar keys on an array equals scalar @array:** both return the element count. Prefer `scalar @array` for clarity unless the surrounding code is already phrased

in keys/values/each terms.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `values` - companion list of the values in the same order as keys; returns aliases, not copies
- `each` - streaming (key, value) pair walk sharing the same iterator that keys resets
- `exists` - test whether a specific key is present without building the full key list
- `delete` - remove a key; safe mid-iteration only for the most recently returned key
- `sort` - impose a deterministic order on the arbitrary sequence keys returns
- `scalar` - force scalar context to get the count without materialising the key list

Processes

kill

Send a signal to a list of processes.

`kill` is Perl's front-end to the `kill(2)` system call. It delivers `SIGNAL` to each process identified by `LIST` and returns the number of arguments that were successfully signalled. It is also the standard way to ask the kernel “does this PID still exist and may I signal it?” without actually disturbing the target, via the signal-zero form.

Synopsis

```
kill SIGNAL, LIST
kill SIGNAL          # tainting side-effect only; returns 0
```

What you get back

The number of arguments in `LIST` that were successfully passed to `kill(2)`. This is **not** necessarily the number of processes that actually received the signal - when `SIGNAL` is negative, or when a PID identifies a process group, one argument can target many processes. On permission or “no such process” errors for an argument, that argument is not counted and `$!` is set to the corresponding `errno` (`EPERM`, `ESRCH`, `EINVAL`).

```
my $cnt = kill 'HUP', $child1, $child2;
kill 'KILL', @goners;
```

With an empty `LIST`, no signal is sent and the return value is `0`. The bare `kill SIGNAL` form is sometimes written purely to trigger taint checks on `SIGNAL`; see `perlsec`.

SIGNAL: name or number

SIGNAL may be either a signal name (a string) or a signal number. The string form is recommended for portability because the same signal can have different numbers on different operating systems.

A signal name may carry the SIG prefix or not - 'TERM' and 'SIGTERM' name the same signal.

```
kill 'TERM', $pid;      # portable
kill 'SIGTERM', $pid;   # also fine
kill 15, $pid;          # SIGTERM on Linux; may differ elsewhere
kill 9, $pid;           # SIGKILL on Linux
```

The list of signal names available on the current platform lives in `$Config{sig_name}`, provided by the Config module.

SIGNAL 0: liveness check

If SIGNAL is the number 0 or the string 'ZERO' (or 'SIGZERO'), no signal is actually delivered. `kill(2)` still performs its permission check, so the call tells you whether it would be *possible* to signal the process - that is, the process exists and is owned by the same user, or the caller is the super-user.

```
if (kill 0, $pid) {
    # process exists and we may signal it
}
elsif (!$?) {
    # no such process - it has exited and been reaped
}
elsif ($?) {
    # it exists but we are not allowed to signal it
}
```

This is the canonical idiom for checking that a child is still alive, even as a zombie, and has not changed its UID. It does **not** reap the child - use `waitpid` for that.

Negative SIGNAL: signal a process group

A negative signal name or number signals a whole process group instead of a single process. `kill '-KILL', $pgrp` and `kill -9, $pgrp` both send SIGKILL to every process in the group identified by \$pgrp.

```
kill '-TERM', $pgrp;    # polite shutdown of the whole group
kill -9, $pgrp;         # forced
```

Positive signals are almost always what you want. Reach for a negative signal only when the target is deliberately a process group leader (for example a child you placed in its own group with `setpgrp` so you can terminate the subtree at once).

PID 0, negative PIDs

The behaviour of `kill` when a *PID* is zero or negative depends on the operating system. On POSIX-conforming systems (Linux included):

- **PID 0** - signal the sender's own process group.
- **PID -1** - signal every process the caller has permission to signal (super-user: the whole system, minus a few special PIDs).
- **Any other negative PID** - behaves like a negative *SIGNAL* against the absolute value: signal the process group whose PGID equals `abs(PID)`.

If both *SIGNAL* and the *PID* are negative, the result is undefined; a future Perl may warn. Do not combine the two.

Examples

Politely ask a child to shut down, then escalate:

```
kill 'TERM', $child;
sleep 2;
kill 'KILL', $child if kill 0, $child;
waitpid $child, 0;
```

Reap nothing, just check whether a PID file still names a live process:

```
open my $fh, "<", "/run/daemon.pid" or die $!;
chomp(my $pid = <$fh>);
close $fh;

if (kill 0, $pid) {
    print "daemon alive (pid $pid)\n";
} else {
    print "stale pid file: $!\n";
    unlink "/run/daemon.pid";
}
```

Signal a whole process group created by a shell pipeline child:

```
my $pgrp = getpgrp($child);
kill '-TERM', $pgrp;
```

Install a handler and send yourself a signal:

```
local $SIG{USR1} = sub { warn "got USR1\n" };
kill 'USR1', $$;          # $$ is our own PID
```

Broadcast to several workers and find out how many were reachable:

```
my @workers = (1234, 1235, 9999); # 9999 has exited
my $hit = kill 'HUP', @workers;   # probably 2
warn "lost a worker: $!\n" if $hit < @workers;
```

Edge cases

- **Empty LIST:** `kill 'TERM'` sends nothing and returns `0`. The call still evaluates `SIGNAL`, which is the only reason to write it (taint propagation).
- **Unknown signal name:** raises an exception (Unrecognized signal name "FOO"). Validate against `$Config{sig_name}` if the name comes from user input.
- **\$! after partial success:** `kill` walks `LIST` left to right. `$!` reflects the **last** failing argument, not all of them. If you need per-PID errors, call `kill` once per PID.
- **Zombies still count as alive:** `kill 0`, `$zombie_pid` returns truthy until the child has been reaped with `wait` or `waitpid`. This is usually what you want for a liveness check; occasionally it surprises.
- **Signal delivery is asynchronous:** a successful kill only means the signal was *queued*. The handler in `%SIG` on the receiving process may run later, and Perl's safe-signal machinery defers delivery to the next opcode boundary.
- **Sending to yourself:** `kill SIGNAL, $$` works and is the usual way to test your own `%SIG` handlers.
- **No LIST, with taint:** under `-T`, writing `kill 'TERM'` propagates taintedness of `'TERM'` for the side-effect of triggering the check - the form has no other use.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `fork` - create the child process you are later going to signal
- `wait` - reap any terminated child, clearing zombies that `kill 0` would still report as alive
- `waitpid` - reap a specific child, optionally non-blocking (`WNOHANG`) so it pairs cleanly with `kill 0`
- `getpgrp/setpgrp` - inspect and change process-group membership, which is what negative-SIGNAL kill operates on
- `%SIG` - install handlers for the signals you send with `kill`
- `perlipc` - full treatment of signals, handlers, and safe-signal semantics

Control flow

last

Exit a loop immediately, skipping the rest of the body and the `continue` block.

`last` is Perl's equivalent of C's `break`. It transfers control out of the enclosing loop to the statement that follows it. The current iteration's remaining statements do not run, and neither does the loop's `continue` block - that is the key difference from `next`. With no argument it targets the innermost enclosing `while`, `until`, `for`, `foreach`, or bare block; with a `LABEL` it targets a named loop further out.

Synopsis

```

last
last LABEL
last EXPR          # LABEL computed at runtime, since 5.18

```

What you get back

Nothing. `last` performs flow control; it does not produce a value. It cannot be used to return a result from a block that normally yields one, such as `eval { ... }`, `sub { ... }`, or `do { ... }`. In those contexts either the construct produces no value or the program dies with Can't "last" outside a loop block - see *Edge cases* below.

Targeting a loop by label

Without a label, `last` refers to the innermost enclosing loop. That is usually what you want:

```

while (my $line = <$fh>) {
    last if $line =~ /^__END__$/;
    process($line);
}

```

With a label you can break out of an outer loop from within a nested one. Labels are conventionally all-uppercase:

```

OUTER: for my $row (@grid) {
    for my $cell (@$row) {
        last OUTER if $cell eq 'STOP';
    }
}
# control resumes here

```

`last EXPR` (Perl 5.18+) computes the label at runtime. `EXPR` must evaluate to a string naming an enclosing label:

```

my $target = 'OUTER';
last $target;

```

Interaction with `continue` blocks

A loop's `continue` block normally runs at the end of every iteration, including the last. `last` **skips** it - that is its defining difference from `next`:

```

my $i = 0;
while ($i < 10) {
    last if $i == 3;
    print "body $i\n";
} continue {
    $i++;
    # runs after every iteration EXCEPT

```

(continues on next page)

(continued from previous page)

```
                                # the one terminated by last
}
# prints body 0, body 1, body 2; then exits the loop.
# $i stays at 3 because continue did not fire the final time.
```

If you want `continue` to run, use `next` with a guard instead, or advance the counter before the `last`.

Behaviour in nested loops

`last` only leaves one loop at a time - the innermost one, unless labelled. In this example the inner `last` exits the `for` but the `while` keeps running:

```
while (1) {
    for my $x (1 .. 5) {
        last if $x == 3;
    }
    # control lands here every iteration of the while
    last;                # needed to exit the while as well
}
```

To unwind several levels at once, label the outer loop and target it:

```
SEARCH: for my $file (@files) {
    open my $fh, '<', $file or next;
    while (my $line = <$fh>) {
        if ($line =~ /$pattern/) {
            $found = $file;
            last SEARCH; # exits both the while and the for
        }
    }
}
```

Bare blocks count as loops

A bare `{ ... }` block is, for flow-control purposes, a loop that executes exactly once. `last` can exit such a block early, which is the idiomatic way to write “try a sequence of checks, bail out on the first match”:

```
{
    last if $cached;
    recompute();
    store_in_cache();
}
# control continues here
```

This is the only `last` form that does not involve an iterating construct.

Examples

Stop reading a file on a blank line - the classic header-terminator idiom:

```
LINE: while (<$fh>) {
    last LINE if /^$/;
    push @header, $_;
}
```

Find-and-report across a two-dimensional structure:

```
my $hit;
ROW: for my $r (0 .. $#grid) {
    for my $c (0 .. ${ $grid[$r] }) {
        if ($grid[$r][$c] eq $needle) {
            $hit = [$r, $c];
            last ROW;
        }
    }
}
```

Early exit from a bare block used as a try-once sequence:

```
my $result;
{
    $result = $cache{$key}, last if exists $cache{$key};
    $result = compute($key);
    $cache{$key} = $result;
}
```

Computed label (Perl 5.18+). Rarely needed, but useful when the target loop is chosen dynamically:

```
my $level = $deep ? 'OUTER' : 'INNER';
OUTER: for (...) {
    INNER: for (...) {
        last $level if $done;
    }
}
```

Edge cases

- **Illegal in `grep` and `map`.** Both expect their block to produce a value for every input element; `last` precludes that. The parser accepts the code (the block is not lexically a loop it can see through), but executing it dies with Can't "last" outside a loop block. Filter with a conditional inside the block instead, or switch to an explicit `for` loop.
- **`do { ... }` is not a loop.** `last` inside `do { ... }` does not terminate the `do`; it unwinds to the nearest enclosing real loop. If there is none, the program dies with Can't "last" outside a loop block. For early exit from a `do` used as an expression, see the workarounds in *perlsyn* under “Statement Modifiers” - typically

a bare block plus `last`, or a one-shot for loop.

- **sub { ... } is not a loop.** `last` inside a sub body does **not** return from the sub; it unwinds past the sub to the enclosing loop in the caller. Use `return` to leave a sub.
- **last inside eval does not escape the eval.** `eval { ... }` is not itself a loop, so `last` looks outward past the `eval` for the nearest enclosing loop and unwinds to it. The `eval` is exited as a side effect of that unwind, but `$@` is left empty - no exception was raised. If the `eval` has no enclosing loop above it, the program dies with `Can't "last" outside a loop block`. To leave an `eval` normally, fall off the end or use `die` (caught by the `eval` itself) or `return` (if the `eval` is inside a sub and you want to leave the sub).
- **Precedence.** Unlike most named operators, `last` has the same precedence as assignment and is exempt from the looks-like-a-function rule. So `last ("foo") . "bar"` passes `"foo" . "bar"` as the label, not just `"foo"`. Use a space, or omit the parentheses, to avoid surprises:

```
last LABEL;  
last "LABEL";
```

- **Unknown label.** `last NOPE` where no enclosing loop is tagged `NOPE` dies with `Label not found for "last NOPE"`. This is a runtime error, not a compile-time one - the label is resolved against the dynamic call stack.
- **Loop modifier form has no block to exit.** `EXPR if COND` and similar statement-modifier forms are not loops; `last` inside the expression unwinds past them.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `next` - skip to the next iteration, running the `continue` block first; the counterpart to `last` that keeps the loop alive
- `redo` - restart the current iteration without re-testing the condition and without running `continue`
- `return` - leave a subroutine with a value; the right tool when you reach for `last` inside `sub { ... }`
- `continue` - the loop-trailer block that `last` skips and `next` honours; read its page for the three-way illustration of `last` / `next` / `redo`

SCALARS and strings

lc

Return a lowercased copy of a string.

lc takes a string, converts every cased character to its lowercase equivalent, and returns the result. The original is never modified. If the argument is omitted, lc operates on `$_`. The exact set of characters that change depends on the string's encoding and the pragmas in scope at the call site - ASCII-only by default, full Unicode under `use feature 'unicode_strings'` or when the input is already a character string with the UTF-8 flag.

Synopsis

```
lc EXPR
lc
```

What you get back

A new string. The return value is always a fresh scalar - modifying it does not affect the argument, and modifying the argument afterward does not affect the returned string. Length in characters is preserved: lc never adds or removes characters, even in the rare Unicode cases where upper- and lower-case forms have different lengths in other directions (see `fc` for folding, which can change length).

```
my $str = lc("Perl is GREAT"); # "perl is great"
```

Global state it touches

- Reads `$_` when called without an argument.
- Observes `use locale` for `LC_CTYPE` - with locale in effect the current locale's lowercasing table applies to code points below 256.
- Observes `use bytes` - under `use bytes` only A-Z change, to a-z.
- Observes `use feature 'unicode_strings'` - forces Unicode rules regardless of the UTF-8 flag on the input.

Which casing rules apply

Perl picks one of four rulesets, in priority order. The first one whose condition holds wins:

1. **use bytes in effect** - ASCII rules. Only A-Z map to a-z; every other byte is left alone, including bytes in the 128-255 range that would otherwise be Latin-1 letters.
2. **use locale for LC_CTYPE in effect** - the current locale's tables apply to code points below 256; code points 256 and above (only reachable when the string already carries the UTF-8 flag) use Unicode rules. From Perl 5.20 onward, a UTF-8 locale uses full Unicode rules throughout.
3. **The argument has the UTF-8 flag set** - Unicode rules apply to every character.
4. **use feature 'unicode_strings' or use locale ':not_characters' in effect** - Unicode rules apply to every character, regardless of the UTF-8 flag.

5. **Otherwise** - ASCII rules. Characters outside A-Z are returned unchanged, including Latin-1 uppercase letters like À-Þ.

The upshot: if you want predictable Unicode behaviour on every string regardless of how it was constructed, enable use feature 'unicode_strings' (or use v5.12 and above, which turns it on for you).

Examples

Basic ASCII lowercasing:

```
my $s = lc("Hello, World!");           # "hello, world!"
```

Default to \$_:

```
for ("FOO", "Bar", "BAZ") {
    print lc, "\n";                     # foo / bar / baz
}
```

Unicode characters only lowercase when the rules allow it:

```
use feature 'unicode_strings';
my $s = lc("ÄÖÜ");                     # "äöü"
```

Without unicode_strings and without the UTF-8 flag on the input, non-ASCII characters pass through unchanged:

```
my $bytes = "\xC4\xD6\xDC";           # Ä Ö Ü as Latin-1 bytes
my $lower = lc $bytes;                 # unchanged: "\xC4\xD6\xDC"
```

lc is what backs the \L... \E escape inside double-quoted strings:

```
my $str = "Perl is \LGREAT\E";        # "Perl is great"
```

Case-insensitive compare by lowercasing both sides:

```
sub ieq { lc($_[0]) eq lc($_[1]) }
ieq("Perl", "PERL");                  # true
```

For correct case-insensitive comparison across Unicode, prefer fc (the Unicode folding function) over lc - see *See also*.

Edge cases

- **undef argument** raises an uninitialized warning under use warnings and returns the empty string.
- **Empty string** returns the empty string.
- **Numbers are stringified first**: lc(42) returns "42".
- **LATIN CAPITAL LETTER SHARP S** (U+1E9E) lowercases to U+00DF (ß) under Unicode rules, but only if the result can be represented without crossing the 255/256 boundary in a way the current ruleset accepts. Under use locale on a non-UTF-8

locale, Perl leaves the character unchanged rather than guessing - and from Perl 5.22 onward this raises a locale warning.

- **Locale and the UTF-8 flag interact:** under `use locale` on a non-UTF-8 locale, characters below 256 follow the locale while characters 256 and above follow Unicode. A string containing both ranges will get two different casing tables applied to it.
- **Tied variables** have their `FETCH` called once. `lc` does not modify the tied value; it returns a plain (non-tied) scalar.
- **`lc` is a unary named operator**, not a list operator. `lc $a, $b` parses as `(lc $a), $b` - only `$a` is lowercased, and the comma operator discards the result. Use parentheses or `map` when you mean to lowercase multiple strings:

```
my @lower = map { lc } @words;
```

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `uc` - the uppercasing counterpart, with the same pragma and locale rules
- `lcfirst` - lowercase only the first character; handy for de-capitalising sentences or identifiers
- `fc` - Unicode casefold, the correct choice for case-insensitive comparison when input may contain non-ASCII
- `$_` - the default subject `lc` reads when called with no argument
- `use locale` - controls whether locale tables or ASCII/Unicode rules apply to characters below 256
- `use feature 'unicode_strings'` - forces full Unicode casing regardless of the UTF-8 flag on the input

SCALARs and strings

lcfirst

Return a copy of a string with its first character lowercased.

`lcfirst` touches **only** the first character; the rest of the string is copied verbatim. The input is not modified - `lcfirst` returns a new string. It is the internal function implementing the `\l` escape in double-quoted strings, so `"\l$name"` and `lcfirst($name)` produce the same result.

If `EXPR` is omitted, `lcfirst` operates on `$_`.

Synopsis

```
lcfirst(EXPR)
lcfirst
```

What you get back

A new string equal to `EXPR` with its first character replaced by the lowercase form that character's case-folding rules produce. The rest of the string is untouched. If `EXPR` is the empty string, the empty string is returned. If `EXPR` is `undef`, the result is the empty string and an uninitialized warning is emitted under `use warnings`.

`lcfirst` never mutates its argument. To lowercase the first character in place, assign the result back:

```
$name = lcfirst $name;
```

For the all-characters variant, see `lc`.

Global state it touches

- `$_` - read when `EXPR` is omitted.
- `LC_CTYPE` locale - consulted when `use locale` is in effect (see *Pragmas and case rules* below).

Pragmas and case rules

`lcfirst` behaves the same way under various pragmas as `lc` does. The active rule set for the first character depends on which pragmas are in scope at the call site:

- **use bytes** - ASCII rules only. Only the bytes A-Z are folded to a-z; every other byte is returned unchanged.
- **use locale** (for `LC_CTYPE`) - the current locale decides the fold for code points below 256. Code points 256 and above fold by Unicode rules when the string has the UTF-8 flag set. A UTF-8 locale engages full Unicode rules from Perl 5.20 onward.
- **UTF-8-flagged string** - Unicode case-folding rules apply.
- **use feature 'unicode_strings'** or **use locale ':not_characters'**
 - Unicode rules apply regardless of the UTF-8 flag.
- **Otherwise** - ASCII rules. Characters outside ASCII are returned unchanged.

This means the same `lcfirst("É...")` can return `"é..."` under `use utf8` or `use feature 'unicode_strings'`, and `"É..."` (unchanged) under pure-ASCII defaults. When portability across pragmas matters, state the rule set explicitly:

```
use feature 'unicode_strings';
my $s = lcfirst $input;      # Unicode rules regardless of UTF-8
                              ↪flag
```

Examples

Basic call, lowercasing the first letter of a word:

```
my $s = lcfirst "Hello, World";    # "hello, World"
```

Title-case normalisation for a list of names, where the caller wants the first letter lowercased for a URL slug:

```
my @slugs = map { lcfirst } @CamelCaseNames;
```

Default-argument form reads from \$_:

```
for ("Apple", "Banana", "Cherry") {
    print lcfirst, "\n";           # "apple", "banana", "cherry"
}
```

Only the **first** character changes - later capitals are preserved. This matters for camel-Case identifiers derived from PascalCase:

```
my $method = lcfirst "GetUserName";    # "getUserName"
```

Unicode rules must be requested explicitly, either via a UTF-8-flagged input string or via the `unicode_strings` feature:

```
use feature 'unicode_strings';
my $s = lcfirst "Éclair";             # "éclair"
```

```
my $s = lcfirst "Éclair";             # without unicode_strings and no
                                     # UTF-8 flag: "Éclair" unchanged
```

The `\l` escape in a double-quoted string is exactly `lcfirst` on the next character:

```
my $name = "Perl";
print "\l$name\n";                   # "perl\n"
```

Edge cases

- **Empty string:** `lcfirst("")` returns `""`. No warning.
- **undef input:** `lcfirst(undef)` returns `""`. Emits an uninitialized warning under `use warnings`.
- **First character already lowercase:** returned unchanged; no special case, no shortcut the user needs to know about.
- **First character has no lowercase mapping** (digit, punctuation, symbol, most CJK): returned unchanged.
- **Multi-codepoint case folds:** some Unicode characters lowercase to more than one code point (e.g. Turkish dotted/dotless I pairs in locale-tailored folds). `lcfirst` follows Unicode's *simple* lowercase mapping, which is always single-codepoint - it does not perform full case folding. Use `lc` combined with normalisation, or an explicit `Unicode::CaseFold`, when full-fold semantics matter.

- **Locale surprises around 255/256:** under use `locale` with a non-UTF-8 locale, a first character whose lowercase would cross the 255/256 boundary is returned unchanged, and from Perl 5.22 onward a locale warning is emitted.
- **Not list-aware:** `lcfirst` is a scalar operator. `lcfirst @list` takes `scalar(@list)` - the element count - and lowercases its first character, almost never what the caller meant. Use `map { lcfirst } @list` for per-element lowercasing.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `lc` - lowercase every character, not only the first; same pragma-sensitivity rules
- `ucfirst` - uppercase the first character; the exact inverse operation
- `uc` - uppercase every character
- `fc` - Unicode case-folding suitable for caseless comparison, not display
- `$_` - default argument when `EXPR` is omitted

SCALARs and strings

length

Return the number of characters in a string.

`length` stringifies its argument and returns how many **characters** that string contains. “Characters” means logical Perl characters, not bytes: a string holding ten codepoints has length 10 regardless of how many bytes those codepoints occupy when encoded as UTF-8 on disk or on the wire.

Synopsis

```
length EXPR
length
```

What you get back

A non-negative integer - the count of characters in the stringified value of `EXPR`. If `EXPR` is omitted, `length` operates on `$_`. If `EXPR` is `undef`, the result is `undef`, not `0`; this is the one case where `length` does not return a number, and it is deliberate - it lets you distinguish an undefined scalar from an empty string without a separate `defined` check:

```
length(undef)      # undef
length("")          # 0
length("abc")       # 3
```


Characters, not bytes

On a normal Perl string, length counts characters. The byte count of the same string encoded as UTF-8 is generally different and is what external systems (file sizes, network payloads, Content-Length headers) want. Get it by encoding first:

```
use Encode;
my $s      = "na\x{EF}ve";           # five characters
my $chars  = length $s;              # 5
my $bytes  = length(encode('UTF-8', $s)); # 6
```

See [Encode](#) and [perlunicode](#) for the full story on how Perl strings carry an internal character-vs-byte flag and when that flag matters.

Examples

Basic count:

```
length "hello"      # 5
length ""           # 0
```

Default argument is \$_:

```
for ("a", "bb", "ccc") {
    print length, "\n";           # 1, 2, 3
}
```

Distinguishing undefined from empty:

```
my $x;
my $y = "";
print defined(length $x) ? "set" : "undef", "\n"; # undef
print defined(length $y) ? "set" : "undef", "\n"; # set
```

A Unicode string - character count vs byte count:

```
my $s = "caf\x{E9}";           # four characters, one of them
    ↪ non-ASCII
length $s;                     # 4
length encode('UTF-8', $s);    # 5
```

Numbers are stringified before counting, which is occasionally useful and occasionally surprising:

```
length 12345                   # 5
length 3.14                   # 4  ("3.14")
length 1e6                     # 7  ("1000000")
```

Edge cases

- **undef in, undef out.** `length(undef)` returns `undef`, not `0`. Under `use warnings` reading an undefined scalar through `length $x` where `$x` is `undef` does **not** emit an uninitialized warning, unlike most other operators.
- **Whole array or hash:** `length` does not count elements. It stringifies its argument, so `length @arr` first evaluates `@arr` in scalar context (its element count) and then counts the digits of that number. For the actual size use `scalar @arr` or `scalar keys %hash`:

```
my @arr = (1) x 100;
length @arr;           # 3  (digits of "100")
scalar @arr;           # 100
```

- **Tied or magical scalars:** `length` triggers `FETCH`, so the counted value is whatever the tie layer returns at that moment.
- **Overloaded objects:** `length $obj` invokes the `"` stringification overload (or falls back to the default reference form) and counts characters in the resulting string.
- **Wide characters from byte input:** a string read from a handle without an encoding layer is a byte string. `length` counts bytes then - because each byte is one character to Perl. Decode first if you want character counts:

```
open my $fh, "<:raw", $path or die $!;
my $bytes = do { local $/; <$fh> };
my $text  = decode('UTF-8', $bytes);
length $bytes;           # byte count
length $text;           # character count
```

- **Locale and pragmas:** unlike `lc` / `uc` / `fc`, `length` does not vary by `use locale`. A character is a character regardless.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `scalar` - force scalar context; the correct way to get an element count from an array (`scalar @arr`) instead of the length of its stringified count
- `substr` - extract a substring by character offset; pairs naturally with `length` for slicing work
- `index` - find the position of a substring; returns a character offset measured the same way `length` measures
- `sprintf` - `%s` field widths are character counts, same notion as `length`
- `reverse` - in scalar context reverses a string character-by-character; same character model as `length`
- `pos` - regex-match position, also in characters

Filehandles, files, directories

link

Create a hard link from NEWFILE to the existing OLDFILE.

link adds a second directory entry that refers to the same underlying file as OLDFILE. After a successful call, the file has two names: OLDFILE and NEWFILE. Both names are equal citizens - the file itself is not deleted until every name is removed and no process holds the file open. The call is a thin wrapper around the `link(2)` system call and inherits its constraints: both paths must resolve on the **same filesystem**, and on POSIX systems only the superuser may hard- link a directory (most Linux kernels refuse even for root).

Synopsis

```
link OLDFILE, NEWFILE
```

What you get back

1 on success, 0 on failure. On failure, `$!` is set to the system error - typical values are EXDEV (cross-device link), EPERM (target is a directory, or filesystem forbids hard links), EEXIST (NEWFILE already exists), EACCES (insufficient permission), and ENOENT (OLDFILE missing).

```
link $src, $dst
or die "link $src -> $dst failed: $!";
```

Global state it touches

- `$!` - set on failure to the system error code.

Examples

Create a second name for an existing file:

```
link "report.txt", "report.bak"
or die "link failed: $!";
```

Use `stat` to observe the link count going up. The third element of the `stat` list is the number of hard links pointing at the inode:

```
link "data", "data.alt" or die $!;
my $nlink = (stat "data")[3];
print "data has $nlink names\n";    # "data has 2 names"
```

Atomically replace a file by building the new version under a temporary name, then swapping with `link + rename`:

```
open my $fh, ">", "config.new" or die $!;
print $fh $new_contents;
close $fh;
rename "config.new", "config" or die "rename: $!";
```

(For the genuinely atomic-under-crash variant you want `link + unlink + rename`; see the commentary in `perlport`.)

Detect a cross-device failure and fall back to copying:

```
use Errno qw(EXDEV);
unless (link $src, $dst) {
    if ($! == EXDEV) {
        # /tmp and $HOME are typically different filesystems
        copy_file($src, $dst);
    } else {
        die "link $src -> $dst: $!";
    }
}
```

Edge cases

- **Directories:** `link` on a directory fails with `EPERM` on effectively every modern system. POSIX forbids it for non-root, and Linux forbids it even for root. Use `symlink` when you want a second name for a directory.
- **Cross-filesystem:** the kernel returns `EXDEV` if `OLDFILE` and `NEWFILE` resolve to different filesystems. Hard links cannot span mount points because they share an inode number, which is only meaningful within one filesystem.
- **Existing target:** if `NEWFILE` already exists, the call fails with `EEXIST`. `link` never overwrites. Remove the target first with `unlink` if replacement is intended.
- **Symbolic link as `OLDFILE`:** the link is made to the symlink's **target**, not to the symlink itself, on most systems. If you need to link the symlink, dereference it explicitly or use a platform that exposes `linkat` with `AT_SYMLINK_FOLLOW` cleared (not available through Perl core).
- **Relative paths** are resolved against the current working directory at the moment of the call, for both operands.
- **Permissions:** `link` does not require write permission on `OLDFILE` - only search permission on its directory and write permission on the directory containing `NEWFILE`.
- **Same path both sides:** `link $f, $f` fails with `EEXIST`.
- **Unlinking:** removing `OLDFILE` with `unlink` after a successful `link` leaves the file fully reachable through `NEWFILE`. The data is freed only when the last name is unlinked and no open file descriptor refers to it.
- **Portability:** not every filesystem supports hard links. FAT, exFAT, and many network filesystems either refuse or silently behave as copies. See `perlport` for the platform matrix.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `symlink` - create a symbolic link instead of a hard link; works across filesystems and on directories
- `unlink` - remove a directory entry; the file itself survives as long as another link or open descriptor remains
- `rename` - move or rename a file within one filesystem; atomic where `link` + `unlink` is not
- `stat` - inspect the link count (field 3) to see how many names refer to the same inode
- `$!` - the system error set when `link` returns 0

Sockets

listen

Mark a socket as passive so it can accept incoming connections.

`listen` is the third step in the classic server setup - after `socket` creates the endpoint and `bind` fixes it to a local address, `listen` switches it into the listening state and sizes the kernel's backlog queue of pending connections. Incoming connections are then drained one at a time with `accept`. `QUEUE_SIZE` is the maximum number of connections the kernel will hold for you before it starts refusing new ones.

Synopsis

```
listen SOCKET, QUEUE_SIZE
```

What you get back

True on success, false on failure (with `$!` set to the `errno` from the underlying `listen(2)`). Always check the return value - a server that silently fails to enter the listening state will accept no connections, and the cause (a stale socket file, a port collision, an unbound descriptor) is only visible through `$!`.

```
listen $srv, SOMAXCONN
or die "listen on $path: $!";
```

Global state it touches

`listen` reads no special variables and writes only `$!` on failure. The call is a thin wrapper over the kernel's `listen(2)`; effects are on the socket descriptor, not on Perl state.

Examples

A minimal TCP server. `socket`, `bind`, `listen`, `accept` in order - miss any step and `accept` fails:

```
use Socket;

socket(my $srv, PF_INET, SOCK_STREAM, getprotobyname("tcp"))
    or die "socket: $!";
setsockopt($srv, SOL_SOCKET, SO_REUSEADDR, 1)
    or die "setsockopt: $!";
bind($srv, sockaddr_in(8080, INADDR_ANY))
    or die "bind: $!";
listen($srv, SOMAXCONN)
    or die "listen: $!";

while (accept(my $cli, $srv)) {
    # serve $cli ...
    close $cli;
}
```

A Unix-domain server. Same sequence, different address family:

```
use Socket;

my $path = "/tmp/myservice.sock";
unlink $path;                                # stale socket file would
↳ cause EADDRINUSE
socket(my $srv, PF_UNIX, SOCK_STREAM, 0) or die "socket: $!";
bind($srv, sockaddr_un($path))           or die "bind: $!";
listen($srv, 16)                         or die "listen: $!";
```

Using the POSIX-defined cap `SOMAXCONN` rather than hard-coding a number. The kernel silently clamps larger values to its own limit anyway; `SOMAXCONN` documents intent:

```
use Socket qw(SOMAXCONN);
listen($srv, SOMAXCONN) or die "listen: $!";
```

A small backlog for a service that processes connections faster than they arrive:

```
listen($srv, 5) or die "listen: $!";
```

Re-listening on an already-listening socket. A second call with a different `QUEUESIZE` is legal on Linux and resizes the backlog in place:

```
listen($srv, 128) or die "listen: $!";
# ... later, under load:
listen($srv, 1024) or die "listen: $!"; # grow the queue
```

Edge cases

- **Call order matters.** Calling `listen` on a socket that has not been `bind`-ed fails with `EDESTADDRREQ` (or on some kernels an autobound ephemeral port - do not rely on that). Calling `accept` before `listen` fails with `EINVAL`.
- **Only for connection-oriented sockets.** `listen` is meaningful on `SOCK_STREAM` and `SOCK_SEQPACKET`. Calling it on a `SOCK_DGRAM` socket fails with `EOPNOTSUPP`; UDP has no notion of pending connections.
- **QUEUE_SIZE is advisory.** The kernel clamps it to `/proc/sys/net/core/somaxconn` (Linux) regardless of what you ask for. A value of `0` is accepted but means “rely on kernel-default backlog”; it does not disable listening.
- **Negative values are silently treated as 0** by the kernel. Perl does not diagnose this - check your input before passing it to `listen`.
- **EADDRINUSE at listen vs at bind.** Port conflicts for TCP/UDP surface at `bind`; `listen` rarely fails with this. For Unix-domain sockets a leftover socket file from a previous run causes `bind` to fail with `EADDRINUSE` - remove it with `unlink` before binding.
- **Stale connections in the queue.** When `accept` runs, the returned connection may already be closed by the peer (the client gave up while queued). Handle that like any other I/O error from the new socket, not as a `listen` problem.
- **After fork.** A listening socket inherited by a child remains listening; both parent and child can `accept` from the same queue. This is the standard pre-fork server pattern - do not call `listen` again in the child.
- **SIGPIPE is unrelated.** `listen` itself never raises `SIGPIPE`; that signal arises on writes to a closed peer after `accept`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `socket` - creates the endpoint `listen` will mark as passive; always the first call in the sequence
- `bind` - assigns a local address to the socket; must run before `listen` or the call fails
- `accept` - pulls the next pending connection off the backlog queue that `listen` established
- `setsockopt` - set `SO_REUSEADDR` before `bind` to avoid `TIME_WAIT` collisions when restarting a server
- `shutdown` - half-close an accepted connection; not used on the listening socket itself
- `$!` - holds the `errno` after a failed `listen`; the only way to tell `EADDRINUSE` from `EACCES` from `EOPNOTSUPP`

Scoping

local

Save the current value of a package variable and restore it when the enclosing scope exits.

`local` is the dynamic-scope operator. It does **not** declare a new variable - it reaches into an existing package (global) variable, stashes the current value on a hidden save stack, and arranges for the original value to be put back when control leaves the current block, `eval`, or `do FILE`. Called subroutines see the temporary value. For most “I want a variable local to this block” needs, the right tool is `my`; reach for `local` specifically when the variable must be a package global - the punctuation variables, filehandles, or entries in the symbol table - and you want that global to have a different value for the duration of a block and every subroutine it calls.

Synopsis

```
local EXPR
local $var
local $var = EXPR
local (@list) = LIST
local $hash{key}
local $array[i]
local *GLOB
```

What you get back

The value of the localized expression (the new value, after any assignment in the same statement). The save-stack bookkeeping is a side effect; the return value matters only when `local` appears inside a larger expression:

```
our $x = 2;
foo($x, local $x = $x + 1, $x); # foo() receives (2, 3, 3)
```

Dynamic scope, not lexical scope

`local` and `my` are different species of scoping:

- `my` creates a **new, lexically scoped** variable. Only code textually inside the enclosing block can see it. Called subroutines cannot.
- `local` does **not** create anything. It temporarily overrides the value of an **existing package variable**. Every piece of code - this block and every subroutine it calls, directly or transitively - sees the new value until the scope ends.

That is why `local` is the right tool for punctuation variables like `$/`, `$\`, `$,`, `$_`, and `$@`: those are package globals that the Perl runtime (and anything called from your code) reads at well-known names. A `my` shadowing would not affect them.

```
sub slurp {
    my ($path) = @_;
    open my $fh, "<", $path or die "open $path: $!";
```

(continues on next page)

(continued from previous page)

```

local $/;                # undef $/ for this sub only
return <$fh>;            # read to EOF in one go
}

```

The `local $/` line above is the single most common local idiom in real code: temporarily undefining the input record separator so that `readline (<$fh>)` returns the whole file. When `slurp` returns, `$/` is restored to whatever it was before - even if the caller had also changed it.

What you can and cannot localize

You can localize:

- **Package scalars, arrays, hashes** - `local $Pkg::var`, `local @Pkg::arr`, `local %Pkg::hash`. Bareword `local $var` targets the variable in the current package.
- **Already-declared `our` variables** - `our` creates a lexical alias to a package variable, so `local` on it localizes the underlying global.
- **Single elements and slices of package arrays/hashes** - `local $hash{key}`, `local $array[3]`, `local @hash{qw(a b)}`, `local @array[0..2]`.
- **Globs** - `local *name` creates a fresh symbol-table entry so that `$name`, `@name`, `%name`, and the name filehandle are all dynamically reset together.
- **Conditional lvalues** - `local ($cond ? $v1 : $v2)` when the chosen branch is itself localizable.

You **cannot** localize:

- A `my` variable. `my` variables are lexicals living in a pad, not in the symbol table; they have no package name for `local` to save under. Trying it is a compile-time error: Can't localize lexical variable `$x`.
- A `state` variable. Same reason.
- Read-only magic globals. `local $1 = 2` fails with Modification of a read-only value attempted - the capture variables cannot be assigned to. (The one exception is `$_`: `local $_` has, since 5.14, explicitly stripped magic so it can be reused safely in a subroutine.)

Localizing composite-type elements

`local $hash{key}` and `local $array[i]` save the **named slot**, not the value that happens to be there. When the scope ends, the original value is restored to that same slot - even if the element was deleted or the array was shortened in the meantime. A deleted hash key springs back; a popped array element reappears, padding the array with `undef` if necessary.

```

our %hash = (a => "is");
{
    local $hash{a} = "drill";
    delete $hash{a};          # gone for now...
}

```

(continues on next page)

(continued from previous page)

```
}  
print $hash{a}, "\n";           # ...but restored to "is"
```

Examples

Basic: give a global a temporary value for one block. Subroutines called from inside the block see the new value.

```
our $verbose = 0;  
sub log_msg { print "[v=$verbose] $_[0]\n" }  
  
{  
    local $verbose = 1;  
    log_msg("inside");           # prints "[v=1] inside"  
}  
log_msg("outside");             # prints "[v=0] outside"
```

Slurp-a-file idiom with \$/:

```
my $content = do {  
    open my $fh, "<", $path or die "open $path: $!";  
    local $/;                     # list context for <> now reads_  
    ↪ all  
    <$fh>;  
};
```

Temporarily trap \$@ so an inner `eval` does not clobber the caller's pending error:

```
sub try_cleanup {  
    local $@;                     # caller's $@ preserved  
    eval { risky_cleanup() };  
    # even if eval failed, caller's $@ is untouched  
}
```

Python-style print defaults, scoped to one block:

```
{  
    local $, = " ";  
    local $\ = "\n";  
    print 1, 2, 3;                # "1 2 3\n"  
}  
print 1, 2, 3;                  # "123" - defaults restored
```

List form needs parentheses. The parentheses also give the right-hand side list context - exactly like `my`:

```
our (@wid, %get);  
local (@wid, %get) = (@defaults, %overrides);
```

Localize a single hash element - useful for threading overrides through a call without touching unrelated keys:

```
our %config = (timeout => 30, retries => 3);
{
    local $config{timeout} = 5;
    do_probe();                # sees timeout => 5
}
# %config back to (timeout => 30, retries => 3)
```

Glob localization - whole symbol-table entry for LOG is replaced:

```
{
    local *LOG;
    open LOG, ">", "/tmp/trace.$$" or die $!;
    run_traced();              # sees our private LOG filehandle
}
# original LOG (if any) is back
```

Edge cases

- **Runtime operator, not a declaration.** `local` executes each time control reaches it. Inside a loop it saves and restores on every iteration - measurable overhead. Lift it out of the loop when the intent is “for the whole loop”:

```
{ local $/ = "\n\n";                # once, not per iteration
  while (<$fh>) { ... }
}
```

- **Assignment is evaluated before the save.** The right-hand side runs in the enclosing dynamic scope, so `local $x = $x + 1` reads the old `$x`, then saves, then assigns the new value.
- **Scope boundary is dynamic, not lexical.** The save is undone when the **current block exits at runtime**, not when the source block ends textually. Exiting via `return`, `die`, `last`, `next`, or a non-local `goto` still triggers the restore.
- **local on a my is a compile-time error.** The parser rejects `my $x; local $x;` with `Can't localize lexical variable $x. Use our (or a fully qualified name) if you genuinely need dynamic scoping on that name.`
- **Tied arrays and hashes** do not currently behave the way the POD describes - localizing the whole aggregate is buggy upstream. Localizing individual elements of a tied aggregate is safe. This caveat is inherited from upstream Perl.
- **Read-only magicals cannot be assigned.** `local $1 = "x"` dies at runtime. You can `local $_` (magic is stripped since 5.14) but not `local $1`, `local $&`, etc.
- **Negative array indices** in `local $array[-1]` are documented as “particularly surprising” upstream; treat the behaviour as unstable and avoid.
- **delete local** extends `local` to the removal of an element: `delete local $hash{key}` deletes the entry for the current block and restores it on exit. It returns the value that was there before localization.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `my` - lexical (compile-time, block-scoped) declaration; the right tool when you want a truly private variable
- `our` - lexical alias to a package variable; common partner for `local` when the global is in another package
- `state` - persistent lexical; cannot be localized
- `delete` - combined with `local` (`delete local $h{k}`) to scope an element deletion to the current block
- `eval` - common `local` target: `local $_` around an inner `eval` prevents it leaking errors to the caller
- `$/` - input record separator; canonical `local` target for slurp-mode file reads
- `$_` - default scalar; `local $_` strips magic (since 5.14) so a subroutine can reuse it safely
- `$@` - last `eval` error; `local $@` shields the caller from an inner `eval`'s error

Time

localtime

Convert an epoch time to a broken-down calendar time in the local time zone.

`localtime` takes a Unix epoch value - seconds since 1970-01-01 UTC, the shape returned by `time` - and decomposes it into year, month, day, hour, minute, second, weekday, year-day, and a daylight-saving flag, all expressed in the **local** time zone. The time zone is read from the `TZ` environment variable if set, otherwise from the system default (typically `/etc/localtime`). For the UTC companion that ignores `TZ`, see `gmtime`.

Synopsis

```
localtime EXPR
localtime
```

What you get back

The return shape depends on context, and `localtime` is one of the handful of built-ins where the **same expression in list context and scalar context produces completely different answers**.

List context - a 9-element list, all numeric, taken straight out of the C struct `tm`:

```
#      0      1      2      3      4      5      6      7      8
my ($sec, $min, $hour, $mday, $mon, $year, $yday, $isdst) =
    localtime(time);
```

Index	Name	Range	Notes
0	\$sec	0..60	60 is possible for leap seconds
1	\$min	0..59	
2	\$hour	0..23	
3	\$mday	1..31	1-based day of the month
4	\$mon	0..11	0-based: 0 = January, 11 = December
5	\$year	years - 1900	e.g. 129 for 2029
6	\$wday	0..6	0 = Sunday, 3 = Wednesday
7	\$yday	0..365	0-based day of the year; 365 in leap year
8	\$isdst	boolean	true when DST is in effect for the local TZ

The 0-based \$mon and 1900-based \$year are long-standing C inheritances; most bugs written against localtime come from forgetting exactly one of them. The fix for \$year is `+= 1900`:

```
my @t = localtime;
my $full_year = $t[5] + 1900;
```

Scalar context - the familiar English `ctime(3)` string, always 24 characters, always in English regardless of locale:

```
my $now_string = localtime;           # "Thu Oct 13 04:54:34 1994"
```

For a locale-aware or custom-formatted string, do not try to pick apart the scalar form. Go through the list form and `POSIX::strftime`:

```
use POSIX qw(strftime);
my $now_string = strftime "%a %b %e %H:%M:%S %Y", localtime;
```

Global state it touches

- **TZ** in `%ENV` - selects the local time zone. A program that wants to interpret a timestamp in a specific zone sets `$ENV{TZ}` and calls `POSIX::tzset` before `localtime` (see *Edge cases* for the caveat).
- System zoneinfo (`/etc/localtime`, `/usr/share/zoneinfo/...`) when `TZ` is unset - read lazily by the C library, not by Perl directly.
- `%ENV` changes to `TZ` do **not** always take effect on the next call; see *Edge cases*.

`localtime` does not touch `$_` or any output-related globals.

Examples

Current wall-clock date in a human-readable string:

```
my $now = localtime;           # "Wed Apr 23 14:07:12 2025"
```

Current date as a 9-element list - note the two offsets:

```
my ($s,$m,$h,$md,$mo,$y,$wd,$yd,$dst) = localtime;
printf "%04d-%02d-%02d %02d:%02d:%02d\n",
    $y+1900, $mo+1, $md, $h, $m, $s;    # "2025-04-23 14:07:12"
```

Month name from `$mon` using a lookup table - this is the canonical way, because `$mon` is 0-based exactly so it indexes an array:

```
my @abbr = qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec);
my ($mday, $mon) = (localtime)[3,4];
print "$abbr[$mon] $mday";    # e.g. "Apr 23"
```

Formatting with `POSIX::strftime`

- the right answer whenever you need anything other than the bare English `ctime` string:

```
use POSIX qw(strftime);
my @now = localtime;
my $stamp = strftime "%Y-%m-%d %H:%M:%S", @now;    # "2025-04-23_
→14:07:12"
```

Interpreting a stored epoch in a specific zone. Set `TZ`, call `POSIX::tzset`, then `localtime`:

```
use POSIX qw(tzset);
local $ENV{TZ} = "Europe/Berlin";
tzset();
my $wall = localtime(1_700_000_000);    # "Tue Nov 14 23:13:20 2023"
```

Context trap - `localtime` inside `print` `LIST` is in **list** context, so you get nine numbers concatenated with no spaces. Use `scalar` to force the string form:

```
print localtime, "\n";    # e.g. "12724232312505230"
print scalar localtime, "\n";    # "Wed Apr 23 14:07:12 2025"
```

Edge cases

- **No argument:** `localtime` with no argument uses `time` - the current wall-clock moment.
- **Negative epoch:** dates before 1970-01-01 UTC are accepted on systems with a signed `time_t`. Portability of very distant past or future times depends on the platform's C library.
- **\$year pitfall:** `$year + 1900` gives the full year. Writing `"19$year"` was a Y2K bug and is still wrong; write `1900 + $year`.
- **\$mon pitfall:** `$mon` is 0-based. `localtime` output feeding anything 1-based (including `POSIX::mktime`) does **not** need adjustment - `mktime` expects the same 0-based convention. External systems (databases, ISO-8601 strings) usually want `$mon + 1`.
- **\$isdst is advisory**, not directly comparable across zones. `1` means “the local zone currently applies its DST offset”; `0` means “standard time”; `-1` occasionally

appears when the zone rules cannot decide (e.g. the hour that does not exist on a spring-forward day).

- **TZ reload caveat:** most C libraries (glibc included) cache the parsed zone rules on the first lookup. A later assignment to `$ENV{TZ}` is **not** picked up automatically - call `POSIX::tzset` after every change, or the old zone will keep being used.
- **Scalar string is always English** and is **not** affected by `LC_TIME` or any other locale setting. For a localized string, go through `POSIX::strftime` with `LC_TIME` set to the desired locale.
- **Thread safety:** on systems where the C `localtime(3)` call is not reentrant, concurrent threads mutating TZ can race. In practice pperl and modern glibc use the reentrant `localtime_r` internally, but TZ is still a process-wide global.
- **Leap seconds:** `$sec` can legally be 60 on systems with leap second support. Most systems smear or ignore them and you will never see it; code that must not break on 60 handles it already.
- **%a / %b width** under `POSIX::strftime`: the abbreviated weekday and month names are **not guaranteed three characters** in every locale. Code that assumes a fixed column width will misalign in German, French, and many others.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `gmtime` - same decomposition in UTC; does not consult TZ and is the right choice when you want a time zone-independent value
- `time` - the epoch source that `localtime` most often receives as its argument
- `POSIX::strftime` - format a `localtime` list into any string layout you need, locale-aware
- `POSIX::mktime` - inverse direction: take a 9-element list back to an epoch value
- `POSIX::tzset` - re-read TZ after changing it, mandatory if you want the change to take effect
- `$ENV{TZ}` - the environment variable `localtime` reads to pick the local zone

Misc

lock

Place an advisory lock on a shared variable, array, hash, or subroutine until the lock goes out of scope.

`lock` is the synchronisation primitive of Perl's interpreter-threads model (use `threads`; use `threads::shared`). When `THING` is a variable that has been declared `:shared` - or a reference to a shared aggregate - `lock` acquires an advisory mutex on that datum for the rest of the enclosing block. When the block exits, the lock is released. Any other thread calling `lock` on the same datum blocks until the holder releases it.

The lock is **advisory**: it stops other threads that also call `lock` on the same variable, nothing else. A thread that just reads or writes the shared variable without locking it is not blocked and not serialised. `lock` is a convention, not a barrier.

Synopsis

```
lock $shared_scalar
lock @shared_array
lock %shared_hash
lock &shared_sub
lock $ref_to_shared_aggregate
```

What you get back

The argument itself:

- For a scalar, the scalar value.
- For an array, hash, or subroutine, a reference to it.

The return value is rarely used. `lock` is called for its side effect (acquiring the mutex), not for its value.

Lock is scoped, not statement-scoped

The lock is released when the innermost enclosing block exits, not when the `lock` statement finishes. Two consequences:

- `lock` inside a `{ ... }` block serialises only that block's duration. Exiting the block - by fall-through, `return`, `last`, `die`, whatever - releases the lock.
- You cannot release a lock early. There is no `unlock`. If you need a shorter critical section, introduce a tighter `{ ... }` block:

```
{
    lock %shared_hash;
    $shared_hash{$key} = $value;
} # lock released here
do_unlocked_work();
```

Weak keyword

`lock` is a **weak keyword**. If a subroutine named `lock` is in scope at the point of the call (declared or imported before the call site), that subroutine is called instead of the built-in. This makes `lock` safe to use as a method or function name in code that does not use threads. To force the built-in when a same-named sub is in scope, call it via its package: `CORE::lock($thing)`.

Examples

Serialise a hash update between threads:

```

use threads;
use threads::shared;

my %counts :shared;

sub bump {
    my $key = shift;
    lock %counts;
    $counts{$key}++;
}

```

Lock an aggregate through a reference. lock follows one level of reference to find the datum to lock:

```

my @queue :shared;
my $queue_ref = \@queue;

sub enqueue {
    my $item = shift;
    lock $queue_ref;      # locks @queue
    push @queue, $item;
}

```

Narrow the critical section with an inner block so unrelated work runs unlocked:

```

my $result;
{
    lock %cache;
    $result = $cache{$key};
}
expensive_post_processing($result); # runs without the lock

```

Without `use threads::shared`, `lock` is a no-op. This makes it harmless to leave in code that may run single-threaded:

```

# no `use threads::shared` here
lock $x;                # does nothing, does not warn

```

Edge cases

- **Unshared variable:** lock on a variable that was not declared `:shared` and is not a reference to shared data does nothing useful. Under upstream this may croak in some cases; under others it is silently a no-op. Do not rely on `lock` to catch the “forgot to share” bug.
- **Reference vs aggregate:** `lock \%h` and `lock %h` both lock `%h`. `lock` dereferences one level to find the shared datum.

- **Recursive locking within one thread:** the same thread may re-acquire a lock it already holds; the mutex is recursive per thread. Another thread still blocks.
- **No unlock:** releasing before block exit requires restructuring into a tighter scope. `undef $lock_guard` patterns from other languages do not apply - lock is not a value you store.
- **Weak-keyword ambiguity:** a sub `lock` declared or imported earlier wins over the built-in. Use `CORE::lock` to disambiguate.
- **Not cross-process:** lock coordinates threads within one interpreter process. For file-level locking across processes use `flock`; for advisory filesystem locks they are separate mechanisms with no interaction.

Differences from upstream

- `ppperl` does not implement interpreter threads. The `lock` opcode is recognised by the parser but has no runtime effect - it behaves exactly as upstream `lock` does under a build without threads: `:shared`: a silent no-op returning its argument. Code that uses `lock` defensively (as the upstream documentation recommends for single-threaded compatibility) runs unchanged. Code that depends on `lock` for actual synchronisation will not be synchronised under `ppperl`.
- The companion distribution `threads::shared` is not available under `ppperl`; use `threads::shared` fails at compile time. Parallelism in `ppperl` is exposed through JIT auto-parallelisation (Rayon) on specific loop shapes, not through user-visible `ithreads`.

See also

- `flock` - advisory lock on a filehandle, coordinating between processes rather than threads
- `wait` - reap a child process; the process-level equivalent of waiting for another unit of execution
- `fork` - the other concurrency primitive in core Perl, with no shared memory by default
- `our` - declare a package-scoped variable; commonly paired with `:shared` in threaded code
- `local` - dynamic-scope binding, sometimes confused with `lock` because both are scope-bounded and both restore on block exit

Numeric functions

log

Return the natural logarithm (base *e*) of a number.

`log` is the inverse of `exp`. It takes a numeric `EXPR`, coerces it to a double-precision float if necessary, and returns its logarithm to the base *e* (Euler's number, approximately 2.718281828459045). If `EXPR` is omitted, `log` operates on `$_`.

There is no built-in for logarithms to an arbitrary base - use basic algebra (see the examples below) or `POSIX::log10` / `POSIX::log2` for the two common fixed bases.

Synopsis

```
log EXPR
log
```

What you get back

A double-precision float: the natural logarithm of `EXPR`. Concretely:

- `log(1)` is 0.
- `log(exp(1))` is 1 (the defining identity).
- `log(x)` is strictly increasing for $x > 0$.

Special inputs follow IEEE-754 rules as provided by the platform C library's `log(3)`:

- `log(0)` returns negative infinity (`-Inf`).
- `log(x)` for $x < 0$ returns NaN.
- `log(Inf)` returns `Inf`; `log(NaN)` returns NaN.

The result is always a float, even when the input is an integer or a string.

Examples

The defining identity - the natural log of e is 1:

```
use POSIX ();
print log(exp(1)), "\n";           # 1
```

A base-10 logarithm, built from `log` by change of base. The base- N log of a number equals its natural log divided by the natural log of N :

```
sub log10 {
    my ($n) = @_;
    return log($n) / log(10);
}

print log10(1000), "\n";           # 3
print log10(0.01), "\n";          # -2
```

The same idiom generalised to any base - useful for bits (base 2), decibels (base 10), or anything else:

```
sub log_base {
    my ($base, $n) = @_;
    return log($n) / log($base);
}
```

(continues on next page)

(continued from previous page)

```
print log_base(2, 1024), "\n";      # 10
print log_base(16, 65536), "\n";    # 4
```

Operating on `$_` - handy inside `map` or a while line loop:

```
my @natural_logs = map { log } 1, 2, 3, 4;
```

Numeric coercion of a string argument - `log` evaluates its operand in numeric context just like the arithmetic operators:

```
print log("7.389056"), "\n";      # ~2 (log of e^2, give or take)
```

Edge cases

- **log(0) is -Inf**, not an error and not a division-by-zero exception. Perl does not die; you get a float infinity back and any further arithmetic propagates it:

```
my $x = log(0);                    # -Inf
print $x + 1, "\n";                # -Inf
```

- **log of a negative number is NaN**. Again, no exception - the caller is responsible for range-checking if that matters:

```
my $x = log(-1);                   # NaN
print $x == $x ? "num" : "nan", "\n"; # nan (NaN != NaN)
```

Guard the input if you need a defined result:

```
die "log domain error" unless $n > 0;
my $l = log($n);
```

- **Non-numeric strings** coerce to 0 and therefore produce -Inf, and under use warnings trigger Argument "... isn't numeric:

```
use warnings;
print log("hello"), "\n";          # -Inf, with warning
```

- **No argument form uses \$_**. `log` with an empty argument list reads the current topic; inside `map { log }` or `while (<>) { print log; }` this is the usual shortcut.
- **Integer vs float input** makes no difference to the result type: the return value is always a float. `log(2)` and `log(2.0)` are the same value.
- **Precision** is whatever the platform C library's `log(3)` delivers
 - typically ~15–17 significant decimal digits on x86-64 Linux. Results are not bit-for-bit portable across architectures.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `exp` - the inverse: `exp(log($x))` is `$x` for `$x > 0`, and `log(exp($x))` is `$x` for all finite `$x`
- `**` - the exponentiation operator; `$x ** $y` is equivalent to `exp($y * log($x))` for positive `$x`
- `POSIX::log10` - base-10 logarithm, direct from the C library; faster and slightly more accurate than `log($x)/log(10)`
- `POSIX::log2` - base-2 logarithm, same story
- `POSIX::log1p` - `log(1 + $x)` computed accurately for small `$x`, where the naive form loses precision

Filehandles, files, directories

lstat

Return the 13-element status list for a path **without** following a symbolic link.

`lstat` does exactly what `stat` does - same 13-element result, same special `_filehandle` cache, same failure semantics - with one difference: if the argument names a symbolic link, `lstat` returns information about the *link itself* (its own mode, owner, size, mtime), not the target the link points to. For anything that is not a symbolic link, `lstat` and `stat` are indistinguishable. On systems without symbolic links `lstat` silently degrades to `stat`.

Synopsis

```
lstat FILEHANDLE
lstat EXPR
lstat DIRHANDLE
lstat
```

What you get back

In list context, the same 13-element list as `stat`:

```
my ($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,
    $atime,$mtime,$ctime,$blksize,$blocks)
    = lstat($path);
```

Field meanings are identical to `stat` - device, inode, mode (type + permissions), link count, owner uid/gid, rdev, size, atime, mtime, ctime, preferred I/O block size, and allocated blocks. The type bits in `$mode` are what distinguish a symbolic link: on the link itself they report `S_IFLNK`, whereas `stat` on the same name would report the type of the *target*.

In scalar context `lstat` returns a true value on success and the empty string on failure. On failure the list form returns the empty list and `$!` is set to the reason (`ENOENT`, `EACCES`, etc.).

If `EXPR` is omitted, `lstat` operates on `$_`.

The `_` filehandle

Every successful `lstat` (and every successful `stat` or `filetest`) populates the interpreter's internal "last stat buffer". Passing the bareword `_` as the argument reuses that buffer instead of making another system call:

```
if (-l $path && (my @s = lstat(_))) {
    # $path was a symlink; @s describes the link, not the target
}
```

Because `-l` itself does an `lstat` under the hood, the `lstat(_)` above costs nothing extra. This is the canonical way to "ask several questions about one file without re-stat'ing it". Note that a `filetest` on a bareword filename (`-l $path`) leaves the buffer describing the link; a `filetest` on a filehandle follows the open descriptor and thus leaves a `stat`-shaped buffer. When you want symlink data, reach for `lstat` explicitly.

Global state it touches

- `$_` - the argument when `EXPR` is omitted.
- `$!` - set on failure to the `errno` from the underlying `lstat(2)` call.
- The special `_` filehandle - overwritten on every successful call.

Examples

Distinguish a symlink from its target. `stat` on a symlink silently follows it; `lstat` does not:

```
symlink "/etc/passwd", "link.tmp";

my $via_stat = (stat "link.tmp")[7]; # size of /etc/passwd
my $via_lstat = (lstat "link.tmp")[7]; # size of the link inode
                                           # (length of "/etc/passwd")
```

Walk a directory and separate links from regular files without a second syscall per entry, using the `_` buffer:

```
use Fcntl ':mode';

for my $name (readdir $dh) {
    next if $name eq '.' || $name eq '..';
    lstat "$dir/$name" or next;
    if (S_ISLNK((lstat(_))[2])) {
        print "link: $name\n";
    } elsif (-f _) {
```

(continues on next page)

(continued from previous page)

```

    print "file: $name\n";
} elsif (-d _) {
    print "dir: $name\n";
}
}

```

Detect a dangling symlink - `lstat` succeeds on the link, `stat` fails because the target is gone:

```

if (lstat $path and not stat $path) {
    warn "$path is a dangling symlink: $!\n";
}

```

Follow the link manually when you want both pieces of information:

```

my @link    = lstat $path      or die "lstat $path: $!";
my $target  = readlink $path   // die "readlink $path: $!";
my @target  = stat $path       or die "stat $path: $!";

```

No-argument form operating on `$_`, idiomatic in `File::Find` preprocess blocks and `grep` filters:

```

my @symlinks = grep { lstat; -l _ } @paths;

```

Edge cases

- **Not a symlink:** `lstat` behaves exactly like `stat` - there is no penalty to preferring `lstat` when you are about to branch on `-l`.
- **Systems without symlinks:** `lstat` silently falls back to `stat`. Code written against `lstat` stays portable; it just loses the distinguishing behaviour.
- **Filehandle / dirhandle argument:** a passed `FILEHANDLE` or `DIRHANDLE` refers to an already-opened descriptor, which cannot itself be a symlink (the kernel resolved the link at `open` time). `lstat FILEHANDLE` therefore returns the same result as `stat` on that handle. Use the `EXPR` form on the *path* when you need the link metadata.
- **`lstat(_)` after a failed `lstat`:** the `_` buffer is left in whatever state the last *successful* `stat` put it in. A failed `lstat` does not clear it. Always check the return value before reading `_`.
- **Relative paths:** resolved against the process's current working directory, same as `stat`.
- **Permission to `lstat`:** requires search (x) permission on every directory component of the path, but - unlike `stat` - does **not** require any permission on the link target. `lstat` therefore succeeds on links into directories you cannot read.
- **Bareword vs expression:** `lstat $fh` where `$fh` holds a filehandle stats the handle; `lstat "$fh"` stringifies `$fh` and stats the resulting (usually meaningless) pathname. Quote intentionally.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `stat` - the link-following counterpart; identical return shape and `_` semantics
- `readlink` - read the *target* string of a symbolic link without stat'ing it
- `symlink` - create a symbolic link
- `-l` - filetest operator for “is a symbolic link”; internally performs an `lstat` and leaves the result in `_`
- `$_` - default argument when `EXPR` is omitted
- `$!` - reason for failure when `lstat` returns the empty list

Regular expressions and pattern matching

`m//`

Search a string for a pattern and report whether - and what - it matched.

`m//` is the match operator. It compiles `PATTERN` as a regular expression (see the [regex guide](#)), runs it against a target string, and returns a value shaped by calling context and by the modifiers you apply. The target is whatever sits on the left of `=~` or `!~`; without a binding operator the target is `$_`. The leading `m` is optional when the delimiter is `/`, so `/PATTERN/` and `m/PATTERN/` mean the same thing.

Synopsis

```
$str =~ m/PATTERN/flags
$str =~ /PATTERN/flags
m/PATTERN/flags          # target is $_
/PATTERN/flags           # target is $_
$str =~ m{PATTERN}flags  # any paired non-word delimiters
```

What you get back

Context decides the shape of the return value.

- **Scalar context, no `/g`:** 1 on match, the empty string on no match. Both are usable as booleans; the empty string is a dual-value false that also equals `0` numerically.
- **List context, no `/g`:** the list of capture values (`$1`, `$2`, `$3`, ...) on a successful match; if the pattern has no capture groups, the singleton `(1)`; on failure, the empty list. This is how if `(my ($x, $y) = $s =~ /(\w+)= (\w+)/)` works.
- **Scalar context, `/g`:** each call advances through the string, returning true for the next match and false once there are no more. `pos` on the target tracks where the next attempt will begin.
- **List context, `/g`:** every match in one shot. With capture groups, the flat list of all captures from every match. Without captures, the list of every full match.

Successful matches also populate the regex special variables (\$1 through \$9, \$&, \$`, \$', \$+, %+, %-) for the enclosing dynamic scope. A failed match leaves them holding their previous values - always test the match itself, never a capture variable, to decide whether a match happened.

Global state it touches

- `$_` - the default target when no `=~` binding is given.
- `$1`, `$2`, ... - numbered captures, set on success, unchanged on failure.
- `$&`, `$``, `$'` - match, prematch, postmatch.
- `$+` - the highest-numbered capture that actually matched (useful with alternations).
- `%+`, `%-` - named-capture hashes.
- `${^LAST_SUCCESSFUL_PATTERN}` - the last pattern that matched in the current dynamic scope; also the pattern the empty form `m/ /` reuses (see *Edge cases*).
- `pos` on the target string - read and updated by `/g` matching; reset on failure unless `/c` is also set.
- Locale / Unicode rule sources when `/l`, `/u`, or `/d` are in effect.

Delimiters

With `m`, any pair of non-whitespace characters works as the delimiter, and bracketing pairs nest:

```
m/pattern/
m{pattern}
m[pattern]
m(pattern)
m<pattern>
m!pattern!
m#pattern#
m,pattern,
```

Picking a delimiter that does not appear in the pattern avoids backslash-clutter - known as LTS, *leaning toothpick syndrome*. A path-matching pattern reads cleanly with `m{...}` or `m!...!` and badly with `m/.../`.

Two delimiter choices change semantics:

- **' (single quote)** - no variable interpolation inside `PATTERN`. `m' $foo '` matches the literal four characters.
- **?** - `m?PATTERN?` matches only once between calls to `reset`. The leading `m` is mandatory; since Perl 5.22 the bare `?...?` form is a syntax error.

When the delimiter is a word character (a letter or digit), a space is required after `m`: `m q foo q` is legal, `m qfoo q` is not.

Modifiers

Pattern-compile modifiers (also accepted by `qr`, `s`, and `split`):

- **m** - multi-line: `^` and `$` match at every embedded newline, not only at string ends.
- **s** - single-line: `.` matches every character including newline.
- **i** - case-insensitive matching.
- **x** - ignore whitespace and `#`-comments in the pattern; `xx` extends this into character classes.
- **p** - preserve copies of the matched string. Since 5.20 this is a no-op - `${^PREMATCH}`, `${^MATCH}`, `${^POSTMATCH}` are always available after a successful match.
- **a, u, l, d** - character-set rules for `\d`, `\s`, `\w`, and the POSIX classes. `/a` restricts them to ASCII; `/aa` additionally forbids ASCII/non-ASCII matching under `/i`.
- **n** - non-capturing: `(...)` behaves like `(?:...)` and does not populate `$1`, `$2`, ...
- **o** - compile the pattern exactly once even if interpolated variables change. Almost always the wrong tool; use `qr` to build a reusable compiled pattern instead.

Match-process modifiers (specific to `m//` and `s///`):

- **g** - global matching. Scalar-context behavior is iterative (advances `pos` on each call); list-context behavior returns every match at once.
- **c** - only meaningful with `/g`. A failed `/g` match keeps `pos` where it was instead of resetting to the start; required for lex-style scanners built around `\G`.

Examples

Test whether a string contains a pattern:

```
if ($line =~ /error/i) {  
    warn "matched: $line";  
}
```

Bind a capture in one go:

```
if (my ($key, $val) = $line =~ /^(\w+)\s*=\s*(.*)$/) {  
    $config{$key} = $val;  
}
```

Pull every number out of a string with list-context `/g`:

```
my @nums = "x=1 y=22 z=333" =~ /(\d+)/g;  
# @nums = (1, 22, 333)
```

Iterate matches one at a time with scalar-context `/g`, using `pos` to see where the engine is:

```
my $s = "foo 1 bar 22 baz 333";  
while ($s =~ /(\d+)/g) {
```

(continues on next page)

(continued from previous page)

```
printf "matched %s at offset %d\n", $1, pos($s) - length($1);
}
```

Extended form with the `x` modifier and named captures:

```
if ($ts =~ m{
    ^ (?<year>\d{4}) -
      (?<mon> \d{2}) -
      (?<day> \d{2}) $
  }x) {
    printf "year=%s mon=%s day=%s\n", ${year}, ${mon}, ${day};
}
```

Avoid LTS by picking a delimiter that does not appear in the pattern:

```
next if $path =~ m{^/usr/local/};
```

Use `\G` with `m//gc` to walk a string token-by-token without losing position on a failed arm:

```
while (1) {
    if ($s =~ /\G(\d+)/gc) { push @tok, ['num', $1] }
    elsif ($s =~ /\G(\w+)/gc) { push @tok, ['word', $1] }
    elsif ($s =~ /\G(\s+)/gc) { next }
    else { last }
}
```

Edge cases

- **Empty pattern:** `//` and `m//` reuse the last successfully matched pattern in the current dynamic scope. If nothing has matched yet, an empty pattern matches everywhere. Passing user input straight into `m/$pat/` when `$pat` might be empty is a sharp edge - wrap it in a non-capturing group: `m/(?:$pat)/`. The last successful pattern is also readable as `$_{^LAST_SUCCESSFUL_PATTERN}`.
- **Defined-or ambiguity:** Perl resolves `$x // $y` as the defined-or operator, never as two empty matches. In pathological positions (`print $fh //`) Perl still assumes defined-or; force a match by writing `m//` explicitly or spacing out the delimiters.
- **Failed match leaves captures stale:** `$1` after a failing `/.../` still holds the capture from the *previous* successful match - always gate capture use on the match result.
- **/o with changing variables:** `m/$x/o` locks the first value of `$x` into the compiled pattern. Later changes to `$x` are silently ignored. Reach for `qr` instead when you want an explicit, reusable compilation.
- **Interpolation when the delimiter is `'`:** `m'$var'` is a literal-dollar-var match. This is rarely what you want, and the same effect is available with `\Q...\E` or `quotemeta` in any other delimiter.
- **/g plus target modification:** modifying the target between `/g` iterations resets `pos` to the start. Iterate over a copy if you need to mutate the original as you go.

- **\G outside /g**: without /g, \G anchors at the `pos` the target had at call time and matches at most once. On a string that has never had a /g applied, \G is equivalent to \A.
- **m?...? reset scope**: `reset` clears m?? state only for the current package. A m?? in one package is not affected by `reset` called from another.
- **Comparison operators near an empty regex**: `$x //= 1` is always the defined-or-assignment; if you genuinely want the empty regex, write `m//` rather than `//`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `qr` - compile a pattern once and reuse it; avoids /o and keeps the pattern a first-class value
- `s` - same pattern syntax, replaces what it matches
- `tr` - character-by-character translation; a different tool with a superficially similar shape
- `split` - when you want the pieces between matches rather than the matches themselves
- `pos` - read or set the position /g matching resumes from
- *Regular expressions guide*
 - the regex language itself: assertions, character classes, backreferences, named captures

Lists

map

Apply a block or expression to each element of a list and return the flattened results.

`map` evaluates `BLOCK` (or `EXPR`) once per element of `LIST`, with `$_` aliased to that element, and collects everything each evaluation returns into a single flat list. Because each evaluation may yield zero, one, or many values, the output length is independent of the input length - that flattening is the point of `map`. In list context `map` returns the generated list; in scalar context it returns the total element count.

Synopsis

```
my @out = map { BLOCK } LIST;           # block form
my @out = map EXPR, LIST;                # expression form
my @out = map { f($_) } grep { ... } sort LIST;  # chained
```

What you get back

A single flat list built by concatenating the return values of each evaluation, in input order:

- Return one value per input → transform (same length out).
- Return a list per input → 1-to-N expansion (longer out).
- Return the empty list () → skip this input (shorter out; map-as-filter idiom).

In scalar context the return value is the count of generated elements, not a reference and not a boolean. Assigning the result to a hash is well-defined: the flat list is consumed as alternating key/value pairs.

```
my @doubled = map { $_ * 2 }      1 .. 4;      # (2, 4, 6, 8)
my @words   = map { split /\s+/, $_ } @lines;  # 1-to-N
my @kept    = map { $_ > 0 ? $_ : () } @nums;   # filter
my $count   = map { $_ * $_ } @nums;          # scalar ctx
```

Global state

`$_` is **aliased** to each element of LIST, not copied. Modifying `$_` inside the block modifies the source element. This is occasionally useful but usually a footgun - reach for a plain foreach loop when you mean to mutate in place.

```
my @v = (1, 2, 3);
my @r = map { $_ *= 10; $_ } @v;
# @v is now (10, 20, 30) - the source was mutated
```

Aliasing to a non-variable (a literal, a function return value, a hash-slice element that doesn't exist yet) and then assigning to `$_` fails at runtime with Modification of a read-only value. See `$_`.

map also localises `$_` around the call, so an outer `$_` visible to the caller is restored when map returns.

Examples

Transform - uppercase every word:

```
my @upper = map { uc } @words;
```

1-to-N expansion - split each line on whitespace into its tokens:

```
my @tokens = map { split /\s+/ } @lines;
```

Filter idiom - keep only truthy, transformed values:

```
my @positives = map { $_ > 0 ? $_ : () } @numbers;
```

Build a hash from a list - each evaluation returns a two-element list:

```
my %seen = map { $_ => 1 } @keys;
my %idx  = map { ($keys[$_] => $_) } 0 .. $#keys;
```

Composed pipeline - sort, filter, transform. `map`, `grep`, and `sort` all take a LIST on the right, so they chain right-to-left:

```
my @top = map { "$_->{name}: $_->{score}" }
           sort { $b->{score} <=> $a->{score} }
           grep { $_->{score} >= 50 }
           @records;
```

Flatten structure - each input contributes a variable number of outputs:

```
my @leaves = map { ref($_) eq 'ARRAY' ? @$_ : $_ } @mixed;
```

Pair with index using `List::Util::pairs` or an explicit index variable - `map` itself does not expose one:

```
my @numbered = do {
    my $i = 0;
    map { sprintf "%d. %s", ++$i, $_ } @items;
};
```

Edge cases

- **`$_` is aliased, not copied.** Assigning to `$_` writes back to the source. If the source element is read-only (a constant, the return value of a function call flattened into LIST) the assignment raises `Modification of a read-only value`. Treat `$_` as input-only unless you specifically want the side effect.
- **`map` is not a loop.** `last`, `next`, `redo`, and loop labels do not apply to `map`. They refer to the enclosing real loop and will behave accordingly - usually not what the author intended. Use `foreach` when you need loop-control flow.
- **Void context is deprecated.** `map { ... } LIST`; with the result discarded emits a `Useless use of map in void context` warning under `use warnings`. When you want the side effects, not the list, write a `foreach`.
- **BLOCK vs EXPR parser ambiguity.** `{ ... }` starts both a block and a hash-reference constructor. `map { ... } LIST` and `map { ... }, LIST` look identical to the parser at the opening brace; Perl guesses based on what it finds next. A block starting with what looks like a hash key plus `=>` is especially likely to be guessed wrong:

```
my %h = map { "\L$_" => 1 } @a;    # guessed EXPR, wrong
my %h = map { +"\L$_" => 1 } @a;    # unary + forces BLOCK
my %h = map { ; "\L$_" => 1 } @a;    # leading ; forces BLOCK
my %h = map { ($_, 1) } @a;         # parens - unambiguous
my %h = map { lc($_) => 1 } @a;     # function call -
↪unambiguous
```

To return an anonymous hash per element, force the hash-ref constructor with `+`:

```
my @hashes = map +{ lc($_) => 1 }, @array;
```

- **Order is stable.** map evaluates left-to-right and concatenates results left-to-right. The generated list preserves input order, with each input's contribution appearing contiguously.
- **Empty LIST.** map evaluates the block zero times and returns the empty list. The block is not evaluated for its side effects.
- **Nested map.** Inner map sees its own `$_`; the outer `$_` is shadowed for the duration of the inner evaluation. Capture the outer value first if both are needed:

```
my @pairs = map {
    my $outer = $_;
    map { [$outer, $_] } @inner
} @outer;
```

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `grep` - same shape as map, but keeps elements for which the block is true rather than transforming them; chains naturally with map and `sort`
- `sort` - orders a list; commonly composed with map via the Schwartzian transform (map → sort → map)
- `for` / `foreach` - use when you want iteration for side effects, mutation of the source, or loop-control (`last/next/redo`)
- `x` - list repetition operator; orthogonal to map but often used to build the LIST that map consumes
- `List::Util::reduce` - fold a list to a single value; use when the result is not a list but an aggregate (sum, max, first match)
- `$_` - the element alias inside the block; read-only when the source element is a literal or function return value

Control flow · Classes and OO

method

Declare a named instance method inside a `class` block.

`method` defines a subroutine that expects to be invoked on an object of the enclosing class. It behaves like `sub`, with two additional guarantees that `sub` does not give you: the lexical `$self` is bound to the invocant automatically, and signatures are always on - you write parameters as if use `feature 'signatures'` were in effect, and `$self` does not appear in that signature.

`method` is only valid inside a `class` declaration. Outside a class block it is a compile-time error.

Note

Experimental feature The `class`, `method`, and `field` keywords are part of the experimental class feature (introduced in Perl 5.38). Enabling it with `use feature 'class'`; emits a warning in the `experimental::class` category. The syntax and semantics may still change between Perl releases.

Synopsis

```
method NAME BLOCK
method NAME SIGNATURE BLOCK
method NAME : ATTRS BLOCK
method SIGNATURE BLOCK      # anonymous
method BLOCK                # anonymous
```

What you get back

`method` is a declaration, not an expression, in its named form - it installs the method in the enclosing class's method table and evaluates to nothing useful at the statement position. The anonymous form returns a code reference that remembers the class context and can be called later with an object invocant.

The body of a method is called with the object as the invocant. The return value of the method is whatever the block returns, following the same context rules as a regular subroutine call.

What method gives you that sub does not

Four things differ from writing `sub` inside a `class` block:

- **`$self` is pre-declared.** A lexical `$self` is created in the method's scope and initialised to the invocant. You do not - and should not - shift it off `@_`.
- **Signatures are always enabled.** You can write `method greet($name) { ... }` without `use feature 'signatures'`. The `$self` invocant is consumed before the signature binds, so it never appears in the parameter list.
- **Fields are in scope.** Any `field` variables declared in the enclosing class are visible by their lexical names inside the method body. A plain `sub` inside a `class` block does not see fields.
- **It is registered as a method.** The method table of the class records the name, which is what the method-resolution machinery looks up for `$obj->name(...)` dispatch. A plain `sub` inside the class block is a helper, not a method.

Put the other way: a `sub` inside a `class` block is an ordinary package subroutine that happens to live in that package's namespace. It has no `$self`, no field visibility, and is not part of the class's method interface - call it via `ClassName::helper(...)` if you must, not via an arrow.

Examples

A minimal class with a method and a field:

```
use v5.38;
use feature 'class';
no warnings 'experimental::class';

class Greeter {
    field $greeting = "Hello";

    method greet($name) {
        return "$greeting, $name";
    }
}

my $g = Greeter->new;
say $g->greet("world");           # Hello, world
```

Signature defaults work exactly as they do for `sub`:

```
class Greeter {
    field $greeting = "Hello";

    method greet($name = "someone") {
        return "$greeting, $name";
    }
}

say Greeter->new->greet;           # Hello, someone
```

A method with no parameters still uses the signature form - the empty `()` is not required, but it is allowed and is the common style when the method takes arguments elsewhere in the class:

```
class Counter {
    field $n = 0;

    method inc      { $n++ }
    method value () { return $n }
}
```

An anonymous method - useful as a factory or as a callback that needs access to `$self` and fields:

```
class AnonMethodFactory {
    method make_printer {
        return method { say "called on $self" };
    }
}
```

A lexical (private) method declared with `my method`. Invoke it via the `->&` operator, which bypasses method lookup and calls the lexical directly as if it were a method:

```
class LexicalMethod {
  my method _check ($x, $y) {
    return $x > 0 && $y > 0;
  }

  method process ($x, $y) {
    return unless $self->&_check($x, $y);
    # ...
  }
}
```

Regular sub inside a class, contrasted with method - the sub is a helper, the method is the public interface:

```
class Temperature {
  field $celsius;

  sub _c_to_f ($c) { $c * 9 / 5 + 32 } # helper, no $self, no _fields

  method fahrenheit {
    return _c_to_f($celsius);
  }
}
```

Edge cases

- **Outside a class block is a compile-time error.** `method foo { ... }` at the top level, or inside a package that is not a class, does not parse. Use `sub` there.
- **`$self` is read-only in the sense that you cannot declare another lexical with the same name** at the top of the method - it is already in scope. Assigning to `$self` (e.g. `$self = ...`) is permitted but rarely useful and does not change the caller's invocant.
- **`@_` is not the argument list you expect.** Under the signatures that method enables, `@_` inside the body is empty after the signature binds, just as in any signature-using sub. Read parameters from the signature, not `@_`.
- **Method-resolution order is set by the enclosing class.** method itself does not take an MRO option; inheritance and dispatch follow whatever the `class` declaration configured via `:isa` and related attributes.
- **method does not participate in prototypes.** Methods are invoked through dispatch, not direct symbol-table calls, so prototypes (the `:prototype( ... )` attribute on a sub) have no effect here. Use the signature to constrain the call shape.
- **Attributes.** The method `NAME : ATTRS BLOCK` form accepts attributes the same way `sub NAME : ATTRS BLOCK` does. The set of useful attributes on methods is still settling with the class feature; consult upstream before relying on custom ones.
- **Anonymous-method return-from-method trap.** `return method { ... }` re-

turns a code reference. `method { ... }` as the last expression of a method without `return` is still a code reference, but readers often misread it as a block - prefer explicit `return` for clarity.

Differences from upstream

Fully compatible with upstream Perl 5.42.

The `class` feature - and therefore `method` - is marked experimental in upstream; `ppperl` inherits that status and the `experimental::class` warning category.

See also

- `class` - declares the package that `method` lives in; `method` only parses inside a `class` block
- `field` - per-instance variables that are visible by name inside every method of the enclosing class
- `sub` - declare a regular subroutine; use this for helpers inside a class that do not need `$self` or field access
- `__CLASS__` - the name of the current class, useful inside methods that need to dispatch back to class-level code
- `return` - explicit return from a method body; recommended when the last expression is itself a `method { ... }` anonymous form
- `my` - prefix a method with `my` to declare a lexical (private) method invoked via `->&`

[Filehandles](#), [files](#), [directories](#)

mkdir

Create a single directory on the filesystem.

`mkdir` asks the operating system to create the directory named by `FILENAME`, with permission bits given by `MODE` (further restricted by the process `umask`). It creates exactly **one** directory

- every parent component in `FILENAME` must already exist. For recursive creation, reach for `File::Path::make_path` instead.

Synopsis

```
mkdir FILENAME, MODE
mkdir FILENAME
mkdir
```

What you get back

1 on success, 0 on failure with `$!` set to the system `errno`. Unlike most I/O primitives, the failure value is a plain false 0, not `undef` - check the return value, then consult `$!` for the reason:

```
mkdir $path, 0755
  or die "mkdir $path failed: $!";
```

Default arguments

Both arguments default:

- MODE defaults to 0777. The kernel masks the supplied mode with the current process `umask`, so the directory's effective mode is `MODE & ~umask`. A typical `umask` of 022 turns the default 0777 into 0755 on disk.
- FILENAME defaults to `$_`. Bare `mkdir`; inside a loop over pathnames creates each in turn:

```
mkdir for qw(logs cache tmp);
```

Pass MODE explicitly whenever the directory should be private (mail spools, key stores, session directories). Otherwise prefer a permissive MODE and let the user's `umask` narrow it - the `umask` page discusses the trade-off in detail. Setting bits outside the permission range (e.g. `setuid/setgid/sticky`) yields implementation-defined behaviour per POSIX 1003.1-2008.

Global state it touches

- Sets `$!` (`errno`) on failure; leaves it untouched on success.
- Reads FILENAME from `$_` when called with no arguments.
- The effective permission bits depend on the process `umask`.

Examples

Create a directory with an explicit mode:

```
mkdir "build", 0755
  or die "mkdir build: $!";
```

Create it only if missing, without treating “already there” as an error:

```
use Errno qw(EEXIST);
unless (mkdir "cache", 0700) {
  die "mkdir cache: $!" unless $! == EEXIST;
}
```

Private directory - do not let `umask` widen it:

```
mkdir "$ENV{HOME}/.secrets", 0700
  or die "mkdir secrets: $!";
```

Bare form over a list, using `$_` as the filename:

```
for (qw(a b c)) {
    mkdir or warn "mkdir $_: $!";
}
```

Recursive creation is **not** mkdir's job - use File::Path:

```
use File::Path qw(make_path);
make_path("var/log/app/2026", { mode => 0755 })
or die "make_path failed: $!";
```

Edge cases

- **Parent must exist.** mkdir "a/b/c" fails with ENOENT if a/b is missing. Use File::Path::make_path for deep paths.
- **Target already exists.** Fails with EEXIST, regardless of whether the existing entry is a directory, file, or symlink. Check \$! against Errno::EEXIST when “already there” is acceptable.
- **Trailing slashes.** POSIX 1003.1-1996 allows any number of trailing slashes; Perl strips them before the syscall so that filesystems which reject "foo/" still behave consistently.
- **Permission mismatch.** MODE is masked by umask. If you need an exact mode on disk, either save and restore the umask, or chmod the directory after creation:

```
mkdir $dir, 0700 or die $!;
chmod 0700, $dir;                                # defeat any umask surprises
```

- **MODE omitted entirely.** mkdir \$path is mkdir \$path, 0777
 - still masked by umask, so the on-disk mode is typically 0755, not 0777.
- **Non-permission bits in MODE.** Setuid / setgid / sticky bits on a mkdir call have implementation-defined results. Set them with a follow-up chmod if you need them.
- **Read-only filesystem or missing write permission on the parent** fails with EROFS or EACCES respectively. \$! discriminates.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `rmdir` - remove an empty directory; the inverse operation
- `chmod` - adjust permission bits after creation when umask would otherwise narrow them
- `umask` - the mask that silently restricts every mkdir MODE
- `stat` - inspect an existing path's type and mode, e.g. to distinguish “already a directory” from “exists as a file”

- `File::Path::make_path` - recursive directory creation; the right tool when intermediate parents may not exist

SysV IPC

msgctl

Perform a control operation on a System V IPC message queue.

`msgctl` is a direct wrapper around the C `msgctl(2)` system call. It inspects, modifies, or removes the message queue identified by ID. The operation selected by CMD determines how ARG is used - either as an input descriptor, an output buffer, or ignored entirely. The symbolic constants needed for CMD (`IPC_STAT`, `IPC_SET`, `IPC_RMID`, and on Linux `IPC_INFO`, `MSG_STAT`, `MSG_INFO`) live in `IPC::SysV`, which you have to pull in before the names resolve.

Synopsis

```
use IPC::SysV;

msgctl $id, IPC_STAT, $buf      # read queue info into $buf
msgctl $id, IPC_SET,  $buf      # write queue info from $buf
msgctl $id, IPC_RMID, 0         # destroy the queue
```

What you get back

Return value follows the same convention as `ioctl`:

- `undef` on error, with `$!` set.
- The string "`0 but true`" when the underlying call returned `0`. This is the zero-valued success case - it tests true in boolean context and compares equal to `0` in numeric context, so one check handles both success and failure.
- The raw non-zero integer return otherwise (for example the index returned by Linux-specific `MSG_STAT`).

The "`0 but true`" convention exists precisely so that `msgctl(...)` or `die` does the right thing even when the syscall legitimately returns zero:

```
msgctl $id, IPC_RMID, 0
or die "msgctl IPC_RMID failed: $!";
```

How ARG is used

ARG plays three different roles depending on CMD. The kernel only touches the memory in the directions the command requires:

- **IPC_STAT** - ARG must be a **writable scalar**. The kernel fills it with the packed `msqid_ds` structure describing the queue (permissions, owner, last-send/last-receive times, queue byte limits, counts). Unpack with `IPC::SysV` helpers rather than hand-rolling the layout; `msqid_ds` varies across kernels and libc versions.

- **IPC_SET** - ARG must contain a packed `msgid_ds` **produced by a prior `IPC_STAT`** on the same (or compatible) queue. Only a few fields are honoured: `msg_perm.uid`, `msg_perm.gid`, `msg_perm.mode` (low 9 bits), and `msg_qbytes`. The rest is ignored.
- **IPC_RMID** - ARG is unused. Pass `0`. The queue is marked for destruction; any blocked `msgsnd` / `msgrcv` on it fails with `EIDRM`.

On Linux only, `MSG_STAT` and `MSG_INFO` also accept a writable scalar for ARG and return extra data; these are non-portable and should be gated behind a runtime check.

Global state it touches

- `$!` - set on failure with the system `errno` (`EPERM`, `EACCES`, `EINVAL`, `EIDRM`, `EFAULT` are the common values).
- No other special variables are touched.

Examples

Read the queue header and inspect the current byte limit:

```
use IPC::SysV qw(IPC_STAT);

my $buf = "";
msgctl($id, IPC_STAT, $buf)
    or die "msgctl IPC_STAT failed: $!";

# Layout is platform-dependent; use IPC::SysV helpers to decode.
```

Shrink the queue's byte budget. Round-trip through `IPC_STAT` so the immutable fields keep their current kernel values:

```
use IPC::SysV qw(IPC_STAT IPC_SET);

my $buf = "";
msgctl($id, IPC_STAT, $buf) or die "stat: $!";
# ... modify msg_qbytes inside $buf via IPC::SysV ...
msgctl($id, IPC_SET, $buf) or die "set: $!";
```

Destroy a queue when done. ARG is ignored, but it still has to be passed - `0` is the conventional placeholder:

```
use IPC::SysV qw(IPC_RMID);

msgctl($qid, IPC_RMID, 0)
    or warn "could not remove queue $qid: $!";
```

Idiomatic success test that works for the "`0` but `true`" case:

```
my $rc = msgctl($id, IPC_RMID, 0);
die "msgctl failed: $!" unless defined $rc; # undef = error
# $rc is now either "0 but true" or a positive integer
```

Edge cases

- **IPC::SysV not loaded:** bare constant names like `IPC_STAT` become barewords and, under `use strict`, a compile-time error. Always use `IPC::SysV` first.
- **Scalar passed to `IPC_STAT` is not writable:** croaks with `Modification of a read-only value attempted`. Use a plain `my $buf = ""` - not a constant, not a string literal.
- **Length mismatch on `IPC_SET`:** if `ARG` is shorter than the kernel's `msqid_ds`, the call fails with `EFAULT`. Always start from an `IPC_STAT` buffer rather than packing one from scratch.
- **Removed queue (`EIDRM`):** once `IPC_RMID` has been issued, any further operation on the same ID fails with `EINVAL` set to `EIDRM`, not `EINVAL`. Scripts that poll a queue must treat `EIDRM` as “queue is gone, stop.”
- **Permission errors:** `IPC_SET` and `IPC_RMID` require effective UID match or `CAP_SYS_ADMIN`. Kernel returns `EPERM`, not `EACCES`, for ownership failures.
- **Return-value trap:** `if (msgctl(...) == 0)` is **wrong** - on success the value is the string `"0 but true"`, which stringifies as non-empty but compares numerically equal to `0`. Test with `defined` for the error case, or rely on `or die`.
- **Platform availability:** System V IPC is not universal. On systems where the kernel omits it, `msgctl` raises the runtime exception `msgctl not implemented`.

Differences from upstream

Fully compatible with upstream Perl 5.42 on Linux. System V IPC is the only supported IPC surface; `pperl` targets Linux, so the portability caveats that upstream `perlport` lists for non-Linux platforms do not apply.

See also

- `msgget` - obtain the queue ID that `msgctl` operates on
- `msgsnd` - send a message to the queue
- `msgrcv` - receive a message from the queue
- `semctl` - the semaphore-set analogue, same return convention and same `IPC_STAT` / `IPC_SET` / `IPC_RMID` pattern
- `shmctl` - the shared-memory-segment analogue
- `IPC::SysV` - the constants (`IPC_STAT`, `IPC_SET`, `IPC_RMID`) and the `msqid_ds` pack/unpack helpers this call requires

SysV IPC

`msgget`

Create or look up a System V IPC message queue and return its id.

`msgget` is the entry point to the System V message-queue family: every later `msgsnd`, `msgrcv`, or `msgctl` call needs an id, and this is where that id comes from. KEY names the

queue across the whole system (a `key_t` integer, typically built with `IPC::SysV::ftok` or the special value `IPC_PRIVATE`). `FLAGS` is the bitwise-or of permission bits (low nine, same layout as file modes) and optional `IPC_CREAT` / `IPC_EXCL` flags. The call is a direct wrapper around the `msgget(2)` syscall.

Synopsis

```
use IPC::SysV qw(IPC_CREAT IPC_EXCL IPC_PRIVATE S_IRUSR S_IWUSR);

my $id = msgget($key, IPC_CREAT | 0600);
my $id = msgget(IPC_PRIVATE, 0600);
my $id = msgget($key, IPC_CREAT | IPC_EXCL | 0600);
```

What you get back

A non-negative integer message-queue id on success, or `undef` on failure (with `$!` set to the `errno` value from the underlying syscall). The id is an opaque handle - do not do arithmetic on it, just pass it back to the other `msg*` built-ins.

```
my $id = msgget($key, IPC_CREAT | 0600)
    // die "msgget($key) failed: $!";
```

Global state it touches

- `$!` - set on failure to the `errno` from `msgget(2)` (commonly `EACCES`, `EEXIST`, `ENOENT`, `ENOSPC`, `ENOMEM`).

No other special variables are involved. The queue itself lives in the kernel, not in the Perl process, and persists across `exec` and after the creating process exits - it must be removed explicitly with `msgctl` and `IPC_RMID`.

Examples

Create a private queue that no other process can name - the usual choice when the queue is shared only with children of this process:

```
use IPC::SysV qw(IPC_PRIVATE);
my $id = msgget(IPC_PRIVATE, 0600) // die "msgget: $!";
```

Create or attach to a named queue using a `key_t` derived from a file path. `ftok` turns a path plus a single-character project id into a stable key so unrelated processes can agree on a queue:

```
use IPC::SysV qw(ftok IPC_CREAT);
my $key = ftok("/var/run/myapp", 'M');
my $id = msgget($key, IPC_CREAT | 0600) // die "msgget: $!";
```

Create exclusively - fail if the queue already exists. This is how you detect that a previous instance of your program is still around:

```
use IPC::SysV qw(IPC_CREAT IPC_EXCL);
my $id = msgget($key, IPC_CREAT | IPC_EXCL | 0600);
if (!defined $id) {
    die "queue already exists" if ${EEXIST};
    die "msgget: $!";
}
```

Attach to an existing queue without creating one - omit `IPC_CREAT`. Returns `undef` with `$!` set to `ENOENT` if no such queue exists:

```
my $id = msgget($key, 0) // die "no queue for key $key: $!";
```

Remove the queue when done. `msgget` itself has no teardown counterpart; removal goes through `msgctl`:

```
use IPC::SysV qw(IPC_RMID);
msgctl($id, IPC_RMID, 0) or warn "msgctl IPC_RMID: $!";
```

Edge cases

- **KEY must be an integer**, typically a `key_t`. Passing a string triggers the usual numeric-conversion rules and will almost certainly name the wrong queue. Build keys with `IPC::SysV::ftok` or use `IPC_PRIVATE`.
- **IPC_PRIVATE always creates a new queue**, regardless of whether `IPC_CREAT` is set. The key is not reusable - you cannot `msgget` the same queue later from an unrelated process using `IPC_PRIVATE`.
- **Permission bits apply at every call, not just creation**. Attaching to an existing queue without `IPC_CREAT` still checks the permission bits against the queue's owner/group and the current process's effective uid/gid. Mismatched bits give `EACCES`.
- **IPC_EXCL without IPC_CREAT is meaningless** - the kernel ignores it. If you want exclusive creation, always pair them.
- **System-wide limits**. Creation can fail with `ENOSPC` when the kernel's `MSGMNI` limit is hit, or `ENOMEM` under memory pressure. These are not bugs in your program; they are operational conditions.
- **Queue lifetime outlives processes**. A queue created here persists until explicitly removed with `msgctl` and `IPC_RMID`, or until the system reboots. Forgotten queues leak kernel resources that `ipcs` and `ipcrm` are used to find and clean up.
- **Not all systems have SysV message queues**. On platforms that omit them, the built-in croaks with `msgget` not implemented on this architecture. The portability warning in upstream POD applies.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `msgsnd` - send a message to the queue returned by `msgget`
- `msgrcv` - receive a message from that queue
- `msgctl` - query, set options on, or remove the queue; teardown after `msgget` goes through here
- `semget` - same API shape for System V semaphores when you need synchronization rather than data transfer
- `shmget` - same API shape for System V shared memory when you need direct memory sharing rather than message passing
- `IPC::SysV` - provides `ftok` and the `IPC_*` / `S_*` constants `msgget` expects in `FLAGS`

SysV IPC

msgrcv

Receive a message from a System V IPC message queue.

`msgrcv` is the Perl binding for the `msgrcv(2)` syscall. It pulls one message off the queue identified by ID, stores the message type and payload into `VAR`, and returns a truthy value on success or false on error. `SIZE` is the maximum number of payload bytes the caller is willing to accept; `TYPE` selects which message to receive; `FLAGS` is the bitmask passed to the kernel (`IPC_NOWAIT`, `MSG_NOERROR`, etc.).

The returned buffer is **not** just the message body - the first `sizeof(long)` bytes are the message type the sender passed to `msgsnd`, followed by the payload. Use `unpack` with the `l! a*` template to split them apart.

Synopsis

```
use IPC::SysV qw(IPC_NOWAIT MSG_NOERROR);

msgrcv $qid, my $buf, 1024, 0, 0           or die "msgrcv: $!";
msgrcv $qid, my $buf, 1024, 0, IPC_NOWAIT; # non-blocking
my ($type, $payload) = unpack "l! a*", $buf;
```

What you get back

True on success, false on error with `$!` set. On success, `VAR` is overwritten with the received frame: a native long (the message type) immediately followed by up to `SIZE` bytes of payload. The variable is marked tainted under `-T`.

`VAR` is grown or shrunk to fit the actual frame length - you do not need to pre-size it. The content before the call is discarded.

To get the type and payload as separate values:

```
my ($type, $msg) = unpack "l! a*", $buf;
```

Arguments

- ID - the queue id previously returned by `msgget`.
- VAR - a lvalue scalar that receives the raw frame. Its previous value is discarded on success.
- SIZE - the maximum payload size in bytes. If the next matching message is larger than this, `msgrcv` fails with E2BIG and the message stays on the queue - **unless** `MSG_NOERROR` is set in `FLAGS`, in which case the message is truncated silently and the extra bytes are lost.
- TYPE - selects which message to receive:
 - 0 - the first message on the queue, whatever its type.
 - positive N - the first message whose type is exactly N.
 - negative -N - the first message of the lowest type that is $\leq \text{abs}(N)$, useful for priority queues.
- FLAGS - bitmask of `IPC_NOWAIT` (return `ENOMSG` immediately instead of blocking when no matching message is available), `MSG_NOERROR` (truncate instead of failing on oversized messages), and the Linux-specific `MSG_EXCEPT` (with a positive TYPE, receive the first message whose type is *not* TYPE).

The constants live in `IPC::SysV`; the bare names do not become visible until you import them.

Global state it touches

- `$!` - set on error to the underlying `errno` (`ENOMSG`, `E2BIG`, `EIDRM`, `EINVAL`, `EACCES`, `EFAULT`, `EINTR`).
- Taint mode - the received VAR is marked tainted, since its contents came from another process.

Examples

Blocking receive of any message, then split type and payload:

```
use IPC::SysV qw(IPC_CREAT S_IRUSR S_IWUSR);

my $qid = msgget(IPC_PRIVATE, S_IRUSR | S_IWUSR) // die "msgget: $!";
msgrcv $qid, my $buf, 4096, 0, 0 or die "msgrcv: $!";
my ($type, $payload) = unpack "l! a*", $buf;
print "got type=$type payload=$payload\n";
```

Receive only messages of type 42:

```
msgrcv $qid, my $buf, 4096, 42, 0 or die "msgrcv: $!";
```

Non-blocking poll - return immediately if the queue is empty:

```

use IPC::SysV qw(IPC_NOWAIT);
use Errno qw(ENOMSG);

if (msgrcv $qid, my $buf, 4096, 0, IPC_NOWAIT) {
    my ($type, $msg) = unpack "l! a*", $buf;
    handle($type, $msg);
} elsif ($! == ENOMSG) {
    # queue was empty; try again later
} else {
    die "msgrcv: $!";
}

```

Truncate oversized messages rather than failing:

```

use IPC::SysV qw(MSG_NOERROR);

msgrcv $qid, my $buf, 64, 0, MSG_NOERROR    or die "msgrcv: $!";
# payload is at most 64 bytes, excess discarded without error

```

Priority-style receive - take the lowest-numbered message whose type is ≤ 3 :

```

msgrcv $qid, my $buf, 4096, -3, 0          or die "msgrcv: $!";

```

Edge cases

- **VAR must be an lvalue.** Passing a constant or a read-only scalar (for instance an element of `@_` that was passed by reference to a literal) raises Modification of a read-only value attempted.
- **Frame layout is native long, not four bytes.** On 64-bit Linux the type occupies 8 bytes; on 32-bit systems it occupies 4. The `l!` in `unpack "l! a*"` - the `!` - is mandatory; a plain `l` assumes 4 bytes and will corrupt payloads on 64-bit hosts.
- **SIZE excludes the type header for MSG_NOERROR purposes.** The kernel compares SIZE against the payload length, not the frame length. Size the buffer for the largest payload you expect; the type header is written beyond it.
- **Signal interruption.** A blocking `msgrcv` returns false with `$! == EINTR` when a signal arrives before a message is available. Retry the call unless the signal is meant to abort the receive loop.
- **Queue removed while blocked.** If another process calls `msgctl` with `IPC_RMID` while this process is asleep in `msgrcv`, the call returns false with `$! == EIDRM`.
- **Permissions.** Receiving requires read permission on the queue. `EACCES` means the queue exists but the current uid/gid doesn't match the queue's mode bits.
- **TYPE = 0 sees FIFO order.** There is no "any" priority short of explicit negative TYPE - messages of all types are pulled in send order.
- **Kernel support.** Linux kernels built without `CONFIG_SYSVIPC`, and container runtimes that don't expose the IPC namespace, fail every `msgrcv` with `ENOSYS`. Check during startup if your deployment target is uncertain.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `msgsnd` - the counterpart that puts messages on the queue; its pack "l! a*" template is the inverse of what you unpack here
- `msgget` - obtain the queue id ID you pass in
- `msgctl` - inspect or delete the queue, including the `IPC_RMID` that causes a blocked `msgrcv` to return `EIDRM`
- `unpack` - split the returned frame into type and payload with `l! a*`
- `IPC::SysV` - source of the `IPC_NOWAIT`, `MSG_NOERROR`, and related flag constants
- `IPC::Msg` - object-oriented wrapper that hides the pack/unpack dance behind a `rcv` method

SysV IPC

`msgsnd`

Send a message to a System V IPC message queue.

`msgsnd` is a thin wrapper around the `msgsnd(2)` syscall. It appends `MSG` to the queue identified by `ID` (the queue id previously returned by `msgget`), and returns true on success or false on error with `$!` set. `MSG` is **not** free-form: it must start with a native long message type, followed by the payload bytes. `FLAGS` control blocking and error-on-overflow behaviour, typically `0` or `IPC_NOWAIT`.

Synopsis

```
msgsnd ID, MSG, FLAGS
```

What you get back

True on success, false on failure. On failure `$!` is set to the `errno` from `msgsnd(2)` - common values are `EAGAIN` (queue full, `IPC_NOWAIT` was set), `EIDRM` (queue was removed), `EINTR` (signal interrupted a blocking call), and `EACCES` (no write permission on the queue).

Because the return value is a plain boolean, always check it before assuming the message was enqueued:

```
msgsnd($qid, $msg, 0)
    or die "msgsnd($qid) failed: $!";
```

Message format

The kernel reserves the first `sizeof(long)` bytes of every queued message for the **message type**, a positive long integer that `msgrcv` callers use to select which messages to dequeue. The payload follows, uninterpreted.

Build the buffer with `pack` using the `l!` (native long) and `a*` (raw bytes) specifiers:

```
my $type    = 1;
my $payload = "hello queue";
my $msg     = pack("l! a*", $type, $payload);
msgsnd($qid, $msg, 0) or die "msgsnd: $!";
```

The message type **must be greater than zero**. Zero and negative values are reserved for the selector semantics of `msgrcv` and cause `msgsnd(2)` to fail with `EINVAL`.

Global state it touches

- `$!` - set to the `errno` from the failed syscall when `msgsnd` returns false. Untouched on success.

Examples

Send a simple typed message with the default blocking behaviour. If the queue is full, the call waits until space is available:

```
use IPC::SysV qw(IPC_PRIVATE IPC_CREAT S_IRUSR S_IWUSR);

my $qid = msgget(IPC_PRIVATE, IPC_CREAT | S_IRUSR | S_IWUSR)
    // die "msgget: $!";

my $msg = pack("l! a*", 1, "job-42");
msgsnd($qid, $msg, 0) or die "msgsnd: $!";
```

Non-blocking send - return immediately with `EAGAIN` in `$!` if the queue is full, rather than sleeping:

```
use IPC::SysV qw(IPC_NOWAIT);
use Errno qw(EAGAIN);

unless (msgsnd($qid, $msg, IPC_NOWAIT)) {
    if ($! == EAGAIN) {
        warn "queue full, dropping message\n";
    } else {
        die "msgsnd: $!";
    }
}
```

Use the type field as a priority / channel tag. Receivers pick by type via `msgrcv`:

```
my %CHANNEL = (log => 1, metric => 2, alert => 3);
```

(continues on next page)

(continued from previous page)

```
sub send_event {
    my ($kind, $body) = @_ ;
    my $msg = pack("l! a*", $CHANNEL{$kind}, $body);
    msgsnd($qid, $msg, 0) or die "msgsnd[$kind]: $!";
}

send_event(alert => "disk > 90%");
send_event(metric => "rps=1240");
```

Send a structured payload - pack the header fields after the mandatory type prefix:

```
my $type = 1;
my $body = pack("N N a*", $job_id, $retry_count, $payload);
msgsnd($qid, pack("l! a*", $type, $body), 0)
    or die "msgsnd: $!";
```

Edge cases

- **Message type must be positive.** Passing 0 or a negative type fails with EINVAL. The type-selection semantics in `msgrcv` reserve those values for selector logic.
- **Message exceeds msg_qbytes.** If MSG is larger than the queue's configured per-message limit, `msgsnd` fails with EINVAL regardless of FLAGS. Check and raise the limit with `msgctl IPC_SET` if you control the queue.
- **Queue full without IPC_NOWAIT.** The call blocks until another process dequeues, the queue is removed (EIDRM), or a signal interrupts it (EINTR). Retry on EINTR if the signal is expected and benign.
- **Queue full with IPC_NOWAIT.** Fails immediately with EAGAIN. This is the only way to enforce bounded latency on the send side.
- **Missing type prefix.** Forgetting to pack the l! type prefix passes the raw payload as-is; the first `sizeof(long)` bytes are then interpreted as the type - typically a large positive or negative number - and the rest as the body. Receivers filtering by type see nothing.
- **ID is a queue id, not a key.** Pass the integer returned by `msgget`, not the KEY you gave to `msgget`. Confusing the two yields EINVAL or, worse, a successful send to an unrelated queue that happens to share the numeric value.
- **Permissions.** The process must have write permission on the queue (the w bit of the queue's mode for its effective uid/gid). Otherwise the call fails with EACCES.
- **Tainting.** MSG is read as a byte string; no taint checking is performed on it here. If MSG came from an untrusted source, the receiver side (`msgrcv`, which *does* taint its output) is where filtering belongs.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `msgget` - obtain the queue id passed as ID; call this first, or receive the id from whoever created the queue
- `msgrcv` - the matching receive primitive; its TYPE argument selects against the `l!` prefix written here
- `msgctl` - inspect the queue's `msg_qbytes` limit or remove the queue when done
- `pack` - build the `l! a*` buffer `msgsnd` requires; also the tool for any further structure inside the payload
- `$!` - carries the `errno` from a failed `msgsnd`; compare against `Errno` constants to branch on specific failures

Scoping

my

Declare one or more lexically scoped variables.

`my` introduces new variables whose visibility is bounded by the enclosing block, file, or `eval`. The compiler records the declaration at compile time; the storage is allocated and (re)initialised each time control enters the scope. Unlike `local`, `my` does not temporarily replace a package variable

- it creates a genuinely new variable that exists only while the scope is live and is invisible outside it, including to any subroutines the scope calls.

Synopsis

```
my EXPR
my TYPE EXPR
my EXPR : ATTRS
my TYPE EXPR : ATTRS
```

A single variable may be written without parentheses; a list of two or more must be parenthesised:

```
my $x;
my ($a, $b, @rest);
my Foo $obj;
my $x : shared = 0;
```

What you get back

`my` is a declaration, not a value-producing operator in the ordinary sense. In rvalue position it yields its variable (or the list of variables) as an lvalue you can assign to. The common idioms are:

```
my $x = $expr;           # scalar context on RHS
my ($first, @rest) = @list; # list context on RHS; destructuring
my @copy = @src;         # list context; @copy gets the elements
my %h = @pairs;          # list context; pairs populate the hash
```

When no initialiser is given, each declared variable starts as `undef` (scalar), empty list (array), or empty hash.

Scope and lifetime

- **Block scope.** A `my` variable is visible from the point immediately after its declaration to the end of the innermost enclosing block, file, `eval`, `do`, `require`, or `use`'d file.
- **Control-expression scope.** The controlling expression of `if` / `unless` / `elsif` / `else`, `while` / `until`, `for` / `foreach` is part of the scope of the declaration it contains. In `while (my $line = <$fh>) { ... }` the variable `$line` is live throughout the loop body and its `continue` block, but not beyond.
- **Fresh storage on every entry.** Each time control re-enters the scope, the variable is created anew. This is the key difference from `state`, which preserves its value across entries.
- **Closures capture storage, not values.** An anonymous sub created inside the scope keeps that particular instance of the variable alive for as long as the sub is reachable.

```
sub make_counter {
    my $n = 0;
    return sub { ++$n };    # closes over this $n
}
my $c = make_counter();
$c->(); $c->();              # 1, 2
```

List form and destructuring

With parentheses, `my` supplies list context to the right-hand side and deconstructs the result:

```
my ($sec, $min, $hour) = localtime;
my ($head, @tail)      = @list;    # @tail absorbs the rest
my ($x, $y)            = ($y, $x); # swap via list assignment
```

Use `undef` as a placeholder to skip a position:

```
my (undef, $min, $hour) = localtime; # discard seconds
```

Without parentheses, `my $foo` is a single-variable declaration and the comma operator applies to whatever follows:

```
my $foo, $bar = 1;           # WRONG: declares $foo only;
                             # $bar is the package variable
my ($foo, $bar) = (0, 1);   # Right.
```

Context traps

`my` does not change how the right-hand side is evaluated - it only modifies the variable on the left. A scalar `my $foo` imposes scalar context; a parenthesised `my (...)` imposes list context:

```
my $line = <$fh>;           # scalar context: one line
my ($line) = <$fh>;         # list context: first of a list
my @lines = <$fh>;          # list context: all remaining lines
```

The classic mistake is writing `my ($foo) = <$fh>` when a single line was wanted - the parentheses turn on list context and the filehandle returns a one-element list; the value happens to be the first line, but the surrounding context is wrong for anything that follows.

my vs local vs our vs state

Four declarations, four different meanings:

- **my** - introduces a **new lexical** variable. Not visible to called code. Fresh storage each entry.
- **local** - does **not** declare anything. It saves the current value of an **existing package** variable and restores it when the enclosing scope exits. Visible to called code through dynamic scope. Use it for the handful of magical globals (`$_`, `$/`, `$\`, `$,`, `$|`, `%ENV`, ...) that cannot be lexicalised.
- **our** - declares a **lexical alias** for a package (global) variable. The storage lives in the package's symbol table; `our` just lets you refer to it without the package prefix for the rest of the lexical scope. Satisfies `use strict 'vars'`.
- **state** - like `my` in visibility, but the storage **persists** across re-entries to the scope. Initialisation runs once; subsequent entries leave the value alone.

```
package Counter;
our $total = 0;           # package global; $Counter::total
my $cache;                # lexical; invisible outside this file
state $calls = 0;         # lexical; keeps its value across calls
sub tick {
    local $\ = "\n";      # temporarily override output separator
    ++$calls;
    print ++$total;
}
```

Critical point: `my $var` does **not** mask a package variable with the same name. The package variable remains accessible through its fully qualified form while the lexical is

in scope:

```
package main;
our $x = 10;
my $x = 20;
print "$x and $::x\n";      # "20 and 10\n"
```

TYPE and attributes

Both forms are part of the language but their semantics are still evolving (upstream wording).

- **TYPE** - a bareword class name, a constant declared via `use constant`, or `__PACKAGE__`. Historically bound to the `fields` pragma; for ordinary use it serves as a hint and does not enforce a type at runtime.

```
my Foo $obj = Foo->new;
```

- **ATTRS** - a colon-introduced attribute list handled by the `attributes` pragma (and `Attribute::Handlers` since 5.8.0).

```
my $x : shared;
my @buf : Locked;
```

See L<perlsub/"Private Variables via my()"> upstream for the authoritative wording.

Declaration is not yet in scope on its own RHS

The newly declared variable is not visible until **after** the declaring statement finishes. That makes this idiom well-defined:

```
my $x = $x;      # new $x initialised from the old $x
```

and this condition dependent on a pre-existing outer `$x`:

```
my $x = 123 and $x == 123; # false unless the old $x was 123
```

Within a single statement, any further mentions of the name refer to the pre-declaration binding (or raise a `strict 'vars'` error if no such binding exists):

```
our $x = 2;
foo($x, my $x = $x + 1, $x); # foo() receives (2, 3, 2)
```

Examples

Typical subroutine entry - name the arguments:

```
sub distance {
    my ($x1, $y1, $x2, $y2) = @_;
    return sqrt( ($x2-$x1)**2 + ($y2-$y1)**2 );
}
```

Loop-local index with `for my`:

```
for my $i (1 .. 10) {
    # $i is fresh each iteration; invisible after the loop
}
```

File-scoped private state - the variable exists for the life of the interpreter but is unreachable from outside this file:

```
my $cache = {};
sub lookup { $cache->{ $_[0] } ||= _compute($_[0]) }
```

Private subroutine via a lexical coderef:

```
my $helper = sub { ... };
$helper->($x);
```

Shadowing warning - redeclaring in the same scope emits a shadow-category warning under `use warnings`:

```
use warnings 'shadow';
my $x = 1;
my $x = 2;                # warning: "my" variable $x masks ...
```

Edge cases

- **Only alphanumeric identifiers.** Magical built-ins such as `$/`, `$_`, `$\`, `%ENV`, `@ARGV` cannot be lexicalised. Use `local` to scope them dynamically.
- **No package qualification.** `my $Foo::bar` is a syntax error: my variables are not package-owned and cannot carry a `::` qualifier.
- **Missing parentheses around a list.** `my $a, $b` declares only `$a`; `$b` is the package variable. Always parenthesise two or more names.
- **List-context surprise.** `my ($line) = <$fh>` forces list context on the read; the RHS is a one-element list, not a scalar. Drop the parentheses if you want scalar semantics.
- **foreach without my.** `for $i (...)` localises `$i` dynamically in the manner of `local`. Prefer `for my $i (...)` to confine the index lexically.
- **Shadowing in loops.** A `my` inside a loop body declares a fresh variable every iteration. If that is unwanted (e.g. a counter that should survive iterations), declare it outside the loop or use `state`.
- **Closure capture.** A closure over a `my` variable keeps that particular instance alive. Different calls to an enclosing sub produce independent closures over independent variables.
- **use strict 'vars' interaction.** `my` satisfies `strict-vars` within its scope; variables without a declaration must be `our`, fully qualified, or predeclared with `use vars`.

- **Compile-time vs run-time.** The name binding happens at compile time; the initialiser runs at run time, and re-runs on each scope entry. Expensive initialisers inside hot loops pay that cost every iteration.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `our` - lexical alias for a package variable; satisfies `strict 'vars'` without creating new storage
- `local` - dynamically scoped save/restore of an existing package variable; the tool for magical globals
- `state` - lexical like `my`, but storage persists across scope re-entries
- `undef` - placeholder in a parenthesised `my` list, and the initial value of an uninitialised scalar
- `use strict` - makes `my` / `our` / fully-qualified names mandatory, catching accidental globals
- `attributes` - pragma that interprets the `: ATTRS` portion of a `my` declaration

Control flow

next

Start the next iteration of the enclosing loop immediately.

`next` is Perl's equivalent of C's `continue` statement: it abandons the rest of the current iteration's body and jumps to the loop's iteration step. If the loop has a `continue` block, that block runs first, then the controlling expression is re-tested (or the next element is taken for `foreach`), and execution resumes at the top of the body. With a label, `next LABEL` targets a specific enclosing loop rather than the innermost one.

Synopsis

```
next
next LABEL
next EXPR
```

What you get back

Nothing. `next` is a flow-control operator, not an expression - it unwinds out of the current iteration and therefore never yields a value to its caller. Attempting to use it in a value-producing position (for example as the right-hand side of an assignment inside `eval {}`, `sub {}`, or `do {}`) is a misuse: the surrounding construct never completes normally, so there is no return value to observe.

Loop targeting

Without a label, `next` refers to the **innermost enclosing loop** (while, until, for, foreach, or a bare block, which counts as a one-shot loop). With a label it refers to the named loop, no matter how deeply nested the `next` sits inside other loops:

```
OUTER: for my $row (@grid) {
    for my $cell (@$row) {
        next OUTER if $cell eq 'skip-row';
        process($cell);
    }
}
```

`next EXPR`, available since Perl 5.18, evaluates `EXPR` at run time and uses its string value as the label name. Otherwise it behaves exactly like `next LABEL`:

```
my $target = pick_label();
next $target;                                # same as: next F00 if $target eq
↪ "F00"
```

Use the runtime form sparingly - a static `next LABEL` is easier to read and to verify by eye.

Interaction with `continue` blocks

A loop's `continue` block runs at the end of every iteration, whether that iteration ended normally or via `next`. This is the whole point of having a `continue` block: it groups the bookkeeping that must happen on every pass.

```
my $i = 0;
LINE: while (my $line = <$fh>) {
    next LINE if $line =~ /\s*#/; # skip comments
    process($line);
}
continue {
    $i++;                        # runs even for skipped lines
}
```

`last` skips the `continue` block; `next` does not. This is the simplest distinguishing property of the two, and the reason `continue` exists as its own construct rather than being baked into the loop body.

Examples

Basic while loop - discard comment lines, process the rest:

```
while (my $line = <$fh>) {
    next if $line =~ /\s*#/;
    next if $line =~ /\s*$/;      # blank
    handle($line);
}
```

foreach with an accumulator - skip entries that fail a predicate:

```
my $total = 0;
for my $n (@values) {
    next unless defined $n;
    next if $n < 0;
    $total += $n;
}
```

Nested loops with a label - move on to the next outer row as soon as we spot a sentinel in the inner scan:

```
ROW: for my $row (@grid) {
    for my $cell (@$row) {
        next ROW if $cell eq 'END';
        count($cell);
    }
}
```

`continue` block that always runs, even on next:

```
my @kept;
ITEM: for my $x (@input) {
    next ITEM if $x->{hidden};
    push @kept, $x;
}
continue {
    log_seen($_);                # every $x is logged, kept or
    ↪not                           ↪
}
```

Escaping a bare block - a standalone `{ ... }` is a one-shot loop, so `next` jumps past the remaining statements and out of it:

```
{
    last_chance() or next;        # bare block: next == fall out
    commit();
}
```

Edge cases

- **Illegal inside `grep` / `map` / a `do { }` value block.** These constructs are not loops - their block is evaluated for its value, not iterated in the loop-control sense. Using `next` to exit them is a misuse and is called out explicitly in upstream perlfunc. Use a conditional expression instead:

```
my @kept = grep { wanted($_) } @xs;           # not: grep {
    ↪next unless ... }
my @mapped = map { transform($_) // () } @ys; # empty list ==
    ↪skip
```

- **Not a `return`.** `next` cannot deliver a value out of a value-returning block such as

`eval { }`, `sub { }`, or `do { }`. If you find yourself reaching for `next` to leave such a block, you want `return`, or a conditional that produces the value you need.

- **A bare block is a one-shot loop.** `{ ... }` with no `while` / `for` prefix is semantically a loop that runs exactly once, so `next` exits it. This is occasionally useful for early-out patterns, but `last` is the more common choice because it makes the intent clearer.
- **Unknown label is a fatal error.** `next NOSUCH` dies at run time with `Can't "next" outside a loop block when no enclosing loop carries that label`.
- **`continue` runs on `next`, not on `last`.** This is the usual cause of surprise when refactoring: moving bookkeeping out of the loop body into `continue` changes behaviour only for `last`, not for `next`.
- **Precedence and parsing quirks.** Unlike most named operators, `next` has the same precedence as assignment, and it is exempt from the looks-like-a-function rule. That means `next ("foo") . "bar"` passes `"foobar"` to `next`, not just `"foo"`. Parenthesise defensively when the label name comes from an expression:

```
next(($cond ? "A" : "B"));
```

- **Threads, signals, destructors.** `next` unwinds the current iteration's lexical scope: `my` variables go out of scope, `local` values are restored, and `DESTROY` fires on scope-exit objects, just as if the iteration had ended normally. Pending `$SIG{...}` handlers are not affected.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `last` - exit the enclosing loop entirely, skipping any `continue` block
- `redo` - restart the current iteration without re-testing the condition and without running `continue`
- `return` - the right tool for leaving `sub { }`, `eval { }`, or `do { }` with a value; `next` cannot do this
- `grep` - filter a list; use a conditional expression, not `next`, to drop elements
- `map` - transform a list; return the empty list `()` from the block to drop an element

Modules

no

The compile-time inverse of `use` - call a module's `unimport` method to turn off what `use` turned on.

`no MODULE LIST` is handled by the parser exactly like `use MODULE LIST`, except it calls `MODULE->unimport(LIST)` instead of `MODULE->import(LIST)`. Both run at compile time, both are lexically scoped, and both require the module to already be loaded. The

common case is switching a pragma off inside a block - no `strict 'refs'`, no `warnings 'uninitialized'`, no `feature 'switch'` - without touching the enclosing file.

Synopsis

```
no MODULE LIST
no MODULE
no MODULE VERSION LIST
no MODULE VERSION
```

What you get back

`no` is a declaration, not an expression. It has no useful return value and cannot appear where a value is expected. The effect is a side effect on the compile-time state of the enclosing lexical scope.

How it expands

The parser rewrites

```
no MODULE LIST;
```

into the rough equivalent of

```
BEGIN {
    require MODULE;
    MODULE->unimport(LIST);
}
```

- the same shape as `use`, with `unimport` swapped in for `import`. Consequences that follow from this:
- **Compile time.** The `BEGIN` block runs as soon as the parser reaches the `no` statement, before any later code in the file is compiled. This is why `no strict 'refs'` takes effect on the statements that follow it in the same file, not at runtime.
- **Lexical scope.** Pragmas that honour lexical scope (`strict`, `warnings`, `feature`, `integer`, `utf8`, and the other core pragmas) see their “off” effect confined to the innermost enclosing block, `eval`, or file. Leaving the block restores the previous state.
- **require still happens.** `no MODULE` loads `MODULE` if it has not been loaded yet, then calls `unimport` on it. You do not need a matching `use` earlier in the file.
- **unimport is mandatory.** If the package has no `unimport` method, the call dies at compile time with `Can't locate object method "unimport" via package "MODULE"`. Most non-pragma modules do not define `unimport` - `no` is really only useful for modules that advertise it.

Examples

Turn off a single warning category inside a block:

```
use warnings;

for my $row (@rows) {
    no warnings 'uninitialized';
    print join(",", @$row), "\n";    # empty fields render as ""
}
# 'uninitialized' warnings are back on here
```

Relax strict 'refs' for one statement that needs symbolic dereferencing:

```
use strict;

sub install {
    my ($name, $code) = @_;
    no strict 'refs';
    *{"main::$name"} = $code;        # symbolic typeglob assignment
}
```

Disable an experimental feature warning while using the feature:

```
use feature 'signatures';
no warnings 'experimental::signatures';

sub greet ($who) { "hello, $who\n" }
```

Roll back a default feature bundle to plain Perl semantics:

```
use v5.36;                                # enables strict, warnings, say, ↵
↵signatures...
no feature 'switch';                      # ...but not given/when
```

Turn off a user pragma that provides its own unimport:

```
use My::Pragma qw(loud);
{
    no My::Pragma;                        # My::Pragma->unimport called ↵
    ↵here
    quiet_code();
}
```

Edge cases

- **no VERSION is a compile error.** `use VERSION` exists as a compile-time version check against the running interpreter; there is no sensible inverse, so `no 5.36;` dies with `No such class 5.36`. The version check only goes one way.
- **no MODULE VERSION does parse.** It requires `MODULE` to be at least `VERSION` (same check as `use`), then calls `MODULE->unimport(LIST)`. The version check is about the loaded module, not the interpreter.

- **Module without `unimport`.** Dies at compile time. There is no fallback to a no-op; if you are writing a module and want no to be a silent no-op, define an empty sub `unimport {}`.
- **Bareword `MODULE` only.** Like `use`, the module name must be a bareword known at compile time. `no $name;` is a syntax error - the parser needs the literal token to decide which `unimport` to call.
- **Does not undo `use`.** `no Foo` calls `Foo->unimport`, which by convention reverses what `Foo->import` did to the current lexical scope - but that is a convention, not a language guarantee. A sloppy `unimport` can leave state behind, and side effects from `require Foo` (BEGIN blocks, @INC hooks, installed subs in other packages) are **not** rolled back.
- **Effect ends at the enclosing block.** For lexical pragmas, the “off” state stops at the next `}`, `eval` boundary, or end of file. Code in a different file that requires or uses this one is unaffected.
- **`import/unimport` are just method calls.** `MODULE->unimport(LIST)` runs exactly like any other class method - it sees `MODULE` as the first argument and `LIST` after. The method can inspect the caller via `caller` to find out which package to modify.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `use` - the positive form; `no` is literally `use` with `unimport` in place of `import`
- `require` - the runtime load step that `no` performs before calling `unimport`
- `package` - declares the current package, which is the scope a pragma’s `unimport` typically modifies
- `import` - the conventional counterpart method; `no` calls `unimport`, `use` calls `import`
- BEGIN - the compile-time phase in which `no` runs
- `caller` - how an `unimport` method identifies the package it should affect

[SCALARs and strings](#) · [Numeric functions](#)

oct

Interpret a string as a number written in octal, hexadecimal, or binary, and return the resulting integer.

The name is misleading: `oct` is really a *radix-prefix* parser. It inspects the leading characters of `EXPR` and picks the base from them - `0x/x` for hexadecimal, `0b/b` for binary, `0o/o` (or no prefix at all) for octal. If `EXPR` is omitted, `oct` works on `$_`. Think of it as the inverse of `sprintf "%o" / "%x" / "%b"` for the common case where you have the textual form and need the number.

Synopsis

```
oct EXPR
oct                # operates on $_
```

What you get back

A non-negative integer. The numeric value of EXPR interpreted in whichever base its prefix selected (octal by default). The result is always ≥ 0 - oct does not parse a leading sign and does not return floating-point values.

An empty string, a string with no recognised digits, or `undef` returns 0.

Prefix rules

The leading characters of EXPR (after any leading whitespace, which is silently skipped) determine the base:

Prefix	Base	Example input	Result
0x or x	16 (hex)	"0xff"	255
0b or b	2 (binary)	"0b1010"	10
0o or o	8 (octal)	"0o755"	493
(none)	8 (octal)	"755"	493
(none, leading 0)	8 (octal)	"0755"	493

The 0o/o octal prefix was added in Perl 5.33.5 to match integer literal syntax; older forms (0755 and a bare 755) have always been octal.

Underscores between digits are ignored, so values copied straight from Perl integer literals parse cleanly:

```
oct("0xDEAD_BEEF")    # 3735928559
oct("0b1111_0000")    # 240
oct("07_55")           # 493
```

Examples

Decode a permission string typed by a user - the whole reason oct exists. The string "755" is **not** the number 755; it is the octal representation of the number 493. `chmod` expects the number:

```
my $mode = "0755";
chmod oct($mode), "script.sh";           # chmod 0755
```

```
my $mode = "755";                       # no leading zero - still
oct                                       # no leading zero - still
chmod oct($mode), "script.sh";           # same result
```

Parse hex and binary literals read from a config file or a command line:

```
my $mask = oct("0xff00");           # 65280
my $flags = oct("0b1010_0101");      # 165
```

Handle arbitrary radix-prefixed strings - decimal, octal, hex, or binary - in one expression. This is the classic oct-as-dispatcher idiom: a leading 0 (in any of its forms) means “not decimal”, so route through oct; otherwise keep the value as-is:

```
my $n = $val =~ /^0/ ? oct($val) : $val;
```

Default-argument form operates on `$_`, handy inside `map`:

```
my @perms = map { oct } @mode_strings;
```

Round-trip with `sprintf` to go the other direction:

```
my $perm = (stat $file)[2] & 07777;
my $text = sprintf "%o", $perm;      # e.g. "644"
my $back = oct($text);               # back to the same integer
```

Edge cases

- **Empty or undef input:** `oct("")` and `oct(undef)` both return 0. `undef` triggers an uninitialized warning under `use warnings`.
- **No prefix defaults to octal:** `oct("12")` is 10, not 12. This is a frequent surprise - if your strings might be decimal, test for a leading 0 first (see the idiom above) or do not use `oct` at all and parse explicitly.
- **Underscores are ignored** wherever they appear between digits. `oct("0x_de_ad")` is 57005. Leading and trailing underscores are also silently accepted.
- **Invalid characters terminate parsing:** parsing stops at the first character that does not belong to the selected base; the rest of the string is discarded without a warning. `oct("0x1g2")` returns 1, `oct("755.5")` returns 493, `oct("12abc")` returns 10 (octal, stopped at a), `oct("0b102")` returns 2 (binary 10, stopped at 2).
- **Leading whitespace is skipped silently.** A leading + or - is **not** a sign - it is a non-digit, so parsing returns 0. `oct` cannot produce a negative number.
- **Floating-point and exponent notation are not understood.** There is no `0x1.8p3` support; the `.` terminates parsing.
- **Case of prefix and digits is insensitive:** `0X`, `0B`, `0O`, and hex digits a-f/A-F are all accepted.
- **Overflow:** values larger than the platform integer silently promote to floating point, losing low-order precision for very large hex/binary inputs.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `hex` - forces base 16 regardless of prefix; use it when the input is *always* hex and you do not want oct's autodetection
- `int` - truncate a numeric value toward zero; unrelated to radix parsing but often reached for by accident
- `chmod` - the canonical consumer of oct; takes a numeric mode, so string-form modes must go through oct first
- `sprintf` - the inverse direction: format %o, %x, %b turn an integer into its octal/hex/binary textual form

Filehandles, files, directories

open

Associate a filehandle with a file, a command, or an in-memory scalar.

`open` is the primary way a Perl program acquires a filehandle. The three-argument form - `open $fh, MODE, EXPR` - is the one to use in new code: the mode is stated explicitly, the filename cannot be reinterpreted as a shell pipeline, and leading or trailing whitespace in `EXPR` is preserved verbatim. The older two-argument form exists for compatibility and for the “magic” filename interpretation it enables; do not feed it untrusted input.

Synopsis

```

open my $fh, "<", $path          or die "read $path: $!";
open my $fh, ">", $path          or die "write $path: $!";
open my $fh, ">>", $path        or die "append $path: $!";
open my $fh, "+<", $path        or die "rw $path: $!";
open my $fh, "<:encoding(UTF-8)", $path or die $!;
open my $mem, ">", \ $buf        or die $!;    # in-memory
open my $pipe, "-|", "ls", "-la" or die $!;    # read from cmd
open my $pipe, "|-", "gzip", ">$f" or die $!;    # write to cmd

```

What you get back

1 on success; `undef` on failure, with `$!` set to the OS error. For the pipe forms (`| - / - |`), the successful return value is the pid of the child process - useful when you need to `waitpid` on it later. After a one- or two-argument pipe open on the literal command `-`, `open` returns twice (parent gets the child pid, child gets 0); use `defined($pid)` or `//` to distinguish the two.

Always check the return value. A failed `open` leaves the filehandle closed, and every subsequent read or write on it will set `$!` to `Bad file descriptor` and emit a warning under `use warnings`.

```
open my $fh, "<", $path
or die "open $path: $!";
```

Global state it touches

- `$!` - set on failure to the OS-level error.
- `${^OPEN}` - default I/O layers installed by the `open` pragma or the `-CioD` switch. These apply only when `MODE` in the three-argument form contains no explicit layer; a bare `<` or `>` honours them, `<:raw` or any other explicit layer suppresses them.
- `$^F` - highest file descriptor not marked close-on-exec; the new fd inherits the close-on-exec flag based on this.
- Selected filehandle via `select` - unchanged by `open`, but becomes relevant the moment you `print` to the new handle without naming it.
- For pipe forms, `$?` and `${^CHILD_ERROR_NATIVE}` are set when the handle is later closed.

Modes

The second argument names an I/O role. The common cases:

Mode	Meaning	fopen(3) equivalent
<code><</code>	read	<code>r</code>
<code>></code>	write, truncate or create	<code>w</code>
<code>>></code>	append, create if missing	<code>a</code>
<code>+<</code>	read and write, no truncation	<code>r+</code>
<code>+></code>	read and write, truncate first	<code>w+</code>
<code> -</code>	pipe to command (we write)	<code>-</code>
<code>- </code>	pipe from command (we read)	<code>-</code>
<code><&, >&, <&=, >&=</code>	dup or fdopen an existing handle	<code>-</code>

Newly created files get permissions `0666 & ~umask`. `+<` is almost always what you want for read-write; `+>` clobbers the file first.

I/O layers attach after the mode: `<:encoding(UTF-8)`, `>:raw`, `>>:crlf`, `<:encoding(UTF-8):crlf`. An explicit layer suppresses `${^OPEN}` defaults; a lone colon (`<:`) falls back to the OS default (`:raw` on Unix, `:crlf` on Windows).

Examples

Read a UTF-8 text file line by line:

```
open my $fh, "<:encoding(UTF-8)", "notes.txt"
or die "open notes.txt: $!";
while (my $line = <$fh>) {
    chomp $line;
    # ... use $line ...
}
```

(continues on next page)

(continued from previous page)

```
}
close $fh or warn "close: $!";
```

Write a binary file with no encoding translation:

```
open my $out, ">:raw", "image.bin"
    or die "open image.bin: $!";
print $out $bytes;
close $out or die "close: $!";
```

In-memory filehandle - `print` into a scalar, useful for capturing output that expects a real handle:

```
my $buf = "";
open my $mem, ">", \$buf or die $!;
print $mem "hello\n";
close $mem;
# $buf now contains "hello\n"
```

Read from a command, list form so arguments are not subject to shell metacharacter interpretation:

```
open my $ls, "-|", "ls", "-la", $dir
    or die "ls: $!";
while (<$ls>) { print }
close $ls;                                # $? holds the exit status
```

Temporary anonymous file - pass `undef` as the third argument with a read-write mode, then `seek` back to the start to read what you wrote:

```
open my $tmp, "+>", undef or die $!;
print $tmp "scratch data\n";
seek $tmp, 0, 0;
my $line = <$tmp>;
```

Edge cases

- **Two-argument “magic” open:** `open $fh, $name` honours leading `<`, `>`, `>>`, a trailing `|`, and strips surrounding whitespace from `$name`. Fine for a literal filename you typed yourself; **never** for a filename that came from user input, the network, or an environment variable. Use the three-argument form.
- **Two-argument `-:`** `open $fh, "<-"` or `open $fh, "-"` opens STDIN; `open $fh, ">-"` opens STDOUT. Three-argument equivalents are `open $fh, "<", "-"` and `open $fh, ">", "-"`.
- **Reopening STDOUT / STDERR:** close them first, then reopen against the new target. `open STDOUT, ">", \$buf` redirects all subsequent `print` STDOUT output into `$buf`.
- **Bareword filehandles** (`open FH, ...`) are package globals and are disabled under use `v5.36` or later via the `bareword_filehandles` feature. Prefer a lexical

scalar.

- **Layers and `${^OPEN}`:** any explicit layer, even `:`, disables the defaults from the open pragma. If you rely on a pragma default, omit the colon entirely (`<`, not `<:`).
- **Pipe to a command with shell metacharacters:** the two-argument pipe form (`open $fh, "|cat -n | lpr"`) runs through `/bin/sh -c`. The three-argument list form (`open $fh, "|-", "cat", "-n"`) does not and is the safer default.
- **In-memory scalar strings:** treated as octet strings. Without truncation, the scalar may not contain code points above `0xFF`.
- **Automatic close on scope exit:** a lexical filehandle closes when its refcount hits zero, but the implicit close does not report errors. For files you wrote to, close explicitly and check the return value.
- **open vs `sysopen`:** `open` wraps `fopen(3)` semantics with PerlIO layers on top. For direct `open(2)` flags like `O_EXCL` or `O_NOFOLLOW`, use `sysopen`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `close` - release a filehandle; check its return value for buffered writes and pipe commands
- `sysopen` - open with raw `open(2)` flags when you need `O_EXCL`, `O_NOFOLLOW`, or similar precision
- `binmode` - push or replace I/O layers on an already open handle
- `read` - fixed-size byte reads from the handle open returned
- `readline` - record-at-a-time reads (what `<$fh>` calls)
- `fileno` - the underlying OS file descriptor, for `flock`, `select`, or `fcntl`
- `$!` - OS error set when open returns `undef`

Filehandles, files, directories

opendir

Open a directory for reading.

`opendir` attaches `DIRHANDLE` to the directory named by `EXPR` so the directory's entries can be read with `readdir` and the position manipulated with `telldir`, `seekdir`, and `rewinddir`. When you're done, `closedir` releases the handle. `DIRHANDLE` lives in a namespace distinct from file handles - a bareword you have already used with `open` is free to reuse here, and vice versa.

Synopsis

```
opendir DIRHANDLE, EXPR
opendir my $dh, EXPR
```

What you get back

1 on success, 0 on failure. On failure `$!` holds the reason - the directory does not exist, is not a directory, is not readable, or the process has hit its open-file limit. Always check the return; a silent failure here turns into a mysteriously empty `readdir` later.

```
opendir my $dh, $path
    or die "opendir $path: $!";
```

Global state it touches

- `$!` - set to the system error string on failure, left untouched on success. Capture it before any other system call disturbs it.

Handle semantics

Three forms of DIRHANDLE are accepted, and the choice matters:

- **Bareword** - `opendir DIR, $path` creates a package-global dirhandle named `DIR` in the current package. It stays open until you `closedir` it explicitly or the program exits. Avoid in new code; the global namespace is shared across the whole package.
- **Undefined scalar** - `opendir my $dh, $path` autovivifies `$dh` into a reference to a fresh anonymous dirhandle. When `$dh` goes out of scope and its refcount drops to zero, the handle is closed automatically. This is the form to use.
- **Expression** - any expression whose value is usable as an indirect dirhandle (typically a real dirhandle name held in a variable). The handle identity comes from the expression's value, not its syntactic form.

Dirhandles and filehandles share the same underlying I/O object: a given object can be open as one or the other at a time, never both. Passing a dirhandle to `read` or a filehandle to `readdir` is a usage error, not a silent read of zero bytes.

Examples

The canonical `opendir` / `readdir` / `closedir` idiom:

```
opendir my $dh, $dir
    or die "opendir $dir: $!";
while (my $entry = readdir $dh) {
    next if $entry eq '.' or $entry eq '..';
    print "$dir/$entry\n";
}
closedir $dh;
```

Read every entry into a list in one go. `readdir` in list context returns all remaining entries:

```
opendir my $dh, '.' or die "opendir : $!";
my @entries = grep { !/^\.\/?$/ } readdir $dh;
closedir $dh;
```

Rely on lexical scope for cleanup - no explicit `closedir` needed:

```
sub list_dir {
    my ($path) = @_;
    opendir my $dh, $path or return;
    return grep { !/^\.\/?$/ } readdir $dh;
    # $dh goes out of scope here, handle is closed
}
```

Handle a missing-or-unreadable directory without dying:

```
if (opendir my $dh, $path) {
    while (defined(my $e = readdir $dh)) { ... }
    closedir $dh;
} else {
    warn "skipping $path: $!";
}
```

Revisit the start of a directory with `rewinddir` rather than reopening:

```
opendir my $dh, $dir or die $!;
my @first_pass = readdir $dh;
rewinddir $dh;
my @second_pass = readdir $dh;
closedir $dh;
```

Edge cases

- **Failure sets \$!, not \$@.** `opendir` reports through `$!` like the rest of the I/O family; `$@` is untouched.
- **Reusing an open dirhandle.** `opendir` on a handle that is already open silently closes the old directory before opening the new one. Prefer explicit `closedir` when the code is meant to be readable.
- **Relative paths.** `EXPR` is resolved against the current working directory at the moment of the call. If you `chdir` later, a relative path you captured earlier may no longer mean the same thing - the already-opened dirhandle keeps pointing at the right inode regardless.
- **Symlinks.** `opendir` follows symlinks by default: opening a link to a directory opens the target. The entries returned by `readdir` are the target's entries, not the link.
- **Bareword vs lexical lifetime.** A bareword dirhandle is global and outlives the enclosing block; a lexical `my $dh` is cleaned up when the scope ends. Mixing the

two by accident (`opendir DH, $path` inside a sub) leaks dirhandles on repeated calls.

- **File vs dir handle mix-up.** You cannot `print` to a dirhandle nor `readdir` from a filehandle. The error surfaces at the later call, not at `opendir`.
- **glob vs opendir.** `glob` expands shell-style patterns and returns paths; `opendir` enumerates raw directory entries (including `.` and `..`) without interpretation. Use `glob` when you want pattern matching; use `opendir` when you want every entry exactly as the filesystem stores it.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `readdir` - read the next entry (or all remaining entries) from a dirhandle opened with `opendir`
- `closedir` - release a dirhandle; implicit when a lexical `$dh` goes out of scope
- `rewinddir` - reset the read position to the start without reopening
- `seekdir` - reposition the dirhandle to a token previously returned by `telldir`
- `telldir` - current read position as an opaque token suitable for `seekdir`
- `glob` - pattern-based alternative when you want shell-style filename expansion instead of raw directory enumeration

SCALARs and strings

ord

Return the Unicode code point of the first character of a string.

`ord` takes a string, looks at its first character, and returns that character's numeric code point as an integer. Note *character*, not *byte* - if the string is a wide-character string containing the letter `é` (U+00E9), `ord` returns 233, regardless of how many bytes `é` occupies in memory. For the empty string, `ord` returns 0. If `EXPR` is omitted, `ord` operates on `$_`.

For the reverse direction - turning a code point back into a character - see `chr`.

Synopsis

```
ord EXPR
ord
ord($str)
```

What you get back

A non-negative integer. For an ASCII input it is in the range 0..127; for a Latin-1 input 0..255; for a general Unicode string anything up to 0x10FFFF. `ord` never returns a negative number and never returns a non-integer. Only the first character of the argument is consulted - trailing characters are ignored. To walk every code point of a string, combine with `split //` or `unpack "U*"`.

Global state it touches

With no argument, `ord` reads `$_`. It neither writes nor reads any other special variable. The interpretation of the input as characters (not bytes) depends on whether the scalar is internally flagged as UTF-8 - see *Edge cases* below.

Examples

Basic ASCII lookup:

```
print ord("A");           # 65
print ord("0");           # 48
print ord("\n");          # 10
```

Empty string returns zero, matching the “no first character” contract:

```
print ord("");            # 0
```

Default-argument form inside a while loop over `$_`:

```
while (<STDIN>) {
    last if ord == 4;      # stop on EOT (Ctrl-D) as first char
    print;
}
```

Decode a wide character. `ord` sees the character, not the bytes:

```
use utf8;
print ord("é");           # 233          (U+00E9)
print ord("€");           # 8364         (U+20AC)
print ord("😊");           # 128512      (U+1F600)
```

Walk every code point in a string - useful for debugging encoding surprises:

```
use utf8;
my $s = "héllö";
print join(",", map { ord } split //, $s), "\n";
# 104,233,108,108,111
```

Pair with `chr` for a no-op round-trip on any single character:

```
my $c = "Z";
print chr(ord($c)) eq $c ? "same\n" : "changed\n";  # same
```

Edge cases

- **Character vs byte semantics:** `ord` returns the code point of the first *character*, which for a UTF-8-flagged scalar is not the same as the first *byte*. `ord("\x{100}")` returns 256, even though that character occupies two bytes in memory. To inspect the first byte instead, force byte semantics:

```
use bytes;
print ord("\x{100}");      # 196    (first UTF-8 byte, 0xC4)
no bytes;
print ord("\x{100}");      # 256
```

Reach for `use bytes` only when you genuinely need byte-level inspection; it is a local, surgical pragma.

- **Non-UTF-8 scalar containing high bytes:** if the scalar is a plain byte string (no UTF-8 flag) and its first byte is `0xC4`, `ord` returns 196, not 256. The byte is interpreted as Latin-1. Whether a given scalar is UTF-8-flagged depends on how it was built - use `utf8` in source, `decode_utf8`, a `:utf8` I/O layer, `chr` of a value above 255, and so on.
- **undef:** `ord(undef)` returns 0 and, under `use warnings`, emits `Use of uninitialized value in ord`.
- **Numeric argument:** `ord(65)` stringifies 65 to "65" first, then takes the first character - so it returns `ord("6") == 54`, not 65. This surprises people; if you have an integer and want to round-trip through a character, use `chr` first or skip `ord` entirely.
- **Multi-character argument:** only the first character matters. `ord("ABC")` returns 65; the B and C are never consulted.
- **Surrogate and non-character code points:** `ord` will happily return `0xD800..0xDFFF` or `0xFFFE/0xFFFF` if those appear in the input. Perl does not refuse to store them; validation, if needed, is the caller's job.
- **Maximum value:** on a standard build, `ord` can return up to `0x7FFFFFFF` (31-bit) for a string produced by `chr` of a value in that range. Real-world Unicode text tops out at `0x10FFFF`.
- **Precedence:** the unary form `ord $x` binds tighter than `,` but looser than most arithmetic operators, so `ord $x + 1` parses as `ord($x) + 1`, not `ord($x + 1)`. Parenthesise when in doubt.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `chr` - the inverse operation; turns a code point back into a one-character string
- `sprintf` - use `%c` to format an integer as its character, or `%x / %04x` to render an `ord` result in hex

- `unpack` - unpack "U*" for every code point of a string; unpack "C*" for every byte
- `lc` / `uc` - case-fold a character before taking its code point when you want case-insensitive comparisons
- `$_` - the default argument when `ord` is called without one

Scoping

`our`

Declare a lexically scoped alias to a package variable.

`our` does **not** create a new variable. It introduces a name into the current lexical scope that refers to the package variable of the same name in the current package. For the duration of that scope, the unqualified name is legal under `use strict 'vars'`, and every read or write through it hits the underlying package variable.

If the package variable did not exist yet, `our` does not conjure it either - it simply names it. Package variables spring into existence when first assigned to, and an `our` declaration makes that first assignment possible without a fully qualified name.

Synopsis

```
our $x;  
our ($a, $b, $c);  
our @EXPORT_OK = qw(foo bar);  
our $VERSION : shared = '1.00';
```

What you get back

`our` returns its list of variables, so it can appear in any expression position where `my` can:

```
(our $x, our $y) = (1, 2);  
our $count = 0;
```

As a declaration it has no separate return value beyond that - the useful effect is the scoping side effect, not the value.

Scope and what the alias actually is

The alias lives in the **enclosing lexical block**, just like `my` and `state`. It ends at the closing brace of that block (or the end of the file for a top-level declaration). Outside the scope, the unqualified name is no longer legal under `use strict 'vars'` - the package variable itself is untouched and still reachable by its fully qualified name.

```
package Foo;  
use strict;  
  
$Foo::foo = 23;
```

(continues on next page)

(continued from previous page)

```

{
    our $foo;          # alias to $Foo::foo
    print $foo;        # 23
}

print $Foo::foo;      # 23 - the package variable is unaffected
print $foo;           # error: requires explicit package name

```

Because our aliases a **package** variable, the package is fixed at the point of declaration, not at the point of use. Changing packages with `package` mid-scope does not rebind the alias:

```

package Foo;
our $bar;          # declares $Foo::bar for the rest of this scope
$bar = 20;

package Bar;
print $bar;        # 20 - still refers to $Foo::bar

```

our vs my

`our` and `my` look alike on the page and are constantly confused. They do opposite things.

- `my` creates a fresh **private** variable that lives only inside its lexical scope. It has no name in any package's symbol table and nothing outside the scope can reach it.
- `our` creates a lexically scoped **alias** to a **public** package variable. The variable lives in the symbol table; any other code that knows its fully qualified name can read or write it.

Use `my` by default. Reach for `our` when you deliberately want package-global state: module exporter lists (`our @EXPORT`, `our @EXPORT_OK`), `our $VERSION`, a package-wide configuration hash other code is expected to poke at, or the traditional `our @ISA` for inheritance.

```

package My::Module;

our $VERSION      = '1.23';
our @EXPORT_OK    = qw(frobnicate);
our @ISA          = ('Exporter');

```

Each of those names is an alias to `$My::Module::VERSION`, `@My::Module::EXPORT_OK`, `@My::Module::ISA` - exactly what `Exporter` and `UNIVERSAL::isa` go looking for.

our vs local

`local` and `our` often appear together and are complementary, not alternatives.

- `our` makes the package variable reachable under its unqualified name inside a lexical scope. It does **not** save or restore the value.

- `local` saves a package variable's current value and arranges for it to be restored when the enclosing dynamic scope ends. It does **not** make the name legal under `use strict 'vars'`.

The idiomatic pairing - temporarily change a package global inside a subroutine - uses both:

```
sub with_verbose {  
    our $verbose;          # make the unqualified name legal here  
    local $verbose = 1;    # save old value, restore on return  
    do_work();  
}
```

our vs state

`state` is the other lexical declaration. It creates a private variable whose value persists across calls to the enclosing subroutine. `our` shares none of that machinery - its variable is public and has no per-call semantics. Pick `state` for per-subroutine memoisation, `our` for package-global state.

Scope of the declaration itself

Like `my`, `state`, and `local`, `our` can appear anywhere an expression can (outside string interpolation). The declaration takes effect **immediately** - even within the same statement - so the variable satisfies `use strict 'vars'` from the declaration onward. But the declaration does **not** retroactively cover earlier uses of the same name in the same statement:

```
package main;  
my $x = 2;  
foo($x, our $x = $x + 1, $x);    # foo() receives (2, 3, 2)  
foo($x, our $z = 5, $z);        # foo() receives (3, 5, 5)
```

The rule: each occurrence of the name is resolved as of its own position in the statement. After the declaration, the name is the alias; before it, the name means whatever it meant before.

The declaration also lets you reference the package variable on the right-hand side of its own initialiser, which is something `my` does not allow:

```
my $foo = $foo;    # error under strict: undeclared $foo on RHS  
our $foo = $foo;    # fine: RHS $foo is the package variable
```

Multiple variables and placeholders

More than one variable requires parentheses:

```
our ($bar, $baz);
```

Inside a parenthesised list, `undef` is a valid placeholder that discards the corresponding value of a list assignment:

```
our ( undef, $min, $hour ) = localtime;
```

Repeated declarations

Multiple `our` declarations with the same name in the same lexical scope are allowed when they are in **different** packages - each rebinds the alias to its package's variable:

```
use warnings;

package Foo;
our $bar;           # declares $Foo::bar for the rest of this scope
$bar = 20;

package Bar;
our $bar = 30;      # declares $Bar::bar - aliases the Bar variable
print $bar;         # 30
```

A second `our` declaration for the same name **in the same package and scope** is merely redundant - it rebinds to the same variable. Under `use warnings` Perl reports it the way it reports a repeated `my`, but there is no fresh variable and nothing is shadowed:

```
use warnings;
package Bar;
our $bar = 30;
our $bar;           # warning, but still the same $Bar::bar
print $bar;         # 30
```

Attributes and TYPE

`our` accepts the full declaration syntax of `my`:

```
our $x : shared;
our MyType $obj;
our $VERSION : shared = '1.00';
```

TYPE is currently bound to the `fields` pragma. Attributes are processed by the `attributes` pragma (and, since 5.8.0, by `Attribute::Handlers`). See the `attributes` pragma for the set of standard attributes. Ordinary user-level code rarely uses either.

Examples

Module preamble - the overwhelmingly most common use:

```
package My::Thing;
use strict;
use warnings;

our $VERSION      = '0.01';
our @ISA          = ('Exporter');
our @EXPORT_OK    = qw(thingify);
```

Declare and assign in one step:

```
package Cfg;
our $timeout = 30;    # same as $Cfg::timeout = 30
```

Temporarily override a package global inside a call:

```
sub quiet_run {
    our $verbose;
    local $verbose = 0;
    run_one_job();
}
```

Cross-package alias - declaration in Foo, use in Bar:

```
package Foo;
our $shared = 'hello';

package Bar;
# $shared here still refers to $Foo::shared, because the lexical
# alias was introduced while the current package was Foo.
print $shared;    # hello
```

Parenthesised list with placeholder:

```
our ( undef, $min, $hour, undef, $mon, $year ) = localtime;
```

Edge cases

- **our does not create the variable.** It declares an alias. The package variable exists the moment anything assigns to it (or takes a reference to it); `our $x;` alone does not allocate a new slot in the symbol table.
- **The alias is lexical, the variable is not.** Leaving the scope of the `our` ends the alias, not the variable. Another scope in another file that declares `our $x;` in the same package sees the same storage.
- **Same-statement resolution.** Only occurrences of the name **after** the `our` declaration in a statement use the alias. Earlier occurrences keep their prior binding (see the `foo()` example above).
- **Package, not current package at use.** The alias is bound to the package in force at the point of declaration. Switching package later does not re-target it.
- **Not the same as `use vars`.** `use vars '$x'` makes `$x` legal under `strict` for the rest of the **package**, regardless of scope. `our $x` is lexical. Prefer `our`; `use vars` is legacy.
- **Not a file-scoped `pragma`.** A top-level `our $x;` in a file covers until the end of the file, which is a lexical scope - but requiring that file from elsewhere does not extend the alias into the requiring file's scope.
- **Attribute and TYPE semantics are still evolving.** Anything beyond `: shared` in user code is rare; read the attributes and fields `pragma` documentation before relying on them.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `my` - declare a private lexical variable; the default choice when you want a new variable
- `local` - dynamically scope a temporary value for a package variable; pairs with `our` when you need both strict legality and save/restore
- `state` - declare a private lexical variable that persists across calls to the enclosing subroutine
- `package` - set the current package, which is what `our` aliases into
- `use` - compile-time module loader; `use strict 'vars'` is the pragma that makes `our` necessary in the first place
- `undef` - valid placeholder inside a parenthesised `our` list to discard a value on assignment

SCALARS and strings · Fixed-length data

pack

Convert a list of Perl values into a binary string according to a template.

`pack` is the low-level serialiser: you describe the byte layout with a `TEMPLATE`, hand it a `LIST` of values, and it returns a scalar whose characters are the concatenated machine-level encoding of those values. It is the inverse of `unpack` and the standard way to build fixed-width records, wire protocols, C struct layouts, raw IP addresses for `sockaddr_in`, and any other byte-level payload that Perl does not model as a first-class type.

This page is a **directive reference**. For a narrative introduction

- “I have this wire protocol, how do I parse it?” - start at the [pack/unpack tutorial](#).

Synopsis

```
my $bytes = pack TEMPLATE, LIST;
my $ip    = pack "C4",      split /\./, "192.168.1.1";
my $rec   = pack "Z8 Z8 L", $user, $host, $ts;
```

What you get back

A plain Perl scalar holding the packed bytes. Its length is the sum of the widths of every template directive; its contents are binary and may contain embedded NUL bytes. Treat it as a byte string, not as text - writing it to a handle with a `:utf8` or `:encoding(...)` layer will re-encode it. Use `binmode $fh` or open with `>:raw` before emitting.

By default the result is in **character mode** (C0). A template that starts with U, or switches to U0 mid-template, produces a UTF-8-encoded Unicode string instead. Do not use this as a substitute for the Encode module.

Template syntax

A TEMPLATE is a sequence of **directives**. Each directive is an ASCII letter, optionally followed by:

- a **repeat count** - a decimal integer, *, or [...]
- one or more **modifiers** - !, <, >
- a **group** - (...) gathers directives so a repeat count or endianness modifier applies to the whole

Whitespace between directives is ignored. A # introduces a comment running to end-of-line - the same convention as Perl source.

Directive table

Every pack directive in one place. **W** is the width in bytes per scalar consumed (a C consumes one value and produces one byte; an a3 consumes one value and produces three bytes). **Endian** column: *native* means the CPU's byte order; *big* / *little* are fixed regardless of host.

Directive	Con- sumes	W	Signed	Endian	Modifiers	Notes
a	1 string	count	-	-	-	NUL-padded to width, truncates if too long
A	1 string	count	-	-	-	Space-padded to width
Z	1 string	count	-	-	-	NUL-terminated; Z* always appends a trailing NUL
b	1 string	count/8	-	-	-	Bit string, LSB first within each byte
B	1 string	count/8	-	-	-	Bit string, MSB first within each byte
h	1 string	count/2	-	-	-	Hex string, low nybble first

continues on next page

Table 1 – continued from previous page

Directive	Con- sumes	W	Signed	Endian	Modifiers	Notes
H	1 string	count/2	-	-	-	Hex string, high nybble first
c	1 integer	1	yes	-	-	Signed char
C	1 integer	1	no	-	-	Unsigned char (octet)
W	1 integer	1	no	-	-	Unsigned char; allows values above 255 in U0 mode
s	1 integer	2	yes	native	! < >	s! = native short
S	1 integer	2	no	native	! < >	S! = native unsigned short
l	1 integer	4	yes	native	! < >	l! = native long
L	1 integer	4	no	native	! < >	L! = native unsigned long
i	1 integer	native	yes	native	! < >	sizeof(int); at least 32 bits
I	1 integer	native	no	native	! < >	sizeof(unsigned int)
q	1 integer	8	yes	native	< >	Requires 64-bit-integer Perl
Q	1 integer	8	no	native	< >	Requires 64-bit-integer Perl
n	1 integer	2	no	big	!	Portable network order; n! = signed

continues on next page

Table 1 – continued from previous page

Directive	Con- sumes	W	Signed	Endian	Modifiers	Notes
N	1 integer	4	no	big	!	Portable network order; N! = signed
v	1 integer	2	no	little	!	“VAX” order; v! = signed
V	1 integer	4	no	little	!	“VAX” order; V! = signed
j	1 integer	IV size	yes	native	< >	Perl-internal signed integer
J	1 integer	UV size	no	native	< >	Perl-internal unsigned integer
f	1 number	4	-	native	< >	Single-precision IEEE 754
d	1 number	8	-	native	< >	Double-precision IEEE 754
F	1 number	NV size	-	native	< >	Perl-internal float (NV)
D	1 number	varies	-	native	< >	Long double; format varies by platform
p	1 string / undef	ptr	-	native	< >	Pointer to NUL-terminated string; undef → null pointer
P	1 string / undef	ptr	-	native	< >	Pointer to fixed-length buffer; count = buffer length
u	1 string	varies	-	-	-	Uen-coded; count = max bytes per output line (default 45)

continues on next page

Table 1 – continued from previous page

Directive	Con- sumes	W	Signed	Endian	Modifiers	Notes
U	1 code- point	varies	-	-	-	Unicode character number; encodes to UTF-8 in U0 mode
w	1 integer ≥ 0	varies	no	-	-	BER-compressed integer, big-endian base-128
x	nothing	1	-	-	!	Insert one NUL; x!N aligns to multiple of N
X	nothing	-1	-	-	!	Back up one byte; X!N aligns backward
@	nothing	absolute	-	-	!	Zero-fill or trun- cate to position N within group
.	1 integer	absolute	-	-	!	Zero-fill or trun- cate to position given by the value
(...)	-	-	-	-	< > !	Group: repeat count and en- dianness prop- agate inside
/	-	-	-	-	-	Unpack-only. In pack, use <i>length-item/</i> <i>item</i>

The modifiers:

Mod	Effect
i-	fier
!	On <code>s / S / l / L / i / I</code> : use native <code>sizeof(...)</code> instead of fixed width. On <code>x / X</code> : turn into alignment commands. On <code>n / N / v / V</code> : interpret as signed. On <code>@ / .</code> : count bytes of the internal representation, not characters.
>	Force big-endian byte order (“big end touches the construct”).
<	Force little-endian byte order.

Applied to a group, `< / >` cascade into every byte-ordered directive inside the group and are silently ignored by directives that do not accept them.

Repeat counts

A directive letter may be followed by:

- a **number** `N` - apply the directive that many times, consuming `N` values from `LIST`
- `*` - consume all remaining values; for `x / X / @` this is equivalent to `0`; for `u` it selects the default of 45
- `[N]` - equivalent to a bare `N`
- `[template]` - the repeat count is the packed byte length of *template*. `x[L]` skips as many bytes as a packed long; `x![d]` aligns to a double boundary

String and bit/nybble directives treat the count as **width of a single value**, not count-of-values: `pack "A4", "abcdef"` produces `"abcd"`, not four copies of `"abcdef"`.

Grouping

Parentheses group directives. A group may take a repeat count or an endianness modifier:

```
pack "(s1)<", -42, 4711      # same as "s<l<", -42, 4711
pack "(CCS)*", @triplets    # repeat group for every triplet
```

Within each repetition of a group, `@` positioning starts over at 0

- `pack '@1A((@2A)@3A)', qw(X Y Z)` produces `"\0X\0\0YZ"`.

Length-prefixed payloads (/)

Write `length-item/sequence-item`. The length is computed from the sequence value and packed according to *length-item*; then the sequence itself is packed according to *sequence-item*:

```
my $msg = pack "n/a*", "hello, world";
# "\x00\x0chello, world" - 16-bit big-endian length, then the bytes
```

The length directive may be any integer directive (`n`, `N`, `w`, `C`, ...) or even a string directive (`A4`, `Z*`) when you want the length written as ASCII.

Examples

Build a fixed-width record with two NUL-terminated strings and a 32-bit native timestamp:

```
my $rec = pack "Z8 Z8 L", "alice", "server01", $now;
# 8 + 8 + 4 = 20 bytes
```

Portable network byte order - prefer `n / N / v / V` over native-width directives when the bytes leave the machine:

```
my $be = pack "n N", 42, 4711;      # big-endian 16 + 32 bit
my $le = pack "v V", 42, 4711;      # little-endian
my $sx = pack "s>l>", -42, 4711;    # signed big-endian via modifier
my $gx = pack "(sl)<", -42, 4711;    # signed little-endian via group
```

Align a field inside a C struct. `x![d]` inserts just enough NUL bytes to reach the next multiple of a double's width:

```
# struct { char c; double d; char cc[2]; }
my $s = pack "c x![d] d c2", $c, $d, $c1, $c2;
```

Round-trip through a hex string:

```
my $raw = pack "H*", "deadbeef";    # "\xde\xad\xbe\xef"
my $hex = unpack "H*", $raw;        # "deadbeef"
```

Edge cases

- **Too few values:** missing values are treated as `" "`. `pack "A4 A4", "hi"` yields `"hi \0\0\0\0"`, not a fatal error.
- **Too many values:** extras are silently ignored.
- **a vs A vs Z:** all three pad to an exact width. `a` pads with NUL, `A` pads with space, `Z` pads with NUL **and** guarantees a trailing NUL - so `Z8` encodes at most 7 data bytes plus the terminator.
- **Character width vs byte width** - the single most common trap. Counts for `a / A / Z` and offsets for `@ / .` are in **characters of the packed string**, not bytes. In `C0` mode a character is one byte; in `U0` mode a character may span multiple UTF-8 bytes. Use the `!` modifier on `@ / .` when you need byte offsets regardless of mode.
- **Endianness on `s / S / l / L / i / I / q / Q`:** native only, not portable. Use `n / N / v / V` or apply `> / <` explicitly. `f / d` are likewise native; IEEE 754 alone does not pin down endianness.
- **Inf / NaN packed as integers:** fatal error. No sensible mapping exists.
- **q / Q on non-64-bit Perl:** raises an exception.
- **p and P capture pointers** into the caller's memory. The referent must remain live until the packed string is consumed - a temporary string passed to `p` may be freed before you read it back. Avoid outside XS code.

- **Grouping and @:** positioning with @ starts over at 0 inside every repetition of a group. `pack '@1A((@2A)@3A)'`, `qw(X Y Z)` produces `"\0X\0\0YZ"`, not what a naive reading suggests.
- **Floating-point round-trip loss:** Perl stores numbers as doubles, so `unpack("f", pack("f", $x))` generally does not equal `$x` - packing through single precision truncates.
- **Writing packed bytes to a text handle:** a handle with `:utf8` or `:encoding(...)` will re-encode the bytes. Use `binmode $fh` or open with `">:raw"`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- [unpack](#) - the inverse operation; same template language
- [sprintf](#) - formatted text output rather than binary bytes
- [vec](#) - bit-level access to a string without a template
- [chr / ord](#) - single-character conversions
- [syswrite](#) - write a packed byte string without PerlIO surprises
- [pack/unpack tutorial](#) - task-oriented walkthrough with worked protocol and file-format examples

[Scoping](#) · [Modules](#) · [Classes and OO](#)

package

Declare the compile-time namespace for the declarations that follow.

`package` tells the compiler which symbol table to use when it resolves unqualified names in the code it is about to parse. It does **not** create a package in any run-time sense - packages spring into existence the first time something is stored in them. What a package statement really does is re-point the compiler: from here on, `$foo` means `$NAMESPACE::foo`, `&bar` means `&NAMESPACE::bar`, and so on, until the next package statement or the end of the enclosing scope.

Synopsis

```
package NAME;  
package NAME VERSION;  
package NAME      { ... }  
package NAME VER { ... }
```

What you get back

`package` is a compile-time directive, not an expression. It has no useful runtime value - treat it as a statement. Its effect is on the parser: all subsequent unqualified identifiers resolve in `NAME`'s symbol table until scope ends.

The form with `VERSION` has one visible side effect: it sets `$NAME::VERSION` to a `version` object built from `VERSION`. This happens exactly once per `package NAME VERSION` statement, at compile time.

Global state it touches

- **The current compile-time package.** Every package statement replaces it for the rest of the scope. `__PACKAGE__` returns the current value at any point in the source.
- `$NAME::VERSION` - set when the `VERSION` argument is provided. Set it only once per package.
- The target symbol table (stash) is created on first use. A package's stash is itself a hash named `%NAME::` (so `main`'s is `%main::`), keyed by the unqualified symbol names it holds. Nothing is populated by the package statement itself; subsequent declarations (`sub`, `our`, assignments to `$NAME::foo`, etc.) fill it.

The four forms

`package NAMESPACE`

Switches the compile-time package to `NAMESPACE` from this point to the end of the current scope (enclosing block, file, or `eval`). This is the classic “one package per file” form:

```
package My::Thing;

sub new { ... }           # defines &My::Thing::new
our $count = 0;           # defines $My::Thing::count

1;
```

`package NAMESPACE VERSION`

Same scoping as the bare form, plus it sets `$NAMESPACE::VERSION` to a `version` object. `VERSION` must be a **strict** version literal: either a positive decimal (1.23, 0.001) or a dotted-decimal v-string with a leading `v` and at least three components (`v1.2.3`). Exponent notation is not allowed.

```
package My::Thing 1.042;
package My::Thing v1.2.3;
```

Set `$VERSION` only once per package - a second `package NAME VERSION` statement in the same package overwrites the first, and `use Module VERSION` consumers see only the final value.

package NAMESPACE BLOCK

Lexically scopes the package switch to BLOCK. Inside the braces the compile-time package is NAMESPACE; **outside the closing brace the previous package is restored**. This is the only form that reverts automatically.

```
package Outer;
our $x = 1;           # $Outer::x

package Inner {
    our $y = 2;       # $Inner::y
}

our $z = 3;           # $Outer::z again
```

package NAMESPACE VERSION BLOCK

The block form with a version literal. Scope reverts at the closing brace; `$NAMESPACE::VERSION` stays set.

```
package My::Sub v0.9.1 {
    sub ping { "pong" }
}
```

What package does and does not scope

package affects **dynamic (package) variables** and unqualified name resolution. It does not affect lexicals:

- `my`, `state` - created in the lexical pad, invisible to package lookup. A package statement does nothing to them.
- `local` - dynamically scopes a package variable, which still belongs to whatever package its fully-qualified name names.
- `our` - declares a lexical alias to a package variable in the *current compile-time package*. This is the mechanism that lets `our $count` inside package `My::Thing` alias `$My::Thing::count`.

Switching packages after an `our` declaration does not retarget the alias: the alias was bound to the package in effect when `our` was compiled.

```
package Foo;
our $x;           # aliases $Foo::x
package Bar;
$x = 1;           # still writes $Foo::x
```

Interaction with `use v5.36` and newer

`use v5.36` (and any `use v5.36` or later) enables `strict` and warnings implicitly. Once `strict` is on, unqualified package variables require an `our` declaration (or full qualification). Combined with a package statement, the idiomatic modern file skeleton is:

```
package My::Thing 1.00;

use v5.36;                # strict, warnings, say, signatures, ...

our @EXPORT_OK = qw(thingify);

sub thingify { ... }

1;
```

The package NAME VERSION form and use VERSION are independent: the first sets \$My::Thing::VERSION; the second enables language features for the current file.

Special identifiers that always mean main::

A small set of identifiers are forced into the main:: package regardless of the current package:

- STDIN, STDOUT, STDERR, ARGV, ARGVOUT
- @ARGV, @INC, %ENV, %SIG, %INC
- All punctuation variables: \$_, \$!, \$@, @_, etc.
- The single-character identifier _, in all its sigils: the underscore filehandle used by `stat` and the `-X` file tests, the scalar \$_, and the array @_.

Writing to \$! inside package Foo still writes to \$main::!. This is by design: these names would be useless if they moved with the package.

Examples

A minimal module file. The file name must match the package name (`require` translates :: to the directory separator):

```
# lib/My/Greeter.pm
package My::Greeter 1.00;
use v5.36;

sub new { my ($cls, %a) = @_; bless { %a }, $cls }
sub greet { my $self = shift; "hello, $self->{name}\n" }

1;
```

Two packages in one file using the block form. Scope reverts automatically, so the 1; at the bottom belongs to no package in particular - it's evaluated in whichever package was active at the top of the file (main for a script, the file's own package for a module):

```
package My::Thing {
    sub do_stuff { ... }
}
```

(continues on next page)

(continued from previous page)

```
package My::Thing::Helper {  
    sub helper { ... }  
}
```

Switching back and forth in the statement form - legal but rarely what you want:

```
package A;  
sub one { 1 }           # &A::one  
  
package B;  
sub two { 2 }           # &B::two  
  
package A;  
sub three { 3 }         # &A::three
```

Setting a version and reading it back. Note that `$VERSION` is a version object, not a plain string; stringify or compare with `version`:

```
package Foo 1.23;  
  
say $Foo::VERSION;      # 1.23  
say ref $Foo::VERSION;  # version
```

`our` inside a package gives you a lexically-scoped alias to a package variable - the canonical way to declare module-level globals under `strict`:

```
package Counter;  
use v5.36;  
  
our $count = 0;          # aliases $Counter::count  
sub bump { $count++ }
```

Edge cases

- **package `NAMESPACE VERSION` accepts only strict version literals.** Exponent notation (1e3), non-numeric strings, and two-component v-strings (v1.2) are compile errors.
- **Setting `$VERSION` twice** in the same package (whether by two package `NAME VERSION` statements or by explicit assignment after a package `NAME VERSION`) is legal but confusing - `use Module VERSION` checks the final value, and CPAN tooling reads whichever one it finds first. Declare the version once.
- **The block form is a separate lexical scope.** `my` and `state` variables declared inside a package `NAME { ... }` are not visible outside, just like any other block.
- **package `NAMESPACE` without a trailing statement or block** affects the rest of the file. A package at the top of a script permanently switches the script out of `main`, which means `$main::ARGV[0]` is still your argument list but freshly declared globals belong to the new package. This is almost never what a script wants.

- **A file that contains only package Foo; and no other code** still compiles cleanly and creates the `Foo::` stash on first reference - but requiring it returns the value of the last statement, which is the package statement itself (false). Put `1;` at the end of every module file.
- **Nested block forms** restore the immediately enclosing package, not main:

```
package Outer;
package Inner {
    package Innermost {
        # __PACKAGE__ is Innermost
    }
    # __PACKAGE__ is Inner again
}
# __PACKAGE__ is Outer again
```

- **package inside eval.** The package switch is scoped to the `eval` body. Symbols created during the `eval` persist in their stashes (stashes are global), but the compile-time package reverts when the `eval` ends.
- **Apostrophe as package separator.** The apostrophe was Perl's original namespace separator: `$main'sail` meant `$main::sail`, long before `::` existed. It survives only for reading Perl 4 era code and is disabled under modern feature bundles, so a current program written with `use v5.36` (or any recent bundle) will not recognise it. Do not write it; the double-colon `::` is the only separator current code should use.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `use` - load a module at compile time and import symbols from its package; typically paired with a package declaration at the top of the loaded file
- `require` - load a module or file at runtime; the loaded file's package statement determines which stash its declarations populate
- `bless` - associate a reference with a package so method calls dispatch to that package's subs; the glue between package and OO
- `my` - declare a lexical variable that package does **not** affect; the counterpart to package globals
- `our` - declare a lexical alias to a package variable in the current compile-time package; the strict-safe way to use module-level globals
- `local` - dynamically scope a package variable's value without changing which package it belongs to

Processes

pipe

Open a pair of connected filehandles - one for reading, one for writing.

pipe wraps the `pipe(2)` system call. It creates an unnamed, unidirectional byte channel in the kernel and hands you two filehandles: `READHANDLE`, from which bytes can be read, and `WRITEHANDLE`, into which bytes can be written. Anything written to `WRITEHANDLE` becomes available on `READHANDLE` in the same order. The classic use is communication between a parent process and a child produced by `fork`: one side writes, the other reads.

Synopsis

```
pipe READHANDLE, WRITEHANDLE
```

What you get back

True on success, false on failure (with `$!` set to the reason - typically `EMFILE` or `ENFILE` when the process or system file-table is full). Both handles are created as side effects on the arguments; there is no “pipe object” return value.

On success you now hold **two** open file descriptors. Every descriptor you create is a descriptor you must `close` - especially the end each process does **not** use after a `fork` (see *Edge cases*).

Global state it touches

- `$!` - set on failure to the `errno` from the underlying `pipe(2)` call.
- `$|` - **autoflush** on the write-side handle. `print` buffers through `PerlIO`, so without autoflush the reader may see nothing until the writer exits or the buffer fills. Set `$|` on the write handle (typically via `select`) before writing, or your protocol will deadlock.
- `$_^F` - the **maximum system filehandle**. New descriptors with `fileno` greater than `$_^F` (default 2, i.e. above `STDERR`) have the close-on-exec flag set automatically, so they do not leak into programs started by `exec`. Lower `$_^F` to keep the pipe open across `exec`; raise it, or dup onto a low-numbered fd, to force inheritance.

Examples

A parent writes a line; a forked child reads it:

```
pipe my $reader, my $writer or die "pipe: $!";

my $pid = fork // die "fork: $!";
if ($pid == 0) {
    # child: reads only
    close $writer;
    my $line = <$reader>;
    print "child saw: $line";
    close $reader;
}
```

(continues on next page)

(continued from previous page)

```

    exit 0;
}

# parent: writes only
close $reader;
$writer->autoflush(1);
print $writer "hello from parent\n";
close $writer;
waitpid $pid, 0;

```

The child reports the other direction - a child produces data, the parent consumes it:

```

pipe my $reader, my $writer or die "pipe: $!";

my $pid = fork // die "fork: $!";
if ($pid == 0) {
    close $reader;
    $writer->autoflush(1);
    print $writer "$_\n" for 1 .. 5;
    close $writer;
    exit 0;
}

close $writer;
while (my $line = <$reader>) {
    print "got: $line";
}
close $reader;
waitpid $pid, 0;

```

Autoflush from the selected-handle side, for code that predates IO::Handle method calls:

```

pipe my $reader, my $writer or die "pipe: $!";
my $old = select $writer; $| = 1; select $old;

```

For most uses, `open` with a pipe mode is simpler - it forks, pipes, and execs for you:

```

open my $fh, '|-', 'gzip', '-c' or die $!; # write to gzip's stdin
print $fh "data\n";
close $fh;

open my $fh, '-|', 'ls', '-l' or die $!; # read from ls's stdout
while (my $line = <$fh>) { print $line }
close $fh;

```

Reach for bare pipe when you need both directions, need to retain the unexec'd child, or are building something `open`'s two pipe-forms do not cover - then `IPC::Open2` / `IPC::Open3` wrap the pipe + `fork` + `exec` dance for you.

Edge cases

- **Close the end you do not use.** After `fork`, **both** processes hold **both** descriptors. The reader must close the write end, and the writer must close the read end. Skip this and you get the classic pipe bug: the reader's `<$reader>` never returns EOF, because the kernel still sees a writer open (the reader's own, idle, copy of `$writer`). The program hangs.
- **Writing to a reader-less pipe raises SIGPIPE.** Once every copy of the read end is closed, the next `print` on the write side delivers SIGPIPE, whose default action is to kill the process. Install `$SIG{PIPE} = 'IGNORE'` (or a handler) to turn it into a normal write failure with `$!` set to EPIPE, which you can then check.
- **Deadlock in pipe loops.** Two processes each writing to the other through pipes will block forever if both write buffers fill before either process reads. Kernel pipe buffers are small (typically 64 KiB). Drain one side, use non-blocking I/O, or use `select` / `IO::Select` to multiplex.
- **Buffering hides writes.** Without `$|` (or `$writer->autoflush(1)`) on the write side, data sits in PerlIO's buffer until the buffer fills or the handle is closed. A reader blocked on `<$reader>` will wait - often indefinitely - for data that has already been "written" from the writer's point of view.
- **Unidirectional only.** pipe produces a one-way channel: write to `WRITEHANDLE`, read from `READHANDLE`, not the reverse. For bidirectional communication use `socketpair` or two pipes.
- **Binary data.** The handles returned by pipe start in the default PerlIO layer stack. If you are moving raw bytes across the pipe, apply `binmode` to both ends; otherwise CRLF or encoding layers inherited from the process defaults can rewrite your data.
- **Close-on-exec and `fileno`.** On systems with `FD_CLOEXEC`, any new descriptor whose `fileno` exceeds `$_^F` is created with the close-on-exec flag set. Pipe descriptors typically land well above 2, so by default they do **not** survive `exec` - which is normally what you want. Lower `$_^F` *before* calling pipe if you intend to pass the descriptors to an `exec'd` program.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `fork` - the other half of the pipe idiom; pipe is almost always called just before a fork
- `open` - the `'| - '` and `'- | '` modes fork, pipe, and exec a child in one call; prefer them when you need exactly one direction to one external command
- `close` - closing the unused end in each process is mandatory, not optional
- `read` - byte-oriented reads from `READHANDLE` when the line-oriented `<$fh>` form is the wrong shape
- `print` - the normal way to push data into `WRITEHANDLE`; remember `$|`

- `fileno` - recover the underlying file descriptor, e.g. to hand to `select` / `IO::Select` or to a lower-level system call

Arrays

pop

Remove and return the last element of an array.

`pop` shortens the array by one and hands you the element that was removed. The array is modified in place. With no argument, `pop` operates on `@_` inside a subroutine and on `@ARGV` everywhere else - the same default target as `shift`.

Synopsis

```
pop @arr
pop @$aref
pop                                     # default: @_ inside a sub, @ARGV outside
```

What you get back

The element that was removed, as a scalar. If the array was empty, the return value is `undef` and the array stays empty - this is not an error and emits no warning, so code that always calls `pop` in a loop needs its own termination check.

```
my @empty;
my $x = pop @empty;           # $x is undef, no warning
```

Note that `pop` also returns `undef` when the last element *was* `undef`. The two cases are indistinguishable from the return value alone; use `@arr` in boolean context, or scalar `@arr`, to tell them apart before popping.

```
my @arr = ('one', 'two', undef);
my $x = pop @arr;             # $x is undef - but so would an empty_
→pop be
```

Global state it touches

`pop` with no argument reads `@_` (inside a subroutine) or `@ARGV` (at file scope, in `BEGIN/INIT/CHECK/END` blocks, and inside `eval STRING`). It does not interact with `$_`, `$,`, `$\`, or any other punctuation variable.

Examples

Basic use - pull the last element off a plain array:

```
my @arr = ('cat', 'dog', 'mouse');
my $item = pop @arr;          # $item is 'mouse'
# @arr is now ('cat', 'dog')
```

LIFO stack - `push` to add, `pop` to remove, both at the same end:

```
my @stack;  
push @stack, 'a';  
push @stack, 'b';  
push @stack, 'c';  
my $top = pop @stack;      # 'c'; @stack is ('a', 'b')
```

Discard the last element when you do not need its value. `pop` in void context is perfectly valid and is the idiomatic way to drop the tail:

```
pop @arr;                  # shorten @arr by one, value thrown away
```

Default target inside a subroutine - `pop` with no argument pops from `@_`, which is how a sub peels its last argument off the argument list:

```
sub last_arg {  
    return pop;            # pops from @_  
}  
last_arg(1, 2, 3);        # returns 3
```

Pop through an array reference. Dereference with `@$aref` (or `@{ $aref }` for any non-trivial expression) and `pop` works exactly as on a named array:

```
my $aref = [10, 20, 30];  
my $x    = pop @$aref;    # 30; $aref now points to [10, 20]
```

Edge cases

- **Empty array returns `undef`, not an error.** No warning is issued, even under `use warnings`. A loop of the form `while (defined(my $x = pop @arr)) { ... }` terminates on empty or on a genuine `undef` element - use `while (@arr)` instead if the array may legitimately contain `undef`.
- **Bareword expression in place of `ARRAY`.** `pop` requires an array, not an arbitrary expression. `pop @$aref` works because `@$aref` is array syntax; `pop $aref` was briefly allowed as an experiment between 5.14 and 5.22 and has been a compile error since 5.24. Always write the explicit `@` sigil on the dereference.
- **Tied arrays.** `pop @tied` calls the tied class's `POP` method if defined; otherwise the default `FETCHSIZE + FETCH + DELETE + STORESIZE` fallback applies. Side effects of those methods fire in that order.
- **`pop` shortens the array.** After `pop @arr`, `$#arr` decreases by one and scalar `@arr` drops by one. If you need to inspect the tail without removing it, read `$arr[-1]` or `$arr[$#arr]` directly - see `$#arr` in `perlvar`.
- **No-argument form changes meaning by scope.** At file scope, `pop;` means `pop @ARGV`. Inside a subroutine it means `pop @_`. The same source line therefore behaves differently depending on where it appears - a common surprise when code is refactored into or out of a sub. Write the target explicitly (`pop @ARGV` or `pop @_`) whenever the intent is not obvious from context.
- **`pop` on an array aliased through `@_`.** Subroutine arguments are aliases to the caller's values. `pop @_` removes the alias entry from `@_` but does not shorten the

caller's array; to shrink the caller's array the sub must be passed the array *by reference* and `pop @$aref` used instead.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `push` - the counterpart; append to the same end of the array that `pop` removes from
- `shift` - remove and return the *first* element; same defaulting rules as `pop` (`@_` in a sub, `@ARGV` otherwise)
- `unshift` - prepend to the front; paired with `shift` the way `push` is paired with `pop`
- `splice` - remove or replace an arbitrary slice; `splice @arr, -1` is the long-hand form of `pop @arr`
- `last` - loop-control keyword, unrelated to array end; listed because the names are often confused
- `$#arr` - last index of `@arr`; changes when `pop` shortens the array

Regular expressions and pattern matching

pos

Report or set where the next `/g` regex match will resume in a string.

`pos` reads (or, as an lvalue, writes) the offset that the regex engine stores on a scalar after a global match (`m//g`, `s//g`). The offset counts characters, not bytes, and is the position **after** the last successful match - the place the next `/g` iteration will start scanning. With no argument `pos` operates on `$_`.

Synopsis

```
pos SCALAR
pos $str = N
pos
```

What you get back

An integer offset, or `undef` when no position is recorded. `0` is a valid offset and means “start of string”; it is not the same as `undef`, which means “no `/g` match has run, or the last one failed and reset the position.” Always distinguish the two with `defined`:

```
if (defined pos $str) {
    # a /g scan is in progress
}
```

Used as an lvalue, `pos SCALAR` returns an assignable location:

```
pos($str) = 5;           # next /g match starts at char offset 5
```

Global state it touches

`pos` reads and writes the per-scalar regex position attached to its operand. With no argument it targets `$_`. The stored offset is what `\G` anchors against in the next match, so every call to `pos` potentially changes where `\G` binds.

Examples

Walk every word in a string with `/g` in scalar context, using `pos` to report progress:

```
my $s = "one two three";
while ($s =~ /(\w+)/g) {
    printf "%-5s ends at %d\n", $1, pos $s;
}
# one    ends at 3
# two    ends at 7
# three  ends at 13
```

Skip ahead before starting the scan. The first match begins at offset 4, not 0:

```
my $s = "AAA BBB CCC";
pos($s) = 4;
$s =~ /(\w+)/g;
print $1, "\n";           # BBB
```

Anchor a follow-up match to the previous one with `\G`. Without `\G` the engine would scan forward past any gap; with it, the match must start exactly where the last one ended:

```
my $s = "12ab34cd";
while ($s =~ /\G(\d+)(\w+?)(?=\d|\z)/g) {
    print "num=$1 tail=$2\n";
}
# num=12 tail=ab
# num=34 tail=cd
```

Restart a scan by clearing the position:

```
pos($s) = undef;         # next /g starts from offset 0 again
```

Edge cases

- **Bare `pos`** targets `$_`. `pos` inside `while (<>) { ... }` therefore reports the position on the current input line.
- **Characters, not bytes.** For a string containing multi-byte characters, `pos` returns the character offset. The (deprecated) use `bytes` pragma switches to byte offsets; new code should not rely on it.

- **Failed /g match resets the position to `undef`** - the next /g starts over at offset 0. Add the /c modifier (m//gc) to preserve the position on failure, which is the usual idiom when composing several alternative \G-anchored patterns against the same string.
- **Reads during a match are stale.** pos reflects the **previous** match's end. Expressions like (#{ pos() = 5 }) or s//pos() = 5/e influence the next match, not the one currently running.
- **Zero-length match flag.** Setting pos also clears the internal *matched with zero-length* flag, so a subsequent zero-width match at the same position is allowed again. See the [performance chapter](#) of the regex guide under *zero-length-match termination*.
- **Non-lvalue operand.** pos requires a real scalar variable for the lvalue form; pos("literal") = 3 is a compile-time error.
- **Offset 0 vs `undef`.** pos of 0 means the next /g starts at the beginning; `undef` means no position is set. They behave identically for the first match but differ after the zero-length-match flag state is considered.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `m` - the /g modifier is what creates and advances the position pos reads; the /c modifier on failure preserves it
- `qr` - precompile a pattern once, then reuse it in \G-anchored /g loops without reparsing
- `split` - the other everyday way to walk a string in pieces; no pos involved, but often a cleaner choice when the delimiters are simple
- `\G anchor`
 - zero-width anchor that binds to the current pos; the main reason to touch pos in the first place
- `$_` - default target of bare pos

I/O

print

Write a list of values to a filehandle.

print is the everyday output primitive. It stringifies each item in LIST, joins them with the current value of \$, (the output field separator, usually empty), writes them to FILEHANDLE, and finally appends \$\ (the output record separator, usually empty). If FILEHANDLE is omitted, output goes to the currently selected handle (see [select](#)), which defaults to STDOUT. If LIST is omitted, \$_ is printed.

Synopsis

```
print LIST
print FILEHANDLE LIST
print {EXPR} LIST
print
```

What you get back

1 on success, `undef` on failure (with `$!` set). The return value is rarely checked - most `print` calls target `STDOUT` or `STDERR` and a write failure to either is already a program-ending condition. Check the return value when writing to a file, pipe, or socket where partial writes and broken pipes matter.

```
print $fh @records
    or die "write to $path failed: $!";
```

Default filehandle, default separators

Three pieces of global state shape every `print` call:

- The **currently selected filehandle**, changed with `select`. `print LIST` (no explicit filehandle) writes there. At program start this is `STDOUT`.
- `$,` - **output field separator**, inserted between list items. Default is the empty string, so `print 1, 2, 3` produces `123`, not `1 2 3`. Also known as `$OFS` under `use English`.
- `$\` - **output record separator**, appended after the whole list. Default is the empty string, so `print` does **not** add a newline. Also known as `$ORS` under `use English`.

Setting both gives Python-like `print` semantics:

```
local $, = " ";
local $\ = "\n";
print 1, 2, 3;           # "1 2 3\n"
```

Use `local` when changing these inside a subroutine so the change does not leak to the caller.

`say` is exactly `print` with `$\` forced to `"\n"` for that one call - reach for it when you want a newline every time and do not want to touch `$\`.

Examples

Basic write to `STDOUT`:

```
print "hello, world\n";
```

Write to a filehandle with the indirect-object form. Note: **no comma** between the filehandle and the list:

```
open my $fh, ">", "out.txt" or die $!;
print $fh "line 1\n", "line 2\n";
close $fh;
```

Write to a filehandle held in an expression. Any filehandle more complex than a bareword or a plain scalar needs the {EXPR} block form:

```
my @logs = (*STDOUT, *STDERR);
print { $logs[$level] } "message\n";
```

Join list items with \$, and terminate the record with \$\\:

```
local $, = ", ";
local $\\ = ".\\n";
print "apples", "pears", "plums"; # "apples, pears, plums.\\n"
```

A list containing undef - each undef stringifies to the empty string and triggers an uninitialized warning under use warnings:

```
use warnings;
my $x;
print "value=", $x, "\\n"; # "value=\\n" plus a warning
```

List context vs scalar context. @arr in LIST position expands to its elements; wrap in scalar to print the count instead:

```
my @arr = (10, 20, 30);
print @arr, "\\n"; # "102030\\n"
print scalar @arr, "\\n"; # "3\\n"
```

Suppressing the undef warning for a block where missing values are expected and intended to render as empty:

```
no warnings 'uninitialized';
print join(",", @row), "\\n"; # empty fields render as ""
```

Edge cases

- **No arguments:** print; prints \$_ to the selected handle. Used idiomatically inside while (<>) { print; }.
- **The filehandle-or-first-arg ambiguity:** Perl decides whether the first token is a filehandle by parsing, not by runtime type. A bareword or a simple scalar variable immediately followed by a term (no comma) is taken as a filehandle:

```
print STDERR "oops\\n"; # STDERR is the filehandle
print $fh "oops\\n"; # $fh is the filehandle
```

An expression that returns a filehandle does **not** work in that position - print \$handles[0] "x" is a syntax error. Use the block form: print { \$handles[0] } "x".

- **List context applies to every argument.** A subroutine call in LIST runs in list context, not scalar:

```
print localtime(), "\n";           # prints nine numbers, no_
↪spaces
print scalar localtime(), "\n";    # prints the familiar date_
↪string
```

- **print (...) parses as a function call.** `print (2+3)*4` prints 5, then discards `*4`. Parenthesise the whole argument list, or prefix with `+`:

```
print((2+3)*4);                   # prints 20
print+(2+3)*4;                    # prints 20
```

- **Closed filehandle:** returns `undef` and sets `$!` to "Bad file descriptor". Under use warnings a `print()` on closed filehandle warning is emitted.
- **Encoding layers:** `print` writes whatever the handle's PerlIO stack says to write. If the handle has a `:utf8` or `:encoding(...)` layer, strings are encoded on the way out. Bytes-mode handles expect byte strings; passing a wide-character string writes raw codepoints and emits a `Wide character in print` warning.
- **Autoflush:** `print` buffers through PerlIO. Set `$|` on the selected handle to flush after every write - commonly needed for progress output on pipes or when coordinating with another process:

```
$| = 1;                           # autoflush current handle
```

- **Printing to a closed pipe or socket** raises `SIGPIPE`. The default action is to terminate the process; install a handler (`$SIG{PIPE} = 'IGNORE'` or a sub) to turn it into a normal write failure you can check.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `say` - `print` with `$\` forced to `"\n"` for the call; use it when every record ends in a newline
- `printf` - same targeting rules as `print`, but the first argument is a `sprintf` format string and `$\` is **not** appended
- `sprintf` - build the formatted string without writing it; combine with `print` when you want to log it somewhere else first
- `select` - change which filehandle the no-argument form of `print` writes to
- `$,` - output field separator inserted between list items
- `$\` - output record separator appended after the list
- `$|` - autoflush flag for the currently selected handle

I/O

printf

Write a formatted string to a filehandle.

`printf` is `print FILEHANDLE sprintf(FORMAT, LIST)` with one deliberate exception: the output record separator `$\` is **not** appended. The targeting rules are otherwise the `print` rules - bareword or simple-scalar filehandle immediately before the argument list, or the `{EXPR}` block form for anything more complex. If `FILEHANDLE` is omitted, output goes to the currently selected handle (see `select`), which starts life as `STDOUT`.

See `sprintf` for the format-specifier reference - everything `%d`, `%s`, `%.3f`, `%.3d`, `%1$s`, `%v`, vector flags, locale-sensitive `%f`, and so on. This page covers only the `printf`-specific surface: filehandle resolution, the format-is-first rule, and the quirks around the no-argument form.

Synopsis

```
printf FORMAT, LIST
printf FILEHANDLE FORMAT, LIST
printf {EXPR} FORMAT, LIST
printf FH                # bareword filehandle, $_ as format
```

What you get back

1 on success, `undef` on failure (with `$!` set). As with `print`, the return value is rarely checked for writes to `STDOUT` / `STDERR`, and worth checking for writes to files, pipes, and sockets where partial writes and broken pipes matter.

```
printf $fh "%-20s %8d\n", $name, $count
or die "write to $path failed: $!";
```

Global state it touches

- **Selected filehandle** (changed via `select`): the destination when `FILEHANDLE` is omitted. Starts as `STDOUT`.
- `$\` (output record separator): deliberately **not** appended - this is the one semantic difference between `printf FORMAT, LIST` and `print FILEHANDLE sprintf(FORMAT, LIST)`. Put the newline in the format string.
- `$,` (output field separator): **not** consulted. `printf` splices list elements into the format according to the conversions, not with a separator.
- `$_`: used as the **format** when both `FORMAT` and `LIST` are omitted, not appended to output. Almost never what you want; see *Edge cases*.
- `$!:` set on write failure.
- `$|`: autoflush flag for the selected handle. `printf` buffers through PerlIO like `print`.

- `LC_NUMERIC` locale: when use `locale` is active and `POSIX::setlocale` has been called, the decimal separator for floating-point conversions (`%f`, `%e`, `%g`) follows the locale.

Examples

Basic formatted write to `STDOUT`. Unlike `say` or `print-with-$\`, the trailing newline is on you:

```
printf "%s is %d years old\n", $name, $age;
```

Width and precision for a tidy column:

```
printf "%-20s %8.2f\n", $item, $price;
```

Write to a filehandle. Bareword and simple scalar work directly; no comma between the filehandle and the format:

```
open my $fh, ">", "report.txt" or die $!;
printf $fh "%5d %s\n", $n, $msg;
close $fh;
```

Filehandle held in a more complex expression - use the `{EXPR}` block form (same rule as `print`):

```
my @logs = (\*STDOUT, \*STDERR);
printf { $logs[$severity] } "[%s] %s\n", $tag, $msg;
```

Repeated format with a list that spans multiple records. `printf` does **not** re-apply the format per record - conversions are consumed once, left to right:

```
printf "%s=%s\n", "user", "alice";           # "user=alice\n"
printf "%s=%s\n%s=%s\n",                    # two records, one call
      "user", "alice", "host", "db1";
```

Locale-sensitive decimal separator:

```
use POSIX qw(setlocale LC_NUMERIC);
use locale;
setlocale(LC_NUMERIC, "de_DE.UTF-8");
printf "%.2f\n", 1234.5;                      # "1234,50\n"
```

Edge cases

- **No LIST given, `$_` becomes the format.** `printf`; and `printf FH`; (bareword filehandle) format `$_`. If `$_` contains any `%` conversions they expand against an empty argument list - each `%s` / `%d` becomes the empty string and `warnings` emits a `Missing argument in printf` warning. To write `$_` verbatim use `print`, not `printf`.
- **Indirect filehandle without a list is ambiguous** and not supported. `printf $fh`; parses `$fh` as the format, not the filehandle. If you need a no-argument form with a

scalar filehandle, go through `print` or pass an explicit format: `printf $fh "%s", $_`.

- **The first argument is always the format.** `printf(@_)` uses `$_[0]` as the format and the remaining elements as arguments - this is a parsing rule, not a convention. Build the format deliberately; don't splat user-supplied lists into `printf` unless the first element is known-safe.
- **No `$\` is appended.** Embedding `"\n"` in the format is the idiomatic terminator. Setting `local $\ = "\n"` has no effect on `printf` output.
- **`$,` is ignored.** The output field separator is a `print` concept; `printf` joins nothing - it substitutes.
- **`printf (...)` parses as a function call**, just like `print`. `printf ("%d", 1)*2` computes `printf(...)` then multiplies the return value by 2 and discards it. Wrap the whole argument list in parens or prefix with `+` if you need to disambiguate.
- **Closed filehandle:** returns `undef` and sets `!` to "Bad file descriptor". Under `use warnings` a `printf()` on closed filehandle warning is emitted.
- **Encoding layers:** `printf` writes through the handle's PerlIO stack. A `:utf8` or `:encoding(...)` layer encodes on the way out; a bytes-mode handle with a wide-character string emits Wide character in `printf` and writes raw code-points.
- **SIGPIPE on closed pipe/socket:** default action terminates the process. Install `$SIG{PIPE}` to turn it into a checkable write failure.
- **Don't use `printf` where `print` suffices.** `printf "%s", $msg` is slower and more error-prone than `print $msg` - the format string is parsed on every call, and a stray `%` in `$msg` used as a format becomes a bug waiting to happen.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `print` - unformatted write with the same filehandle rules; faster, and appends `$\` where `printf` does not
- `sprintf` - build the formatted string without writing it; the format-specifier reference lives here
- `say` - `print` with `$\` forced to `"\n"`; reach for it when every record ends in a new-line
- `select` - change which filehandle the no-filehandle form of `printf` writes to
- `$\` - output record separator; deliberately not appended by `printf`
- `$_` - used as the format when both `FORMAT` and `LIST` are omitted

Misc

prototype

Return the prototype string of a subroutine, or `undef` if it has none.

`prototype` inspects a subroutine and gives back the prototype the parser will use when compiling calls to it. `FUNCTION` is either a name (bareword or string) or a code reference. With no argument, the subroutine named by `$_` is looked up. A `CORE::` prefix asks for the prototype-equivalent of a core built-in.

Prototypes are the old compile-time parsing hints declared with `sub foo ($$@) { . . . }`. They control how the parser shapes calls - they are **not** subroutine signatures (`sub foo ($x, $y) { . . . }`), which are a separate feature that `prototype` does not report on.

Synopsis

```

prototype FUNCTION
prototype
prototype \&foo
prototype 'Some::Package::foo'
prototype 'CORE::push'

```

What you get back

A string describing the prototype, or `undef` when:

- the subroutine exists but has no prototype declared, or
- the name resolves to a `CORE::` built-in whose argument shape the prototype mini-language cannot express (for example `system`, `print`, `sort`).

The string is exactly the text that appeared between the parentheses in the sub declaration, canonicalised: whitespace removed, each character one of the prototype letters below.

Character	Meaning in the prototype
\$	one scalar argument
@	list - absorbs all remaining arguments
%	hash - absorbs all remaining arguments as a list
&	code block or code reference
*	typoglob or bareword treated as one
\\$ \@ \% \& *	reference to a scalar / array / hash / code / glob (caller writes @ary, callee receives \@ary)
\[. . .]	reference where any of the bracketed sigils is accepted
+	array or hash, passed by reference automatically
_	scalar that defaults to <code>\$_</code> if omitted
;	subsequent arguments are optional

A leading `&` with no parentheses after it, as in `sub f (&) { . . . }`, is what lets `map`, `grep` and `sort`-lookalikes take a block without a comma.

Global state it touches

prototype with no argument reads `$_` as the subroutine name. No other globals are consulted or modified.

Examples

Read a user-defined prototype:

```

sub add ($$) { $_[0] + $_[1] }
print prototype 'add';           # $$
print prototype \&add;           # $$

```

A sub with no prototype returns `undef`:

```

sub plain { "hi" }
print defined(prototype 'plain') ? "yes" : "no"; # no

```

A `CORE::` query for a built-in whose shape fits the prototype language:

```

print prototype 'CORE::push';    # \@@
print prototype 'CORE::keys';    # \[%@]
print prototype 'CORE::pos';     # ${\;}

```

A `CORE::` query for a built-in that cannot be expressed as a prototype:

```

print defined(prototype 'CORE::system') ? "yes" : "no"; # no
print defined(prototype 'CORE::print') ? "yes" : "no"; # no

```

Use it to build a wrapper that inherits the callee's calling convention - useful when writing generic dispatchers:

```

sub wrap {
  my $name = shift;
  my $proto = prototype $name;
  # install a new sub with the same prototype string
  eval "sub wrapped_$name ($proto) { goto &$name }";
}

```

Edge cases

- **No argument uses `$_`.** prototype with nothing between it and the next term consults `$_` as the sub name. A loop like `for (@names) { print prototype, "\n" }` works by accident of this rule.
- **Signatures are not prototypes.** `sub f ($x, $y) { ... }` declares a signature-prototype `'f'` returns `undef`, because the sub has no prototype. To declare both, use the `:prototype(...)` attribute: `sub f :prototype($$) ($x, $y) { ... }`.
- **Unknown name returns `undef`.** If the sub has never been declared, prototype returns `undef` without raising an error. Combine with `defined` and `exists`

`&Some::Package::name` when you need to tell “no prototype” apart from “no such sub”.

- **Auto-loaded subs.** If the sub is handled by `AUTOLOAD`, it has not been compiled yet, so `prototype` reports `undef` even when the eventual callable would have one. Force a stub first (`*foo = \&foo` after loading) if you must query it.
- **References versus names are interchangeable.** `prototype \&foo` and `prototype 'foo'` return the same string; pick whichever is cheaper at the call site. A reference skips package-name lookup.
- **Fully qualified names.** Unqualified names are looked up in the current package. For callers-of-callers, always qualify: `prototype 'Other::Pkg::fn'`.
- **Empty prototype is not `undef`.** `sub nullary () { 42 }` gives `prototype 'nullary'` the empty string `''`, which is defined and false. Use `defined`, not `truthiness`, to test for “has a prototype”.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `sub` - declare the subroutine whose prototype you are about to read
- `defined` - the correct test for “does this sub have a prototype,” since an empty-string prototype is valid
- `ref` - companion inspector when `FUNCTION` is a code reference and you want to verify the `ref` type before calling `prototype`
- `caller` - neighbour introspector that answers “who called me,” often used with `prototype` when building wrappers
- `wantarray` - runtime context query, the counterpart to compile-time prototype inspection
- `perlsub` - the prototype mini-language and how the parser applies it to call sites

Arrays

push

Append one or more values to the end of an array.

`push` adds every element of `LIST` to the end of `ARRAY`, in order. It is the paired counterpart of `pop` at the tail and of `unshift` / `shift` at the head. The array grows in place; no new array is returned.

Synopsis

```
push @arr, $val
push @arr, @more
```

(continues on next page)

(continued from previous page)

```
push @arr, list, of, values
push @$aref, $val
```

What you get back

The **new number of elements** in ARRAY after the append - i.e. scalar @arr as it stands once push returns. Not the pushed values, not the old length, not the array itself.

```
my @colors = ("red");
my $n = push @colors, "blue", "green";    # $n is 3
```

Useful when the push and the subsequent length check would otherwise be two statements:

```
if (push @queue, $job > $limit) { ... }    # one call, count + test
```

Examples

Basic append of a single value:

```
my @animals = ("cat");
push @animals, "mouse";                    # ("cat", "mouse")
```

Append several values in one call - each argument becomes its own element:

```
my @colors = ("red");
push @colors, "blue", "green";             # ("red", "blue", "green")
```

Append the contents of another array. The second array is flattened into the list, so its **elements** are appended, not the array as a single item:

```
my @a = (1, 2);
my @b = (3, 4);
push @a, @b;                              # (1, 2, 3, 4)
```

Queue discipline - push at the tail, `shift` at the head:

```
my @queue;
push @queue, $job;                         # enqueue
my $next = shift @queue;                   # dequeue
```

Stack discipline - push and `pop` at the same end:

```
my @stack;
push @stack, $frame;                       # enter
my $top = pop @stack;                      # leave
```

Push onto an array held by reference. The reference must be dereferenced with `@{ ... }` (or the short `@$ref` form) before push sees an array:

```
my $aref = [1, 2];
push @$aref, 3;           # $aref now refers to [1, 2, 3]
push @{ $aref }, 4, 5;    # same, explicit deref
```

Count elements after appending without a temporary:

```
my $size = push @log, $entry;
warn "log is large: $size entries" if $size > 1000;
```

Edge cases

- **Arrayref needs a deref.** `push $ref, $x` is a compile-time error in Perl 5.24+. Use `push @$ref, $x` or `push @{ $ref }, $x`. An experimental feature allowed a scalar first argument in 5.14–5.22 and was removed; do not rely on it.
- **Array arguments auto-flatten.** `push @a, @b` appends @b's elements individually. To append @b as a single nested entry you need an arrayref: `push @a, \@b` or `push @a, [ @b ]`.
- **Hash arguments also flatten.** `push @a, %h` appends alternating keys and values in hash iteration order - almost never what you want. Use `push @a, \%h` to store the hash by reference instead.
- **List context but scalar return.** `push` is documented as taking a LIST and therefore evaluates its arguments in list context, but the value `push` itself produces is always the resulting count. Inside `print push @a, @b` the count is what gets printed, not the array.
- **Empty LIST.** `push @a;` (no second argument) is a no-op and returns the current scalar `@a`. Legal and occasionally useful for “give me the count without modifying anything” - though scalar `@a` is clearer.
- **Tied arrays.** `push` on a tied array calls `PUSH` on the tied object if defined; otherwise it falls back to repeated `STORE` calls at `FETCHSIZE`-derived indices. See `perltie`.
- **Autovivification.** Pushing through an undef-valued reference vivifies an anonymous array:

```
my %index;
push @{ $index{$key} }, $value;           # $index{$key} becomes
↳ an arrayref
```

After the first push, `$index{$key}` holds an arrayref containing `$value`. This is the standard idiom for building a hash-of-arrays incrementally.

- **Readonly array.** Pushing onto a readonly or fixed-size array (e.g. an element of `@_` bound to a caller's variable, or an array locked with `Hash::Util::lock_value`) raises `Modification of a read-only value attempted`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `pop` - remove and return the last element; paired with `push` to form a stack
- `shift` - remove and return the first element; paired with `push` to form a queue
- `unshift` - `push`'s mirror image at the head of the array; returns the new length the same way
- `splice` - insert, remove, or replace any range; `push @a, @b` is equivalent to `splice @a, scalar @a, 0, @b`
- `scalar` - force scalar context on an array when you want the count without a `push`
- `@ARGV` - the canonical array most programs push onto or consume with `shift`

SCALARS and strings

q//

Single-quoted string literal with a choice of delimiter.

`q//` is the generic form of the customary `'...'` string literal. It produces a compile-time constant string, does **not** interpolate variables, and does **not** process the backslash escape sequences that `qq//` and regex literals recognise. Reach for `q//` whenever the literal you want to write would be awkward under `'...'` - most often because the string itself contains single quotes - or when a different delimiter simply reads better in context.

Synopsis

```
q/STRING/
q(STRING)
q[STRING]
q{STRING}
q<STRING>
q!STRING!
q#STRING#
'STRING'
```

Any non-alphanumeric character works as a delimiter. The four ASCII bracket pairs `((`), `[]`), `{}`), `<>`) nest; all other delimiters do not. `'...'` is a shorthand for `q(...)` with identical semantics.

What you get back

A string in scalar context - the exact byte sequence between the opening and closing delimiter, with two rewrites:

- `\\` becomes a single backslash.

- `\<delim>` becomes a literal copy of the delimiter character (this is what lets you include the delimiter inside the body).

Every other character, including every other `\`, stays verbatim. There is no `\n`, no `\t`, no `$var` expansion, no `\x{...}`, no `\N{...}` - the inside of `q//` is as close to “what you typed is what you get” as Perl offers.

In list context `q//` still evaluates to the single string.

Delimiters and nesting

Non-bracketing delimiters use the same character fore and aft:

```
my $a = q!I said, "You said, 'She said it.'";
my $b = q/path/to/file/;    # WRONG - ends at the second "/"
my $b = q{path/to/file};    # RIGHT - braces nest, slashes are
    ↪ literal
```

Bracket delimiters nest, so balanced brackets inside the body are fine and only an unbalanced closer ends the string:

```
my $c = q{foo{bar}baz};    # "foo{bar}baz"
my $d = q(f(o(o)b)ar);    # "f(o(o)b)ar"
```

Whitespace between `q` and the delimiter is allowed **except** when the delimiter is `#` - `q#foo#` is the string `"foo"`, but `q #foo#` is the operator `q` followed by a line comment, and the string body continues on the next line. When the delimiter is a word character (matches `/w/`), whitespace between `q` and the delimiter is **required**: `q XfooX` is `"foo"`, `qXfooX` is a syntax error.

Escape rules

Inside `q//` the backslash has exactly two special jobs:

Sequence	Produces
<code>\\</code>	a single backslash
<code>\<delim></code>	a literal copy of the delimiter

Anything else beginning with `\` stays as two characters:

```
print q(\n), "\n";        # prints "\n" then a real newline
print length q(\n);       # 2
print q(\t\r\0);          # prints "\t\r\0" - six characters
```

This is the single most frequent source of confusion for authors coming from double-quoted strings - in `q//` and `'...'`, `\n` is a backslash followed by the letter `n`, **not** a newline.

Examples

Embedding both quote marks without escaping:

```
my $msg = q{I said "hello", she said 'hi'.};
```

Windows-style path, where backslashes are plentiful and double-quote interpolation would be a hazard:

```
my $path = q(C:\Users\alice\Documents);
print $path;                # C:\Users\alice\Documents
```

A literal \$ without interpolation - qq// would require escaping every \$ and @, q// leaves them alone:

```
my $expr = q($total = $x + $y);
print $expr;                # $total = $x + $y
```

The one-and-only way to put the delimiter inside the body: escape it with a backslash. Note that the backslash itself is consumed.

```
my $s = q!He said "ouch\!" loudly!;
print $s;                    # He said "ouch!" loudly
```

A regex pattern stored as a plain string, deferred to qr// at use site - q// keeps backslashes literal so the pattern survives intact:

```
my $pat = q(\d{3}-\d{4});
if ($phone =~ /$pat/) { ... }
```

Multi-line literal with no escape processing - each line break in the source becomes a line break in the string:

```
my $sql = q{
    SELECT id, name
    FROM   users
    WHERE  active = 1
};
```

Edge cases

- **q// is compile-time.** The parser resolves the string when the source is read; there is no runtime cost beyond using the already- built constant. Same as '... '.
- **'...' and q(...) are indistinguishable** after parsing. The choice is purely stylistic - pick whichever reads better for the content.
- **Empty string.** q() and '' both yield the zero-length string.
- **No \n, no \t, no interpolation,** ever. If you need either, switch to qq// or "...". Mixing the two in one line is fine: q(literal) . qq(interpolated \$x).
- **Backslash at end of body.** A single \ immediately before the closing delimiter escapes it, so q(foo\) is a syntax error - the \) becomes a literal) and the string is unterminated. Write q(foo\\) to end a string with one backslash.

- **Unbalanced brackets with bracket delimiters.** Because brackets nest, `q{foo{bar}` is a syntax error - the inner `{` opens a nested level and the outer `}` closes it, leaving the string unterminated. Use a non-bracket delimiter, or balance the brackets, or escape one of them: `q{foo\{bar}`.
- **# as delimiter needs no space from q,** but every other delimiter that is a word character (letters, digits, `_`) requires whitespace between `q` and the opening delimiter.
- **Quoting Perl code is fragile.** `q{ if ($x eq "}") { ... } }` is a syntax error: the `}` inside the embedded double-quoted string closes the outer brace before Perl sees the inner string. Tools like `Text::Balanced` exist because hand-writing a robust Perl source quoter is unpleasant.
- **Parser ambiguity with / as delimiter.** `q/.../` uses `/` the same way `m//` does. Inside the body no regex syntax applies, but a slash in the content will end the string prematurely - pick a different delimiter (`q{...}`, `q(...)`) when slashes appear in the text.
- **Heredocs with <<'TAG'** follow the same escape rules as `q//`: no interpolation, no escape processing beyond `\\` and `\TAG` not being special (the tag terminates on its own line). Use a single-quoted heredoc when you want `q//` semantics across many lines without a closing delimiter in the middle of the text.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `qq//` - double-quoted literal; same delimiter flexibility as `q//` plus variable interpolation and the full `\n\t\x{...}` escape vocabulary
- `qw//` - list of whitespace-separated words; what you want when the literal is a list of short tokens, not a single string
- `qr//` - compile a regex literal; the pattern-matching member of the quote-like family
- `qx//` - run a command in the shell and capture its output; same delimiter rules, semantics of backticks
- `quotemeta` - runtime equivalent of `\Q... \E`, useful when you need to feed a `q//` literal into a regex as a fixed string
- `$` - list separator used when arrays interpolate inside `qq//`; irrelevant to `q//` itself but the usual reason someone reaches for `q//` is to **avoid** this machinery

SCALARs and strings

qq//

Build a double-quoted, interpolated string with a delimiter you pick.

`qq//` is the long form of the `"..."` string literal. It produces the same string `"` would, so `qq/hello $name\n/` and `"hello $name\n"` are interchangeable. The point of `qq` is

the delimiter: when the content contains " (or backslash sequences that would interfere with double quotes), you pick a different delimiter and keep the string readable instead of threading \" through it.

Synopsis

```
qq/STRING/
qq{STRING}
qq(STRING)
qq[STRING]
qq<STRING>
qqXSTRINGX      # any non-word, non-whitespace char works as
↳delimiter
```

What you get back

A scalar string. Variables referenced inside are interpolated at the point qq// is evaluated (not at parse time); backslash escape sequences are expanded. The result has no implicit newline - you have to write \n yourself.

Choosing a delimiter

Any character that is not a word character (/w/) and not whitespace may follow qq. The four bracket pairs nest:

```
qq(foo (bar) baz)      # "foo (bar) baz" - inner () balanced
qq{ if ($x) { 1 } else { 0 } }
qq[a[b[c]d]e]
qq<a<b>c>>
```

Non-bracketing delimiters use the same character at both ends and do not nest:

```
qq|one|two|           # SYNTAX ERROR - first | closes
qq/a\b/               # "a/b" - backslash escapes the delimiter
```

If the delimiter is a word character, whitespace between qq and the delimiter is required:

```
qq XfooX              # OK - "foo"
qqXfooX               # WRONG - parsed as identifier qqXfooX
```

as delimiter needs no space and is occasionally handy for strings containing every other punctuation mark:

```
qq#He said "hi" & waved.# # "He said \"hi\" & waved."
```

The `extra_paired_delimiters` feature (5.36+) recognises a wide set of Unicode bracket pairs as nesting delimiters; enable it with use feature 'extra_paired_delimiters' when you want, say, guillemets or CJK brackets.

What gets interpolated

- **Scalars:** `$x`, `$hash{key}`, `$array[0]`, `$ref->{k}[2]`.
- **Arrays:** `@arr` expands in order, elements joined by `$` (default `" "`). Equivalent to `join $", @arr`.
- **Slices:** `@h{qw(a b)}`, `@a[1..3]`.
- **Dereferences with arrows and subscripts:** `$ref->{key}[0]`.

What is **not** interpolated:

- **Method calls:** `$obj->meth` inside `qq//` interpolates `$obj` and leaves the literal text `->meth` in the string.
- **Function calls:** `@{[ func() ]}` is the usual workaround - array reference dereference runs arbitrary code and splices its result in.
- **Punctuation arrays** other than `@_`, `@+`, `@-`: only interpolate when braced (`@{*}`). Bare `@*` is a literal `@*`.

```
my $obj = Some::Class->new;
qq/$obj->name/;           # interpolates $obj, leaves "-
    <->name"
qq/@{[ $obj->name ]}/;    # calls the method,
    <->interpolates result
```

Escape sequences

Available inside `qq//` (and anywhere else interpolation happens):

Sequence	Meaning
<code>\t</code>	tab (HT)
<code>\n</code>	newline (platform-virtual line terminator)
<code>\r</code>	carriage return (CR)
<code>\f</code>	form feed (FF)
<code>\b</code>	backspace (BS)
<code>\a</code>	alarm / bell (BEL)
<code>\e</code>	escape (ESC)
<code>\0</code>	NUL
<code>\\</code>	literal backslash
<code>\"</code>	literal double quote
<code>\x1b</code>	hex char, 0x00–0xFF
<code>\x{263A}</code>	hex char, any codepoint (braced form)
<code>\o{23072}</code>	octal char, any codepoint
<code>\033</code>	octal char, 000–777 (prefer <code>\o{ }</code>)
<code>\cA</code>	control char - 64 xor'd with uppercased next char
<code>\N{GREEK SMALL LETTER ALPHA}</code>	named Unicode char (needs use <code>chardnames</code>)
<code>\N{U+03B1}</code>	Unicode char by codepoint (always Unicode, even on EBCDIC)

Case and quoting modifiers apply until \E or end of string:

Sequence	Meaning
\l	lowercase the next character only
\u	titlecase (not uppercase!) the next character only
\L	lowercase until \E
\U	uppercase until \E
\F	foldcase until \E
\Q	quote regex metacharacters until \E (calls <code>quotemeta</code>)
\E	end the most recent \L / \U / \F / \Q

\L, \U, \F, \Q stack - each needs its own \E.

```
say "\u$name";           # capitalise first letter
say "\U$name\E's account"; # uppercase $name, rest untouched
say "match $file literally: \Q$file\E";
```

\Q inside qq// is applied **after** interpolation, so qq/\Q\$pat\E/ equals `quotemeta($pat)`. (In regex quote operators qr// and m//, \Q is applied after interpolation but before regex compilation - the difference matters when \$pat itself contains regex metacharacters you want to keep literal.)

Perl has no \v escape in string literals - it is a regex metacharacter. Use \x0b, \cK, or \N{VT} for a vertical tab.

Global state it touches

- `$"` - list separator used when interpolating arrays and slices. Default `" "`. Change with `local $" = ", "` for one scope.

No other interpreter globals participate in qq// evaluation itself. Interpolated variables are of course whatever their current values are at the point qq// runs.

Examples

Plain interpolation, identical in meaning to the `"..."` form:

```
my $name = "world";
my $s = qq/hello, $name!\n/;           # "hello, world!\n"
```

String that contains double quotes - qq with a different delimiter avoids backslash litter:

```
my $msg = qq{He said "yes" and then "maybe."};
# vs:   "He said \"yes\" and then \"maybe\"."
```

Multi-line HTML snippet with nested braces - the `()` pair nests:

```
my $user = "rj";
my $html = qq(
```

(continues on next page)

(continued from previous page)

```
<p>Welcome, $user.</p>
<p>You have @{{ scalar @inbox }} messages.</p>
);
```

Array interpolation with a custom separator:

```
my @tags = qw(perl rust docs);
local $" = ", ";
say qq/tags: @tags/;           # "tags: perl, rust, docs"
```

Method-call trap -> is literal inside qq//:

```
my $obj = File::Spec->new;
qq/path is $obj->rel2abs('.')/;      # "path is <stringified $obj>-
    ↪->rel2abs('.')"
qq/path is @{{ $obj->rel2abs('.') }}/; # actually calls rel2abs()
```

Building an SQL fragment where \$ident is untrusted - \Q..\E keeps embedded %, _ and backslashes literal under a LIKE clause only after you have handled proper parameterisation; for actual SQL escaping use the driver's quoting, not \Q:

```
my $needle = "100% sure";
my $pat = qq/\Q$needle\E/;          # the regex-safe form
# $pat eq "100\\%\\ sure"
```

Stacked modifiers:

```
say qq/\Qfoo \ubar \Ubaz\E qux\E done/;
# "foo\ Bar\ BAZ qux done"
# \Q quotes spaces/?; \u titlecases b; \U uppercases baz until \E;
# outer \E ends \Q.
```

Edge cases

- **Parse-time delimiter choice.** Perl picks the closing delimiter the moment it reads the opening one. qq/.../ expects a /; there is no way to switch mid-string. If you need both kinds of braces, pick a non-bracket delimiter.
- **Backslash at end of string with non-bracket delimiter.** qq/a\ is a/ (backslash escapes the delimiter, string continues); to get a literal trailing backslash, change delimiter: qq{a\\}.
- **No nesting for non-bracket delimiters.** qq|a|b| is the string a followed by the identifier b followed by | - a syntax error in most contexts. Pick bracket delimiters when the content contains the delimiter char.
- **# as delimiter** is a special case only because of how comments interact: qq#...# is a string, but qq #...# (with whitespace) is the operator qq followed by a comment, with the string taken from the next line. This mirrors the behaviour of all q-family operators.

- **Empty string.** `qq//` is the empty string, same as `""`.
- **Interpolated `undef`.** Stringifies to the empty string and, under `use warnings`, triggers an uninitialized warning at the point of interpolation, not at the point `$var` was assigned `undef`.
- **\$ or @ followed by non-identifier.** Stays literal: `qq/cost: \$/` and `qq/cost: $/` both produce `cost: $/` (there is no variable named `$5` in the Perl 5 identifier sense - `$5` is a regex capture, which does exist, so the second form interpolates capture group 5, whatever it currently is). For a literal `$` or `@`, escape it: `\$, \@`.
- **`\N{...}` without `charnames`.** Named characters require `use charnames 'full'` (or similar) to be loaded in scope; without it you get a compile-time error. `\N{U+HEX}` and the single-name forms in the `:loose` set work without it.
- **Locale-sensitive case mapping.** With `use locale` including `LC_CTYPE`, `\l`, `\u`, `\L`, `\U` use the current locale's case tables. Without it, Unicode case tables are used for codepoints above `0xFF`, and ASCII rules below. `\F` uses Unicode foldcase regardless of locale (except under a UTF-8 locale, where it matches `\L`).
- **One level only.** The result of interpolation is not re-scanned. `my $v = '$x'; qq/$v/` is the four-character string `$x`, not the value of `$x`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `q//` - single-quoted literal, no interpolation, same delimiter rules
- `qw//` - word-list literal, produces a list of strings split on whitespace
- `qx//` - interpolated string executed as a shell command, captures stdout
- `qr//` - interpolated regex, compiled once and reusable
- `sprintf` - reach for this when the work is formatting numbers or padding fields, not splicing variables into text
- `quotemeta` - the function `\Q... \E` calls
- `join` - what array interpolation is secretly doing, with `$"` as the separator
- `$"` - list separator used when an array interpolates into a double-quoted string
- *Quote and Quote-like Operators* - the full table of `q`, `qq`, `qx`, `qw`, `qr`, `m`, `s`, `tr` and their shared delimiter and escape rules

Regular expressions and pattern matching

qr//

Compile a pattern once and hand back a reusable regex object.

`qr//` quotes its `STRING` as a regular expression, interpolates it the same way `m` does, and returns a `Regexp` object you can store, pass around, and splice into other patterns. The

compiled pattern carries its modifiers with it, so every later use sees the flags the pattern was built with.

Synopsis

```
my $re = qr/STRING/;
my $re = qr/STRING/msixpodualn;
my $re = qr{STRING};
my $re = qr'String';           # single-quote delimiter: no_
↪ interpolation
```

What you get back

A blessed Regexp object. `ref` returns the string "Regexp". Stringifying the object yields a normalised form of the pattern with its modifiers encoded in a `(?flags:...)` wrapper, which is exactly the form the regex engine embeds when the object is interpolated into another pattern:

```
my $re = qr/my.STRING/is;
print $re;           # (?^si:my.STRING)
```

The object is opaque - do not dereference it. The stringified form is intended for interpolation and debugging, not for manual parsing.

Global state it touches

- `$1, $2, ..., $+, $&, $`, $', %+, %-` - **not** set by `qr//` itself. Compilation does not match. These are populated only when the compiled pattern is later used by `m`, `s`, or `split`.
- `$@` - set if the pattern contains a syntax error and the error is trapped with `eval`. An untrapped compile error raises a fatal Bareword "... " not allowed-style exception at the point of the `qr//`.
- Locale and Unicode pragmas in effect at the `qr//` site (use `locale`, use `utf8`) are baked into the compiled object. Changing the pragma later does not change the object.

Modifiers

All modifiers that apply to `m` also apply to `qr / /`. They are captured in the returned object and propagate when the object is interpolated into another pattern.

Flag	Meaning
<code>m</code>	Multi-line: <code>^</code> and <code>\$</code> match at internal newlines.
<code>s</code>	Single-line: <code>.</code> matches a newline.
<code>i</code>	Case-insensitive match.
<code>x</code>	Extended: whitespace and <code>#</code> comments ignored in the pattern. <code>xx</code> also ignores whitespace inside <code>[. . .]</code> .
<code>p</code>	Preserve-match (no-op on Perl 5.20+; <code>\${^PREMATCH}</code> , <code>\${^MATCH}</code> , <code>\${^POSTMATCH}</code> are always available).
<code>o</code>	Compile once; interpolated variables are frozen at first use. Rarely needed - <code>qr / /</code> already caches.
<code>a</code>	ASCII-restrict <code>\d</code> , <code>\s</code> , <code>\w</code> , and POSIX classes. <code>aa</code> further forbids ASCII-to-non-ASCII matches under <code>i</code> .
<code>l</code>	Use the current locale's rules.
<code>u</code>	Use Unicode rules.
<code>d</code>	Default dual rules (legacy; usually picked up automatically).
<code>n</code>	Non-capture: unnamed (. . .) groups do not fill <code>\$1</code> , <code>\$2</code> , ...

When a compiled pattern is embedded in a larger pattern, the character-set and `msixn` flags it was built with stay in effect for that span only. The `o` modifier is the one exception - it is not propagated.

Examples

Compile once, use many times. Looping over patterns built ahead of time avoids recompiling on every iteration:

```
my @rules = map qr/$_/i, qw(error warn fail panic);
for my $line (@lines) {
    for my $rx (@rules) {
        print $line if $line =~ $rx;
    }
}
```

Interpolate a compiled pattern inside another pattern. The outer pattern sees the inner pattern's flags via the `(?flags: . . . )` wrapper:

```
my $word = qr/\w+/;
my $csv = qr/^$word(?:,$word)*$/;
"alpha,beta,gamma" =~ $csv;           # matches
```

Single-quote delimiter for a literal pattern with no interpolation - useful when the pattern contains `$` or `@` that must stay literal:

```
my $price = qr'\$ \d+ \. \d{2}';      # matches "$1.99", not a sigil
"total: $9.95" =~ $price;           # matches
```

Pattern re-use across calls. Storing the compiled form in a closure amortises compilation across every call to the returned sub:

```
sub matcher_for {
    my $pat = shift;
    my $rx = qr/\Q$pat\E/i;          # \Q...\E quotes regex_
    ↪metacharacters
    return sub { $_[0] =~ $rx };
}

my $is_error = matcher_for("ERROR:");
$is_error->($line);
```

Capture inside an interpolated qr// still populates \$1 at match time, not at qr// time:

```
my $num = qr/(\d+)/;
"port 8080" =~ /:$num$/;
print $1;                      # 8080
```

Using /n to suppress capture in \$1, \$2, ... while keeping named captures available:

```
my $rx = qr/(foo)(?<kw>bar)/n;
"foobar" =~ $rx;
print $+{kw};                  # "bar"
print defined $1 ? "yes" : "no"; # "no"
```

Edge cases

- **No match at compile time.** qr// only *compiles*. To test a string, use the object with `m` or `=~`: `$string =~ $rx`.
- **Interpolated variables are captured at qr// time**, not at use time. `my $rx = qr/$pat/; $pat = "other";` leaves `$rx` holding the pattern built from the original `$pat`. To re-capture, rebuild the qr//.
- **Compile-time pattern errors are fatal.** qr/(/ raises an exception at the qr// site. Wrap in `eval` when the pattern comes from user input and an error should be recoverable:

```
my $rx = eval { qr/$user_input/ };
die "bad pattern: $@" if $@;
```

- **ref(\$rx) returns "Regexp"** - not "SCALAR", not "CODE". Use this to detect a compiled pattern in a polymorphic argument slot: `ref($arg) eq 'Regexp'`.
- **Stringification is lossy for re-parsing.** The normalised form is legal as an interpolation target but is not guaranteed to be a human-readable copy of the original source. Do not re-qr// the stringified output.
- **Locale and Unicode bake-in.** A qr// built inside `use locale` behaves with locale rules forever, even when used outside that scope. Rebuild under the desired pragma if the behaviour needs to change.

- **\Q . . \E for literal text.** To compile a pattern that matches a user-provided string verbatim, wrap it with `\Q . . \E`: `qr/\Q$literal\E/`. Without `\Q`, metacharacters in the string are interpreted as regex syntax.
- **Empty pattern.** `qr//` compiles to the empty pattern, which is treated specially at match time: `$str =~ $empty` reuses the *last successful* pattern in that scope. This is a match-time behaviour of `m`, not a `qr//`-specific trait, but it is a common trap when passing a default `qr//` around.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `m` - match a compiled pattern or a literal pattern against a string; the primary consumer of a `qr//` object
- `s` - substitute using a compiled pattern; accepts a `qr//` in the search slot
- `split` - split on a compiled pattern; the `qr//` form is the most efficient
- *Regular expressions guide*
 - full regex syntax reference; read this for the meaning of the modifiers and escape sequences
- `ref` - returns "Regexp" for a `qr//` object; the standard type check
- `eval` - wrap `qr//` when the pattern source is untrusted and a compile error should be recoverable

Regular expressions and pattern matching

quotemeta

Return a copy of a string with every regex-significant character backslash-escaped, so the result can be interpolated into a pattern and match its own literal content.

`quotemeta` exists to bridge string data and regex syntax. When user input, filenames, or configuration values end up on the right-hand side of `=~`, any character that happens to be a regex metacharacter (`.`, `*`, `+`, `?`, `(`, `[`, `\`, `|`, ...) would otherwise fire its special behaviour. `quotemeta` preempts that by inserting a backslash before every ASCII non-word character, leaving letters, digits, and underscore untouched.

Synopsis

```
quotemeta EXPR
quotemeta
```

What you get back

A new string. Every ASCII character that does **not** match `/[A-Za-z_0-9]/` is preceded by a backslash; word characters pass through unchanged. The result is always safe to paste into a regex as a literal pattern fragment.

`quotemeta` is a pure value producer - it does not modify its argument.

```
my $safe = quotemeta 'a.b*c';    # 'a\.b\*c'
```

Global state it touches

Reads `$_` when called with no argument. Within the scope of `use locale`, additional non-ASCII Latin-1 characters are quoted to protect against locales that treat punctuation like `|` as a word character.

Examples

Bare form defaults to `$_`, like most list built-ins:

```
for ('a+b', 'c.d') {
    print quotemeta, "\n";    # a\b
                              # c\.d
}
```

Interpolating user input into a substitution. Without `quotemeta`, `.*?` in `$substring` would be regex-active and match far more than intended:

```
my $sentence = 'The quick brown fox jumped over the lazy dog';
my $substring = 'quick.*?fox';
my $quoted   = quotemeta $substring;
$sentence =~ s{$quoted}{big bad wolf};
# $sentence unchanged - the literal text 'quick.*?fox' is not
→present
```

The `\Q... \E` escape in a double-quoted string is exactly `quotemeta` applied to the enclosed region. The two forms below are equivalent:

```
my $pat1 = "\Q$substring\E";
my $pat2 = quotemeta($substring);
```

Building a regex that matches a literal filename anywhere in a path:

```
my $name = 'my.config[prod].ini';
if ($path =~ /\Q$name\E/z/) {
    # matches exactly the filename, no metachar surprises
}
```

Escaping a list of terms for an alternation:

```
my @terms = ('C++', 'C#', '.NET');
my $alt    = join '|', map { quotemeta } @terms;
# 'C\+\+|C\#|\.NET'
```

Edge cases

- **No argument:** `quotemeta` with no expression quotes `$_`. Inside a loop like `for (@input) { push @safe, quotemeta }` this is the idiomatic form.
- **Already-safe strings are untouched:** pure word-character input `([A-Za-z0-9_]+)` round-trips identically.
- **Literal backslashes inside `\Q... \E`** interact with double-quotish backslash interpolation before `\Q` sees the text. A sequence like `"\Q\t\E"` quotes a literal tab character, not a backslash followed by `t`. When you need actual backslashes in the quoted region, keep the data in a variable and use `quotemeta $var` rather than embedding it in a double-quoted literal.
- **No way to inject a literal `$` or `@` inside `\Q... \E`:** a protected `\$` becomes the four-character sequence `\\\$` in the output, and an unprotected `$` starts scalar interpolation before `\Q` runs. Drop out of the quoted region to splice in sigils.
- **Unicode-aware quoting** (Perl 5.16+): on UTF-8 strings - and on byte strings under `use feature 'unicode_strings'` or use v5.12 or greater - non-ASCII characters are quoted only when they carry the Unicode properties `Pattern_Syntax`, `Pattern_White_Space`, `White_Space`, `Default_Ignorable_Code_Point`, or `General_Category=Control`. Identifier-class characters (letters, marks, digits) are left alone. This is the stable contract: Perl promises that any future regex metacharacter will have `Pattern_Syntax` set, so strings safe today stay safe.
- **Legacy non-UTF-8 strings** outside `unicode_strings` scope have every upper-Latin-1 code point (`\x80-\xFF`) quoted, for backwards compatibility with pre-5.16 behaviour.
- **Locale quoting:** within `use locale`, all non-ASCII Latin-1 code points are quoted whether the string is UTF-8 or not. ASCII-range quoting is unaffected by locale.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `m` - the match operator; `quotemeta` exists to feed safe literals into its pattern
- `qr` - compiled regex objects; `qr/\Q$var\E/` is the usual way to pin a pattern around a literal string
- `s` - substitution; the most common site of `quotemeta`-style escaping on the left-hand pattern
- `\Q` and `\E` - the double-quotish escape pair that is just `quotemeta` with different syntax
- `$_` - default input when `quotemeta` is called without an argument

Lists

qw//

Build a list of barewords by splitting a delimited string on whitespace, without quoting or comma-separating each word.

qw// is a compile-time quote-like operator, not a runtime function. The parser sees `qw(foo bar baz)` and emits the constant list `("foo", "bar", "baz")` - no splitting happens at run time, and the list is assembled once regardless of how often the code is executed. Reach for it wherever you would otherwise type a list of short, simple string literals separated by commas: `import lists`, `@ISA`, `@EXPORT`, hash-key sets, test-name tables.

Synopsis

```
qw(foo bar baz)
qw[foo bar baz]
qw{foo bar baz}
qw<foo bar baz>
qw!foo bar baz!
qw/foo bar baz/
```

Any non-alphanumeric character works as a delimiter. The four bracket pairs `()`, `[]`, `{}`, `<>` nest; all other delimiters do not.

What you get back

A list of strings in list context - one element per run of non-whitespace characters in the body, with leading and trailing whitespace discarded and runs of internal whitespace treated as a single separator.

In scalar context `qw//` evaluates to the **last** element of the list, which is almost never what the author intended. `use warnings` flags the common scalar-context mistakes (see *Edge cases*).

Examples

Import list for a module - the canonical use:

```
use POSIX qw(setlocale localeconv strftime);
```

`@EXPORT` / `@EXPORT_OK` / `@ISA` tables:

```
our @ISA      = qw(Exporter);
our @EXPORT   = qw(sum min max);
```

Build a small hash where every key maps to 1 (a set):

```
my %is_keyword = map { $_ => 1 } qw(if unless while until for);
if ($is_keyword{$word}) { ... }
```

Compare the three ways of writing the same list - commas-and-quotes, fat-comma auto-quoting, and `qw//`:

```
my @a = ("foo", "bar", "baz");           # explicit strings
my @b = (foo => "1", bar => "2", baz => "3"); # fat comma only
my @c = qw(foo bar baz);                 # qw//
```

Array initialisation with many short tokens:

```
my @months = qw(
    Jan Feb Mar Apr May Jun
    Jul Aug Sep Oct Nov Dec
);
```

Positional argument list to a function:

```
run_tests(qw(smoke integration regression));
```

Edge cases

- **No commas inside the body.** A comma is taken as part of a word: `qw(foo, bar)` yields the two-element list `("foo,", "bar")`, not three elements. `use warnings` emits `Possible attempt to separate words with commas`.
- **No comment syntax.** `#` is a literal character inside the body. `qw(foo # comment\n bar)` contains the word `#` and the word `comment`. `use warnings` emits `Possible attempt to put comments in qw() list`.
- **No backslash escapes.** `\` is literal - you cannot escape a whitespace character to include it in a word, and you cannot `\`-escape the delimiter. `use warnings` emits a warning for any `\` in the body.
- **No interpolation.** `qw($x @y)` produces the two literal strings `"$x"` and `"@y"`. Use `("$x", @y)` (or `q()/qq()`) when you want a variable's value.
- **Whitespace is the only separator.** Tabs and newlines split words exactly like spaces, which is why multi-line `qw( . . . )` lists are idiomatic for long tables.
- **Words with embedded whitespace are impossible.** If you need `"hello world"` as a single element, `qw//` is the wrong tool - use `("hello world", . . .)` with real quotes.
- **Empty body.** `qw()` is the empty list, equivalent to `()`.
- **Scalar context** yields the last element, not the count. Wrap in `scalar(() = qw(. . .))` if you genuinely want the count, or assign to an array first:

```
my @w = qw(foo bar baz);
print scalar @w;           # 3
```

- **Parser ambiguity with `//`.** `qw//` uses `/` as a delimiter just like `m//` and `s///`. Inside the body, no regex metacharacters apply - it is a flat string split on whitespace. The visual similarity to a regex is a reading hazard, not a semantic one.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `q//` - single-quote literal string; no interpolation, unlike `qq//`
- `qq//` - double-quote literal string with full interpolation
- `qr//` - compile a regex literal; the `//` family member for pattern matching
- `split` - the runtime equivalent when your word list is not known until run time: `split(" ", $str)` behaves almost exactly like a dynamic `qw//`
- `join` - the inverse operation, gluing a list back into a single delimited string

Processes

`qx//`

Run a shell command and capture its standard output.

`qx//` (and its twin, the backtick form ``STRING``) interpolates `STRING` like a double-quoted string, hands the result to the system shell, waits for the child to finish, and returns whatever the child wrote to `STDOUT`. Standard error is untouched - it still goes to the parent's `STDERR` unless you redirect it inside the command. The child's exit status lands in `$?`.

Synopsis

```
my $out = `date`;
my $out = qx/date/;
my @lines = `ls -l /etc`;
my $raw = qx'echo $HOME';      # single-quote form: no Perl
↪ interpolation
```

What you get back

Context-sensitive:

- **Scalar context** - one string containing the entire captured `STDOUT`, including any embedded newlines. `undef` if the shell or command could not be started.
- **List context** - a list of lines, split on the current value of `$/` (the input record separator, newline by default). Empty list if the shell or command could not be started.

The exit status of the child is always placed in `$?`, whether or not the captured output is used. A child that writes nothing and exits cleanly yields `""` in scalar context, `()` in list context, and `0` in `$?` - distinguishing “started but produced no output” from “failed to start” requires checking `$?` against `-1` or `$!`.

Global state it touches

- `$?` - set to the child's wait status (waitpid-style) on every call. -1 means the child could not be started, in which case `$!` holds the reason.
- `$!` - set when the shell or command failed to start.
- `$/` - governs how list-context output is split into lines.
- `$|` - affects whether buffered Perl output reaches the child's environment before the child runs; set it on output handles before `qx//` if ordering with prior print matters.
- `%ENV` - exported to the child as its environment.
- The current working directory, umask, and open file descriptors are inherited by the child.

Examples

Capture a single line:

```
my $today = `date`;
chomp $today;
print "today is $today\n";
```

Capture line-by-line in list context:

```
my @users = `who`;
for my $line (@users) {
    my ($name) = split /\s+/, $line;
    print "logged in: $name\n";
}
```

Interpolation happens before the shell sees the string - Perl variables expand first, shell variables expand in the child:

```
my $dir = "/etc";
my $out = `ls $dir`;           # Perl interpolates $dir
my $sh = qx'echo $HOME';      # single-quote form: $HOME is shell
↪ 'S
```

Redirect STDERR into the captured stream:

```
my $both = `gcc -v hello.c 2>&1`;
```

Discard STDERR, keep only STDOUT:

```
my $out = `find / -name core 2>/dev/null`;
```

Check the exit status:

```
my $out = `make test`;
if ($? == -1) {
```

(continues on next page)

(continued from previous page)

```
    die "failed to execute make: $!";
}
elsif ($? & 127) {
    die sprintf "make died with signal %d", $? & 127;
}
elsif ($? >> 8) {
    die sprintf "make exited with status %d", $? >> 8;
}
```

Edge cases

- **No shell metacharacters:** if `STRING` contains no shell metacharacters, Perl bypasses the shell and execs the command directly - same optimisation as `system`. This is faster and avoids one `sh -c` level of quoting.
- **Single-quote delimiter:** `qx'...'` suppresses Perl's double-quote interpolation. The literal `$var` is passed to the shell, which does its own expansion. Useful when the variable names belong to the shell, not to your Perl program.
- **Alternate delimiters:** `qx//` accepts any balanced or non-alphanumeric delimiter - `qx{ls -l}`, `qx(ls -l)`, `qx[ls -l]`, `qx!ls -l!`. Pick one that doesn't collide with shell syntax in your command. `qXfooX` is a syntax error - the delimiter must not be alphanumeric.
- **Backticks don't capture STDERR:** nothing written to the child's STDERR ends up in the returned string. Redirect with `2>&1` inside the command if you want it, or run the command through `open` with a pipe for finer control.
- **STDIN is inherited:** the child reads from whatever the Perl process had on STDIN at the time of the call. To feed the child specific input, use `open` in pipe mode instead.
- **Exit status vs \$!:** a non-zero `$?` means the child ran and exited with an error; `$!` is only meaningful when `$?` is -1 (the shell itself could not be launched).
- **Memory cost in scalar context:** the whole captured output is buffered in memory before being returned. For commands that produce very large or unbounded output, open a pipe with `open` and read it incrementally.
- **Function form:** `qx//` can also be spelled `readpipe` when you want a function-call syntax - useful when the command string is built dynamically and you'd rather pass it as a scalar than splice it into a quote-like.
- **Output buffering:** Perl flushes its buffered output handles before forking the child, but portability varies. When preceding `qx//` with `print` to a pipe or file, set `$|` on that handle to be sure your output reaches the destination before the child starts.
- **Shell selection:** the shell used is `/bin/sh` on Unix. Commands that rely on bash-isms need an explicit `bash -c '...'` wrapper.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `system` - run a shell command without capturing output; reach for this when you only care about the exit status
- `exec` - replace the current process with a command; no capture, no return
- `readpipe` - the named-function form of `qx//`; takes the command as a regular string argument
- `open` - pipe-mode (`open my $fh, "-|", CMD`) for streaming a child's output line by line instead of buffering the whole capture
- `$?` - the child wait status set by every `qx//` call; decode with `>> 8` for exit code and `& 127` for terminating signal
- `$/` - controls how list-context `qx//` splits output into lines

Numeric functions

rand

Return a pseudo-random floating-point number in the half-open range $[0, \text{EXPR})$.

`rand` draws the next value from the interpreter's pseudo-random number stream and scales it to $0 \leq n < \text{EXPR}$. The result is a regular Perl number (a double), never an integer on its own - wrap with `int` when you want a whole number. `EXPR` is optional and defaults to 1, so a bare `rand` yields a float in $[0, 1)$.

Synopsis

```
rand EXPR
rand
```

What you get back

A double-precision float `n` with $0 \leq n < \text{EXPR}$. The upper bound is **exclusive**: `rand(6)` never returns 6, only values strictly below it. For an integer result, compose with `int`:

```
my $die = int(rand(6)) + 1;      # 1..6
```

The distribution is (approximately) uniform across the range. `rand` is a scalar operator - it returns one value per call, regardless of context.

Global state it touches

`rand` consumes the interpreter's internal PRNG state, shared with `srand`. Two rules follow from that:

- **Autoseeding:** the first call to `rand` in a program automatically calls `srand` if you haven't already. The seed is chosen from process-level entropy, so unseeded programs produce a different stream on every run.
- **Shared stream:** every subsequent `rand` call - anywhere in the program, in any module - draws from the same sequence. Calling `srand($seed)` with a fixed seed makes that sequence deterministic and reproducible, which is what you want for test fixtures and debugging. Calling `srand()` with no argument re-seeds from entropy.

There is no per-thread or per-scope isolation. If you need an independent stream for one piece of code, do not use `rand`; reach for a dedicated generator module.

Examples

Basic draw in `[0, 1)`:

```
my $u = rand(); # e.g. 0.4173...
```

Dice roll - canonical `int(rand(N)) + 1` idiom:

```
my $roll = int(rand(6)) + 1; # 1..6
```

Random element of an array:

```
my @colors = qw(red green blue yellow);
my $pick = $colors[ rand @colors ];
```

`rand @colors` uses the array's length (scalar context forced by `rand`'s prototype), so this works for any array size.

Reproducible stream - fix the seed for tests:

```
srand(42);
my @sample = map { rand() } 1..5; # same five numbers every run
```

One-line shuffle idiom. Readable, but **biased** - `sort` compares pairs more than once, and a one-shot `rand()` per comparison does not produce a uniform permutation:

```
my @shuffled = sort { rand() <=> 0.5 } @list; # biased; do not use
```

For a fair shuffle, use `List::Util::shuffle`:

```
use List::Util qw(shuffle);
my @shuffled = shuffle @list;
```

Simulate a weighted coin - note the exclusive upper bound means `rand() < 0.3` fires with probability exactly `0.3`:

```
my $heads = rand() < 0.3 ? "heads" : "tails";
```

Edge cases

- **EXPR is 0:** treated as 1, so `rand(0)` behaves like `rand(1)`. This is documented upstream behaviour and is preserved here, but it is a historical quirk - do not rely on it for new code.
- **EXPR is negative:** the result is undefined in practice - you may get 0, you may get a negative float, and the behaviour is not portable across Perl versions. If you want a value in `[-N, 0)`, write `rand($n) - $n` explicitly.
- **EXPR is undef:** stringifies to "", numifies to 0, and therefore behaves like `rand(0)` → `rand(1)`. Under use warnings you get an uninitialized warning.
- **EXPR is non-numeric:** the usual scalar-to-number coercion applies. `rand("3abc")` is `rand(3)` with a non-numeric warning.
- **Not cryptographically secure:** `rand` uses a fast PRNG suitable for simulation, shuffling, and test fixtures. It is **not** suitable for session tokens, password salts, key material, or any security-sensitive value. For those, read from `/dev/urandom` directly or use `Crypt::URandom` / `Crypt::PRNG`.
- **Thread/fork behaviour:** after `fork`, both parent and child inherit the same PRNG state and will produce the **same** stream unless one of them calls `srand()` to re-seed. A common footgun in pre-forking servers - seed in the child.
- **Large EXPR:** the generator produces 48 bits of randomness per draw; for EXPR close to or above 2^{48} , the output is no longer finely distributed over the range. For cryptographic widths you shouldn't be using `rand` anyway (see above).

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `srand` - seed the PRNG explicitly for reproducibility, or re-seed in a forked child
- `int` - truncate a `rand` result toward zero to get a uniform integer in `[0, N)`
- `List::Util::shuffle` - fair Fisher-Yates shuffle; prefer this over the `sort { rand() <=> 0.5 }` idiom
- `Crypt::URandom` - cryptographically-secure random bytes from the OS entropy source; use this for tokens, salts, and key material
- `Crypt::PRNG` - cryptographically-secure PRNG with a rich API when you need more than raw bytes

I/O · Fixed-length data

read

Read a fixed amount of buffered input from a filehandle into a scalar.

`read` pulls up to `LENGTH` characters from `FILEHANDLE` and stores them in `SCALAR`, returning how many were actually read. It goes through the handle's PerlIO stack and is therefore buffered on top of the underlying OS read - contrast with `sysread`, which bypasses the buffer and calls `read(2)` directly. The optional `OFFSET` argument lets you splice the incoming data into the middle of `SCALAR` rather than overwriting it.

Synopsis

```
read FILEHANDLE, SCALAR, LENGTH
read FILEHANDLE, SCALAR, LENGTH, OFFSET
```

What you get back

- The **number of characters read**, which may be less than `LENGTH`.
- `0` at end of file.
- `undef` on error, with `$!` set.

`SCALAR` is grown or shrunk so that the last character actually read becomes the last character of the scalar - unless `OFFSET` is given, in which case only the slice at `OFFSET` is overwritten and anything beyond it is left alone (see *The OFFSET argument* below).

A short read is **not** an error. On a regular file it normally means you reached end of file; on a pipe, socket, or terminal it means no more data is available right now. Loop until you either have the bytes you need or `read` returns `0` / `undef`:

```
my $buf = "";
my $want = 4096;
while ($want > 0) {
    my $got = read($fh, $buf, $want, length $buf);
    die "read error: $!" unless defined $got;
    last if $got == 0;           # EOF
    $want -= $got;
}
```

Global state it touches

- `$!` - set when `read` returns `undef`.
- `${^UTF8CACHE}` / the handle's PerlIO layers - determine whether `LENGTH` is counted in bytes or in characters (see *Character vs byte semantics* below).

`read` does not interact with `$_`, `$/`, `$\`, or `$,`. Unlike `readline`, it does not care about the input record separator.

The OFFSET argument

OFFSET controls **where in SCALAR** the incoming data lands. It does not seek the filehandle.

- **Omitted** - data replaces the entire contents of SCALAR.
- **Positive, within length** - data is written starting at position OFFSET. Characters before OFFSET are preserved; characters from OFFSET to the end of SCALAR are overwritten or extended.
- **Positive, beyond length** - SCALAR is first padded with "\0" bytes out to OFFSET, then the read is appended. Useful for reading into a fixed slot inside a larger buffer you are assembling.
- **Negative** - counts backwards from the end of SCALAR. -1 means “overwrite the last character and append from there.”

```
my $buf = "HEADER";
read($fh, $buf, 16, length $buf);      # append 16 chars after
    ↳ "HEADER"

my $slab = "";
read($fh, $slab, 512, 1024);           # pad to 1024 "\0" bytes,
    ↳ then                               # read 512 chars - $slab is
                                         # now 1536 chars long
```

Character vs byte semantics

LENGTH is measured in whatever unit the handle deals in:

- **Byte-mode handle** (the default, and every handle opened without an encoding layer): LENGTH is a byte count. `read($fh, $buf, 10)` pulls 10 bytes and `length $buf` is 10.
- **:utf8 layer**: LENGTH is a **character** count. Perl decodes UTF-8 on the way in, and `$buf` holds decoded codepoints. The number of bytes consumed from the file can be anywhere from LENGTH to `4 * LENGTH`, depending on the text.
- **:encoding(...)** layer: same rule as `:utf8`, for any encoding the layer knows.

```
open my $fh, "<:utf8", "greek.txt" or die $!;
read($fh, my $buf, 5);                 # 5 characters, not 5 bytes
```

Mixing a byte-mode read with UTF-8 data produces mojibake and, under `use warnings`, a `Malformed UTF-8` warning if you later decode the result. Pick the layer at `open` time and stick with it.

Buffered vs unbuffered I/O

`read` is **stdio-buffered** through `PerlIO` - internally it calls `fread(3)` (or `PerlIO`’s replacement) against the handle’s buffer. That has two consequences worth remembering:

- You can mix `read`, `readline` / `<$fh`, `getc`, and `seek` freely on the same handle. They all see the same buffer.
- You must **not** mix `read` with `sysread` on the same handle. `sysread` bypasses the buffer and goes straight to `read(2)`; any bytes already pulled into the buffer by a previous `read` become invisible to `sysread`, and vice versa. If you need raw syscall semantics, use `sysread` exclusively on that handle.

For byte-accurate, non-buffered input - for example on a non-blocking socket, or when implementing a protocol where a short read is meaningful rather than “try again” - reach for `sysread`.

Examples

Read a fixed-size header from a binary file:

```
open my $fh, "<", "packet.bin" or die "open: $!";
binmode $fh;
my $header;
my $n = read($fh, $header, 16);
die "short header: got $n bytes" unless $n == 16;
```

Append 16 bytes to the end of an existing buffer by using OFFSET equal to the current length:

```
my $buf = "PRELUDE:";
read($fh, $buf, 16, length $buf);      # $buf is now "PRELUDE:" . 16_
↳new bytes
```

Read into position 1024 of a scalar, padding the gap with `"\0"`:

```
my $slot = "";
read($fh, $slot, 64, 1024);             # length($slot) == 1088
                                         # substr($slot, 0, 1024) is
↳"\0" x 1024
```

Loop until you have exactly N bytes or hit EOF - the correct pattern for pipes and sockets where a single read often returns fewer bytes than requested:

```
sub read_exact {
    my ($fh, $n) = @_;
    my $buf = "";
    while (length($buf) < $n) {
        my $got = read($fh, $buf, $n - length($buf), length $buf);
        return undef unless defined $got;
        return $buf if $got == 0;      # EOF; caller inspects_
↳length
        # loop
    }
    return $buf;
}
```

Character-counted read through a UTF-8 layer:

```
open my $fh, "<:encoding(UTF-8)", "notes.txt" or die $!;
read($fh, my $chunk, 100);           # 100 characters
printf "chars=%d bytes=%d\n", length $chunk, do {
    use bytes; length $chunk;
};
```

Edge cases

- **Closed filehandle:** returns `undef` and sets `$!` to "Bad file descriptor". Under use warnings a `read()` on closed filehandle warning is emitted.
- **Unopened filehandle:** same as closed - `undef` and `$!` set.
- **LENGTH of 0:** `read` returns `0` immediately and does not touch SCALAR. It is **not** a reliable EOF probe; use `eof` for that.
- **Negative LENGTH:** a fatal runtime error (Negative length at ...). Validate LENGTH before calling.
- **Negative OFFSET whose magnitude exceeds the current length of SCALAR:** a fatal runtime error (Offset outside string). Clamp with `max($offset, -length $buf)` when the offset is computed.
- **Short read on a pipe or socket:** not an error. `read` returns fewer characters than requested whenever the PerlIO buffer empties before LENGTH is reached. Loop if you need the full count.
- **EOF mid-read:** returns the partial count. The next call returns `0`. After that, `$fh` stays at EOF until you `seek` or `clearerr`.
- **Reading from a tied handle:** `read` dispatches to the tie class's `READ` method, which is responsible for honouring LENGTH and OFFSET. Misbehaving tie classes can violate the "grow SCALAR so the last character read is the last character" contract.
- **Interaction with `sysread`:** do not mix them on one handle. `read` fills the PerlIO buffer in chunks of its own choosing; `sysread` ignores that buffer entirely.
- **Binary data on a text-mode handle:** on Unix-like systems there is no distinct text mode, but an encoding layer still transforms bytes. `binmode $fh` (or `open ..., "<:raw", ...`) before reading binary data.
- **FILEHANDLE as an expression:** a bareword or simple scalar is fine. Anything more complex must be parenthesised: `read(($handles[$i]), $buf, $len)`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `open` - acquires the filehandle and decides whether subsequent reads are byte-counted or character-counted via the I/O layer
- `sysread` - the unbuffered counterpart, a direct `read(2)` system call; use it for non-blocking I/O or when a short read is meaningful

- `readline` / `<$fh>` - record-oriented input that respects `$/` instead of a byte/character count
- `getc` - read a single character; roughly `read($fh, $c, 1)` but with different EOF/undef reporting
- `binmode` - remove or add I/O layers so that `LENGTH` is unambiguously a byte count or a character count
- `eof` - the right way to test for end of file, rather than reading a zero-length chunk

I/O

readdir

Read the next entry, or all remaining entries, from a directory handle opened by `opendir`.

`readdir` is the directory counterpart of `readline`: it pulls names out of a directory handle one at a time in scalar context, or all at once in list context. The handle must already have been opened with `opendir`; iteration advances a position that `rewinddir`, `seekdir`, and `telldir` can manipulate.

Synopsis

```
readdir DIRHANDLE
my $name = readdir $dh;
my @names = readdir $dh;
while (readdir $dh) { ... }      # sets $_
```

What you get back

Context-dependent:

- **Scalar context:** the next entry's name as a string, or `undef` when the directory is exhausted.
- **List context:** every remaining entry as a list of strings, or the empty list when the directory is already exhausted.

Entries are **bare filenames**, not paths. `readdir` returns `"foo.txt"`, never `"/some/dir/foo.txt"`. If you plan to hand the result to a file test, `open`, `stat`, or any other call that resolves paths against the current working directory, prepend the directory yourself - `readdir` does **not** `chdir` into the directory it is reading.

The list always includes the entries `.` (current directory) and `..` (parent directory) as reported by the underlying filesystem. Filter them out explicitly if you do not want them:

```
my @real = grep { $_ ne '.' && $_ ne '..' } readdir $dh;
```

Order is filesystem-dependent, not alphabetical. ext4, XFS, tmpfs, NFS, and FAT each return entries in their own internal order, which may be insertion order, hash order, or something else entirely. Never assume sorted output. If you need a predictable order, sort the result:


```
my @names = sort readdir $dh;
```

Examples

Read every name into a list, drop the dot entries, and sort:

```
opendir(my $dh, $some_dir) or die "opendir $some_dir: $!";
my @names = sort grep { !/^\.\.?$/ } readdir $dh;
closedir $dh;
```

Iterate one entry at a time. A bare `readdir` in a `while` condition assigns to `$_` and tests the result for **definedness**, not truth - so an entry literally named `"0"` still keeps the loop running:

```
opendir(my $dh, $some_dir) or die "opendir $some_dir: $!";
while (readdir $dh) {
    print "$some_dir/$_\n";
}
closedir $dh;
```

Filetest with the directory prepended - forgetting the prefix is the classic `readdir` bug:

```
opendir(my $dh, $some_dir) or die "opendir $some_dir: $!";
my @dotfiles = grep { /^\.\/ && -f "$some_dir/$_" } readdir $dh;
closedir $dh;
```

Scalar context - one name per call, `undef` signals end of directory:

```
opendir(my $dh, $some_dir) or die "opendir $some_dir: $!";
while (defined(my $name = readdir $dh)) {
    next if $name eq '.' || $name eq '..';
    process("$some_dir/$name");
}
closedir $dh;
```

Rewind and read a second time - the handle stays open and usable:

```
my @first = readdir $dh;
rewinddir $dh;
my @second = readdir $dh; # same entries, possibly different order
```

Edge cases

- **Exhausted handle:** further calls return `undef` in scalar context, empty list in list context. Use `rewinddir` to start over.
- **Closed or invalid handle:** returns `undef` (scalar) or empty list (list) and sets `$!`. Under use warnings Perl emits `readdir() attempted on invalid dirhandle`.

- **Definedness test in while:** both `while (readdir $dh)` and `while (my $n = readdir $dh)` (and their `for` equivalents) are special-cased to test defined, not truth. An entry named `"0"` is returned and the loop continues.
- **`.` and `..` are always present** on POSIX filesystems, even in an otherwise empty directory. Filter them if you want only real content.
- **No path prefix:** entries are bare names. Any filetest or `open` on the raw return value tests relative to the current working directory, not `$some_dir`.
- **No sorting:** the order is whatever the kernel returns. Apply `sort` explicitly if stability matters.
- **Concurrent directory changes:** entries added or removed during iteration may or may not appear, per the platform. The only safe assumption is that entries present at `opendir` time and not removed will be reported.
- **Encoding:** `readdir` returns byte strings as the kernel stores them. On a filesystem using UTF-8 filenames, apply `Encode::decode_utf8` or configure the open pragma if you need character strings.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `opendir` - opens the directory handle that `readdir` reads from; always the first call in the sequence
- `closedir` - releases the handle once iteration is done
- `rewinddir` - resets the iteration position to the start of the directory
- `seekdir` - jumps to a previously recorded position
- `telldir` - records the current position for later `seekdir`
- `glob` - shell-style pattern matching over directory contents; higher-level when you want `*.txt` rather than every entry

I/O

readline

Read one or more records from a filehandle.

`readline` is the input primitive behind the `<FH>` diamond operator. It reads from the filehandle named by `EXPR` (a `typeglob`, a `glob` reference, or an `IO::Handle`), using the current value of `$/` to decide where one record ends. If `EXPR` is omitted, `*ARGV` is read. In scalar context it returns the next record; in list context it returns every remaining record. End-of-file is signalled by `undef` (scalar) or an empty list.

Synopsis

```

my $line = readline $fh;      # one record, scalar context
my @all  = readline $fh;      # every remaining record
my $line = <$fh>;             # same as scalar readline $fh
my @all  = <$fh>;             # same as list  readline $fh
readline;                     # reads from *ARGV

```

What you get back

- **Scalar context:** the next record as a string, or `undef` at end-of-file or on error. The record includes its terminator (the current `$/`) unless the last record in the file is missing one. After the returned value, `$.` is incremented - see the section on global state below.
- **List context:** a list of every remaining record until end-of-file. An empty list means the handle was already at EOF.
- **Slurp mode on an empty file:** when `$/` is `undef` and context is scalar, the first call on a zero-byte handle returns `' '` (empty string, defined); the next call returns `undef`. This lets `defined($slurp = readline $fh)` distinguish an empty file from a missing one without a separate `-s` test.

Global state it touches

- `$/` - input record separator. Controls what counts as “one record”. Default is `"\\n"`. Special values are covered under *Record modes* below. Also known as `$INPUT_RECORD_SEPARATOR` or `$RS` under `use English`.
- `$.` - current input line number of the last handle read. `readline` increments `$.` once per record returned. `$.` is reset when the handle is explicitly `closed`, but **not** when `*ARGV` rolls from one file to the next - that continues counting across the whole input. Also `$INPUT_LINE_NUMBER` or `$NR` under `use English`.
- `$_` - the default scalar. When a `readline` call (or its `<FH>` equivalent) is used **as the condition of a while or for loop**, Perl implicitly assigns the result to `$_` and tests `defined`, not truth. `while (<$fh>) { ... }` is therefore safe on lines containing `"0"` or `" "`.
- `$_` - `errno`. On a read error, `readline` returns `undef` (scalar) or a truncated list, and sets `$_` to the operating-system error.
- `@ARGV` - when `EXPR` is omitted (or the handle is `ARGV`), each end-of-file causes Perl to `open` the next filename from `@ARGV` and continue reading. An empty `@ARGV` makes `ARGV` read `STDIN`.

Record modes

`$/` selects one of four reading modes:

- **Line mode** (default) - `$/` is a non-empty string. A record ends at the next occurrence of that string. With the default `"\\n"`, one call returns one line including its newline.

- **Paragraph mode** - `$/ = ""`. A record ends at the next run of two or more consecutive newlines. Leading blank lines before the first paragraph are skipped; the terminating blank line(s) are included in the returned string. Use `local $/ = ""` to confine the change to a block.
- **Slurp mode** - `$/ = undef`. The first call returns the entire remaining contents of the handle as one string; the next call returns `undef`. On an empty file, see the special case under *What you get back*.
- **Fixed-record mode** - `$/ = \N` (a reference to a positive integer) or `$/ = \N`. Each call reads exactly N bytes, or fewer at end-of-file. Useful for binary framing. The result is still counted as one record for `$..`

Changes to `$/` take effect on the next `readline` call. Scope them with `local` so a subroutine does not silently change the caller's idea of "line".

Examples

Idiomatic line-by-line read. The while condition implicitly assigns to `$_` and tests `defined`:

```
open my $fh, "<", "input.txt" or die "open: $!";
while (<$fh>) {
    chomp;
    print "line $.: $_\n";
}
close $fh;
```

Explicit `readline` with error detection. `eof` must be checked separately from the return value, because a legitimate line can be the empty string:

```
while ( ! eof($fh) ) {
    defined( my $line = readline $fh )
        or die "readline failed: $!";
    # ... process $line ...
}
```

Slurp a whole file into a scalar:

```
my $contents = do {
    local $/; # undef - slurp mode
    open my $fh, "<", $path or die "open $path: $!";
    readline $fh;
};
```

Paragraph mode - read an RFC-822-style message header block:

```
open my $fh, "<", "message.eml" or die $!;
local $/ = "";
my $headers = <$fh>; # up to the first blank line
my $body = do { local $/; <$fh> };
```

Fixed-record mode - read 512-byte blocks from a device file:

```
open my $fh, "<:raw", "/dev/sda" or die $!;
local $/ = \512;
while (defined(my $block = <$fh>)) {
    last if length($block) < 512;    # short read at EOF
    # ... process $block ...
}
```

List context - pull every remaining line at once. Cheap for small files, ruinous for large ones:

```
open my $fh, "<", "hosts" or die $!;
my @lines = <$fh>;           # every line, including terminators
chomp @lines;
```

Reading from *ARGV with no explicit handle - the classic Perl filter idiom. @ARGV is consumed as a list of filenames, or STDIN is read if @ARGV is empty:

```
while (defined(my $line = readline)) {
    print $line;             # cat(1) in one line
}
```

Edge cases

- **EOF vs. error:** scalar readline returns `undef` for both. Call `eof($fh)` or check `$!` to distinguish them. Inside `while (<$fh>)` the loop exits on either; add an explicit `defined` plus `$!` check when you need to tell them apart.
- **A file with no trailing newline:** the last record is returned without a terminator in line mode. `chomp` is a no-op on such a value, which is the correct behaviour.
- **Paragraph mode on a file of only blank lines:** returns `undef` immediately - there is no non-empty paragraph to yield.
- **Slurp mode, empty file:** first call returns `' '` (defined, zero-length); the second returns `undef`. Do not branch on truthiness of the first read.
- **Fixed-record mode with a short read at EOF:** returns the remaining bytes (possibly fewer than N), then `undef` on the next call. Check `length` if you need exactly N.
- **The ARGV magic handle:** when reading from *ARGV, `eof()` with no argument returns true only at the very end of **all** @ARGV files, while `eof(ARGV)` is true at the end of **each** file. You cannot use the `defined(readline) or die "...: $!"` idiom against *ARGV to distinguish EOF from error - open each filename yourself if you need that.
- **Already-closed handle:** readline returns `undef` and sets `$!` to "Bad file descriptor". Underuse warnings a `readline()` on closed filehandle warning is emitted.
- **Encoding layers:** readline returns whatever the handle's PerlIO stack produces. A `:utf8` or `:encoding(...)` layer decodes bytes into characters; a `:raw` handle returns bytes. Mixing a multi-byte `$/` with a byte handle can split characters.

- **Mid-record `$/` change:** only the next `readline` call observes the new separator. Any already-buffered data is re-scanned against it.
- **`<FH>` vs. `<$fh>` vs. `<>`:** these are all `readline` under the hood. `<FH>` reads from the typeglob `FH`; `<$fh>` reads from the handle in `$fh`; `<>` (or `<<>>` for safer filename handling) reads from `*ARGV`. The `<pattern>` form with any other content is `glob`, not `readline` - a different function entirely.
- **Loop condition assignment:** `while (my $line = <$fh>) { ... }` tests `defined`, not `truth`, only because the parser rewrites it. `while (($line = <$fh>)) { ... }` with the extra parentheses does **not** get the rewrite, and will loop forever on input containing `"0\n"`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `open` - obtain the filehandle that `readline` consumes
- `read` - fixed-length byte read that ignores `$/`; use when you want a byte count, not a record
- `getc` - single-character read, likewise ignoring `$/`
- `eof` - test whether the next `readline` would return `undef`; behaves specially for `*ARGV`
- `$/` - input record separator; controls line, paragraph, slurp, and fixed-record modes
- `$.` - current input line number, incremented by every successful `readline`

Filehandles, files, directories

readlink

Return the target path a symbolic link points to.

`readlink` asks the kernel what string is stored inside a symbolic link, without following it and without touching the file the link eventually resolves to. The argument is the path of the link itself; the return value is the raw target string as recorded when the link was created. If `EXPR` is omitted, `$_` is used.

Synopsis

```
readlink EXPR
readlink
```

What you get back

On success, the link target as a string - exactly the bytes stored in the symlink, without any interpretation. On failure, `undef`, with `#!` set to the system error. Typical error codes:

- EINVAL - the path exists but is not a symbolic link
- ENOENT - the path does not exist
- EACCES - a directory component is not searchable
- ELOOP, ENAMETOOLONG, ENOTDIR - the usual path-resolution failures on components leading up to the link

The returned string is **not** canonicalised and **not** made absolute. A link created with `ln -s ../lib/libfoo.so libfoo.so` returns the literal `../lib/libfoo.so`, relative to the directory that contains the link itself - not to the current working directory. Resolving the target to an absolute, real path is the caller's job; see `Cwd::abs_path`.

Global state it touches

- `$_` - read as the default argument when `EXPR` is omitted
- `$_!` - set on failure to the C `errno` from the underlying `readlink(2)` call

Examples

Read a link target directly:

```
my $target = readlink "/usr/bin/vi";
print defined $target ? "points to $target\n" : "not a symlink: $_!\n";
```

Default argument form inside a loop over filenames:

```
for (@paths) {
    next unless -l;                # skip anything that isn't a
    ↪symlink
    my $t = readlink;              # reads $_
    print "$_ -> $t\n" if defined $t;
}
```

Guard against non-symlinks explicitly. `readlink` on a regular file returns `undef` and sets `$_!` to `EINVAL`:

```
my $t = readlink $path;
if (!defined $t) {
    die "readlink $path: $_!" unless $_{EINVAL};
    # not a symlink - handle as a plain file
}
```

Resolve a relative link target against the link's own directory, which is the correct base - not the current working directory:

```
use File::Basename qw(dirname);
use File::Spec;

my $target = readlink $link // die "readlink $link: $_!";
my $resolved = File::Spec->rel2abs($target, dirname($link));
```

Canonicalise fully (follow every link in the chain, collapse . and . . , produce an absolute path) with `Cwd::abs_path`:

```
use Cwd qw(abs_path);
my $real = abs_path($link);           # undef if any component is_
↪missing
```

Edge cases

- **Not a symlink:** `readlink` on a regular file, directory, or anything else that isn't a symbolic link returns `undef` and sets `$!` to `EINVAL`. Test with `-l` first if you want to distinguish “not a link” from other errors.
- **Missing path:** a component of `EXPR` that doesn't exist gives `ENOENT`, same as any other path-lookup failure.
- **Default argument:** bare `readlink` uses `$_`. No warning is emitted when `$_` is `undef`; the call simply fails with a path-lookup error.
- **Trailing components:** `readlink` operates on the final component only - intermediate symlinks in the path are followed normally. `readlink "/a/b/c"` where `/a` is itself a symlink reads the link at `c`, not at `a`.
- **Relative target, unchanged:** the returned string is stored verbatim. Never assume the string is absolute, and never join it onto the current working directory - join onto the directory that contains the link, or feed the link path (not the target) to `Cwd::abs_path`.
- **Dangling links:** a link whose target does not exist still reads fine. `readlink` does not stat the target. Use `lstat` to inspect the link itself, `stat` to inspect what it points to (and fail if the target is missing).
- **Platforms without symlinks:** on systems that do not implement symbolic links, `readlink` raises an exception rather than returning `undef`. Linux (the only supported pperl target) always has them, so this is relevant only for portable code shared with other targets.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `symlink` - create a symbolic link; inverse operation
- `link` - create a hard link (not readable with `readlink`; hard links are indistinguishable from the original name)
- `lstat` - stat the symlink itself rather than its target; use together with `readlink` to inspect a link fully
- `-l` - file test for “is a symbolic link”; check before calling `readlink` to disambiguate `EINVAL` from other errors
- `$!` - `errno` set on failure; `#{EINVAL}` distinguishes “not a symlink” from other errors

Processes

readpipe

Run a shell command and return its standard output.

`readpipe` hands `EXPR` to the shell, waits for the child to finish, and returns whatever the child wrote to its standard output. It is the named form of the `qx` (backticks) operator - ``cmd`` and `qx/cmd/` both compile down to `readpipe`. Reach for the named form when you want the command in a variable, when you want to suppress the double-quote interpolation `qx` performs, or when you want to read a captured command out of `@ARGV`. If `EXPR` is omitted, `$_` is used.

Synopsis

```
readpipe EXPR
readpipe
$out = readpipe $cmd;
@rows = readpipe $cmd;
```

What you get back

Context decides the shape:

- **Scalar context** - one string containing the entire captured output, newlines and all.
- **List context** - a list of records, split on `$/` (the input record separator, newline by default). The separator stays attached to each element unless you `chomp` it off.

`$?` is set to the child's wait status after the command finishes, exactly as with `system`. A non-zero `$?` is the usual signal that the command failed - the return value alone cannot tell you, since a failing command may still have written output (or nothing) before exiting.

```
my $out = readpipe "uname -a";
die "uname failed: \">$? = $?" if $?;
```

Global state it touches

- `$_` - used as the command when `EXPR` is omitted.
- `$/` - controls how list-context output is split into records. Set it to `undef` to slurp the whole output into a single element, or to `" "` for paragraph mode.
- `$?` - set to the child's wait status on return.
- `#!` - set if the shell itself could not be launched.
- `%ENV` - the shell inherits the current environment.

Examples

Capture one line of output:

```
my $host = readpipe "hostname";
chomp $host;
print "running on $host\n";
```

Command held in a variable - the reason `readpipe` exists as a named function. Backticks would still work here, but `readpipe` makes the intent explicit and avoids the visual noise of nested quoting:

```
my $cmd = "ls -l /etc";
my @entries = readpipe $cmd;
chomp @entries;
print scalar(@entries), " entries under /etc\n";
```

Slurp everything as one string by disabling record separation:

```
local $/; # slurp mode
my $log = readpipe "dmesg";
print length($log), " bytes of kernel log\n";
```

Paragraph mode - each element is one blank-line-separated block:

```
local $/ = "";
my @paragraphs = readpipe "cat README";
print scalar(@paragraphs), " paragraphs\n";
```

Check the exit status properly. The return value is output; success lives in `$?` :

```
my @files = readpipe "find . -name '*.tmp' 2>/dev/null";
if ($? == 0) {
    chomp @files;
    unlink @files;
} else {
    warn "find exited with status ", ($? >> 8), "\n";
}
```

Use `$_` as the command. Rarely the clearest form, but legal:

```
$_ = "date";
my $when = readpipe; # same as readpipe $_
```

Edge cases

- **No interpolation protection.** `readpipe` interpolates its argument exactly like a double-quoted string - the same as `qx`. `readpipe "rm $path"` with `$path` coming from user input is a shell-injection hole. Quote the value, or better, switch to a list-form `system / piped open` and skip the shell entirely.
- **Standard error is not captured.** Only the child's standard output comes back. Redirect `stderr` in the command itself if you want it merged: `readpipe "cmd`

2>&1".

- **Shell is mandatory.** Even a single-word readpipe "true" goes through /bin/sh, which means shell quoting, globbing, and environment-variable expansion all apply. No list form exists.
- **Large output buffers in memory.** The whole child output is read before readpipe returns. For a command that produces gigabytes, use a piped `open` and read incrementally instead.
- **Empty output, zero exit.** A successful command that writes nothing returns the empty string in scalar context and the empty list in list context. Distinguish “ran and said nothing” from “never ran” by checking `$?`.
- **Child died on signal.** `$?` encodes both exit code and signal
 - mask with `$? & 127` to get the signal number; `$? >> 8` for the exit code.
- **Windows quoting rules.** readpipe hands the string to the platform shell; on Windows that is `cmd.exe`, whose quoting and redirection rules differ from POSIX. `ppperl` is Linux-only, so this is only relevant when porting scripts in.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `qx` - the operator form; ``cmd`` and `qx/cmd/` are shorthand for readpipe and carry identical semantics
- `system` - run a command without capturing output; use when you only care about exit status
- `exec` - replace the current process with the command; no return, no capture
- `open` - piped `open` (`open my $fh, "-|", @cmd`) streams the child's output line-by-line without buffering it all
- `$?` - child wait status set by every process-launch builtin
- `$/` - input record separator that splits list-context output

Sockets

recv

Read an incoming message from a socket into a scalar.

`recv` reads up to `LENGTH` bytes from `SOCKET` into `SCALAR`, replacing whatever `SCALAR` previously held. `SCALAR` is grown or shrunk to the number of bytes actually received. `FLAGS` is the same bit mask the underlying `recvfrom(2)` system call takes - `0` for ordinary reads; the common non-zero values are `MSG_PEEK` (look without consuming), `MSG_WAITALL` (block until `LENGTH` bytes arrive), and `MSG_DONTWAIT` (never block). For a connectionless socket `recv` returns the sender's packed address; for a connected socket it returns the empty string. Under the hood this is `recvfrom(2)`, not `recv(2)`.

Synopsis

```
recv SOCKET, SCALAR, LENGTH, FLAGS
```

What you get back

- On success from a connectionless socket (UDP, UNIX datagram): the sender's address as a packed sockaddr string. Unpack it with `Socket`'s `sockaddr_in`, `sockaddr_in6`, or `sockaddr_un` depending on the address family.
- On success from a connected socket (TCP, connected UDP): the empty string `""`. The empty string is **true** for the defined-vs-undef check but **false** in boolean context - test with `defined`, not plain truthiness.
- On error: `undef`, with `$!` set to the `errno` from the failing `recvfrom(2)` call. Typical errors: `EAGAIN` / `EWOULDBLOCK` on a non-blocking socket with no data ready, `ECONNRESET` on a TCP peer reset, `EBADF` on a closed handle.

SCALAR is always resized to the byte count read. A zero-byte read on a connected stream socket means the peer closed its write side cleanly - `recv` returns `""` and SCALAR becomes the empty string.

Global state it touches

- `$!` is set on error to the system `errno`.
- No other special variables are involved. `recv` does not honour `$/`, `$,`, or `$\` - those are line-oriented I/O concepts and `recv` is a raw datagram / byte-stream primitive.

Examples

Read one UDP datagram and identify the sender:

```
use Socket;
my $from = recv($sock, my $buf, 1500, 0)
    or die "recv: $!";
my ($port, $ip) = sockaddr_in($from);
print "got ", length($buf), " bytes from ",
      inet_ntoa($ip), ":$port\n";
```

Read from a connected TCP socket. The sender address is empty, so check with `defined` rather than boolean truth:

```
defined(recv($sock, my $buf, 4096, 0))
    or die "recv: $!";
# $buf now holds 0..4096 bytes; 0 means the peer closed.
```

Peek at the next datagram without consuming it. The second call returns the same bytes:

```
use Socket;
recv($sock, my $peek, 1500, MSG_PEEK) or die $!;
recv($sock, my $consume, 1500, 0)      or die $!;
# $peek and $consume hold the same datagram.
```

Non-blocking read loop. EAGAIN / EWOULDBLOCK is the normal “no data yet” signal, not a real error:

```
use Errno qw(EAGAIN EWOULDBLOCK);
while (1) {
    my $from = recv($sock, my $buf, 1500, 0);
    unless (defined $from) {
        last if $! == EAGAIN || $! == EWOULDBLOCK;
        die "recv: $!";
    }
    handle_datagram($from, $buf);
}
```

Force a full-length read on a stream socket that might fragment the message - MSG_WAITALL blocks until LENGTH bytes are in or the connection dies:

```
use Socket;
defined(recv($sock, my $frame, 64, MSG_WAITALL))
    or die "recv: $!";
# $frame is exactly 64 bytes, or the peer closed early.
```

Edge cases

- **UTF-8-layered socket:** if SOCKET has a :utf8 layer (directly or via :encoding(. .)), recv throws an exception. recv is byte-oriented and does not interact with PerlIO layers; use `binmode` to strip the layer before reading, or decode the resulting bytes yourself.
- **LENGTH vs datagram size:** for datagram sockets, if LENGTH is smaller than the incoming datagram, the excess is **discarded silently**. Size the buffer to the protocol's MTU (typically 1500 for UDP over Ethernet, 65507 for the IPv4 UDP maximum).
- **"" return is not false-enough:** recv on a connected socket returns the empty string, which ! treats as false. Code like `recv( . . . ) or die $!` dies on every successful connected read. Use `defined`:

```
defined(recv($sock, $buf, $len, 0)) or die "recv: $!";
```

- **Zero-byte result on a stream socket:** recv returns "" and SCALAR becomes empty when the peer has performed an orderly shutdown. Distinguish end-of-stream from a real error by checking `length($buf) == 0` after a defined return.
- **Signals:** a blocking recv interrupted by a signal returns `undef` with `$! == EINTR`. Retry or install a proper `sigaction` with `SA_RESTART` if that's not what you want.
- **Closed socket:** recv on a closed or never-opened handle returns `undef`, sets `$!` to `EBADF`, and emits a `recv()` on closed socket warning under `use warnings`.
- **Pre-existing SCALAR contents are destroyed.** recv does not append; it overwrites. Use a fresh lexical or clear the scalar before each call.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `send` - the write side of the same socket; same FLAGS vocabulary, same connectionless-vs-connected distinction
- `socket` - create the SOCKET handle `recv` reads from
- `bind` - required before `recv` on a server-side datagram socket so the kernel knows which port to deliver to
- `connect` - when used on a datagram socket, makes `recv` return "" like a stream socket and filters out datagrams from other peers
- `sysread` - byte-level read for stream sockets when you do not need the sender address; integrates with `select`-based loops
- `Socket` - `sockaddr_in`, `inet_ntoa`, `MSG_PEEK`, `MSG_WAITALL`, and the rest of the constants `recv` consumes

Control flow

redo

Restart the current iteration of a loop without re-testing the condition and without running the `continue` block.

`redo` jumps back to the top of the loop body and re-runs it on the *same* input - the conditional that guards the loop is not re-evaluated, `$_` is not advanced by `while (<FH>)`, the iterator of a `foreach` is not advanced, and any `continue { }` block attached to the loop is skipped. The current pass happens again from the beginning.

Synopsis

```
redo
redo LABEL
redo EXPR
```

What you get back

`redo` does not return a value. It is a flow-control operator: it transfers control and never falls through to the next statement. Do not try to use it as the value of an expression - in particular, it cannot be used to return a value from `eval { }`, `sub { }`, or `do { }`.

Which loop it targets

With no argument, `redo` refers to the innermost enclosing loop. With a LABEL, it restarts the loop tagged with that label, allowing you to reach out of nested loops:

```

OUTER: while (...) {
    while (...) {
        redo OUTER;      # restart the while (...) above
    }
}

```

`redo` *EXPR* (added in Perl 5.18) computes the label at runtime. The expression must evaluate to a string naming a label currently in scope:

```

my $target = "OUTER";
redo $target;

```

A bare block counts as a loop that runs once, so `redo` inside a block turns it into a looping construct:

```

{
    my $line = <STDIN>;
    redo if $line =~ /\s*$/; # skip blank lines, re-read
    print $line;
}

```

redo vs next vs last

All three take an optional LABEL and all three affect loop control, but they differ in what they skip:

- `last` - exit the loop entirely. `continue` block is **not** run.
- `next` - go to the next iteration. `continue` block **is** run, then the loop condition is re-tested.
- `redo` - re-run the *same* iteration. `continue` block is **not** run, and the loop condition is **not** re-tested.

The `continue`-block difference is the one that catches people out. If a loop uses `continue { }` to increment a counter or advance a cursor, `redo` will not advance it - that's the whole point, but it is also the whole foot-gun.

Examples

Re-prompt until the user gives valid input. The `continue` block is skipped, so `$attempts` only counts *completed* iterations, which is usually what you want:

```

my $attempts = 0;
while (1) {
    print "Enter a positive integer: ";
    my $n = <STDIN>;
    chomp $n;
    redo unless $n =~ /^[0-9]+$/ && $n > 0;
    print "Got $n after $attempts rejected tries.\n";
    last;
} continue {

```

(continues on next page)

(continued from previous page)

```

    $attempts++;
}

```

Restart processing of the current line after patching it - the classic “lie to yourself about what was just read” pattern, here stripping Pascal-style block comments that may span multiple input lines:

```

LINE: while (<STDIN>) {
    while (s|({.*}.*){.*}|$1 |) {}
    s|{.*}| |;
    if (s|{.*}| |) {
        my $front = $_;
        while (<STDIN>) {
            if (/\/) {           # end of comment?
                s|^\|$front\{ |;
                redo LINE;       # re-process the patched line
            }
        }
    }
    print;
}

```

Target an outer loop by label. `redo INNER` would restart the inner loop on the current `$x`; `redo OUTER` restarts the outer loop on the current `$x` *without* advancing the `foreach` iterator:

```

OUTER: foreach my $x (@xs) {
    INNER: foreach my $y (@ys) {
        redo OUTER if should_restart($x, $y);
    }
}

```

Turn a bare block into a retry loop. Without `redo` this block runs exactly once; with `redo` it runs until the fetch succeeds:

```

my $tries = 0;
{
    my $ok = try_fetch();
    if (!$ok && ++$tries < 3) {
        sleep 1;
        redo;
    }
}

```

Edge cases

- **continue is not run.** This is the defining difference from `next`. If your loop relies on `continue { }` for counters, cursors, or cleanup, `redo` will bypass all of it. Either move the work into the body or use `next` instead.

- **Loop condition is not re-tested.** `while (cond) { ... redo; }` will keep looping even if `cond` would now be false. You are responsible for ensuring the `redo` path eventually exits via `last`, `return`, or a conditional fall-through.
- **Easy infinite loop.** `redo` with no state change is a tight infinite loop - typically not what you want. Always pair `redo` with a condition that eventually becomes false, or with a retry counter that triggers `last`.
- **foreach does not advance.** `foreach my $x (@xs) { redo }` keeps `$x` pinned at the current element and loops on it forever. The iterator only advances on fall-through or `next`.
- **while (<FH>) does not read the next line.** `$_` retains the current line across the `redo`, which is precisely why the Pascal comment stripper above works.
- **redo out of grep / map is not supported.** Upstream documents this as undefined; do not use `redo` to exit or restart a `grep BLOCK LIST` or `map BLOCK LIST`.
- **redo out of eval, sub, or do is not a return.** It performs flow control; there is no value to return. A `redo` that would have to cross such a boundary to reach a loop is a runtime error (“Can’t redo outside a loop block”).
- **Precedence quirk.** Unlike most named operators, `redo` has the precedence of assignment and is exempt from the looks-like-a-function rule. `redo ( "foo" ) . "bar"` passes the whole concatenation as the argument to `redo` - the parentheses do not isolate `"foo"`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `last` - exit the loop entirely; `continue` block is not run, same as `redo`
- `next` - advance to the next iteration; `continue` block is run and the loop condition is re-tested
- `return` - return from the enclosing subroutine, not just the loop; unlike `redo`, it yields a value
- `continue` - the block that `redo` skips but `next` runs; the core of the three-way distinction

Classes and OO

ref

Return a string describing what a reference points to.

`ref` examines its operand and returns a string identifying the kind of referent. For a blessed reference it returns the class name; for an unblessed reference it returns one of the built-in type strings (SCALAR, ARRAY, HASH, CODE, GLOB, LVALUE, IO, FORMAT, REF, VSTRING, REGEXP). For anything that is not a reference it returns the empty string. With no argument, `ref` operates on `$_`.

Synopsis

```
ref EXPR
ref
```

What you get back

A string. Three mutually exclusive categories:

- **Blessed reference** - the name of the class the referent is blessed into. For example, a `bless {}, 'Dog'` reference returns "Dog". Subclasses return their own name, not the parent's: a reference blessed into `Puppy` (an `ISA` of `Dog`) returns "Puppy". See [bless](#).
- **Unblessed reference** - one of the built-in type strings listed below, identifying the physical type of the referent.
- **Non-reference** - the empty string "".

The empty string is the **only** false value `ref` ever returns, so `if (ref $x)` is a safe “is this a reference?” test. See the edge cases section for why `if (ref $x eq 'HASH')` is *not* a safe “is this a hash reference?” test.

Built-in type strings

Returned for unblessed references according to the physical referent:

String	Referent
SCALAR	reference to an ordinary scalar
ARRAY	reference to an array
HASH	reference to a hash
CODE	reference to a subroutine
GLOB	reference to a typeglob
LVALUE	reference to an assignable expression
IO	reference to an I/O handle
FORMAT	reference to a format
REF	reference to another reference
VSTRING	reference to a v-string scalar
REGEXP	reference to a raw (unblessed) regexp

A compiled regex from `qr//` is *blessed* into the class `Regexp` at creation time, so `ref qr/.../` normally returns "Regexp", not "REGEXP". The bare `REGEXP` string appears only for unblessed regex referents, which are rare outside of internals.

Examples

Basic type probes:

```
ref \1      # "SCALAR"
ref []      # "ARRAY"
```

(continues on next page)

(continued from previous page)

```

ref {}           # "HASH"
ref sub { }      # "CODE"
ref \*STDOUT     # "GLOB"
ref \\1          # "REF"      (reference to a reference)
ref qr/foo/      # "Regexp"  (blessed)
ref "plain string" # ""       (not a reference)
ref 42           # ""
ref undef        # ""

```

Blessed references return the class name:

```

my $obj = bless {}, 'My::Widget';
ref $obj;           # "My::Widget"

```

Guard before dereferencing:

```

sub stringify {
    my ($v) = @_;
    return "[undef]" if !defined $v;
    return "[ref: " . ref($v) . "]" if ref $v;
    return $v;
}

```

No-argument form reads \$_:

```

for my $item (@things) {
    print "skipping non-ref\n" and next unless ref;
    ...
}

```

Picking the right test

ref is the oldest introspection primitive in the language and it conflates three different questions. Modern code should pick the operator that matches the question being asked.

Is this a reference at all?

```

if (ref $x) { ... }           # empty string is the only false_
↪return

```

Is this an object of (or derived from) a given class? Use the `isa` operator (or the `UNIVERSAL::isa` method) - it walks @ISA and honours subclassing, which a literal ref comparison does not:

```

if ($obj isa 'My::Widget') { ... }      # true for subclasses too
if ($obj->isa('My::Widget')) { ... }     # method form

if (ref($obj) eq 'My::Widget') { ... }  # FRAGILE: fails for_
↪subclasses

```

What is the underlying physical type, regardless of blessing? Use `Scalar::Util::reftype` - it ignores the class and returns the built-in type string even for blessed references:

```
use Scalar::Util qw(reftype blessed);
reftype($obj);           # "HASH" if $obj is blessed hash ref
blessed($obj);           # class name, or undef if not
↳blessed
```

Edge cases

- **Never use `ref` as a truth value of the referent.** A reference blessed into a class literally named `0` makes `ref $obj` return the string `"0"`, which is false. This is rare by construction but the foot-gun is old and well known. Compare to the empty string instead: `if (ref $x ne "")`.
- **Class name clashes with built-in type strings.** Nothing stops you from writing `package HASH; sub new { bless {}, shift }`. Then `ref(HASH->new)` returns `"HASH"` - indistinguishable from an unblessed hash reference by `ref` alone. Use `Scalar::Util::blessed` to disambiguate, or `reftype` to ask the physical-type question directly.
- **`ref` on `qr//` returns `"Regexp"`, not `"REGEXP"`.** `qr//` objects are blessed at construction; the uppercase form is only produced for the (unusual) unblessed regex reference.
- **Tied variables:** `ref` sees through the tie to the underlying reference. Use `tied` to retrieve the object backing the tie.
- **Weak references** are still references: `ref` behaves identically for weak and strong refs. Use `Scalar::Util::isweak` to detect weakness.
- **Overloaded stringification does not affect `ref`.** The return value of `ref` is produced by the runtime, not by the object's `" "` overload.
- **`ref` `EXPR` has very high precedence** - it is parsed as a named unary operator. Parenthesise when the argument is anything more complex than a simple variable:

```
ref $a || $b;           # parsed as (ref $a) || $b
ref($a || $b);          # what you probably meant
```

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `bless` - install a class name on a reference; the name installed here is exactly what `ref` returns afterwards
- `isa` - class-membership test that honours `@ISA` inheritance; prefer this over `ref ( . . ) eq 'ClassName'` for OO dispatch

- `tied` - retrieve the object implementing a tied variable; `ref` on a tied scalar follows the tie to the referent
- `Scalar::Util::reftype` - physical-type string for a reference, ignoring blessing
- `Scalar::Util::blessed` - class name if blessed, `undef` otherwise; the unambiguous counterpart to `ref` for OO code
- `perlref`, `perlobj` - reference and object model reference manuals

Filehandles, files, directories

rename

Change the name of a file.

`rename` asks the operating system to rebind the path `OLDNAME` to `NEWNAME`. On a POSIX filesystem this is a single atomic directory operation: after the call returns successfully, `NEWNAME` refers to the file that used to be reachable as `OLDNAME`, and `OLDNAME` no longer exists. It is the same primitive the shell `mv` command reaches for when source and destination live on the same filesystem.

Synopsis

```
rename OLDNAME, NEWNAME
```

What you get back

1 on success, 0 on failure with `$!` set to the `errno` the underlying `rename(2)` syscall returned. Always check the return value - a silent failure here leaves the filesystem in a state the rest of the program does not expect:

```
rename $tmp, $final
or die "rename $tmp -> $final failed: $!";
```

Atomicity and the same-filesystem rule

`rename` is atomic **only when `OLDNAME` and `NEWNAME` resolve to the same filesystem**. Within that filesystem, no intermediate state is ever visible: another process reading the directory sees either the old name or the new name, never both and never neither. This is why the canonical “write a file safely” idiom writes to a temporary file in the same directory and then renames it into place:

```
open my $fh, ">", "$path.tmp" or die "open: $!";
print $fh $data;
close $fh or die "close: $!";
rename "$path.tmp", $path
or die "rename: $!";
```

Across filesystems the call fails with `EXDEV` (“Invalid cross-device link”) and nothing is moved. The system `mv` command papers over this by falling back to copy-then-unlink; `rename` does not. For a portable cross-device move, use `move` from `File::Copy`, which performs the copy-and-unlink fallback itself.

Examples

Rename within a directory:

```
rename "draft.txt", "final.txt"
    or die "rename: $!";
```

Atomic file replacement - write to a sibling tempfile, then rename:

```
my $tmp = "$path.$$tmp";
open my $fh, ">", $tmp or die "open $tmp: $!";
print $fh $content;
close $fh          or die "close $tmp: $!";
rename $tmp, $path or die "rename $tmp -> $path: $!";
```

Detect the cross-device case and fall back to `File::Copy::move`:

```
use Errno qw(EXDEV);
use File::Copy qw(move);

unless (rename $src, $dst) {
    if ($! == EXDEV) {
        move($src, $dst) or die "move: $!";
    }
    else {
        die "rename: $!";
    }
}
```

Rename every `.log` in a directory to `.log.old`:

```
for my $f (glob "*.log") {
    rename $f, "$f.old"
        or warn "rename $f: $!";
}
```

Edge cases

- **NEWNAME already exists:** silently clobbered. `rename` does **not** ask, warn, or refuse. If the target is a regular file it is unlinked as part of the operation; if it is a directory the call fails with `ENOTDIR` or `EISDIR` depending on whether `OLDNAME` is a directory. To get “refuse to overwrite” semantics, test with `-e` first (and accept that the check is racy - there is no portable `renameat2(RENAMENOREPLACE)` exposed through `rename` itself).
- **Cross-filesystem (EXDEV):** fails, leaves both paths intact. Use `File::Copy::move` when the destination may be on another mount.
- **Open filehandles:** on Linux (and POSIX generally) renaming a file that is currently open is fine. The open filehandle keeps pointing at the same inode; subsequent reads and writes through it continue to work. New opens use the new name.
- **Directories:** renaming a directory is allowed, but only when `NEWNAME` does not

exist or is an empty directory. Moving a directory into a subdirectory of itself fails with EINVAL.

- **Permissions:** rename needs write + execute permission on both the source and destination parent directories, not on the file itself. Failure shows up as EACCES or EPERM.
- **Sticky-bit directories** (/tmp): if the destination parent has the sticky bit set, you must own either the source file or the destination parent; otherwise the call fails with EPERM even when the directory is world-writable.
- **Symlinks:** rename operates on the link itself, never on its target. rename "link", "other" renames the symlink; the file it points at is untouched.
- **Trailing slash on OLDNAME:** many kernels reject this with ENOTDIR when the source is not a directory. Do not rely on trailing-slash behaviour being portable.
- **Same-path rename:** rename \$p, \$p succeeds and does nothing (POSIX-required behaviour) as long as \$p exists.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- [unlink](#) - remove a file by name; the tear-down counterpart to creating one with rename into place
- [link](#) - make an additional hard-link name for an existing file without removing the original
- [symlink](#) - create a symbolic link; note that rename on a symlink moves the link, not its target
- `File::Copy` - portable move and copy that handle the cross-filesystem (EXDEV) case rename cannot
- `$!` - errno after a failed rename; compare against constants from [Errno](#) (EXDEV, EACCES, ENOENT, ...)

Modules

require

Load a Perl source file at runtime, or demand a minimum Perl version.

require has three forms that do quite different things. With a version number it asserts the running interpreter is new enough. With a bareword it loads a module by transforming `Foo::Bar` into `Foo/Bar.pm` and searching `@INC`. With a runtime string or `$_` it loads a filename verbatim. All three are **runtime** operations - contrast with [use](#), which is `require + import` wrapped in a compile-time BEGIN block.

Synopsis

```
require VERSION
require EXPR
require
```

What you get back

For the version-check form: 1 if the running Perl is new enough; otherwise an exception is thrown (Perl vX.Y.Z required--this is only ...). Nothing to check.

For the file-loading forms: the true value returned by the last statement of the loaded file, or 1 if the file enables the `module_true` feature (default under use v5.38 and later). A second `require` of the same path returns 1 without re-executing the file. Failure throws an exception - either the compile/runtime error from the file, `Can't locate Foo/Bar.pm in @INC ...`, or `Foo/Bar.pm` did not return true value.

Wrap in `eval` when you want to test whether a module is loadable:

```
eval { require Some::Optional::Module };
if ($?) { ... }                      # module missing or broken
```

Global state it touches

- `@INC` - the search path for bareword and relative-filename forms. Entries may be directory strings, coderefs, array refs, or blessed objects (see `@INC hooks` below).
- `%INC` - maps the requested filename (`Foo/Bar.pm`) to the absolute path that was loaded. `require` consults it to avoid double-loading and sets it on success. An entry of `undef` marks a file whose compilation failed.
- `$_` - used as the filename when `require` is called with no argument.
- `${^HOOK}{require__before}` and `${^HOOK}{require__after}` - optional coderefs run around every `require`, available for tracing and instrumentation wrappers.

The three forms

`require VERSION` - minimum Perl check

`VERSION` is either a v-string like `v5.38.0` (compared against `${^V}`) or a numeric literal like `5.038000` (compared against `$]`). If the running interpreter is older, `require` throws.

```
require v5.38;           # preferred
require 5.038;           # numeric; same effect
require 5.038_000;       # same; pre-5.6 syntax
```

This is a **runtime** check. If you want the check to happen at compile time (so the script never starts up on an old Perl), use `use VERSION` instead:

```
use v5.38;                # compile-time equivalent
```


require EXPR where EXPR is a bareword - module by name

A bareword is interpreted as a module name. `require` replaces every `::` with `/` and appends `.pm`, then searches `@INC`:

```
require Foo::Bar;      # looks for Foo/Bar.pm in @INC
```

This form also **autovivifies the stash** at compile time - the `Foo::Bar::` package hash springs into existence even if the load later fails. No symbols are imported; `require` only loads code.

Before trying `Foo/Bar.pm`, `require` looks for `Foo/Bar.pmc` in the same directory and prefers it if found. The `.pmc` hook lets build tools ship precompiled companion files alongside sources.

require EXPR where EXPR is a string - file by path

Any non-bareword expression is treated as a literal filename. No `::` translation, no `.pm` appending, no stash autovivification:

```
require "Foo/Bar.pm";      # explicit .pm path
require "$config_dir/init.pl"; # arbitrary filename

my $class = 'Foo::Bar';
require $class;             # ERROR: looks for file "Foo::Bar"
```

The last line is the classic trap: once the module name is in a scalar, it is a string, not a bareword. To load a module whose name is computed, either do the translation by hand or go through `eval`:

```
(my $file = "$class.pm") =~ s{::}{/}g;
require $file;

eval "require $class; 1" or die $@;
```

If EXPR is omitted entirely, `$_` is used.

The %INC guard - loaded files stay loaded

Before searching `@INC`, `require` consults `%INC`. If the requested filename is present and has a true value, `require` returns 1 immediately without re-reading the file. If the value is `undef`, a previous load failed and `require` throws `Compilation failed in require` - Perl refuses to retry a file that has already blown up once in this interpreter.

This means a module's top-level code runs **exactly once** per interpreter, no matter how many places `require` it. Setup in a module's file body is a one-shot operation; do not rely on it running for every `require` call in every caller.

A hook is free to set `%INC` itself; if it does not, `require` records the hook as the "source" of the load.

The 1; convention

Historically every loadable file had to return a **true** value as its last expression, or require threw filename did not return true value. The canonical way to guarantee this is a trailing 1;:

```
package My::Thing;
# ... module code ...
1;                                # mandatory, pre-5.38
```

As of Perl 5.38 the `module_true` feature - enabled by default under `use v5.38` or later - makes every required file implicitly return true on successful compilation. Files that opt in no longer need 1;. Files that don't, still do.

This only affects the compilation unit that turned the feature on; legacy modules loaded by a modern caller still need their own 1;.

When the loaded file does return a value and `module_true` is off, that value is what `require` returns - but **only on the first load**. Subsequent `require` calls of the same file return 1 from the `%INC` cache. To capture a file's return value reliably, use `do`:

```
my $config = do "config.pl"
or die "config.pl: ", $@ || $!;
```

@INC hooks - coderefs, array refs, and objects

Entries in `@INC` do not have to be directory strings. A reference stored in `@INC` is called a **hook** and gets a chance to satisfy the `require`.

Subroutine reference

Called with two arguments: the coderef itself and the filename (`Foo/Bar.pm`). It should return either nothing (meaning “I can't handle this; try the next entry”) or up to four values:

1. A scalar reference to source text prepended to the file.
2. A filehandle the rest of the source is read from.
3. A generator or filter subroutine (see `perldoc perlfunc` for the exact protocol).
4. Optional state passed back to the generator.

```
push @INC, sub {
    my ($self, $filename) = @_;
    return unless $filename eq 'Virtual/Module.pm';
    open my $fh, '<', \"package Virtual::Module; 1;";
    return $fh;
};
```

Array reference

The first element is a coderef or object as above; the rest is state the hook can read on each call. When the first element is an object with an INC or INC DIR method, the method receives the array ref as its third argument.

```
push @INC, [\&my_loader, $db_handle, $cache];
```

Blessed object

If the object has an INC or INC DIR method, require calls that method; otherwise, if the object is also a coderef, it's invoked like a plain subroutine reference. INC methods must be declared with their fully qualified name (sub Foo::INC { ... }), because the unqualified symbol INC is hard-wired to package main.

A hook may freely update %INC to record the path it supplied; if it doesn't, require stores the hook reference itself.

require vs use

```
use Foo::Bar qw(x y);

# is almost exactly:

BEGIN {
    require Foo::Bar;
    Foo::Bar->import(qw(x y));
}
```

Aspect	use	require
When it runs	Compile time (inside BEGIN)	Runtime
Calls import	Yes	No
Bareword :: → /	Yes	Yes, when called on a bareword
Version check form	use v5.38	require v5.38
Good for	Fixed dependencies	Optional / late-bound modules

Reach for require when the module is optional, loaded from a computed name, or loaded only inside a branch that may never run. Reach for use for everything else.

Examples

Minimum Perl version at runtime:

```
require v5.38;
```

Bareword module load - searches @INC, autovivifies Data::Dumper:: at compile time:

```
require Data::Dumper;
Data::Dumper->import;
print Data::Dumper::Dumper(\%ENV);
```

Optional module - fall back gracefully when missing:

```
my $have_json = eval { require JSON::PP; 1 };
if ($have_json) {
    print JSON::PP->new->encode($data);
}
```

Load a module whose name is computed at runtime. Convert to a filename path; require \$class would search for a literal file named Foo::Bar:

```
my $driver = $ENV{DB_DRIVER} // 'SQLite';
my $class  = "DBD:$driver";
(my $file  = "$class.pm") =~ s{::}{/}g;
require $file;
```

Load a configuration fragment by pathname, keeping its return value - note the `do`, not `require`, because we want the value every call:

```
my $config = do '/etc/myapp/config.pl'
    or die "config load failed: ", $@ || $!;
```

Install an `@INC` hook that serves a single virtual module from an in-memory string:

```
unshift @INC, sub {
    my ($self, $filename) = @_;
    return unless $filename eq 'MyApp/Version.pm';
    my $src = "package MyApp::Version; our \$VERSION = '1.23'; 1;";
    open my $fh, '<', \$src;
    $INC{$filename} = '(generated)';
    return $fh;
};
require MyApp::Version;
```

Edge cases

- **Bareword vs string:** `require Foo::Bar` and `require "Foo::Bar"` are **not** the same. The bareword form translates to `Foo/Bar.pm`; the string form looks for a file literally named `Foo::Bar`.
- **Computed class names:** `require $class` where `$class` holds `'Foo::Bar'` searches for a file named `Foo::Bar`, not `Foo/Bar.pm`. Translate the name yourself, or use `eval "require $class"` when the name is trusted.
- **No argument:** `require`; uses `$_` as the filename. Rare in modern code; almost always an accident.
- **Second-chance retries don't work:** If a `require` fails partway through compilation, `%INC` records `undef` for that filename. A later `require` of the same name sees

the `undef` and throws `Compilation failed in require` without re-running the file. Delete the `%INC` entry if you genuinely want to retry (this is a sharp edge; prefer fixing the underlying failure).

- **Relative . removal:** `.` was removed from the default `@INC` in Perl 5.26. A script that used to find `Foo/Bar.pm` next to itself now has to add its own directory explicitly (use `FindBin`; use `lib $FindBin::Bin`).
- **.pmc shadowing:** If `Foo/Bar.pmc` exists in the same directory as `Foo/Bar.pm`, the `.pmc` wins. Stale `.pmc` files are a classic “why is my edit not taking effect” trap.
- **Parsing `require (EXPR)`:** `require` is parsed as a named unary operator; `require EXPR + 1` means `require (EXPR + 1)`. Use explicit parentheses when the argument is a non-trivial expression.
- **Stash autovivification:** `require Foo::Bar` creates the `Foo::Bar::` stash even if the file is missing. Downstream defined `&Foo::Bar::some_sub` checks may pass the “package exists” stage but fail the “sub exists” stage.
- **Filehandle from a hook must be real:** Tied filehandles are silently ignored by the hook protocol; processing falls off the end and `require` reports “not found”. Use a real typeglob.
- **INC must be fully qualified in hook classes:** `sub Foo::INC { ... }` works; `sub INC { ... }` inside `package Foo` installs `main::INC` because of a hard-coded package rule.
- **`require()` is hard to wrap:** Many modules inspect the call stack to find their caller. A wrapper that adds a frame often breaks them. Prefer the `${^HOOK}{require__before}/${^HOOK}{require__after}` hooks (added in 5.38) for tracing.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `use` - compile-time `require + import`; the normal way to pull in a module you depend on unconditionally
- `do` - load a file and return its value on every call, with no `%INC` caching and no “must return true” rule
- `import` - the method `use` calls after `require`; `require` never calls it for you
- `@INC` - search path consulted for bareword and relative-filename loads; accepts directory strings, coderefs, array refs, and objects as hooks
- `%INC` - one-entry-per-loaded-file cache that makes `require` idempotent and records where each module came from
- `eval` - wrap `require` when the load is allowed to fail

Misc

reset

Clear every package variable whose name begins with one of a set of letters, and re-arm one-shot `m?pattern?` matches.

`reset` is a holdover from Perl 4's globals-only world. In modern code it survives in two narrow roles: re-arming `m? . . ?` one-match regexes so they fire again, and - at the end of a loop's `continue` block - wiping a batch of package variables before the next iteration. Its reach is limited to the current package and to **package** variables; anything declared with `my` is untouched.

Synopsis

```
reset EXPR
reset
```

EXPR is a string interpreted as a set of single characters, with hyphens allowed as ranges: 'X', 'a-z', 'abc', 'A-Za-z'. Every scalar, array, and hash in the **current package** whose name starts with one of those letters is reset to its pristine, undefined state.

With no argument, only `m?pattern?` one-match searches in the current package are re-armed - variables are not touched.

What you get back

Always 1. The return value carries no success signal; there is nothing for `reset` to fail at.

Global state it touches

- **Every package variable in the current package** whose name starts with a listed letter: scalars, arrays, and hashes all together. Scalars become `undef`, arrays and hashes become empty.
- **One-match regexes** (`m?pattern?`) compiled in the current package - their “already matched” flag is cleared so the next evaluation matches again.
- Does **not** cross package boundaries. `reset 'X'` inside package `Foo` leaves `$Bar::X` alone.
- Does **not** touch lexicals. `my $x` and `our $x` behave differently here: `our` is an alias to the package variable and is reset; `my` is not.

Examples

Re-arm all one-shot matches in the current package - the typical modern use of `reset`:

```
reset;
```

Clear every package variable starting with X in the current package:

```
package Stats;
our $Xtotal = 0;
our @Xsamples = (1, 2, 3);
our %Xindex = (a => 1);
reset 'X';
# $Xtotal, @Xsamples, %Xindex are all emptied
```

Reset at the bottom of a loop so the next iteration starts clean:

```
while (more_work()) {
    process();
} continue {
    reset 'abc';    # wipe $a*, @b*, %c* package vars for next pass
}
```

A character range - lower-case variables only:

```
reset 'a-z';
```

Combining individual letters and ranges:

```
reset 'a-cXY';    # a, b, c, X, Y
```

Edge cases

- **reset with no argument does not touch variables at all.** It re-arms one-shot `m?. ..?` matches in the current package and nothing else. If you want to clear variables you must pass an argument.
- **Lexicals are invisible to reset.** A my `$Xtotal` is completely unaffected by `reset 'X'`; only the package variable `$Xtotal` is. Code that migrated from `our/package` globals to `my` will find `reset` has silently become a no-op for those variables.
- **reset 'A-Z' is a foot-gun.** In the main package that wipes `@ARGV`, `@INC`, `%ENV` and every other upper-case global, almost certainly not what was intended.
- **Scalars, arrays, and hashes share one namespace here.** `reset 'X'` clears every variable of every sigil whose name starts with `X`, not just scalars. There is no way to restrict the reset to one sigil.
- **Current package is decided at compile time** via the enclosing `package` declaration, not by the caller. A `reset` call inside a module clears that module's variables, not the caller's.
- **No regex argument form.** `reset` takes a string of letters, not a regex or a list of names. You cannot write `reset qr/^X/` or `reset '$Xtotal'` - the latter tries to reset variables starting with `$`, `X`, `t`, `o`, `a`, `l`, which is almost never what the author meant.

Differences from upstream

- pperl's `reset` is a no-op - the call parses, returns 1, and has no side effects. Package variables are not cleared and `m?pattern?` one-shot match state is not tracked. Code that relies on `reset` to wipe a batch of globals at the end of a loop, or to re-fire a one-match regex, will not see that behaviour under pperl. Prefer `my` and ordinary `m//` regexes - both work identically on pperl and upstream, and are the recommended idiom upstream as well.

See also

- `my` - lexical variables; preferred in all new code and unaffected by `reset`
- `local` - dynamically scoped save/restore of a package variable, the usual answer when you want a temporary change without a manual cleanup
- `m` - the regular `m//` form; `m?...?` is the one-shot variant that `reset` re-arms
- `continue` - loop-end block where batch-reset of iteration state is conventionally placed
- `package` - declares which package's variables `reset` will operate on
- `undef` - clear a single named variable, the per-variable alternative to a broad `reset`

Control flow

return

Leave the current subroutine, `eval`, `do FILE`, `sort` block, or regex eval block, yielding a value to the caller.

`return` is the explicit exit from a named or anonymous sub. It stops evaluation of the sub body at the point of call and hands `EXPR` back to whoever invoked the sub. The value `EXPR` produces depends on the caller's context - list, scalar, or void - which the body can inspect with `wantarray` before deciding what to return.

Synopsis

```
return EXPR
return
```

What you get back

`return` itself does not have a return value - it does not continue in the current frame. It terminates the frame and the caller receives `EXPR`, evaluated in the caller's context:

- **List context** - `EXPR` is evaluated in list context; its list value is what the caller sees.
- **Scalar context** - `EXPR` is evaluated in scalar context; its scalar value is what the caller sees.
- **Void context** - `EXPR` is still evaluated (for side effects), but the value is discarded.

With no `EXPR` the caller receives:

- **list context** - the empty list `()`
- **scalar context** - `undef`
- **void context** - nothing

This asymmetry matters: `return;` (bare) is **not** the same as `return undef;`. A bare `return` in list context yields `()`, which is false and has zero elements; `return undef;` in list context yields a one-element list containing `undef`, which is **true** (because the list has one element). Error-signalling subs should almost always use bare `return`.

```
sub find_it {
    my ($key) = @_;
    return unless exists $cache{$key}; # bare: () in list, undef in_
    ↪ scalar
    return $cache{$key};
}

if (my @hits = find_it($k)) { ... } # () is false; undef-in-list_
    ↪ is true
```

Where return is legal

`return` unwinds these frames:

- a named or anonymous **subroutine**
- a string or block **eval** - `return` from inside `eval { ... }` leaves the `eval`, not the enclosing sub
- a **do FILE** - `return` `EXPR` is the file's value
- a **sort comparator block** - returns the comparison result
- a **regex eval block** - `/ (? { ... } ) /` and `/ (? &NAME ) /` code blocks

`return` is **not** legal inside these blocks - it is a compile-time or run-time error depending on the context:

- **grep BLOCK LIST** and **map BLOCK LIST** - the block is not a sub frame; use the block's last expression instead
- **do BLOCK** - the block is an expression, not a call; its value is already the block's last expression
- bare blocks, `if` / `while` / `for` bodies - same reason; these do not form a call frame

If you need “return a value from this block”, make it the block's last evaluated expression:

```
my @evens = grep { $_ % 2 == 0 } @nums; # not: grep { return ... }
```

Context propagation with wantarray

The caller's context reaches the callee through `wantarray`:

- `wantarray` is **true** in list context
- `wantarray` is **defined and false** ("") in scalar context
- `wantarray` is **undef** in void context

A sub that wants to return shape-appropriate values inspects it:

```
sub stats {
    my @xs = @_;
    return unless @xs;           # bare: () / undef as _
    ↪above
    my $sum = 0; $sum += $_ for @xs;
    my $avg = $sum / @xs;
    return wantarray ? ($sum, $avg) : $avg;
}

my ($s, $a) = stats(@data);      # list context → two _
    ↪values
my $a       = stats(@data);      # scalar context → _
    ↪average only
stats(@data);                    # void context → _
    ↪computed, discarded
```

The same call site can be evaluated in different contexts across different invocations; return re-resolves EXPR each time.

Implicit return: last expression wins

If control reaches the end of a sub (or `eval`, or `do FILE`) without hitting `return`, the value of the last evaluated expression is returned, in the caller's context. This is the idiomatic “trailing expression” style:

```
sub square { my ($x) = @_; $x * $x }      # implicit return of $x *
    ↪$x

sub classify {
    my ($n) = @_;
    $n < 0 ? "negative"
    : $n == 0 ? "zero"
    : "positive"                          # trailing ternary is the _
    ↪value
}
```

Use explicit return when you exit early, or when the end of the sub is a statement (e.g. a loop, an assignment) whose value is not what you want the caller to see.

Examples

Early exit on a guard:

```
sub lookup {
    my ($id) = @_;
    return unless defined $id;
    return $table{$id};
}
```

Return from an `eval` block without unwinding the enclosing sub:

```
sub try_parse {
    my ($text) = @_;
    my $parsed = eval {
        return {} if $text eq "";           # leaves eval, not try_
    }
    ↪ parse
    parse($text);
};
return $@ ? undef : $parsed;
}
```

Context-sensitive return:

```
sub users {
    return wantarray ? @all_users : scalar @all_users;
}

my @u = users();           # the users themselves
my $n = users();           # how many there are
```

Parser exemption - return is not a looks-like-a-function operator:

```
sub greet { return ("hello") . " world" } # returns "hello world"
                                           # the . " world" is part_
↪ of EXPR
greet();                                   # "hello world"
```

This is why `return (EXPR)` does **not** insulate `EXPR` from a trailing operator; anything after the closing paren is still argument to `return`. Same rule as `print`.

Illegal: return inside `grep` / `map`:

```
my @odd = grep { return $_ % 2 } @n;      # wrong; leaves the sub
my @odd = grep { $_ % 2 } @n;            # right
```

Inside a `grep` / `map` the block's last expression is already the value; a return there exits the **enclosing sub**, which is almost never what you mean.

Edge cases

- **Bare return vs return undef:** as above, bare return yields `()` in list context, which lets `if (my @x = f())` work as a “no results” test. `return undef` puts one element in the list and flips that test to true. Prefer bare return unless you specifically want a defined-undef scalar.
- **return from a sort comparator:** legitimate and common in complex comparators:

```
sort { return $a->[0] <=> $b->[0] || $a->[1] cmp $b->[1] }  
    ↪ @rows ;
```

- **return from do FILE:** becomes the file’s value, which is what `require` and `use` inspect to decide module-load success. A `.pm` ending in `1;` is the common idiom; `return 1;` near the top of the file is equivalent.
- **return from a regex (?{ ... }) block:** ends the code block only, not the regex match. Rarely useful; normal block flow is almost always clearer.
- **Void-context side effects still happen:** `return expensive();` in void context still calls `expensive()`. `return` is not an optimisation hint.
- **Not a function, no parentheses needed:** `return` is an operator, not a function. `return(1, 2, 3)` and `return 1, 2, 3` are the same. The parser exemption above means parens around `EXPR` do not delimit it from the rest of the statement - parens are purely for grouping inside `EXPR`.
- **Return in a block form of eval:** `eval { return $x }` leaves the `eval` with `$x`; `$@` is cleared (no exception). To return from the enclosing sub via an exception, `die` instead.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `wantarray` - inspect the caller’s context so `return` can yield the right shape
- `caller` - the other half of caller introspection; useful when a sub decides what to return based on who called it
- `eval` - return inside an `eval` block leaves the `eval`, not the enclosing sub
- `sub` - how subroutines are defined and how implicit return interacts with the last expression
- `die` - the non-local exit; use when you want to unwind past intermediate frames instead of returning through each one
- `last` - exits a loop, not a sub; do not confuse the two

[SCALARS and strings](#) · [Lists](#)

reverse

Reverse a list - or, in scalar context, reverse the characters of a string.

`reverse` is two operators wearing the same name. Context decides which one you get. In list context it returns LIST with its elements in the opposite order; element values are untouched. In scalar context it concatenates LIST into a single string and returns that string with its **characters** reversed. The common mistake of writing `my $s = reverse $str` and expecting a reversed list (or writing `print reverse $str` and expecting a reversed string) is the single biggest trap this function has - see *Edge cases*.

Synopsis

```

my @r    = reverse @arr           # list context: elements reversed
my $s    = reverse $str          # scalar context: characters_
↪reversed
my %inv   = reverse %h           # hash invert (value ↪ key)
reverse   # in scalar context, reverses $_

```

What you get back

Context-dependent. The operator inspects the context of its caller and branches:

- **List context** → a new list with the elements of LIST in reverse order. Element values are not stringified, not modified, not copied at the character level. A list of refs stays a list of the same refs.
- **Scalar context** → a single string: the concatenation of the stringified elements of LIST, with all characters in reverse order. Under use `utf8` / wide strings, reversal is character-wise, not byte-wise.

This means the *same* call site returns radically different shapes depending on how you use it:

```

my @back = reverse "abc", "def"; # ("def", "abc")
my $back = reverse "abc", "def"; # "fedcba"

```

Used with no arguments in scalar context, `reverse` reverses `$_`. Used with no arguments in list context, `reverse` returns the empty list - **not** a reversed `$_`. `print reverse;` produces no output.

Global state it touches

Reads `$_` when called with no arguments in scalar context. That is the only global it touches. It does not modify `$_`; it returns the reversed string.

Examples

Reverse a list. The elements themselves are unchanged:

```

my @a = (1, 2, 3, 4, 5);
my @r = reverse @a;           # (5, 4, 3, 2, 1)

```

Reverse a string. Force scalar context with `scalar` when the surrounding expression would otherwise be list:

```
my $s = reverse "Hello, world"; # "dlrow ,olleH"
print scalar reverse "Hello";   # "olleH"
```

Iterate an array backward without building a second array. `reverse` in the loop header is a list-context call; Perl consumes the reversed list lazily from the loop machinery:

```
my @stack = ("first", "second", "third");
for my $elem (reverse @stack) {
    print "$elem\n";           # third / second / first
}
```

Descending sort. `reverse sort` is the idiom when your comparison is ascending and you want the opposite. For anything more complex than default string compare, swap the operands in the comparator instead:

```
my @asc  = sort { $a <=> $b } @nums;
my @desc = reverse sort { $a <=> $b } @nums;    # works
my @desc2 = sort { $b <=> $a } @nums;          # preferred
```

Invert a hash. Values become keys, keys become values:

```
my %by_name = (alice => 1, bob => 2, carol => 3);
my %by_id   = reverse %by_name;                # (1 => "alice", ..
→.)
print $by_id{2};                               # "bob"
```

The "dlrow ,olleH" trick - using `reverse` in scalar context as a one-shot character flip on a string variable:

```
my $word = "stressed";
my $rev  = scalar reverse $word;                # "desserts"
```

Edge cases

- **The #1 trap: context.** `my $s = reverse @arr` does **not** give you a reversed array; it gives you the reversed *string* formed by concatenating the array's elements and flipping every character. If you want a reversed array stored via a scalar, use a reference: `my $ref = [ reverse @arr ]`.
- **Single scalar, wrong context.** `my @r = reverse $str` returns `($str)` - a one-element list containing `$str` unchanged. There is nothing to reorder. To reverse the characters, force scalar context: `my $r = reverse $str` or `my @r = (scalar reverse $str)`.
- **Hash invert loses data on duplicate values.** `reverse %h` walks the key/value pairs as a flat list and rebuilds a hash. If two keys share a value, one of them wins and the other is silently dropped; which one wins is not defined. Check for uniqueness first if it matters:

```
my %seen;
$seen{$_}++ for values %h;
die "non-unique values" if grep { $_ > 1 } values %seen;
my %inv = reverse %h;
```

- **No arguments in list context does nothing useful.** `reverse` with no args returns `()` in list context. `$_` is only consulted in scalar context. `print reverse;` prints nothing; `print scalar reverse;` prints the reverse of `$_`.
- **Character-wise, not byte-wise.** For wide-character strings, `reverse` reverses characters, not bytes. A `:utf8` string of three code points stays three code points after reversal, regardless of their byte widths. Reversing a raw byte string with multi-byte characters you never decoded will corrupt it - decode first.
- **Self-assignment preserves holes.** `@a = reverse @a` on a non-magical or well-behaved tied array preserves non-existent elements in their reversed positions; a sparse array stays sparse.
- **`reverse sort` is not the same as `sort { $b cmp $a }`.** Both give descending order for a stable default compare, but `reverse sort` does two passes over the data and is measurably slower on large lists. Prefer swapping the comparator operands.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `sort` - pair with `reverse` for descending order, or swap `$a/$b` in the comparator to avoid the second pass
- `pop` - remove from the end of an array when you just want the last element, not the whole list reversed
- `shift` - remove from the front; combined with a reversed copy you get stack-like access to the original tail
- `split` - produce a list of characters (with `//`) or fields that you can then reverse
- `join` - the natural partner when you reverse a list of strings and want a single string back
- `$_` - the default target when `reverse` is called with no arguments in scalar context

I/O

rewinddir

Reset a directory handle to the start of its listing.

`rewinddir` sets the current position of `DIRHANDLE` back to the beginning of the directory, so that the next `readdir` call returns the first entry again. It is the directory-handle counterpart to `seek` on a file handle: no re-open, no re-scan of the parent directory - just a position reset on the open handle.

Synopsis

```
rewinddir DIRHANDLE
```

What you get back

1 on success, 0 on failure (with `$!` set). Failure is rare in practice - the usual cause is a handle that was never opened or has already been closed. Check the return value if you care about either case:

```
rewinddir $dh
    or die "rewinddir failed: $!";
```

Typical use

You have already walked a directory with `readdir` and want to walk it again without paying for another `opendir`:

```
opendir my $dh, $path or die "opendir $path: $!";

my @all = readdir $dh;           # first pass: collect everything

rewinddir $dh;                   # back to the top

while (my $entry = readdir $dh) { # second pass: process one by one
    next if $entry =~ /^\.\/?$/;
    process("$path/$entry");
}

closedir $dh;
```

A second common pattern is reusing one handle across several scans with different filters, where re-opening would be wasteful or would race against concurrent directory changes:

```
opendir my $dh, $path or die $!;

my @perl = grep { /\.[plm]\z/ } readdir $dh;
rewinddir $dh;
my @text = grep { /\.txt\z/ } readdir $dh;

closedir $dh;
```

Edge cases

- **Unopened or closed handle:** returns 0 and sets `$!`. Under use warnings you also get a `rewinddir()` attempted on invalid dirhandle warning.
- **Mid-scan directory changes:** `rewinddir` is **not** guaranteed to surface entries created (or hide entries removed) after `opendir` returned. The underlying rewind-

`dir(3)` operates on the kernel's cached directory stream; whether a rewind scan observes concurrent `mkdir`, `unlink`, or `rename` in the same directory is filesystem- and OS-dependent. If you need a fresh snapshot, `closedir` and `opendir` again.

- **Not a seek to an arbitrary position:** `rewinddir` resets to the start and nothing else. To return to a remembered mid-directory position use `telldir` / `seekdir` instead.
- **Entry order is unspecified:** rewinding replays the same stream the kernel gave you, but the order of entries is whatever the filesystem produces - not sorted, not creation order, not the order you saw last time on a different handle. Sort explicitly if order matters.
- **Portability:** `rewinddir` is a POSIX call. On platforms without a working `rewinddir(3)` it may be implemented by closing and re-opening the directory; see `L<perlport/rewinddir>` for the historical list. On Linux (the platform `ppperl` targets) this is not a concern.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `opendir` - open a directory and obtain the handle that `rewinddir` operates on
- `readdir` - read the next entry; resumes from wherever `rewinddir` left the stream (the beginning)
- `closedir` - release the handle; pair with `opendir` when a rewind scan is not enough and you need a fresh snapshot of the directory
- `telldir` - record the current position so you can return to it later
- `seekdir` - restore a position previously captured by `telldir`; `rewinddir $dh` is equivalent to `seekdir $dh, 0` on conforming systems, but `rewinddir` is the portable spelling

SCALARS and strings

rindex

Find the position of the **last** occurrence of a substring inside a string.

`rindex` is the right-to-left counterpart of `index`. It scans `STR` for `SUBSTR` and returns the zero-based offset of the rightmost match, or `-1` if no match exists. When `POSITION` is given, the search is constrained to matches whose starting offset is less than or equal to `POSITION` - that is, the last occurrence at or before that position.

Synopsis

```
rindex $str, $substr
rindex $str, $substr, $position
```

What you get back

An integer. Zero or positive means the zero-based offset of the rightmost match; -1 means SUBSTR does not occur in STR (subject to the POSITION constraint, if any).

Offsets are measured in **characters** when STR holds a character string, and in **bytes** when STR is a byte string. This matches `index` and the rest of Perl's string built-ins - the unit is whatever the string itself is using.

How POSITION shapes the search

POSITION is an **upper bound on the match's starting offset**, not on where scanning begins. The result satisfies:

- `result <= POSITION`
- `result + length(SUBSTR) <= length(STR)` (as always)

Concretely:

- POSITION omitted or `undef` - search the whole string; return the rightmost match.
- POSITION greater than or equal to `length(STR)` - behaves as if POSITION were `length(STR)`; the entire string is considered.
- POSITION less than 0 - behaves as if POSITION were 0; only a match starting at offset 0 can satisfy the constraint, so the return value is 0 when STR begins with SUBSTR and -1 otherwise.
- Empty SUBSTR - every offset from 0 to `length(STR)` is a valid match position, so `rindex` returns `min(POSITION, length(STR))` (or `length(STR)` when POSITION is omitted).

Think of POSITION as “find the last match at or before here,” not as “start scanning here.”

Examples

Basic right-to-left search:

```
rindex("banana", "a");      # 5  (the final 'a')
rindex("banana", "na");     # 4  (last "na")
rindex("banana", "z");     # -1 (not found)
```

Stepping backwards through every occurrence by using each result, minus one, as the next POSITION:

```
my $str = "banana";
my $pos = length($str);
while ((my $hit = rindex($str, "a", $pos - 1)) >= 0) {
    print "match at $hit\n";    # 5, 3, 1
    $pos = $hit;
}
```

POSITION as an upper bound - the match must **start** at or before it:

```
rindex("banana", "na", 3);    # 4? No: 4 > 3, so the match at 4
                                #   is excluded; returns 2
rindex("banana", "na", 4);    # 4   (match starting at 4 is allowed)
```

Empty SUBSTR matches at every offset, so the result is the capped position:

```
rindex("banana", "", 4);      # 4
rindex("banana", "");        # 6   (length of "banana")
```

Pairing with `index` to locate the last segment of a delimited string - a common idiom for splitting off a final extension or path component:

```
my $path = "/usr/local/bin/perl";
my $slash = rindex($path, "/");
my $file  = substr($path, $slash + 1);    # "perl"
```

Edge cases

- **Substring longer than string:** `rindex("abc", "abcd")` returns -1. No match can fit.
- **Substring equals string:** `rindex("abc", "abc")` returns 0. The last (and only) match starts at offset 0.
- **POSITION past the end:** `rindex($str, $sub, 9999)` behaves as if POSITION were `length($str)` - the entire string is scanned.
- **POSITION is negative:** treated as 0. `rindex("abc", "b", -5)` returns -1 (no match can start at offset 0 or earlier); `rindex("abc", "a", -5)` returns 0.
- **POSITION is undef:** same as omitting it. Under `use warnings` this emits an uninitialized warning.
- **Empty STR, non-empty SUBSTR:** returns -1.
- **Empty STR, empty SUBSTR:** returns 0.
- **Unicode:** for character strings (those the interpreter knows are text), offsets count characters, not bytes. A string flipped to byte semantics via `use bytes` or `encode_utf8` will report byte offsets instead. The rule mirrors `index` and `substr`.
- **Tainted arguments:** the result inherits taint from STR and SUBSTR under `-T`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `index` - the left-to-right counterpart; identical argument conventions, returns the **first** occurrence
- `substr` - typically the next call after a successful `rindex`, to slice the matched region or what follows it

- `length` - useful when computing the trailing slice: `substr($s, $hit + length($sub))`
- `pos` - for regex-driven scanning where you need the engine's match state rather than a single substring offset
- `split` - when you want every segment delimited by SUBSTR, not just the last one
- `reverse` - combined with `index` it is sometimes mistaken for a `rindex` substitute; `rindex` is both clearer and faster

Filehandles, files, directories

rmmdir

Remove an empty directory.

`rmmdir` asks the operating system to delete the directory named by `FILENAME`. The directory must be empty - it must contain no entries other than `.` and `...`. If `FILENAME` is omitted, `rmmdir` operates on `$_`. This is a thin wrapper over the `rmmdir(2)` system call and carries all of its restrictions: you cannot remove a non-empty directory, you cannot remove a directory you lack write permission on the parent of, and on most systems you cannot remove your current working directory.

Synopsis

```
rmmdir FILENAME
rmmdir
```

What you get back

1 on success, 0 on failure (with `$!` set to the `errno` from the failed system call). Unlike most I/O primitives `rmmdir` does not return `undef` on failure - it returns the integer 0, so the common test `rmmdir $dir or die $!` still works because 0 is false.

Always check the return value. A failed `rmmdir` is silent otherwise; the next line of code will happily proceed as if the directory had been removed.

```
rmmdir $dir
or die "rmmdir $dir failed: $!";
```

Global state it touches

- `$_` - used as `FILENAME` when no argument is given.
- `$!` - set to the `errno` of the failed `rmmdir(2)` call on failure; left untouched on success. Read it immediately; any intervening system call may overwrite it.

Examples

Plain call, explicit directory:

```
rmmdir "build/tmp"
    or die "rmmdir build/tmp: $!";
```

Default target. Inside `while (readdir ...)` or a loop that leaves the name in `$_`:

```
for (@empty_dirs) {
    rmmdir or warn "rmmdir $_: $!";
}
```

Distinguish “already gone” from a real error. `ENOENT` means the directory does not exist, which is often fine for cleanup code:

```
use Errno qw(ENOENT);
unless (rmmdir $dir) {
    die "rmmdir $dir: $!" unless $! == ENOENT;
}
```

Remove a directory only after emptying it first:

```
opendir my $dh, $dir or die "opendir $dir: $!";
while (my $entry = readdir $dh) {
    next if $entry eq '.' or $entry eq '..';
    unlink "$dir/$entry" or die "unlink $dir/$entry: $!";
}
closedir $dh;
rmmdir $dir or die "rmmdir $dir: $!";
```

Recursive removal is **not** `rmmdir`’s job. Use `File::Path`’s `remove_tree` - it walks the tree and deletes entries bottom-up:

```
use File::Path qw(remove_tree);
remove_tree("build", { error => \my $err });
die "remove_tree: @$err" if @$err;
```

Edge cases

- **Directory not empty:** `$!` is set to `ENOTEMPTY` on most Unix systems (some BSDs use `EEXIST`). This is the single most common `rmmdir` failure. List the directory contents or call `remove_tree` if recursive deletion is what you actually want.
- **No argument:** `rmmdir` with no argument uses `$_`, not `@ARGV` and not the current directory. Writing `rmmdir()` with empty parentheses is the same as `rmmdir` with no argument - it still reads `$_`.
- **Symlink to a directory:** `rmmdir` refuses to remove symlinks; it operates on directories only. Use `unlink` on the symlink itself. `$!` is typically set to `ENOTDIR`.
- **Current working directory:** On Linux, `rmmdir "."`; usually fails with `EBUSY` or `EINVAL` depending on the filesystem. Do not rely on being able to remove the directory you are sitting in; `chdir` elsewhere first.

- **Permissions:** you need write + execute permission on the **parent** directory, not on the directory being removed. Removing `/a/b/c` checks `/a/b`'s mode, not `/a/b/c`'s.
- **Sticky-bit parent** (e.g. `/tmp`): you also need to own the directory being removed or own the parent; otherwise `EPERM`.
- **Trailing slash:** `rmdir "foo/"` works and means the same as `rmdir "foo"`.
- **Race with another process:** between your `opendir/readdir` loop and the final `rmdir`, another process may create a file in the directory. You will then see `ENOTEMPTY` despite having just emptied it. Retry or fail loudly - do not silently ignore.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `mkdir` - create a directory; the inverse operation, with the same `FILENAME-or-$_` argument convention
- `unlink` - remove **files** (and symlinks); `rmdir` removes only empty directories
- `chdir` - change the current working directory, often used before `rmdir` to avoid trying to remove the directory you are in
- `opendir` / `readdir` - list a directory's contents before emptying and removing it
- `$!` - holds the `errno` of a failed `rmdir`; check for `ENOENT` / `ENOTEMPTY` / `EPERM` to branch on the reason

Regular expressions and pattern matching

`s///`

Search a string for a pattern and replace every match with a replacement.

`s///` is Perl's substitution operator. It finds text matching `PATTERN` in the target string, replaces each match with `REPLACEMENT`, and by default returns the number of substitutions made. With no explicit target (no `=~` or `!~`), it operates on `$_`. The target must be a scalar lvalue - a variable, an array or hash element, or a `substr` / assignment producing one - unless the `/r` flag is used, which works on any value and returns the modified copy.

Synopsis

```
s/PATTERN/REPLACEMENT/FLAGS
$str =~ s/PATTERN/REPLACEMENT/FLAGS
$str !~ s/PATTERN/REPLACEMENT/FLAGS
my $copy = $str =~ s/PATTERN/REPLACEMENT/r
```

What you get back

By default, the count of substitutions performed as an integer: 0 when nothing matched (which is false in boolean context), a positive integer on success. The target scalar is modified in place.

With the `/r` flag, the semantics flip: the target is **not** modified, and the operator returns the (possibly changed) string. If nothing matched, `/r` still returns a copy of the original. The result of `/r` is always a plain string, even when the source is a tied variable or a blessed object with overloading.

`!~` inverts the boolean sense of the return value: it yields true when nothing was substituted, false when at least one substitution happened. The target is still modified either way.

```
my $n = ($text =~ s/foo/bar/g);    # count of replacements
my $out = $text =~ s/foo/bar/r;    # copy-and-modify, $text
→ untouched
```

Global state it touches

- `$_` - the default target when no `=~` or `!~` is given. `s/old/new/` with no binding reads and writes `$_`.
- `$&`, `[$`](../perlvar)`, `[$`](../perlvar)`, `[$1](../perlvar)` and friends - set by each successful match, the same way `[m/](m)` sets them. Inside `'REPLACEMENT'` they refer to the current match, not any earlier one.
- `pos` - cleared on the target scalar by a successful non-`/g` substitution; under `/g` it advances with each match and can be inspected between iterations.
- The **last successful pattern** - used when `PATTERN` evaluates to the empty string (`s//new/` re-uses the last successful regex from this scope).
- `$/, $\\` and output separators are not touched - `s///` produces a string, it does not write one.

Flags

The full set is `msixpodualngcer`. Match-side flags behave exactly as in `m//` and `qr`; see the [modifiers chapter](#) of the regex guide for the match-side details. Replacement-specific flags:

- `e` - evaluate `REPLACEMENT` as a block of Perl code. The block's value is the replacement string. Equivalent to wrapping it in `do { ... }` and stringifying the result.
- `ee` - evaluate as code, then `eval` the resulting string again. Equivalent to `eval (do { REPLACEMENT })`. Additional `e` modifiers add more `eval` layers.
- `r` - non-destructive: leave the target unchanged and return the modified copy (or an unchanged copy on no match). Cannot be combined with a non-scalar-lvalue target restriction - any expression is a valid target under `/r`.
- `g` - global: replace every non-overlapping match, not just the first.

- `c` - accepted for syntactic symmetry with `m//` but has no behavioural effect on `s///`; triggers a warning under `use warnings`.

Match-side flags in one line: `m` (multiline `^/$`), `s` (`.` matches newline), `i` (case-insensitive), `x` / `xx` (extended whitespace, comments), `p` (preserve match variables; obsolete since 5.20), `o` (compile pattern once), `d` / `u` / `a` / `l` (character-set semantics), `n` (non-capturing by default). See the *modifiers chapter*.

Delimiters

Any non-whitespace character may replace `/`. If the chosen delimiter is one of `(`, `[`, `{`, or `<`, the operator takes **two** bracketed groups and a delimiter between them is optional:

```
s(foo)(bar)
s{foo}{bar}
s<foo>/bar/
s[foo]{bar}g
```

Single-quote delimiters turn off interpolation in both PATTERN and REPLACEMENT (unless `/e` is in force, which always parses REPLACEMENT as code):

```
s'$name'$value';    # literal $name → literal $value, no
→ interpolation
```

Backticks are ordinary delimiters; `s`x`y`` does not run a shell command.

When the delimiter is a character that is also an identifier character, a space after the `s` is required so the parser does not read it as part of the operator name.

Examples

Basic in-place substitution on `$_`:

```
$_ = 'the quick brown fox';
s/quick/slow/;                # $_ is now 'the slow brown fox'
```

Explicit target with `=~`:

```
my $path = '/usr/bin/perl';
$path =~ s|/usr/bin|/usr/local/bin|;
# $path is '/usr/local/bin/perl'
```

Global replacement, capturing the count:

```
my $text = 'one two two three two';
my $n = ($text =~ s/two/2/g);    # $text: 'one 2 2 three 2'; $n == 3
```

Non-destructive `/r` - leaves the source alone, returns the copy. Idiomatic in `map`:

```
my @clean = map { s/\s+\z//r } @lines;    # @lines untouched
```

Code evaluation with `/e`. The right-hand side is a Perl expression whose value becomes the replacement:


```
my $s = 'abc123xyz';
$s =~ s/\d+/$& * 2/e;           # $s is 'abc246xyz'

my %percent = (n => "\n", t => "\t");
$s =~ s/%(.)/$percent{$1} || $&/ge;    # expand %-escapes
```

Chained /r - pure-functional transforms with no intermediate variable:

```
my $out = $text =~ s/foo/bar/r
           =~ s/baz/qux/r;
```

Capture groups and backreferences in the replacement - note the \$1 form, not \1:

```
# swap the first two whitespace-separated fields
s/(\S+)\s+(\S+)/$2 $1/;
```

The 1 while idiom for substitutions that create new matches:

```
# insert commas into an integer from the right
my $n = '1234567890';
1 while $n =~ s/(\d)(\d\d\d\d)(?! \d)/$1,$2/;
# $n is '1,234,567,890'
```

Edge cases

- **Target must be a scalar lvalue** (unless /r): a variable, element, or `substr` lvalue. `s/x/y/` against a string literal or the result of a function call is a compile-time error. Under /r any expression is fair game since nothing is modified.
- **Empty pattern re-uses the last successful match.** `s//new/` substitutes against whatever PATTERN most recently matched in the enclosing scope. Convenient inside while `(m/.../g) { s//x/ }`, surprising when unrelated code has run in between.
- **Interpolation happens at runtime.** `s/$foo/$bar/` recompiles the pattern every call if `$foo` changes. Use /o to compile once and freeze the captured value - or precompile with `qr`, which is the modern alternative.
- **/e parses the replacement at compile time** even though it runs at match time. Syntax errors surface when the script is loaded, not when the substitution fires. Each extra e wraps one more `eval` around the value.
- **/g with an empty match advances by one character** to avoid an infinite loop: `s/\b/-/g` inserts between every token, it does not hang at the first position.
- **!~ flips the return**, not the work. `$x !~ s/A/a/g` still modifies `$x`; it just returns true when no A was found.
- **Delimiter matching for brackets** uses the mirrored closer: `s{pat}{rep}`, `s<pat><rep>`. A delimiter other than a bracket must appear three times total: `s/pat/rep/`.
- **/c is a no-op on s///**. Accepted for parser symmetry with `m//`; using it produces a warning under `use warnings` and nothing more.

- **Assignment-as-target idiom.** (`$copy = $src`) =~ `s/x/y/g` populates `$copy` with the modified value and leaves `$src` alone. Modern code prefers `my $copy = $src =~ s/x/y/gr`.
- **Tied or overloaded targets.** Without `/r`, `STORE` is invoked after the substitution to write the new value back; the fetched value is copied into a plain buffer before the pattern is applied. With `/r` the result is always a plain string - tie / overloading is not preserved in the return.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `m//` - the matching operator; shares match-side flag semantics and return-vs-context rules
- `tr///` - character-by-character transliteration; much faster when the job is literal character translation, not a regex
- `qr//` - precompile a pattern once and interpolate it into many `s///` or `m//` calls; the modern alternative to `/o`
- `split` - when the goal is to break a string on a pattern rather than replace matches, reach for `split` instead
- *Substitution chapter*
 - the chapter dedicated to `s///`; covers `/e`, `/r`, `/g`, `\K`, and zero-length-match handling
- *Regular expressions guide*
 - the full pattern-language reference; every match-side flag on `s///` is defined there
- `$&` - the matched text, available inside the REPLACEMENT of every `s///`

I/O

say

Print a list of values followed by a newline.

`say` is `print` with the output record separator forced to `"\n"` for that one call. It stringifies each item in `LIST`, joins them with `$,` (output field separator, usually empty), writes them to `FILEHANDLE`, and appends a literal `"\n"` - **not** the value of `$\`. If `FILEHANDLE` is omitted, output goes to the currently selected handle (see `select`), which defaults to `STDOUT`. If `LIST` is omitted, `$_` is printed.

`say` is a feature-gated built-in. It is available only when the "say" feature is enabled - automatic under use `v5.10` or higher, or explicit with use `feature 'say'` - or when called as `CORE::say`.

Synopsis

```
say LIST
say FILEHANDLE LIST
say {EXPR} LIST
say
```

What you get back

1 on success, `undef` on failure (with `$!` set). As with `print`, the return value is rarely checked for STDOUT or STDERR. Check it when writing to a file, pipe, or socket where a write failure is a recoverable condition:

```
say $fh $record
    or die "write to $path failed: $!";
```

Default filehandle, default separators

`say` reads the same global state as `print`, with one deliberate exception:

- The **currently selected filehandle**, changed with `select`. `say LIST` writes there when no filehandle is given. Defaults to STDOUT.
- `$,` - **output field separator**, inserted between list items. Defaults to the empty string, so `say 1, 2, 3` produces `"123\n"`, not `"1 2 3\n"`.
- `$\` - **output record separator**. `say` ignores `$\` entirely and appends a literal `"\n"` instead. Setting `$\` has no effect on `say`.

That last point is the whole reason `say` exists: one record, one newline, no fiddling with `$\`.

Examples

Basic write to STDOUT:

```
use feature 'say';
say "hello, world";                # "hello, world\n"
```

Write to a filehandle. Same indirect-object form as `print` - **no comma** between the filehandle and the list:

```
open my $fh, ">", "out.txt" or die $!;
say $fh "line 1";
say $fh "line 2";
close $fh;
```

Write to a filehandle held in an expression. Any filehandle more complex than a bareword or a plain scalar needs the `{EXPR}` block form:

```
my @logs = (\*STDOUT, \*STDERR);
say { $logs[$level] } "message";
```

say joins list items with \$, and then appends "\n" regardless of \$\\:

```
local $, = ", ";
local $\\ = ".\\n";           # ignored by say
say "apples", "pears", "plums"; # "apples, pears, plums\\n"
```

No arguments - print \$_ plus a newline. The filehandle must be a bareword in this form; say \$fh; would parse \$fh as the content, not as the target:

```
for (qw(alpha beta gamma)) {
    say;                               # "alpha\\n", "beta\\n", "gamma\\n"
}
```

List context applies to every argument:

```
my @arr = (10, 20, 30);
say @arr;                               # "102030\\n"
say scalar @arr;                        # "3\\n"
```

Edge cases

- **Feature gate:** without use feature 'say' or use v5.10, say is parsed as an ordinary subroutine call and will fail at compile time if no such sub is in scope. Use CORE::say(...) to call it unconditionally.
- **No arguments with an explicit filehandle:** say FH; prints \$_ to FH, but only for **bareword** filehandles. say \$fh; parses \$fh as the content, not the target - write say \$fh \$_; if you mean that. This matches `print`.
- **\\ is ignored, not overridden:** say does not set or localise \$. Other code that reads \$ (including nested `print` calls) sees whatever value was already there.
- **Filehandle-or-first-arg ambiguity:** same rule as `print`. A bareword or a simple scalar immediately followed by a term (no comma) is taken as the filehandle. An expression returning a filehandle needs the block form: say { \$handles[0] } "x";.
- **say (...) parses as a function call.** say (2+3)*4 prints 5 then discards *4. Parenthesise the whole argument list, or prefix with +:

```
say((2+3)*4);           # "20\\n"
say +(2+3)*4;           # "20\\n"
```

- **Closed filehandle:** returns `undef` and sets \$! to "Bad file descriptor". Under use warnings a say() on closed filehandle warning is emitted.
- **Encoding layers:** say writes through the handle's PerlIO stack exactly like `print`. A handle with a :utf8 or :encoding(...) layer encodes on the way out; the appended newline is a single "\n" byte and is encoded by the same layer. A wide-character string written to a bytes-mode handle triggers a Wide character in say warning.
- **Autoflush:** say buffers through PerlIO like `print`. Set \$| on the selected handle for line-buffered output.

- **SIGPIPE**: writing to a closed pipe or socket raises SIGPIPE. The default action terminates the process; install a handler to convert it into a normal write failure.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `print` - same targeting and separator rules, but appends `$\` (empty by default) instead of a forced newline
- `printf` - formatted output; does not append a newline, you put one in the format string
- `select` - change which filehandle the no-argument form of `say` writes to
- `$,` - output field separator inserted between list items; `say` honours it
- `$\` - output record separator; `say` **ignores it** and appends `"\n"` instead
- `$|` - autoflush flag for the currently selected handle

Misc

scalar

Force `EXPR` to be evaluated in scalar context and return its value.

`scalar` is not really an operator in its own right - it is a context-coercion marker the parser recognises in front of a unary expression. Perl evaluates the expression in scalar context and yields whatever that expression produces there. The function exists because most positions in a Perl program already impose a context (list or scalar) on their operands, and occasionally you need to override list context with scalar context explicitly.

Synopsis

```
scalar EXPR
scalar @array
scalar %hash
scalar localtime
```

What you get back

The value of `EXPR` as evaluated in scalar context. The exact shape depends entirely on what `EXPR` is:

- For an array, the element count.
- For a hash, a true value (non-empty) or `0` (empty); in older Perls this was a bucket-usage string, but from 5.26 onward it is the key count.

- For a list-context-sensitive function, whatever that function returns when called in scalar context (for example, `localtime` returns its ctime-style string instead of the nine-element list).
- For an already-scalar expression, the same value - scalar is a no-op there.

Global state it touches

None. `scalar` is a pure context cast; it does not read or write any special variables of its own. The *expression* you pass in may, of course, touch whatever globals it normally touches.

Examples

Count the elements of an array inside a list-context position:

```
my @items = (10, 20, 30);
my @counts = (scalar @items);      # (3)
print scalar @items, "\n";        # 3
```

Without `scalar`, `@items` would expand to its elements and `@counts` would be `(10, 20, 30)`.

Print the human-readable date from `localtime`. In list context `localtime` returns nine fields; in scalar context it returns the familiar ctime string:

```
print localtime, "\n";            # nine numbers run together
print scalar localtime, "\n";     # "Thu Apr 23 14:02:11 2026"
```

The same trap appears everywhere a builtin is list-context-sensitive:

```
my @parts = localtime;           # (sec, min, hour, mday, ...)
my $now   = scalar localtime;    # ctime string
```

Test whether a hash is non-empty without allocating its key list:

```
if (scalar %config) {
    # %config has at least one key
}
```

`if (%config)` already imposes scalar (boolean) context, so the `scalar` here is redundant but explicit; writers sometimes include it to make intent visible to readers.

Build a list of counts from several arrays in one expression:

```
my @counts = (scalar @a, scalar @b, scalar @c);
```

Coerce a function call that would otherwise splat into a surrounding list:

```
my @row = ($id, scalar get_tags($id), $mtime);
```

Without `scalar`, `get_tags` would run in list context and its returned list would be flattened into `@row` alongside `$id` and `$mtime`.

Edge cases

- **No list-context counterpart.** There is no `list` `EXPR` operator; list context is the default in almost every position where you might want to impose it. If you genuinely need to coerce a scalar expression into list context, the idiom is `@{ [ EXPR ] }`, though in most cases plain `(EXPR)` is enough.
- **Parenthesised list trap.** `scalar` is a unary operator. Passing it a parenthesised comma expression does **not** evaluate that expression as a list; it evaluates it as a *scalar comma operator*, which runs every sub-expression in void context except the last and returns the last in scalar context:

```
print uc(scalar(foo(), $bar)), $baz;
# equivalent to:
foo();                                # void context
print(uc($bar), $baz);                # $bar in scalar context
```

If you wanted `foo()`'s return values included, `scalar (list)` is the wrong tool.

- **Hash in scalar context.** From Perl 5.26 onward, `scalar %h` returns the number of keys, the same as `scalar keys %h`. On older Perls it returned a string like "4/8" reporting bucket usage; pperl targets 5.42 and so always returns the key count.
- **Subroutine prototypes.** A subroutine declared with a scalar prototype (`($)`) already imposes scalar context on its argument, so `scalar` is redundant in front of the call. A `(@)` or `(\@)` prototype does not, so `scalar` still matters:

```
sub count ($) { $_[0] }
count @items;                                # @items in scalar context:
↪ 3
```

- **Redundant on already-scalar expressions.** `scalar $x`, `scalar 42`, `scalar "text"` are no-ops. Perl does not warn about them; they are occasionally used for emphasis but add nothing.
- **Not a function call.** Because `scalar` is a parser-level context marker, you cannot take `\&scalar`, cannot override it with a `sub scalar { ... }`, and cannot pass it as a code reference.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- [wantarray](#) - inspect the context you were called in, rather than impose one on a sub-expression
- [defined](#) - similarly a unary operator applied to an expression; useful alongside `scalar` when testing function results
- [localtime](#) - canonical example of a builtin whose return shape depends on context

- `reverse` - also context-sensitive; scalar `reverse $s` reverses characters, `reverse @a` reverses a list
- `keys` - scalar `keys %h` is the long form of scalar `%h` with the same result on 5.26+

I/O

seek

Reposition a filehandle for random-access reads or writes.

`seek` moves `FILEHANDLE`'s read/write pointer to a new byte offset, mirroring the C `fseek(3)` call. After a successful `seek`, the next `read`, `readline`, or `print` on the handle starts from the new position. `POSITION` is a signed byte offset; `WHENCE` selects the anchor it is measured from.

Synopsis

```
seek FILEHANDLE, POSITION, WHENCE
seek $fh, 0, 0           # rewind to start
seek $fh, 0, 1           # no-op: clear EOF, keep position
seek $fh, -1024, 2       # 1024 bytes before EOF
```

What you get back

1 on success, a false value on failure (with `$!` set). Always check the return value - seeking past a short file, on an unseekable handle (pipe, socket, TTY), or after a write error all fail here rather than at the next read.

```
seek $fh, $offset, 0
or die "seek to $offset failed: $!";
```

WHENCE values

`WHENCE` is an integer with three meaningful values. Use the symbolic constants from `Fcntl` for readability:

- 0 / `SEEK_SET` - `POSITION` is measured from the **start of the file**. `POSITION` must be non-negative.
- 1 / `SEEK_CUR` - `POSITION` is added to the **current position**. Negative values move backward, positive values forward. `seek $fh, 0, 1` is the canonical “move nowhere, but clear EOF” idiom.
- 2 / `SEEK_END` - `POSITION` is added to the **end-of-file** offset. `POSITION` is typically zero or negative.

```
use Fcntl qw(SEEK_SET SEEK_CUR SEEK_END);

seek $fh, 0,      SEEK_SET;  # rewind
```

(continues on next page)

(continued from previous page)

```
seek $fh, 0, SEEK_END; # go to EOF (e.g. to append)
seek $fh, -$n, SEEK_CUR; # $n bytes back from here
```

Bytes, not characters

Even when the handle has a character-oriented layer such as `:encoding(UTF-8)`, `seek`, `tell`, and the `sysseek` family operate on **byte offsets**. A seek to a byte offset that lands in the middle of a multi-byte sequence will produce decoding errors or replacement characters on the next read.

If you need to position by characters, read forward from a known byte boundary rather than trying to translate character counts to byte offsets. `tell` values are safe to feed back into `seek` because they were produced at byte boundaries the I/O layer already crossed cleanly.

Global state it touches

- `$!` - set to the system error message on failure.
- The filehandle's PerlIO buffer is discarded on success, so any data read ahead by buffering is dropped and the next read pulls fresh bytes from the underlying file.
- The end-of-file flag on `FILEHANDLE` is cleared on success, even when `POSITION` and `WHENCE` leave the position unchanged.

Examples

Rewind to the start of a file before re-reading it:

```
seek $fh, 0, 0 or die "rewind failed: $!";
while (my $line = <$fh>) { ... }
```

Append by positioning at end-of-file, then writing. For append-only use, prefer opening with `>>` - this form is for handles already open for read-write:

```
seek $fh, 0, 2 or die $!; # SEEK_END
print $fh "appended line\n";
```

Read a fixed-width record by index, where each record is 128 bytes:

```
my $record = 42;
seek $fh, $record * 128, 0 or die $!;
read $fh, my $buf, 128;
```

Emulate `tail -f` - the `seek $fh, 0, 1` resets the EOF condition so the next `readline` retries the file for new data:

```
while (1) {
    while (my $line = <$fh>) { print $line }
    sleep 1;
}
```

(continues on next page)

(continued from previous page)

```
    seek $fh, 0, 1;                # clear EOF, keep position
}
```

Save a position with `tell`, read ahead, then restore:

```
my $mark = tell $fh;
my $peek = <$fh>;
seek $fh, $mark, 0 or die $!;      # back to where we were
```

Edge cases

- **Unseekable handles:** pipes, sockets, TTYs, and most special devices return false and set `$!` to `ESPIPE` (Illegal seek). Check the return value; do not assume every filehandle is seekable.
- **Mixing reads and writes on the same handle** on a file opened `+<` or `+>` requires a seek (or `tell`, or an explicit flush) when switching direction. A `WHENCE` of 1 with `POSITION` of 0 is the usual no-op separator:

```
seek $fh, 0, 1;                    # allowed to switch read <->
↪write
```

- **Seeking past EOF** on a file opened for writing is legal and creates a sparse hole up to `POSITION` on filesystems that support holes; the next write fills part of the hole and the bytes in between read back as `"\0"`.
- **`sysread` / `syswrite` users:** do **not** mix seek with `sysread` or `syswrite`. seek operates on the PerlIO buffer layer; the unbuffered sys-family bypasses it, so their effective positions drift apart. Use `sysseek` for handles you access through the sys-family.
- **Character offsets via arithmetic:** multiplying a character count by an assumed encoding width is wrong for UTF-8 and other variable-width encodings. Use byte offsets obtained from `tell`, or read forward from a known boundary.
- **Directory handles:** seek does not work on directory handles. Use `seekdir` with a position from `telldir`.
- **Closed filehandle:** returns a false value and sets `$!`; under use warnings a `seek() on closed filehandle` warning is emitted.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `tell` - read the current byte offset; the value round-trips safely through seek with `WHENCE 0`
- `sysseek` - unbuffered seek for handles accessed via `sysread` / `syswrite`; use this instead of seek when bypassing PerlIO

- `read` - buffered fixed-length read; the natural partner for positioning by byte offset
- `readline` - line-oriented read; seek `$fh, 0, 1` before a retry is the tail -f idiom
- `eof` - test for end-of-file; seek clears the EOF flag a prior read may have set
- `Fcntl` - source of the `SEEK_SET`, `SEEK_CUR`, `SEEK_END` constants

I/O

seekdir

Restore a directory handle to a position previously captured by `telldir`.

`seekdir` moves the read cursor of `DIRHANDLE` to `POS`, so that the next `readdir` call returns the entry that would have been returned when `POS` was first obtained from `telldir` on the same handle. It is the directory counterpart of `seek` on a file handle - with the crucial restriction that `POS` is not a byte offset you can compute, it is an opaque cookie you must have taken from `telldir` earlier.

Synopsis

```
seekdir DIRHANDLE, POS
```

What you get back

No meaningful value. `seekdir` returns what the underlying `seekdir(3)` returns, which on Linux is nothing useful - do **not** branch on the return value. To detect trouble, check `$_` after the call, or verify that the handle is still open before calling.

Typical use

You are walking a directory, notice an entry you want to revisit later, and remember where you were so you can return to it after reading on:

```
opendir my $dh, $path or die "opendir $path: $_";

my %bookmarks;
while (my $entry = readdir $dh) {
    $bookmarks{$entry} = telldir $dh # position AFTER reading
    ↪$entry
    if $entry =~ /\.conf$/;
}

# later: re-read the entry that follows the first .conf file
seekdir $dh, $bookmarks{(sort keys %bookmarks)[0]};
my $next = readdir $dh;

closedir $dh;
```

Returning to the start is the common special case - a `seekdir $dh, 0` call is equivalent to `rewinddir` on conforming systems, but `rewinddir` is the idiomatic spelling for that intent:

```
seekdir $dh, 0;      # works, but say what you mean:
rewinddir $dh;       # clearer
```

Edge cases

- **POS must come from `telldir` on the same handle.** Passing an integer you made up, a value from a different handle, or a stale value from a handle that has since been closed and re-opened is undefined behaviour. The kernel may accept it silently and hand you garbage on the next `readdir`, skip entries, or return duplicates.
- **Directory compaction between `telldir` and `seekdir`:** if the filesystem has compacted or reorganised the directory between the two calls (files created, deleted, or renamed in the same directory by any process), the POS cookie may no longer refer to the same entry - or to any entry at all. This is a property of the underlying `seekdir(3)` and is not a pperl limitation. For stable iteration over a changing directory, snapshot the list with `readdir` in list context and work off the copy.
- **Unopened or closed handle:** sets `$!`. Under `use warnings` you get a `seekdir()` attempted on invalid dirhandle warning. Nothing in the return value tells you this happened - check `$!` or use `defined fileno($dh)` / explicit bookkeeping if you need to guard.
- **Not a byte offset:** POS is an opaque token. Arithmetic on it (`seekdir $dh, $pos + 1`) is meaningless; the only safe values are `0` (start of directory, and even then prefer `rewinddir`) and values returned by `telldir`.
- **Entry order is unspecified:** `seekdir` replays whatever order the kernel originally returned. That order is not sorted, not alphabetical, and not guaranteed to match a fresh `opendir` of the same path later. If your code sorts, sort explicitly each time.
- **SEEK_CUR-style relative seeks do not exist** for directory handles. There is no equivalent of `seek`'s whence argument.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `telldir` - produce the POS cookie that `seekdir` consumes; the two are only useful as a pair
- `readdir` - read the next entry; resumes from the position `seekdir` just set
- `rewinddir` - the idiomatic way to return to the start of a directory; prefer it over `seekdir $dh, 0`
- `opendir` - open the directory and obtain the handle `seekdir` operates on

- `seek` - the file-handle analogue, but with a true byte offset and a whence argument; directory handles have neither

I/O · Filehandles, files, directories

select

Either set the default output filehandle, or call the `select(2)` syscall for I/O multiplexing. Same name, two unrelated jobs - disambiguated by argument count.

The one-argument form (or no-argument form) is about where `print`, `printf`, `say`, and `write` send their output when you don't name a filehandle. The four-argument form is a direct wrapper around the BSD `select(2)` syscall for waiting on sets of file descriptors.

Synopsis

```
select FILEHANDLE          # set default output handle, return ↵
↵old                        # return current default output ↵
select                     # return current default output ↵
↵handle
select RBITS, WBITS, EBITS, TIMEOUT # select(2) syscall
```

What you get back

select FILEHANDLE returns the *previously* selected filehandle (as a typeglob), so the idiomatic save/restore pattern works:

```
my $old = select $fh;
$| = 1;
select $old;
```

select (no arguments) returns the currently selected filehandle. At program start this is STDOUT.

select RBITS, WBITS, EBITS, TIMEOUT returns different things in different contexts:

- In scalar context: the number of filehandles ready, or -1 on error with `$!` set.
- In list context: (`$nfound`, `$timeleft`) - the same `$nfound` plus the remaining time from `TIMEOUT`. Many kernels do not update `$timeleft`; on those systems it is always equal to the input `TIMEOUT`.

On timeout expiry `$nfound` is 0. The three bit masks (`RBITS`, `WBITS`, `EBITS`) are updated in place to indicate which descriptors are ready - pass copies if you want to preserve the original masks.

Global state it touches

The one-argument form changes one piece of interpreter-global state: the currently selected output filehandle. While that handle is selected, these variables refer to **it**, not to STDOUT:

- `$|` - autoflush flag

- \$, - output field separator
- \$\ - output record separator
- \$#, \$%, \$=, \$-, \$~, \$^ - format-related variables

That is why `my $old = select($fh); $| = 1; select($old);` works: `$|` is a *property of the selected handle*, not a global scalar.

The four-argument form touches no per-interpreter state. It may leave `$!` set on error.

Examples

Enable autoflush on a handle without disturbing the default:

```
my $old = select $fh; $| = 1; select $old;
```

Redirect print temporarily and restore on block exit:

```
{
    my $old = select $log_fh;
    print "entry: $msg\n";      # goes to $log_fh
    select $old;
}
```

Sleep for 250 milliseconds using the four-argument form:

```
select undef, undef, undef, 0.25;
```

Wait up to 5 seconds for STDIN to become readable:

```
my $rin = '';
vec($rin, fileno(STDIN), 1) = 1;
my $nfound = select my $rout = $rin, undef, undef, 5;
if ($nfound) {
    my $line = <STDIN>;
}
```

Build a read mask over multiple handles:

```
sub fhbits {
    my $bits = '';
    vec($bits, fileno($_), 1) = 1 for @_;
    return $bits;
}
my $rin = fhbits(\*STDIN, $sock);
my ($nfound, $timeleft) =
    select my $rout = $rin, undef, undef, 10;
```

Edge cases

- **Method-style alternative.** `$fh->autoflush(1)` sets the autoflush flag on `$fh` without changing the default handle. Prefer it when you only need to tweak one property of one handle - no global state involved.
- **Typglob localization is cleaner for scoped redirects.** `local *STDOUT = $fh` rebinds `STDOUT` itself for the dynamic scope; on die the original is restored automatically. `select` does **not** restore on die. The two are not equivalent: with `select` an explicit `print STDOUT ...` still reaches the *real* `STDOUT`; with the `local *STDOUT` form, even `print STDOUT ...` goes to the rebound handle.
- **Bit-mask form requires bytes, not a list of fileno integers.** A bit mask is a packed bit string built with `vec`; passing a list of integers or a reference silently produces wrong results (or an empty mask).
- **undef for a mask means “don’t wait on this set.”** Pass `undef` for masks you don’t care about. An empty string `''` technically also works but is less idiomatic.
- **Fractional timeouts work.** `TIMEOUT` is a float in seconds. `0.25` is 250 ms; `0` returns immediately (polling).
- **TIMEOUT of undef** means block indefinitely until at least one descriptor is ready or a signal arrives.
- **Masks are modified in place.** On return, each mask holds only the bits for descriptors that are ready. If you need the originals afterwards, pass copies - the idiom `my $rout = $rin` on the call line does exactly that.
- **Do not mix buffered I/O with the four-argument form.** `select` reports readiness at the kernel level; `readline`, `read`, and the diamond operator `<>` read through PerlIO buffers that may already hold unread data the kernel doesn’t know about. Use `sysread` for data once `select` says a descriptor is ready.
- **Spurious readiness on sockets.** On some Unixes `select(2)` may report a socket as “ready for reading” when a following read would still block. Set `O_NONBLOCK` on the socket if this matters.
- **Restart-after-signal behaviour is system-dependent.** A signal (e.g. `SIGALRM`) may cause `select` to return early with `#! == EINTR`, or the kernel may restart the call transparently. Portable code checks for `EINTR` explicitly.
- **Argument count disambiguates the two forms.** One argument (or none) is the default-handle form; exactly four is the syscall form. Other counts are a syntax error.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `print` - reads the selected handle when no filehandle is given; `select` is how you change which handle that is
- `printf` - same default-handle rule as `print`

- `write` - format-based output; the top-of-form and page-length variables follow the selected handle, not STDOUT
- `fileno` - maps a filehandle to the integer descriptor number you need when building a mask
- `vec` - packs descriptor numbers into the bit string the four-argument form expects
- `sysread` - the unbuffered read you use after the four-argument `select` says a descriptor is ready
- `IO::Select` - object wrapper over the four-argument form that hides the bit-mask arithmetic

SysV IPC

`semctl`

Perform a control operation on a System V semaphore set.

`semctl` is the Perl-level binding for the `semctl(2)` system call. It reads or modifies the state of the semaphore set identified by ID - getting or setting a single semaphore's value, fetching or installing the full value vector, reading the associated `semid_ds` metadata, or removing the set entirely. `CMD` selects the operation; `SEMNUM` selects which semaphore within the set (ignored for set-wide commands like `IPC_STAT`, `IPC_RMID`, `GETALL`, `SETALL`); `ARG` is the command's input or output buffer.

`semctl` is a thin wrapper - the semantics, the command constants, and the shape of `ARG` all come straight from the C-level call. You will almost always want:

```
use IPC::SysV;
```

first, to pull in the `IPC_STAT`, `IPC_RMID`, `GETVAL`, `SETVAL`, `GETALL`, `SETALL`, `GETPID`, `GETNCNT`, `GETZCNT` constants.

Synopsis

```
semctl $id, $semnum, $cmd, $arg
semctl $id, 0,          IPC_RMID, 0           # destroy set
semctl $id, $n,         SETVAL,  $value       # set one
semctl $id, 0,          SETALL,  $packed      # set all
```

What you get back

Return follows the same convention as `ioctl`:

- `undef` on error (with `$!` set).
- The string "`0 but true`" when the underlying call returned `0` - this string is false numerically (so it prints `0` in numeric context) but true in boolean context, letting you distinguish success-with-zero from failure in one test.
- The actual integer return value otherwise.

The idiomatic test is therefore:


```
my $rc = semctl($id, $n, GETVAL, 0);
defined $rc or die "semctl: $!";
```

Check defined, not truth - a legitimately zero semaphore value would fail a plain if (\$rc).

For CMD values that read data into ARG (IPC_STAT, GETALL), ARG must be a **modifiable lvalue** (a plain variable, not a literal or expression); on success the buffer is overwritten in place with the returned bytes and you unpack it yourself.

Global state it touches

Sets \$! on failure. No other interpreter globals involved.

Examples

Read a single semaphore's current value:

```
use IPC::SysV qw(GETVAL);
my $value = semctl($id, $semnum, GETVAL, 0);
defined $value or die "GETVAL: $!";
```

Set a single semaphore:

```
use IPC::SysV qw(SETVAL);
semctl($id, $semnum, SETVAL, 1)
    or die "SETVAL: $!";
```

Read all semaphore values at once. The buffer has to be pre-allocated to the right size - nsems native shorts - before the call. pack with an unused count is the usual way:

```
use IPC::SysV qw(GETALL);
my $nsems = 4;
my $buf = pack("s!*", (0) x $nsems);      # allocate room
semctl($id, 0, GETALL, $buf) or die $!;
my @vals = unpack("s!*", $buf);
```

Install a full set of values:

```
use IPC::SysV qw(SETALL);
my $buf = pack("s!*", 1, 0, 0, 1);
semctl($id, 0, SETALL, $buf)
    or die "SETALL: $!";
```

Fetch the semid_ds structure for the set. ARG is written in place with the packed struct; its layout is platform-specific, so unpack with the help of IPC::Semaphore rather than by hand unless you have to:

```
use IPC::SysV qw(IPC_STAT);
my $ds = "";
semctl($id, 0, IPC_STAT, $ds)
    or die "IPC_STAT: $!";
```

Destroy the semaphore set. SEMNUM and ARG are unused; pass 0:

```
use IPC::SysV qw(IPC_RMID);
semctl($id, 0, IPC_RMID, 0)
    or die "IPC_RMID: $!";
```

Edge cases

- **ARG must be a real variable for read commands.** GETALL and IPC_STAT write into ARG; passing a literal (`semctl($id, 0, GETALL, "")`) or a read-only expression is a runtime error. The variable's existing contents are overwritten, not appended to.
- **ARG must be pre-sized.** The kernel writes exactly `nsems * sizeof(short)` bytes for GETALL, or `sizeof(struct semid_ds)` for IPC_STAT. Passing a shorter buffer is undefined behaviour in the underlying C call; `pack("s!*", (0) x $nsems)` is the standard idiom for GETALL.
- **Native short, not portable short.** Semaphore values are C short (signed, typically 16-bit). Use `s!` in `pack / unpack` - the `!` picks the platform-native type. Plain `s` forces little-endian 16-bit and will produce wrong results on big-endian or LP64-short systems.
- **"0 but true" returns.** GETVAL on a semaphore whose value is 0, GETPID/GETNCNT/GETZCNT returning zero, and other zero-valued successes all return the literal string "0 but true". Arithmetic contexts see 0; boolean contexts see true. Use `defined` to check for failure.
- **SEMNUM is ignored for set-wide commands.** IPC_STAT, IPC_RMID, SETALL, GETALL act on the whole set. Pass 0 by convention; passing any other value is harmless but obscures intent.
- **After IPC_RMID the id is invalid.** Every subsequent `semctl` or `semop` on `$id` fails with EINVAL / EIDRM. Do not cache the id past removal.
- **The set is kernel-global and persists across process exits.** A set created by a crashed script remains until someone calls IPC_RMID (or the system reboots). Use `ipcs -s / ipcrm -s` from the shell to inspect or clean up leaked sets.
- **Permissions come from semget.** The `IPC_CREAT | mode` passed at creation time controls who may issue which `semctl` commands. EACCES from `semctl` means the mode bits disallow the requested operation for the current effective uid.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `semget` - create or look up the semaphore set whose id you then pass to `semctl`
- `semop` - the actual wait / signal operations; `semctl` is for control and inspection, `semop` does the work

- `msgctl` - the analogous control call for System V message queues; same return-value convention
- `shmctl` - the analogous control call for System V shared memory; same return-value convention
- `pack` - how to build the ARG buffer with `s!` for semaphore values and the `semid_ds` struct
- `IPC::Semaphore` - object wrapper that hides `semctl` behind named methods; reach for it when a script does more than one or two calls

SysV IPC

semget

Create or look up a System V semaphore set and return its identifier.

`semget` is the entry point into the SysV semaphore API. It wraps the `semget(2)` system call: given a numeric `KEY` that names the set, the number of semaphores `NSEMS` in the set, and a `FLAGS` word combining permission bits with creation flags, the kernel either returns the id of a matching existing set or, if `IPC_CREAT` is in `FLAGS`, creates a new set and returns its id. The returned id is the handle every other semaphore call (`semop`, `semctl`) expects.

Synopsis

```
semget KEY, NSEMS, FLAGS
```

What you get back

The semaphore set identifier (a non-negative integer) on success, or `undef` on failure with `$!` set to the `errno` from the underlying `semget(2)` call. The id is **not** a file descriptor - it is a kernel-wide SysV IPC handle that outlives your process and persists until explicitly removed with `semctl($id, 0, IPC_RMID, 0)`.

```
use IPC::SysV qw(IPC_CREAT IPC_PRIVATE);

my $semid = semget(IPC_PRIVATE, 3, IPC_CREAT | 0600)
    // die "semget: $!";
```

Global state it touches

- `$!` - set to the `errno` value from `semget(2)` on failure (`EACCES`, `EEXIST`, `ENOENT`, `ENOSPC`, `EINVAL`, `ENOMEM`).

The KEY argument

`KEY` is an integer that names the semaphore set system-wide. Three conventions matter:

- **`IPC_PRIVATE`** (imported from `IPC::SysV`) - the kernel allocates a fresh key unused by any other set. The resulting id is known only to this process and anyone it passes

the id to, typically children after `fork`. Use this when the set is an implementation detail of one process tree.

- **A fixed integer literal** - every cooperating process hard-codes the same number. Fragile across unrelated programs that might pick the same integer.
- **A key derived from `ftok(3)`** - `IPC::SysV::ftok($path, $proj_id)` turns a filesystem path plus a project byte into a numeric key. The standard recipe for unrelated processes that need to agree on a key without hard-coding a magic number.

The FLAGS argument

FLAGS combines **permission bits** (low 9 bits, like a `chmod` mode: `0600`, `0660`, `0666`) with **creation flags** from `IPC::SysV`:

- `IPC_CREAT` - create the set if it does not already exist. Without this flag, `semget` only looks up an existing set with the given key.
- `IPC_CREAT | IPC_EXCL` - create the set, but fail with `EEXIST` if one already exists for KEY. The SysV analogue of `O_CREAT | O_EXCL` in `open`.

Permission bits are checked by the kernel on every subsequent `semop` and `semctl` call against the invoking process's uid/gid.

Examples

Create a private set of three semaphores usable by this process and its children:

```
use IPC::SysV qw(IPC_PRIVATE IPC_CREAT);
my $semid = semget(IPC_PRIVATE, 3, IPC_CREAT | 0600)
    // die "semget: $!";
```

Look up an existing, named set without creating it. `NSEMS` must match the size the set was created with, or can be `0` if you do not care:

```
use IPC::SysV qw(ftok);
my $key = ftok("/var/run/myapp.pid", ord('M'));
my $semid = semget($key, 0, 0) // die "no set for key $key: $!";
```

Create-or-open idiom - try exclusive create, fall back to lookup. The first process through initialises the set; later ones just attach:

```
use IPC::SysV qw(ftok IPC_CREAT IPC_EXCL);

my $key = ftok("/var/run/myapp.pid", ord('M'));
my $semid = semget($key, 4, IPC_CREAT | IPC_EXCL | 0660);
if (defined $semid) {
    # We created it - initialise every semaphore to 1.
    semctl($semid, 0, SETALL, pack("s!*", (1) x 4))
        or die "SETALL: $!";
} else {
    $semid = semget($key, 4, 0) // die "semget lookup: $!";
}
```

Remove a set when you are done with it. SysV sets leak across process exit - nothing cleans them up for you:

```
use IPC::SysV qw(IPC_RMID);
semctl($semid, 0, IPC_RMID, 0) or warn "semctl RMID: $!";
```

Edge cases

- **NSEMS on lookup:** when looking up an existing set (no `IPC_CREAT`), `NSEMS` of 0 means “any size”; a positive value must be less than or equal to the set’s actual size or `semget` fails with `EINVAL`.
- **IPC_PRIVATE still obeys FLAGS:** permission bits apply. A set created with `0400` is read-only even to the creating process for `semop` calls that modify it.
- **Kernel limits:** `semget` can fail with `ENOSPC` when the system-wide semaphore-set limit is reached, or `EINVAL` when `NSEMS` exceeds the per-set maximum. See `ipcs -sl` for the current limits on Linux.
- **Leaked sets:** a set persists until `IPC_RMID` or reboot. Long-running daemons that forget to remove their sets accumulate them; `ipcs -s` lists the survivors, `ipcrm -s $id` removes one.
- **Key collisions:** two unrelated programs picking the same hard-coded KEY will see each other’s set - hence the `ftok(3)` idiom.
- **Not a filehandle:** `$semid` is an integer, not a handle. Do not pass it to `close`, `fcntl`, or any filehandle operation.
- **Forks inherit ids, not ownership:** a child sees the same `$semid` value as the parent and can use it, but the set itself is owned by the uid that created it, not by any particular process.
- **Numeric context:** the id is a plain integer, safe to print, compare, and store in a file for another process to pick up.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `semop` - perform wait/signal operations on the set returned by `semget`
- `semctl` - control operations on the set: initialise values (`SETALL`, `SETVAL`), read them (`GETALL`, `GETVAL`), and remove the set (`IPC_RMID`)
- `msgget` - the message-queue counterpart, same KEY, FLAGS conventions
- `shmget` - the shared-memory counterpart, same KEY, FLAGS conventions
- `IPC::SysV` - imports `IPC_CREAT`, `IPC_EXCL`, `IPC_PRIVATE`, `IPC_RMID`, `SETALL`, `GETALL`, and the `ftok` helper
- `IPC::Semaphore` - object wrapper around the raw `sem*` built-ins when you would rather not pack structs by hand

SysV IPC

semop

Perform one or more System V semaphore operations atomically.

`semop` is the workhorse of SysV semaphore use: it takes a semaphore set identifier (from `semget`) and a packed list of operations, then asks the kernel to apply them all or none. Each operation names a single semaphore within the set, an amount to add to it (typically -1 to wait, +1 to signal), and a flag word. The kernel blocks the caller until every requested operation can proceed, then commits them as one unit. This is what makes `semop` usable as a mutual-exclusion or counting-resource primitive between unrelated processes.

Synopsis

```
semop $semid, $opstring
```

What you get back

True on success, false on error with `$!` set. The truth value is the only result - `semop` communicates outcome by the side effect on the semaphore set, not through its return. Check it on every call:

```
semop($semid, $ops)
    or die "semop failed: $!";
```

A false return means no operation in `OPSTRING` was applied; SysV semantics guarantee all-or-nothing.

Global state it touches

Sets `$!` on failure, with the usual `errno` values from `semop(2)`: `EAGAIN` when `IPC_NOWAIT` was set and the operation would block, `EIDRM` when the semaphore set was removed while waiting, `EINTR` when a signal interrupted the wait, `EINVAL` for a bad semaphore number or identifier, `EACCES` for a permission mismatch against the set's mode.

The operation structure

`OPSTRING` is a concatenation of fixed-width records. Each record is three native-width short integers and is produced with `pack`:

```
pack("s!3", $semnum, $semop, $semflag)
```

- `$semnum` - the index of the target semaphore within the set (0 through `nsems-1`, where `nsems` is the size passed to `semget`).
- `$semop` - the amount to add to the semaphore's value. Negative values wait until the semaphore is at least `|$semop|` before subtracting; positive values add unconditionally and wake waiters; zero waits until the semaphore reaches zero.

- `$semflag` - bitwise OR of `IPC_NOWAIT` (fail with `EAGAIN` instead of blocking) and `SEM_UNDO` (have the kernel reverse the operation if the process exits without undoing it - essential for lock-like uses where a crash must not leave the semaphore held).

The length of `OPSTRING` implies the number of operations: the kernel processes `length($opstring) / sizeof(struct sembuf)` records. Passing two records packed end-to-end asks for two operations applied atomically.

The `s!` in the template is the **native** short, which matches the `sembuf` layout on Linux. Plain `s` is network-order signed 16-bit and will silently mis-pack on any platform where `sizeof(short) != 2` or where endianness differs from network order.

Examples

Wait on (decrement by 1) semaphore 0 of the set in `$semid`:

```
my $op = pack("s!3", 0, -1, 0);
semop($semid, $op)
    or die "wait on semaphore failed: $!";
```

Signal (increment by 1) the same semaphore:

```
my $op = pack("s!3", 0, 1, 0);
semop($semid, $op)
    or die "signal failed: $!";
```

Try to acquire without blocking - fail fast if someone else holds it:

```
use POSIX qw(EAGAIN);
my $op = pack("s!3", 0, -1, IPC_NOWAIT);
unless (semop($semid, $op)) {
    die "semop: $!" unless $! == EAGAIN;
    # contended; caller decides what to do
}
```

Acquire semaphore 0 with automatic release on process exit - the pattern for crash-safe mutual exclusion:

```
my $op = pack("s!3", 0, -1, SEM_UNDO);
semop($semid, $op) or die "lock: $!";
# critical section
my $release = pack("s!3", 0, 1, SEM_UNDO);
semop($semid, $release) or die "unlock: $!";
```

Two operations applied atomically - wait for semaphore 0 and semaphore 1 together, releasing neither if either would block:

```
my $ops = pack("s!3", 0, -1, 0)
    . pack("s!3", 1, -1, 0);
semop($semid, $ops) or die "paired wait: $!";
```

Wait for semaphore 0 to reach zero - the “barrier” form, where `$semop` is zero rather than negative:


```
my $op = pack("s!3", 0, 0, 0);
semop($semid, $op) or die "barrier: $!";
```

Edge cases

- **Short OPSTRING:** the length must be an exact multiple of the sembuf record size. A truncated or padded string gives EINVAL. Always build the string from `pack` calls, never by hand.
- **Native width matters:** `s!3` not `s3`. A script written with plain `s` works on a 16-bit-short host by accident and breaks elsewhere.
- **Zero-length OPSTRING:** asks for zero operations. The kernel accepts this and returns success without side effects - rarely what you want. Guard the call with `return` unless `length $ops`.
- **Signal interruption:** a blocked `semop` returning with `$!` set to EINTR is not a bug - retry unless you installed a handler that wants to abort. Perl's default `$SIG{*}` handling does not automatically restart the syscall.
- **SEM_UNDO with matched pairs:** the kernel tracks the cumulative adjustment per semaphore per process. Acquiring twice with `SEM_UNDO` then releasing once leaves an outstanding -1 to be undone on exit. Pair every -1 with a matching +1 in normal flow; `SEM_UNDO` is insurance against crashes, not a substitute for releasing.
- **Removed set while waiting:** if another process calls `semctl` with `IPC_RMID`, every blocked `semop` returns false with `$!` set to EIDRM. The semaphore identifier is invalid from that point on.
- **Permissions:** `semop` checks alter-permission on the set. A process that can `semget` a read-only handle still cannot `semop` it.
- **Large operation counts:** the kernel caps the number of operations per call at `SEMOPM` (typically 32). Building an OPSTRING with more records fails with E2BIG.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `semget` - obtain the identifier you pass as the first argument; the two are always used together
- `semctl` - set or query semaphore values, and remove the set when you're done with it
- `pack` - build each sembuf record; `s!3` is the template you want
- `msgsnd` - the message-queue counterpart when you need to pass data rather than signal availability
- `shmget` - the shared-memory counterpart; semaphores are typically used to guard access to a SysV shared-memory segment

- `IPC::SysV` - exports the `IPC_NOWAIT`, `SEM_UNDO`, and `IPC_RMID` constants used in `$semflag` and in `semctl` commands
- `IPC::Semaphore` - object wrapper over `semget` / `semop` / `semctl` when a procedural interface feels too bare

Sockets

send

Send a message on a socket.

`send` writes the scalar `MSG` to `SOCKET` through the operating system's socket layer, not through `PerlIO`. On a connected socket (`SOCK_STREAM` after `connect`, or an accepted peer) the three-argument form issues a `send(2)` - use the four-argument form on an unconnected datagram socket (`SOCK_DGRAM`) to route each message to `TO`, which triggers `sendto(2)` under the hood. `FLAGS` is the bitmask the system call accepts (commonly `0`; see your platform's `send(2)` man page for `MSG_DONTWAIT`, `MSG_OOB`, `MSG_NOSIGNAL`, and friends).

Synopsis

```
send SOCKET, MSG, FLAGS, TO
send SOCKET, MSG, FLAGS
```

What you get back

The **number of bytes** accepted by the kernel, or `undef` on error with `$!` set. Always check the return value:

```
my $n = send $sock, $buf, 0
    or die "send failed: $!";
```

Two properties bite people:

- The count is **bytes**, not characters. `length $buf` on a wide string and the returned count will not agree.
- On `SOCK_STREAM` sockets the kernel may accept **fewer bytes than you gave it** - a short write. Loop until the whole buffer is drained, or use `syswrite` which has the same short-write semantics but works on any filehandle.

Global state it touches

- `$!` - set to the system error on failure.
- The socket's `PerlIO` layer stack is **not** consulted. Output layers (`:utf8`, `:encoding(...)`, `:crlf`) have no effect on what `send` writes. In particular, a `:utf8` layer causes `send` to **throw an exception** rather than silently encode - the socket layer and the `PerlIO` layer cannot both own byte framing.

Examples

Connected TCP - three-argument form:

```
use Socket;
socket my $sock, PF_INET, SOCK_STREAM, getprotobyname("tcp")
    or die "socket: $!";
connect $sock, sockaddr_in(80, inet_aton("example.com"))
    or die "connect: $!";
send $sock, "GET / HTTP/1.0\r\n\r\n", 0
    or die "send: $!";
```

Unconnected UDP - four-argument form with an explicit destination:

```
use Socket;
socket my $sock, PF_INET, SOCK_DGRAM, getprotobyname("udp")
    or die "socket: $!";
my $to = sockaddr_in(514, inet_aton("10.0.0.1"));
send $sock, "<14>hello syslog\n", 0, $to
    or die "send: $!";
```

Handle short writes on a stream socket:

```
my $offset = 0;
while ($offset < length $buf) {
    my $n = send $sock, substr($buf, $offset), 0;
    defined $n or die "send: $!";
    $offset += $n;
}
```

Send with a flag - MSG_DONTWAIT makes a single non-blocking attempt and returns undefined with \$! set to EAGAIN/EWOULDBLOCK if the kernel buffer is full:

```
use Socket qw(MSG_DONTWAIT);
my $n = send $sock, $buf, MSG_DONTWAIT;
if (!defined $n) {
    if ($!{EAGAIN} || $!{EWOULDBLOCK}) {
        # back off, retry later
    } else {
        die "send: $!";
    }
}
```

Edge cases

- **Wide characters:** passing a string with codepoints above 0xFF on a byte socket writes the UTF-8 encoding of each character and emits a Wide character in send warning under use warnings. Encode explicitly with Encode::encode before calling send.
- **:utf8 layer on the socket:** send **croaks**. Either open the socket without the layer, or call binmode \$sock to strip it before the first send. An :encoding(...) layer

implicitly adds `:utf8`, so it's banned too.

- **Four-arg form on a connected socket:** passing `T0` to a connected `SOCK_STREAM` socket is an error - the kernel returns `EISCONN`. Use the three-argument form once connected.
- **Three-arg form on an unconnected datagram socket:** fails with `ENOTCONN` (or `EDESTADDRREQ` on some platforms). Either `connect` the datagram socket first (which pins a default peer), or supply `T0`.
- **Empty MSG on UDP:** valid - sends a zero-length datagram the peer can read with `recv`. On TCP it is a no-op that returns `0`.
- **SIGPIPE on a closed stream peer:** writing to a `SOCK_STREAM` whose remote end has closed raises `SIGPIPE`, which by default terminates the process. Pass `MSG_NOSIGNAL` in `FLAGS` (Linux) or install `$SIG{PIPE} = 'IGNORE'` to get an `EPIPE` return instead.
- **sendmsg(2):** Perl exposes `send/sendto`, not `sendmsg`. Scatter/gather writes and ancillary data (credentials, file descriptors over `AF_UNIX`) require an XS module such as `Socket::MsgHdr`.
- **Non-socket filehandle:** `send` on a non-socket returns `undef` with `$!` set to `ENOTSOCK`. Use `syswrite` for pipes and regular files.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `recv` - the inverse; reads bytes from a socket with the same layer restrictions and the same `FLAGS` argument
- `syswrite` - raw write that bypasses `PerlIO` buffering on any filehandle; use it for pipes and files where `send` refuses to work
- `connect` - establishes the peer address that lets you use the three-argument form of `send`
- `socket` - creates the filehandle `send` writes to; the socket type (`SOCK_STREAM` vs `SOCK_DGRAM`) decides which form of `send` is legal
- `bind` - assigns a local address; needed before `send` on a datagram socket if the peer expects replies to a known port
- `binmode` - strips a stray `:utf8` layer that would otherwise make `send` throw

User and group info

setgrent

Rewind the group-database iterator back to the first entry.

`setgrent` tells the system's group-database iterator to start over. The next call to `getgrent` returns the first group record again, regardless of how far a previous iteration walked. It takes no arguments, returns no useful value, and exists so that a program can

re-scan the group database without having to close and reopen it by hand. Pair it with `endgrent` at the end of a scan to release the underlying file descriptor.

Synopsis

```
setgrent;  
setgrent();
```

What you get back

The return value is unspecified and conventionally ignored. Call `setgrent` for its side effect - positioning the iterator - not for its value.

Global state it touches

- The **per-process group-database iterator** maintained by the C library (`getgrent(3)`). It is global to the process, not scoped to a package or a lexical: any other code walking `getgrent` in the same process, including inside threads that share the iterator, sees the rewind.
- `$!` - on systems whose `setgrent(3)` reports errors (rare; most implementations return void), a failure leaves `errno` set. Portable code does not rely on this.

`setgrent` does not touch `$_, @ARGV`, or any filehandle.

Examples

Rewind between two passes over the group database - first to collect all group names, then to collect the members of a specific group:

```
my @names;  
while (my @g = getgrent) {  
    push @names, $g[0];  
}  
  
setgrent;                                # start over  
while (my @g = getgrent) {  
    next unless $g[0] eq 'wheel';  
    print "wheel members: $g[3]\n";  
    last;  
}  
endgrent;
```

Use `setgrent` at the top of a sub that is called repeatedly and wants a fresh scan every time:

```
sub groups_containing {  
    my ($user) = @_;  
    setgrent;  
    my @groups;
```

(continues on next page)

(continued from previous page)

```

while (my ($name, undef, undef, $members) = getgrent) {
    push @groups, $name if grep { $_ eq $user } split / /,
↪$members;
}
endgrent;
return @groups;
}

```

Pair `setgrent` / `endgrent` around a scan so an unrelated caller that was mid-iteration is not surprised when control returns:

```

setgrent;
while (my @g = getgrent) {
    # ... process ...
}
endgrent;

```

Edge cases

- **No argument form only.** Unlike `sethostent`, `setnetent`, `setprotoent`, and `setservent`, which take a `STAYOPEN` flag, `setgrent` takes nothing. Passing an argument is a compile-time error.
- **Calling `setgrent` without an intervening `getgrent` is a no-op on most systems** - the iterator is simply positioned at the first entry, and the first `getgrent` after it reads that entry. Don't rely on `setgrent` as an “open the group file” primitive; its only defined behaviour is “rewind”.
- **Process-global state.** `setgrent` inside one module rewinds the iterator for every other module in the same process. If a library walks `getgrent` to completion and a caller then calls `setgrent`, the library's cursor is gone. Treat the iterator as a shared resource: own it for the duration of a scan, bracketed by `setgrent` and `endgrent`.
- **Threads share the iterator.** Two threads calling `getgrent` concurrently will each see a subset of the records; `setgrent` from one thread disturbs the other. For per-thread iteration, read `/etc/group` directly or use a thread-safe wrapper; the built-in is not reentrant.
- **Not all platforms implement it identically.** On systems without `setgrent(3)` the call is a no-op and silently succeeds. On systems that do implement it, the exact cost (reopening `/etc/group`, restarting an NSS query, re-querying LDAP) depends on the backing name-service and can be non-trivial - avoid calling it in a tight loop.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `getgrent` - read the next group record; consumes the iterator that `setgrent` rewinds
- `endgrent` - close the iterator when the scan is finished; the symmetric bookend to `setgrent`
- `getgrnam` - lookup a single group by name without touching the iterator at all; prefer this for one-shot queries
- `getgrgid` - lookup a single group by GID; same reasoning as `getgrnam`
- `setpwent` - the equivalent rewind for the password database

Network info

sethostent

Open or rewind the hosts database for iteration.

`sethostent` prepares the system hosts database (typically `/etc/hosts` plus any name-service sources configured by the platform) for a sequential walk with `gethostent`, and rewinds the cursor if iteration is already in progress. `STAYOPEN` is a flag passed through to the underlying C `sethostent(3)` call: a true value asks the resolver to keep the database open across subsequent `gethostby*` lookups, which on some systems avoids reopening the file or reconnecting to the name service for every query.

Synopsis

```
sethostent STAYOPEN
sethostent 0
sethostent 1
```

What you get back

Returns nothing useful. The call is invoked for its side effect of priming the iterator; ignore the return value.

Global state it touches

Mutates **process-global** resolver state maintained by the C library: the hosts-database file descriptor or name-service handle, and the per-process iteration cursor read by `gethostent`. Two pieces of code in the same process walking the hosts database simultaneously will step on each other - there is one cursor, not one per caller. On failure `$!` is set by the underlying system call; there is no `h_errno`-style reporting for this function.

Examples

Walk every entry in the hosts database:

```
sethostent(0);
while (my ($name, $aliases, $addrtype, $length, @addrs)
      = gethostent()) {
    print "$name\n";
}
endhostent();
```

Ask the resolver to keep the database open while a batch of name-to-address lookups runs, then close it:

```
sethostent(1);
for my $host (@hosts) {
    my $packed = gethostbyname($host);
    # ...
}
endhostent();
```

Rewind an in-progress iteration to start over from the first entry:

```
sethostent(0);
my $first = (gethostent())[0];
sethostent(0);                                # back to the top
my $first_again = (gethostent())[0];
```

Edge cases

- **STAYOPEN is mandatory syntax, not mandatory semantics.** The argument must be present - `sethostent`; is a compile error - but many platforms ignore the flag and always keep the database open for the duration of the iteration. Pass `0` when you have no preference.
- **Not every platform has a meaningful hosts iterator.** On systems where the hosts database is served exclusively by DNS or a name service that does not expose enumeration, `gethostent` returns the empty list on the first call regardless of `sethostent`. The call itself still succeeds.
- **Thread safety.** The resolver cursor is per-process, not per-thread. Two threads iterating concurrently will interleave records and neither will see the full list. Upstream Perl substitutes `*_r` variants for point lookups like `gethostbyname` where available; there is no thread-safe variant of the iterator.
- **Interaction with `gethostby*` caches.** On some libc implementations a `sethostent(1)` issued before a burst of `gethostbyname` / `gethostbyaddr` calls keeps the backing file open between queries; on others it has no effect. Measure before relying on it as an optimisation.
- **Pair with `endhostent`.** Leaving the database open after iteration can leak a file descriptor or a resolver connection. Always close the walk with `endhostent`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `gethostent` - read the next entry from the hosts database after `sethostent` has primed the cursor
- `endhostent` - close the database and release resolver state when iteration is done
- `gethostbyname` - point lookup by name; benefits from `sethostent(1)` on platforms that honour it
- `gethostbyaddr` - point lookup by address; same caveat as `gethostbyname`
- `setnetent` - the networks-database counterpart, same iteration pattern
- `setservent` - the services-database counterpart

Network info

setnetent

Open or rewind the networks database for iteration.

`setnetent` prepares the system networks database so that subsequent calls to `getnetent` return entries starting from the first one. It is a direct wrapper around the C library's `setnetent(3)` and reads the same underlying source as the C call - typically `/etc/networks` on Linux, or whatever `nsswitch.conf` / NSS backends have been configured to provide.

The `STAYOPEN` argument is a hint to the C library: a true value asks it to keep the database file open across related calls (`getnetbyname`, `getnetbyaddr`, `getnetent`) so individual lookups do not each pay the cost of opening and closing the file. On modern glibc this flag is effectively ignored - the database is handled efficiently either way

- but passing it costs nothing and keeps the code portable to systems where it still matters.

Synopsis

```
setnetent STAYOPEN
setnetent 0
setnetent 1
```

What you get back

`setnetent` returns nothing meaningful. In list context it returns an empty list, in scalar context `undef`. Do not rely on the return value; the call's effect is the side effect of positioning the database iterator at the start.

If the underlying C call fails (for example the database file is missing or unreadable) the error is silent from Perl's perspective - `getnetent` will simply return an empty list on the first call.

Global state it touches

- The process-wide **networks-database cursor** maintained by the C library. Every call to `getnetent` advances this cursor; `setnetent` rewinds it. `endnetent` closes the database and releases any resources held by a STAYOPEN hint.
- No Perl special variables are read or written.

Because the cursor is per-process, not per-thread, interleaving networks-database iteration from multiple threads in the same interpreter will corrupt each iterator's position. Perform the iteration from a single thread, or serialise access.

Examples

Rewind the database and iterate from the first entry:

```
setnetent(1);
while (my ($name, $aliases, $addrtype, $net) = getnetent()) {
    printf "%-20s %d\n", $name, $net;
}
endnetent();
```

Re-run the iteration from the top without closing in between. The second `setnetent` rewinds, so the second loop sees every entry again:

```
setnetent(1);
my @first_pass = collect_nets();
setnetent(1);
my @second_pass = collect_nets();
endnetent();

sub collect_nets {
    my @out;
    while (my @ent = getnetent()) { push @out, $ent[0] }
    return @out;
}
```

Pairing with `Net::netent` for a named-field interface over the same data source - `setnetent` still controls the cursor:

```
use Net::netent;
setnetent(1);
while (my $n = getnetent()) {
    printf "%s -> %s\n", $n->name, $n->net;
}
endnetent();
```

Edge cases

- **STAYOPEN is a hint, not a guarantee.** On systems where the C library ignores it, passing 1 and passing 0 produce identical observable behaviour.

- **No implicit `setnetent` before `getnetent`.** A fresh process that calls `getnetent` without first calling `setnetent` still starts from the first entry on most systems, but this is a libc convention rather than a guarantee. Call `setnetent` explicitly when you want to be sure.
- **Calling `setnetent` mid-iteration rewinds the cursor.** Any in-progress while (`getnetent`) loop restarts from the top - useful for a deliberate second pass, a bug if it happens by accident from a helper sub that also iterates.
- **Interaction with `getnetbyname` / `getnetbyaddr`.** On systems where the C library honours STAYOPEN, a by-name or by-address lookup during iteration will *not* disturb the iterator. On systems that ignore the hint, each lookup may reset the cursor. Do not mix by-name lookups with `getnetent` iteration if you need stable positioning.
- **Permissions.** The networks database is typically world-readable, but NSS backends (LDAP, sssd) may impose their own authorisation. A lookup that worked as root can yield nothing as a normal user without Perl seeing an error.
- **Empty or missing `/etc/networks`.** On most Linux systems this file is empty or absent by default. `setnetent` still succeeds; the following `getnetent` returns an empty list on the first call. This is not an error condition.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `getnetent` - the iterator primed by `setnetent`; reaches for the next row on each call
- `endnetent` - close the database and release any resources held open by STAYOPEN; call it when iteration is done
- `getnetbyname` - lookup a single entry by name without running the iterator
- `getnetbyaddr` - lookup a single entry by packed address without running the iterator
- `sethostent` - the same idiom for the hosts database
- `setservent` - the same idiom for the services database
- `Net::netent` - object-oriented wrapper returning named fields instead of a positional list

Processes

setpgrp

Set the process group of a process.

`setpgrp` moves the process identified by PID into the process group identified by PGRP. A PID of 0 means the current process; a PGRP of 0 means “use PID as the new group id” - that is, make the process a group leader. With no arguments it defaults to `setpgrp(0,`

0), which makes the current process the leader of a new process group whose id equals its own pid.

This is the Perl interface to the underlying `setpgid(2)` / `setpgrp(2)` system call. It is the building block for detaching a process from its controlling terminal and for job-control schemes that want to send signals to a whole group at once with `kill`.

Synopsis

```
setpgrp PID, PGRP
setpgrp                # same as setpgrp(0, 0)
```

What you get back

1 on success, `undef` on failure with `$!` set to the reason (commonly `EPERM` when the target process is not the caller or a child of the caller, or `ESRCH` when no such process exists). Check the return value - silently proceeding with a process still in the old group is almost never what you want.

```
setpgrp(0, 0)
    or die "setpgrp failed: $!";
```

Global state it touches

- `$!` - set to the `errno` value on failure. Not touched on success.

Examples

Detach the current process into a new process group of its own. This is the classic first step of a daemonisation sequence, run after `fork` and before closing the standard handles:

```
setpgrp(0, 0)
    or die "setpgrp failed: $!";
```

Move a just-spawned child into its own group so that a `SIGTERM` sent to the group kills the child and any grandchildren it spawns, without touching the parent:

```
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    setpgrp(0, 0) or die "setpgrp: $!";
    exec @cmd or die "exec: $!";
}
# parent: signal the whole group with a negative pid
kill 'TERM', -$pid;
```

Read the group id back after setting it, using `getpgrp`:

```
setpgrp(0, 0) or die $!;
my $pgid = getpgrp(0);
print "now leader of group $pgid\n";
```

Handle the portability failure mode - on a platform without the underlying system call the function raises an exception rather than returning `undef`:

```
eval { setpgrp(0, 0) };  
if ($?) {  
    warn "no process-group support on this platform: $@";  
}
```

Edge cases

- **No arguments defaults to (0, 0).** Writing `setpgrp`; is the same as `setpgrp(0, 0)` - make the current process a new group leader. The no-argument form is the portable form (see below).
- **BSD 4.2 compatibility.** The historical BSD 4.2 `setpgrp` took no arguments. Scripts that must run on such systems can only rely on the `setpgrp(0, 0)` / `setpgrp()` forms; any other argument combination is non-portable.
- **Raises, not returns undef, on unsupported platforms.** On a machine that implements neither POSIX `setpgid(2)` nor BSD `setpgrp(2)`, `setpgrp` raises an exception. This is the one failure mode that does **not** go through `$_` - wrap in `eval` if you need to degrade gracefully.
- **EPERM on cross-session moves.** The kernel rejects attempts to move a process into a group in a different session, or to move a process that has already called `exec`. Catch `$_ == EPERM` and fall back rather than retrying.
- **Signalling the whole group.** Once a group is established, `kill SIG, -$pgid` delivers SIG to every process in the group. The negative-pid convention is a property of `kill`, not of `setpgrp` - but it's the reason you usually call `setpgrp` in the first place.
- **Not a session leader.** `setpgrp` changes process group, not session. To detach from the controlling terminal entirely you need POSIX `::setsid`, which creates a new session and a new group in one step.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `getpgrp` - read the current process group id; the obvious companion for verifying a `setpgrp` call took effect
- `POSIX::setpgid` - the POSIX-named wrapper around the same system call, preferred in new code because its argument order and semantics are unambiguous across platforms
- `POSIX::setsid` - create a new session *and* a new process group in one call; use this, not `setpgrp`, for full terminal detachment
- `fork` - the call that almost always precedes `setpgrp`; a child that wants its own group calls `setpgrp(0, 0)` right after `fork` returns 0

- `kill` - how you actually use a process group once you've built one: `kill SIG, -$pgid` signals every member

Processes

setpriority

Set the scheduling priority (nice value) of a process, process group, or user.

`setpriority` is the Perl-level wrapper around the `setpriority(2)` system call. `WHICH` selects the target class (`PRIO_PROCESS`, `PRIO_PGRP`, or `PRIO_USER`), `WHO` names the specific target within that class (`0` means “self” or “current” depending on `WHICH`), and `PRIORITY` is the new nice value. Higher values mean *lower* scheduling priority; on Linux the useful range is `-20` (most favourable) through `19` (least favourable).

Synopsis

```
setpriority WHICH, WHO, PRIORITY
```

What you get back

Returns true on success, `undef` on failure (with `$!` set to the underlying `errno`). The most common failures are `EACCES` or `EPERM` - lowering the nice value below the current one, or touching another user's processes, requires privilege. Always check the return value when the priority change is not just cosmetic:

```
setpriority(PRIO_PROCESS, $$, 10)
or die "cannot renice self: $!";
```

Global state it touches

- `$!` - set on failure to the `errno` from the underlying `setpriority(2)` call.

The three `WHICH` classes

`WHICH` is one of three integer constants, each selecting a different interpretation of `WHO`:

- `PRIO_PROCESS` - `WHO` is a process ID. `0` means the current process.
- `PRIO_PGRP` - `WHO` is a process group ID. `0` means the process group of the current process.
- `PRIO_USER` - `WHO` is a numeric user ID (UID). `0` means the real UID of the current process. Setting by user changes the priority of *every* process owned by that user.

These constants are not built in - import them from `POSIX`:

```
use POSIX qw(PRIO_PROCESS PRIO_PGRP PRIO_USER);
```

Examples

Renice the current process to 10 (friendlier to other work):

```
use POSIX qw(PRIO_PROCESS);
setpriority(PRIO_PROCESS, 0, 10)
    or die "renice failed: $!";
```

Renice a child process after fork:

```
use POSIX qw(PRIO_PROCESS);
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    setpriority(PRIO_PROCESS, $$, 15); # child runs at low priority
    exec @cmd;
}
```

Renice every process in the current process group - useful in a session leader that wants its whole pipeline to yield:

```
use POSIX qw(PRIO_PGRP);
setpriority(PRIO_PGRP, 0, 5)
    or warn "could not renice process group: $!";
```

Pair with `getpriority` to adjust relative to the current value rather than setting an absolute one:

```
use POSIX qw(PRIO_PROCESS);
my $cur = getpriority(PRIO_PROCESS, 0);
setpriority(PRIO_PROCESS, 0, $cur + 5)
    or die "renice failed: $!";
```

Edge cases

- **Negative priorities require privilege.** Lowering the nice value (more favourable scheduling) below the current one needs `CAP_SYS_NICE` on Linux, or root. An unprivileged attempt fails with `EACCES` or `EPERM` depending on kernel version.
- **Nice values are clamped.** The kernel silently clamps `PRIORITY` to the supported range (`-20..19` on Linux). Passing `100` does *not* fail; it ends up as `19`.
- **WHO = 0 is context-sensitive.** Its meaning depends on `WHICH`: current PID for `PRIORITY_PROCESS`, current process group for `PRIORITY_PGRP`, real UID for `PRIORITY_USER`. It is not a universal “self” token.
- **Non-existent target.** A PID, PGID, or UID that does not match any running process fails with `ESRCH`.
- **Platform without `setpriority(2)`.** Perl raises a fatal exception (`setpriority()` not implemented). `ppperl` targets Linux only, so this case does not arise in practice, but portable code should still guard with `eval`.
- **Signed return from `getpriority` vs. success of `setpriority`.** `getpriority` can legitimately return `-1` as a nice value, which is why it sets `$!` to distinguish

error from result; `setpriority` has no such ambiguity - treat its return purely as a boolean.

- **Nice value is per-process, not per-thread.** On Linux this differs from POSIX: the kernel applies the nice value to the calling task (thread). Perl is single-threaded at the interpreter level, so this only matters if XS code has spawned kernel threads.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `getpriority` - read the current nice value; the natural counterpart when adjusting relative to the existing priority
- `POSIX` - source of the `PRIO_PROCESS`, `PRIO_PGRP`, and `PRIO_USER` constants
- `fork` - typical pairing: spawn a child, then `renice` it before `exec`
- `$!` - holds the `errno` describing why a failed `setpriority` call returned `undef`
- `kill` - other per-process control operation with the same “target by PID or PGID” convention

Network info

setprotoent

Open the protocols database and prepare it for sequential reads.

`setprotoent` rewinds (or opens) the system protocols database - on most Unix-like systems that's `/etc/protocols` - so that a subsequent sequence of `getprotoent` calls starts at the first entry. It is the first leg of the `setprotoent` / `getprotoent` / `endprotoent` iteration triad. The single `STAYOPEN` argument is a hint to the C library about whether the file handle backing the iteration should stay open between calls: a true value asks the library to keep it open (faster for long walks), a false value lets it close and reopen per call.

Calling `getprotobyname` or `getprotobynumber` anywhere mid-iteration may reset the walk position on some platforms; see *Edge cases*.

Synopsis

```
setprotoent STAYOPEN
setprotoent(1)
```

What you get back

No meaningful return value. Like its `endprotoent` and `sethostent` siblings, `setprotoent` is called for its side effect: rewinding the iteration and, depending on `STAYOPEN`, pinning the descriptor open. There is nothing worth checking in its return value.

Global state it touches

setprotoent manipulates a **per-process**, C-library-owned cursor into the protocols database. That state is shared across every subsequent `getprotoent` call in the same process - including calls from other modules you did not write. Code that walks the database must either fully drain it with `endprotoent` or accept that a later caller may see an iteration already in progress. No Perl-level special variables are read or written.

Examples

Open the database and iterate every entry, keeping the underlying descriptor alive for the full walk:

```
setprotoent(1);                                # 1 = keep file open between
↪ calls
while (my @p = getprotoent()) {
    my ($name, $aliases, $proto) = @p;
    print "$proto\t$name\n";
}
endprotoent();
```

Restart an in-progress iteration from the beginning:

```
setprotoent(1);
my @first = getprotoent();
my @second = getprotoent();

setprotoent(1);                                # rewind
my @again = getprotoent();                      # same as @first
endprotoent();
```

Let the C library close and reopen `/etc/protocols` on every `getprotoent` - cheaper in memory, more expensive in syscalls, useful only when you read a handful of entries:

```
setprotoent(0);
my @tcp = getprotoent();
endprotoent();
```

Pair with a one-shot lookup - `getprotobyname` and `getprotobynumber` do not participate in the iteration and do not need `setprotoent`:

```
my @tcp = getprotobyname('tcp');                # no setprotoent needed
```

Edge cases

- **STAYOPEN is a hint, not a contract:** many C libraries ignore it entirely and behave as if the descriptor were always kept open. Do not rely on the argument to control resource usage at a fine grain.
- **Mixing with single-shot lookups:** calling `getprotobyname` or `getprotobynumber` during a walk may reset the iteration cursor on some platforms. Portable code drains the walk first, or re-calls `setprotoent` afterwards to restart from the top.

- **Not reentrant:** the protocols database cursor is a single global resource inside the C library. Two concurrent iterations in the same process (threads, signal handlers, anything) corrupt each other. Use one walk at a time.
- **Argument is mandatory at the language level:** `setprotoent` without an argument is a syntax error - the built-in has arity one. Pass `0` if you have no preference.
- **Platforms without a protocols database:** on systems where the C library has no `/etc/protocols` equivalent, the call is a no-op rather than an error. See `perlport` for the full list of network-database quirks.
- **Untainted result of the iteration:** the records returned by `getprotoent` after `setprotoent` come from a file the kernel reads on your behalf and are not marked tainted. Treat them as untrusted anyway if the database can be user-edited.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `getprotoent` - read the next entry; the body of the loop that `setprotoent` sets up
- `endprotoent` - close the database after iteration; the terminator for the triad `setprotoent` starts
- `getprotobyname` - single-shot lookup by name; does not participate in the iteration
- `getprotobynumber` - single-shot lookup by protocol number; likewise independent
- `sethostent` - analogous opener for the hosts database; same idiom, different table
- `setservent` - analogous opener for the services database

User and group info

setpwent

Rewind the password-database iterator so the next `getpwent` call returns the first entry again.

`setpwent` wraps the C library's `setpwent(3)` call. It (re)opens the system password database and resets the iteration cursor used by `getpwent`, so a subsequent loop starts from the top. Use it when you want to iterate the password database more than once in the same process, or when you want to restart iteration after a partial walk. Pair it with `endpwent` to release the database when done.

Synopsis

```
setpwent;  
setpwent();
```

What you get back

Return value is **not useful** and must not be relied on. On most systems the underlying `setpwent(3)` has a void return; Perl yields a single meaningless true value. Call `setpwent` for its side effect, never for its value.

Global state it touches

- The per-process cursor shared by `getpwent`, `setpwent`, and `endpwent`. This cursor is owned by the C library, not by Perl - forking a child inherits an open database handle, and two threads iterating concurrently will interleave and lose records.
- `$!` is **not** set by `setpwent`; errors from the underlying `setpwent(3)` are swallowed by the platform.

Examples

Iterate the password database twice in the same program:

```
while (my @pw = getpwent) {  
    print "$pw[0]\n";           # first pass: every login name  
}  
setpwent;  
while (my @pw = getpwent) {  
    print "uid=$pw[2]\n";       # second pass: every uid  
}  
endpwent;
```

Restart iteration after an early exit:

```
while (my @pw = getpwent) {  
    last if $pw[0] eq 'root';  
}  
setpwent;                       # rewind so the next walk sees root  
↪again
```

Typical cleanup pattern - always call `endpwent` when done, whether or not you called `setpwent`:

```
setpwent;  
while (my ($name, undef, $uid) = getpwent) {  
    print "$name => $uid\n" if $uid >= 1000;  
}  
endpwent;
```

Edge cases

- **No arguments.** `setpwent` takes no arguments. Writing `setpwent $x` is a syntax trap: like other list operators, it will try to consume `$x` as the first element of a list.
- **Before any `getpwent` call.** Safe. `setpwent` opens the database if it is not already open, so it doubles as an explicit “open” call.

- **After endpwent.** Safe. The database is reopened.
- **Forked children.** Each process has its own iteration cursor after the fork, but the underlying file descriptor may be shared; interleaved reads from parent and child produce missing or duplicated records. Call `setpwent` in the child to get a clean iterator.
- **Threads.** `getpwent` iteration is per-process, not per-thread. Concurrent iteration from multiple threads loses records. Upstream Perl substitutes thread-safe `_r` variants for `getpwnam` and `getpwuid` when the platform provides them; there is no `setpwent_r`, and this function remains process-global.
- **NSS-backed databases.** On systems where `/etc/nsswitch.conf` routes `passwd` through LDAP, SSSD, or similar, `setpwent` may block on a network round-trip. Do not call it from signal handlers or latency-critical code paths.
- **Shadow passwords.** `setpwent` does not affect the shadow-password iterator (if any); it only resets the regular `passwd` walk.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `getpwent` - fetch the next entry; the function `setpwent` exists to rewind
- `endpwent` - release the password database when you are finished iterating
- `getpwnam` - direct lookup by login name; does not interact with the iteration cursor
- `getpwuid` - direct lookup by uid; does not interact with the iteration cursor
- `setgrent` - the same rewind operation for the group database
- `User::pwent` - object-oriented wrapper that returns named-field objects instead of 10-element lists

Network info

setservent

Rewind the services database and optionally keep it open across lookups.

`setservent` is the services-database counterpart of the C library's `setservent(3)`. It resets the internal cursor used by `getservent` so the next call starts again at the first entry of `/etc/services` (or whatever source the system resolver is configured to use), and it controls whether the underlying file stays open between by-name and by-port lookups.

Synopsis

```
setservent STAYOPEN
setservent 0          # close file between lookups (default)
setservent 1          # keep file open across lookups
```

What you get back

The return value is whatever the underlying C `setservent(3)` returns

- on most systems that's nothing useful, and scripts do not check it. Treat `setservent` as called for its side effect on the iterator, not for a value.

Global state it touches

- The per-process services-database cursor shared with `getservent`, `getservbyname`, `getservbyport` and `endservent`. All four read and mutate the same state; iteration is not thread-safe and not reentrant.
- The open/closed status of the services database file. `STAYOPEN` true leaves the file open until `endservent` is called; `STAYOPEN` false (the default) causes each subsequent by-name or by-port lookup to open and close the file again.

The STAYOPEN argument

Pass a true value to hold the file open across a run of individual `getservbyname` or `getservbyport` lookups. This is an optimisation for batch resolution, not a semantic change: results are identical either way.

```
setservent 1;
for my $name (@service_names) {
    my @svc = getservbyname $name, "tcp";
    # ... process @svc ...
}
endservent;
```

Pass a false value (or `0`) when you want each lookup to reopen the file. This is the default behaviour and the one most programs want.

Examples

Walk every entry in the services database and print name/port pairs:

```
setservent 0;
while (my ($name, undef, $port, $proto) = getservent) {
    print "$name $port/$proto\n";
}
endservent;
```

Rewind mid-iteration to start over:

```
setservent 0;
my @first_pass;
while (my @svc = getservent) {
    push @first_pass, $svc[0];
    last if @first_pass == 10;
}
setservent 0;                                # cursor back to start
```

(continues on next page)

(continued from previous page)

```
my @second_pass = getservent;      # first entry again
endservent;
```

Batch-resolve with the file held open:

```
my @wanted = qw(http https ssh smtp);
setservent 1;
my %port = map { $_ => scalar getservbyname($_, "tcp") } @wanted;
endservent;
```

Edge cases

- **STAYOPEN is required at the Perl level.** `setservent` is parsed with list-operator precedence and reads one argument; a bare `setservent`; with no argument is a compile-time error. Pass `0` when you want the default “close after each lookup” behaviour.
- **Platform dependence.** The services database is whatever the system resolver uses - typically `/etc/services`, but NIS, LDAP, and other name-service backends can supply entries. `setservent` rewinds whichever source the C library selects; the Perl level has no say in it.
- **Non-Unix platforms.** On systems without `setservent(3)` the call croaks with `The setservent function is unimplemented.` See `perlport` for the availability matrix.
- **Threads.** Iterator state is per-process, not per-thread; two threads walking the services database concurrently will interleave and neither will see every record. See the shared note in `perlfunc` on `getpwnam`-family thread safety.
- **Pairing with `endservent`.** When `STAYOPEN` was true, call `endservent` to release the file descriptor; otherwise the handle leaks for the life of the process.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `getservent` - walk the services database one entry at a time; this is the iterator `setservent` rewinds
- `endservent` - close the services database file opened by `setservent 1`
- `getservbyname` - look up a single service by name; honours the open/closed state set here
- `getservbyport` - look up a single service by port number; same open/closed semantics
- `setprotoent` - same pattern for the protocols database when you are rewinding both together

- `Net::servent` - object-oriented wrapper over the `getserv*` family for when positional return lists get unwieldy

Sockets

setsockopt

Set a kernel-level option on an open socket.

`setsockopt` is the Perl wrapper around the POSIX `setsockopt(2)` system call. It writes the option named by `OPTNAME` at protocol `LEVEL` on the already-opened `SOCKET` filehandle, using the value packed into `OPTVAL`. It is the knob for everything the kernel lets you tune on a socket - address reuse, keepalives, timeouts, Nagle's algorithm, multicast membership, send and receive buffer sizes, `SO_LINGER` behaviour on `close`, and so on. Symbolic names for `LEVEL` and `OPTNAME` come from the `Socket` module.

Synopsis

```
setsockopt SOCKET, LEVEL, OPTNAME, OPTVAL
```

What you get back

A true value on success, `undef` on failure (with `$!` set to the underlying `errno`: `EBADF`, `ENOPROTOPT`, `EINVAL`, `ENOTSOCK`, etc.). Always check the return value - a silently-ignored option means the program runs with defaults that differ from what the code reads:

```
setsockopt($sock, SOL_SOCKET, SO_REUSEADDR, pack("l", 1))  
or die "setsockopt SO_REUSEADDR: $!";
```

Global state it touches

None directly. `setsockopt` sets `$!` on failure; that is the only interpreter global involved. The effect is on the kernel-side state of the socket, not on any Perl-level variable.

OPTVAL - packed bytes or a bare integer

Most socket options take a C `int` flag. `setsockopt` accepts either form for convenience:

- A **packed byte string** whose layout matches the option's C type, typically built with `pack`: `pack("l", 1)` for a boolean, `pack("ll", $on_secs, $on_usecs)` for a struct `timeval`, `pack("ii", $on, $linger)` for struct `linger`.
- A **plain integer**, which is shorthand for `pack("i", OPTVAL)`. Handy for `SO_REUSEADDR`, `TCP_NODELAY`, and other boolean flags:

```
setsockopt($sock, IPPROTO_TCP, TCP_NODELAY, 1);
```

Options whose value is a struct (`SO_LINGER`, `SO_RCVTIMEO`, `IP_ADD_MEMBERSHIP`) must be passed as an explicit **packed** string. A bare integer there produces `EINVAL` or, worse, silently writes wrong-sized memory that the kernel misinterprets.

Examples

Disable Nagle's algorithm on a TCP socket so small writes go out immediately - useful for latency-sensitive protocols (interactive shells, game traffic, RPC):

```
use Socket qw(IPPROTO_TCP TCP_NODELAY);

setsockopt($sock, IPPROTO_TCP, TCP_NODELAY, 1)
    or die "TCP_NODELAY: $!";
```

Allow a server to rebind a port that is still in TIME_WAIT from a previous process - the single most common setsockopt call in practice, and it must come **before** bind:

```
use Socket qw(SOL_SOCKET SO_REUSEADDR);

setsockopt($srv, SOL_SOCKET, SO_REUSEADDR, pack("l", 1))
    or die "SO_REUSEADDR: $!";
bind($srv, sockaddr_in(9000, INADDR_ANY))
    or die "bind: $!";
```

Set a receive timeout so a blocking `recv` or read from the socket returns with EAGAIN / EWOULDBLOCK after 5 seconds instead of hanging forever. The option value is a packed struct timeval:

```
use Socket qw(SOL_SOCKET SO_RCVTIMEO);

my $tv = pack("l!l!", 5, 0);    # 5 seconds, 0 microseconds
setsockopt($sock, SOL_SOCKET, SO_RCVTIMEO, $tv)
    or die "SO_RCVTIMEO: $!";
```

Enable TCP keepalive probes so half-open connections (peer crashed, cable yanked) are eventually noticed and the socket errors out rather than blocking indefinitely:

```
use Socket qw(SOL_SOCKET SO_KEEPALIVE);

setsockopt($sock, SOL_SOCKET, SO_KEEPALIVE, 1)
    or die "SO_KEEPALIVE: $!";
```

Make `close` block for up to 10 seconds while the kernel flushes unsent data, then discard and reset. SO_LINGER takes a two-integer struct: the on-flag and the timeout in seconds:

```
use Socket qw(SOL_SOCKET SO_LINGER);

setsockopt($sock, SOL_SOCKET, SO_LINGER, pack("ii", 1, 10))
    or die "SO_LINGER: $!";
```

Enlarge the kernel receive buffer for a bulk-transfer socket. The kernel typically doubles the value you request and caps it at `net.core.rmem_max`:

```
use Socket qw(SOL_SOCKET SO_RCVBUF);

setsockopt($sock, SOL_SOCKET, SO_RCVBUF, 1 << 20)    # 1 MiB
    or die "SO_RCVBUF: $!";
```


Edge cases

- **SOCKET must already be open.** `setsockopt` does not create the socket - call `socket` first. A closed or never-opened filehandle fails with `EBADF`.
- **Ordering matters.** `SO_REUSEADDR` must be set *before* `bind`. `SO_KEEPALIVE` can go before or after `connect`, but for server sockets set it on the accepted child filehandle returned by `accept`, not on the listening one - options do not always propagate.
- **Integer shorthand is `pack("i", N)`, not `pack("l", N)`.** On platforms where `int` and `long` differ in size, passing a plain integer and passing `pack("l", N)` write different byte counts. Both happen to work for the common `int`-sized boolean options on LP64 Linux, but the shorthand form is the portable default.
- **Struct-valued options need explicit `pack`.** `SO_LINGER`, `SO_RCVTIMEO`, `SO_SNDTIMEO`, `IP_ADD_MEMBERSHIP`, `IP_MREQ`, and similar take fixed-layout structs. Pass a packed string of the right length or the kernel rejects the call with `EINVAL`.
- **LEVEL picks the option namespace.** `SOL_SOCKET` for generic options, `IPPROTO_TCP` for TCP-specific (`TCP_NODELAY`, `TCP_KEEPIIDLE`), `IPPROTO_IP` for IPv4 (`IP_TTL`, `IP_ADD_MEMBERSHIP`), `IPPROTO_IPV6` for IPv6. Mismatching `LEVEL` and `OPTNAME` gives `ENOPROTOOPT`.
- **Read-only or unsupported options** fail with `ENOPROTOOPT`. This also fires when the kernel simply does not know the option - common when copying code between operating systems.
- **Privileged options.** A few options (`SO_PRIORITY` above certain values, `IP_TRANSPARENT`, `SO_BINDTODEVICE`) require `CAP_NET_ADMIN` on Linux; an unprivileged call fails with `EPERM`.
- **Reading back is `getsockopt`, not `setsockopt`.** `setsockopt` has no query form; use `getsockopt` to confirm what the kernel actually stored (it may round, clamp, or double the value).

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `getsockopt` - the query counterpart; read back an option to see what the kernel actually applied
- `socket` - create the socket filehandle that `setsockopt` configures; must be called first
- `bind` - set `SO_REUSEADDR` with `setsockopt` before calling `bind` on a server socket
- `accept` - options set on the listener do not all propagate; re-apply per-connection options on the accepted handle
- `pack` - builds the `OPTVAL` byte string for struct-valued options like `SO_LINGER` and `SO_RCVTIMEO`
- `Socket` - source of `SOL_SOCKET`, `IPPROTO_TCP`, `SO_REUSEADDR`, `TCP_NODELAY`, and the rest of the symbolic constants

Arrays

shift

Remove and return the first element of an array.

`shift` mutates its argument: it removes the element at index 0, shifts every remaining element down by one position, and returns the removed value. The array shrinks by one. If the array is empty, nothing is removed and `undef` is returned.

Synopsis

```

shift @arr
shift @$aref
shift                               # @_ inside a sub, @ARGV outside

```

What you get back

The scalar that used to be at index 0, or `undef` if the array was empty. Because `undef` is also a legal array element, a bare `undef` return does not by itself mean the array was empty - use `scalar @arr` or a prior `defined` check when the distinction matters:

```

my @arr = (undef, 'two');
my $item = shift @arr;      # undef - but @arr was not empty

```

Default array: @ARGV vs @_

Called without an argument, `shift` picks its array from the surrounding context:

- Inside a named or anonymous subroutine - `@_`. This is the idiomatic way to consume positional arguments one at a time.
- Outside any sub (including in the file scope of a script, and in `eval STRING, BEGIN, INIT, CHECK, UNITCHECK, END` blocks)
 - `@ARGV`. This is how top-of-script argument consumption is usually written.

The choice is made at compile time based on lexical position, not at runtime. A `shift` inside a sub always targets `@_` even if the sub is called from file scope.

Examples

Consume a single positional argument inside a method. This is the canonical first line of almost every Perl OO method:

```

sub greet {
    my $self = shift;
    my $name = shift;
    return "hello, $name, from $self";
}

```

The same method written with a list-unpack `@_` slice. Both forms are idiomatic; `shift` wins when you want to peel off the invocant before deciding what to do with the rest of `@_`:

```
sub greet {  
    my ($self, $name) = @_;  
    return "hello, $name, from $self";  
}
```

Consume `@ARGV` at the top of a script, one argument at a time:

```
my $mode = shift // 'default';      # first CLI arg, or 'default'  
my $path = shift // die "need a path\n";  
# @ARGV now holds whatever is left
```

A FIFO queue: `push` on the tail, `shift` off the head:

```
my @queue;  
push @queue, 'a', 'b', 'c';  
while (defined(my $job = shift @queue)) {  
    handle($job);  
}
```

Dereference an array ref in place - `shift` works on `@$aref` exactly as it does on a named array:

```
my $aref = [10, 20, 30];  
my $head = shift @$aref;           # 10, $aref now [20, 30]
```

Edge cases

- **Empty array:** returns `undef` and leaves the array untouched. `shift` on an empty array is not an error.
- **Complexity is $O(n)$:** every remaining element is moved down by one slot. For large arrays used as queues this adds up. When FIFO ordering is not required, `pop` is $O(1)$; when it is, consider a deque-style data structure or a pair of stacks rather than `shift`-ing a million-element array in a loop.
- **`shift` with no argument is context-sensitive by lexical position:** inside a sub it targets `@_`, outside it targets `@ARGV`. Moving a line of code between those scopes silently changes which array it mutates.
- **Tied arrays:** `shift` calls the `SHIFT` method on the tied object if one is defined; otherwise it falls back to `FETCH` on element 0, then `DELETE`, then a sequence of `FETCH/STORE` pairs to shift the remaining elements down, then `STORESIZE`. A custom `SHIFT` handler is almost always worth writing for performance.
- **Array references:** `shift @$aref` and `shift @{ $expr }` both work. `shift $aref` (without the `@` sigil) was an experimental feature added in Perl 5.14 and removed in 5.24 - it is a syntax error in the Perl version `ppperl` targets.
- **`shift` is an lvalue-consumer, not an lvalue:** you cannot assign to the result of `shift` to modify the array. Use `splice` or direct index assignment for that.

- **Return value in list context:** `shift` always returns a single scalar. Assigning its result to a list - `my ($x) = shift @arr` - works only because the single scalar fills the first list slot; the remaining slots are `undef`. Use `splice` to remove multiple elements at once.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `pop` - remove and return the **last** element; $O(1)$, the right choice when FIFO order is not needed
- `push` - append to the tail; pairs with `shift` to form a FIFO queue
- `unshift` - prepend to the head; the inverse of `shift`, and also $O(n)$ for the same reason
- `splice` - remove, insert, or replace any contiguous run of elements; the general tool when `shift` / `pop` / `unshift` / `push` are not enough
- `@ARGV` - the default array `shift` consumes at file scope
- `@_` - the default array `shift` consumes inside a sub; holds the caller's argument list

SysV IPC

shmctl

Control or query a System V shared memory segment.

`shmctl` is the management primitive for shared memory segments created with `shmget`. It performs one of a small set of operations named by `CMD` - reading the segment's meta-data, changing its permissions, or marking it for removal. `ID` is the segment identifier returned by `shmget`. `ARG` is a buffer scalar that either supplies an input `shmid_ds` structure (for `IPC_SET`) or receives the returned one (for `IPC_STAT`); for `IPC_RMID` it is ignored.

Synopsis

```
use IPC::SysV qw(IPC_STAT IPC_SET IPC_RMID);

shmctl $id, IPC_STAT, $buf or die "shmctl STAT: $!";
shmctl $id, IPC_SET,  $buf or die "shmctl SET: $!";
shmctl $id, IPC_RMID, 0   or die "shmctl RMID: $!";
```

What you get back

Return value follows the `ioctl` convention, not the usual 0-on-success Unix pattern:

- `undef` on error, with `$!` set to the `errno`.

- The string "0 but true" when the underlying call returned zero. This compares numerically equal to 0 but is boolean-true, so `shmctl(...)` or `die` still works.
- Any other non-zero value otherwise.

The practical rule: test for falsehood with `shmctl(...)` or `die "...: $!"` and never compare the return value to literal 0 with `==`.

The `IPC_STAT` form writes a packed `shmid_ds` structure into `ARG`. Unpack it with `IPC::SysV`'s helper or with a manual `unpack` template; the structure layout is platform-dependent so relying on a fixed template is not portable.

Global state it touches

- `$!` is set on failure to the `errno` from the underlying `shmctl(2)` call. Typical values: `EINVAL` (bad ID or CMD), `EACCES` (insufficient permissions for the operation), `EPERM` (non-owner attempting `IPC_SET` or `IPC_RMID`), `EFAULT` (bad `ARG` buffer for `IPC_STAT/IPC_SET`).

Examples

Read a segment's metadata. `IPC::SysV` provides the `shmid_ds` unpacker; without it, the structure is an opaque byte string:

```
use IPC::SysV qw(IPC_STAT);

my $buf = "";
shmctl($id, IPC_STAT, $buf) or die "stat failed: $!";
# $buf now holds the packed shmid_ds structure
```

Change the permissions on an existing segment. The pattern is read-modify-write on the packed structure:

```
use IPC::SysV qw(IPC_STAT IPC_SET);

my $buf = "";
shmctl($id, IPC_STAT, $buf) or die $!;
# ... modify the mode bits inside $buf ...
shmctl($id, IPC_SET, $buf) or die $!;
```

Mark a segment for removal. The segment is destroyed once the last process detaches from it; `IPC_RMID` only flags it:

```
use IPC::SysV qw(IPC_RMID);

shmctl($id, IPC_RMID, 0) or die "rmid failed: $!";
```

Guarding on the "0 but true" return. Because the zero case is a string that is boolean-true, the idiomatic check is the `or die` form, not a numeric comparison:

```
my $rv = shmctl($id, IPC_STAT, $buf);
defined $rv or die "error: $!";
# do NOT write: $rv == 0 and ... # misses the "0 but true" case
```

Edge cases

- **Missing constants:** `IPC_STAT`, `IPC_SET`, `IPC_RMID` are not built-in. Without use `IPC::SysV` they are bareword function calls and the code either fails to compile under use `strict` or silently passes the wrong integer. Always use `IPC::SysV qw( ... )`.
- **Owner-only operations:** `IPC_SET` and `IPC_RMID` require the effective uid to match the segment's creator or owner, or the process to have `CAP_SYS_ADMIN`. Otherwise `$!` becomes `EPERM`.
- **IPC_RMID semantics:** marking for removal does not detach the caller and does not invalidate existing attachments. The segment stays usable until every process has called `shmdt` on it; only then is it actually destroyed. A process that attaches after `IPC_RMID` gets `EIDRM` or `EINVAL` depending on timing.
- **Structure layout is not portable:** `shmid_ds` fields and offsets vary between kernels and libc versions. Do not hand-code `unpack` templates for scripts that need to run on multiple platforms; use `IPC::SysV::ShmId` or a similar wrapper.
- **Buffer sizing for IPC_STAT:** `ARG` must be a scalar of at least `sizeof(shmid_ds)` bytes on output. Perl grows it as needed; pre-sizing with `$buf = "\0" x 256` is common but not required.
- **Unconfigured kernels:** on Linux built without `CONFIG_SYSVIPC`, the call fails with `ENOSYS`. Container images that drop SysV IPC exhibit the same behaviour.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `shmget` - create or look up the segment whose ID `shmctl` operates on
- `shmread` - copy bytes out of an attached segment; uses the ID returned by `shmget`, not an `shmctl` result
- `shmwrite` - copy bytes into an attached segment
- `msgctl` - same control pattern for System V message queues; same "`0` but `true`" return convention
- `semctl` - same control pattern for System V semaphore sets; richer CMD set but identical return-value rules
- `IPC::SysV` - provides the `IPC_STAT` / `IPC_SET` / `IPC_RMID` constants and the `shmid_ds` packers

SysV IPC

shmget

Create or look up a System V shared memory segment and return its identifier.

`shmget` is the allocation step for SysV shared memory. Given a numeric `KEY` (either `IPC_PRIVATE` for a fresh anonymous segment or a well-known number agreed with

other processes), a requested SIZE in bytes, and a bit-mask of FLAGS (creation flags OR'd with a 9-bit mode), it returns a kernel-assigned integer identifier. That identifier - the shmid - is what every subsequent call (`shmctl`, `shmread`, `shmwrite`) uses to refer to the segment. The segment persists in the kernel until it is explicitly removed with `shmctl($shmid, IPC_RMID, 0)` or the system reboots; closing the creating process does **not** release it.

Synopsis

```
shmget KEY, SIZE, FLAGS
```

What you get back

An integer shared-memory identifier (shmid) on success, suitable for passing to `shmctl`, `shmread`, and `shmwrite`. On failure returns a false value and sets `$!` to the underlying `errno`.

```
use IPC::SysV qw(IPC_PRIVATE IPC_CREAT S_IRUSR S_IWUSR);

my $shmid = shmget(IPC_PRIVATE, 4096, IPC_CREAT | S_IRUSR | S_IWUSR)
    or die "shmget: $!";
```

The returned identifier is an opaque small integer. Do not do arithmetic on it, do not compare it to 0 for truth - the identifier 0 is a legitimate value on some systems. Always check the return against the documented failure sentinel (see *Differences from upstream*).

Global state it touches

- `$!` - set on failure to the underlying `errno`: EACCES (permissions), EEXIST (segment for that key already exists and both `IPC_CREAT` and `IPC_EXCL` were given), EINVAL (SIZE below `SHMMIN` or above `SHMMAX`, or SIZE differs from the existing segment when looking one up), ENFILE / ENOSPC (kernel segment-table full), ENOENT (no segment for that key and `IPC_CREAT` was not given), ENOMEM (insufficient memory).

The function is otherwise pure with respect to Perl-visible state: no special variables beyond `$!` are touched, `$_` is not read.

Examples

Create a fresh private segment - the usual pattern when parent and children share memory via fork:

```
use IPC::SysV qw(IPC_PRIVATE IPC_CREAT S_IRUSR S_IWUSR);

my $size = 4096;
my $shmid = shmget(IPC_PRIVATE, $size, IPC_CREAT | S_IRUSR | S_
    ↪IWUSR)
    or die "shmget: $!";
# children inherit $shmid across fork and attach via shmread/
    ↪shmwrite
```

Attach to a pre-existing segment by its well-known key. SIZE and mode bits are ignored in this form; pass 0 for both:

```
my $key    = 0xDEADBEEF;
my $shmid = shmget($key, 0, 0)
    or die "shmget: $!";
```

Create-or-open with IPC_CREAT. If the key already exists the existing shmid is returned; otherwise a new segment of SIZE bytes is created with the given mode:

```
use IPC::SysV qw(IPC_CREAT S_IRUSR S_IWUSR S_IRGRP);

my $shmid = shmget(0x1234, 8192, IPC_CREAT | S_IRUSR | S_IWUSR | S_
    ↳IRGRP)
    or die "shmget: $!";
```

Fail if a segment already exists, using IPC_EXCL. This is the safe pattern for a process that wants to be the unique creator:

```
use IPC::SysV qw(IPC_CREAT IPC_EXCL S_IRUSR S_IWUSR);
use Errno qw(EEXIST);

my $shmid = shmget(0x1234, 4096, IPC_CREAT | IPC_EXCL | S_IRUSR | S_
    ↳IWUSR);
if (!$shmid) {
    die "shmget: $!" unless $! == EEXIST;
    # another process won the race; fall back to plain lookup
    $shmid = shmget(0x1234, 0, 0) or die "shmget (lookup): $!";
}
```

Full create-use-destroy cycle with guaranteed cleanup via END:

```
use IPC::SysV qw(IPC_PRIVATE IPC_CREAT IPC_RMID S_IRUSR S_IWUSR);

my $shmid = shmget(IPC_PRIVATE, 1024, IPC_CREAT | S_IRUSR | S_IWUSR)
    or die "shmget: $!";

END {
    shmctl($shmid, IPC_RMID, 0) if defined $shmid;
}

shmwrite($shmid, "hello", 0, 5) or die "shmwrite: $!";
shmread($shmid, my $buf, 0, 5) or die "shmread: $!";
```

Edge cases

- **use IPC::SysV is effectively mandatory.** The flag constants (IPC_PRIVATE, IPC_CREAT, IPC_EXCL, IPC_RMID) and mode bits (S_IRUSR, S_IWUSR, S_IRGRP, ...) are not built-ins. Without the import you end up passing bare barewords that stringify to 0 and silently do the wrong thing.
- **Key 0 is not IPC_PRIVATE.** IPC_PRIVATE is defined as 0 on Linux, but passing a

literal 0 with no IPC_CREAT asks the kernel for the segment at key 0, which is the IPC_PRIVATE sentinel and yields EINVAL. Import the constant by name and use it.

- **SIZE is rounded up to a page boundary** by the kernel; the returned segment may be larger than requested. Ask the kernel for the effective size with `shmctl($shmid, IPC_STAT, $ds)` and unpack the `shmid_ds` structure if you need the real number.
- **Segment persists beyond process exit.** A crashed or killed creator leaves the segment in the kernel. Use `ipcs -m` to list orphaned segments and `ipcrm -m <shmid>` to remove them. Install an END block or a signal handler to call `shmctl IPC_RMID` on shutdown.
- **IPC_RMID is deferred, not immediate.** It marks the segment for destruction; actual removal happens once the attach count drops to zero. Safe to call right after `shmget` if you know no one else will attach.
- **Kernel limits.** SHMMNI caps the total number of segments system-wide (`/proc/sys/kernel/shmmni`). SHMMAX caps the size of one segment (`/proc/sys/kernel/shmmax`). On a shared host these are easy to hit; treat ENOSPC and EINVAL-from-large-size as configuration issues, not bugs.
- **SysV IPC is not available on every platform.** Upstream Perl built on a system without HAS_SHM dies with "System V IPC is not implemented on this machine". pperl targets Linux only, where SysV IPC is always present, so the call is always dispatched.

Differences from upstream

- Returns -1 rather than `undef` on failure. Both values are false under Perl's truth rules, so the idiomatic `shmget(...)` or `die "$!"` works unchanged, but code that distinguishes "not an identifier" via `defined $shmid` will read -1 as defined. Prefer checking for truth, or compare against -1 explicitly when you want to branch on the failure sentinel. The -1 is the raw `shmget(2)` return value, which is the documented error sentinel for that syscall.

See also

- `shmctl` - control operations on the segment `shmget` returned; use IPC_RMID to delete it, IPC_STAT to inspect size and permissions
- `shmread` - read SIZE bytes from the segment into a Perl scalar
- `shmwrite` - write a Perl scalar into the segment at an offset
- `msgget` - the equivalent allocation step for SysV message queues when you want a queue, not a flat buffer
- `semget` - the equivalent for SysV semaphores when the goal is synchronisation rather than shared data
- `IPC : SysV` - source of the `IPC_*` and `S_*` flag constants `shmget` requires; no local reference page, see upstream POD

SysV IPC

shmread

Copy bytes out of a System V shared memory segment into a Perl scalar.

shmread attaches to the shared memory segment identified by ID, reads SIZE bytes starting at byte offset POS, writes them into VAR, and detaches. VAR is assigned to as a whole - whatever it held before is replaced by exactly SIZE bytes of data. The variable is marked **tainted** under -T, because its contents came from outside the program.

Synopsis

```

use IPC::SysV qw(IPC_PRIVATE S_IRUSR S_IWUSR);
my $id = shmget(IPC_PRIVATE, 4096, S_IRUSR | S_IWUSR)
    or die "shmget: $!";
shmread($id, my $buf, 0, 128)
    or die "shmread: $!";

```

What you get back

Truthy on success, false on failure with \$! set. On success VAR has been overwritten with exactly SIZE bytes; no short reads - either the whole range is copied in or the call fails. On failure VAR's contents are unspecified; don't rely on them.

Always check the return value. A bad ID, an out-of-range POS/ SIZE, or a segment the process lacks permission to attach all surface here as a false return:

```

shmread($id, my $buf, $pos, $size)
    or die "shmread(id=$id, pos=$pos, size=$size): $!";

```

Global state it touches

- \$! - set on failure (any of the underlying shmat/memcpy/shmdt errors surface here).
- Taint flag on VAR - set under -T regardless of how VAR was declared, because the bytes originate outside the program.

Examples

Read a fixed-size record out of a segment:

```

use IPC::SysV qw(IPC_PRIVATE S_IRUSR S_IWUSR);
my $id = shmget(IPC_PRIVATE, 1024, S_IRUSR | S_IWUSR) or die
    ↪ "shmget: $!";
shmwrite($id, "hello, world", 0, 12) or die
    ↪ "shmwrite: $!";
shmread($id, my $buf, 0, 12) or die
    ↪ "shmread: $!";
print $buf, "\n"; # hello, world

```

Read at an offset. POS and SIZE are in bytes, not elements - the segment has no notion of records:

```
shmread($id, my $chunk, 256, 64)
    or die "shmread: $!";                                     # bytes 256..
↪319
```

Unpack a binary record straight out of shared memory:

```
shmread($id, my $raw, 0, 16) or die "shmread: $!";
my ($magic, $len, $flags) = unpack "a4 N N", $raw;
```

Reading tainted data under -T. The assignment is guaranteed to produce a tainted scalar even if VAR already held an untainted value:

```
# perl -T
shmread($id, my $buf, 0, $n) or die "shmread: $!";
# $buf is tainted; untaint via a regex match before using
# it in a sensitive context.
```

Edge cases

- **VAR is assigned to, not appended to.** Any previous value is discarded. Use a fresh lexical if you need both old and new data.
- **SIZE is exact, not a maximum.** shmread does not do short reads; requesting more bytes than the segment holds from POS onward fails rather than returning what fits.
- **POS and SIZE must lie within the segment** declared at shmget time. POS + SIZE past the segment's end fails with EINVAL.
- **Zero SIZE** produces an empty string in VAR and succeeds, assuming POS itself is valid.
- **Permissions.** The calling process must have read permission on the segment, as dictated by the mode bits passed to shmget (and, where applicable, by the segment's owner and current uid/gid).
- **VAR need not be pre-sized.** Unlike sysread, you do not substr-pad or preallocate VAR - shmread assigns the result directly.
- **Taint.** Under -T, VAR is tainted on success regardless of what it held before. Launder through a capture-group match (\$data =~ /\A(.*)\z/s and \$data = \$1) only once you've validated the bytes are safe for the sensitive operation you're about to perform.
- **Not every platform has SysV IPC.** On systems without shmat(2) the call fails at runtime (typically ENOSYS); portable code should guard with an eval or feature probe.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `shmwrite` - the inverse: copy a string into a segment at a given offset
- `shmget` - obtain or create the segment ID that `shmread` operates on
- `shmctl` - query segment metadata (`IPC_STAT` fills in an `shmid_ds` you can use to bound-check `POS + SIZE`) or remove the segment when done
- `sysread` - the file-descriptor analogue; same “bytes into a scalar” shape, different source
- `unpack` - turn the raw bytes returned in `VAR` into typed fields
- `IPC::SysV` - the module that supplies `IPC_PRIVATE`, `S_IRUSR`, and the other constants you’ll pass to `shmget`

SysV IPC

shmwrite

Copy bytes into a System V shared memory segment.

`shmwrite` attaches to the shared memory segment identified by `ID`, copies up to `SIZE` bytes from `STRING` starting at byte offset `POS` inside the segment, and detaches. If `STRING` is longer than `SIZE`, only the first `SIZE` bytes are written; if it is shorter, the remainder of the `SIZE`-byte window is zero-filled with NUL bytes. The segment ID is the value returned by an earlier `shmget` call (or obtained from another process via some out-of-band channel).

Synopsis

```
shmwrite ID, STRING, POS, SIZE
```

What you get back

True on success, false on error, with `$!` set to the reason (typically `EINVAL`, `EACCES`, or `EIDRM`). Unlike most I/O built-ins the return value is a plain boolean - there is no byte-count variant, because the count is always exactly `SIZE` when the call succeeds.

```
shmwrite($id, $payload, 0, 4096)
    or die "shmwrite($id) failed: $!";
```

Global state it touches

- `$!` is set on failure to the underlying `shmat` / `memcpy` / `shmdt` errno.
- No other special variables are read or written. Unlike `shmread`, `shmwrite` does not taint anything - it is the *source* of bytes, not a sink.

Examples

Publish a fixed-size record at the start of a segment:

```
use IPC::SysV qw(IPC_PRIVATE S_IRUSR S_IWUSR);
my $id = shmget(IPC_PRIVATE, 4096, S_IRUSR | S_IWUSR)
    or die "shmget: $!";
shmwrite($id, "hello, world", 0, 4096)
    or die "shmwrite: $!";
```

A STRING shorter than SIZE is NUL-padded to fill the window - useful for clearing stale trailing bytes from a previous writer:

```
shmwrite($id, "short", 0, 64);    # 5 bytes of "short", then 59 NULs
```

A STRING longer than SIZE is truncated; the tail is silently discarded, so size your window to the largest record you ever intend to publish:

```
shmwrite($id, "x" x 10_000, 0, 128);    # only the first 128 x's land
```

Writing to a non-zero offset lets multiple fixed-size records share one segment. Pair it with a matching `shmread` on the reader side:

```
my $slot = 3;
my $len  = 64;
shmwrite($id, $record, $slot * $len, $len)
    or die "shmwrite slot $slot: $!";
```

Check the return value - a segment that has been marked for deletion with `IPC_RMID` continues to accept writes from processes already attached, but fails for new `shmwrite` calls with `EIDRM`:

```
unless (shmwrite($id, $msg, 0, $SIZE)) {
    warn "segment $id gone: $!";
    # reacquire via shmget() and retry
}
```

Edge cases

- **Kernel must be built with SysV IPC.** On systems without SysV shared memory `shmwrite` croaks with `shmwrite not implemented`. This is a compile-time property of the kernel / libc, not a runtime condition you can probe for.
- **SIZE of 0** is accepted by the kernel and writes nothing; `shmwrite` returns true. Use it only when you genuinely mean “no-op”, not as an error signal.
- **POS + SIZE must fit inside the segment.** Overrunning the segment boundary fails with `EINVAL` and leaves the segment untouched - the copy is not partial.
- **Binary payloads are fine.** STRING is treated as a byte string; embedded NULs are written literally. If the string carries the UTF-8 flag, the raw internal bytes are what hits the segment - use `pack` or `Encode::encode` first if you need a specific encoding on the wire.

- **No locking.** The kernel guarantees that each `shmwrite` call is atomic with respect to `shmat` / `shmdt`, but two concurrent writers can still interleave byte ranges within the segment. Coordinate with `semop` or a user-space lock if multiple processes write the same region.
- **Permissions** come from the segment's mode at `shmget` time. Attaching read-only and calling `shmwrite` fails with `EACCES`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `shmread` - inverse operation; copy bytes *out* of a segment into a Perl scalar
- `shmget` - obtain the ID that `shmwrite` writes into
- `shmctl` - inspect segment attributes, change permissions, or mark a segment for deletion
- `semop` - the usual companion primitive for serialising access to a shared segment
- `pack` - build the fixed-layout byte string you hand to `shmwrite` when the segment holds structured records
- `IPC::SysV` - constants (`IPC_PRIVATE`, `S_IRUSR`, `IPC_RMID`) used to construct the arguments to `shmget` and `shmctl`

Sockets

shutdown

Shut down one direction of a socket connection, or both.

`shutdown` tells the kernel to stop a socket doing further reads, further writes, or both - without closing the file descriptor. Unlike `close`, the effect is immediate and propagates through every forked copy of the descriptor, not just this process's. `HOW` has the same meaning as the argument to the `shutdown(2)` syscall.

Synopsis

```
shutdown SOCKET, HOW
```

What you get back

1 on success. On failure, `undef` if `SOCKET` is not a valid filehandle, or `0` with `$!` set for any other error (the socket is not connected, already shut down in that direction, etc.).

Two distinct failure returns, so a three-way test is sometimes warranted:

```
my $r = shutdown($sock, 1);
if      (!defined $r) { die "bad filehandle" }
elsif  (!$r)         { die "shutdown failed: $!" }
```

The HOW argument

HOW is a small integer matching the POSIX SHUT_* constants:

- 0 - stop reading. The peer can still send, but data will be discarded by the kernel and read / `sysread` on this end will return end-of-file.
- 1 - stop writing. Queued output is flushed, then the peer sees end-of-file on its read side. `print` / `syswrite` on this end will fail and raise SIGPIPE.
- 2 - stop both directions. Equivalent to calling shutdown with 0 and then 1.

The symbolic names SHUT_RD, SHUT_WR, SHUT_RDWR live in `Socket` if you prefer readable code:

```
use Socket qw(SHUT_WR);
shutdown $sock, SHUT_WR;
```

shutdown vs close

`close` drops the file descriptor in the current process only. Other processes that inherited the descriptor through fork keep their copies open, and the peer does not see end-of-file until the last copy is closed.

`shutdown` acts on the underlying socket, not on the descriptor. After `shutdown($sock, 1)` every forked copy sees the write half closed and the peer reads end-of-file right away - even though the descriptor itself is still open and can still be read from.

That is the usual reason to reach for shutdown: a parent and child share a socket, the child has finished sending, and the parent must keep reading replies. `close` in the child would be invisible to the peer; `shutdown($sock, 1)` is not.

Half-close for request/response protocols

A client that sends a complete request and then needs the server to know there is no more input uses shutdown with HOW = 1:

```
use IO::Socket::INET;
my $sock = IO::Socket::INET->new(PeerAddr => "example.com:80")
    or die "connect: $!";

print $sock "GET / HTTP/1.0\r\nHost: example.com\r\n\r\n";
shutdown $sock, 1;          # signal end-of-request to the server

while (my $line = <$sock>) {
    print $line;
}
close $sock;
```

Without the shutdown, servers that read until end-of-input would block waiting for more request data.

Examples

Stop reading but keep writing - the peer may continue to send, we just will not look at it:

```
shutdown $sock, 0;
```

Stop writing, keep reading - typical half-close after finishing a request:

```
shutdown $sock, 1;
```

Full shutdown, then close the descriptor:

```
shutdown $sock, 2;
close $sock;
```

Using `Socket` constants for readability:

```
use Socket qw(SHUT_RD SHUT_WR SHUT_RDWR);
shutdown $sock, SHUT_WR;
```

Check the specific error when a shutdown fails on an already-closed direction:

```
unless (shutdown $sock, 1) {
    warn "shutdown write failed: $!";
}
```

Edge cases

- **Writing after shutdown(\$sock, 1)** fails and raises SIGPIPE. The default action terminates the process; install `$SIG{PIPE} = 'IGNORE'` (or a handler) to convert it into a normal write failure you can check.
- **Reading after shutdown(\$sock, 0)** returns end-of-file immediately, not an error. Any data already queued in the kernel's receive buffer is discarded.
- **Not a socket:** calling shutdown on a regular filehandle sets `$!` to `ENOTSOCK` and returns `0`.
- **Not connected:** shutdown on a socket that was never connected (a fresh `socket` result, or a listening socket) sets `$!` to `ENOTCONN` and returns `0`.
- **Double shutdown:** shutting down the same half twice typically returns `0` with `ENOTCONN`. Portable code ignores the second failure rather than treating it as fatal.
- **Closed filehandle:** returns `undef`; under `use warnings` a `shutdown()` on `closed socket` warning is emitted.
- **Descriptor is not closed:** shutdown disables the directions you asked for but leaves the file descriptor open. Call `close` (or let the handle go out of scope) when you are actually done with it.
- **Shared descriptors across fork:** this is the case shutdown is designed for. All processes holding the descriptor see the shutdown take effect immediately; they do not get separate views.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `close` - release the descriptor in this process; does not affect forked copies and does not signal end-of-file to the peer until the last copy closes
- `socket` - create the socket that shutdown operates on
- `connect` - establish the connection; shutdown on an unconnected socket fails with ENOTCONN
- `accept` - produces the connected socket on the server side that typically gets half-closed with shutdown
- `Socket` - exports SHUT_RD, SHUT_WR, SHUT_RDWR and the rest of the socket-level constants
- `$!` - system error string to check after a `0` return

Numeric functions

`sin`

Return the sine of a number given in radians.

`sin` is the standard trigonometric sine: it takes an angle measured in **radians** and returns the ratio of the opposite side to the hypotenuse of the corresponding right triangle. If `EXPR` is omitted, `sin` operates on `$_`. The result is always a floating-point number in the closed interval `[-1, 1]`.

Synopsis

```
sin EXPR
sin
```

What you get back

A double-precision float in `[-1, 1]`. Perl forwards the call to the platform's C `sin(3)` routine, so the precision and rounding behaviour match your libm. Special values follow IEEE 754: `sin(0)` is exactly `0`, `sin("inf")` and `sin("nan")` are both NaN.

Radians, not degrees

The single most common mistake with `sin` is feeding it degrees. A full turn is $2 * \pi$ radians, not 360. Convert once and keep the converted value:

```
use POSIX qw(acos);
my $pi = acos(-1);           #  $\pi$  to full double precision

sub deg2rad { $_[0] * $pi / 180 }
```

(continues on next page)

(continued from previous page)

```
print sin( deg2rad(30) ), "\n";      # 0.5
print sin( 30 ), "\n";              # -0.988031624092862 - wrong if
↪you meant degrees
```

exports `deg2rad` and `rad2deg` if you would rather not write the conversion yourself.

Examples

$\sin(\pi/2)$ is mathematically exactly 1, but because π is not representable in binary floating-point the computed result falls a few ulps short:

```
use POSIX qw(acos);
my $pi = acos(-1);

printf "%.17f\n", sin($pi / 2);      # 1.00000000000000000
printf "%.17f\n", sin($pi);         # 0.00000000000000012 - not
↪exactly 0
```

Treat any “should be zero” trig result as approximately zero. Compare with a tolerance, not with `==`.

Degree-to-radian conversion inline, without pulling in a module. 3.14159 is accurate to six digits - fine for plotting, not for numeric analysis:

```
for my $deg (0, 30, 45, 60, 90) {
    printf "sin(%2d°) = %.4f\n",
           $deg, sin($deg * 3.14159 / 180);
}
# sin( 0°) = +0.0000
# sin(30°) = +0.5000
# sin(45°) = +0.7071
# sin(60°) = +0.8660
# sin(90°) = +1.0000
```

For work that needs full double precision, replace 3.14159 with `acos(-1)` or `atan2(1, 1) * 4`.

Numerical derivative of `sin` by finite differences. The symmetric two-point formula $(f(x+h) - f(x-h)) / (2h)$ should recover `cos(x)`:

```
my $x = 1.0;
my $h = 1e-6;
my $deriv = (sin($x + $h) - sin($x - $h)) / (2 * $h);

printf "derivative: %.10f\n", $deriv;  # 0.5403023059
printf "cos(%.1f):   %.10f\n", $x, cos($x);
```

Picking `$h` is a trade-off: too large and truncation error dominates, too small and subtraction cancellation destroys the result. 1e-6 is a reasonable default for double precision.

Walk the unit circle and print `(cos  $\theta$ , sin  $\theta$ )` at eight evenly spaced angles:

```
use POSIX qw(acos);
my $pi = acos(-1);

for my $k (0 .. 7) {
    my $theta = $k * $pi / 4;
    printf "%.3f  %+.3f  %+.3f\n",
        $theta, cos($theta), sin($theta);
}
```

Inverse sine (arcsine) via the identity from perlfunc:

```
sub asin { atan2( $_[0], sqrt(1 - $_[0] * $_[0]) ) }

print asin(0.5), "\n";           # 0.523598775598299 (=  $\pi/6$ )
```

Edge cases

- **Non-numeric arguments coerce to 0.** `sin("hello")` returns 0 because the string is numified to 0 first. Under `use warnings` this produces an `Argument "hello" isn't numeric in sin` warning. Validate input with `looks_like_number` from `Scalar::Util` if the source is untrusted.
- **Very large arguments lose precision.** `sin` first reduces its argument modulo 2π , and for $|x|$ on the order of $1e16$ the nearest representable double differs from x by more than π . The result is still a number in $[-1, 1]$, but it is essentially random. Reduce the angle yourself before the call if you are accumulating a phase over many iterations.
- **undef coerces to 0** and returns 0, with the usual uninitialized warning under `use warnings`.
- **IEEE specials:** `sin("inf")`, `sin("-inf")`, and `sin("nan")` all return NaN. Compare with `$result != $result` (the only value not equal to itself) to detect NaN, or use `POSIX::isnan`.
- **Default argument.** `sin`; with no argument reads `$_`. Inside a `while (<>)` loop this operates on the current line, which is almost always a bug.
- **No complex-number support** in the built-in. For complex arguments use `,` which overloads `sin` via operator overloading:

```
use Math::Complex;
my $z = cplx(1, 2);
print sin($z), "\n";           # 3.16577851321617+1.
                                ↪ 9596010414216i
```

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `cos` - cosine of an angle in radians; the companion function most often used alongside `sin`
- `atan2` - two-argument arctangent, the usual building block for inverse trig (`asin`, `acos`) in pure Perl
- `sqrt` - square root; appears in the classical `asin` identity `asin(x) = atan2(x, sqrt(1 - x*x))`
- - `asin`, `acos`, `atan`, hyperbolic variants, `deg2rad/rad2deg`, and great-circle helpers
- - overloads `sin` (and the rest of the trig family) for complex arguments
- POSIX - exposes `asin` directly, plus `isnan/isinf` for detecting the non-finite results discussed above

Processes

sleep

Pause the process for a number of whole seconds.

`sleep` suspends execution of the current process for `EXPR` seconds, or - with no argument - until a signal wakes it. The argument is truncated to an integer; sub-second delays require a different tool (see below). The return value is the number of seconds actually slept, which can be less than requested if a signal interrupted the call.

Synopsis

```
sleep EXPR
sleep
```

What you get back

An integer: the number of seconds the process was actually suspended. On an uninterrupted call this equals the (truncated) argument. If a signal delivered during the sleep unblocks the call early, the return value is the shorter, real elapsed time. For bare `sleep`, the value is whatever the implementation slept before a signal arrived.

```
my $slept = sleep 10;
warn "woken early after $slept s" if $slept < 10;
```

Argument handling

- **Positive integer** - the normal case. Sleep that many seconds.
- **Non-integer** (e.g. 2.9) - the fractional part is **discarded**. `sleep 2.9` sleeps two seconds, not nearly three. For fractional delays use `Time::HiRes::sleep`, `Time::HiRes::usleep`, or the four-argument form of `select` - see below.

- **Negative integer** - `sleep` does **not** sleep. It emits a warning, sets `$!` (`errno`), and returns zero.
- **Zero** - `sleep 0` is permitted, but still performs a call into the underlying platform sleep primitive with whatever side effects that entails. It is therefore **not** exactly identical to skipping the call. In particular, `sleep 0` is not the same as a bare `sleep` with no argument.
- **No argument** - `sleep` with no argument sleeps indefinitely, until a signal is delivered. This is distinct from `sleep 0`.

Interruption by signals

`sleep` returns early if the process receives a signal whose handler runs (or whose default action does not terminate the process). The canonical pattern for a bounded wait that can be cancelled from elsewhere is an `alarm` combined with a signal handler:

```
eval {  
    local $SIG{ALRM} = sub { die "Alarm!\n" };  
    sleep;  
};  
die $@ unless $@ eq "Alarm!\n";
```

Because `sleep` is itself commonly implemented on top of `alarm` on Unix systems, you generally **cannot** mix `alarm` and `sleep` calls in the same program without surprises. Pick one.

Sub-second delays

`sleep` is integer-only by design. For finer-grained waits use one of:

- `Time::HiRes::sleep($seconds)` - accepts a floating-point number of seconds. Since Perl 5.8 `Time::HiRes` ships with the core distribution.
- `Time::HiRes::usleep($microseconds)` - integer microseconds.
- The four-argument form of `select` with the first three arguments `undef`:

```
select undef, undef, undef, 0.25;    # sleep 250 ms
```

- `syscall` into `setitimer(2)` if your platform supports it and you need tight control over timers.

See also `POSIX::pause`, which blocks the process until any signal arrives - effectively a bare `sleep` without an implementation-defined maximum.

Examples

Fixed delay:

```
print "working...\n";  
sleep 3;  
print "done\n";
```

Poll loop with a back-off:

```
until (ready()) {
    sleep 1;
}
```

Detecting an early wake:

```
my $want = 30;
my $got  = sleep $want;
if ($got < $want) {
    warn "interrupted after $got of $want seconds\n";
}
```

Wait for a signal without burning CPU - bare sleep suspends until SIGALRM (or any other caught signal) arrives:

```
$SIG{USR1} = sub { warn "got USR1\n" };
sleep;                                     # wakes on USR1 delivery
```

Cancelling a long wait with alarm:

```
eval {
    local $SIG{ALRM} = sub { die "timeout\n" };
    alarm 5;
    sleep 3600;                      # at most 5 seconds really
    alarm 0;
};
alarm 0;
die "$@" if "$@" && "$@" ne "timeout\n";
```

Edge cases

- **Bare sleep does not truly sleep “forever”.** The implementation has a maximum - in practice, whatever the underlying C `sleep(3)` or equivalent accepts - but for all realistic purposes it blocks until a signal arrives. Treat it as an indefinite wait.
- **sleep 0 ≠ no call.** The platform `sleep(0)` syscall still runs. On most systems this yields the CPU briefly; do not rely on the specific behaviour across platforms.
- **sleep versus wall-clock time.** On older systems `sleep` could return up to a full second *early* due to coarse second counting. Modern systems sleep the full amount, but may overshoot: the scheduler may not wake your process immediately on a busy system. `sleep N` guarantees *at least* N seconds of suspension on modern kernels, not *exactly* N.
- **Negative arguments are not clamped.** They are a hard no-op that sets `$!`. If you compute the argument, validate it first.
- **Mixing with alarm is fragile.** The alarm timer and the sleep timer may be the same resource. Establish one discipline (either `alarm` + bare `sleep`, or plain `sleep N` without alarms) and stick to it within a program.

- **Signals that merely set a flag still wake `sleep`.** Any signal whose delivery causes a handler to run returns control to Perl; `sleep` then returns early even if the handler itself does nothing visible.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `alarm` - schedule a SIGALRM after a given number of seconds; the classic way to bound a `sleep` or any other blocking call
- `select` - in its four-argument form with an undefined `readfds/writefds/exceptfds` triple, provides sub-second sleeps
- `wait`, `waitpid` - block on child processes rather than on time
- `Time::HiRes::sleep` - floating-point seconds
- `Time::HiRes::usleep` - integer microseconds
- `POSIX::pause` - block until any signal arrives

Sockets

socket

Create a socket filehandle.

`socket` opens a socket of the kind described by `DOMAIN`, `TYPE`, and `PROTOCOL` and attaches it to `SOCKET`, which becomes a usable Perl filehandle. It is the thin Perl veneer over the `socket(2)` system call - nothing more, nothing less. The new socket is unbound and unconnected; you still need `bind`, `connect`, or `listen+accept` to do anything with it. Pull the symbolic constants in with `use Socket`; before the call - writing the raw integers is portable only by accident.

Synopsis

```
use Socket;  
socket SOCKET, DOMAIN, TYPE, PROTOCOL
```

What you get back

1 on success, `undef` on failure (with `$!` set to the `errno` from the underlying `socket(2)`). On success `SOCKET` is now a filehandle you can `bind`, `connect`, `send / recv` on, or read and write with the ordinary I/O operators once it reaches a connected state. The return value is worth checking on every call - a failure here means you have nothing useful to pass to the next socket call, which will then fail in a more confusing way.

```
socket my $sock, PF_INET, SOCK_STREAM, getprotobyname("tcp")  
    or die "socket: $!";
```

SOCKET can be a bareword filehandle (SOCK), a glob (*SOCK), or an undefined scalar that will be auto-vivified into an anonymous filehandle reference. The auto-vivified form is the modern idiom and the only one that scopes cleanly with `my`.

Global state it touches

- `$!` - set to the `errno` from `socket(2)` on failure.
- `$_^F` - the system-file-descriptor threshold. On systems with `close-on-exec`, the new descriptor gets `FD_CLOEXEC` set when its number exceeds `$_^F` (default 2, so descriptors 0–2 stay inheritable and everything else is auto-closed across `exec`). Raise it before `socket` if you specifically want the new socket to survive an `exec`.

Examples

A client socket for a TCP connection. Create, connect, talk:

```
use Socket;
socket my $sock, PF_INET, SOCK_STREAM, getprotobyname("tcp")
    or die "socket: $!";
my $addr = pack_sockaddr_in(80, inet_aton("example.com"));
connect $sock, $addr or die "connect: $!";
syswrite $sock, "GET / HTTP/1.0\r\nHost: example.com\r\n\r\n";
```

A listening TCP server socket. Create, mark reusable, bind, listen:

```
use Socket;
socket my $srv, PF_INET, SOCK_STREAM, getprotobyname("tcp")
    or die "socket: $!";
setsockopt $srv, SOL_SOCKET, SO_REUSEADDR, pack("l", 1)
    or die "setsockopt: $!";
bind $srv, pack_sockaddr_in(8080, INADDR_ANY) or die "bind: $!";
listen $srv, SOMAXCONN or die "listen: $!";
```

A UDP socket - `SOCK_DGRAM` instead of `SOCK_STREAM`, and no `connect` is needed before `send` when you pass an explicit destination:

```
use Socket;
socket my $udp, PF_INET, SOCK_DGRAM, getprotobyname("udp")
    or die "socket: $!";
my $to = pack_sockaddr_in(53, inet_aton("1.1.1.1"));
send $udp, $query, 0, $to or die "send: $!";
```

A Unix-domain stream socket for local IPC. `PF_UNIX` with a filesystem path instead of an IP address:

```
use Socket;
socket my $sock, PF_UNIX, SOCK_STREAM, 0 or die "socket: $!";
connect $sock, pack_sockaddr_un("/run/app.sock") or die "connect: $!";
↪;
```

Letting `PROTOCOL` default to `0`. The kernel picks the default protocol for the domain/type pair - for `PF_INET` + `SOCK_STREAM` that is TCP, for `PF_INET` + `SOCK_DGRAM` that is UDP. The

`getprotobyname` form above is more explicit but the `0` form is equivalent and avoids one lookup:

```
use Socket;  
socket my $sock, PF_INET, SOCK_STREAM, 0 or die "socket: $!";
```

Edge cases

- **use Socket is not optional.** Without it, `PF_INET`, `SOCK_STREAM`, and friends are bareword strings, not integer constants, and the call fails at the syscall boundary with `Invalid argument`. The import is what turns them into numbers.
- **SOCKET is clobbered on success.** If the slot holds an open filehandle, it is closed first. There is no warning - the previous socket just silently goes away. Use a fresh `my` variable for each call.
- **Failure leaves SOCKET unchanged.** On `undef` return the scalar is neither opened nor modified; a prior value (including an earlier, still-open socket) is still there.
- **Descriptor limits.** `socket` is the point where `EMFILE` (per-process) and `ENFILE` (system-wide) descriptor-table exhaustion shows up. Long-running servers should treat these as recoverable rather than fatal.
- **EAFNOSUPPORT / EPROTONOSUPPORT / ESOCKTNOSUPPORT.** The kernel rejected the domain/type/protocol triple. Common cause: asking for `PF_INET6` on a kernel built without IPv6, or an invalid protocol number passed through from a stale `getprotobyname` lookup.
- **Non-blocking mode is a separate step.** `socket` returns a blocking descriptor. Use `fcntl` with `F_SETFL + O_NONBLOCK` (or the `IO::Socket` equivalent) if you need non-blocking I/O.
- **Close-on-exec is automatic past \$^F.** Perl sets `FD_CLOEXEC` on the new descriptor when its fd number is greater than `$^F`. A socket will therefore not survive an `exec` unless you lowered that threshold or clear the flag with `fcntl`.
- **Numeric domain/type values are not portable.** `2` happens to be `AF_INET` on Linux; it is something else elsewhere. Always go through the `Socket` module constants.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `bind` - attach a socket created by `socket` to a local address, required before `listen` or before sending from a known port
- `connect` - initiate an outgoing connection on a `SOCK_STREAM` socket, or set the default peer on a `SOCK_DGRAM` socket
- `listen` - mark a bound `SOCK_STREAM` socket as passive so `accept` can pull connections off it

- `accept` - return the next pending connection on a listening socket as a fresh file-handle
- `socketpair` - create a connected pair of sockets in one call; the usual choice when both ends live in the same process or parent/child
- `Socket` - the module that supplies `PF_INET`, `SOCK_STREAM`, `inet_aton`, `pack_sockaddr_in` and the rest of the constants and packers you pass to `socket`
- `$^F` - descriptor threshold that controls close-on-exec for the new socket

Sockets

socketpair

Create an unnamed, connected pair of sockets that talk to each other.

`socketpair` opens two fresh filehandles in `SOCKET1` and `SOCKET2` already wired together, so anything written to one comes out of the other. No addresses, no `bind`, no `listen`, no `accept`. The commonest use is talking to a child process created by `fork` - each side keeps one handle and `closes` the other.

Synopsis

```
use Socket;
socketpair(SOCKET1, SOCKET2, DOMAIN, TYPE, PROTOCOL) or die $!;
socketpair(my $s1, my $s2, AF_UNIX, SOCK_STREAM, PF_UNSPEC) or die
    $!;
```

What you get back

True on success, false on failure (with `$!` set). On success both filehandle arguments are populated with fresh handles; on failure they are left untouched. Always check the return value - every domain/type/protocol combination is platform-dependent.

```
socketpair(my $a, my $b, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
    or die "socketpair: $!";
```

The `DOMAIN`, `TYPE`, and `PROTOCOL` constants come from the `Socket` module. `AF_UNIX` with `SOCK_STREAM` and `PF_UNSPEC` is the portable default.

Global state it touches

- `$!` is set on failure to the system `errno`.
- `$^F` - the maximum system file descriptor below which close-on-exec is **not** set. On systems with `FD_CLOEXEC`, descriptors above `$^F` get close-on-exec automatically; below it, they survive `exec`. Default is 2 (stdin/stdout/stderr). Raise it before `socketpair` if you want the new handles to be inherited across `exec`.

Examples

Plain parent/child pipe using a stream socketpair. Each side closes the end it does not need, so EOF propagates correctly:

```
use Socket;

socketpair(my $child, my $parent, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
    or die "socketpair: $!";

my $pid = fork // die "fork: $!";
if ($pid == 0) {
    close $child;
    print $parent "hello from child\n";
    close $parent;
    exit;
}
close $parent;
chomp(my $msg = <$child>);          # "hello from child"
waitpid $pid, 0;
```

Bidirectional stream: both processes read *and* write on the same handle. No `shutdown` is needed because neither side is closing a direction early:

```
use Socket;

socketpair(my $a, my $b, AF_UNIX, SOCK_STREAM, PF_UNSPEC) or die $!;
my $pid = fork // die $!;
if ($pid == 0) {
    close $a;
    while (my $line = <$b>) {
        chomp $line;
        print $b "echo: $line\n";
    }
    exit;
}
close $b;
print $a "ping\n";
chomp(my $reply = <$a>);           # "echo: ping"
close $a;
waitpid $pid, 0;
```

Emulating pipe with socketpair. `shutdown` turns a bidirectional socket into a one-way channel:

```
use Socket;
socketpair(my $rdr, my $wtr, AF_UNIX, SOCK_STREAM, PF_UNSPEC) or die $!;
shutdown($rdr, 1);                # reader cannot write
shutdown($wtr, 0);                # writer cannot read
```

Datagram pair - messages preserve boundaries, so a single `recv` returns exactly one

send:

```
use Socket;
socketpair(my $a, my $b, AF_UNIX, SOCK_DGRAM, PF_UNSPEC) or die $!;
send($a, "one", 0) or die $!;
send($b, "two", 0) or die $!;
recv($b, my $m1, 64, 0);           # "one"
recv($a, my $m2, 64, 0);           # "two"
```

Edge cases

- **AF_UNIX is the only portable domain.** On most systems `socketpair` only works with `AF_UNIX`; other families return `EAFNOSUPPORT`. Use the constant, not a hard-coded 1.
- **Unbuffered output.** Stream `socketpair` handles buffer like any other filehandle. Set `$|` on the writer (after `selecting` it) or the reader will block waiting for output that is still in the parent's buffer. `autoflush` from `IO::Handle` is the usual form:

```
$parent->autoflush(1);
```

- **Close both ends or EOF never arrives.** A stream `socketpair` delivers EOF on a reader only after *every* writable copy of the peer handle is closed. A child that forgets to close the parent's end will hang the parent on `<$handle>` forever.
- **Close-on-exec via `$^F`.** Filehandles whose numeric descriptor is greater than `$^F` get `FD_CLOEXEC` on platforms that support it, so they vanish across `exec`. Raise `$^F` before the call if the handle must survive an `exec`.
- **Unimplemented on the platform.** If the underlying `socketpair(2)` syscall does not exist, perl raises an exception rather than returning false. Trap with `eval` if you need to fall back.
- **Datagram vs stream.** `SOCK_STREAM` is byte-oriented (no message boundaries, use newline framing or length prefixes). `SOCK_DGRAM` preserves message boundaries but loses them silently if the receive buffer is smaller than the message.
- **PROTOCOL.** For `AF_UNIX` the protocol is ignored; `PF_UNSPEC` (or `0`) is conventional. Specifying a protocol that does not match the domain/type combination returns `EPROTONOSUPPORT`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `socket` - create a single socket when only one end is yours (the other comes from the network via `accept` or `connect`)
- `pipe` - unidirectional byte channel; lighter weight than `socketpair` but cannot do bidirectional traffic on one call
- `send` / `recv` - message-oriented I/O, required for `SOCK_DGRAM` `socketpairs`

- `fork` - the typical partner: one `socketpair` before the fork gives parent and child a ready-made conduit
- `Socket` - source of `AF_UNIX`, `SOCK_STREAM`, `SOCK_DGRAM`, `PF_UNSPEC` and the other constants `socketpair` expects
- `$^F` - threshold above which new descriptors get close-on-exec; matters when the handles must survive `exec`

Lists

sort

Sort a list, returning the sorted list.

`sort` takes a list and returns a new list with the elements rearranged in order. With no comparator it uses standard string comparison. With a `BLOCK` or a `SUBNAME` you supply the comparison rule as a small function that is called for each pair of elements and must return a negative, zero, or positive value the way `cmp` and `<=>` do.

Sorting is stable - elements that compare equal keep their original relative order. The algorithm is mergesort.

Synopsis

```
sort LIST
sort BLOCK LIST
sort SUBNAME LIST
```

Three idiomatic forms:

```
my @out = sort @in;                # default: string cmp, _
↪ascending
my @out = sort { $a <=> $b } @in;   # numeric, ascending
my @out = sort by_name @in;        # named comparator sub
```

What you get back

A **new** sorted list. `sort` does **not** modify the input in place. To replace an array with its sorted version you assign back:

```
@a = sort @a;
```

In list context the return is the sorted list. In scalar context the return value is undefined - do not use scalar `sort`.

The returned list contains **aliases** into the original list, exactly like a `foreach` index variable. Modifying an element of the result inside a later `foreach`, `map`, or `grep` therefore modifies the original element too. This is almost never what you want; treat the result as read-only.

Global state: \$a and \$b

Inside the comparator BLOCK or SUBNAME, the two elements being compared are exposed as the package globals \$a and \$b. These are not parameters and they are not lexicals - they are real package variables that sort localises for you around the call.

- \$a and \$b live in **the package that called sort**. In main, that is \$main::a and \$main::b; in Foo, \$Foo::a and \$Foo::b.
- **Never declare my \$a or my \$b** anywhere the sort block can see. A lexical \$a or \$b shadows the package global; the block then compares two unrelated variables and the sort silently produces garbage. Under use warnings you get no warning for this.
- If you genuinely need \$a / \$b as lexicals elsewhere, refer to the sort variables by full name inside the block:

```
print sort { $::a cmp $::b }      qw(A C E G B D F H);
print sort { our $a cmp our $b }  qw(A C E G B D F H);
```

- A comparator defined with the prototype (\$\$) (or an equivalent signature attribute) receives the pair in @_ instead of \$a and \$b. This is slower than the \$a/\$b convention but decouples the sub from the caller's package.

Both elements are passed **by reference**; mutating \$a or \$b in the block mutates the original list. Don't.

Examples

Default string sort:

```
my @sorted = sort qw(banana apple cherry);
# ('apple', 'banana', 'cherry')
```

Numeric ascending - the default sort @nums would sort "10" before "2" because comparison is by string:

```
my @nums = sort { $a <=> $b } (10, 2, 33, 4);
# (2, 4, 10, 33)
```

Numeric descending - swap \$a and \$b:

```
my @nums = sort { $b <=> $a } (10, 2, 33, 4);
# (33, 10, 4, 2)
```

Multi-key sort with a tiebreaker. || runs the second comparator only when the first returns 0:

```
my @people = sort {
    $a->{last} cmp $b->{last}
  || $a->{first} cmp $b->{first}
} @records;
```

Sort a hash's keys by the associated value:

```
my %age = (alice => 30, bob => 25, carol => 40);
my @by_age = sort { $age{$a} <=> $age{$b} } keys %age;
# ('bob', 'alice', 'carol')
```

Schwartzian Transform - when the sort key is expensive to compute, precompute it once per element, sort the [key, value] pairs, then strip the keys. Three stages, read bottom-up:

```
my @sorted =
    map { $_->[1] }                # 3. strip key
    sort { $a->[0] <=> $b->[0] }    # 2. sort by key
    map { [ expensive_key($_), $_ ] } # 1. attach key
    @input;
```

Named comparator sub. The sub reads \$a and \$b from the caller's package, so it must be in (or referenced from) that package:

```
sub by_name { $a->{name} cmp $b->{name} }

my @sorted = sort by_name @people;
```

Prototyped comparator - arguments arrive in @_, so it works from any package:

```
sub numeric ($$) { $_[0] <=> $_[1] }

my @sorted = sort numeric @nums;
```

Edge cases

- **\$a and \$b are package globals, not lexicals.** A stray my \$a in scope silently breaks every sort in that scope. If you must, use our \$a or \$: : a inside the block.
- **<=> is numeric, cmp is string.** Using the wrong one is the most common sort bug. sort { \$a cmp \$b } (10, 2, 33) yields (10, 2, 33) - lexicographic, "10" before "2". Use <=> for numbers.
- **Empty list returns an empty list.** sort () is (). No error.
- **Single element is returned unchanged.** No comparisons happen.
- **Comparator must be consistent.** If it sometimes says \$x < \$y and sometimes says the opposite for the same pair, the result is undefined. The comparator must return a total ordering: negative, zero, or positive, stably, for every call.
- **NaN poisons <=>.** \$a <=> \$b returns undef if either operand is NaN, which sort then treats as 0 - equal - giving an arbitrary placement. Filter first:

```
my @clean = sort { $a <=> $b } grep { $_ == $_ } @input;
```

- **Stable sort.** Equal elements keep their input order. Rely on this for multi-pass sorts (sort by secondary key first, then by primary key). use sort 'stable' is the default and a no-op; use sort '_mergesort' / '_qsort' pragmas are historical and have no effect on current Perl.

- **sort is not in-place.** `sort @a` does not touch `@a`. Use `@a = sort @a` to replace it. Assigning to `sort`'s result list aliases back into the original, so `(sort @a)[0] = ...` modifies `@a`'s smallest element.
- **No loop control out of the block.** `last`, `next`, `redo`, `return`, and `goto LABEL` do not work inside a sort comparator
 - the block is not a loop body.
- **Parser trap: sorting a function's return value.** `sort` takes a LIST, so a following bareword can be mistaken for a SUBNAME. To sort the result of `find_records(@key)`:

```
my @out = sort { $a cmp $b } find_records @key; # OK
my @out = sort +find_records(@key);           # OK
my @out = sort(find_records(@key));           # OK
```

To use `find_records` as the comparator over `@key` instead:

```
my @out = sort find_records @key;               # comparator
↪ form
my @out = sort { find_records() } @key;         # block form
```

- **Locale-aware sorting.** Under `use locale` (without `:not_characters`) the default string comparison follows the current collation locale instead of codepoint order.
- **XSUB comparators.** If the comparator is an XSUB, the pair is passed on the argument stack the way XS functions normally receive arguments; `$a` and `$b` are not set.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `reverse` - reverses a list; combine with `sort` for descending sorts when the comparator is hard to invert by swapping `$a` and `$b`
- `map` - the other half of the Schwartzian Transform; use to attach and strip sort keys around a sort
- `grep` - prefilter the list (e.g. remove NaN or `undef`) before handing it to `sort`
- `$a` - the first element in the comparison pair, aliased as a package global for the duration of the sort
- `$b` - the second element in the comparison pair, same lifetime and scoping rules as `$a`
- `<=>` - numeric three-way comparison; the usual operator inside a numeric sort block
- `cmp` - string three-way comparison; the usual operator inside a string sort block

Arrays

splice

Remove and/or replace a slice of an array in place, and return the removed elements.

splice is the general-purpose array surgeon. Every other array mutator - [push](#), [pop](#), [shift](#), [unshift](#), single-element assignment - can be written as a splice call. Pick the position with OFFSET, pick how many elements to cut with LENGTH, and optionally supply a LIST of replacements. The array grows or shrinks to fit, and you get back whatever was cut out.

Synopsis

```
splice @array;                # remove everything
splice @array, $offset;       # remove from $offset to
    ↳the end
splice @array, $offset, $length; # remove $length
    ↳elements
splice @array, $offset, $length, @list; # remove and replace
    ↳with @list
```

What you get back

- **List context:** the list of elements that were removed, in their original order. Empty list if nothing was removed.
- **Scalar context:** the *last* element removed, or `undef` if nothing was removed. This is rarely what you want - reach for list context unless you specifically need only the tail.
- **Void context:** nothing; use this form when you only care about the in-place mutation.

Examples

Basic removal - cut two elements starting at index 1:

```
my @a = ('a', 'b', 'c', 'd', 'e');
my @gone = splice @a, 1, 2;
# @a == ('a', 'd', 'e')
# @gone == ('b', 'c')
```

Insert without removing - LENGTH is 0, so nothing comes out and the LIST is spliced in at OFFSET:

```
my @a = (1, 2, 5, 6);
splice @a, 2, 0, 3, 4;
# @a == (1, 2, 3, 4, 5, 6)
```

Replace a fragment - LENGTH and replacement size need not match. Here three elements leave, one arrives:


```
my @a = ('x', 'y', 'z', 'w', 'v');
my @gone = splice @a, 1, 3, 'Y';
# @a      == ('x', 'Y', 'v')
# @gone   == ('y', 'z', 'w')
```

Remove the trailing tail - omit LENGTH entirely to cut from OFFSET to the end:

```
my @a = (1, 2, 3, 4, 5);
my @tail = splice @a, 3;
# @a      == (1, 2, 3)
# @tail   == (4, 5)
```

Negative OFFSET counts from the end. Pop the last element:

```
my @a = ('a', 'b', 'c');
my $last = splice @a, -1;
# @a      == ('a', 'b')
# $last    == 'c'
```

Classic idiom - consume an array \$n elements at a time:

```
while (my @chunk = splice @queue, 0, $n) {
    process(@chunk);
}
```

All the common array primitives written as splice (assuming \$#a >= \$i):

```
push    @a, $x, $y;    # splice @a, scalar(@a), 0, $x, $y
pop      @a;           # splice @a, -1
shift    @a;           # splice @a, 0, 1
unshift  @a, $x, $y;    # splice @a, 0, 0, $x, $y
$a[$i] = $y;           # splice @a, $i, 1, $y
```

Edge cases

- **Negative OFFSET** counts from the end: -1 is the last element, -2 the one before, and so on. If the magnitude exceeds the array length, Perl issues a warning and clamps to 0.
- **Negative LENGTH** leaves that many elements at the end of the array untouched. `splice @a, 2, -1` removes from index 2 up to but not including the final element.
- **OFFSET past the end**: with an explicit LENGTH, Perl issues a warning and splices at the end of the array (equivalent to `splice @a, scalar(@a), 0, LIST`). Without a LENGTH, nothing is removed.
- **LENGTH past the end**: silently clamped. `splice @a, 1, 999` simply removes everything from index 1 onward.
- **Both OFFSET and LENGTH omitted**: removes every element. The array is left empty.

- **Replacement list size differs from LENGTH:** the array grows or shrinks to match. There is no requirement that `scalar(@list)` equal `LENGTH` - that asymmetry is the whole point of `splice`.
- **Array reference:** dereference inline. `splice @$ref, 0, 1` and `splice @{$obj->{items}} , -2` both work.
- **Splicing a slice is not supported:** `splice @a[1..3], 0, 1` is a syntax error. `splice` requires a named array or dereferenced array, not a list expression. To operate on a range, pass the bounds as `OFFSET` and `LENGTH` instead.
- **Void context is fine.** If the return value is not needed, call `splice` as a statement - no allocation of the removed-elements list happens in that case.
- **Scalar expressions as the first argument:** not allowed. An experimental feature permitted this from 5.14 through 5.22 and was removed in 5.24. The first argument must be an array (or dereferenced arrayref).

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `push` - append to the end; equivalent to `splice @a, scalar(@a), 0, LIST`
- `pop` - remove the last element; equivalent to `splice @a, -1`
- `shift` - remove the first element; equivalent to `splice @a, 0, 1`
- `unshift` - prepend; equivalent to `splice @a, 0, 0, LIST`
- `delete` - superficially similar but semantically different. `delete $a[$i]` leaves a hole (the slot becomes `undef`) and does **not** renumber later indices; the array's length is unchanged unless the deleted slot was the last one. `splice @a, $i, 1` physically removes the slot and shifts every later element down by one. Use `delete` for sparse arrays where index identity matters; use `splice` when you want the array to close up.
- `grep` - filter an array into a new one by predicate; reach for it instead of `splice` when you want “keep only the ones that match” rather than “cut out a contiguous range”

Regular expressions and pattern matching

split

Cut a string into a list of fields using a regex separator.

`split` scans `EXPR` for matches of `PATTERN` and returns the pieces between the matches as a list. The matched text itself is the separator and is **not** included in the result - unless the pattern contains capture groups, in which case each capture becomes an extra element in the output. In scalar context the list size is returned.

If `PATTERN` is omitted, `split` behaves like the `awk` default: runs of whitespace are the separator and any leading whitespace in `EXPR` is stripped. If `EXPR` is omitted, `$_` is split.

Synopsis

```
split /PATTERN/, EXPR, LIMIT
split /PATTERN/, EXPR
split /PATTERN/
split
```

What you get back

A list of substrings (fields) in list context; the count of fields in scalar context. The separator text is never part of a field.

```
my @fields = split /,/, "a,b,c";      # ("a", "b", "c")
my $n      = split /,/, "a,b,c";      # 3
```

Splitting an EXPR that is the empty string always yields zero fields, regardless of LIMIT:

```
my @x = split /,/, "", -1;           # ()
```

Prior to Perl 5.11, `split` in void or scalar context overwrote `@_`. Modern Perl does not; never rely on that old side effect.

The LIMIT argument

LIMIT controls how many fields are produced and, crucially, whether trailing empty fields are kept.

- **Positive LIMIT** - upper bound on the number of fields. EXPR is split at most LIMIT - 1 times; the final field holds the rest of the string verbatim. LIMIT = 1 means no splits at all - you get the whole string back as a single element.

```
my @x = split /,/, "a,b,c", 1;      # ("a,b,c")
my @x = split /,/, "a,b,c", 2;      # ("a", "b,c")
my @x = split /,/, "a,b,c", 3;      # ("a", "b", "c")
my @x = split /,/, "a,b,c", 4;      # ("a", "b", "c")
```

- **Negative LIMIT** - treated as arbitrarily large. As many fields as possible are produced, including **all** trailing empty fields.

```
my @x = split /,/, "a,b,c,,, ", -1; # ("a", "b", "c", "", "", " "
    ↪ " ")
```

- **Omitted or zero LIMIT** - like negative, **except** trailing empty fields are stripped. Leading empty fields are always kept.

```
my @x = split /,/, "a,b,c,,, ";      # ("a", "b", "c")
my @x = split /,/, ",,a,b";          # ("", "", "a", "b")
```

- **Implicit LIMIT in list assignment.** When assigning to a fixed list, Perl sets LIMIT to one more than the number of targets, so the trailing fields do not need to be scanned. The following gets LIMIT = 3 automatically:

```
my ($login, $passwd) = split /:/;
```

In time-critical code, pass an explicit LIMIT rather than letting the whole string be scanned.

The awk-style whitespace case: `split " "`

A literal single space as the pattern - `split " "`, not `split / /`

- triggers a special case: PATTERN is treated as `/\s+/` **and** any leading whitespace in EXPR is stripped before splitting. This matches classic awk behaviour.

```
my @x = split " ", " Quick brown fox\n";
# ("Quick", "brown", "fox")

my @x = split " ", "RED\tGREEN\tBLUE";
# ("RED", "GREEN", "BLUE")
```

To split on a *single* literal space only, use the regex form `/ /` - it is not special-cased:

```
my @x = split / /, " abc";           # ("", "abc")
```

Since Perl 5.18, the trigger is any expression whose value is the single-character string `" "` (not just a literal), and since Perl 5.28 the rule works correctly under `use feature 'unicode_strings'`. Under Perl 5.39.9+ the `/x` default modifier does **not** affect `split STRING`, so `split " "` still means awk-emulation even inside `use re "/x"`. If you want to split on one literal space under `use re "/x"`, write `split /(?-x: )/` or `split /\x{20}/`.

If PATTERN is omitted entirely, it defaults to `" "`, so bare `split` is the awk-style form.

The empty-pattern case: splitting into characters

If PATTERN matches the empty string, the split happens at every match position - i.e. between characters.

```
my @x = split //, "abc";           # ("a", "b", "c")
```

As a split-specific rule, the bare match operator `//` is **not** the “repeat the last successful match” form here; it is the literal empty pattern. A zero-width match at the very start of EXPR never produces an empty leading field, which is why splitting a leading space is handled like this:

```
my @x = split //, " abc";           # (" ", "a", "b", "c") - 4,
↳not 5
my @x = split //, " abc", -1;       # (" ", "a", "b", "c", "") -
↳trailing empty with -1
```

A positive-width match at the start *does* produce an empty leading field:

```
my @x = split / /, " abc";           # ("", "abc")
```

Capture groups become extra elements

If PATTERN contains capturing groups, each capture is inserted into the output list for every separator match, in the order the groups are declared. A group that does not participate in the match contributes `undef`. These extras do **not** count toward LIMIT.

```
my @x = split /-|,/ , "1-10,20", 3;
# ("1", "10", "20")

my @x = split /(-|,)/ , "1-10,20", 3;
# ("1", "-", "10", ",", "20")

my @x = split /-|(,)/ , "1-10,20", 3;
# ("1", undef, "10", ",", "20")

my @x = split /(-)|,/ , "1-10,20", 3;
# ("1", "-", "10", undef, "20")

my @x = split /(-)|(,)/ , "1-10,20", 3;
# ("1", "-", undef, "10", undef, ",", "20")
```

Use this when you want the separators preserved alongside the fields - typical for tokenisers where the delimiters are themselves part of the output stream.

Global state it touches

- `$_` - EXPR defaults to it when no second argument is given.
- `@_` - modern Perl does **not** touch it. Only relevant if you support pre-5.11 perls.
- The regex engine's match variables (`$1`, `$2`, ..., `$&`, `$'`, `$``) are **not** set as a side effect of `split`, even when PATTERN captures - the captures go into the returned list.

Examples

Classic CSV-style line, no embedded commas:

```
my @cells = split /,/, "alice,bob,carol"; # ("alice", "bob",
↪ "carol")
```

Parse a `/etc/passwd` line into known fields; implicit LIMIT stops at the first seven separators:

```
my ($user, $pw, $uid, $gid, $gecos, $home, $shell)
    = split /:/, $line;
```

Awk-style whitespace tokenising, leading whitespace discarded:

```
my @words = split " ", "  one\ttwo  three\n";
# ("one", "two", "three")
```

Characters of a string - the empty pattern:

```
my @chars = split //, "hello";           # ("h", "e", "l", "l",  
↪ "o")
```

Keep every trailing empty field for a strict column-count reader:

```
my @cols = split /\t/, $line, -1;
```

Preserve separators by capturing - round-trip reconstructible:

```
my @parts = split /([,;]+)/, "a,b;c";     # ("a", ",", "b", ";",  
↪ "c")  
my $same = join "", @parts;              # "a,b;c"
```

Default form reads `$_` and does awk-style whitespace splitting:

```
for (@lines) {  
    my @f = split;                       # split " ", $_  
    ...  
}
```

Edge cases

- **Empty input** - `split /X/, ""` returns the empty list for every `LIMIT`, including `-1`.
- **`LIMIT = 1`** - no splitting occurs; the whole string is the one and only field. Useful when you want `split` semantics conditionally without losing the input.
- **`/^/` pattern** - treated as if `/^/m` so it matches at every line start, which is the only useful behaviour. Splits a string into lines keeping the newlines.
- **Zero-width match at position 0** - never produces an empty leading field (see `split //, " abc"` above).
- **Positive-width match at position 0** - produces an empty leading field, always.
- **Match at end of string** - produces an empty trailing field, which is then stripped unless `LIMIT` is non-zero.
- **Patterns with modifiers** - the usual `qr//` modifiers (`/i`, `/m`, `/s`, `/x`, `/u`, `/a`, `/l`, `/n`) apply. The pattern need not be a literal; any expression producing a regex or string works, and `qr//` objects are accepted.
- **Single-space string that is not a literal** - from Perl 5.18 on, any expression evaluating to `" "` triggers the awk-style special case, not just the literal `" "`.
- **`split " "` vs `split / /`** - the first is awk-style whitespace (`\s+` with leading-ws stripping), the second is a single literal space. The two are not interchangeable.
- **Unicode whitespace** - under use feature `'unicode_strings'` (default since 5.28 for the awk-style case), `split " "` treats Unicode whitespace as a separator too. Outside that scope the behaviour is affected by the “Unicode Bug”.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `join` - inverse operation; stitches a list back into a string with a chosen glue
- `m` - the match operator; when you want to *find* something rather than cut the string around it
- `qr` - pre-compile a pattern once for repeated `split` calls in a hot loop
- `index` - locate a fixed substring without building the full list of fields; cheaper when you only need the first split
- `substr` - extract by byte/character offset when the field boundaries are positional, not delimited

SCALARs and strings

sprintf

Build a formatted string from a format template and a list of values.

`sprintf` takes a `FORMAT` string containing literal text interspersed with %-introduced conversion specifiers, consumes values from `LIST` to fill each specifier, and returns the resulting string. It is the same machinery that drives `printf`, minus the write to a file-handle. Perl implements its own formatter; it does not call the C library's `sprintf(3)` except for the final floating-point digit generation, so non-standard extensions that happen to exist in your `libc` are not available here.

Synopsis

```
my $s = sprintf $format, @args;
my $s = sprintf "%-10s %5d", $name, $count;
my $s = sprintf "%.3f", $x;
```

What you get back

A single scalar string. Never a list, regardless of context. If `FORMAT` contains no % specifiers, the return value is `FORMAT` unchanged and `LIST` is ignored. If `FORMAT` references more arguments than `LIST` supplies, the missing values are treated as `undef` (numeric zero or the empty string depending on the conversion), and a `Missing argument in sprintf` warning is issued under `use warnings`.

Passing an array as the first argument is almost always a mistake - the array is evaluated in scalar context, so Perl uses its element count as the format. Use `sprintf $format, @rest` with the format held separately, or `sprintf "@array"` for the array-as-string case.

Global state it touches

- LC_NUMERIC locale, when use locale is in effect and `POSIX::setlocale` has been called, determines the decimal-point character used by %e, %f, %g and their upper-case variants. Without use locale the decimal point is always ., even if the process locale says otherwise.
- No other special variables are consulted. sprintf does not read \$_; an explicit list is required.

Conversion specifiers

The core set, in the order you will actually reach for:

Spec	Meaning
%s	String. Stringifies the argument.
%d, %i	Signed integer in decimal. Floats are truncated toward zero.
%u	Unsigned integer in decimal.
%f, %F	Fixed-point floating point. Default 6 digits after the decimal.
%e, %E	Scientific notation. Upper-case E selects the upper-case form.
%g, %G	%e or %f, whichever is shorter. Precision is <i>significant digits</i> , not digits after the decimal.
%x, %X	Unsigned integer in hexadecimal; %X uses upper-case A–F.
%o	Unsigned integer in octal.
%b, %B	Unsigned integer in binary.
%c	Single character whose codepoint is the given integer.
%%	A literal %. Consumes no argument.
%p	Perl-value address, in hexadecimal. Diagnostic use only.
%n	Stores the number of characters written so far into the next argument (which must be a writable scalar).
%a, %A	Hexadecimal floating point.

Upper-case size synonyms %D %U %O and %F exist for historical compatibility; prefer the lower-case forms.

Flags, width, precision

Between % and the conversion letter, in order:

1. **Argument index** - N\$ picks the N-th argument (1-based): `sprintf '%2$s %1$s', 'world', 'hello'` yields "hello world".
2. **Flags** - any combination of:
 - - left-justify within the field (default is right-justify).

- `+` prefix non-negative numbers with `+`.
 - `` `` (a space) prefix non-negative numbers with a space. When both space and `+` are given, `+` wins.
 - `0` pad numeric conversions with zeros instead of spaces. Ignored when `-` is also set or a precision is given for integers.
 - `#` alternate form: `0` prefix for octal, `0x/0X` prefix for non-zero hex, `0b/0B` prefix for non-zero binary.
3. **Minimum width** - a decimal number, or `*` to take the width from the next argument, or `*N$` for an explicit argument. Negative width (via `*`) is equivalent to the `-` flag.
 4. **Precision** - `.` followed by a decimal number, or `.` `*` (next argument), or `.` `*N$`. Meaning depends on the conversion:
 - `%f / %e`: digits after the decimal point (default 6).
 - `%g`: total significant digits.
 - `%d / %x / %o / %b`: minimum digits; the number is zero-padded to this width. The `0` flag is ignored when a precision is present for integers.
 - `%s`: maximum width; truncates.
 5. **Size modifier** - `hh, h, j, l, q, L, ll, t, z, V`. Interprets the argument as a specific C type. Almost never useful from Perl - Perl has one integer type and one float type, and Perl-level values do not overflow into the sizes these modifiers select. `V` is a no-op meaning “Perl’s default size”.
 6. **Conversion letter** - one of the table entries above.

The vector flag

Prefix the conversion letter with `v` (optionally `*v` or `*N$v` to supply a separator) and Perl treats the argument as a vector of codepoints, applying the conversion to each codepoint and joining the results with `.` (or the supplied separator):

```
sprintf "%vd", "AB\x{100}";           # "65.66.256"
sprintf "%vX", "\x01\x02\x1f";       # "1.2.1F"
sprintf "%*vX", ":", "\xde\xad\xbe\xef"; # "DE:AD:BE:EF"
```

Common uses: rendering `^V`, formatting IPv4 / IPv6 byte sequences, or dumping bit-strings. The argument is always read as a string of codepoints, not as a list.

Examples

Pad and align strings in a fixed-width report column:

```
printf "%-10s %5d\n", "widgets", 42;   # "widgets      42\n"
printf "%-10s %5d\n", "sprockets", 7;  # "sprockets     7\n"
```

Zero-pad a fixed-width integer, e.g. for lexicographic sort keys or line numbers:

```
sprintf "%08d", 1234;           # "00001234"  
sprintf "%05d", -7;           # "-0007" (sign consumes a slot)  
→slot)
```

Round to a fixed number of decimal places. `sprintf` does the rounding; `printf` is a common way to coerce a display value:

```
sprintf "%.3f", 3.14159265;     # "3.142"  
sprintf "%.0f", 2.5;           # "2" or "3" - banker's rounding  
→rounding  
# is platform-dependent
```

Scientific notation, with and without a chosen precision:

```
sprintf "%e", 6.022e23;         # "6.022000e+23"  
sprintf "%.2e", 6.022e23;      # "6.02e+23"  
sprintf "%g", 0.0001234;       # "0.0001234"  
sprintf "%g", 0.00001234;      # "1.234e-05" (switches to  
→%e)
```

Hex, octal, and binary with the `#` alternate-form prefix:

```
sprintf "%#x", 255;            # "0xff"  
sprintf "%#o", 8;             # "010"  
sprintf "%#b", 10;            # "0b1010"  
sprintf "%08b", 10;           # "00001010" (no prefix, width 8)  
→width 8)
```

Character from a codepoint - `%c` is essentially `chr` routed through the format engine:

```
sprintf "%c", 65;              # "A"  
sprintf "%c", 0x2014;          # "-" (em dash)
```

Field width from the argument list - useful when the width is computed at runtime:

```
my $w = 12;  
sprintf "%-*s|", "name", $w;   # "name      |"
```

Out-of-order arguments - useful when the same value appears multiple times or when localised formats reorder placeholders:

```
sprintf "%2\${s} loves %1\${s}", "Alice", "Bob"; # "Bob loves Alice"
```

Build an IPv4 address string:

```
sprintf "%vd", "\x7f\x00\x00\x01"; # "127.0.0.1"
```

Capture length via `%n` (the receiving argument must be a writable scalar):

```
my $written;  
my $s = sprintf "hello%n, world", $written;
```

(continues on next page)

(continued from previous page)

```
# $s      is "hello, world"
# $written is 5
```

Edge cases

- **Integer truncation on %d with floats:** %d converts via integer coercion, which truncates toward zero. `printf "%d", 3.9` is "3", not "4". Use `%.0f` if you want banker's-rounded display, or `int / sprintf "%.0f"` explicitly first.
- **%s on an object:** stringification runs, which may invoke overloaded `""` or fall back to `"ClassName=HASH(0x...)"`. There is no way to suppress overloading inside `printf`; call `overload::StrVal($obj)` first if you need the raw form.
- **Missing arguments:** `printf "%s %s", "one"` returns "one " and warns `Missing argument in printf at ... under use warnings`. No exception is raised.
- **Too many arguments:** extras are ignored silently. No warning.
- **%s with precision truncates, %d with precision zero-pads.** `printf "%.3s", "abcdef"` is "abc". `printf "%.3d", 7` is "007".
- **%0 flag vs precision on integers:** precision wins. `printf "%05.3d", 7` is "007", not "00007" - the 0 flag is ignored once you give an explicit integer precision.
- **Negative width via *:** `printf "%*s", -6, "x"` is "x ". Negative precision via `.*` counts as no precision at all.
- **NaN and Inf:** `printf "%f", 9**9**9` yields "Inf"; `printf "%f", -sin(9**9**9)` yields "NaN". Exact spelling ("Inf" vs "inf", "NaN" vs "nan") follows the Perl build's convention, not the %f / %F case distinction.
- **Locale decimal point:** with `use locale` and a `de_DE` locale, `printf "%.2f", 1.5` can produce "1,50". Without `use locale` the result is always "1.50". Code that writes numeric output for consumption by other programs should either avoid `use locale` in that scope or use `use locale ':not_characters'` and format numbers outside locale influence.
- **%p is diagnostic-only:** the hexadecimal pointer value changes from run to run and is meaningless outside the running process. Do not use it as an identity key - two references may share the same address after the first has been freed.
- **Array passed as FORMAT:** `printf @a` formats using the scalar count of @a as the format string. Almost always a bug. Always keep the format in its own scalar.
- **Unsupported size modifier:** on platforms lacking e.g. long doubles, the `L/q/l` size on a float conversion silently falls back to the default size, and a `printf` warning is issued under `use warnings`. To promote this to an exception, use `use warnings FATAL => 'printf'`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `printf` - same formatting engine, but writes to a filehandle instead of returning the string
- `print` - use with a pre-built `sprintf` result when you want to format once and write to several handles
- `chr` - single codepoint to single character; equivalent to `sprintf "%c", $n`
- `hex`, `oct` - inverse operations: parse a string in hex/octal/binary back into a number
- `pack` - build a binary string from a template; reach for it when the output is bytes, not a human-readable format
- `$_` - *not* consulted by `sprintf`; unlike many built-ins it always needs an explicit list

Numeric functions

`sqrt`

Return the non-negative square root of `EXPR`.

`sqrt` computes the **principal square root** - the unique non-negative real number whose square equals `EXPR`. For any `EXPR >= 0` the result is `>= 0`; `sqrt` never returns the negative root. If `EXPR` is omitted, `$_` is used.

The computation is performed in double-precision floating point (IEEE 754 binary64), so the result is a float even when the input is an integer and even when the mathematical answer is itself an integer.

Synopsis

```
sqrt EXPR
sqrt
sqrt(EXPR)
```

What you get back

A floating-point number: the non-negative square root of `EXPR`. For negative input the result is NaN (and a warning is emitted under `use warnings`); see [Edge cases](#). For complex-valued roots load `, which` overloads `sqrt` so that `sqrt(-1)` returns `i`.

```
my $r = sqrt 2;           # 1.4142135623731
my $r = sqrt;             # sqrt of $_
```

The default argument, precedence, and parsing

Three parser details shape every `sqrt` call:

- **No argument uses `$_`.** `sqrt` with no argument and no parentheses reads the topic variable. Handy inside `map` and `grep` blocks that already bind `$_`.
- **Named-unary precedence.** `sqrt` is a named unary operator, so it binds tighter than most binary operators but looser than multiplication and exponentiation. `sqrt 2 + 3` means `sqrt(2) + 3`, not `sqrt(2 + 3)` - parenthesise when in doubt.
- **`sqrt(...)` is a function call.** As with every named unary, writing an opening paren immediately after the name turns the remainder into the argument list: `sqrt(4) * 2` is 4 (the `sqrt` of 4) times 2, i.e. 4.

```
my @roots = map { sqrt } 1, 4, 9;    # (1, 2, 3); $_ is the topic
my $x = sqrt 2 + 3;                 # 4.41421... - sqrt(2) + 3
my $y = sqrt(2 + 3);                 # 2.23606... - sqrt(5)
```

Examples

Classic irrational root. The printed decimal is the closest binary64 approximation, not the exact value:

```
printf "%.15f\n", sqrt(2);           # 1.414213562373095
```

Pythagorean distance between two points:

```
sub distance {
    my ($x1, $y1, $x2, $y2) = @_;
    return sqrt(($x2 - $x1) ** 2 + ($y2 - $y1) ** 2);
}

print distance(0, 0, 3, 4), "\n";    # 5
```

Zero is its own square root, and so is one:

```
print sqrt(0), "\n";                 # 0
print sqrt(1), "\n";                 # 1
```

Perfect squares round-trip exactly for small integers, because the mathematical result is representable in binary64:

```
print sqrt(16), "\n";                # 4
print sqrt(144), "\n";               # 12
print sqrt(1_000_000), "\n";         # 1000
```

Large perfect squares may **not** round-trip exactly - the result is the nearest representable float, which can be slightly off:

```
my $n = 9_007_199_254_740_996;       # 2**53 + 4
printf "%.3f\n", sqrt($n) ** 2 - $n; # non-zero in general
```

Compare to the exponentiation operator, which is the general way to take any real power:

```
print 2 ** 0.5, "\n";           # same as sqrt(2)
print 27 ** (1/3), "\n";        # cube root of 27 ≈ 3
```

Using to get a complex result from a negative input:

```
use Math::Complex;
my $z = sqrt(-1);               # i
print $z, "\n";                 # "i"
print sqrt(-4), "\n";           # "2i"
```

Edge cases

- **Negative input without**: returns NaN and triggers an `Invalid operation` warning under `use warnings`. NaN compares unequal to itself, so never test with `==`; use `POSIX::isnan` or check `$x != $x`.

```
my $r = sqrt(-1);               # NaN
print "bad\n" if $r != $r;      # NaN self-compare trick
```

- **Negative input with**: once is loaded, `sqrt` is overloaded for complex scalars and returns the principal complex root. Real inputs still go through the built-in.
- **Non-numeric input**: coerced to a number using the usual rules - leading whitespace and a numeric prefix are consumed, the rest is ignored. Under `use warnings` an `isn't numeric` warning fires when the string has no numeric prefix at all. `undef` coerces to `0`.

```
print sqrt("9abc"), "\n";       # 3, with a warning under
↪warnings
print sqrt(""), "\n";           # 0, with a warning
print sqrt(undef), "\n";        # 0, with an uninitialized
↪warning
```

- **Floating-point imprecision for perfect squares**: for sufficiently large integers, the mathematically exact root may not be representable. `sqrt($n*$n)` is not guaranteed to equal `$n` for large `$n`. If you need an exact integer square root, do not use `sqrt` - use `'s` `bsqrt` or a dedicated `isqrt` implementation.
- **Special IEEE values**: `sqrt("Inf")` is `Inf`; `sqrt("NaN")` is `NaN`; `sqrt(-0.0)` is `-0.0` (negative zero, which compares equal to `0`). `sqrt("-Inf")` is `NaN`.
- **Integer overflow is not an issue**: because the result is always a float, `sqrt` never overflows for any finite numeric input.
- **List context is irrelevant**: `sqrt` always returns a single scalar. In list context that scalar is the only element of the returned list.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `abs` - absolute value; pair with `sqrt` when you want `sqrt(abs($x))` to sidestep negative input
- `**` - exponentiation operator; `$x ** 0.5` is `sqrt`, and `$x ** (1/$n)` generalises to other roots
- `hypot` - from [POSIX](#), computes `sqrt($x*$x + $y*$y)` without intermediate overflow
- `-` - overloads `sqrt` and friends so that `sqrt(-1)` returns `i` instead of `NaN`
- `,` - arbitrary-precision square roots via `bsqrt` when `binary64` is not enough

Numeric functions

`srand`

Seed the pseudo-random number generator.

`srand` sets the internal state of the PRNG that `rand` draws from. Call it with an explicit integer to get a reproducible stream, or call it with no argument to reseed from the best entropy source the platform offers. Since Perl 5.14 `srand` **returns the seed it used**, so you can log it and replay the same run later.

Synopsis

```
srand EXPR
srand
```

What you get back

The seed that was installed, as an integer. For `srand($n)` the return value is `$n` (after the usual integer truncation). For argument-less `srand()` the return value is the auto-chosen seed drawn from system entropy - capture it if you ever want to reproduce the run.

```
my $seed = srand();           # auto-seed, remember what we
    ↪ used
warn "seed for this run: $seed\n";
```

Before Perl 5.14 `srand` returned nothing useful. Code that relies on the seed return should declare the minimum version:

```
use v5.14;                    # so srand returns the seed
```

Global state

`srand` mutates a single process-wide PRNG. Every subsequent call to `rand` in the same process - from any package, any module, any thread-free context - reads from that one state. There is no per-package or per-scope RNG. Consequences:

- Calling `srand($k)` anywhere in the program affects *every* later `rand` call, including those inside modules you did not write.
- Two processes that call `srand($k)` with the same `$k` produce identical `rand` streams. This is the basis of reproducible tests and the reason you must reseed after `fork`.
- If `srand` is never called explicitly, it is called implicitly **without argument** the first time `rand` is invoked.

Examples

Reproducible test case with an explicit seed - same input, same sequence, every run:

```
srand(42);  
my @sample = map { rand() } 1..5;    # identical on every invocation
```

Seed from `time()` for rare-collision randomness across separate runs started at different wall-clock seconds. This is the classical pattern; it is **not** cryptographic-grade:

```
srand(time() ^ $$);                  # time XOR pid, old-school  
→spread
```

Save-seed, restore-seed pattern for a test that picks a random subset of its cases but logs enough to replay any failure:

```
my $seed = srand();                  # auto-seed, capture it  
eval { run_randomised_tests() };  
if ($?) {  
    warn "FAILED with seed=$seed - reproduce with: srand($seed)\n";  
    die $@;  
}
```

Reseed after `fork` so parent and child do not share a stream:

```
my $pid = fork();  
die "fork: $!" unless defined $pid;  
if ($pid == 0) {  
    srand();                          # child gets fresh entropy  
    exec_child_work();  
}
```

Two independent streams from the same seed - useful when you want to checkpoint and resume, or diff two runs that diverged after a decision point:

```
srand(12345);  
my @first = map { rand() } 1..10;
```

(continues on next page)

(continued from previous page)

```
srand(12345);
my @second = map { rand() } 1..10; # identical to @first
```

Edge cases

- **No-argument form auto-seeds with the best available entropy.** On Linux this draws from `/dev/urandom` (or `getrandom(2)`). The seed is wide enough that you should treat it as opaque, not as a small integer.
- **Do not call `srand()` without arguments more than once per process.** The PRNG's internal state already carries more entropy than any single seed can restore, so re-seeding *loses* randomness rather than adding it. The two legitimate reasons to call it repeatedly are (a) with an explicit seed for reproducibility, and (b) once in each child after `fork`.
- **Decimal arguments are silently truncated to integer.** `srand(42)` and `srand(42.9)` install the same seed. Always pass an integer so intent is obvious.
- **Same seed $\Rightarrow$ identical `rand` stream.** That is the whole point for tests; it is the whole problem for security.
- **Not cryptographically secure.** `rand`/`srand` use a fast non-cryptographic PRNG. Do not use them to generate session tokens, nonces, keys, password salts, or anything an adversary must not be able to predict. Use `Crypt::URandom` or read from `/dev/urandom` directly.
- **PERL_RAND_SEED environment variable.** If set to a non-negative integer at process startup, argument-less `srand()` (including the implicit call triggered by the first `rand`) installs a deterministic seed derived from that value. The derivation is deliberately unspecified; only same-value-same-binary-same-code reproducibility is guaranteed. Intended for debugging and performance analysis. The variable is read **once** at startup; mutating `%ENV` later has no effect on the current process.
- **Threads** (use `threads`): each interpreter clone gets its own PRNG state. Seeding in one thread does not affect another.
- **Scope of the seed:** there is no way to `local`-ise the PRNG state. If a library you call in between does an `srand`, your stream is perturbed. Capture and restore manually if that matters.

Differences from upstream

Fully compatible with upstream Perl 5.42. `PERL_RAND_SEED` is honoured with the same “deliberately unspecified but reproducible” contract as `perl5`.

See also

- `rand` - the consumer of the state `srand` installs; draws the next value from the seeded stream
- `time` - seconds since the epoch; the classic quick-and-dirty seed source for non-security code (`srand(time() ^ $$)`)

- `Crypt::URandom` - cryptographically secure random bytes from the OS entropy pool; use this, not `rand`, for tokens and keys
- `Math::Random::MT` - Mersenne Twister PRNG with its own independent state, when you need a second stream that is not perturbed by library calls to `rand`
- `perlrun` - documents the `PERL_RAND_SEED` environment variable and other startup knobs

Filehandles, files, directories

stat

Get a file's status information.

`stat` asks the operating system for the metadata record of a file - size, ownership, permissions, timestamps, inode - and returns it as a 13-element list. It is the underlying primitive behind the `-X` file tests (`-e`, `-f`, `-r`, ...), behind modules like `File::stat`, and behind any script that needs to inspect a file without opening it. `stat` dereferences symlinks and reports on the target; use `lstat` to get the link itself.

Synopsis

```
stat FILEHANDLE
stat EXPR
stat DIRHANDLE
stat
stat _
```

What you get back

In list context, a 13-element list of numeric fields. In scalar context, a boolean: true on success, false on failure. Either way, on success the kernel's stat buffer is cached in the special filehandle `_`, which any subsequent `stat`, `lstat`, or `-X` test can re-use without a new syscall.

On failure, list context returns the empty list and `$!` is set to the `errno` value from the failed `stat(2)`.

The fields, in order:

Index	Field	Meaning
0	dev	device number of the filesystem holding the file
1	ino	inode number
2	mode	file mode - type bits and permission bits combined
3	nlink	number of hard links to the file
4	uid	numeric user ID of the owner
5	gid	numeric group ID of the owner
6	rdev	device identifier, for character/block special files
7	size	total size in bytes
8	atime	last access time, seconds since the epoch (1970-01-01 00:00 GMT)
9	mtime	last modify time, seconds since the epoch
10	ctime	inode change time, seconds since the epoch - not creation time
11	blksize	preferred I/O block size for reads/writes against this file
12	blocks	number of filesystem blocks actually allocated (often 512 B each)

Field 2 (mode) packs both the file type and the permission bits into one number. Mask with 07777 to see just the permissions; use the S_IF* constants and S_IS* helpers from Fcntl to test the type portably.

```
my $mode = (stat($filename))[2];
printf "permissions are %04o\n", $mode & 07777;
```

Not every filesystem populates every field. atime, blksize, and blocks may come back as 0 on filesystems that do not track them (for instance some FUSE mounts or network filesystems); ctime is not a portable “creation time” - it tracks the last change to the inode itself (permissions, ownership, link count), not when the file was first created.

Global state it touches

- `$_` - read when stat is called with no argument. stat with no EXPR stats the filename held in `$_`, **not** the cached `_` handle.
- `$!` - set to the errno of the failed stat(2) syscall when the call fails.
- The **special `_` filehandle** - a per-interpreter cache of the most recent successful stat, lstat, or -X result. Passing `_` as the argument reuses that cached buffer and issues no syscall. Successful calls refresh it; failed calls leave the previous contents in place.

Examples

Basic list-context unpack:

```
my ($dev, $ino, $mode, $nlink, $uid, $gid, $rdev, $size,
    $atime, $mtime, $ctime, $blksize, $blocks) = stat($filename);
```

Pick out a single field with a list slice:

```
my $size = (stat($filename))[7];
my $mtime = (stat($filename))[9];
```

Scalar context as an existence-plus-readable-metadata probe:

```
if (stat($path)) {  
    my $size = (stat _)[7];           # reuse the cached buffer  
    print "size = $size\n";  
} else {  
    warn "stat $path: $!";  
}
```

Reusing the cache across an -X test and a stat:

```
if (-x $file && (($d) = stat(_)) && $d < 0) {  
    print "$file is executable NFS file\n";  
}
```

Named-accessor access through File::stat:

```
use File::stat;  
my $sb = stat($filename);  
printf "file %s, size %s, perm %04o, mtime %s\n",  
    $filename, $sb->size, $sb->mode & 0777,  
    scalar localtime $sb->mtime;
```

Decoding the mode with Fcntl:

```
use Fcntl ':mode';  
my $mode = (stat($filename))[2];  
printf "permissions %04o\n", S_IMODE($mode);  
print "is a directory\n" if S_ISDIR($mode);
```

Edge cases

- **No argument:** `stat`; `stats $_,` **not** the cached `_` handle. To re-read the cache you must write `stat(_)` with the underscore as an explicit bareword.
- **The `_` bareword is special:** it names the stat cache, not a file. Quoting it (`stat("_")`) stats a file literally called `_` in the current directory.
- **Symlinks are followed:** `stat` reports on the target of a symlink. For the link itself, use `lstat`. A broken symlink makes `stat` fail with `ENOENT`; `lstat` on the same path succeeds.
- **DIRHANDLE form:** passing a directory handle from `opendir` returns the stat of the open directory itself. Not every platform supports this; on those that do not, the call fails and `$!` is set.
- **Inode numbers may overflow integers:** on filesystems with 64-bit inode numbers beyond Perl's integer range, `ino` is returned as a decimal string to preserve the full value. Compare inode numbers with `eq`, not `==`, to stay correct on those filesystems. `eq` works fine on numeric inodes too.
- **c time is not creation time:** it is the inode-change time - updated by `chmod`, `chown`, `rename`, link-count changes, and the file's own writes. Linux exposes a true creation time (`btime`) via `statx(2)`, but that is not part of `stat`'s return list.

- **Zero fields on some filesystems:** `atime` (on `noatime` mounts), `blksize`, and `blocks` may be reported as `0`. Treat `0` as “unavailable”, not as a meaningful value.
- **Scalar context does not populate `_` on failure:** a failed `stat` leaves the previous cached buffer untouched. Always check the return value before a follow-up `stat(_)` or `-X _`.
- **Race windows:** the metadata returned is a snapshot. Anything that acts on a path based on `stat`’s result (permission checks, TOCTOU patterns) must be written to tolerate the file changing between the `stat` and the follow-up operation - or better, open the file first and `stat` the filehandle.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `lstat` - like `stat` but does **not** follow symlinks; reports on the link itself
- `-X` file-test operators (`-e`, `-f`, `-r`, `-w`, `-x`, ...) - built on top of the same `stat` buffer; share the `_` cache with `stat` and `lstat`
- `chmod` - change the permission bits reported in field 2 (mode)
- `chown` - change the uid and gid reported in fields 4 and 5
- `File::stat` - named accessors (`$sb->size`, `$sb->mtime`, ...) over the same 13-field list; use `File::stat` overrides `stat` in the current scope
- `Fcntl` - `:mode` tag exports the `S_I*` constants and `S_IS*` helpers for decoding field 2

Scoping

state

Declare a lexically scoped variable whose value persists across calls to its enclosing subroutine.

A `state` variable has the same visibility rules as a `my` variable - it is visible only inside the block that declares it - but it is **initialised exactly once** per enclosing closure, not on every entry to the block. This is Perl’s built-in way to write persistent private variables: counters, memoisation caches, one-time-computed lookup tables, anything you would otherwise hide inside a file-scoped lexical above a sub.

Synopsis

```
state $scalar
state $scalar = EXPR
state ($a, $b, undef, $c)
state @array
state %hash
```

(continues on next page)

(continued from previous page)

```
state TYPE $var
state $var : ATTRS
```

What you get back

`state` is a declaration, not a function. Like `my` it returns the variable(s) it declares, so it can be used in any expression position where the declared variable would appear:

```
(state $n //= 0)++;           # declare-and-use in one
↪expression
print ++(state $calls), "\n"; # count calls to this line
```

The initialiser on the right of `=` is evaluated **only the first time control flows through the declaration** in that particular closure instance. On every subsequent entry, the declaration re-exposes the already-initialised variable and the initialiser is skipped.

Persistence scope - what “once” means

“Once per enclosing closure” is the precise rule. A state variable lives as long as the CV (code value) that contains its declaration:

- In a **named sub**, the CV is created once at compile time. The state variable is initialised on the first call and retains its value across every later call for the life of the process.
- In an **anonymous sub / closure**, each `sub { ... }` evaluation produces a fresh CV with its own fresh state slot. Two closures made from the same source do **not** share state.
- In a **nested block** inside a sub, the state still belongs to the surrounding CV - it is not re-created on each entry to the inner block.

```
sub counter { state $n = 0; ++$n }
counter(); counter(); counter();    # returns 1, 2, 3

my $make = sub { sub { state $n = 0; ++$n } };
my $a = $make->();
my $b = $make->();
$a->(); $a->(); $b->();                 # $a sees 1, 2; $b sees 1
```

Feature gate and availability

`state` has always been feature-gated. Two ways to enable it:

- use feature `'state'`; - turns on just this feature.
- use `v5.10`; (or any later version bundle, including use `v5.36` and use `v5.42`) - implicit via the version feature bundle. Every modern program already has `state` available through its use `vX.Y` line.

Without the feature in scope, `state` is parsed as an ordinary bareword. To reach the built-in unconditionally, write `CORE::state`.

```
use v5.36;                                # 'state' feature is on by default
sub hits { state $n = 0; ++$n }
```

List declarations and initialisers

state supports the parenthesised list form for declaring several variables at once, with `undef` as a placeholder:

```
state ($x, $y, undef, $z);
```

What it does **not** yet support is **initialising a list** of state variables. Upstream explicitly notes this: initialisation of state variables in a parenthesised list is not currently possible, so the list form is useful only for declaration.

Aggregates - `state @array` and `state %hash` - have historically had the same restriction on initialisers (Perl emitted an experimental warning, and in several versions a syntax error, for `state @a = (1, 2, 3)`). Build the aggregate inside the body instead:

```
sub primes_below_100 {
    state @p;
    unless (@p) {
        @p = _sieve(100);           # fill on first call only
    }
    return @p;
}
```

The `unless (@p)` / `unless (%h)` idiom above is the portable replacement for an aggregate initialiser.

Declaration placement and same-statement semantics

Like `my`, `our`, and `local`, a state declaration can appear anywhere an expression is allowed (except inside string interpolation). The new binding takes effect for **subsequent statements**, not for other mentions of the same variable in the same statement:

```
package main;
use feature 'state';
our $x = 2;
foo($x, state $x = $x + 1, $x);    # foo() receives (2, 3, 2)
foo($x, $main::$x);               # foo() receives (3, 2)
```

The first `$x` and the trailing `$x` in the same call still refer to `our $x`; only the next statement sees the state `$x`.

Examples

A hit counter - the canonical one-liner:

```
sub hits { state $n = 0; ++$n }
hits(); hits(); hits();           # returns 1, 2, 3
```

Memoisation of an expensive pure function:

```
sub fib {
    my ($n) = @_ ;
    state %cache;
    return $cache{$n} //= $n < 2 ? $n : fib($n-1) + fib($n-2);
}
```

Lazy one-time setup of a read-only table:

```
sub months {
    state @m = qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec);
    return @m;
}
```

Here the initialiser is a simple list, which works because the RHS of `state @m = ...` is constant-folded to a list literal and current Perl permits this form; the more general case of a computed list still wants the `unless (@m) { ... }` pattern above for portability.

Per-closure state (fresh slot per `sub { ... }` evaluation):

```
sub make_counter {
    return sub { state $n = 0; ++$n };
}
my $a = make_counter();
my $b = make_counter();
$a->(); $a->();           # 1, 2
$b->();                 # 1 - independent of $a
```

Declare-and-use in a single expression:

```
while (my $line = <$fh>) {
    warn "first line was: $line" if (state $first //= $line) eq
    ↪$line;
}
```

Edge cases

- **Aggregate initialisers.** `state @a = (1, 2, 3)` and `state %h = (k => 1)` have a long history of being restricted or experimental. When in doubt, declare the aggregate and fill it under `unless (@a) { ... }` on the first call. Scalar initialisers (`state $x = EXPR`) have always been fine.
- **Same-statement reuse.** A fresh state declaration does **not** apply to other mentions of the same name within the same statement; those still resolve to whatever was in scope before the declaration. See the `foo($x, state $x = $x + 1, $x)` example above.
- **Shadowing warning.** Redefining the same name in the same scope - `state $n; state $n; -` shadows the first. Under `use warnings` this emits a warning in the shadow category. Almost always a bug.
- **Anonymous-sub multiplicity.** Every evaluation of `sub { ... }` produces a new CV with its own state slot. If you build many closures from the same source, each

has its own independent state - that is a feature, but it surprises people expecting file-scoped persistence.

- **Threads and forks.** state lives in the CV, which is cloned on fork (copy-on-write in the OS sense - each process continues with its own copy) and on threads->create (deep-cloned). Two processes or threads do not share a state counter.
- **local does not apply.** state variables are lexicals, not package variables, so `local` cannot save and restore them. Wrap the whole sub in its own scope or use a different mechanism if you need dynamic scoping.
- **Access via CORE::state.** When writing code that must parse even without the feature enabled, prefix the keyword: `CORE::state $n = 0;`
- **Destruction.** state variables live as long as the CV. For a named sub in the main program that means “until global destruction.” Do not put a file-handle or a large buffer in a state variable expecting it to be freed at sub exit - it will not be.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `my` - ordinary lexical; reinitialised on every block entry, the natural contrast to state
- `our` - lexical alias for a package variable; use when callers need to see or local-ise the value
- `local` - dynamic scoping for package variables; orthogonal to state and cannot be applied to it
- `sub` - state lives inside a CV; understanding how named vs anonymous subs differ explains per-closure state
- `use` - use v5.10 and later enable the state feature as part of the version bundle
- `undef` - legal placeholder inside state (. . .) list declarations

Regular expressions and pattern matching

study

A no-op retained for source compatibility with older Perl code.

`study` accepts a scalar argument (or operates on `$_` when called with none) and does nothing with it. The call is valid, the argument is evaluated, and control returns immediately. No index is built, no regex engine state is altered, no later match runs any faster or slower because of it.

The name survives because decades of Perl code - books, FAQs, CPAN distributions, in-house scripts - sprinkle `study` in front of long strings about to be matched against many patterns. Those programs still parse and still run. They just no longer get the optimisation the keyword once promised.

Synopsis

```
study SCALAR  
study
```

What you get back

Always 1. The return value is not documented as useful and no idiomatic code inspects it. Treat `study` as returning nothing worth capturing.

History

Through Perl 5.14 `study` built an inverted index of the bytes in `SCALAR`, then biased the regex engine to anchor each subsequent match on the rarest character in the pattern (rarity estimated from a static frequency table baked into the interpreter). The intended workload was “one long string, many `m//` against it”: the index amortised across the matches.

The optimisation was removed in Perl 5.16. It interacted badly with Unicode, `/i` case folding, and the post-5.10 regex engine, and the maintenance cost had long outweighed the narrow workload it helped. The keyword was kept as a no-op so existing source kept compiling.

Examples

Legacy idiom - still parses, no longer has any effect:

```
study $big_text;  
while ($big_text =~ /$pattern/g) {  
    # ...  
}
```

No argument - applied to `$_`, also a no-op:

```
for (@lines) {  
    study;  
    print if /error/;  
}
```

Replacement guidance: if the original code relied on `study` for speed, the modern answer is to compile the pattern once with `qr` and reuse it, or to restructure so the repeated work is the pattern side, not the string side:

```
my $re = qr/$pattern/;  
for my $line (@lines) {  
    print $line if $line =~ $re;  
}
```

Edge cases

- **No effect on regex performance.** Removing every `study` call from a program changes nothing observable, including timing beyond measurement noise.
- **Argument is still evaluated.** `study some_expensive_call()` runs `some_expensive_call()` for its side effects and return value, then discards the result. This is almost never what the author intended.
- **No warning on use.** `study` does not warn under `use warnings` and is not deprecated, so a linter is the only way to flag stale occurrences in a codebase.
- **Not a pragma, not a hint.** `study` is a regular built-in expression. It does not persist across statements and there is no “studied” state attached to the scalar.
- **Tainting and magic are untouched.** `study $tainted` does not launder taint, does not fire `FETCH` on a tied scalar more than a plain read would, and does not mark the scalar in any way.

Differences from upstream

Fully compatible with upstream Perl 5.42. `study` is a no-op in both.

See also

- `qr` - compile a pattern once and reuse the compiled form; this is the modern answer to “make many matches against the same text faster”
- `m` - the match operator whose performance `study` used to influence
- `index` - fixed-string search, faster than a regex when the pattern is a literal substring
- `pos` - where the last `/g` match left off, relevant to the “one long string, many matches” workload `study` once targeted

Control flow

sub

Declare or define a subroutine.

`sub` is the keyword that introduces a subroutine. With a `NAME` and a `BLOCK` it defines a named subroutine in the current package. Without a `BLOCK` it is a forward declaration. Without a `NAME` it is an anonymous `sub` expression that evaluates to a code reference - `sub { ... }` is the only form of `sub` that is actually an expression; the named forms are declarations and return nothing useful.

Synopsis

```
sub NAME BLOCK                                # named definition
sub NAME (PROTO) BLOCK                       # with prototype
sub NAME : ATTRS BLOCK                       # with attributes
sub NAME (PROTO) : ATTRS BLOCK               # prototype + attributes
```

(continues on next page)

(continued from previous page)

```

sub NAME;                                # forward declaration
sub NAME (PROTO);                        # forward with prototype

my $ref = sub BLOCK;                     # anonymous, returns code ref
my $ref = sub (PROTO) : ATTRS BLOCK;    # anonymous with proto + ↵
↵attrs

use feature 'signatures';
sub NAME ($x, $y = 0, @rest) BLOCK      # named with signature
my $ref = sub ($x, $y) BLOCK;           # anonymous with signature

```

What you get back

The **named** forms (sub NAME BLOCK, sub NAME (PROTO) BLOCK, etc.) are declarations. They install the subroutine into the current package's symbol table at compile time and contribute nothing to the surrounding expression. Writing `my $x = sub foo { 1 };` is a common mistake - that is two statements glued together, and `$x` is left undefined.

The **anonymous** form `sub BLOCK` is an expression. It evaluates - at the point the expression is reached at runtime, not at compile time - to a code reference closing over the lexical variables in scope. The same `sub { ... }` expression evaluated twice produces two distinct code references that share nothing:

```
my @refs = map { sub { $_ } } 1..3;    # three independent closures
```

Forward declarations (sub NAME; with no block) do nothing at runtime and return nothing. Their purpose is to announce the name to the parser so that later calls to it can parse without parentheses.

Global state it touches

- The current package (set by `package`) determines the fully-qualified name of a named sub. `sub foo { }` inside package `Acme`; installs `Acme::foo`, not `main::foo`.
- `@_` inside the body holds the argument list (aliased to the caller's scalars, except when a signature is in effect - see *Signatures* below). Assigning to `@_` as a whole breaks the aliasing for the rest of the call.
- `$_[N]` elements are aliases to the caller's scalars. Modifying `$_[0]` modifies the caller's variable; passing a literal and then modifying `$_[0]` is a fatal error.
- `wantarray` inside the body reflects the context the subroutine was called in.

Named definition

```
sub greet {
    my ($name) = @_;
    return "Hello, $name";
}

print greet("world"), "\n";           # Hello, world
```

The body is compiled at compile time. The name is visible from that point in the file onward; forward references inside the same package work because Perl defers method-style and bareword calls.

Redefining a named sub at runtime (eval "sub foo { ... }" or a second sub foo { } in the same package) replaces the previous definition and emits a Subroutine foo redefined warning under use warnings. Suppress it with no warnings 'redefine' for the scope of the redefinition.

Anonymous sub and closures

```
sub make_counter {
    my $n = 0;
    return sub { ++$n };
}

my $c = make_counter();
print $c->(), $c->(), $c->(), "\n";    # 123
```

The inner sub captures \$n from its enclosing scope. Each call to make_counter creates a fresh \$n and a fresh closure over it.

```
my @doubblers = map { my $k = $_; sub { $k * 2 } } 1..3;
print $doubblers[2]->(), "\n";       # 6
```

Note the intermediate my \$k = \$_ - closures capture variables, not values, and \$_ is shared across the map iterations. Without the copy, every closure would see the last value of \$_.

Signatures

With use feature 'signatures' (on by default under use v5.36 and later), the parameter list appears between the name and the block, and arguments are bound to lexical variables automatically:

```
use feature 'signatures';

sub add ($x, $y) { $x + $y }

sub greet ($name, $greeting = "Hello") {
    return "$greeting, $name";
}
```

(continues on next page)

(continued from previous page)

```
sub log_all ($level, @msgs) {  
    print "[$level] $_\n" for @msgs;  
}
```

Signature forms:

- `$x` - required positional parameter.
- `$x = EXPR` - optional positional with default; EXPR is evaluated each call when the argument is missing.
- `$x = undef` - optional, defaults to undef (useful for documenting that a slot is nullable rather than omitted).
- `@rest / %rest` - slurpy, consumes all remaining arguments.
- `$ / @ / %` - unnamed placeholder, asserts arity without binding a variable.

Calling with the wrong arity croaks: Too many arguments for subroutine or Too few arguments for subroutine.

Under a signature, `@_` still exists but accessing it is a warning under `use warnings 'experimental::args_array_with_signatures'` in earlier Perls and discouraged in 5.42. Use the signature variables instead.

Prototypes

```
sub mypush (\@@) {  
    my $aref = shift;  
    push @$aref, @_;  
}  
  
my @a = (1, 2);  
mypush @a, 3, 4;                                # @a is now (1, 2, 3, 4)
```

Prototypes are a **compile-time parser hint**, not a type check. They change how call sites are parsed: `\@` in the prototype above forces the first argument to be an array and passes a reference to it. See [prototype](#) to inspect a sub's prototype.

Prototypes do nothing when the sub is called via `&name(...)` or through a code reference. If you want a signature, use `use feature 'signatures';` prototypes and signatures can coexist with the `:prototype(...)` attribute:

```
use feature 'signatures';  
sub each_pair :prototype(\@) ($aref) { ... }
```

Attributes

Attributes appear after `:` and modify how the sub is compiled or registered:

```

sub critical :lvalue { $state }      # usable on the left of =
sub noop     :method { }             # marked as a method (for
→some tools)

```

Common core attributes: lvalue, method, prototype(...), const. Modules like Attribute::Handlers let a package define its own. Unknown attributes are a compile-time error.

Forward declaration

```

sub later;                               # announce the name
plan later(3);                           # now parses without parens
sub later { ... }                        # defined later in the file

```

A forward declaration is only useful to tell the parser that later is a subroutine so that later ARG parses as a call rather than an indirect-object construct or a comma expression. It is not required if you call with parentheses.

Examples

Named sub, explicit argument unpacking:

```

sub area {
    my ($w, $h) = @_;
    return $w * $h;
}
print area(3, 4), "\n";                  # 12

```

Anonymous sub passed to a higher-order function:

```

my @lengths = map { sub { length shift } }->(@_),
    ("alpha", "beta", "gamma");          # not quite - see below

```

More useful: passing a sub directly:

```

use List::Util qw(first);
my $first_even = first { $_ % 2 == 0 } 1, 3, 4, 7;    # 4

```

Implicit return - the value of the last expression is returned:

```

sub double { $_[0] * 2 }                 # no return needed
print double(21), "\n";                  # 42

```

Context-sensitive return:

```

sub names {
    my @n = qw(alice bob carol);
    return wantarray ? @n : scalar @n;
}

```

(continues on next page)

(continued from previous page)

```
my @all = names();           # ("alice","bob","carol")
my $n   = names();           # 3
```

Modifying caller arguments via @_ aliasing:

```
sub upcase_in {
    for (@_) { $_ = uc }
}
my $s = "hello";
upcase_in($s);
print $s, "\n";              # HELLO
```

Recursion with `__SUB__` - a reference to the current sub without naming it:

```
use feature 'current_sub';
my $fact = sub {
    my $n = shift;
    $n < 2 ? 1 : $n * __SUB__->($n - 1);
};
print $fact->(5), "\n";       # 120
```

Edge cases

- **A named sub definition is a statement, not an expression.** It contributes nothing to its enclosing expression. `my $x = sub foo {};` parses as two statements; `$x` ends up undef.
- **The sub NAME BLOCK forms are compile-time.** The block body is parsed and compiled when the enclosing file or `eval` is compiled. Side effects inside the block run only when the sub is *called*, not when it is defined. By contrast, `sub BLOCK` (anonymous) evaluates at runtime every time the expression is reached, producing a new closure each time.
- **Nested named subs close over nothing useful.** A sub defined inside another sub is still installed in the package at compile time and does **not** capture lexicals from the enclosing call. Each call to the outer sub sees the same inner sub, warped around the *first* call's lexicals. Use an anonymous sub for closures.
- **Trailing return is optional but explicit is clearer.** Without `return`, the value of the last *expression* is returned. If the last statement is a loop or a block, the return value is unspecified - do not rely on it.
- **Assigning to @_ wholesale breaks aliasing.** `@_ = @_` or `@_ = (1, 2)` replaces the array and later `$_[0] = X` no longer reaches the caller.
- **Passing a literal then modifying \$_[0] is fatal.** `upcase_in("x")` dies with Modification of a read-only value attempted because `$_[0]` is aliased to the literal.
- **Redefining a sub emits Subroutine X redefined under use warnings.** Wrapping a redefinition requires `no warnings 'redefine'` for the scope:


```
no warnings 'redefine';
*Foo::bar = sub { ... };
```

- **sub inside a BEGIN block** is defined at compile time of the surrounding unit, same as any sub declaration; the BEGIN wrapping does not change the timing of the declaration itself.
- **A prototype after a space gotcha.** `sub foo ($$) with use feature 'signatures' is parsed as a signature, not a prototype. To attach a prototype under signatures, use sub foo :prototype($$) ($x, $y) { ... }.`

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `return` - explicit return from a sub; without it the last expression's value is returned
- `wantarray` - inspect the calling context (list / scalar / void) from inside the body
- `caller` - information about who called this sub and from where
- `prototype` - read back the prototype of a named or anonymous sub
- `__SUB__` - reference to the current sub without naming it; the tool for anonymous recursion
- `my` - the usual way to unpack `@_` into named lexicals when not using signatures
- `method` - class-syntax method declaration; use feature 'class' equivalent of sub for methods

SCALARs and strings

substr

Extract, replace, or alias a contiguous slice of a string.

`substr` picks out a run of characters from `EXPR` starting at `OFFSET` and spanning `LENGTH` characters, and returns that slice. The same call can also *modify* the original string in three distinct ways: as an lvalue on the left of `=`, through the four-argument `REPLACEMENT` form, or through an aliased lvalue stored in a variable that keeps tracking the slice across subsequent writes. Character zero is the first character. Negative `OFFSET` counts from the end; negative `LENGTH` stops that many characters short of the end. If `LENGTH` is omitted, the slice runs to the end of the string.

Synopsis

```
substr EXPR, OFFSET
substr EXPR, OFFSET, LENGTH
substr EXPR, OFFSET, LENGTH, REPLACEMENT
substr(EXPR, OFFSET, LENGTH) = NEWVALUE           # lvalue form
```

What you get back

In rvalue position, a new scalar containing the extracted slice. In lvalue position, a magic scalar that, when assigned to, splices the new value into EXPR at the recorded position - the returned scalar does **not** hold a copy of the original characters. The four-argument form assigns REPLACEMENT into EXPR in place and returns the substring that was there before, so the extract and the replace happen in one call.

If the requested span lies partially outside the string, only the in-bounds portion comes back. If it lies entirely outside the string, the rvalue forms return `undef` and warn; the lvalue forms raise an exception.

Global state it touches

None. `substr` is pure with respect to interpreter globals. It does not read or write `$_, $/, $\\, $!`, or any other special variable. Warnings about out-of-range access are governed by the usual `use warnings / $^W` scope.

Positions and negative indices

OFFSET is measured from the start of the string when non-negative, and from the end when negative: -1 is the position of the last character, -2 the second-to-last, and so on.

LENGTH, when non-negative, is the number of characters to include. When negative, it is a right-anchor: the slice stops that many characters short of the end of the string. Omitting LENGTH is the same as “to the end.”

```
my $s = "The black cat climbed the green tree";
my $color = substr $s, 4, 5;           # "black"
my $middle = substr $s, 4, -11;        # "black cat climbed the"
my $end = substr $s, 14;               # "climbed the green tree"
my $tail = substr $s, -4;              # "tree"
my $z = substr $s, -4, 2;              # "tr"
```

Using substr as an lvalue

If EXPR is itself an lvalue, `substr(EXPR, OFFSET, LENGTH)` can appear on the left-hand side of an assignment. The right-hand value replaces the slice. The replacement does not have to match LENGTH: a shorter replacement shortens the string, a longer one extends it.

```
my $name = 'fred';
substr($name, 0, 2) = 'SH';            # $name is now "SHed"
substr($name, 1, 2) = 'ayfiel';        # $name is now "Shayfield"
```

Assigning past the current end of the string is allowed, up to one position past the end (append-at-end). Any further and the assignment raises an exception.

```
my $name = 'fred';
substr($name, 4) = 'dy';                # $name is now "freddy"
substr($name, 7) = 'gap';                # exception
```

To keep the string length fixed, pad or truncate the replacement explicitly - for example with `sprintf "%-*s", $len, $val`.

The four-argument replacement form

`substr EXPR, OFFSET, LENGTH, REPLACEMENT` replaces the slice in place and returns the substring that *was* there. This is usually preferable to the lvalue form: the operation is a single expression, and the old value is available as the return value.

```
my $s = "The black cat climbed the green tree";
my $z = substr $s, 14, 7, "jumped from";    # $z is "climbed"
# $s is now "The black cat jumped from the green tree"
```

This mirrors the extract-and-replace idiom of `splice` on arrays.

The aliased-lvalue trick

The scalar returned by the three-argument form is **not** a copy of the characters - it is a magic bullet that remembers which region of the original string it refers to. Stashing that lvalue in a variable (via `for`, or `\` taken through an lvalue accessor) lets repeated writes keep splicing into the same slot of the original string, even as that string changes size:

```
my $x = '1234';
for (substr($x, 1, 2)) {
    $_ = 'a';    print $x, "\n";    # 1a4
    $_ = 'xyz';  print $x, "\n";    # 1xyz4
    $x = '56789';
    $_ = 'pq';   print $x, "\n";    # 5pq9
}
```

With a negative OFFSET, the alias tracks the end of the string:

```
my $x = '1234';
for (substr($x, -3, 2)) {
    $_ = 'a';    print $x, "\n";    # 1a4
    $x = 'abcdefg';
    print $_, "\n";                # f
}
```

This is useful when repeatedly rewriting the same slot of a buffer (header fields in a fixed-layout record, for instance) - but it is also a classic source of surprises. Prefer the four-argument form unless you specifically want the aliasing.

Examples

Extract a fixed field from a record:

```
my $line = "2026-04-22 richard.jelinek@example.com";
my $date = substr $line, 0, 10;    # "2026-04-22"
```

Strip a leading prefix of known length:

```
my $msg = "ERROR: disk full";
my $rest = substr $msg, length("ERROR: "); # "disk full"
```

Overwrite a fixed field in place with the four-argument form, and keep the old value:

```
my $record = "name:      John      age: 42";
my $old     = substr $record, 12, 10, sprintf("%-10s", "Jane");
# $record is now "name:      Jane      age: 42"
# $old is "John      "
```

Walk a string in windows of three characters without copying the whole string each time:

```
my $s = "ABCDEFGH";
for (my $i = 0; $i + 3 <= length $s; $i++) {
    print substr($s, $i, 3), "\n"; # ABC BCD CDE DEF ...
}
```

Rewrite the last byte of a buffer repeatedly via an aliased lvalue:

```
my $buf = "packet: 0";
my $tail = \substr($buf, -1, 1);
$$tail = '1'; # $buf is now "packet: 1"
$$tail = '2'; # $buf is now "packet: 2"
```

Edge cases

- **Out-of-range rvalue, partially in bounds:** returns just the in-bounds portion, no warning.

```
my $s = "abcd";
substr($s, 2, 100); # "cd"
```

- **Out-of-range rvalue, entirely past the end:** returns `undef` and warns under use warnings. The empty-tail case (exactly at the end) returns `"` without a warning:

```
my $name = 'fred';
my $null = substr $name, 4, 2; # "" (no warning)
my $oops = substr $name, 7;   # undef, with warning
```

- **Out-of-range lvalue:** raises an exception. `substr` refuses to silently extend the string by a gap of unspecified contents.
- **Negative OFFSET larger than the string:** counts as “before the start”, and is clamped to position 0 in current Perls; under use warnings this issues a “substr outside of string” warning in some versions. Prefer to clamp explicitly.
- **Negative LENGTH that exceeds the remaining tail:** yields an empty string, because the right anchor falls at or before the left anchor.

```
my $s = "abcde";
substr($s, 1, -10); # ""
```

- **Unicode:** `substr` operates in characters, not bytes. On a string with the UTF-8 flag on, positions count codepoints; on a byte string, positions count bytes. Mixing the two by treating a decoded string as a byte buffer produces wrong offsets - decode first, then index.
- **Tied or magical scalars:** `lvalue substr` honours STORE-time magic on the target, so tied variables see the mutated whole string, not the slice.
- **Aliased lvalue kept beyond its target's lifetime:** if the original variable goes out of scope while the alias is still alive, later writes through the alias act on a freed string; keep the alias lifetime inside the target's.
- **Pre-5.10 corner:** repeated writes through a single `lvalue` had unspecified behaviour in older perls, as did negative offsets before 5.16. Modern Perl (and `ppperl`) follow the semantics documented on this page.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- [index](#) - find the position of a substring, to feed into `substr` as `OFFSET`
- [rindex](#) - same, searching from the right
- [length](#) - the upper bound on any meaningful `OFFSET` or `LENGTH` you pass to `substr`
- [splice](#) - the array-level analogue of the four-argument replacement form
- [pack](#) / [unpack](#) - for fixed-layout records where many fields are read or rewritten in one shot, prefer these over a chain of `substr` calls
- [tr///](#) - bulk character-level substitution that doesn't need positional indices

[Filehandles, files, directories](#)

symlink

Create a symbolic link at `NEWFILE` that points at the string `OLDFILE`.

`symlink` is the thin wrapper around the POSIX `symlink(2)` system call. It does not copy anything, does not read `OLDFILE`, and does not require `OLDFILE` to exist. It writes a new directory entry at `NEWFILE` whose payload is the literal bytes of `OLDFILE`, and the kernel later resolves that payload whenever something opens, stats, or traverses the link.

Synopsis

```
symlink OLDFILE, NEWFILE
```

What you get back

1 on success, 0 on failure (with `$!` set). On systems that do not support symbolic links at all, `symlink` raises an exception rather than returning 0. To probe for support at runtime without dying, wrap a harmless call in `eval`:

```
my $symlink_exists = eval { symlink("", ""); 1 };
```

The `eval` pattern is the idiomatic capability check - it runs a no-op `symlink` whose only job is to fail cheaply on platforms where the call exists, and to throw on platforms where it does not.

How symlinks differ from hard links

A symlink and a hard link (`link`) look similar from the shell but behave very differently:

- `link` creates a second directory entry pointing at the same inode. Both names are equal; deleting either one leaves the file alive under the other.
- `symlink` creates a tiny file of type `lnk` whose contents are a *pathname*. The target is resolved by name at every lookup. The two names are **not** equal - the symlink has its own inode, its own permissions bits (mostly ignored), and its own timestamps.

Practical consequences:

- `OLDFILE` is stored **verbatim**. It is not canonicalised, not resolved against the current working directory, and not checked for existence. A symlink can be created pointing at a path that does not exist yet (a *dangling* symlink) and will start working the moment the target appears.
- Because `OLDFILE` is just a string, it can be relative. Relative symlink targets are resolved at lookup time relative to the directory containing `NEWFILE`, **not** the process's current working directory. This is the single biggest source of surprise.
- Symlinks can cross filesystem boundaries. Hard links cannot.
- Symlinks can point at directories. Hard links to directories are forbidden on every mainstream Unix.
- `unlink` on a symlink removes the **link**, not the target. To remove the target, `unlink` the target's path directly.

Examples

Create a symlink to an existing file:

```
symlink "/etc/hostname", "hostname.lnk"  
or die "symlink failed: $!";
```

Dangling symlink - the target does not exist yet, and that is fine:

```
symlink "/will/appear/later", "pending.lnk"  
or die "symlink failed: $!";  
# opening pending.lnk now fails with ENOENT;  
# it will start working once /will/appear/later exists.
```

Relative target resolved against the link's directory, not the cwd:

```
chdir "/var/log" or die $!;
symlink "syslog", "/tmp/current.lnk"
    or die "symlink failed: $!";
# readlink("/tmp/current.lnk") eq "syslog"
# opening /tmp/current.lnk resolves syslog in /tmp, not /var/log.
```

Symlink to a directory - this works, unlike link:

```
symlink "/var/log", "logs"
    or die "symlink failed: $!";
opendir my $dh, "logs" or die $!; # traverses the symlink
```

Inspect and remove a symlink without touching its target:

```
my $target = readlink "hostname.lnk"; # the stored OLDFILE string
unlink "hostname.lnk" or die $!;      # removes the link only
# /etc/hostname is untouched.
```

Atomic-swap pattern - replace a symlink pointing at the “current” release directory by writing a new symlink and renaming over it:

```
symlink "release-2", "current.new" or die $!;
rename "current.new", "current"    or die $!;
# rename over an existing symlink is atomic on POSIX filesystems.
```

Edge cases

- **OLDFILE is stored verbatim.** No normalisation, no resolution, no existence check. `symlink "", "foo"` succeeds on most systems and creates an empty-target symlink that every lookup rejects with `ENOENT`. This is why the capability-probe idiom uses `("", "")`
 - a cheap call that always fails on supporting platforms.
- **Relative OLDFILE** is resolved relative to the directory of `NEWFILE`, not the current working directory. If you build `NEWFILE` from a relative path, think carefully about what directory the kernel will see it in.
- **NEWFILE already exists** → failure with `$!` set to `EEXIST`. There is no “replace” flag; to overwrite, write to a fresh name and `rename` over the old one (atomic).
- **Permissions on the symlink itself** are almost always ignored. The kernel enforces the permissions of the *target*. On most filesystems the symlink is reported as `lrwxrwxrwx` regardless of what `chmod` is called on it.
- **stat follows symlinks, lstat does not.** To examine the link itself - its type, its link count, the length of its target string - use `lstat`. To examine what the link points at, use `stat`.
- **unlink removes the link, not the target.** There is no way to delete the target through the symlink except by resolving it first.

- **Platforms without symlink support** raise an exception on any call, even one that would otherwise fail. Use the `eval` probe shown above rather than assuming a return value.
- **Filesystems without symlink support** (some FAT variants, certain network mounts) fail with `EPERM` or `EOPNOTSUPP` even on a kernel that implements `symlink(2)`. Check `$!` on failure rather than assuming the call was rejected for the obvious reason.
- **Symlink loops** are not detected at creation time. `symlink "a", "b"` followed by `symlink "b", "a"` succeeds; the loop only surfaces when something tries to resolve either name, which fails with `ELOOP` after the kernel hits its chase limit (typically 40).

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `link` - create a hard link (same inode, same filesystem, file only); the strict counterpart to `symlink`
- `readlink` - read the `OLDFILE` string stored in a symlink without resolving it
- `lstat` - stat the symlink itself rather than its target; the only way to see that a path is a symlink from Perl
- `stat` - stat what the symlink points at, following the chain of links
- `unlink` - remove a symlink (the link, not the target); also the way to remove a dangling symlink
- `$!` - `errno` set on failure; inspect it to distinguish `EEXIST`, `EPERM`, `EOPNOTSUPP`, `ENOENT` on the parent directory

I/O · Fixed-length data

syscall

Invoke a raw system call by its kernel number, passing the remaining arguments as `int` or as a pointer to a string buffer.

`syscall` is the escape hatch for operating-system calls the rest of the language does not wrap. The first argument is the syscall number - on Linux, the value of `SYS_<name>` from `syscall.ph`. Each remaining argument is passed either as an integer (if Perl already considers it numeric) or as a pointer to the scalar's string buffer. It returns whatever the kernel returns. On failure it returns `-1` and sets `$!`.

Synopsis

```
syscall NUMBER, LIST
```


What you get back

Whatever the kernel returned. Success values are syscall-specific - often a byte count, a file descriptor, or 0. Failure is -1 with `$!` set to the `errno` value.

Some syscalls (`lseek`, read on a pipe at EOF with errors, `ptrace`) can legitimately return -1 on success. To tell them apart from failures, clear `$!` before the call:

```
$! = 0;
my $r = syscall(SYS_lseek(), fileno($fh), 0, 1);
die "lseek: $!" if $r == -1 && $! != 0;
```

Global state it touches

- `$!` - set when the kernel returns -1. Not touched on success.

Argument passing rules

Each argument in `LIST` is inspected at call time:

- **Numeric** argument (a literal number, or a scalar that has been used in a numeric context) - passed as a C `int`.
- **Non-numeric** argument - the scalar's string buffer is passed by pointer. You are responsible for pre-extending the buffer to hold any data the kernel might write. A too-short buffer is a memory corruption bug, not a Perl error.
- **Read-only string** (a literal, a `__DATA__` string, anything Perl refuses to write through) - rejected, because `syscall` has to assume any string pointer might be written through.
- **Integer stored as a string** (e.g. a value read from a file that was never used arithmetically) - may be passed as a pointer rather than an `int`. Force numeric interpretation by adding 0:

```
syscall(SYS_write(), 0 + $fd, $buf, length $buf);
```

Up to 14 arguments are supported after the syscall number.

Examples

Write a buffer to `STDOUT` using the raw `write(2)` syscall:

```
require 'syscall.ph';
my $s = "hi there\n";
syscall(SYS_write(), fileno(STDOUT), $s, length $s);
```

Pre-extend a buffer before a read:

```
require 'syscall.ph';
my $buf = "\0" x 4096;           # 4 KiB writable buffer
my $n = syscall(SYS_read(), fileno($fh), $buf, 4096);
die "read: $!" if $n == -1;
substr($buf, $n) = '';          # trim to actual bytes read
```

Call `gettid(2)` to get the kernel thread id (no libc wrapper on older glibc):

```
require 'syscall.ph';
my $tid = syscall(SYS_gettid());
```

Distinguish a legitimate -1 return from a real failure:

```
#!/ = 0;
my $off = syscall(SYS_lseek(), fileno($fh), 0, 1); # SEEK_CUR
if ($off == -1 && $! != 0) {
    die "lseek failed: $!";
}
```

Force a stringified integer to be passed as int:

```
my $fd = read_config("fd"); # returns "3"
syscall(SYS_close(), 0 + $fd); # without "0 +" it'd pass a
    ↪ pointer
```

Edge cases

- **syscall.ph missing:** `require 'syscall.ph'` fails unless `h2ph` has been run against the system headers. Without it the `SYS_*` constants are undefined and calls become hand-maintained magic numbers.
- **Read-only argument:** passing a string literal or other read-only scalar croaks with Modification of a read-only value. Copy into a lexical first: `my $s = "literal"; syscall(..., $s, ...)`.
- **Buffer too short:** the kernel will happily write past the end of a scalar's buffer if you lied about the length. This corrupts the Perl heap. Always pre-extend with `$buf = "\0" x $n` or `vec($buf, $n-1, 8) = 0` before the call, and never pass length `$buf` larger than the actual capacity.
- **Argument count ceiling:** 14 arguments after `NUMBER`. Syscalls that take more (rare) are not reachable via `syscall`.
- **SYS_pipe() on Linux:** the raw `syscall` returns the read-end fd but gives you no handle on the write-end fd. Use `pipe` instead, which exposes both.
- **64-bit values on 32-bit platforms:** some syscalls take 64-bit arguments (offsets, sizes) that do not fit in a Perl int on 32-bit builds. There is no portable way to pass them through `syscall` - use a dedicated wrapper (`Fcntl::lseek`, `sysseek`, etc.) or a CPAN binding.
- **Unimplemented:** on platforms without a `syscall(2)` entry point (or when built without support), `syscall` raises The `syscall` function is unimplemented.
- **Signal interruption:** a `syscall` interrupted by a signal returns -1 with `$!` set to `EINTR`. `syscall` does not restart automatically - retry in user code if needed.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `syswrite` - buffered-bypass write through a Perl filehandle; almost always what you want instead of `syscall(SYS_write(), ...)`
- `sysread` - buffered-bypass read; pairs with `syswrite` and handles buffer management for you
- `sysopen` - `open(2)` with the full flag vocabulary, without needing the raw `syscall`
- `pipe` - the right way to create a pipe; avoids the `SYS_pipe()` write-end problem
- `pack` - build the binary structs (`struct timeval`, `struct stat`, etc.) that syscalls expect in pointer arguments
- `unpack` - decode the binary structs a syscall writes into your buffer

Filehandles, files, directories

sysopen

Open a file the low-level way, passing an integer `MODE` bitmask straight through to the underlying `open(2)` system call.

`sysopen` is the escape hatch for when `open`'s string-mode DWIM is in your way - when you need `O_EXCL` for an atomic create-if-not-exist, `O_NOFOLLOW` to refuse symlinks, `O_NONBLOCK` on a FIFO, or any other flag that has no corresponding `"<" / ">" / ">>"` string. The flag constants live in `Fcntl` and are combined with `|`.

Synopsis

```
sysopen FILEHANDLE, FILENAME, MODE
sysopen FILEHANDLE, FILENAME, MODE, PERMS
```

What you get back

1 on success, `undef` on failure (with `$!` set to the `errno` from the failed syscall). Unlike `open`, there is no magic - `FILENAME` is taken literally, no leading `>`, `<`, `|`, or `-` is interpreted, no layered mode string is parsed.

If `FILEHANDLE` is an undefined scalar, it is autovivified into a fresh handle:

```
sysopen my $fh, $path, O_RDONLY
or die "sysopen $path: $!";
```

MODE - the Fcntl bitmask

MODE is an integer built by OR-ing constants from `Fcntl1`. The access mode is mandatory and mutually exclusive; the modifier flags are optional and combined freely:

Flag	Meaning
<code>O_RDONLY</code>	Open for reading only.
<code>O_WRONLY</code>	Open for writing only.
<code>O_RDWR</code>	Open for reading and writing.
<code>O_CREAT</code>	Create the file if it does not exist.
<code>O_EXCL</code>	With <code>O_CREAT</code> : fail if the file already exists.
<code>O_APPEND</code>	Every write seeks to end-of-file first (atomic append).
<code>O_TRUNC</code>	Truncate the file to zero length on open.
<code>O_NONBLOCK</code>	Non-blocking mode for the resulting descriptor.
<code>O_NOFOLLOW</code>	Fail with <code>ELLOOP</code> if the final path component is a symlink.

Import them by name or with a tag:

```
use Fcntl qw(O_RDWR O_CREAT O_EXCL O_APPEND);  
# or  
use Fcntl ':DEFAULT';
```

The legacy values 0, 1, 2 for read-only, write-only, read-write also work on every system Perl supports, but name the constants - the resulting code survives cross-platform review and reads without a glossary.

PERMS - the create-time permission bits

PERMS is an octal mode (like `chmod` takes) applied to the inode **only when `O_CREAT` actually creates the file**. If PERMS is omitted, Perl passes 0666. The process `umask` is applied by the kernel on top, so a default `umask` of 022 yields 0644 on disk.

```
sysopen my $fh, $path, O_WRONLY | O_CREAT | O_EXCL, 0600  
or die "create $path: $!";
```

Do not write 0644 as the PERMS argument without a reason. A hard-coded 0644 strips group-write even from users who deliberately set a permissive `umask`; 0666 plus the user's `umask` is the portable default and is what `sysopen` uses when you omit the argument entirely.

Global state it touches

- `$!` - set on failure to the `errno` returned by `open(2)`.
- `${^OPEN}` - the `open` pragma's layer configuration. `sysopen` applies the **same** default PerlIO stack `open` would apply for a layer-less call. Use `binmode` immediately after a successful `sysopen` when you need to override or strip layers (typical for binary files, sockets-as-files, or `O_NONBLOCK` descriptors).
- Process `umask` - masks PERMS when `O_CREAT` creates the file.

Examples

Atomic create-if-not-exist - the canonical reason to reach for sysopen:

```
use Fcntl qw(O_WRONLY O_CREAT O_EXCL);
sysopen my $fh, "lock.pid", O_WRONLY | O_CREAT | O_EXCL, 0644
    or die "another instance is already running: $!";
print $fh "$$\n";
close $fh;
```

Open-or-create for read/write, preserving existing contents:

```
use Fcntl qw(O_RDWR O_CREAT);
sysopen my $fh, $path, O_RDWR | O_CREAT, 0666
    or die "sysopen $path: $!";
```

Append-only log with the append guarantee - every write lands at EOF regardless of interleaving with other writers:

```
use Fcntl qw(O_WRONLY O_CREAT O_APPEND);
sysopen my $log, "/var/log/app.log", O_WRONLY | O_CREAT | O_APPEND,
    ↪0644
    or die "open log: $!";
print $log "startup\n";
```

Open a FIFO without blocking on the reader-side handshake:

```
use Fcntl qw(O_RDONLY O_NONBLOCK);
sysopen my $fifo, "/tmp/q", O_RDONLY | O_NONBLOCK
    or die "sysopen fifo: $!";
```

Refuse to follow a symlink at the leaf - useful in directories writable by untrusted users:

```
use Fcntl qw(O_RDONLY O_NOFOLLOW);
sysopen my $fh, "$dir/config", O_RDONLY | O_NOFOLLOW
    or die "sysopen config: $!";
```

Force byte-level I/O after opening - strip any :utf8 or :crlf layer the default PerlIO stack would otherwise install:

```
sysopen my $fh, $path, O_RDONLY or die $!;
binmode $fh;
```

Edge cases

- **O_EXCL without O_CREAT is a no-op.** The exclusivity check only triggers when O_CREAT asks the kernel to create the file. Always write them together: O_CREAT | O_EXCL.
- **O_EXCL is not a lock.** It prevents a successful open on an already-existing file, nothing more. Once the file exists, every subsequent sysopen with O_CREAT | O_EXCL fails. It does not serialize access to the contents - use `flock` for that.

- **O_EXCL on network filesystems.** NFS implementations vary; older NFS versions silently lose the exclusive semantics. Do not rely on O_EXCL across NFSv2.
- **O_CREAT | O_EXCL and symlinks.** With both flags set, the kernel refuses to open a pre-existing symlink at the final path component. It does **not** protect intermediate path components; use O_NOFOLLOW or resolve the directory with `opendir` plus `openat` semantics if you need that.
- **O_TRUNC with O_RDONLY.** Behavior is undefined by POSIX. Do not combine them.
- **Omitted PERMS.** The default is 0666, masked by `umask`. It is **not** 0644. Omit PERMS unless you actively want to override the user's `umask`.
- **FILEHANDLE as expression.** If FILEHANDLE is a bareword, it names a package global. If it is an expression evaluating to a glob, globref, or `IO::Handle`, that handle is opened. If it is an undefined scalar, a fresh handle is autovivified into it - the lexical form used throughout these examples.
- **No magic on FILENAME.** A FILENAME of `"-"` opens a file literally named `-`, not STDIN. A leading `>` opens a file literally named `>filename`, not a write to `filename`. This is the entire point of `sysopen` versus `open`.
- **Default PerlIO layers still apply.** `sysopen` is “low-level” relative to the mode string, not relative to PerlIO. The resulting handle has the same layer stack as a layer-less `open`; call `binmode` to change it.
- **Closed or invalid FILEHANDLE expression.** Returns `undef` with `$!` set (typically EBADF for a bad existing descriptor; ENOENT, EACCES, EEXIST, ELOOP for the path-level errors).

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `open` - higher-level opener with string-mode parsing, pipes, `-` as STDIN / STDOUT, and layered modes; use it unless you specifically need a flag `sysopen` exposes
- `fcntl` - change flags on an already-open descriptor (e.g. set O_NONBLOCK after the fact); shares the `Fcntl` constant vocabulary
- `sysread` - unbuffered read; works on any handle, not just one obtained from `sysopen`, despite the name
- `syswrite` - unbuffered write, same note
- `Fcntl` - the module that exports `O_*` constants and the `:DEFAULT` / `:flock` / `:mode` tags
- `binmode` - set or strip PerlIO layers on the handle after opening
- `umask` - the process mask that subtracts bits from PERMS when O_CREAT creates the file

I/O · Fixed-length data

sysread

Read raw bytes from a filehandle by calling the underlying `read(2)` system call.

`sysread` tries to read `LENGTH` bytes from `FILEHANDLE` and store them in `SCALAR`. It bypasses the PerlIO buffering stack, which is the whole point - reads go straight to the kernel and return whatever a single `read(2)` returns, which may be fewer bytes than requested. Use it for sockets, pipes, devices, and any other situation where you want control over individual system calls rather than line- or record-oriented input.

Synopsis

```
sysread FILEHANDLE, SCALAR, LENGTH
sysread FILEHANDLE, SCALAR, LENGTH, OFFSET
```

What you get back

- The number of bytes actually read, which may be **less than** `LENGTH` on sockets, pipes, ttys, and during signal interruption.
- `0` at end of file.
- `undef` on error, with `$!` set to the `errno` from the failed `read(2)`.

`SCALAR` is grown or shrunk so that the last byte read becomes the last byte of the scalar. After a short read, the scalar holds only the bytes that were actually received - nothing is zero-padded to `LENGTH`.

```
my $n = sysread $sock, my $buf, 4096;
defined $n      or die "read failed: $!";
$n == 0        and return;           # peer closed
process_chunk($buf);                 # length($buf) == $n
```

Global state it touches

- `$!` - set when `sysread` returns `undef`. Interrupted reads show up as `EINTR`; would-block reads on non-blocking handles show up as `EAGAIN` / `EWOULDBLOCK`.
- `SCALAR` itself - always assigned to, even on failure (it is truncated to `OFFSET` bytes when `OFFSET` is given and the read fails or returns zero).
- The filehandle's byte position advances by the number of bytes returned; this is the same kernel-level position manipulated by `sysseek`.

`sysread` does not consult or touch `$/`, `$\`, `$,`, or `$_`. It is a byte-level primitive; record semantics do not apply.

OFFSET - where the bytes land inside SCALAR

Without `OFFSET`, `SCALAR` is replaced by the bytes read.

With `OFFSET`, the bytes are written **into** `SCALAR` starting at that position:

- `OFFSET >= 0` - start writing at byte `OFFSET`. If `OFFSET` is greater than the current length of `SCALAR`, the string is first padded with `"\0"` bytes up to `OFFSET`, then the new data is appended.
- `OFFSET < 0` - counted backwards from the end of `SCALAR`. `-1` means “overwrite the last byte”, `-10` means “overwrite the last 10 bytes”.

This is how you grow a buffer incrementally without repeatedly concatenating:

```
my $buf = "";
while ((my $n = sysread $fh, $buf, 8192, length $buf) > 0) {
    # each call appends at the current end of $buf
}
defined $n or die "read failed: $!";
```

Examples

Read up to 64 bytes from a file. `$buf` ends up exactly `$n` bytes long:

```
open my $fh, "<", "input.bin" or die $!;
my $n = sysread $fh, my $buf, 64;
defined $n or die "sysread: $!";
```

Read up to 32 bytes and place them starting at position 512 in `$buf`, padding with `"\0"` if `$buf` was shorter:

```
my $buf = "header";
my $n = sysread $fh, $buf, 32, 512; # $buf is now 512 + $n bytes
```

Drain a socket until the peer closes. Checking for `0` - not `eof` - is the only correct end-of-stream test for `sysread`:

```
while (1) {
    my $n = sysread $sock, my $chunk, 4096;
    defined $n or die "read: $!";
    last if $n == 0; # orderly shutdown from peer
    handle($chunk);
}
```

Restart a read interrupted by a signal. On a blocking handle, `EINTR` is the one failure you typically retry rather than propagate:

```
use Errno qw(EINTR);
my $n;
RETRY: {
    $n = sysread $fh, my $buf, $want;
    redo RETRY if !defined $n && $! == EINTR;
}
defined $n or die "read: $!";
```

Fill a fixed-size record, tolerating short reads. A single `sysread` can return fewer bytes than asked for even on a regular file near EOF, and routinely does on sockets and pipes:


```

sub read_exact {
    my ($fh, $want) = @_;
    my $buf = "";
    while (length($buf) < $want) {
        my $n = sysread $fh, $buf, $want - length($buf), length
        ↪ $buf;
        defined $n      or die "read: $!";
        $n == 0 and die "short read: got ", length $buf, " of $want
        ↪ ";
    }
    return $buf;
}

```

Edge cases

- **Never mix with buffered I/O on the same handle.** `readline`, `read`, `<$fh>`, `getc`, `eof`, `seek`, and `tell` all go through PerlIO buffers; `sysread` goes around them. Interleaving them on the same handle leaves data stranded in the buffer that `sysread` will never see, or skips over data the buffered reader already consumed. Use one style per handle.
- **:utf8 is forbidden.** A handle with a `:utf8` layer (including the implicit layer added by `:encoding(...)`) makes `sysread` throw an exception. Strip the layer with `binmode $fh` with no second argument restores raw bytes - before calling `sysread`.
- **:crlf / :perlio still buffer.** Even without `:utf8`, the default `:perlio` stack buffers reads and translates line endings under `:crlf`. `sysread` ignores those layers, so bytes the buffered reader already pulled into the PerlIO buffer are silently lost to `sysread`. Open handles destined for `sysread` with `sysopen` or call `binmode $fh` to disable `:crlf`.
- **Short reads are normal, not errors.** A return of `$n` with $0 < n < \text{LENGTH}$ is success. Only `undef` indicates failure; `0` indicates EOF. Loop until you have what you need.
- **No syseof.** There is no separate end-of-file test for `sysread`; `eof` looks at the PerlIO buffer and is meaningless here. The return value of `0` is the end-of-file signal.
- **OFFSET beyond current length.** `sysread $fh, $buf, 10, 1_000_000` pads `$buf` with a million `"\0"` bytes before writing. This is occasionally useful for pre-allocating, but usually a bug.
- **Negative OFFSET** counts from the current end of SCALAR, not from LENGTH. `sysread $fh, $buf, 4, -4` overwrites the last four bytes of whatever `$buf` currently holds.
- **Non-blocking handles.** On a handle with `O_NONBLOCK` set, a read with nothing available returns `undef` with `$!` equal to `EAGAIN` or `EWOULDBLOCK`. Treat that as “try again later”, not as an error.
- **Signals.** A blocking `sysread` interrupted by a signal returns `undef` with `$!` equal to `EINTR`. Retry the call.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `syswrite` - the write-side counterpart; same buffering rules, same short-I/O semantics
- `sysopen` - opens a handle with `raw open(2)` flags, the natural partner for `sysread` on files
- `sysseek` - byte-accurate seek that composes correctly with `sysread`; do not mix `seek` with `sysread`
- `read` - the buffered counterpart; use it for line- and record-oriented input, not for sockets or partial reads
- `binmode` - strip `:utf8` or `:crlf` from a handle before using `sysread`
- `$!` - the `errno` value after a failed `sysread`, including `EINTR`, `EAGAIN`, and `EWOULD-BLOCK`

I/O · Fixed-length data

`sysseek`

Reposition a filehandle at the system level, bypassing PerlIO buffering.

`sysseek` moves `FILEHANDLE`'s kernel file-descriptor offset to a new byte position by calling the underlying `lseek(2)` directly. It is the companion of `sysread` and `syswrite`: the three form the unbuffered I/O family, which talks to the kernel without going through Perl's `:perlio` or `stdio`-like buffer stack. Use `sysseek` on handles you access with the `sys`-family; use `seek` on handles you access with `read`, `readline`, or `print`.

Synopsis

```
sysseek FILEHANDLE, POSITION, WHENCE
sysseek $fh, 0, 0           # rewind to start
sysseek $fh, 0, 1           # report current position
sysseek $fh, -1024, 2       # 1024 bytes before EOF
```

What you get back

The new absolute byte offset on success, or `undef` on failure (with `$!` set). A position of zero is returned as the dualvar string `"0 but true"` so that the value prints as `0` in numeric context yet remains truthy in boolean context - a check like `if (sysseek ...)` is safe at offset zero.

```
defined(sysseek $fh, $offset, 0)
  or die "sysseek to $offset failed: $!";
```

Because the return value carries the new offset, `sysseek` doubles as a “where am I” primitive - the reason the idiomatic `sys tell` is one line:

```
use Fcntl 'SEEK_CUR';
sub systell { sysseek($_[0], 0, SEEK_CUR) }
```

WHENCE values

WHENCE is an integer with three meaningful values. Prefer the symbolic constants from `Fcntl` - they also make the call portable to platforms that do not use `0 / 1 / 2`:

- `0 / SEEK_SET` - POSITION is measured from the **start of the file**. POSITION must be non-negative.
- `1 / SEEK_CUR` - POSITION is added to the **current position**. Negative values move backward, positive values forward. `sysseek $fh, 0, 1` returns the current offset without moving the pointer.
- `2 / SEEK_END` - POSITION is added to the **end-of-file** offset. POSITION is typically zero or negative.

```
use Fcntl qw(SEEK_SET SEEK_CUR SEEK_END);

sysseek $fh, 0, SEEK_SET;    # rewind
sysseek $fh, 0, SEEK_END;    # move to EOF
sysseek $fh, -$n, SEEK_CUR;  # $n bytes back from here
```

Bytes, not characters

`sysseek` operates on **byte offsets**, always. Even when the handle has a character-oriented layer such as `:encoding(UTF-8)`, the offset it accepts and returns is a raw byte count. This matches the kernel view but is irrelevant in practice: `sysseek` is meant for handles that do not use PerlIO layers, and mixing it with a decoding layer is one of the confusion traps listed below.

If the data is character-oriented, use `seek` and `tell` on the buffered side instead of `sysseek`.

Do not mix with buffered I/O

`sysseek` talks directly to the file descriptor. `read`, `readline`, `print`, `write`, `seek`, `tell`, and `eof` talk to the PerlIO buffer sitting in front of the descriptor. The buffer's position and the kernel's position are independent - a `sysseek` moves the kernel pointer but does not flush or invalidate the buffer, so the next buffered read returns stale bytes from before the `sysseek`, and the next buffered write lands wherever the buffer thought it was.

The rule is simple: pick one family per handle.

- Handle used with `sysread` / `syswrite` → reposition with `sysseek`.
- Handle used with `read` / `print` / `readline` → reposition with `seek`.

Global state it touches

- `$!` - set to the system error message on failure.
- Unlike `seek`, `sysseek` does **not** flush or discard the PerlIO buffer and does **not** clear the EOF flag on the handle. That asymmetry is the source of the “mixing” hazard above and the reason the two families must not share a handle.

Examples

Random-access read of a fixed-length record. The file holds 128-byte records; jump straight to record 42:

```
use Fcntl 'SEEK_SET';
sysopen my $fh, $path, O_RDONLY or die $!;
sysseek $fh, 42 * 128, SEEK_SET or die "sysseek: $!";
sysread $fh, my $rec, 128 or die "sysread: $!";
```

Report the current offset without moving the pointer - the `sys tell` idiom:

```
use Fcntl 'SEEK_CUR';
my $pos = sysseek $fh, 0, SEEK_CUR;
defined $pos or die "sysseek: $!";
print "at byte $pos\n";
```

Append to a file held open for read-write, then rewind to replay from the start:

```
use Fcntl qw(SEEK_SET SEEK_END);
sysseek $fh, 0, SEEK_END or die $!;
syswrite $fh, $record or die $!;
sysseek $fh, 0, SEEK_SET or die $!;
```

Zero is still true - the `dualvar` lets you write the terse form without accidentally treating a successful rewind as a failure:

```
if (my $pos = sysseek $fh, 0, 0) {
    # entered even when $pos stringifies to "0"
    printf "rewound; pos=%d\n", $pos;
}
else {
    die "sysseek failed: $!";
}
```

Seek past end-of-file to create a sparse hole, then write the tail marker:

```
use Fcntl 'SEEK_SET';
sysseek $fh, 1_000_000, SEEK_SET or die $!;
syswrite $fh, "END" or die $!;
```

Edge cases

- **Unseekable handles:** pipes, sockets, TTYs, and most special devices fail with `$!` set to `ESPIPE` (Illegal seek). The return value is `undef`; always check with `defined`.
- **Do not mix with buffered I/O on the same handle.** `sysseek` bypasses the PerlIO buffer; `read`, `readline`, `print`, `write`, `seek`, `tell`, and `eof` use it. Interleaving the two families produces silently wrong results. Pick one family per handle and stay with it.
- **Zero stringifies as "0 but true".** Boolean tests work at offset zero - if `(sysseek ...)` stays truthy - but string comparisons against the literal `"0"` do not:

```
my $pos = sysseek $fh, 0, 0;
$pos == 0      or die;      # true
$pos eq "0"    and die "nope"; # false - the string is "0 but true"
```

Use `defined` for success/failure detection and numeric comparison for the offset value.

- **Encoding layers:** `sysseek` on a handle with an `:encoding(...)` layer is almost always a mistake - the byte offset can land mid codepoint, and the sys-family is meant for raw bytes anyway. If the data needs decoding, use `seek` with the buffered family.
- **Negative offsets with `SEEK_SET`:** `POSITION` must be non-negative when `WHENCE` is `0`. A negative value fails with `EINVAL`.
- **Very large files:** `sysseek` returns the kernel's full 64-bit offset; positions beyond 2 GiB round-trip correctly through subsequent `sysseek` calls on all supported 64-bit builds.
- **Closed filehandle:** returns `undef` and sets `$!`; under `use warnings` a `sysseek()` on closed filehandle warning is emitted.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `seek` - buffered counterpart; use it on handles accessed via `read`, `readline`, or `print`
- `sysread` - the unbuffered read paired with `sysseek`; both skip PerlIO and speak directly to the file descriptor
- `syswrite` - the unbuffered write; the third member of the family that belongs on the same handle as `sysseek`
- `sysopen` - opens a handle at the descriptor level, the usual way to get a handle you will later drive with the sys-family

- `tell` - buffered-side current-offset query; pair it with `seek`, not with `sysseek`
- `Fcntl` - source of the `SEEK_SET`, `SEEK_CUR`, `SEEK_END` constants

Processes

system

Run a separate program and wait for it to finish.

`system` does exactly the same thing as `exec`, except that a `fork` is done first and the parent process waits for the child to exit. The return value is the wait status of the child, not its output - if you want the output of a command, use backticks or `qx//` instead.

Synopsis

```
system LIST
system PROGRAM LIST
```

The indirect `PROGRAM LIST` form lies to the program about its own name: `PROGRAM` is the file actually executed, while the first element of `LIST` becomes `argv[0]` inside the child. Same trick as `exec`.

What you get back

The return value is the raw status word from the underlying `wait` call - the same value that lands in `$?` after the call. To recover the program's actual exit code, shift right by eight:

```
system("make", "install");
my $exit = $? >> 8;
```

-1 means the program could not be started at all (or the `wait` itself failed); inspect `$!` for the reason. A return of 0 means the child exited successfully with status 0.

Shell-vs-direct execution

Argument processing depends on the shape of `LIST`, and mirrors `exec` exactly:

- **More than one argument** (or an array with more than one element): the first element is the program, the rest are its arguments. The shell is **not** invoked; `execvp` runs the program directly. This is the safe form.
- **Exactly one scalar argument**: Perl scans it for shell metacharacters. If any are present, the entire string is passed to `/bin/sh -c` for parsing. If none are present, Perl splits on whitespace and calls `execvp` directly - saving a shell.

Prefer the list form whenever the arguments come from untrusted input or contain characters the shell would interpret:

```
system("ls", "-l", $dir);           # safe, no shell
system("ls -l $dir");               # shell parses $dir - hazardous
```

Signals vs. exit

The status word combines three pieces of information: a signal number (if the child died on a signal), a core-dump flag, and the exit value. Decode it explicitly when you need to tell them apart:

```
system(@cmd);
if ($? == -1) {
    print "failed to execute: $!\n";
}
elsif ($? & 127) {
    printf "child died with signal %d, %s coredump\n",
        ($? & 127), ($? & 128) ? 'with' : 'without';
}
else {
    printf "child exited with value %d\n", $? >> 8;
}
```

For a structured decode, inspect `${^CHILD_ERROR_NATIVE}` - the full native status word - with the `WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG` macros from POSIX:

```
use POSIX qw(WIFEXITED WEXITSTATUS WIFSIGNALED WTERMSIG);
system(@cmd);
my $n = ${^CHILD_ERROR_NATIVE};
if (WIFEXITED($n)) { my $code = WEXITSTATUS($n); ... }
if (WIFSIGNALED($n)) { my $sig = WTERMSIG($n); ... }
```

`SIGINT` and `SIGQUIT` are ignored in the parent while the child runs. If you want your own program to die when the user hits Ctrl-C during `system`, check the return status yourself and re-raise:

```
system(@cmd);
kill 'INT', $$ if ($? & 127) == 2;    # re-raise SIGINT on self
```

Stdio flushing

Before the fork, Perl attempts to flush every filehandle opened for output. This prevents buffered lines from being duplicated by both parent and child. The flush is best-effort; on platforms where it is unreliable, set `$|` (`$AUTOFLUSH`) or call the `autoflush` method from `IO::Handle` on handles you care about.

The die idiom

The canonical check - terse, and it embeds the status word in the message so the caller can still decode it:

```
my @args = ("command", "arg1", "arg2");
system(@args) == 0
    or die "system @args failed: $?";
```

For a module that should throw on any failed `system` (and many other bits of Perl), use the `autodie` pragma.

Examples

Run a command and ignore the result:

```
system("clear");
```

Run a command, die on failure, and keep the status intact for downstream inspection:

```
system("tar", "xf", $archive) == 0  
    or die "untar $archive failed: $?";
```

Detect whether a program is available by running it with `--version` and discarding its output:

```
my $ok = system("which $prog >/dev/null 2>&1") == 0;
```

Run under a specific shell explicitly, bypassing Perl's metacharacter-detection heuristic:

```
system("/bin/bash", "-c", $pipeline);
```

Lie about `argv[0]` using the indirect form - the kernel executes `/usr/bin/vim`, but the program sees its name as `editor`:

```
system { "/usr/bin/vim" } "editor", $file;
```

Edge cases

- **Return of -1:** the child was never created, or `wait` itself failed. Distinct from “child exited with status 255”. Check `$!` for the reason - typically `ENOENT` (program not found) or `EACCES` (not executable).
- **Exit value 255 from a shell:** when the shell form is used and `/bin/sh` cannot find the program, the shell itself exits with 127. Programs that return 255 from Perl typically come through `die` in another Perl program.
- **SIGCHLD handler interaction:** `system` internally forks and waits. A `$SIG{CHLD}` handler can observe or race with this wait; see `perlipc` for the reaping discipline.
- **Shell quirks:** when the shell form is triggered, return codes and error reporting are whatever `/bin/sh` says they are - not always what the underlying program returned. Use the list form to avoid this.
- **Not for capturing output:** `system` writes the child's output to the inherited `STDOUT/STDERR`. Use `qx//` or open a pipe with `open` if you need the output as a string.
- **Windows:** only the `system PROGRAM LIST` syntax reliably avoids the shell. `ppperl` is Linux-only, so this does not apply here; noted for portability.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `exec` - replace the current process instead of forking and waiting; same argument-processing rules as `system`
- `fork` - create a child process without running a new program; `system` is `fork + exec + waitpid`
- `waitpid` - reap a specific child, used when managing multiple children directly
- `qx` - run a command and capture its output as a string; what you want when `system`'s `stdio` targeting is not enough
- `$?` - child error status word set by `system`, `backticks`, `pipe close`, and `wait`
- `$!` - `errno` after a failed `fork` or `exec` (return value -1)
- `${^CHILD_ERROR_NATIVE}` - full native wait status for decoding with POSIX `W*` macros

I/O · Fixed-length data

syswrite

Write bytes to a filehandle with the raw `write(2)` system call, bypassing Perl's buffered I/O.

`syswrite` hands `LENGTH` bytes from `SCALAR` directly to the kernel via `write(2)`, skipping the PerlIO buffering that `print`, `printf`, and `write` go through. The kernel may accept fewer bytes than requested - `syswrite` returns whatever it actually wrote, and well-written code always checks. If `LENGTH` is omitted, the whole `SCALAR` is written. An optional `OFFSET` starts the write at a position other than the beginning of the string.

Synopsis

```
syswrite FILEHANDLE, SCALAR, LENGTH, OFFSET
syswrite FILEHANDLE, SCALAR, LENGTH
syswrite FILEHANDLE, SCALAR
```

What you get back

The number of bytes actually written, or `undef` on error (with `$!` set). **The returned count can be less than `LENGTH`.** This is normal on pipes, sockets, and non-blocking handles; it is not an error. Code that needs every byte delivered must loop:

```
my $off = 0;
my $len = length $buf;
while ($off < $len) {
    my $n = syswrite $fh, $buf, $len - $off, $off;
    defined $n or die "write failed: $!";
    $off += $n;
}
```

A return of `0` is legal and means “no bytes written this call, try again” - not end-of-stream. `undef` is the only error signal.

Global state it touches

- `$!` - set on error (any return of `undef`).
- No interaction with `$,`, `$\`, or `$|`. `syswrite` does not format, does not append record separators, and does not honour the autoflush flag because there is no buffer to flush - bytes go straight to the kernel.
- No interaction with the selected filehandle from `select`. `FILEHANDLE` is mandatory; there is no default.

Examples

Write a string verbatim to STDOUT, no separators added:

```
syswrite STDOUT, "hello\n";          # 6 bytes written
```

Write a fixed-size record and check for a short write:

```
my $rec = pack "A8 N", $name, $id;   # 12 bytes
my $n = syswrite $fh, $rec;
defined $n or die "syswrite: $!";
$n == length $rec or die "short write: $n of ${\length $rec}";
```

Write from the middle of a buffer with OFFSET. Useful inside a drain loop where `$off` tracks how much has been accepted so far:

```
my $buf = "ABCDEFGHIJ";
syswrite $fh, $buf, 4, 3;             # writes "DEFG"
```

Negative OFFSET counts from the end:

```
my $buf = "ABCDEFGHIJ";
syswrite $fh, $buf, 3, -4;            # writes "GHI"
```

Write to a non-blocking socket and handle EAGAIN:

```
use Errno qw(EAGAIN EWOULDBLOCK);

my $n = syswrite $sock, $buf;
if (!defined $n) {
    if ($! == EAGAIN || $! == EWOULDBLOCK) {
        # kernel buffer full - try again after select()
    } else {
        die "syswrite: $!";
    }
}
```

Edge cases

- **Do not mix with buffered I/O on the same handle.** `syswrite` bypasses `PerlIO`, but `print`, `printf`, `write`, `seek`, `tell`, and `eof` all go through the `:perlio` and `:crlf` buffering layers. Interleaving them on one handle corrupts the output order. Pick one mode per handle. If you must mix, use `read` / `sysread` symmetry rules and be deliberate.
- **Short writes are normal.** Pipes, sockets, terminals, and non-blocking handles routinely accept fewer bytes than offered. The drain-loop idiom above is the correct pattern for any handle where short writes matter.
- **Return value 0** means the kernel accepted zero bytes this call. It is not end-of-stream and not an error; retry.
- **Length larger than available data:** if `LENGTH` exceeds `length(SCALAR) - OFFSET`, only the available bytes are written and that count is returned.
- **Empty SCALAR:** if `SCALAR` has length zero, `OFFSET` must be 0. Any other value raises an exception.
- **:utf8 and :encoding(...) layers raise an exception.** `syswrite` operates on bytes, not characters, and refuses any handle carrying a UTF-8 encoding layer. `:encoding(...)` implicitly adds `:utf8`, so it is equally forbidden. Use `binmode` to remove the layer, or write through `print` instead.
- **Characters above U+00FF** on a byte-mode handle also raise an exception - there is no way to serialise a codepoint greater than 0xFF as a single byte. Encode the string explicitly before calling `syswrite`:

```
use Encode qw(encode);
syswrite $fh, encode("UTF-8", $text);
```

- **Broken pipe / closed socket** raises `SIGPIPE`. The default action terminates the process. Install `$SIG{PIPE} = 'IGNORE'` (or a handler) to turn it into a normal `syswrite` error with `$!` set to `EPIPE`.
- **Closed filehandle** returns `undef` with `$!` set to "Bad file descriptor"; under use warnings a `syswrite()` on closed filehandle warning is emitted.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `sysread` - the read counterpart; same buffer-bypassing semantics and same short-read caveat
- `sysopen` - open a handle at the `open(2)` level so `syswrite` can talk to it without any `PerlIO` translation layers in the way
- `sysseek` - reposition the handle using `lseek(2)`, paired with `syswrite` for random-access I/O on raw handles

- `print` - the buffered counterpart you almost always want instead; honours `$,,`, `$\`, and `$|`
- `binmode` - remove a `:utf8` or `:encoding(...)` layer from a handle so `syswrite` can be used on it
- `$!` - inspect after a `undef` return to distinguish EAGAIN from a real failure

I/O

tell

Return the current byte position of a filehandle.

`tell` reports where the next read or write on `FILEHANDLE` will happen, measured in bytes from the start of the file. Pair it with `seek` to save a position now and return to it later. If `FILEHANDLE` is omitted, `tell` reports the position of the handle last read from - including the implicit handle inside `while (<FH>)` loops.

Synopsis

```
tell FILEHANDLE
tell
```

What you get back

A non-negative integer byte offset on success, `-1` on error. The value is suitable as the `POSITION` argument to `seek` with `WHENCE` set to `0` (absolute seek). On pipes, FIFOs, sockets, and sometimes on the standard streams, the underlying OS has no seekable position and `tell` returns `-1`; `$!` is not necessarily set in that case, so treat `-1` as the authoritative failure signal.

```
my $pos = tell $fh;
die "not a seekable handle" if $pos < 0;
# ... read or write ...
seek $fh, $pos, 0;           # back to where we were
```

Global state it touches

- **Last-read filehandle:** the no-argument form `tell` uses the handle most recently read from in the current scope. Every `readline`, `<FH>`, or `<>` updates this. Mixing `tell` with no argument and multiple active handles is a source of surprising results - name the handle explicitly when in doubt.
- `$!` is not a reliable error channel here. Check for `-1` directly rather than testing `$!` after the call.

Bytes, not characters

`tell` always returns a **byte** offset, even when the handle has a character-oriented PerlIO layer such as `:encoding(UTF-8)` or `:utf8`. This is deliberate: translating a character count into a byte offset in a variable-width encoding would require rereading the file from the beginning. The same rule applies to `seek` and `sysseek` - they all speak bytes.

Consequence: on a UTF-8 text file you cannot in general use `tell` to count characters. Use it only to mark a position you intend to feed back into `seek`.

Examples

Remember a position, read ahead, return:

```
open my $fh, "<", "log.txt" or die $!;
my $mark = tell $fh;           # 0
my $first = <$fh>;             # read one line
seek $fh, $mark, 0;           # rewind to the mark
my $again = <$fh>;            # same line again
```

Record the end-of-header offset so later code can rescan the body without reparsing the header:

```
while (my $line = <$fh>) {
    last if $line =~ /^\\r?\\n\\z/;    # blank line ends headers
}
my $body_start = tell $fh;
# ... process body ...
seek $fh, $body_start, 0;          # second pass over the body
```

The no-argument form uses the handle last read from. Inside a `while (<>)` loop that handle is the implicit ARGV:

```
while (<>) {
    print "at ", tell, ": $_";      # offset within the current file
}
```

`tell` on a pipe returns -1 because pipes are not seekable:

```
open my $ph, "-|", "ls" or die $!;
my $pos = tell $ph;               # -1
```

Filehandles inside complex expressions need the same block form as with `print` - `tell $handles[0]` is a syntax error. Use:

```
my @handles = (*STDIN, $fh);
my $pos = tell { $handles[1] };
```

Edge cases

- **Closed filehandle:** returns -1. Under use warnings a tell() on closed filehandle warning is emitted.
- **No-argument form, nothing read yet:** returns -1. The “last-read filehandle” is only meaningful after an actual read.
- **After sysread / syswrite / sysseek:** do not use tell on a handle you have manipulated with the sys* family. Those functions bypass PerlIO buffering; tell reports the buffered position, which will be wrong by the buffer contents. Use `sysseek` with a whence of 1 and an offset of 0 as the sys*-safe position query.
- **Standard streams:** tell STDIN, tell STDOUT, tell STDERR may return -1 or a meaningful offset depending on whether the stream is backed by a regular file (shell redirection) or a terminal, pipe, or socket.
- **No systell:** there is no separate function for buffered-bypassing position. The idiom is:

```
my $pos = sysseek $fh, 0, 1;
```

- **Binary-mode vs text-mode on non-Linux platforms:** pperl targets Linux only, so tell offsets always match raw file bytes. CRLF translation layers that complicate tell on Windows-family perls do not apply here.
- **Append mode (>>):** after open my \$fh, ">>", ..., the initial tell \$fh may report 0 until the first write, because some platforms position the handle at end-of-file only at each write. Don't rely on the initial position; write once, then tell.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `seek` - jump to an absolute byte offset previously returned by tell
- `sysseek` - position query and jump for handles used with `sysread` / `syswrite`, bypassing PerlIO buffering
- `tellldir` - the directory-handle analogue; pairs with `seekdir`
- `eof` - test for end-of-file rather than absolute position
- `open` - produce the filehandle in the first place; mode and PerlIO layers determine whether the handle is seekable

I/O

tellldir

Return the current read position of a directory handle as an opaque token.

tellldir reports where the next `readdir` on DIRHANDLE will pick up. The returned value is a bookmark, not a byte offset: hand it back to `seekdir` to resume iteration at the

same entry, or discard it. The number has no arithmetic meaning - it is not an index into the directory, not a count of entries read, and not comparable across directory handles. Treat it the way you would treat the return value of `tell` on a file handle: opaque to you, meaningful only to the matching `seekdir`.

Synopsis

```
telldir DIRHANDLE
```

What you get back

An integer on success, `undef` on failure (for example, if `DIRHANDLE` is not an open directory handle). The integer is only guaranteed to round-trip through `seekdir` on the **same** open directory handle; closing and reopening the directory invalidates every previously returned token.

```
opendir my $dh, $path or die "opendir $path: $!";
my $pos = telldir $dh;
# ... readdir a few entries ...
seekdir $dh, $pos;           # back to where $pos was taken
```

Global state it touches

Sets `$!` on failure.

Examples

Bookmark a position, then return to it:

```
opendir my $dh, "." or die "opendir: $!";
readdir $dh;           # consume "."
my $mark = telldir $dh;
my @rest = readdir $dh; # drain the rest
seekdir $dh, $mark;
my @again = readdir $dh; # same entries as @rest
closedir $dh;
```

Record a token before each entry so you can rewind one step:

```
opendir my $dh, $path or die "opendir $path: $!";
my $prev;
while (defined(my $entry = readdir $dh)) {
    last if $entry eq "STOP";
    $prev = telldir $dh; # position *after* the current entry
}
seekdir $dh, $prev if defined $prev;
```

Rewinding to the start is better done with `rewinddir` than by remembering the token from before the first `readdir`:

```
opendir my $dh, $path or die "opendir $path: $!";
my @first_pass = readdir $dh;
rewinddir $dh;           # cleaner than seekdir $dh, $start_
    ↪ token
my @second_pass = readdir $dh;
```

Check for failure explicitly when the handle might be closed:

```
my $pos = telldir $dh;
defined $pos or die "telldir failed: $!";
```

Edge cases

- **Token is opaque.** Do not compare, add, subtract, or persist the value. It is valid only for `seekdir` on the same open directory handle within the same process.
- **Not portable across handles.** Two `opendir` calls on the same directory produce independent handles; a token from one is meaningless to the other.
- **Not stable across `closedir` / `opendir`.** Reopening the directory starts a fresh iteration; old tokens no longer refer to anything.
- **Directory compaction.** If the underlying filesystem or another process modifies the directory (adds, removes, or renames entries) between `telldir` and the matching `seekdir`, the resumed position may skip entries, repeat entries, or land on a different entry than the one originally bookmarked. This is a property of the C library `telldir(3)` / `seekdir(3)` pair and is not something Perl papers over. Snapshot the directory with `readdir` into a list if you need stability.
- **Closed or never-opened handle.** Returns `undef` and sets `$!`.
- **Bareword vs. scalar handle.** `telldir DH` and `telldir $dh` both work. The bareword form is a package global; the scalar form holds a reference to an anonymous directory handle created by `opendir my $dh, ....`

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `seekdir` - the only consumer of a `telldir` token; restores `DIRHANDLE` to the bookmarked position
- `readdir` - the iterator whose position `telldir` reports; advances the cursor that `telldir` observes
- `rewinddir` - cheaper than `seekdir` to a start-of-stream token when you just want to iterate the directory again
- `opendir` - creates the `DIRHANDLE` that `telldir` queries; every `telldir` token is tied to one open call
- `closedir` - invalidates every `telldir` token for the handle

- `tell` - the file-handle analogue; same opaque-token contract, different underlying system call

Classes and OO

tie

Bind a variable to a class so that every access to the variable dispatches through that class's methods.

`tie` enchants `VARIABLE` so that subsequent reads, writes, and other operations on it are forwarded to `CLASSNAME`'s tie-interface methods (`FETCH`, `STORE`, `FIRSTKEY`, `PRINT`, ...). `LIST` is passed to the class's constructor (`TIESCALAR`, `TIEARRAY`, `TIEHASH`, or `TIEHANDLE`, depending on the sigil of `VARIABLE`). The constructor returns the underlying tied object, and that object - **not** `VARIABLE` - is the return value of `tie`. Keep it if you want to call extra methods on the implementation directly.

The method contract each class must satisfy is documented at length in `perltie`; this page documents `tie` the built-in.

Synopsis

```
tie my $scalar, 'ScalarClass', @args;
tie my @array, 'ArrayClass', @args;
tie my %hash, 'HashClass', @args;
tie *FH, 'HandleClass', @args;

my $obj = tie my %h, 'HashClass', @args; # keep the tied object
```

What you get back

The object returned by the class's `TIE*` constructor - an ordinary blessed reference, same as any other object. Call methods on it directly when the class exposes operations that don't map onto plain variable access:

```
my $db = tie my %cache, 'My::Cache', path => '/var/cache/app.db';
$db->flush;           # method on the tied object
%cache{key} = 'value'; # goes through STORE
```

`tie` does not return the variable itself. Code that writes `my $x = tie ...` and then treats `$x` as the scalar is a common beginner mistake; the scalar is `$x` only if you also tied `$x`, and the assignment target is the tied object, not the variable.

To recover the tied object from a variable later, use `tied`. To dissolve the binding, use `untie`.

Which constructor is called

The sigil of `VARIABLE` picks the constructor. `LIST` is forwarded verbatim:

VARIABLE form	Constructor called	Interface reference
<code>\$scalar</code>	<code>CLASSNAME->TIESCALAR(LIST)</code>	<code>Tie::Scalar</code>
<code>@array</code>	<code>CLASSNAME->TIEARRAY(LIST)</code>	<code>Tie::Array</code>
<code>%hash</code>	<code>CLASSNAME->TIEHASH(LIST)</code>	<code>Tie::Hash</code>
<code>*FH (glob)</code>	<code>CLASSNAME->TIEHANDLE(LIST)</code>	<code>Tie::Handle</code>

The constructor's arguments are conventionally those you'd pass to something like `dbm_open(3)` - a filename, a mode, a permission mask - but anything the class expects is fine.

Methods each class must provide

`tie` itself only invokes `TIE*`. Every other operation on the variable dispatches to a method named after that operation. Not all methods are mandatory - missing methods make the corresponding operations fail with a runtime error. The full menus are:

Tied scalar (TIESCALAR):

```
TIESCALAR classname, LIST
FETCH      this
STORE      this, value
DESTROY    this
UNTIE      this
```

Tied array (TIEARRAY):

```
TIEARRAY  classname, LIST
FETCH      this, index
STORE      this, index, value
FETCHSIZE  this
STORESIZE  this, count
CLEAR      this
PUSH       this, LIST
POP        this
SHIFT      this
UNSHIFT    this, LIST
SPLICE     this, offset, length, LIST
EXTEND     this, count
DELETE     this, index
EXISTS     this, index
DESTROY    this
UNTIE      this
```

Tied hash (TIEHASH):

```
TIEHASH    classname, LIST
FETCH      this, key
STORE      this, key, value
DELETE     this, key
CLEAR      this
```

(continues on next page)

(continued from previous page)

```

EXISTS    this, key
FIRSTKEY  this
NEXTKEY   this, lastkey
SCALAR    this
DESTROY   this
UNTIE     this

```

Tied filehandle (TIEHANDLE):

```

TIEHANDLE this, LIST
READ       this, scalar, length, offset
READLINE   this
GETC       this
WRITE      this, scalar, length, offset
PRINT      this, LIST
PRINTF     this, format, LIST
BINMODE    this
EOF        this
FILENO     this
SEEK       this, position, whence
TELL       this
OPEN       this, mode, LIST
CLOSE      this
DESTROY    this
UNTIE      this

```

See `perltie` for the full specification (method signatures, when each is called, defaults inherited from the `Tie::*` base classes).

Examples

Bind a hash to an on-disk DBM file and iterate with `each`. `keys` and `values` would materialise the whole set at once - for large DBM files that's expensive:

```

use NDBM_File;
tie(my %HIST, 'NDBM_File', '/usr/lib/news/history', 1, 0);
while (my ($key, $val) = each %HIST) {
    print $key, ' = ', unpack('L', $val), "\n";
}

```

Keep the constructor's return value so you can talk to the tied object directly:

```

my $impl = tie my %cache, 'My::Cache', ttl => 60;
$cache{session} = $data;      # STORE
$impl->expire_now;            # direct method call

```

Tie a scalar to a read-only class that logs every access:

```

package Tie::Logged {
    sub TIESCALAR { my ($class, $v) = @_; bless \$v, $class }

```

(continues on next page)

(continued from previous page)

```

sub FETCH      { my $self = shift; warn "read\n"; $$self }
sub STORE      { my ($s, $v) = @_; warn "write\n"; $$s = $v }
}

tie my $x, 'Tie::Logged', 42;
print $x;      # warns "read",  prints 42
$x = 99;       # warns "write"

```

Tie a filehandle to a class that captures output into a string:

```

package Tie::StringHandle {
    sub TIEHANDLE { my $c = shift; bless { buf => '' }, $c }
    sub PRINT      { my $s = shift; $s->{buf} .= join '', @_; 1 }
    sub buf        { $_[0]->{buf} }
}

tie *OUT, 'Tie::StringHandle';
print OUT "hello\n";
print OUT "world\n";
my $obj = tied *OUT;
print $obj->buf;    # "hello\nworld\n"

```

Edge cases

- **Return value is the object, not the variable.** `my $r = tie my %h, ...` binds `$r` to the tied object. Assigning to `$r` assigns to a scalar; it does not write through to `%h`.
- **Module must be loaded first.** Unlike `dbmopen`, `tie` does **not** use or require `CLASSNAME` for you. Load it explicitly:

```

use My::TiedClass;
tie my %h, 'My::TiedClass';

```

Forgetting this is the typical cause of Can't locate object method "TIEHASH" via package

- **Missing methods are fatal at the point of use**, not at tie time. A tied hash whose class defines `FETCH` but not `STORE` ties successfully, then raises Can't locate object method "STORE" ... on the first assignment.
- **Sigil mismatch is a tie-time error.** `tie` picks the constructor from the variable's sigil, so a class that only defines `TIESCALAR` cannot be tie-d to a hash - the lookup of `TIEHASH` fails at the tie call itself.
- **tieing a tied variable** implicitly `unties` the previous binding first, triggering `UNTIE` and eventually `DESTROY` on the old object (once the last reference to it drops).
- **local on a tied variable** saves and restores the *binding*, not just the value. At scope exit the variable is re-tied to its original object.
- **Holding an extra reference to the tied object** keeps it alive past `untie`. Under use warnings this produces a `untie attempted while N inner references`

still exist warning. Drop the reference (e.g. `undef $obj`) before calling `untie` if you want DESTROY to fire promptly.

- **tie does not stash the object inside the variable in any user-visible way** - use `tied` to retrieve it; there's no dereference syntax that exposes it.
- **Performance.** Every operation on a tied variable dispatches through a method call. In tight loops this is visibly slower than a plain variable; for bulk work, read into a plain array or hash first, operate, then write back.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `tied` - recover the object a variable is currently tied to, or `undef` if it isn't tied
- `untie` - dissolve the binding and let the tied object be destroyed
- `bless` - the low-level primitive TIE* constructors use to turn a reference into an object
- `dbmopen` - older shortcut for the specific case of tying a hash to a DBM file; loads the DBM class for you, which `tie` does not
- `ref` - inspect the class of the object returned by `tie`

Classes and OO

tied

Return the object backing a tied variable.

`tied` hands you the same reference that `tie` originally returned when VARIABLE was bound to a class. Through that reference you can call methods directly on the tie implementation - bypassing the usual `FETCH` / `STORE` dispatch - which is how tie classes expose extra operations that have no natural scalar / array / hash syntax. If VARIABLE is not tied, `tied` returns `undef`.

Synopsis

```
tied VARIABLE
```

What you get back

A reference to the underlying tie object when VARIABLE is tied, and `undef` when it is not. The reference is the exact value originally returned by the matching `tie` call - same class, same blessed identity - so `ref(tied $v)` names the tie class and method calls on it dispatch normally:

```
my $obj = tied %hash;
$obj->flush if defined $obj;
```

Use `defined` (or a boolean test, when the tie class never blesses into a false-overloading package) to distinguish a tied variable from an untied one before calling methods on the result.

The “tie handle” trick

`tied` is the only supported way to talk to the tie implementation directly. Any method on the tie class that is not one of the standard hooks (`FETCH`, `STORE`, `EXISTS`, `DELETE`, `CLEAR`, `FIRSTKEY`, `NEXTKEY`, and their array / scalar / handle equivalents) is reachable only through the object tied returns:

```
tie my %cache, 'My::LRU', size => 1024;
tied(%cache)->stats;           # class-specific method, no hash_
↪syntax
tied(%cache)->evict_older_than(3600);
```

This is also the pattern for introspection - asking a tied variable “who are you tied to?” without assuming knowledge of the binding site:

```
my $class = ref tied @array;    # e.g. "Tie::File"
```

Examples

Basic test for tiedness:

```
tie my %h, 'My::Tied::Hash';
print defined(tied %h) ? "tied\n" : "not tied\n";    # tied
```

Calling a class-specific method through the tie object:

```
tie my @log, 'Tie::File', 'app.log' or die $!;
tied(@log)->flock(2);           # LOCK_EX; Tie::File API
# ... mutate @log ...
tied(@log)->flock(8);           # LOCK_UN
```

A plain scalar, array, hash, or filehandle that was never tied:

```
my $x;
print defined(tied $x) ? "yes" : "no", "\n";         # no
```

`tied` does **not** untie - the variable stays tied after the call, and successive calls return the same reference:

```
tie my $t, 'My::Counter';
my $a = tied $t;
my $b = tied $t;
print $a == $b ? "same\n" : "different\n";          # same
```

Round-tripping through `ref` to learn the tie class:

```
tie my %db, 'DB_File', 'data.db', O_RDWR | O_CREAT, 0644;
print ref tied %db, "\n";        # DB_File
```

Edge cases

- **Not tied:** returns `undef`. No warning, no exception. This is the idiomatic way to check whether a variable is tied at all - there is no separate `is_tied` built-in.
- **Filehandles:** `tied *FH` returns the object for a tied filehandle (bound with `tie *FH, ...`). A glob that was never tied yields `undef` just like any other untied variable.
- **tied does not untie.** It is a pure accessor. To release the binding, call `untie`. Holding the reference returned by `tied` past the point where the variable is untied keeps the tie object alive, which is deliberate - that is how `untie` warnings about “`untie` attempted while N inner references still exist” are triggered.
- **Tied element of an aggregate:** when an individual array or hash element is itself tied (separate from a tie on the container), `tied $array[0]` or `tied $hash{key}` returns the per-element object, not the container’s. A container tie and an element tie are independent bindings.
- **Copied values are not tied.** `my $copy = $tied_scalar;` stringifies through `FETCH`; `$copy` is a plain scalar and `tied $copy` returns `undef`. Tie is a property of the variable, not of the value.
- **References to tied variables preserve the tie.** `\%hash` where `%hash` is tied still refers to the tied container, so `tied %{ $ref }` returns the same tie object.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `tie` - bind a variable to a class; establishes the tie relationship that `tied` queries
- `untie` - release the tie; after calling it, `tied` on the same variable returns `undef`
- `ref` - get the class name of the object returned by `tied` without a separate method call
- `bless` - `tied` returns a blessed reference when the tie class blessed the object it returned from its `TIE*` constructor
- `defined` - the standard test for “is this variable tied?” is `defined tied $var`

Time

time

Return the current wall-clock time as an integer number of seconds since the system epoch.

`time` is the starting point for every other date and time operation in Perl. It reads the system clock and returns a single integer - the number of non-leap seconds since the epoch, which on every supported platform is 00:00:00 UTC, 1 January 1970 (Unix time). The result is suitable for feeding directly to `gmtime` or `localtime` to break it down into a calendar date, or for subtracting two readings to measure elapsed time.

Synopsis

```
time
```

What you get back

A plain integer: seconds since 1970-01-01 00:00:00 UTC, truncated (not rounded) toward zero. No sub-second precision - two calls within the same second return the same value. For millisecond or microsecond granularity, use `Time::HiRes::time`.

`time` takes no arguments; the bare word is the entire call. Writing `time()` is also fine and sometimes clearer next to other function calls.

Examples

Raw reading of the clock:

```
my $now = time;
print "$now\n";
```

e.g. 1745424000

Human-readable local date and time - `time` pairs with `localtime` in scalar context for a formatted string:

```
print scalar localtime, "\n";
```

Wed Apr 23 12:00:00 2026

The same reading broken into fields for calendar arithmetic:

```
my ($sec, $min, $hour, $mday, $mon, $year) = localtime time;
$year += 1900;
$mon += 1;
printf "%04d-%02d-%02d %02d:%02d:%02d\n",
    $year, $mon, $mday, $hour, $min, $sec;
```

Elapsed-time measurement around a block of work - the idiomatic whole-second stopwatch:

```
my $t0 = time;
do_work();
my $elapsed = time - $t0;
print "took ${elapsed}s\n";
```

UTC rather than local time - feed the same integer to `gmtime`:

```
my @utc = gmtime time;
print scalar gmtime, "\n";
```

same fields, UTC-based
Wed Apr 23 10:00:00 2026

Edge cases

- **No sub-second resolution.** Back-to-back calls in the same second return the same integer, so `time - $t0` for fast work prints 0. Use `Time::HiRes::time` for a floating-point reading with microsecond precision:


```
use Time::HiRes qw(time);
my $t0 = time;                # now a float
do_work();
printf "%.6f s\n", time - $t0;
```

- **Leap seconds.** The returned count excludes leap seconds; it is POSIX “seconds since the epoch” as reported by the kernel. Code that subtracts two time readings measures wall-clock elapsed seconds, not SI seconds, across a leap-second boundary.
- **Integer width.** The value is a signed integer wide enough for current and far-future timestamps on all supported platforms (64-bit). The 2038 problem does not apply here.
- **Timezone.** time itself is timezone-agnostic - the integer names a single instant in UTC. Timezones only enter the picture when you convert via [localtime](#); [gmtime](#) stays in UTC.
- **Monotonicity.** time follows the system clock. If the clock is stepped backward (NTP adjustment, manual change), a later call may return a smaller value than an earlier one. For elapsed measurements that must not go backward, use a monotonic clock via `Time::HiRes::clock_gettime(CLOCK_MONOTONIC)`.
- **In arithmetic.** `time + 3600` is one hour from now; `time - 86400` is 24 hours ago. Feed the result to [localtime](#) or [gmtime](#) to render it.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- [gmtime](#) - break a time value into UTC calendar fields (or a formatted string in scalar context)
- [localtime](#) - same, but in the local timezone; the most common partner for time
- [sleep](#) - pause for a whole number of seconds; pairs naturally with time for coarse scheduling
- [alarm](#) - schedule a SIGALRM a given number of seconds in the future
- `Time::HiRes::time` - floating-point seconds with microsecond resolution when whole-second granularity is not enough

[Processes](#) · [Time](#)

times

Report CPU time consumed by this process and its terminated children.

times asks the kernel how many seconds of user-mode and kernel-mode CPU have been charged to the current process, and - separately - to children that have already been reaped. The return is four floating-point seconds: user, system, child-user, child-system.

It answers “how much CPU have I burned?”, not “how much wall-clock time has passed?”
- for that, see `time`.

Synopsis

```
my ($user, $system, $cuser, $csystem) = times;  
my $user_only                        = times;    # scalar context
```

What you get back

In list context, four values:

- `$user` - CPU time spent in user-mode code of this process.
- `$system` - CPU time spent in the kernel on behalf of this process.
- `$cuser` - user-mode CPU time of **terminated** child processes.
- `$csystem` - kernel-mode CPU time of **terminated** child processes.

All four are seconds as double-precision floats; resolution is microsecond on Linux (sourced from `getrusage`).

In scalar context, `times` returns just `$user`.

`times` takes no arguments and cannot fail in a way visible to the caller.

Children’s times - the “terminated” rule

`$cuser` and `$csystem` aggregate CPU only from children that have been **reaped** - i.e. a `wait` or `waitpid` has successfully collected their exit status, or they were autoreaped because `$SIG{CHLD}` was set to `'IGNORE'`. A child that is still running, or has exited but not yet been reaped (a zombie), contributes nothing.

This matters when measuring a pipeline that spawns workers:

```
my $pid = fork;  
if ($pid == 0) { do_work(); exit }  
# (busy child runs here)  
my ($u0, $s0, $cu0, $cs0) = times;    # $cu0, $cs0 still 0  
waitpid($pid, 0);  
my ($u1, $s1, $cu1, $cs1) = times;    # now $cu1, $cs1 reflect  
↳child
```

Until the `waitpid` returns, the child’s CPU does not appear in the parent’s times.

Examples

Measure CPU cost of a block of work, excluding wall-clock sleeps or I/O waits:

```
my ($u0, $s0) = times;  
compute_heavy();  
my ($u1, $s1) = times;
```

(continues on next page)

(continued from previous page)

```
printf "compute used %.3f s user + %.3f s system\n",
    $u1 - $u0, $s1 - $s0;
```

Compare CPU time to wall-clock time - a low ratio means the process was mostly waiting rather than computing:

```
use Time::HiRes qw(time);
my $t0 = time;
my ($u0, $s0) = times;
do_mixed_io_and_compute();
my $wall = time - $t0;
my ($u1, $s1) = times;
printf "cpu/wall = %.2f\n", ($u1 - $u0 + $s1 - $s0) / $wall;
```

Attribute a parallel workload to its children. Reap every child before reading times:

```
my @pids = map { my $p = fork; $p == 0 ? (worker($_), exit) : $p }
    1..4;
waitpid($_, 0) for @pids;
my (undef, undef, $cu, $cs) = times;
printf "workers used %.2f s user + %.2f s system\n", $cu, $cs;
```

Scalar context when only the user time matters - e.g. a quick self-profiling print:

```
printf STDERR "[%.2f] reached checkpoint\n", scalar times;
```

Edge cases

- **No arguments, no prototype to worry about.** `times` is a zero-argument built-in; `times()` and `times` parse identically. It is **not** the transliteration operator - that is `tr///` (alias `y///`), which is a separate token.
- **Scalar vs list context is load-bearing.** Writing `my $t = times;` binds `$user` to `$t`, discarding the other three values silently. Use `my ($u,$s,$cu,$cs) = times;` when you want them all.
- **Unreaped children don't count.** See the “terminated” rule above. Reading `times` before `wait` gives `$cuser == $csystem == 0` even when the child is visibly burning CPU in top.
- **Threads do count.** On Linux, kernel-threaded work in the current process shows up in `$user`/`$system` just like single-threaded work. Threads in a separate process don't, until that process exits and is reaped.
- **Clock is monotonic-within-process, not wall-clock.** `times` never goes backwards across a single process's lifetime, but successive `times` calls are not directly comparable between parent and child - each `getrusage` accounting is local.
- **Resolution.** Linux reports microsecond-resolution ticks via `getrusage`, so short measurements (sub-millisecond) are noisy but not zero. For high-precision wall-clock timing, use `Time::HiRes::time`, not `times`.

- **Overflow.** Values are floats; they can in principle lose sub-microsecond precision after very long runs (years of CPU), but this is not a concern for normal programs.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `time` - wall-clock seconds since the epoch; use it when you want elapsed real time, not CPU time
- `Time::HiRes::time` - high-resolution wall-clock time, the usual partner for CPU accounting
- `POSIX::times` - POSIX wrapper returning an additional real-time “clock ticks since boot” value; use it when you need the elapsed field the built-in omits
- `wait` - reap a child so its CPU starts counting in `$cuser / $csystem`
- `waitpid` - reap a specific child for the same reason, without blocking on unrelated ones
- `fork` - the operation whose cost shows up in the child columns once the child is reaped

SCALARs and strings

`tr///`

Character-by-character substitution. `tr` scans a string and replaces every occurrence of a character from SEARCHLIST with the positionally corresponding character from REPLACEMENTLIST, returning the count of characters it touched.

It is not a regex. SEARCHLIST is a set of literal characters (with range shorthand like `a-z`), not a pattern. No metacharacters, no `\d`, no variable interpolation. The translation table is built once, at compile time.

Synopsis

```
tr/SEARCHLIST/REPLACEMENTLIST/cdsr
y/SEARCHLIST/REPLACEMENTLIST/cdsr
$str =~ tr/abc/xyz/;
$count = ($str =~ tr/aeiou//);
```

`y///` is a sed-compatibility synonym for `tr///` - identical behaviour, identical modifiers. See `y///`.

What you get back

Without the `/r` modifier, `tr` mutates the target string in place and returns the **number of characters matched** - or, under `/d`, matched-or-deleted; under `/s`, matched-before-squashing. The return value is an integer and is commonly used to *count* characters without changing anything (see the `tr/*/*/` idiom below).

With `/r`, `tr` leaves the original string alone, builds a transliterated copy, and returns **that copy**. The returned string is always a plain string - even if the target was a tied variable or a blessed object.

Target without `=~`: `tr` operates on `$_`. Target with `=~`: the left-hand side must be an lvalue (scalar, array element, hash element) unless `/r` is used, in which case any expression works.

Modifiers

The four trailing flags change what `tr` does with each character it sees. They can be combined:

- **c** - complement SEARCHLIST. The set of characters acted on becomes *every character NOT in* SEARCHLIST. Under `/c`, SEARCHLIST is sorted by codepoint after complementing, and any REPLACEMENTLIST is applied to that sorted set - so the order of characters you wrote in SEARCHLIST no longer matters.
- **d** - delete. Characters matched by SEARCHLIST that have no position in REPLACEMENTLIST are removed from the target entirely rather than replaced with the final replacement character.
- **s** - squash. A run of characters that all translate to the same output character collapses to a single instance of that output.
- **r** - return modified copy. Do not touch the target. Return the transliterated string instead of a count.

Replacement list length rules

The relationship between SEARCHLIST and REPLACEMENTLIST changes with the modifiers:

- If REPLACEMENTLIST is **shorter** than SEARCHLIST and `/d` is **not** in effect, the final replacement character is *replicated* to fill the gap. `tr/abcd/AB/` is the same as `tr/abcd/ABBB/`.
- If REPLACEMENTLIST is shorter than SEARCHLIST **and** `/d` is in effect, surplus SEARCHLIST characters are deleted. `tr/abcd/AB/d` does `tr/ab/AB/` plus `s/[cd]/g`.
- If REPLACEMENTLIST is **empty** and `/d` is not in effect, REPLACEMENTLIST becomes a copy of SEARCHLIST. This is the count-only form: `tr/abc//` counts a, b, c without changing anything.
- If REPLACEMENTLIST is empty **and** `/d` is in effect, every character in SEARCHLIST is deleted. `tr/abc//d` removes all a, b, c from the target.

Character ranges

A hyphen between two characters specifies an inclusive codepoint range: `tr/A-J/0-9/` is `tr/ABCDEFGHJIJ/0123456789/`. A hyphen at the start, at the end, or preceded by a backslash is taken literally - `tr/-ab-/xyz/` and `tr/\-abc/wxyz/` treat the hyphen as the literal character `-`.

Only use ASCII-alphabet (same case) or ASCII-digit ranges. a-z, A-Z, 0-9, and clean subsets like h-k or B-E are portable and mean exactly what they read as. Mixed-case ranges, \x-ranges, or ranges straddling the alphabet/digit boundary look obvious but produce different results on non-ASCII platforms - spell them out.

Portable Unicode ranges use \N{U+...} endpoints:

```
$str =~ tr/\N{U+20}-\N{U+7E}//d;    # delete all ASCII printables
```

Delimiters

The three delimiters around SEARCHLIST and REPLACEMENTLIST can be any matching pair of printable characters, not just /. Bracketing pairs ({}, (), [], <>) require two independent pairs, one for each list:

```
tr(aeiouy)(yuoiea);  
tr[+ \ - * /]"ABCD";
```

When the delimiter is a single quote (tr '...' '...'), the lists are treated almost literally - only \\ pairs are collapsed. Hyphens inside single-quoted tr are **not** range specifiers; they are literal hyphens.

Global state it touches

- \$_ - default target when neither =~ nor !~ is used. tr/// alone reads and (unless /r is used) mutates \$_.
- No other special variables. tr does not set \$&, \$1, etc. - those belong to regex matching, and tr is not regex.

Examples

ASCII lowercase to uppercase:

```
my $name = "Perl";  
$name =~ tr/a-z/A-Z/;    # $name is now "PERL"
```

Count without changing - the tr/X/X/ idiom. The replacement is the same character, so the string is unchanged; the return value is the count:

```
my $stars = "***hello**";  
my $cnt   = ($stars =~ tr/*/*/);    # $cnt == 4
```

Empty REPLACEMENTLIST without /d counts, too:

```
my $digits = ($text =~ tr/0-9//);    # count ASCII digits in $text
```

Delete characters with /d:

```
my $s = "(555) 123-4567";  
$s =~ tr/0-9//cd;    # keep only digits: "5551234567"
```

Squash runs with /s:

```
my $line = "hello    world";
$line =~ tr/ //s;           # "hello world"
```

Non-destructive /r - build a copy, leave the original:

```
my $host  = "example.com";
my $HOST  = $host =~ tr/a-z/A-Z/r;    # $host unchanged, $HOST_
→uppercased
```

Chain with other /r operators:

```
my $tag = $host =~ tr/a-z/A-Z/r
           =~ s/\./_/gr;    # "EXAMPLE_COM"
```

map with /r - turn a list of strings into a transformed list:

```
my @upper = map tr/a-z/A-Z/r, @names;
```

Whole-word compacting - collapse non-alphabetic runs to a single space:

```
$sentence =~ tr/a-zA-Z/ /cs;    # "foo, bar; BAZ!" -> "foo bar BAZ "
```

Strip the high bit off every byte:

```
$bytes =~ tr[\200-\377][\000-\177];
```

Edge cases

- **Not a regex.** `tr/\d/X/` matches the literal characters `\` and `d`, not “any digit”. Character classes, anchors, quantifiers, and backreferences are all silently literal. If you want regex, use `s///`.
- **No variable interpolation.** `$var` and `@var` inside the lists are literal `$ / @` plus the following name. The table is compiled once, when the program is parsed. To build the lists dynamically, wrap in `eval`:

```
eval "tr/$from/$to/";
die "$@" if $@;
```

- **Repeated characters in SEARCHLIST** - only the first mapping counts. `tr/AAA/XYZ/` translates every A to X; the Y and Z positions are unreachable.
- **No rescanning.** Transliterated output is not re-fed into the table, even if the replacement character appears in SEARCHLIST too. `tr/ox/xo/` turns “`oxxo`” into “`xoox`”, not some fixed point.
- **!~ with tr** - returns 0 if any character was changed, 1 otherwise. Useful only for quick “nothing happened” checks:

```
$foo !~ tr/A/a/    # true iff no 'A' was in $foo
```

- **Lvalue requirement.** Without /r, the `=~` target must be assignable. “`constant`” `=~ tr/a/b/` is a compile-time error.

- **/r** always returns a **plain string**. If the target is a tied scalar or a blessed string-overloaded object, /r strips the magic and gives you a fresh plain string.
- **tr** is not the shell **tr(1)**. Similar syntax, overlapping semantics, but case-folding beyond ASCII belongs to `lc` / `uc` / `lcfirst` / `ucfirst` or to `s///` with `\U` / `\L` - not to `tr`.
- **/c** with a multi-character **REPLACEMENTLIST** is non-portable across character sets. Under /c the search set is sorted by codepoint, and that sort order differs between ASCII and EBCDIC. Stick to single-character REPLACEMENTLIST when combining with /c, or use /c only with /d (where the replacement doesn't matter).

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `y///` - sed-compatibility synonym for `tr///`; identical behaviour, pick whichever reads better in context
- `s///` - full regex substitution; reach for it when you need patterns, captures, or variable interpolation in the search side
- `lc` - Unicode-aware lowercase; use instead of `tr/A-Z/a-z/` when the input may contain non-ASCII letters
- `uc` - Unicode-aware uppercase; complement to `lc`
- `tr` in `perlop` - the upstream reference for the quote-like operator form, delimiter rules, and range portability
- `$_` - default target when `tr` is used without `=~` or `!~`

I/O

truncate

Shorten (or extend) a file to an exact byte length.

`truncate` adjusts the size of the file opened on `FILEHANDLE`, or the file named by `EXPR`, so it is exactly `LENGTH` bytes long. Bytes past `LENGTH` are discarded. The file's current read/write position is **not** changed - a handle sitting past the new end of file stays there, which is almost never what you want, so plan a `seek` before the next write.

Synopsis

```
truncate FILEHANDLE, LENGTH
truncate EXPR, LENGTH
```


What you get back

1 on success, `undef` on failure with `$!` set to the underlying `errno`. A missing platform implementation raises an exception rather than returning `undef`; that is an install-time property, not something to branch on at runtime.

```
truncate $fh, 0
or die "truncate failed: $!";
```

Filehandle vs. filename form

Two forms, chosen by what you already have in hand:

- **truncate FILEHANDLE, LENGTH** - the handle must be open for writing (or read/write). Use this when the caller already owns an open write handle; it avoids a fresh open/close round-trip and works even after the file has been unlinked.
- **truncate EXPR, LENGTH** - EXPR is stringified as a pathname. The file is opened, truncated, and closed internally. Use this when you only have a path and do not need the handle afterwards.

Both forms require write permission on the target. On the filehandle form, that is a property of the open mode; on the pathname form, of the filesystem permissions.

LENGTH semantics

- LENGTH is in **bytes**, not characters. Encoding layers on the filehandle are irrelevant - the length argument goes straight to the OS.
- LENGTH less than the current file size discards the tail.
- LENGTH equal to the current size is a no-op that still succeeds.
- LENGTH greater than the current size extends the file with a hole of zero bytes on filesystems that support sparse files, or with literal zero bytes otherwise. Upstream `perlfunc` calls this case undefined; in practice it behaves like POSIX `ftruncate(2)` on Linux, which is what `pperl` targets.

Global state it touches

- `$!` is set on failure to the underlying `errno`.

Examples

Empty a logfile without closing the handle the rest of the program is still writing to:

```
open my $log, "+<", "app.log" or die $!;
truncate $log, 0 or die "truncate: $!";
seek $log, 0, 0;                # rewind; see "Position is not reset"
"
```

Truncate a file by name, no handle involved:

```
truncate "scratch.dat", 0
or die "truncate scratch.dat: $!";
```

Rotate a pre-allocated buffer file down to the used portion:

```
my $used = tell $fh;           # bytes actually written
truncate $fh, $used
or die "truncate: $!";
```

Extend a file to a fixed size (sparse hole on Linux):

```
open my $fh, ">", "image.raw" or die $!;
truncate $fh, 1024 * 1024 * 1024 # 1 GiB sparse file
or die "truncate: $!";
```

Edge cases

- **Position is not reset.** After `truncate $fh, 0`, the handle still remembers whatever offset it had. A following `print $fh ...` without an intervening `seek` may write past the new end of file, leaving a zero-filled hole.
- **Read-only handle:** truncating a handle opened with `<` fails with `EBADF / EINVAL` depending on the platform. Open with `+` (read/write) or `>>` plus an appropriate mode if you need to truncate through the same handle you read from.
- **Negative LENGTH:** the kernel rejects it and `truncate` returns `undef` with `$!` set to `EINVAL`.
- **Directory as EXPR:** fails with `EISDIR`. `truncate` is strictly a file operation.
- **File unlinked but still open:** the filehandle form works fine - the inode is still there as long as any handle holds it open.
- **LENGTH larger than current size:** on Linux this extends with a sparse hole; reading the hole yields `"\0"` bytes. Upstream `perlfunc` flags this as undefined, so portable code should not rely on it.
- **Platform without ftruncate/truncate:** raises an exception (The `truncate` function is unimplemented). Not a concern on pperl's supported platform (Linux), but matters for portable code.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `open` - produce a writable filehandle suitable for the filehandle form of `truncate`
- `seek` - reposition the handle after truncating, before writing again
- `tell` - current offset, useful as the `LENGTH` argument when trimming to “what I have written so far”

- `sysopen` - lower-level open when you need explicit flags (`O_TRUNC` truncates at open time; `truncate` is the after-the-fact equivalent)
- `unlink` - reach for this instead when you want the file gone, not merely emptied
- `$!` - `errno` on failure

Control flow

try

Run a block and divert any exception it throws into a `catch` block, with an optional `finally` block that always runs on the way out.

`try/catch` is Perl's structured exception-handling syntax. The `try` block executes; if any statement in it throws (via `die`, an uncaught exception from a callee, or a runtime error), control transfers to the `catch` block with the exception value bound to the parenthesised lexical. If nothing throws, the `catch` block is skipped. Unlike `eval`, a `return` inside `try` returns from the enclosing sub, not from the block - `try` is control flow, not a call frame.

The syntax must be enabled with a feature guard:

```
use feature 'try';
```

Synopsis

```
try BLOCK catch (VAR) BLOCK
try BLOCK catch (VAR) BLOCK finally BLOCK
```

What you get back

`try` is a statement, not an expression, but it yields the last evaluated value of whichever branch ran - the `try` block on success, or the `catch` block on failure. Used as the final statement of a sub or a `do` block, that value becomes the result:

```
my $value = do {
    try {
        fetch_thing(@args);
    }
    catch ($e) {
        warn "fetch failed: $e";
        $DEFAULT;
    }
};
```

Do **not** write `return` inside a `try` that is being used this way

- `return` exits the enclosing sub, bypassing the assignment.

A `finally` block's final expression is discarded; it cannot contribute to the value.

Global state it touches

- `$@` is **not** the mechanism here. `try/catch` binds the exception to the lexical named in `catch` (`VAR`), not to `$@`. Code inside `catch` sees whatever `$@` happened to hold before the `try`; rely on the lexical.
- `$_`, `$!`, and other dynamic globals are untouched by the construct itself - only by whatever code runs inside the blocks.

The catch variable

`catch` must be followed immediately by a parenthesised variable declaration. `my` is implicit - write `catch ($e)`, not `catch (my $e)`. The variable is lexical to the `catch` block and holds the thrown value exactly as `die` delivered it: a string, a blessed reference, or any scalar:

```
try {
    risky();
}
catch ($e) {
    if (ref $e && $e->isa('My::Exception')) {
        $e->rethrow if $e->is_fatal;
        warn $e->message;
    }
    else {
        warn "plain die: $e";
    }
}
```

There is no multi-catch dispatch on exception class - write the classification yourself inside the single `catch` block.

finally

An optional third block runs on every exit path from the `try/catch` construct: normal fall-through, exception caught by `catch`, exception rethrown from `catch`, or a non-local transfer (`return`, `last`, `next`, `redo`, `goto`) out of either block.

```
try {
    open_resource();
    use_resource();
}
catch ($e) {
    warn "failed: $e";
}
finally {
    close_resource();
}
```

`finally` is the right place for release code that must run no matter how the protected section exits - file handles, locks, restoring global state.

A `finally` block has restrictions the other two blocks do not:

- No `return`, no `goto`, no `last` / `next` / `redo`. Attempting any of these is a compile-time or runtime error.
- The final expression value is ignored - `finally` cannot contribute to the construct's yielded value.

`finally` remains **experimental** in Perl 5.42 and emits a warning in the `experimental::try` category when used. `try` and `catch` themselves were de-experimentalised in 5.40 and no longer warn.

Contrast with `eval { ... } / $@`

The older pattern is still supported, but `try/catch` fixes three long-standing pitfalls:

Pitfall	<code>eval / \$@</code>	<code>try / catch</code>
<code>\$@</code> clobbering between <code>eval</code> exit and the <code>if (\$@)</code> check	real hazard, needs local <code>\$@</code> dance	exception bound to a fresh lexical
<code>return</code> inside the protected block	returns from the <code>eval</code> , not the sub	returns from the enclosing sub
Distinguishing “no error” from “error was false”	<code>\$@</code> can be falsey-but-set	<code>catch</code> only runs on a real throw

Use `try` for new code. Reach for `eval` `EXPR` only when you actually need `string-eval`'s runtime compilation.

Examples

Trap an exception and log it:

```
use feature 'try';

try {
    my $x = call_a_function();
    $x < 100 or die "too big: $x";
    send_output($x);
}
catch ($e) {
    warn "unable to output a value: $e";
}
print "finished\n";
```

Use `try` as an expression inside `do` to pick a fallback:

```
use feature 'try';

my $config = do {
    try { load_config($path) }
    catch ($e) { warn "using defaults: $e"; default_config() }
};
```

Release a resource unconditionally with `finally`:

```
use feature 'try';
no warnings 'experimental::try'; # finally is still experimental

my $lock = acquire_lock();
try {
    do_critical_work();
}
catch ($e) {
    warn "critical work failed: $e";
}
finally {
    release_lock($lock);
}
```

Classify exceptions inside a single catch block:

```
use feature 'try';

try {
    fetch_remote();
}
catch ($e) {
    if (ref $e && $e->isa('Net::Timeout')) {
        retry_later();
    }
    elsif (ref $e && $e->isa('Net::Auth')) {
        die $e; # rethrow auth failures
    }
    else {
        warn "unexpected: $e";
    }
}
```

return inside try returns from the enclosing sub - not just the block:

```
use feature 'try';

sub first_valid {
    for my $path (@_) {
        try {
            return load($path); # exits first_valid on success
        }
        catch ($e) {
            warn "skip $path: $e";
        }
    }
    return; # all paths failed
}
```

Edge cases

- **Feature guard required.** Without use feature 'try' (or a use v5.34 or later bundle that enables it), try is a plain identifier and the construct is a syntax error.
- **catch is mandatory.** try { ... } on its own is not legal - every try needs a catch. Use `defer` or `finally` if you want cleanup without exception handling, paired with a trivial catch `($e) { die $e }` rethrow if you also want to let the exception propagate.
- **Rethrowing.** Inside catch, `die $e` rethrows. There is no implicit rethrow if you simply fall off the end of catch; doing so **swallows** the exception. Decide deliberately.
- **\$@ is not cleared** by entering try and is not set by a caught exception. Do not read `$@` inside catch expecting the thrown value - read the lexical.
- **caller() does not see try.** Like while or foreach, try is not a call frame. caller skips it, which is correct: since return propagates through try, try has no independent return semantics worth exposing to introspection.
- **Loop controls pass through.** `last`, `next`, and `redo` inside try or catch act on the enclosing loop, not on the try construct. A finally block will still run before the transfer completes.
- **finally restrictions are enforced.** return, goto, and loop controls inside finally are errors. If you need conditional cleanup that may abort, do it in catch or outside the construct.
- **No try-expression sugar beyond last-statement yield.** try is not a general expression; it can only produce a value when it is the final statement of a block that yields (`do`, a sub). `my $x = try { ... } catch ($e) { ... };` is a syntax error.

Differences from upstream

Fully compatible with upstream Perl 5.42.

- try and catch are non-experimental (as of 5.40) and emit no warnings.
- finally is still experimental in 5.42 and emits `experimental::try` warnings when used; silence with no warnings `'experimental::try'`.

See also

- `catch` - the mandatory second block; receives the exception in a parenthesised lexical
- `finally` - optional third block for unconditional cleanup; still experimental in 5.42
- `die` - the throw side of the construct; any scalar (string, blessed ref, object) can be the thrown value
- `eval` - the older block-eval / `$@` pattern, still needed for string eval; prefer try for pure exception handling
- `defer` - unconditional cleanup without a surrounding try/catch; the semantic basis on which finally is defined

- `return` - inside `try` returns from the enclosing sub, not from the block; inside `finally` it is an error

SCALARs and strings

uc

Return an uppercased copy of a string.

`uc` walks `EXPR` character by character and returns a new string in which every cased character has been replaced by its uppercase counterpart. Characters that have no uppercase mapping - digits, punctuation, already-uppercase letters, symbols - are passed through unchanged. If `EXPR` is omitted, `uc` operates on `$_`. The input is never modified; `uc` always returns a fresh string.

`uc` is the internal function implementing the `\U` escape in double-quoted strings, so `"\Ufoo\E"` and `uc("foo")` produce the same result.

Synopsis

```
uc EXPR
uc
```

What you get back

A new scalar string containing the uppercased version of `EXPR`. The length in characters is unchanged for all cased letters in the Basic Multilingual Plane; a few characters fold to a longer form under Unicode rules (see *Edge cases*), so the byte length and character length of the result can both grow.

```
my $s = uc("Perl is GREAT");    # "PERL IS GREAT"
```

Global state it touches

- `$_` - used as the implicit argument when `EXPR` is omitted.
- `LC_CTYPE` locale - consulted when `use locale` is active. `uc` otherwise ignores the environment.
- `use bytes` / `use feature 'unicode_strings'` / `use locale` - the lexical pragmas in scope at the **call site** select which of the casing rule sets below is applied.

Which rules apply

`uc` picks one of three rule sets based on the pragmas in effect and the internal representation of the string. The logic matches `lc` exactly; the mapping table is just the reverse.

- **use bytes in effect** - ASCII rules. Only `a-z` change, to `A-Z` respectively. Every byte outside `0x61..0x7A` is passed through unchanged, regardless of what it would mean as Latin-1 or UTF-8. Use this only when you have deliberately opted out of Unicode handling.

- **use locale for LC_CTYPE in effect** - the current locale governs code points below 256; Unicode rules govern the rest (the latter only reachable if the string has the UTF-8 flag set). From v5.20 onward, a UTF-8 locale gives full Unicode rules for the whole string. Under non-UTF-8 locales, case changes that cross the 255/256 boundary are not well-defined and, since v5.22, trigger a locale warning. See `perllocale`.
- **String has the UTF-8 flag set** - Unicode rules.
- **use feature 'unicode_strings' or use locale ':not_characters' in effect** - Unicode rules, even for byte strings.
- **Otherwise** - ASCII rules. Anything outside a-z is passed through unchanged. This is the historic default for strings that have neither the UTF-8 flag nor a lexical pragma opting into Unicode.

The practical takeaway: if you want predictable Unicode uppercasing of arbitrary input, put `use feature 'unicode_strings';` (or a modern `use v5.12;` or higher) at the top of the file and stop worrying about which representation the string happens to have.

Examples

Basic ASCII:

```
my $s = uc("hello");           # "HELLO"
```

Omitted argument operates on `$_`:

```
for ("alpha", "beta") {
    print uc, "\n";             # "ALPHA\n", "BETA\n"
}
```

`\U` in a double-quoted string is the same operation:

```
my $s = "Perl is \Ugreat\E";    # "Perl is GREAT"
```

Unicode uppercasing under `unicode_strings`:

```
use feature 'unicode_strings';
my $s = uc("straße");           # "STRASSE"
```

The German sharp s has no single-character uppercase in Unicode; it maps to the two-character sequence SS. The returned string is therefore one character longer than the input.

Greek lowercase letters uppercase into capitals, with unchanged characters passed through:

```
use feature 'unicode_strings';
my $s = uc("Καλημέρα, world!"); # "ΚΑΛΗΜΕΡΑ, WORLD!"
```

Byte mode confines the operation to ASCII, which is occasionally what you want for protocol tokens:

```
use bytes;  
my $s = uc("héllö");           # "HéLLÖ" - only the ASCII letters  
↪ change
```

Edge cases

- **undef**: `uc(undef)` returns "" and triggers an uninitialized warning under `use warnings`. Guard inputs that may be undefined if that warning matters.
- **Empty string**: `uc("")` returns "". No warning.
- **One-to-many mappings**: A small number of characters expand to multiple characters on uppercase - the best-known is U+00DF (ß) → SS, and the Greek U+0390 / U+03B0 expansions. The result string is longer than the input in those cases. Do not assume `length(uc($s)) == length($s)`.
- **Titlecase vs uppercase**: `uc` does **not** titlecase the first letter. Titlecase is a distinct Unicode category; `ucfirst` applies it to the first character and leaves the rest alone.
- **Non-cased characters**: Digits, punctuation, whitespace, symbols, and CJK ideographs are returned unchanged. `uc("42!")` is "42!".
- **In-place update is not a thing**: `uc` returns a new value; it never modifies its argument. To uppercase in place, assign back: `$s = uc $s`.
- **Context**: `uc` is always scalar. Calling it in list context still produces a single string.
- **Byte-string / Unicode-flag surprises**: A string built from a `read` without a `:utf8` layer has no UTF-8 flag, so `uc` without `use feature 'unicode_strings'` will apply **ASCII rules** even if the bytes are valid UTF-8. Decode first, or enable the pragma, to get the casing the bytes look like they should get.
- **use locale and non-UTF-8 locales (v5.22+)**: Case changes that would cross the 255/256 boundary emit a Can't do uc("...") on non-UTF-8 locale warning and return the input character unchanged.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `lc` - the inverse operation; lowercases every cased character using the same rule selection
- `ucfirst` - uppercase (titlecase) only the first character of the string; for a Unicode-aware `ucfirst`. `lc $rest` pattern when titlecasing a whole word
- `lcfirst` - lowercase only the first character
- `fc` - Unicode casefolding for case-insensitive comparison; use this, not `uc` or `lc`, when comparing strings for equivalence
- `sprintf` - the `%s` conversion does not casefold; combine with `uc` when a format needs an uppercased field

- `tr///` - the `tr/a-z/A-Z/` idiom uppercases only ASCII and is faster than `uc` for guaranteed-ASCII input

SCALARs and strings

ucfirst

Return a copy of a string with its first character titlecased.

`ucfirst` takes a string, converts its **first character** to the Unicode **titlecase** form, and returns the result. Every other character is left untouched. The original is never modified. If the argument is omitted, `ucfirst` operates on `$_`. Titlecase usually coincides with uppercase - "a" becomes "A" - but differs for a small number of digraphs in scripts that distinguish the two (see *Edge cases*). The same pragma, locale, and UTF-8-flag rules that govern `lc` and `uc` govern `ucfirst`.

Synopsis

```
ucfirst(EXPR)
ucfirst
```

What you get back

A new string of the **same length in characters** as the input - only the first character is altered. The return value is always a fresh scalar; modifying it does not affect the argument, and modifying the argument afterward does not affect the returned string. The byte length may change by a few bytes if the first character's titlecase form has a different UTF-8 encoding from its original form.

```
my $str = ucfirst("hello world!"); # "Hello world!"
```

Global state it touches

- Reads `$_` when called without an argument.
- Observes `use locale` for `LC_CTYPE` - with locale in effect the current locale's up-casing table applies to a first character whose code point is below 256.
- Observes `use bytes` - under `use bytes` only a leading a-z changes, to A-Z.
- Observes `use feature 'unicode_strings'` - forces Unicode rules regardless of the UTF-8 flag on the input.

Which casing rules apply

`ucfirst` picks one of five rulesets for the **first character**, in priority order. The first one whose condition holds wins. Subsequent characters are always passed through unchanged, regardless of which ruleset applied to the first.

1. **use bytes in effect** - ASCII rules. A leading a-z maps to A-Z; every other leading byte is left alone, including bytes in the 128-255 range that would otherwise be Latin-1 letters.

2. **use locale for LC_CTYPE in effect** - the current locale's tables apply to a first character below code point 256; a first character at 256 or above (only reachable when the string already carries the UTF-8 flag) uses Unicode rules. From Perl 5.20 onward, a UTF-8 locale uses full Unicode rules throughout.
3. **The argument has the UTF-8 flag set** - Unicode titlecase rules apply to the first character.
4. **use feature 'unicode_strings' or use locale ':not_characters' in effect** - Unicode titlecase rules apply to the first character, regardless of the UTF-8 flag.
5. **Otherwise** - ASCII rules. A leading character outside a-z is returned unchanged, including Latin-1 lowercase letters like à-þ.

The upshot: if you want predictable Unicode behaviour on every string regardless of how it was constructed, enable `use feature 'unicode_strings'` (or use v5.12 and above, which turns it on for you).

Titlecase vs uppercase

For the overwhelming majority of characters, titlecase and uppercase are the same codepoint - `ucfirst("abc")` and `uc(substr("abc", 0, 1))` . `substr("abc", 1)` produce identical results. They differ only for a handful of Unicode characters whose uppercase form is a sequence of two capital letters but whose titlecase form is a single capital-followed-by-lowercase letter. The standard example is the Latin digraph dz (U+01F3):

- uppercase is DZ (U+01F1) - "DZ" rendered as one capital glyph.
- titlecase is Dz (U+01F2) - "Dz", capital D followed by small z.

`ucfirst` returns titlecase, which is the correct form when the character stands at the start of a capitalised word. `uc` on the same first character would return the two-capital form, which reads as if the whole word were in caps.

Examples

Basic ASCII first-character uppercasing:

```
my $s = ucfirst("hello, world!");      # "Hello, world!"
```

Default to \$_:

```
for ("foo", "bar", "baz") {
    print ucfirst, "\n";                # Foo / Bar / Baz
}
```

Unicode characters only titlecase when the rules allow it:

```
use feature 'unicode_strings';
my $s = ucfirst("äpfel");               # "Äpfel"
```

Without `unicode_strings` and without the UTF-8 flag on the input, a non-ASCII first character passes through unchanged:

```
my $bytes = "\xE4pfe1";           # ä as a single Latin-1 byte
my $cap   = ucfirst $bytes;       # unchanged: "\xE4pfe1"
```

ucfirst is what backs the \u escape inside double-quoted strings:

```
my $str = "\uperl\E is great";    # "Perl is great"
```

Capitalise every word - combine with `split`, `map`, and `join`:

```
my $title = join " ", map { ucfirst lc $_ } split / /, "HELLO world"
→ "Hello World"
```

Titlecase a digraph that distinguishes titlecase from uppercase:

```
use feature 'unicode_strings';
my $t = ucfirst("\x{01F3}xyz");    # "\x{01F2}xyz" - Dz, not DZ
```

Edge cases

- **undef argument** raises an uninitialized warning under `use warnings` and returns the empty string.
- **Empty string** returns the empty string - there is no first character to change.
- **Single-character string** behaves exactly like `uc` would for that one character under the same ruleset, except in the digraph cases described above where titlecase and uppercase differ.
- **Numbers are stringified first**: `ucfirst(42)` returns `"42"` - the leading `"4"` is not a cased character, so nothing changes.
- **Leading whitespace or punctuation is not skipped**. `ucfirst(" hello")` returns `" hello"` - the first character is the space, which has no titlecase form. If you need to capitalise the first *letter*, strip leading non-letters first or use a regex (`s/\b(\w)/\u$1/`).
- **LATIN SMALL LETTER SHARP S (U+00DE, ß)** titlecases to `"Ss"` under full Unicode rules - a length change that contradicts the usual “same length in characters” guarantee. Under `use locale` on a non-UTF-8 locale, Perl leaves the character unchanged rather than guessing, and from Perl 5.22 onward this raises a locale warning.
- **Locale and the UTF-8 flag interact**: under `use locale` on a non-UTF-8 locale, a first character below 256 follows the locale while a first character 256 or above follows Unicode. In practice `ucfirst` only ever touches one character, so the interaction is subtler than for `lc` / `uc`, but it is still observable in strings that begin with supplementary-plane characters.
- **Tied variables** have their `FETCH` called once. `ucfirst` does not modify the tied value; it returns a plain (non-tied) scalar.
- **ucfirst is a unary named operator**, not a list operator. `ucfirst $a, $b` parses as `(ucfirst $a), $b` - only `$a` is transformed, and the comma operator discards the result. Use parentheses or `map` when you mean to titlecase multiple strings:

```
my @capped = map { ucfirst } @words;
```

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `lcfirst` - the lowercasing counterpart; turns the first character into its lowercase form
- `uc` - uppercase every character in the string; use when you want the whole string capitalised, not just the first letter
- `fc` - Unicode casefold, the correct choice for case-insensitive comparison when input may contain non-ASCII
- `$_` - the default subject `ucfirst` reads when called with no argument
- `use locale` - controls whether locale tables or ASCII/Unicode rules apply to the first character when it is below code point 256
- `use feature 'unicode_strings'` - forces full Unicode titlecasing regardless of the UTF-8 flag on the input

Filehandles, files, directories

umask

Set or read the process file-creation mode mask.

`umask` controls which permission bits are **cleared** from the `MODE` argument of every file- or directory-creation call made by this process from now on. It is a property of the running process, not of any particular filehandle, and it is inherited by child processes. Call `umask EXPR` to install a new mask and receive the previous one; call bare `umask` to read the current mask without changing it.

Synopsis

```
umask EXPR
umask
```

What you get back

The **previous** `umask` as an integer, so you can save it and restore it later:

```
my $old = umask 0077;
# ... create private files ...
umask $old;
```

Bare `umask` returns the current mask without changing it. If the system has no `umask(2)` call and the caller is trying to restrict access for itself (`(EXPR & 0700) > 0`), an exception is raised; if the call is not implemented and the caller is not restricting its own access, `undef` is returned. On Linux this case does not arise.

How the mask is applied

The mask is **subtractive**. Every bit set in the `umask` is cleared from the `MODE` passed to `mkdir`, `sysopen`, `open` in creation modes, and the underlying `creat(2)` / `mkdir(2)` system calls. The effective permissions of the new file are `MODE & ~umask`.

A Unix permission like `rw-r-x---` is three sets of three bits, or three octal digits - `0750`. The leading `0` marks the literal as octal and is not itself a permission digit. A `umask` is written the same way and names the bits you want **disabled** for newly created files.

Worked example. With `umask 0022`, asking `sysopen` for `0666` actually creates a file with mode `0644`:

```
0666 & ~ 0022 = 0644
rw-rw-rw-   - - - -w - -w-   rw-r--r--
```

With `umask 0027` (group may not write, others may not read, write, or execute), the same `0666` request yields `0640` (`0666 & ~0027 == 0640`).

Global state it touches

The mask itself is process-global kernel state set via `umask(2)`. `umask` does not read or write any Perl special variable directly, but every creation call in the process observes it, and `$!` is set by the underlying system calls that *consume* the mask (`mkdir`, `sysopen`, `open`), not by `umask` itself. Child processes spawned via `fork`, `exec`, or `system` inherit the current value.

Examples

Install a mask that hides the file from everyone but the owner, keeping the previous value for restoration:

```
my $old = umask 0077;
open my $fh, ">", "secret.txt" or die "open: $!";
# secret.txt is created mode 0666 & ~0077 == 0600
close $fh;
umask $old;
```

Read the current mask without changing it - useful for logging or diagnostics:

```
printf "current umask: %04o\n", umask;      # e.g. "current umask:
↪0022"
```

Typical values to pass - these are the three masks actually used in practice:

```
umask 0022;  # group + other: read/execute, no write (default on
↪most systems)
```

(continues on next page)

(continued from previous page)

```
umask 0027; # group: read/execute; other: nothing
umask 0077; # owner-only; nothing for group or other
```

Scoped change around a block of file-creating code. `umask` has no `local` magic of its own - save and restore explicitly:

```
sub write_private_file {
    my ($path, $data) = @_;
    my $old = umask 0077;
    open my $fh, ">", $path or do {
        umask $old;
        die "open $path: $!";
    };
    print $fh $data;
    close $fh;
    umask $old;
}
```

Octal literal vs. string - a common trap. A `umask` is a **number**, not a string of octal digits. Reading one from configuration requires `oct`:

```
my $from_config = "0027";           # string
umask oct $from_config;              # correct: 0027 as a
↪number                             #
umask $from_config;                  # WRONG: decimal 27 ==
↪033 octal
```

Edge cases

- **Bare `umask` in list context** still returns a single integer. The function is scalar-valued.
- **Negative or out-of-range values** are masked to the nine low-order permission bits by the kernel; only bits 0777 survive. Passing 0777 disables all permissions on every newly created file, which is almost never what you want.
- **`umask` does not affect existing files.** It only filters the `MODE` argument of *creation* calls. To change an existing file's permissions use `chmod`.
- **`umask` does not affect `open` for an existing file** - opening a file that already exists does not re-apply the mask. It applies only when a file is being created (`O_CREAT` in `sysopen`, or `>/>/+>` modes creating a new pathname in `open`).
- **Inheritance across `fork/exec`.** Children start with the parent's current `umask`. Setting the mask before `system` or `exec` changes the environment the child inherits; restore it afterwards if the parent still cares.
- **Threads.** The `umask` is a per-process attribute in the kernel, so on systems where Perl threads share a process (the usual case) every thread sees the same mask. Changing it in one thread changes it for all.
- **`umask(2)` not implemented.** The POD documents a fallback path for systems without the syscall: raise on attempts to restrict the owner's own access, otherwise re-

turn `undef`. Every platform pperl targets has `umask(2)`, so this path is effectively unreachable here.

- **Octal presentation.** Print the mask with `printf "%04o"`; the default `%d` prints decimal, which is unreadable as permission bits.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `chmod` - change permissions on an **existing** file; the counterpart to the creation-time filtering `umask` performs
- `mkdir` - directory creation call whose `MODE` argument is filtered through the current `umask`
- `sysopen` - low-level file creation whose `MODE` argument is filtered through the current `umask`; the canonical place where the mask is felt
- `open` - in creation modes (`>`, `>>`, `>+`) the underlying `creat(2)` applies the `umask` to the default `0666`
- `oct` - convert a string like `"0027"` to the integer value `umask` expects
- `$!` - set by the creation calls that consume the mask, not by `umask` itself; check it after the `open` / `sysopen` / `mkdir` that fails

Misc

undef

The undefined value, and the operator that produces it.

`undef` wears two hats. Bare `undef` is a **value** - the canonical undefined scalar; the thing `defined` returns false for. `undef EXPR` is a **unary operator** that undefines its argument in place: the variable, array, hash, subroutine, or typeglob named by `EXPR` is emptied out and, where possible, its storage is released.

Both forms evaluate to the undefined value.

Synopsis

```
undef                # the undefined value
undef EXPR           # undefine the lvalue EXPR, return undef
```

`EXPR` must be an lvalue: a scalar, an array (with `@`), a hash (with `%`), a subroutine (with `&`), or a typeglob (with `*`).

What you get back

Always the undefined value. The operator form returns `undef` *after* wiping its argument; the value form just hands you `undef`.

```
my $x = undef;           # $x is now undef
return undef if $oops;   # explicit "I mean no value"
my $was = undef $counter; # $counter wiped, $was is undef
```

The value vs. the operator

Bare `undef` is the undefined scalar. Assigning it to a variable leaves the variable's storage in place and sets its value to `undef`:

```
my @big = (1) x 1_000_000;
$big[0] = undef;           # slot 0 holds undef; @big still huge
@big = ();                 # empties the array, keeps the AV
@big = undef;              # assigns the ONE-element list (undef)
```

Note the last line: `@big = undef` is a list assignment of a one-element list whose only element is `undef`, so `@big` ends up with length 1. It is almost never what you want. Use `@big = ()` to empty, or `undef @big` to empty *and* release the backing storage.

The operator form does more than assign `undef`:

```
undef $scalar;            # scalar: sets value to undef, frees any PV buffer
undef @array;             # array: frees all elements AND the AV's storage
undef %hash;              # hash:  frees all entries AND the HV's buckets
undef &sub;               # sub:   frees the CV body; the name still exists
undef *glob;              # glob:  destroys $glob, @glob, %glob, &glob, etc.
```

The distinction matters for memory. `@array = ()` leaves the allocated capacity intact so the next push is cheap; `undef @array` releases it. For a large, one-shot array that you want to truly let go of, use `undef`. For a working array you will refill, use `= ()`.

Examples

Returning “no value” from a subroutine. In scalar context this is a clean way to signal failure; in list context `undef` alone produces a one-element list containing `undef`, which is truthy:

```
sub find_user {
    my ($id) = @_;
```

(continues on next page)

(continued from previous page)

```

    return undef unless $id;          # scalar-context caller: false
    ...
}

```

For subroutines meant to signal failure in both contexts, return an empty list instead - `return` with no argument does the right thing:

```

sub find_user {
    my ($id) = @_;
    return unless $id;                # () in list ctx, undef in _
    ↪ scalar ctx
    ...
}

```

Skipping positional values in a list assignment. `undef` on the **left-hand side** of a list assignment is a placeholder that throws away the matching value:

```

my ($x, undef, $z) = foo();          # discard the middle value
my (undef, @rest) = @_;              # drop the first arg, keep the _
↪ rest

```

Classic `select` idiom for a sub-second sleep - three ignored filehandle sets plus a timeout:

```

select undef, undef, undef, 0.25;   # sleep 250 ms

```

Dual-context return:

```

return wantarray ? (undef, $errmsg) : undef if $they_blew_it;

```

Releasing a large working set:

```

my %cache = build_huge_cache();
use_cache(\%cache);
undef %cache;                        # free buckets and entries now

```

Undefining a subroutine. The name in the symbol table stays, but calling it will complain about an undefined subroutine:

```

sub greet { print "hi\n" }
undef &greet;
greet();                             # Undefined subroutine &
↪ main::greet

```

Undefining a typeglob nuked every slot at once:

```

our $xyz = 1;
our @xyz = (1, 2);
our %xyz = (a => 1);
sub xyz { 42 }

```

(continues on next page)

(continued from previous page)

```
undef *xyz;                                # $xyz, @xyz, %xyz, &xyz all
↳gone
```

Edge cases

- **undef \$hash{\$key} is not delete.** It sets the value at that key to undef; the key itself still exists and `exists` still returns true. Use `delete $hash{$key}` to remove the key. The same applies to array elements: `undef $arr[3]` leaves index 3 in place with value undef, it does not shorten the array.
- **On tied variables and DBM values,** `undef $tied{$key}` invokes the tie's STORE, not its DELETE. The perldoc warning (“probably not do what you expect on most predefined variables or DBM list values”) is specifically about this.
- **undef is a unary operator, not a list operator.** It takes exactly one argument, so `undef $a, $b` parses as `(undef $a), $b` - only \$a is undefined. To undefine several variables, call it several times, or use a loop:

```
undef $_ for $a, $b, $c;
```

- **undef EXPR requires an lvalue.** `undef 42` or `undef func()` is a compile-time error (*Can't modify ... in undef operator*).
- **Left-hand undef in list assignment is syntax, not a value.** `(undef, $x) = @pair` is parsed specially; you cannot write `my $slot = undef; ($slot, $x) = @pair` and expect the first value to be discarded - that assigns into \$slot.
- **Constant undef in boolean context is false,** in numeric context is 0, in string context is "". Under `use warnings` each of those conversions emits a `Use of uninitialized value` warning at the point of use, not at the point the variable became undef.
- **undef &sub does not remove the subroutine name** from its package stash - only the code. To remove the name as well, undefine the whole glob slot: `delete $Package::{'sub_name'}` or `undef *Package::{'sub_name'}` if you also want the sibling slots gone.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- **defined** - the predicate that asks “is this value not undef?”; the natural counterpart to the `undef` value
- **delete** - remove a hash key or array element entirely; use this, not `undef`, when you want `exists` to become false
- **exists** - check whether a hash key or array index is present, regardless of whether its value is defined

- `local` - temporarily replace a package variable's value (often with `undef`) for the duration of a scope
- `wantarray` - pairs with `undef` in the idiomatic `wantarray ? (undef, $err) : undef` dual-context failure return

Filehandles, files, directories

unlink

Remove one or more directory entries that name files.

`unlink` asks the kernel to drop the link between each name in `LIST` and the file it refers to. If that link was the last one and no process still holds the file open, the file's storage is reclaimed; otherwise the file remains on disk, nameless but accessible through any open descriptor, until the last handle is closed. `unlink` operates on **directory entries**, not on file contents.

Synopsis

```
unlink LIST
unlink
```

What you get back

The number of files `unlink` actually removed, as an integer. On total success that equals scalar `@LIST`; on partial or total failure it is the count of names that were removed before and after the ones that failed. Zero is a valid “false” return when nothing could be removed. `$!` is set to the error from the **last** failing call - it does **not** tell you which name failed.

```
my $removed = unlink 'a', 'b', 'c';
unlink @goners;
unlink glob "/*.bak";
```

If `LIST` is omitted, `unlink` operates on the single filename in `$_`.

Global state it touches

- `$_` - used as the operand when `unlink` is called with no arguments.
- `$!` - set to `errno` from the last `unlink(2)` that failed. Because it reflects only the final failure in the list, it is not a reliable way to diagnose which name could not be removed.

Finding out which file failed

`unlink` returns a count, not a per-file status, and overwrites `$!` on every iteration. To attribute errors to individual names, call `unlink` once per file:

```
foreach my $file (@goners) {  
    unlink $file  
    or warn "could not unlink $file: $!";  
}
```

The per-file form also lets you keep going after a failure, which the list form does implicitly but silently.

Directories

`unlink` does **not** remove directories. On Linux it returns 0 and sets `$!` to `EISDIR` when asked to. The perl binary recognises a `-U` command-line switch that, combined with running as superuser, lifts the check - but even then unlinking a directory bypasses the filesystem's directory-integrity guarantees and can corrupt the parent directory's link count. Use `rmdir` instead; it calls `rmdir(2)`, which is the correct syscall for removing an empty directory entry.

```
rmdir $dir or die "rmdir $dir: $!";    # correct  
unlink $dir;                          # fails with EISDIR
```

POSIX deletion semantics

`unlink` removes the **name**, not the file. As long as any process holds the file open, the file's data and inode remain valid and that process can keep reading and writing through its descriptor. Storage is reclaimed only when the last name is gone **and** the last descriptor is closed. This is the idiomatic pattern for creating a temporary file that cleans itself up even on a crash:

```
open my $fh, "+>", $path or die "open $path: $!";  
unlink $path or die "unlink $path: $!";  
# $fh is still usable; $path is gone from the directory.  
# When $fh closes or the process exits, the storage is freed.
```

It is also why a long-running program can keep writing to a log file whose name has been unlinked by an operator - the writes succeed, but nothing on disk is visible under the old name. `df` will report the space as used until the handle closes.

Symbolic links

If a name in `LIST` is a symbolic link, `unlink` removes the **link itself**, not the file it points at. The target is left untouched even if the link is dangling or points to a directory. This matches the `unlink(2)` syscall and is almost always what you want; use `readlink` first if you specifically need to act on the target.

Examples

Remove a single file, check the result:

```
unlink $path  
or die "cannot remove $path: $!";
```

Remove several and report the tally:

```
my @files = glob "*.tmp";
my $gone = unlink @files;
warn "removed $gone of ", scalar @files, " files\n"
    if $gone != @files;
```

Per-file diagnostics (the only reliable way to know which name failed):

```
my @failed;
for my $f (@goners) {
    unlink $f or push @failed, "$f: $!";
}
die "unlink failures:\n", join("\n", @failed), "\n" if @failed;
```

Rely on `$_` in a chain of file tests:

```
for (glob "*.bak") {
    next unless -f && -M _ > 30;    # older than 30 days
    unlink or warn "skip $_: $!";
}
```

Self-cleaning scratch file (POSIX unlink-while-open idiom):

```
open my $scratch, "+>", "/tmp/work.$$" or die $!;
unlink "/tmp/work.$$" or die $!;
# write through $scratch, no cleanup needed on exit
```

Edge cases

- **Empty LIST:** `unlink ( )` returns 0 and does not touch `$!`. Passing an empty array behaves the same way.
- **Non-existent name:** returns 0 for that entry and sets `$!` to `ENOENT`. The rest of the list is still processed.
- **Permission denied:** `unlink(2)` checks write permission on the **containing directory**, not on the file itself. Removing a file you cannot write is allowed as long as you can write its parent directory; removing a file you own in a directory you cannot write is not. `$!` will be `EACCES` or `EPERM`.
- **Sticky directories** (`/tmp`, mode 1777): non-root users may only unlink names they own. Failures surface as `EPERM`.
- **Return value in boolean context:** `unlink @files` or `die` fires only when **every** name failed (count is 0). To die on **any** failure, compare against `scalar @files` or use the per-file loop.
- **Busy file on Linux:** Linux does not prevent `unlink` of a file that is `mmap`'ed, executing, or open - the name is removed and the POSIX semantics above apply. Other operating systems may return `ETXTBSY` or `EBUSY`; this page covers Linux.
- **Taint mode:** names coming from tainted data are allowed as operands to `unlink`; see `perlsec` for the usual caveats about accepting untrusted paths.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `rmdir` - remove an **empty** directory; the correct tool when the name refers to a directory entry of type DIR
- `rename` - move or re-link a name without dropping it, often combined with `unlink` when replacing a file atomically
- `link` - create an additional hard link to a file; `unlink` is its inverse for one name at a time
- `symlink` - create a symbolic link; note that `unlink` removes the symlink itself, not its target
- `truncate` - shrink a file in place without unlinking it when you want to reuse the name

[Lists](#) · [Fixed-length data](#)

unpack

Extract typed values from a binary or fixed-width string according to a template.

`unpack` is the inverse of `pack`. It walks `EXPR` left to right, consuming the bytes described by each directive in `TEMPLATE` and turning them into Perl values. The result is a list - one value per directive, or per repetition when a directive has a count. If `EXPR` is omitted, `unpack` reads from `$_`.

This page is a **directive reference**. For a narrative introduction with worked protocol and file-format examples, start at the [pack/unpack tutorial](#).

Synopsis

```
unpack TEMPLATE, EXPR
unpack TEMPLATE                                # reads from $_
my @fields = unpack "A10 A10 A*", $line;
my ($ver, $len, $payload) = unpack "n N A*", $msg;
```

What you get back

A list of values, one per directive (or per repetition when a directive has a count). In **scalar context**, only the **first** value produced is returned:

```
my @all    = unpack "A4 A4 A*", $rec;    # three values
my $first  = unpack "A4 A4 A*", $rec;    # just the first - not a
↳count
```

Each directive determines the Perl type of its result - see the “Result type” column in the directive table below.

Global state it touches

`unpack TEMPLATE` with no `EXPR` reads `$_`. No other interpreter globals are consulted. The `C0` / `U0` directives switch template byte-vs-character interpretation locally within the template but do not mutate any outside state.

Template syntax

Identical to `pack`: a sequence of directive letters, optionally followed by repeat counts `N` / `*` / `[...]` and modifiers `!` / `<` / `>`. Parentheses form groups; `#` introduces a template comment. See `pack` for the full syntax rules - this page covers only the `unpack`-specific semantics.

Directive table

Every `unpack` directive in one place. **W** is the width in bytes consumed from the input per scalar produced.

Directive	W	Signed	Endian	Result type	Modifiers	Notes
<code>a</code>	count	-	-	string	-	Returns bytes untouched
<code>A</code>	count	-	-	string	-	Strips trailing whitespace and NUL
<code>Z</code>	count	-	-	string	-	Returns everything up to the first NUL
<code>b</code>	count/8	-	-	bit string	-	“0”/“1” characters, LSB of each byte first
<code>B</code>	count/8	-	-	bit string	-	“0”/“1” characters, MSB of each byte first
<code>h</code>	count/2	-	-	hex string	-	[0-9a-f] characters, low nybble first

continues on next page

Table 2 – continued from previous page

Directive	W	Signed	Endian	Result type	Modifiers	Notes
H	count/2	-	-	hex string	-	[0-9a-f] characters, high nybble first
c	1	yes	-	integer	-	Signed char
C	1	no	-	integer	-	Unsigned char
W	1	no	-	integer	-	Unsigned char; yields codepoint in U0 mode
s	2	yes	native	integer	! < >	s! = native short
S	2	no	native	integer	! < >	S! = native unsigned short
l	4	yes	native	integer	! < >	l! = native long
L	4	no	native	integer	! < >	L! = native unsigned long
i	native	yes	native	integer	! < >	sizeof(int)
I	native	no	native	integer	! < >	sizeof(unsigned int)
q	8	yes	native	integer	< >	Requires 64-bit-integer Perl
Q	8	no	native	integer	< >	Requires 64-bit-integer Perl
n	2	no	big	integer	!	Network order; n! = signed
N	4	no	big	integer	!	Network order; N! = signed
v	2	no	little	integer	!	“VAX” order; v! = signed

continues on next page

Table 2 – continued from previous page

Directive	W	Signed	Endian	Result type	Modifiers	Notes
V	4	no	little	integer	!	“VAX” order; V! = signed
j	IV size	yes	native	integer	< >	Perl-internal IV
J	UV size	no	native	integer	< >	Perl-internal UV
f	4	-	native	number	< >	Single-precision IEEE 754
d	8	-	native	number	< >	Double-precision IEEE 754
F	NV size	-	native	number	< >	Perl-internal float
D	varies	-	native	number	< >	Long double
p	ptr size	-	native	string	< >	Dereferences pointer to NUL-terminated string
P	ptr size	-	native	string	< >	Dereferences pointer; count = bytes to read
u	varies	-	-	string	-	Ude-coded bytes
U	varies	-	-	integer	-	Unicode codepoint number
w	varies	no	-	integer	-	BER-compressed integer
x	1	-	-	(none)	!	Skip one byte forward; x!N aligns forward
X	-1	-	-	(none)	!	Back up one byte; X!N aligns backward

continues on next page

Table 2 – continued from previous page

Directive	W	Signed	Endian	Result type	Modifiers	Notes
@	absolute	-	-	(none)	!	Jump to position N within the innermost group
.	-	-	-	integer	!	Return current position (relative to start of group / string)
(...)	-	-	-	-	< > !	Group: repeat count and endianness propagate inside
/	-	-	-	-	-	See “Length-prefixed payloads” below
%N	varies	-	-	integer	-	Prefix. N-bit checksum of the following directive’s values

The modifiers have the same meaning as in `pack`: `!` selects native sizes, alignment semantics, signedness for `n` / `N` / `v` / `V`, or byte offsets for `@` / `.`; `<` / `>` force endianness.

Differences from `pack`

- **a strips nothing** (returns bytes raw); **A strips trailing whitespace and NUL**; **Z stops at the first NUL**. With `pack` all three pad; with `unpack` all three un-pad differently.
- **x skips forward by W, X backs up by W**. X before the start of the string is a fatal error.
- **/ reads a count from the data, then applies it to the next directive:**

```
unpack("W/a", "\004Gurusamy")           # ("Guru")
unpack("a3/A A*", "007 Bond J ")        # (" Bond", "J")
unpack("n/a*", "\x00\x0chello, world")  # ("hello, world")
```

With pack the form is *length-item/item* and the length is computed from the value's length. With unpack the form is */item* (no length-item before) - the preceding integer directive supplies the count.

- **%N is unpack-only.** It replaces the following directive's normal output with the N-bit sum of the values that directive would otherwise have produced. %32W* is the System V sum checksum. %32b* counts set bits.
- **. returns the current byte position** rather than zero-filling to one. Useful after x / X navigation to learn where you are.
- **p and P dereference pointers read from the input** - almost always unsafe unless you know the input was produced by pack "p" / pack "P" in the same process.
- **Under-run / over-run behaviour:** if the template asks for more data than EXPR contains, the result is not well-defined (may yield empty strings, zeros, a decreased repeat count, or an exception). If EXPR is longer than the template consumes, trailing bytes are silently ignored.

Examples

Parse a minimal binary protocol header - a 16-bit version, a 32-bit length, then a variable-length payload, all network byte order:

```
my ($ver, $len, $payload) = unpack "n N A*", $msg;
```

Split a fixed-width text record - faster and clearer than chained `substr` calls:

```
my ($date, $desc, $amount) = unpack "A10 A27 A*", $line;
# "2026-04-22 coffee at the station          3.50"
# $date = "2026-04-22", $desc = "coffee at the station",
# $amount = "3.50"
```

Extract a System V-style checksum with the % prefix - the 32-bit sum of every byte, masked to 16 bits:

```
my $checksum = do {
    local $/;                               # slurp
    unpack "%32W*", readline $fh;
} % 65535;
```

Count set bits in a bit vector - %32b* reads the whole mask as bits and returns their sum:

```
my $setbits = unpack "%32b*", $selectmask;
```

Read a sequence of little-endian 32-bit integers off a raw buffer:

```
my @ints = unpack "V*", $buf;
```

Use x to skip filler bytes, X to back up after peeking:

```
# 2-byte tag, skip 2 reserved bytes, then 4 32-bit big-endian values
my ($tag, @vals) = unpack "n x2 N4", $frame;
```

Edge cases

- **Scalar context returns only the first value.** Assigning unpack to a scalar is almost never what you want. To count fields, assign to a list first, or use `() = unpack ...` in list context:

```
my $n = () = unpack "A10 A10 A*", $line;    # 3
```

- *** is greedy, not a placeholder.** `A*` consumes all remaining bytes as one string; there is no backtracking. A `*` directive must be the last one in its sequence, or nothing after it runs.
- **Templates are not regular expressions.** Whitespace is ignored, but there is no alternation, no `|`, no lookahead. If data shape depends on earlier fields, unpack in stages: decode a header, then call unpack again on the payload using a template derived from what you just read.
- **Byte vs. character semantics.** unpack runs in byte mode by default on a byte string and in character mode on a string with the UTF-8 flag set. `C0` forces byte mode from that point; `U0` forces UTF-8 mode. Mixing a UTF-8-flagged string with numeric directives like `N` reads codepoints, not bytes - usually not what you want. Call `utf8::encode` first, or operate on byte-level data.
- **X past the start of the string is fatal.**
- **p / P on an arbitrary input is undefined behaviour.** Never use them on data you did not pack yourself in the same process.
- **No EXPR argument:** `unpack TEMPLATE` reads `$_`. Useful inside `while (<$fh>)` { ... } loops over fixed-width records.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- [pack](#) - the inverse operation; same template language
- [substr](#) - extract a single fixed slice; simpler for one field, slower for many
- [sprintf](#) - round-trip between binary and human-readable
- [read](#) - read raw bytes off a filehandle into a buffer for unpack
- [vec](#) - indexed access to a bit vector without a template
- [pack/unpack tutorial](#) - task-oriented walkthrough with worked protocol and file-format examples

Arrays

unshift

Prepend one or more values to the start of an array and return the array's new length.

`unshift` is the mirror of `push`: where `push` appends at the tail, `unshift` inserts at the head. The values in `LIST` are prepended as a group, in the order given, so the first argument of `LIST` becomes the new element 0, the second becomes element 1, and so on, with the original contents sliding to higher indices.

Synopsis

```
unshift @arr, $val
unshift @arr, @more
unshift @arr, $first, $second, $third
```

What you get back

The new number of elements in the array, as an integer. This is the same value you would get from `scalar @arr` after the call.

```
my @colors = ("red");
my $n = unshift @colors, "blue", "green";    # $n == 3
```

Most calls discard the return value; it is occasionally useful when you want to prepend and branch on the resulting size in one step.

Examples

Prepend a single element:

```
my @animals = ("cat");
unshift @animals, "mouse";
# @animals is now ("mouse", "cat")
```

Prepend several elements - **the order is preserved**, not reversed. This is the single most common `unshift` confusion:

```
my @a = (10, 20, 30);
unshift @a, 1, 2, 3;
# @a is now (1, 2, 3, 10, 20, 30)
# NOT (3, 2, 1, 10, 20, 30)
```

If you genuinely want the prepended values reversed, reverse them explicitly:

```
my @a = (10, 20, 30);
unshift @a, reverse(1, 2, 3);
# @a is now (3, 2, 1, 10, 20, 30)
```

Prepend a whole array - the elements are spliced in, not the array as a single value:

```
my @head = ("a", "b");
my @tail = ("c", "d");
unshift @tail, @head;
# @tail is now ("a", "b", "c", "d")
```

Insert at the front vs. insert at the rear. These two calls produce mirror-image results on the same starting array:

```
my @x = (2, 3);
push @x, 4;      # @x is (2, 3, 4) - 4 goes to the rear
unshift @x, 1;   # @x is (1, 2, 3, 4) - 1 goes to the front
```

Work on an array through a reference. `unshift` accepts a dereferenced array on the left:

```
my $ref = [10, 20, 30];
unshift @$ref, 0;
# $$ref[0] is now 0, @$ref is (0, 10, 20, 30)
```

The arrow-deref form works the same way; write the deref as its own expression rather than stretching a method chain across the `unshift` argument:

```
my @list = @$ref;
unshift @list, -1;
```

Edge cases

- **Cost is $O(n)$.** Every existing element is shifted up by `scalar(LIST)` slots, so prepending to a large array is proportionally expensive. If you are building a list by repeated prepending, build it with `push` and `reverse` at the end - or build it backwards with indices - instead of unshifting in a loop.
- **List is prepended whole, not reversed.** `unshift @a, 1, 2, 3` places 1 at index 0, 2 at index 1, 3 at index 2. Readers coming from languages where “push to front” naturally reverses (because each element becomes the new head) often expect the opposite; Perl does not.
- **Empty LIST is a no-op.** `unshift @arr` with nothing after it leaves the array unchanged and returns the current element count.
- **Tied arrays:** `unshift` on a tied array calls the `UNSHIFT` handler if the package provides one; otherwise it falls back to a sequence of `SPLICE` or `STORE / STORESIZE` calls. A tie class that only implements `FETCH` and `STORE` can still be unshifted, but every call rewrites every index.
- **Dereferenced array works:** `unshift @$ref, $x` modifies the array referenced by `$ref` in place. `unshift @{$h{key}}, $x` autovivifies `$h{key}` to an empty array reference if it did not previously exist.
- **Scalar expression on the left is a syntax error.** A Perl 5.14 experiment briefly accepted `unshift $scalar, ...` to mean “treat `$scalar` as an array reference and prepend”; it was removed in Perl 5.24 and stays removed. Write `unshift @$scalar, ....`

- **Return value in scalar vs. list context** is the same integer. Unlike many builtins, `unshift` does not behave differently in list context - it always returns the new length.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `push` - the other end: append to the tail of the array and return the new length
- `shift` - the inverse operation: remove and return the first element
- `pop` - symmetric partner to `shift`, removes from the tail
- `splice` - general-purpose insert/remove at an arbitrary position; `unshift @a, LIST` is equivalent to `splice(@a, 0, 0, LIST)`
- `reverse` - pair with `unshift` when you do want the prepended values to end up in reverse order
- `List::Util` - no dedicated prepend; for head-insert semantics `unshift` is the idiomatic tool, and `List::Util::pairs` / `List::Util::zip` compose with it when you are restructuring list shapes rather than just adding to one

Classes and OO

untie

Break the binding between a variable and its tied class.

`untie` reverses a prior `tie`: after the call, `VARIABLE` goes back to being a plain scalar, array, hash, or filehandle, and subsequent reads and writes touch the raw storage rather than being routed through the tied package's `FETCH`, `STORE`, and friends. If the variable is not tied, `untie` is a no-op.

Synopsis

```
untie VARIABLE
```

What you get back

Returns a true value if the variable was tied and the binding was removed, false otherwise. The return value is rarely checked - call sites know whether they are unwinding a tie they set up themselves.

As a side effect, if the tied package defines an `UNTIE` method, it is invoked on the tied object before the binding is dropped. `UNTIE` receives the object and a count of any outstanding inner references held to it (see *Edge cases* below).

What it does

1. Locate the magic on VARIABLE that a prior `tie` installed.
2. If the tied package provides UNTIE, call it on the object.
3. Remove the magic so the variable reverts to ordinary storage.
4. Release the interpreter's reference to the tied object. The object is destroyed (triggering DESTROY if defined) once no other references to it remain.

After `untie`, any value `tied` previously returned for this variable no longer corresponds to a live binding.

Examples

Tie a hash, use it, then release it:

```
use Tie::File;
tie my @lines, 'Tie::File', 'log.txt' or die $!;
push @lines, "new entry";
untie @lines;                                # flush and release
```

A round-trip through `tied` - grab the underlying object so you can call a non-tied method on it, then drop the binding:

```
my $obj = tied %cache;
$obj->flush;
untie %cache;
```

`untie` on a variable that was never tied is harmless:

```
my %h;
untie %h;                                    # no-op, no warning
```

An UNTIE hook that runs before the binding goes away:

```
package My::Tied;
sub TIEHASH { bless {}, shift }
sub UNTIE   { my ($self, $count) = @_;
              warn "untie with $count outstanding refs" if $count }
# ...
tie my %h, 'My::Tied';
untie %h;                                    # calls My::Tied::UNTIE($obj, 0)
```

Edge cases

- **Outstanding inner references:** if code stashed the object returned by `tie` or `tied` somewhere that still holds it at the moment `untie` runs, Perl issues

```
untie attempted while N inner references still exist
```

under use warnings. The binding is removed anyway, but those stashed references continue to point at the now-detached object. Drop those references first, or accept that they will now see the un-tied object in isolation.

- **Calling `untie` on a non-tied variable** is a no-op. No warning, no error, no method call.
- **Filehandle tie:** `untie *FH` removes the tie installed by `tie *FH, $class, ....`. The underlying glob and any real file descriptor behind it are unaffected - `untie` does not `close`.
- **UNTIE is optional:** if the tied package does not define one, the binding is simply removed. Do not rely on UNTIE running; many tied classes omit it.
- **Destruction order:** `untie` does not itself call DESTROY. The object is destroyed when its refcount hits zero, which may be immediately after `untie` (if nothing else holds it) or later (if the caller kept a copy from `tied`).
- **Localised tied variables:** a tie installed inside a scope on a `localised` variable is undone automatically when the scope exits; you do not need an explicit `untie`, and calling one is harmless.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `tie` - install the binding that `untie` removes; shows the full set of tie-hook method names
- `tied` - fetch the object currently bound to a tied variable, typically just before calling `untie` on it
- `bless` - `tie` blesses the object it is given; `untie` does not unbless, it only detaches the variable from it
- `ref` - inspect what kind of reference `tied` returned before deciding to `untie`
- `local` - scope-bounded alternative when you want the tie to go away on block exit without an explicit `untie`

[Scoping](#) · [Modules](#) · [Classes and OO](#)

use

Load a module at compile time and import its symbols into the current package.

`use` is the everyday mechanism for pulling in library code. It runs at **compile time**, not at run time: the named module is located on `@INC`, loaded (if not already in `%INC`), and given a chance to alter the caller's namespace - typically by aliasing subroutine or variable names into it. The same syntax also drives **pragmas** (compiler directives like `strict` or `warnings`); the only difference is convention - pragmas are lowercase and usually have lexical, not package-wide, effects.

A `use` statement is defined to be exactly equivalent to:

```
BEGIN { require Module; Module->import(LIST); }
```

The BEGIN forces both the load and the import to happen while the surrounding file is still being parsed, so the imported names are visible to the compiler for the rest of the file.

Synopsis

```
use Module VERSION LIST
use Module VERSION
use Module LIST
use Module
use Module ( )
use VERSION
```

What you get back

use is a declaration, not an expression - it has no useful return value and cannot appear on the right-hand side of anything. Its effect is a **side effect on the current compilation unit**: package %INC gets an entry, the importing package gets whatever symbols `import` aliased in, and - for use VERSION - the current lexical scope gets a feature bundle turned on.

If the module cannot be loaded, or if `import` dies, the BEGIN block propagates the exception and compilation aborts.

The four module forms

use Module

Loads the module and calls its `import` method with **no arguments**. The module decides what “no arguments” means; for Exporter-based modules it typically means “export your @EXPORT defaults”:

```
use File::Spec;                                # calls File::Spec->import()
```

use Module LIST

Loads the module and calls `import` with LIST:

```
use List::Util qw(first sum max);
```

This is the common “import exactly these names” form. LIST is passed verbatim to `import`; its meaning is the module’s business, not the language’s.

`use Module ()`

Empty parentheses disable the import call entirely. This is distinct from omitting the list - no import method is invoked at all:

```

use POSIX ();                # load, but do not import anything
POSIX::strftime(...);        # call by full name instead

```

Equivalent to:

```

BEGIN { require POSIX }

```

Use this when you want the module loaded but do not want your namespace altered - typically because you will call everything by fully qualified name, or because the module's default exports would clobber something you care about.

`use Module VERSION / use Module VERSION LIST`

When a VERSION appears immediately after Module (no comma), use calls the module's VERSION method with that version as an argument **before** calling `import`:

```

use List::Util 1.45 qw(uniq);

```

is equivalent to:

```

BEGIN {
    require List::Util;
    List::Util->VERSION(1.45);
    List::Util->import(qw(uniq));
}

```

The default VERSION method, inherited from UNIVERSAL, croaks if the requested version is greater than `$Module::VERSION`.

VERSION must look like a version literal - a digit, or v followed by a digit. Arbitrary expressions in that position are parsed as the start of LIST, not as a version. Note there is **no comma after VERSION**.

`use VERSION - request a Perl language level`

`use VERSION` (with no module name) declares the minimum Perl version the file requires, and **lexically enables the matching feature bundle**:

```

use v5.36;                # v-string form - preferred
use 5.036;                # numeric form - equivalent
use 5.036_000;            # older numeric form

```

An exception is raised at compile time if the running Perl is older than VERSION; the rest of the file is not even parsed.

Beyond the version check, `use VERSION` turns on the features bundled with that release:

- **use v5.12** or higher - implicit `use strict`.

- **use v5.35** or higher - implicit use warnings.
- **use v5.36** - the common modern baseline: adds say, signatures, isa, and the earlier bundles' features.
- **use v5.39** or higher - lexically imports builtin functions for the matching bundle.
- **use v5.41** or higher - enables `source::encoding 'ascii'`.

`use VERSION` does **not** load `feature.pm`, `strict.pm`, `warnings.pm`, or `builtin.pm` - it implements the equivalent behaviour directly. Place it at the top of the file, before anything that depends on those features being active.

From Perl 5.39 onward, issuing a second `use VERSION` while one is already in effect is a fatal error. In older bundles it was merely deprecated.

Modules vs pragmas

Syntactically, pragmas and modules are identical - both are just packages whose `import` method you invoke via `use`. The conventional differences:

- **Naming.** Pragmas are lowercase (`strict`, `warnings`, `feature`, `integer`); modules are Studly: `:Caps`.
- **Scope of effect.** Pragmas usually alter the **current lexical scope** (a block, not a package) and unwind at the closing brace. Modules usually alter the **current package** and persist until end-of-file.
- **no.** The counterpart `no` statement calls `unimport` instead of `import`. Pragmas typically implement `unimport` to undo their effect; most ordinary modules do not.

```
use strict;                # lexical - affects this block / ↵
↵file
use warnings;              # lexical
{
    no warnings 'uninitialized';
    # warnings off only in this block
}

use List::Util qw(sum);    # package-scoped - affects main::
```

Conditional loading

Because `use` runs at compile time, it **ignores ordinary runtime control flow**. A `use` inside the false branch of an `if` is still executed:

```
if (0) {
    use Heavy::Module;      # loaded anyway - compile time!
}
```

For genuinely conditional loading, either use `require` at run time, or delegate to the `if` pragma:

```
use if $] >= 5.036, 'experimental', 'builtin';
use if $^O eq 'MSWin32', 'Win32::API';
```

Examples

Load a module with its default exports:

```
use Carp;                                # croak, carp, confess visible here
```

Import a specific subset:

```
use List::Util qw(first min max sum);
```

Load without importing - call by full name:

```
use Scalar::Util ();
my $ok = Scalar::Util::looks_like_number($x);
```

Require a minimum module version:

```
use Moo 2.000000;
```

Declare the Perl language level and pick up its feature bundle:

```
use v5.36;                                # strict, warnings, say, signatures.
↪...
sub greet ($name) {
    say "hello, $name";
}
```

Turn a pragma on for a file, then off for one block:

```
use warnings;

sub legacy_parser {
    no warnings 'uninitialized';
    return join ",", @_;          # undefs render as "" without_
↪warnings
}
```

Conditional pragma via the if helper:

```
use if $] < 5.010, 'MRO::Compat';
```

Edge cases

- **Module must be a bareword.** `use $name;` is a syntax error - use does not accept a runtime expression for the module name. Use `require` for that.
- **No comma between Module and VERSION.** `use List::Util 1.45` is a version check; `use List::Util, 1.45` is a syntax error.

- **The version literal rule is strict.** Anything that does not start with a digit or with `v` followed by a digit is parsed as the beginning of `LIST`, not as a version. `use Module "1.45"` passes the string `"1.45"` to `import`; it does **not** check a version.
- **Empty `LIST` vs omitted `LIST`.** `use Module;` calls `Module->import()`. `use Module ();` does not call `import` at all. The distinction matters for modules whose `import` has side effects even with no arguments.
- **Missing `import` method.** If the module does not define (and does not inherit) an `import` method, the call is silently skipped - no error, even if `AUTOLOAD` is defined.
- **`use` inside `eval`.** `BEGIN` runs at compile time of the `eval`'d code, so `eval "use Some::Module"` delays loading until the `eval` runs. This is the common way to probe for an optional module at run time:

```
eval { require Some::Module; Some::Module->import };  
my $have_it = !$@;
```

- **Ordering of `use VERSION` and other pragmas.** Place `use VERSION` first, so the feature bundle is in effect before later pragmas (which may depend on it) run. A later `no strict 'refs'` still overrides the strictures implied by `use v5.12+`, but relying on that is discouraged.
- **Pragmas that look like modules.** Some pragmas ship with a capitalised name (`UNIVERSAL`, `Internals`); some modules ship lowercase (`if`, `lib`, `ok`). The naming convention is a hint, not a rule - what matters is what the module's `import` does.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `require` - the run-time half of `use`; loads a module without invoking `import`, and accepts a runtime expression for the module name
- `no` - the counterpart declaration; calls `unimport` instead of `import` to undo a pragma's effect
- `import` - not a built-in; an ordinary class method invoked by `use`, usually provided via inheritance from `Exporter`
- `package` - declares the package whose namespace `use` will alter
- `@INC` - the list of directories searched for the module file
- `%INC` - records every file loaded by `use` or `require`, keyed by relative path

[Filehandles](#), [files](#), [directories](#)

utime

Set access and modification times on a list of files.

`utime` applies the numeric access time `ATIME` and modification time `MTIME` - both Unix epoch seconds - to every file named in `LIST`. It is the Perl-level interface to the `utime(2)`

and `utimes(2)` system calls, and the tool behind the Unix `touch(1)` command. The inode change time (`ctime`) of each file is set to the current time as a side effect - this is a kernel invariant, not a Perl choice, and cannot be suppressed.

Synopsis

```
utime ATIME, MTIME, LIST
utime undef, undef, LIST      # set both to "now" (since 5.8.0)
```

What you get back

The number of files whose timestamps were successfully updated. Any file that could not be touched is silently skipped in the count; `$!` reflects the error from the *last* failing file only. Check the return value against scalar `@files` when you need to detect partial failure:

```
my @files = ("a", "b", "c");
my $n = utime $atime, $mtime, @files;
warn "only $n of ", scalar @files, " files updated: $!"
    if $n != @files;
```

Arguments

- `ATIME` - new access time, in seconds since the Unix epoch. Must be numeric; non-numeric scalars are coerced the usual way and a string like "now" becomes 0 (midnight 1970-01-01).
- `MTIME` - new modification time, same format.
- `LIST` - one or more filenames. On systems that support `futimes(2)` (Linux does), filehandles may appear in `LIST` as well. Filehandles must be passed as globs (`*FH`) or glob references (`\*FH`); a bareword in that position is parsed as a filename, not a handle.

The `undef, undef` form

Since Perl 5.8.0, passing `undef` for both `ATIME` and `MTIME` calls `utime(2)` with a null second argument. The kernel then sets both times to the current wall-clock time, and - crucially - this form succeeds on files the caller does not own, provided the caller has **write permission** on the file. The owner-or-root requirement only applies to the explicit-timestamp form.

```
for my $file (@ARGV) {
    utime undef, undef, $file
        or warn "Couldn't touch $file: $!";
}
```

This is the form the real `touch(1)` command uses internally, and is the right choice for a generic “touch” script.

Permissions

Two distinct rules, depending on which form you use:

Form	Who may succeed
<code>utime \$atime, \$mtime, ...</code>	File owner, or root
<code>utime undef, undef, ...</code>	Anyone with write permission on the file

A non-owner attempting the explicit-timestamp form gets `EPERM` in `$!` and the file is left untouched (it still counts as a failure, not a silent no-op).

Examples

Plain Unix-style touch on a list of files you own:

```
my $now = time;
utime $now, $now, @ARGV;
```

Generic touch that also works on files you merely have write access to:

```
for my $file (@ARGV) {
    utime undef, undef, $file
        or warn "Couldn't touch $file: $!";
}
```

Copy timestamps from one file to another using `stat`:

```
my @st = stat $src or die "stat $src: $!";
utime $st[8], $st[9], $dst
    or die "utime $dst: $!";
```

Touch an already-open filehandle on Linux (needs `futimes(2)`):

```
open my $fh, ">>", "log" or die $!;
utime undef, undef, *$fh;      # glob deref, not bareword
```

Set a file one hour into the past, useful for build-system tests:

```
my $t = time - 3600;
utime $t, $t, "stale.tmp";
```

Edge cases

- **Mixed undef and number:** passing `undef` for only one of the first two arguments does **not** trigger the “both-undef” path. The `undef` is coerced to `0` (epoch midnight 1970-01-01), the other value is used verbatim, and an uninitialized warning fires under `use warnings`. This is almost never what you want.

```
use warnings;
utime undef, time, $file;      # atime = 1970-01-01, warning
```

- **Non-numeric timestamps:** ATIME and MTIME must be numeric. String values are coerced - "1700000000" works; "yesterday" silently becomes 0.
- **ctime always moves:** there is no way to preserve a file's inode change time across a utime call. If you need that, you need filesystem-level tools or a snapshot, not Perl.
- **Filehandle in LIST:** must be a glob (*FH) or glob reference (*FH). A bareword like FH in that position is parsed as the filename "FH", and a string scalar holding a handle name is likewise treated as a filename.
- **Non-existent file:** counts as a failure and does not increment the return value; \$! is set to ENOENT (No such file or directory).
- **Partial failure:** the count reflects successes only. To know *which* file failed, loop over the list one at a time.
- **NFS clock skew:** the server's clock is authoritative, not the client's. A noticeable difference between the two shows up as timestamps that do not match what `time` on the client returned.
- **Sub-second precision:** Perl's utime operates on whole seconds. Filesystems and kernels that support nanosecond timestamps will see the fractional component zeroed. Reach for `Time::HiRes::utime(XS)` if you need better resolution.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `stat` - read the timestamps utime writes; elements 8 (atime), 9 (mtime), and 10 (ctime) of the returned list
- `time` - current epoch seconds, the usual source for both ATIME and MTIME
- `open` and `close` - filehandles that utime can touch on systems with `futimes(2)`
- `$!` - system error from the last failing file when the return count is less than scalar @files
- `utime(2)`, `utimes(2)`, `futimes(2)` - the underlying system calls

[Arrays](#) · [Hashes](#)

values

List every value of a hash (or every element of an array).

`values` is the value-side companion to `keys`. Given a hash, it returns all the values; given an array, it returns all the elements. The elements come back as **aliases** into the underlying container, so assigning to them from a loop modifies the container in place. As a side effect, calling `values` resets the iterator that `each`, `keys`, and `values` share for that hash or array.

Synopsis

```
my @v = values %hash;
my @v = values @array;
my $n = scalar values %hash;    # number of values
for (values %hash) { $_ *= 2 } # mutate in place via aliasing
```

What you get back

In list context: the full list of values from a hash, or the full list of elements from an array. Array entries come back in index order (lowest index first). Hash entries come back in an implementation-defined, per-hash random order - see *Edge cases*.

In scalar context: the number of entries, identical to `scalar keys %h` for hashes and `scalar @a` for arrays. `values` in void context does nothing observable **except** reset the iterator - that is the idiomatic way to force-reset an iterator mid-traversal:

```
values %h;                                # void context: iterator reset only
```

The returned list is not a copy. Each element is an alias to the live value in the container. Aliasing is documented behaviour and is the basis for the in-place mutation idiom under *Examples*.

Global state it touches

`values` resets the per-container iterator state that `each` reads and advances. Any `while (my ($k, $v) = each %h) { ... }` loop in progress on the same hash will restart from the beginning the next time `each` is called. `keys` has the same effect. The iterator is a property of the container, not of any particular call site - so a caller in one sub can disturb an `each` loop running in another sub if they operate on the same hash.

Examples

Iterate a hash in key-sorted order, which `values` alone does not give you:

```
my %score = (alice => 90, bob => 75, carol => 82);
for my $k (sort keys %score) {
    print "$k: $score{$k}\n";
}
```

Sum every value of a hash:

```
use List::Util qw(sum0);
my %count = (apples => 3, pears => 5, plums => 2);
my $total = sum0 values %count;    # 10
```

Mutate every value in place through the alias:

```
my %prices = (bread => 3.00, milk => 2.50, eggs => 4.20);
$_ *= 1.10 for values %prices;    # apply 10% price rise
# %prices is now (bread => 3.30, milk => 2.75, eggs => 4.62)
```

The same idiom works on an array, though for arrays a plain `@array` loop is more conventional:

```
my @nums = (1, 2, 3, 4);
$_ **= 2 for values @nums;           # @nums is now (1, 4, 9, 16)
```

Force-reset an in-progress `each` iterator without consuming its output:

```
while (my ($k, $v) = each %h) {
    last if $v > $threshold;
}
values %h;                               # void context: drop iterator_
↪ state
# next each %h starts from the beginning
```

Count entries without building the list - scalar context never materialises the values:

```
if (scalar values %cache > 10_000) {
    prune(\%cache);
}
```

Edge cases

- **Aliasing is the feature and the footgun.** `for (values %h)` gives you aliases, not copies. A harmless-looking `s///` inside such a loop rewrites the hash. If you want copies, assign first: `my @v = values %h; for (@v) { ... }`.
- **Hash order is randomised.** Perl 5.18+ randomises the traversal order per-hash. Two hashes built from the same keys can yield values in different orders. Never rely on the order for anything that leaves the process - serialise via `sort keys` if you need stability.
- **Order is stable only while the hash is unmodified.** Insertions and deletions may permute the order. The one documented exception is deleting the key most recently returned by `each` or `keys`, which leaves the remaining order intact.
- **keys, values, each agree.** For a hash that has not been modified between calls, `keys %h` and `values %h` produce corresponding lists in the same order - `(keys %h)[i]` and `(values %h)[i]` name the same entry.
- **Array form behaves like @array plus a reset.** `values @a` in list context is equivalent to `@a`, with the side effect of resetting the array's iterator. For the iterator-reset alone, `keys @a` in void context is the conventional choice.
- **Empty hash or array** returns the empty list in list context and `0` in scalar context. No warning, no error.
- **Tied hashes.** Behaviour is delegated to the tie class's `FIRSTKEY` / `NEXTKEY` / `FETCH` methods. Order, aliasing, and iterator-reset semantics follow whatever the tie implementation provides - the pure-hash guarantees above do not automatically carry over.
- **Removed experiment: scalar argument.** Perl 5.14–5.22 experimentally accepted `values $ref`; that experiment was removed in 5.24. Dereference explicitly: `values %$ref, values @$ref`.

- **Array support requires 5.12+.** Running `values @a` on an older Perl raises a syntax error. Use `use v5.12;` at the top of a file that relies on array form to make the requirement explicit.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `keys` - the key-side companion; same ordering rules, same iterator, scalar context returns the count
- `each` - single-step iteration returning `($key, $value)` pairs; shares the iterator that `values` resets
- `exists` - test whether a key is present without triggering autovivification
- `delete` - remove a key/value pair; deleting the most-recently-returned key of `each` or `keys` leaves the remaining traversal order intact
- `map` - transform values into a derived list without mutating the hash (the non-aliasing alternative to `for (values %h)`)
- `sort` - pair with `keys` when you need a deterministic traversal order

Fixed-length data

vec

Read or write a fixed-width slot inside a string treated as a packed bit vector.

`vec` views `EXPR` as an array of unsigned integers, each `BITS` wide, packed back-to-back from the start of the string. `OFFSET` indexes into that array - not into bytes, not into bits, but into `BITS`-sized elements. Read form returns the integer at that slot; the `lvalue` form writes one. Widths are 1, 2, 4, 8, 16, 32, and on 64-bit builds 64.

Synopsis

```
my $n = vec($buf, $offset, $bits);
vec($buf, $offset, $bits) = $n;
```

What you get back

In `rvalue` context, an unsigned integer - the contents of the selected slot, zero-extended to a Perl number. In `lvalue` context, an assignable slot; assigning truncates the right-hand value to `BITS` and writes it into the string, extending the string with zero bytes if `OFFSET` lies past the current end.

The parentheses around `vec(...)` in the `lvalue` form are required - without them, `vec $buf, $o, $b = 3` parses the `=` as part of the argument list.

How the bits are laid out

The layout depends on BITS, and is chosen so code is portable across big- and little-endian machines:

- **BITS == 8**: each slot is one byte of the string. `vec($s, $i, 8)` is the unsigned value of `substr($s, $i, 1)`.
- **BITS == 16, 32, 64**: bytes of the string are grouped into chunks of BITS/8 and interpreted in **big-endian** order - equivalent to `unpack` with `n`, `N`, or (on 64-bit builds) `Q>` / the moral equivalent. `vec($s, 0, 32)` reads the first four bytes as a big-endian `uint32`.
- **BITS == 4, 2, 1**: the string is broken into bytes, and each byte is split into 8/ BITS groups, numbered **little-endian-ish** within the byte. The bit values from low to high are `0x01`, `0x02`, `0x04`, `0x08`, `0x10`, `0x20`, `0x40`, `0x80`. So for `chr(0x36)` (`0b00110110`):
 - BITS == 4 gives the two nibbles (`0x6`, `0x3`).
 - BITS == 2 gives the four 2-bit groups (`0x2`, `0x1`, `0x3`, `0x0`).
 - BITS == 1 gives the eight bits (`0`, `1`, `1`, `0`, `1`, `1`, `0`, `0`).

A slot entirely off the end of the string reads as `0`. Writing past the end grows the string with zero bytes to reach the slot. A negative OFFSET is a fatal error.

Global state it touches

None. `vec` operates purely on its arguments.

Examples

Read a byte at a given index:

```
my $s = "Perl";
print vec($s, 0, 8);           # 80    (== ord 'P')
print vec($s, 3, 8);           # 108   (== ord 'l')
```

Build a string by writing 32-bit big-endian words:

```
my $buf = '';
vec($buf, 0, 32) = 0x5065726C; # "Perl"
vec($buf, 1, 32) = 0x50657270; # "PerlPerp"
print $buf;                   # PerlPerp
```

Use `vec` as a compact boolean array - one bit per flag:

```
my $flags = '';
vec($flags, 17, 1) = 1;
vec($flags, 42, 1) = 1;
print vec($flags, 17, 1);      # 1
print vec($flags, 18, 1);      # 0  (slot never set, still zero)
print length $flags;           # 6  (string auto-extended to fit
↪ bit 42)
```

Count the set bits in a bit vector without looping bit by bit - the idiomatic pattern uses `unpack`:

```
my $ones = unpack("%32b*", $flags); # population count
```

Convert a bit vector into a string of 0s and 1s for display:

```
my $bits = unpack("b*", $flags); # "00...010...010..."
```

Combine two bit vectors with the bitwise string operators - those treat string operands as bit vectors of the same shape `vec` reads and writes:

```
my $union      = $flags_a | $flags_b;
my $intersection = $flags_a & $flags_b;
my $diff       = $flags_a ^ $flags_b;
```

Edge cases

- **Lvalue precedence:** `vec(EXPR, 0, B = N)` is a syntax error. Always write `vec(EXPR, 0, B) = N`.
- **Off-the-end read:** `vec($short, 1_000_000, 8)` returns 0, never dies, never warns.
- **Off-the-end write:** the string is zero-padded up to the slot. For `BITS == 1`, writing bit 94 grows the string to 12 bytes.
- **Negative OFFSET:** fatal - "Negative offset to vec in lvalue context" or the rvalue equivalent.
- **BITS not a supported power of two:** fatal with "Illegal number of bits in vec". Valid widths are 1, 2, 4, 8, 16, 32, and 64 on 64-bit builds.
- **UTF-8 encoded strings:** `vec` wants a byte string. If the scalar is flagged UTF-8, Perl first tries to downgrade it to a one-byte-per-character representation. If any character has a codepoint of 256 or higher, that fails fatally with "Wide character in vec". Call `utf8::downgrade` deliberately, or pack the data with `pack "C*"` first, before reaching for `vec`.
- **Read on undef:** under `use warnings`, triggers an uninitialized warning on the string argument; returns 0.
- **Assignment value wider than BITS:** the value is masked to the low BITS bits. `vec($s, 0, 4) = 0x1F` stores 0xF.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `pack` - build multi-field binary structures; `vec` is the random-access counterpart when every field has the same width

- `unpack` - pull fields out of a binary string; use `unpack("b*", $v)` or `unpack("%32b*", $v)` to render or popcount a vec bit vector
- `substr` - byte-level random access when BITS would be 8 and you also want the lvalue to grow or shrink the string
- `sprintf` - format the integer a vec read returns, e.g. `sprintf "%08b", vec($s, $i, 8)`
- `ord` - one-shot equivalent of `vec($s, $i, 8)` when you only need the byte value and never assign back

Processes

wait

Block until any child process exits and reap it.

`wait` is the simplest front end to the underlying `wait(2)` system call. It suspends the current process until one of its children terminates, collects that child's exit status into `$?` and `${^CHILD_ERROR_NATIVE}`, and returns the reaped child's PID. If there are no children to wait on, it returns -1 immediately. It is exactly equivalent to `waitpid(-1, 0)`.

Synopsis

```
my $pid = wait;
```

What you get back

The PID of the child that was reaped (a positive integer), or -1 if there are no unreaped children left to wait on. The child's status goes into `$?`; the raw, OS-specific status word goes into `${^CHILD_ERROR_NATIVE}`.

To decode `$?` after a successful wait:

```
my $exit  = $? >> 8;      # normal exit code (0..255)
my $signal = $? & 0x7f;   # termination signal, if any
my $core  = $? & 0x80;   # core-dump flag
```

A -1 return does **not** always mean “never had any children” - it can also mean “something already reaped them for me” (see below).

Global state it touches

- `$?` - set to the reaped child's 16-bit wait status (high byte: exit code, low byte: termination signal and core flag). Also readable as `$CHILD_ERROR` under `use English`.
- `${^CHILD_ERROR_NATIVE}` - the platform's raw status word, for callers that want to use POSIX macros (`WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG`) from `POSIX`.
- `%SIG` - if `$SIG{CHLD}` is set to 'IGNORE', or a handler in it calls `wait` itself, this call's result changes. See *Edge cases*.

Examples

Fork a worker, do something, then reap it:

```
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    # child
    exec "/usr/bin/true" or die "exec: $!";
}
my $reaped = wait;           # blocks until the child exits
die "unexpected pid" if $reaped != $pid;
printf "child %d exited %d\n", $reaped, $? >> 8;
```

Reap every outstanding child in a loop. `wait` returns -1 once the process has none left:

```
while ((my $kid = wait) > 0) {
    warn "reaped $kid, status " . ($? >> 8) . "\n";
}
```

Distinguish normal exit from signal termination:

```
my $pid = wait;
if ($pid > 0) {
    if ($? == 0)          { say "clean exit" }
    elsif ($? & 0x7f)     { say "killed by signal " . ($? & 0x7f) }
    else                  { say "exit " . ($? >> 8) }
}
```

Decode the native status word with `POSIX` macros, which is more portable than manual bit-shifting:

```
use POSIX ":sys_wait_h";
my $pid = wait;
if ($pid > 0 && WIFEXITED(${$^CHILD_ERROR_NATIVE})) {
    say "exit code: ", WEXITSTATUS(${$^CHILD_ERROR_NATIVE});
}
```

Non-blocking reap of every pending zombie - use `waitpid` with `WNOHANG`, not `wait`:

```
use POSIX ":sys_wait_h";
1 while waitpid(-1, WNOHANG) > 0;
```

Edge cases

- **No children to wait on:** returns -1 immediately. `wait` does not block in this case - it only blocks when at least one child is still alive and unreaped.
- **`$SIG{CHLD} = 'IGNORE'`:** on POSIX systems this asks the kernel to auto-reap children. Under this setting, `wait` almost always returns -1 because the children are gone before user code can call it. `?` is not set. If you want both fire-and-forget children **and** status reporting, use a real handler, not 'IGNORE'.

- **Custom `$SIG{CHLD}` handler:** if the handler calls `wait` (or `waitpid`) itself, a subsequent `wait` in the main flow will find nothing left and return `-1`. Decide up-front which code path does the reaping.
- **Interaction with `qx//` and `system`:** both of these internally fork, `exec`, and `wait` on their own child. A `$SIG{CHLD}` handler that blindly calls `wait` can accidentally reap that internal child out from under `qx//` or `system`, leaving them unable to collect the status and typically returning `-1` with `$_` set to `ECHILD`. Handlers should loop with `waitpid(-1, WNOHANG)` and only act on PIDs they recognise.
- **`$_` is only written on a successful reap.** After a `-1` return, `$_` retains whatever it had before the call. Don't inspect it unless `wait` returned a positive PID.
- **Interrupted by a signal:** if a signal is delivered while `wait` is blocked and the handler returns normally, Perl restarts the `wait` - you do not see `EINTR` at the Perl level. If the handler `dies` or `exits`, control leaves `wait` the usual way for that mechanism.
- **Threaded builds:** `wait` waits for children of the current process, not children of a specific thread. In a multi-threaded program the reaping thread is whichever one called `wait` first.
- **Return value is a PID, not a boolean:** `if (wait) { ... }` is true for both successful reaps and `-1`. Always compare to `0` or check `> 0` explicitly, as in the loop form above.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `waitpid` - wait for a specific child, or poll non-blockingly with `WNOHANG`; `wait` is the `waitpid(-1, 0)` form
- `fork` - create the children that `wait` reaps
- `exec` - what a forked child typically calls to become a different program
- `system` - `fork` + `exec` + `wait` in one call; uses its own internal `wait` and sets `$_` for you
- `kill` - send a signal to a child you no longer want to wait for
- `$_` - where the reaped child's status lands, encoded as exit-code high byte plus signal/core low byte

Processes

`waitpid`

Wait for a specific child process to terminate and reap it.

`waitpid` is the targeted counterpart of `wait`. Where `wait` reaps *any* child and blocks until one exits, `waitpid` lets you name which child (or which group of children) to wait for, and lets you poll instead of block by passing `WNOHANG` in `FLAGS`. The exit status of the reaped child lands in `$_`, identically to `wait`.

Synopsis

```
use POSIX ":sys_wait_h";

waitpid $pid, 0;           # block until $pid exits
waitpid $pid, WNOHANG;     # poll: is $pid done yet?
waitpid -1, WNOHANG;       # reap any ready zombie, don't
    ↪ block
```

What you get back

- The PID of the reaped child on success.
- 0 when WNOHANG is set and at least one matching child exists but none have terminated yet - i.e. “still running, try again later”.
- -1 when no child process matches PID (never existed, already reaped, or - on systems that auto-reap - was harvested by the kernel without Perl’s help).

The child’s exit status is written to `$?` and the raw platform status to `${^CHILD_ERROR_NATIVE}` on every successful reap. Decode `$?` the same way you would after `wait` or a backtick call:

```
my $reaped = waitpid $pid, 0;
if ($reaped > 0) {
    my $exit = $? >> 8;
    my $signal = $? & 127;
    my $core = $? & 128;
}
```

What PID means

PID selects which children `waitpid` is willing to reap:

- **Positive PID** - wait for that specific child. -1 on return means no such child (you already reaped it, or it was never yours).
- **0** - wait for any child in the *current* process group. Useful when your program has joined a process group with other cooperating processes and you only want to harvest your own.
- **-1** - wait for any child, same scope as `wait`.
- **Less than -1** - wait for any child whose process group ID equals the absolute value of PID. `waitpid -$pgid, 0` waits for any member of process group `$pgid`.

FLAGS is a bitmask. The portable flag is WNOHANG, which makes the call non-blocking: if no matching child has exited, `waitpid` returns 0 immediately instead of sleeping. FLAGS of 0 means “block until something is reappable”, and is implemented on every platform Perl runs on - even those without a native `waitpid(2)` syscall, where Perl emulates it by remembering exit statuses of children it has already seen.

Global state it touches

- `$?` - set to the exit status of the reaped child (shifted: high byte = exit code, low 7 bits = signal, bit `0x80` = core-dumped flag).
- `${^CHILD_ERROR_NATIVE}` - set to the raw platform status word, before Perl normalises it into `$?`.

Neither variable is touched when `waitpid` returns `0` (`WNOHANG` and nothing ready) or `-1` (no matching child).

Examples

Block until a known child finishes, then inspect its exit code:

```
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    exec "/usr/bin/gzip", "big.log" or die "exec: $!";
}
waitpid $pid, 0;
die "gzip failed with status ", $? >> 8 if $?;
```

Poll without blocking - useful inside an event loop or a `SIGCHLD` handler:

```
use POSIX ":sys_wait_h";

sub harvest_children {
    while ((my $kid = waitpid -1, WNOHANG) > 0) {
        warn "child $kid exited with ", $? >> 8, "\n";
    }
}
```

The classic “reap every pending zombie, don’t block” idiom:

```
use POSIX ":sys_wait_h";
1 while waitpid(-1, WNOHANG) > 0;
```

Wait only for children in a specific process group. `setpgid` first, then target the group by negating its ID:

```
my $leader = fork // die "fork: $!";
if ($leader == 0) {
    setpgid 0, 0;          # become process-group leader
    exec @worker_cmd or die "exec: $!";
}
setpgid $leader, $leader; # parent matches, races with child

# ... later, reap any member of that group:
while ((my $kid = waitpid -$leader, WNOHANG) > 0) {
    log_exit($kid, $?);
}
```

Distinguish the three return values in one place:

```
my $r = waitpid $pid, WNOHANG;
if ($r == $pid) { finalize_child($pid, $?) }
elsif ($r == 0) { still_running($pid) }
elsif ($r == -1) { warn "no such child $pid (already reaped?)" }
```

Edge cases

- **WNOHANG requires use POSIX.** The constant is not a Perl built-in; it comes from the `POSIX` module, typically via `use POSIX ":sys_wait_h";`. Writing a bare integer (`waitpid $pid, 1`) works on Linux but is not portable and hides intent.
- **`$SIG{CHLD} = 'IGNORE'` races with `waitpid`.** When you tell the kernel to auto-reap children, there is nothing left for `waitpid` to find - it returns -1 with `$!` set to `ECHILD`. Choose one reaping strategy per program, not both.
- **Already-reaped PIDs** return -1. A single child's PID can only be reaped once; after that, it is forgotten. If you hand the same PID to `waitpid` twice, the second call returns -1.
- **Negative PID is a process group, not “any but this one”.** `waitpid -1234, 0` waits for members of process group 1234, not “every child except PID 1234”. Readers unfamiliar with the POSIX convention often misread this.
- **WNOHANG with PID = -1** is the standard “drain the zombie queue” form. Combined with a while loop it reaps every child that has already exited and stops as soon as the kernel has nothing more to hand back.
- **Return value of -1 can also mean auto-reap.** On systems configured to reap children automatically (e.g. a `$SIG{CHLD} = 'IGNORE'` handler installed at startup), `waitpid` sees no child and returns -1 even for a PID you just forked. See `perlipc` for the platform notes.
- **Blocking wait is always available.** Even on platforms without a native `waitpid(2)` or `wait4(2)` syscall, `waitpid PID, 0` works - Perl emulates it by retaining the statuses of children it has already collected.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `wait` - block for *any* child; equivalent to `waitpid -1, 0` on systems that support it
- `fork` - create the children you will later reap with `waitpid`
- `kill` - send a signal to the same PIDs and process groups `waitpid` accepts
- `$?` - exit status of the reaped child; decode as `$? >> 8` for the exit code, `$? & 127` for the signal
- `$_{^CHILD_ERROR_NATIVE}` - raw platform status word for the same child
- `POSIX` - source of `WNOHANG` and the rest of `:sys_wait_h`

Control flow

wantarray

Report the calling context of the currently executing subroutine.

wantarray lets a subroutine ask the runtime *how* its caller wrote the call: was the result assigned to a list, to a scalar, or thrown away? The answer is one of three distinct values, and a sub can use it to return different shapes of data, or to skip expensive work altogether when the caller asked for nothing.

Synopsis

```
wantarray
```

Takes no arguments and has no parentheses form - it is a keyword, not a function call.

What you get back

wantarray returns one of exactly three values, corresponding to the three contexts Perl distinguishes:

Return value	Context of the caller	Typical caller syntax
true (1)	list	<code>my @x = f(); (f())[0]</code>
false ("")	scalar	<code>my \$x = f(); if (f()) {...}</code>
undef	void	<code>f();</code> as a statement

The three-way distinction is why the function “should have been named wantlist()”: the boolean-looking return is actually tri-state, and defined wantarray is the right test for “did the caller want anything at all”.

The canonical idiom

The pattern wantarray exists for is a sub that adapts its return shape to the caller:

```
sub records {
    my @rows = heavy_query();
    return wantarray ? @rows : "@rows";
}

my @r = records();           # gets the list
my $s = records();           # gets a space-joined string
records();                   # still runs heavy_query - see below
```

Pair that with an early-out when the caller asked for nothing:

```
sub records {
    return unless defined wantarray;    # caller wrote `records();`
    my @rows = heavy_query();           # skipped in void context
```

(continues on next page)

(continued from previous page)

```

    return wantarray ? @rows : "@rows";
}

```

defined wantarray is the void-context guard. wantarray alone is not - it is false in both scalar and void, and an early `return` on a false wantarray would break every scalar-context caller.

Context is propagated, not re-interpreted

wantarray reports the context the caller *imposed* on the sub, not what the sub does internally:

```

sub ctx { wantarray ? "list" : defined wantarray ? "scalar" : "void"
    ↪ }

my @a = ctx();      # "list"
my $s = ctx();      # "scalar"
ctx();              # "void"
print ctx(), "\n";  # "list" - print's LIST is list context
scalar ctx();       # "scalar" - scalar() forces scalar context

```

Note the last two: `print` imposes list context on its arguments, and `scalar` explicitly forces scalar context. Neither is about what `ctx` *does* with the result.

Calls from void context

A sub called as a bare statement runs in void context, and wantarray returns `undef` *inside that sub*. This does not cascade: a sub called from within a void-context sub sees whatever context *its own* caller imposed.

```

sub inner { defined wantarray ? "wanted" : "void" }
sub outer { my $x = inner(); return $x }

outer();          # outer is void; inner is scalar → "wanted"

```

Each call site establishes its own context independently. wantarray always reports the nearest enclosing sub's context, not the dynamic chain's.

Works inside eval too

wantarray also reports the context of an `eval` block or `eval EXPR`, not just named sub-routines:

```

my @x = eval { wantarray ? (1, 2, 3) : "scalar" }; # (1,2,3)
my $x = eval { wantarray ? (1, 2, 3) : "scalar" }; # "scalar"
eval { defined wantarray ? 1 : 0 };                # void → 0

```

This is the same mechanism - `eval` is a call frame in the same sense a sub is.

Where wantarray has no useful answer

wantarray's result is **unspecified** at these places, and code should not rely on any particular value:

- The top level of a file (outside any sub).
- Inside a BEGIN, UNITCHECK, CHECK, INIT, or END block.
- Inside a DESTROY method.

Treat wantarray as meaningful only within an ordinary sub or `eval`. Anywhere else, design your code so the answer does not matter.

Examples

A getter that returns the full list to list callers, the count to scalar callers, and does nothing in void context:

```
sub warnings {
    return unless defined wantarray;
    my @w = collect_warnings();
    return wantarray ? @w : scalar @w;
}

my @all    = warnings();    # every warning
my $count  = warnings();    # just the number
warnings();                 # collect_warnings() is not called
```

A sub that refuses to be called in void context because the return value is the whole point:

```
sub must_use {
    defined wantarray
        or croak "must_use: return value must be used";
    return compute();
}
```

A sub that emits one element per call in scalar context and the whole batch in list context - a shape some iterator-style APIs expose:

```
sub next_batch {
    state @queue;
    @queue = refill() unless @queue;
    return wantarray ? splice(@queue) : shift @queue;
}
```

Edge cases

- **No parentheses, no arguments.** `wantarray()` parses but the empty parens are noise; the keyword takes nothing. `wantarray $x` is a syntax error, not a call with an argument.

- **Not the caller's context for operators.** `wantarray` reports the call-frame context only. An expression like `$sub->() + 1` puts the call in scalar context; `wantarray` inside `$sub` sees scalar, as expected. There is no way to ask “was I called inside an arithmetic expression” - only list / scalar / void.
- **return honours the same context.** `return @list` in scalar context yields the last element, not the count - because `return` itself sees the caller's context. Use `return wantarray ? @list : scalar @list` when you want the count for scalar callers.
- **Boolean context is scalar context.** `if (f())` calls `f` in scalar context; `wantarray` returns false there, not `undef`.
- **List assignment to an empty list is still list context.** `() = f();` is list context even though the result is discarded. `wantarray` returns true. This is useful for forcing list context on a sub whose side effects depend on it.
- **wantarray is not available in XS callers the same way.** An XSUB inspects context through `GIMME_V`; a Perl sub called *from* an XSUB sees the context the XSUB established for the call.
- **Prototypes do not change wantarray.** A `( $ )` prototype forces an argument's context, not the sub's own call context.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `return` - honours the same list/scalar/void context `wantarray` reports; the two are designed to be used together
- `caller` - inspect the call stack itself (package, file, line, and, in its three-arg form, more context details including whether `wantarray` would return true there)
- `scalar` - force scalar context on an expression, which is what makes `wantarray` return false in the callee
- `eval` - establishes a call frame whose context `wantarray` also reports

I/O

warn

Emit a warning to `STDERR`.

`warn` stringifies `LIST` and prints the result to `STDERR` - the same formatting machinery as `die`, minus the unwind. Control returns to the caller; nothing is thrown, nothing is caught. Reach for it when the program should keep running but the user, the operator, or the log deserves to hear about something unexpected.

Synopsis

```
warn LIST
warn $message
warn                                     # re-surface current $@ as a warning
```

What you get back

warn returns 1 under normal circumstances. The return value is almost never useful - the point of the call is the side effect on STDERR (or on whatever \$SIG{__WARN__} does with the message).

Unlike `die`, warn does **not** unwind. Execution continues with the statement after the call.

Global state it touches

- `$@` - read by the no-argument form. If `$@` holds a non-empty value, warn re-surfaces it (see *Propagation* below). warn does **not** write `$@`.
- `$,`, `$\` - **not** consulted. warn is not `print`; the output separators and record separator play no role. The message is written as a single string.
- `$.` - the current input line number, appended to messages that lack a trailing newline.
- `$SIG{__WARN__}` - if set, the handler runs in place of writing to STDERR. See *The \$SIG{__WARN__} hook* below.
- STDERR - the default destination. Any redirection of STDERR (shell-level `2>`, open STDERR, `. . .`, a PerlIO layer) affects where warn output lands.

The trailing-newline rule

The same rule as `die` applies, and for the same reason:

- **Message ends in `"\n"`** - used verbatim. Nothing is appended. Use this for warnings that are complete user-facing diagnostics.
- **Message does not end in `"\n"`** - Perl appends " at FILE line N", and if a file is being read, ", <HANDLE> line M", then a final "." and newline. Use this when pinpointing the call site is more useful than a tidy message.

```
warn "cache is stale\n";           # "cache is stale\n"
warn "cache is stale";             # "cache is stale at main.pl_
↪line 42.\n"
```

Reference-valued warnings never get location information appended. Stringifying an object to add file and line would defeat the point of passing one.

Propagation: warn with no argument (or empty string)

When LIST is empty or stringifies to "", warn does not emit a fresh message - it re-surfaces whatever is in \$@:

- If \$@ is **non-empty**, warn appends "\t...caught" to it and emits the result. This is the canonical way to notice that an exception was caught without letting it die silently:

```
eval { risky() };
warn if $@;                                # "<whatever died>\n\t...caught_"
    at ..."
```

- If \$@ is also empty, the string "Warning: Something's wrong" is used.

This is deliberately asymmetric with **die**: **die** with no argument **re-throws** (triggering the propagation machinery and the PROPAGATE hook for objects); warn with no argument **reports** what is already in \$@.

The \$SIG{__WARN__} hook

Setting \$SIG{__WARN__} installs a handler that runs **instead of** the default write to STDERR. The handler receives the exception derived from LIST as its single argument and is responsible for disposing of it - logging it, rewriting it, escalating it to **die**, or dropping it on the floor:

```
local $SIG{__WARN__} = sub {
    my ($msg) = @_;
    return if $msg =~ /^Use of uninitialized/; # silently drop
    log_warning($msg);
};
```

Three asymmetries with \$SIG{__DIE__} are worth internalising:

- **The default message is suppressed.** \$SIG{__DIE__} runs *alongside* the exception machinery; \$SIG{__WARN__} runs *instead of* the default STDERR write. If the handler does nothing, the warning is silently dropped.
- **To pass the warning through, call warn again.** The handler is not re-entered for its own warn call, so there is no infinite loop:

```
local $SIG{__WARN__} = sub {
    log_warning($_[0]);
    warn $_[0];                                # re-emit to STDERR as well
};
```

- **A __WARN__ handler can silence *mandatory* warnings** - warnings that no warnings cannot turn off. This makes \$SIG{__WARN__} a sledgehammer; prefer no warnings 'category' for targeted suppression and reserve the handler for routing or escalation.

```
# wipe out *all* compile-time warnings, then switch on at runtime
BEGIN { $SIG{'__WARN__'} = sub { warn $_[0] if $DOWARN } }
```

(continues on next page)

(continued from previous page)

```
my $foo = 10;
my $foo = 20;           # duplicate-my silenced
$DOWARN = 1;
warn "\$foo is alive and $foo";  # now shows up
```

warn the built-in vs. use warnings

These are different things. Do not confuse them:

- **warn** is a built-in function that emits a single message right now. You call it.
- **use warnings** is a pragma that enables categories of compile-time and run-time diagnostics (uninitialized values, deprecated features, unopened filehandles, and so on). The compiler and interpreter emit those diagnostics on your behalf, using the same machinery **warn** uses.

The pragma's diagnostics flow through `$SIG{__WARN__}` exactly like explicit **warn** calls. `no warnings '<category>'` suppresses the pragma's diagnostics in its lexical scope but has no effect on an explicit **warn** you wrote yourself - that call always fires.

Examples

Report a recoverable problem and keep going:

```
open my $fh, "<", $path
  or warn "skipping $path: $!\n" and next;
```

Development-mode diagnostic with location appended:

```
warn "unexpected record shape";
# unexpected record shape at parser.pl line 87.
```

Noticing a caught exception without suppressing it:

```
eval { load_config() };
warn if $@;           # surfaces "...caught at ..."
```

Route every warning through a logger, still printing to STDERR:

```
local $SIG{__WARN__} = sub {
  my ($msg) = @_;
  $logger->warn($msg);
  print STDERR $msg;           # preserve default visibility
};
```

Escalate a specific warning to a fatal error:

```
local $SIG{__WARN__} = sub {
  die @_ if $_[0] =~ /corrupt database/;
  warn @_;           # everything else: default path
};
```

Pass a blessed object for structured warnings (the receiver of a `$SIG{__WARN__}` hook gets the reference, not a stringified form):

```
local $SIG{__WARN__} = sub {
    my ($w) = @_;
    if (ref($w) && $w->isa('MyApp::Warning')) {
        $logger->record($w);
    }
    else {
        print STDERR $w;
    }
};
warn MyApp::Warning->new(code => 'SLOW_IO', detail => $path);
```

Edge cases

- **List with two or more items** is stringified and concatenated: `warn "bad record", $n, ": ", $raw`. The newline rule applies to the concatenated result.
- **Empty list** (`warn`; or `warn ""`;) triggers the propagation path, not a fresh message.
- **Reference-valued warning inside a `$SIG{__WARN__}` handler**: the handler receives the reference unchanged. If no handler is installed, the reference is stringified to STDERR via its `""` overload (or Perl's default `MyClass=HASH(0x...)`).
- **warn inside `$SIG{__WARN__}`**: does **not** re-enter the handler. The call writes to STDERR as if no handler were installed. This is what makes the “pass through” idiom safe.
- **warn during global destruction**: still works, but STDERR may already be closed in pathological cases. Warnings from DESTROY handlers running at interpreter exit can vanish.
- **STDERR closed or redirected**: `warn` writes to whatever STDERR currently points at. If STDERR is closed, the write fails silently; no error is raised, because `warn` itself does not check.
- **No relation to `$\ / $,`**: adding `local $\ = "\n"` does not make `warn "x"` print a newline. The trailing-newline rule is the only newline-adding mechanism for `warn`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `die` - same formatting rules (trailing-newline suppresses location, references pass through unchanged), but raises an exception instead of reporting and continuing
- `eval` - pairs with the no-argument form of `warn` for the “catch, inspect, re-surface” idiom
- `$@` - the source the no-argument `warn` reports from

- `$SIG{__WARN__}` - the hook that replaces the default STDERR write; use it for routing, filtering, or escalation
- `Carp` - `carp`, `cluck`, `croak`, `confess`: warnings and exceptions that report from the caller's perspective rather than the call site, which is almost always what a library wants

I/O

write

Render one record to a filehandle through its associated `format`.

`write` is the executor half of Perl's report-generation mechanism: it takes the picture declared by a `format`, plugs the values from the matching argument lines into the fields, and emits the resulting lines to FILEHANDLE. It also tracks the page: when not enough lines remain, it writes a form feed, runs the top-of-page format, and then writes the record. `write` is the **only** way a declared format produces output; a format is inert until `write` references it.

Do not confuse this with `syswrite` or the inverse of `read`: `write` is strictly the picture-record executor. It has nothing to do with raw byte output.

Synopsis

```
write FILEHANDLE
write EXPR
write
```

What you get back

1 on success, `undef` on failure (with `$!` set). A failure here is typically either a closed handle or an I/O error on the underlying write - not "format not found," which croaks rather than returning `undef`.

How write picks the format

Every filehandle has **two** format slots: the record format and the top-of-page format. By default:

- The **record format** is the format whose name matches the filehandle (format `STDOUT` for `STDOUT`, format `REPORT` for a handle opened as `REPORT`).
- The **top-of-page format** is the filehandle name with `_TOP` appended (`STDOUT_TOP`, `REPORT_TOP`), falling back to `top` in the current package.

Both are overridable per-handle through `$~` (current format name) and `$^` (current top-of-page format name). These variables act on the **currently selected** handle, so changing them for anything other than `STDOUT` requires a `select` dance:

```
my $ofh = select REPORT;
$~ = "My_Format";
```

(continues on next page)

(continued from previous page)

```
$^ = "My_Top";  
select $ofh;
```

If FILEHANDLE is omitted, write targets the currently selected handle (default STDOUT). If the argument is an EXPR rather than a bareword, the expression is evaluated and its string value is looked up as the filehandle name at runtime.

Global state it touches

write reads and updates a cluster of per-handle format variables. All are per-filehandle, reached via `select`:

- `$~` - name of the record format for the selected handle. Read at every write call.
- `$^` - name of the top-of-page format. Consulted when a new page begins.
- `$=` - page length in lines. Default 60.
- `$-` - lines remaining on the current page. Decrement by every write. Set to 0 to force a page break on the next call.
- `$%` - current page number. Incremented on each top-of-page.
- `$$L` - string emitted before every top-of-page except the first. Default "\f" (form feed).
- `$:` - characters on which fill-mode (^) fields may break.
- `$$A` - the format accumulator; the buffer `formline` writes into and write flushes. Rarely touched directly unless you are bypassing write with `formline`.

write also writes through the PerlIO stack of the target handle, so `$|` (autoflush on the selected handle) still applies.

Top-of-form processing

Before emitting the record, write checks whether the record fits in the lines remaining on the page (`$-`). If it does not:

1. Emit `$$L` (form feed by default) - skipped on the very first page.
2. Run the top-of-page format named by `$^` to produce the page header.
3. Reset `$-` to `$=` and increment `$%`.
4. Emit the record.

On the very first write to a handle, step 1 is skipped so the output does not begin with a form feed.

Examples

Basic report with a header:


```
format STDOUT_TOP =  
                Passwd File  
Name            Login      Office    Uid       Gid Home  
-----  
.   
format STDOUT =  
@<<<<<<<<<<<<<<< @||||| @<<<<<@>>> @>>> @<<<<<<<<<<<<<<  
$name,           $login,   $office,$uid,$gid, $home  
.  
  
while (my @row = getpwent) {  
    ($name, $login, $uid, $gid, $office, $home) = @row[0,0,2,3,5,7];  
    write;  
}
```

Write to a named filehandle. The format name defaults to the handle name; no `select` is needed if the default is acceptable:

```
open my $fh, ">", "report.txt" or die $!;
format REPORT =
    @<<<<<<<<<<<<<<<<<<      @>>>>>>>>>
        $item,                      $count
.
# rebind the glob so REPORT refers to $fh
*REPORT = *$fh;
for my $r (@records) {
    ($item, $count) = @$r;
    write REPORT;
}
close $fh;
```

Switch format at runtime by setting `$~` on the selected handle. `local` confines the change to the current scope:

```
sub print_summary {
    local $~ = "SUMMARY_FORMAT";
    write;
}
```

Force a page break by zeroing \$- before the next call:

```
$- = 0; # next write starts a new page
write;
```

write EXPR picks the handle at runtime from a string - useful when the target is data-driven:

```
my $handle_name = $verbose ? "LOG" : "STDOUT";
write $handle name;
```

Interleaving `print` with `write`: legal, but `$-` only tracks `write` output. If a `print` burst consumes page rows you care about, decrement `$-` by hand:

```
print "extra note: $msg\n";
$- -= 1;                # keep the page counter honest
write;
```

Edge cases

- **Format not found:** if the format named by `$~` does not exist, write croaks with Undefined format "NAME" called. This is a fatal error, not a return-undef-with-`$!` failure.
- **Closed filehandle:** returns `undef` and sets `$!`. Under use warnings a `write()` on closed filehandle warning is emitted.
- **Record taller than page:** if the record itself has more lines than `$=`, write still emits all of its lines on one page - it does not split a single record across a page boundary. The oversize is silently tolerated.
- **_TOP fallback:** if no `<HANDLE>_TOP` format exists, write falls back to the format named `top` in the current package. This is rarely what you want for autovivified handles; set `$^` explicitly when it matters.
- **Lexical variables in formats:** my variables declared outside a format are **not visible** inside it unless the format is declared within the lexical scope of those variables. This bites anyone who wraps `write` inside a subroutine and wonders why the report is blank - move the format declaration inside the sub, or use package globals for the argument-line expressions.
- **Top-of-page triggers mid-record, not mid-page:** write checks page room once per record. A record that starts near the bottom of the page will either fit in full or trigger top-of-page before being written; it never straddles.
- **LC_NUMERIC and numeric fields:** if use locale is in effect when the format is declared, the decimal point in numeric fields follows the locale at declaration time - not at write time. Switching locales after the format is compiled does not change the separator.
- **Control characters in text fields** are replaced with spaces so they cannot scramble column alignment. The sole exception is `\r` in fill-mode fields, where it forces a line break.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `format` - declare the picture that write renders; the two are meaningless without each other
- `formline` - run one picture line against an argument list into `$^A` without any page tracking; the low-level primitive behind `write`
- `select` - change which handle write targets when called without an argument, and the prerequisite for setting any of the `$~`, `$^`, `$=`, `$-` per-handle

- `print` - free-form output when columnar reporting is overkill; may be interleaved with `write` on the same handle
- `$~` - override the record format name for the selected handle
- `$-` - lines left on the current page; set to 0 to force a new page before the next `write`
- `perlform.pod` - full picture-line grammar, fill-mode rules, and worked report examples

SCALARs and strings

`y///`

Transliterate characters in a string. `y///` is a synonym for `tr///` - the two are identical in every respect.

The `y` spelling is provided for **sed** devotees, whose muscle memory produces `y/abc/ABC/` rather than `tr/abc/ABC/`. Perl's parser accepts either; the resulting op is the same. Use whichever spelling reads better in the surrounding code - if the file already quotes a `sed` script, `y///` often fits; in fresh Perl, `tr///` is more common.

Synopsis

```
y/SEARCHLIST/REPLACEMENTLIST/cdsr
$var =~ y/SEARCHLIST/REPLACEMENTLIST/cdsr
y/SEARCHLIST/REPLACEMENTLIST/      # transliterates $_
```

What you get back

Same as `tr///`:

- Without `/r`: the number of characters matched (and replaced or deleted), after modifying the target in place.
- With `/r`: a new string with the transliteration applied; the original is untouched.

Examples

Classic `sed`-style uppercase transliteration on `$_`:

```
while (<>) {
    y/a-z/A-Z/;
    print;
}
```

Counting vowels without modifying the string, using `/r` to discard the transliterated copy:

```
my $vowels = ($text =~ y/aeiouAEIOU//) ;    # count in place is fine.
↪too
my $count  = () = $text =~ /[aeiouAEIOU]/g;
```

Strip everything except digits from a string (non-destructive copy via /r, complement + delete):

```
my $digits = $phone =~ y/0-9//cdr;      # "(415) 555-1212" →  
→ "4155551212"
```

Rot13 - the textbook y/// one-liner:

```
(my $coded = $plain) =~ y/A-Za-z/N-ZA-Mn-za-m/;
```

Edge cases

See `tr///` for the full list. Two worth repeating here:

- **No variable interpolation.** `y/$x/$y/` transliterates the literal characters `$`, `x` against `$`, `y` - not the contents of those variables. This matches `tr///` and is the single most common sed-to-Perl translation bug.
- **`y'...'...` disables ranges.** With single-quote delimiters, `-` is literal: `y'a-z'A-Z'` transliterates only `a`, `-`, `z` against `A`, `-`, `Z`.

Differences from upstream

Fully compatible with upstream Perl 5.42.

See also

- `tr///` - the canonical spelling; identical semantics, same flags (`/c`, `/d`, `/s`, `/r`), same edge cases
- `s///` - pattern-based substitution when character-by-character mapping is not enough
- `lc`, `uc` - locale- and Unicode-aware case change; prefer these over `y/a-z/A-Z/` for anything beyond ASCII
- `$_` - the default target when no `=~` binding is given

5.1.3 Operators

Perl's operator inventory is large but the structure is regular. Operators group into a handful of families - arithmetic, comparison, logical, bitwise, regex binding, assignment - and a few one-off mechanisms (range, ternary, comma, arrow, subscripts). This reference is split into one page per family.

PetaPerl implements the full Perl 5 operator set with identical semantics: same precedence, same associativity, same context rules. Differences are localised to the *PetaPerl-specific notes* at the bottom of each page where they apply.

Choose a family

Arithmetic operators

The numeric four-function-calculator set, plus modulus and exponentiation, plus the unary sign operators.

Operator	Operation	Arity	Associativity
**	exponentiation	binary	right
*	multiplication	binary	left
/	division	binary	left
%	modulus	binary	left
+	addition	binary	left
-	subtraction	binary	left
+	unary plus	unary	right
-	unary negation	unary	right

All arithmetic operators evaluate their operands in numeric context

- strings are coerced via the rule “leading numeric prefix wins, the rest is ignored (with an Argument “...” isn't numeric warning under use warnings)”.

```

my $x = 5 + 3;      # 8
my $y = 10 / 3;     # 3.33333333333333...
my $z = 2 ** 10;    # 1024
my $rem = 17 % 5;   # 2
my $neg = -$x;      # -8

```

Precedence and associativity

****** binds tightest and is *right*-associative. *****, **/**, **%** form one group; **+**, **-** form another. Unary **+** and **-** bind between ****** and ***** (high precedence).

```

2 ** 3 ** 2      # 2 ** 9      = 512    (right-assoc: top-down)
-2 ** 2          # -(2 ** 2)   = -4      (** binds tighter than unary -)
(-2) ** 2        # 4           (parens override)
6 / 2 / 3        # (6 / 2) / 3 = 1      (left-assoc on /)

```

The `-2 ** 2` case bites everyone exactly once. Parenthesise the sign when the base might be negative.

Division

/ always returns a floating-point result, even for integer operands that divide exactly:

```

10 / 5          # 2      -- numerically equal, stored as int or
↳ float
                  #      depending on the runtime; treat as
↳ numeric

```

(continues on next page)

(continued from previous page)

```
10 / 4      # 2.5
10 / 3      # 3.3333333333333335
```

For integer division, use `int`:

```
int(10 / 4)  # 2
int(-7 / 2)  # -3    (truncation toward zero, not floor)
```

Division by zero is a fatal error (not a warning, not an infinity-producing operation):

```
my $bad = 1 / 0;    # dies: Illegal division by zero
```

Modulus

`%` returns the integer remainder. Both operands are converted to integers first; the sign of the result follows the right operand (Perl-flavoured “sign of divisor”, same as Python, unlike C).

```
17 % 5      # 2
-17 % 5     # 3      -- result follows sign of $b ( 5)
17 % -5     # -3     -- result follows sign of $b (-5)
-17 % -5    # -2
```

For floating-point remainder, use POSIX: `:fmod` from *POSIX*.

Exponentiation

`**` accepts any pair of numbers including non-integer exponents:

```
2 ** 10      # 1024
2 ** 0.5     # 1.4142135623730951    -- square root
$x ** -1     # 1/$x                 -- reciprocal
```

For `sqrt`, the dedicated *sqrt* function is clearer (and named-unary, so it parses with one argument greedily - see *precedence*).

Unary + and unary -

Unary - negates. Unary + is a no-op on numbers; its only practical use is to *force* numeric context, or to disambiguate a syntactic edge case where Perl would otherwise treat the next token as a function name:

```
sub foo { 42 }
print foo (1+2)*3;    # prints 42*3 ... wait, no:
                      # parses as: print( foo(1+2) ) * 3
                      # because foo(...) sticks to the parens.
print foo +(1+2)*3;    # parses as: print( foo + ((1+2)*3) )
                      # the unary + signals "not a function call"
```

You will rarely need this. It is documented because it appears in older code.

Compound assignment

Every binary arithmetic operator has a compound assignment form (+=, -=, *=, /=, %=, **=). See [assignment](#).

See also

- [String operators](#) - . and x, the string equivalents of + and *.
- [Numeric comparison](#) - < <= == > >= != <=>.
- [Precedence](#) - full table.
- [abs](#), [sqrt](#), [int](#) - perlfunc forms of related operations.

String operators

Two operators in this family - concatenation and repetition. They play the same role for strings that + and * play for numbers.

Operator	Operation	Arity	Associativity	Context
.	concatenation	binary	left	string
x	repetition	binary	left	mixed

Both coerce their operands to string context (with the usual numeric-to-string conversion: 42 becomes "42", 1.5 becomes "1.5", etc.).

. - concatenation

Joins the two operands. The result is the concatenation of the left and right strings:

```
"Hello" . " " . "World"      # "Hello World"
"x = " . 42                  # "x = 42"                (numeric coercion)
"err: " . $!                 # "err: <strerror>"
```

Compound .= appends in place:

```
my $s = "a";
$s .= "b";      # "ab"
$s .= "c";      # "abc"
```

.= is faster than \$s = \$s . "... " for repeated appending - not by an asymptotic margin in modern Perl, but enough to matter in tight loops on long strings. pperl's runtime mutates the internal buffer in place when the string's reference count is 1 (see [PetaPerl-specific notes](#) below).

For large numbers of pieces, join is generally clearer than a chain of .:

```
join(", ", @items)
"hello, world, !"      # hand-built; readable for small
```

(continues on next page)

(continued from previous page)

```

↪arity

join(", ", $a, $b, $c, $d)    # cleaner than $a.", ".$b.", ".$c.",
↪".$d

```

x - repetition

Repeats its left operand a number of times given by the right operand. Behaviour depends on the *context* of the left operand:

- **Scalar context** on the left → string repetition.
- **List context** on the left (parens around the left operand) → list repetition.

```

"ab" x 3           # "ababab"           (string)
"- " x 40          # 40-character ruler

(0) x 5            # (0, 0, 0, 0, 0)     (list)
(1, 2) x 3         # (1, 2, 1, 2, 1, 2)

```

The parens-on-the-left distinction trips beginners. Without parens, the left side is in scalar context regardless of how it looks:

```

1, 2 x 3           # (1, "222")           -- Whoops.
                    # x binds tighter than , so this is
                    # (1, (2 x 3)) - and 2 in scalar context
                    # stringifies, so 2 x 3 is "222".

(1, 2) x 3         # (1, 2, 1, 2, 1, 2)   -- list repetition

```

Compound x= replicates in place:

```

my $line = "=";
$line x= 40;      # "====================================="

```

A non-positive count returns the empty string (or empty list for list context), without warning:

```

"abc" x 0          # ""
"abc" x -3         # ""
("a", "b") x 0     # ()

```

A non-integer count is truncated toward zero:

```

"x" x 3.7          # "xxx"             (3.7 truncates to 3)
"x" x -0.5         # ""

```


Numeric vs string operators

Mixing numeric and string operators on the same operand produces predictable but sometimes surprising results, because each operator imposes its own context:

```
"5" + "3"           # 8           (+ wants numeric → both coerce to
↳numbers)
5 . 3               # "53"        (. wants string → both coerce to
↳strings)
"5" * "3"           # 15
5 x 3               # "555"       (x wants string left, number right)
```

There is no operator that “guesses” - the operator picks the context, full stop.

See also

- *Arithmetic operators* - + and * are the numeric analogues of . and x.
- *String comparison* - comparing strings rather than building them.
- `join`, `sprintf`, `substr` - perlfunc forms of common string-building operations.

Numeric comparison operators

Seven operators that compare two numbers and return either a boolean (the first six) or a three-way result (<=>).

Operator	Question	Returns
<	less than	true / false
<=	less or equal	true / false
==	equal	true / false
>=	greater or equal	true / false
>	greater than	true / false
!=	not equal	true / false
<=>	three-way (sort)	-1, 0, or +1

All seven coerce both operands to numbers before comparing.

```
$age    >= 18           # true if 18 or older
$count  == 0           # true if numerically zero
$x      != $y          # true if they differ numerically
$a      <=> $b          # -1 if $a < $b, 0 if equal, +1 if $a
↳> $b
```

== is for numbers, eq is for strings

This is the single most common bug new Perl programmers ship. == compares **numerically** - both operands are run through numeric coercion, and any non-numeric string becomes 0:

```
"foo" == "bar"      # TRUE  -- both coerce to 0
"42x" == "42y"      # TRUE  -- both coerce to 42
"1e2" == 100        # TRUE  -- "1e2" is 100 in scientific notation
```

For string equality, use `eq`:

```
"foo" eq "bar"      # FALSE  -- the actual question
"42x" eq "42y"      # FALSE
```

If `use warnings` is in scope, the Argument `"..."` isn't numeric warning fires every time `==` (or any other numeric operator) sees a non-numeric string. That warning is your friend.

`<=>` - the spaceship operator

`<=>` returns -1 if the left is smaller, 0 if equal, +1 if greater. It is *the* sort comparator:

```
my @sorted = sort { $a <=> $b } @numbers;      # ascending
my @rev    = sort { $b <=> $a } @numbers;      # descending

# multi-key: by length, ties broken by content
my @lines = sort { length($a) <=> length($b) || $a cmp $b } @raw;
```

The pattern `cmp1 || cmp2 || cmp3` works because `<=>` returns 0 on ties and `||` falls through on 0. See *logical*.

`<=>` and `cmp` are the only two operators on this family that are *not* boolean - they are integer-valued and exist for sorting, not for if-conditions.

Floating-point equality

`==` on floats is exact bit-for-bit equality. For values produced by floating-point arithmetic this is almost never what you want:

```
0.1 + 0.2 == 0.3      # FALSE  -- 0.30000000000000004
```

Compare with a tolerance:

```
abs($a - $b) < 1e-9      # absolute tolerance
abs($a - $b) < 1e-9 * abs($a) # relative tolerance, when $a ≠ 0
```

`abs` is a *named-unary* operator.

Chaining is not magic

Perl does **not** support mathematical chaining of comparisons. `$a < $b < $c` does not test “\$b is between \$a and \$c” - it parses as `($a < $b) < $c`, where the first comparison yields a boolean ("" or 1) which is then numerically compared with `$c`:

```
1 < 5 < 3      # parses as (1 < 5) < 3
                #           = 1 < 3
```

(continues on next page)

(continued from previous page)

```
# = TRUE -- which is *not* what you
↳meant.
```

Write the conjunction explicitly:

```
$a < $b && $b < $c # the actual "between" test
```

Precedence

Comparison operators sit at row 11 of the *precedence* table - looser than the arithmetic operators (so `$a + $b == $c + $d` works), tighter than the logical operators (so `$a < $b && $c > $d` works). They are **non-associative**: `$a < $b < $c` is not invalid, but the non-associativity warning explained above applies.

See also

- *String comparison* - `eq ne lt le gt ge cmp`, the lexical-order family.
- *Logical operators* - combining comparison results.
- `<=>` *in sort* - the canonical use case.
- *Boolean Logic for Perl Programmers*
 - the “two flavours of comparison” overview that motivates the numeric/string split.
- *Numbers and strings* - when a scalar gets coerced to a number, and the precision/Inf/NaN story behind `==`.

String comparison operators

The string-flavoured counterparts of the numeric comparison family. Same shape, same precedence row, but operating lexicographically (Unicode code-point order by default) instead of numerically.

Operator	Question	Returns
<code>lt</code>	less than	true / false
<code>le</code>	less or equal	true / false
<code>eq</code>	equal	true / false
<code>ge</code>	greater or equal	true / false
<code>gt</code>	greater than	true / false
<code>ne</code>	not equal	true / false
<code>cmp</code>	three-way (sort)	-1, 0, or +1

Both operands are coerced to strings before comparing.

```
$name eq "John" # exact string match
$kind ne "guest" # negated match
```

(continues on next page)

(continued from previous page)

```
$word lt "m"           # alphabetically before "m"
$a      cmp $b         # sort comparator
```

Lexicographic order

Strings compare character by character, code-point by code-point. The first differing character decides; if one string is a prefix of the other, the shorter wins.

```
"abc" lt "abd"          # TRUE  -- 'c' < 'd' at position 2
"abc" lt "abcd"         # TRUE  -- prefix loses
"ABC" lt "abc"          # TRUE  -- ASCII: uppercase < lowercase
"10" lt "9"             # TRUE  -- '1' < '9' at position 0 (lex, not
↳ numeric!)
```

The last example is the canonical reason `eq/lt/gt` exist: when you want digit-strings sorted *numerically*, you must use `<=>` or pre-coerce.

Unicode

Code-point order is **not** the same as locale-aware “alphabetical” order:

- “ä” `gt` “z” is TRUE under code-point order because U+00E4 is beyond U+007A.
- Under German DIN 5007-1 (“dictionary”) order, “ä” should sort with “a” - long before “z”.

For locale-correct collation, use `Unicode::Collate` from *perlfunc* or use `locale` with a suitable locale set. The bare `lt/gt/cmp` give you ordered, stable, language-independent comparison - which is exactly the right thing for sort keys, hash bucketing, deterministic test output, and so on. It is the *wrong* thing for human-facing alphabetical listings in any non-English language.

cmp for sorting

`cmp` is the string-comparison spaceship. It returns -1, 0, or +1 and chains the same way `<=>` does:

```
my @sorted = sort { $a cmp $b } @names;      # ascending lex order
my @cased  = sort {
    lc($a) cmp lc($b) || $a cmp $b           # case-insensitive,
                                              # ties broken by case
} @names;
```

Mixing flavours: a worked bug

The compound-key sort idiom from *numeric comparison* showed `||`-chaining of `<=>` and `cmp`. The bug to avoid is using the *wrong* operator for the *type* of the key:

```
# version strings like "1.10", "1.2", "1.20", ...
sort @versions                # ASCII order:  "1.10",
    ↪ "1.2", "1.20"
sort { $a <=> $b } @versions   # numeric mash: works
    ↪ only by accident        # (everything past first
    ↪ dot ignored)
sort { sortkey($a) <=> sortkey($b) } @versions # parse first, then
    ↪ compare
```

The right answer for version strings is a parser like `Sort::Versions` or hand-rolled `(\d+)`-tokenisation; neither `cmp` nor `<=>` does it correctly on its own.

Precedence

String comparison shares row 11 of the *precedence* table with numeric comparison. They are non-associative - the chaining caveat from *numeric comparison* applies here too:

```
"a" lt "b" lt "c"           # parses as ("a" lt "b") lt "c"
                             #           = 1           lt "c"
                             #           = TRUE  (because "1" < "c"
    ↪lexically)
                             # - accidentally right for the wrong reason.
```

Write the conjunction explicitly with `&&`.

See also

- *Numeric comparison* - the parallel family.
- `sort`, `reverse`, `lc`, `uc`, `fc`
 - perlfunc tools that pair with string comparison.
- *Unicode in Perl* - the locale / code-point-order distinction in depth.

Logical operators

The boolean family - short-circuit AND, OR, defined-or; their low-precedence word-form siblings; the negations; and `xor`.

Operator	Reads as	Short-circuit	Returns	Precedence
!	not	n/a	normalised 1 or ""	high
&&	and	yes	left if false, else right (the operand)	high
	or	yes	left if true, else right (the operand)	high
//	defined-or	yes	left if defined, else right (operand)	high
xor	exclusive or	no	normalised 1 or ""	very low
not	not	n/a	normalised 1 or ""	very low
and	and	yes	same as &&	very low
or	or	yes	same as	very low

The full conceptual treatment of these operators - what truthiness means, why && returns one of its operands, how implication and XOR fit in, the De Morgan transformations that let you simplify tangled conditions - lives in the *Boolean Logic for Perl Programmers* tutorial. This page is the operator-reference companion.

Short-circuit and operand return

&& and || evaluate left-to-right and stop as soon as the result is determined.

- A && B - if A is false, return A without evaluating B. Otherwise return B.
- A || B - if A is true, return A without evaluating B. Otherwise return B.

The return value is the **operand itself**, not a normalised true/false. This is what makes Perl's defaulting idioms work:

```
my $port = $config{port} || 8080;           # operand of ||, not
↳boolean
my $name = $user_input || "anonymous";
my $val = $cache{$key} ||= compute($key);  # ||= sets if currently
↳false
```

For comparison, !, not, and xor *do* normalise: they always return either 1 or the empty string "".

// - defined-or

// short-circuits on **definedness** instead of truthiness:

- A // B - if A is defined, return A. Otherwise return B.

The difference matters when 0, "", or "0" is a legitimate value rather than a stand-in for “missing”:

```
my $port = $config{port} || 8080;           # 0 means "use default" -
↳wrong
```

(continues on next page)

(continued from previous page)

```
my $port      = $config{port} // 8080;      # 0 means "0", undef means
↳ "default"
my $verbose = $opt{verbose} // 0;          # 0 is a real choice
```

//= is the compound form:

```
$config{port} //= 8080;                    # set only if currently_
↳ undef
```

xor

The exclusive-or operator. Has only the very-low-precedence word form (there is no ^^), and unlike and/or it does *not* short-circuit (it cannot - both sides have to be evaluated to determine the result):

```
if ($admin xor $guest) { ... }           # exactly one of the two
```

The result is normalised - 1 or "", not an operand. For bitwise XOR on integers, see *bitwise*.

Symbolic vs word forms - precedence, not style

&&, ||, ! bind tightly (rows 15–16). and, or, not bind very loosely (rows 22–24). This matters at the boundary with assignment:

```
my $fh = open $h, '<', $path || die "no $path: $!";
#
# parses as: open $h, '<', ($path || die "no $path: $!")
# - open of "either $path or the die-message". Catastrophe.

my $fh = open $h, '<', $path or die "no $path: $!";
# parses as: ($fh = open $h, '<', $path) or die "no $path: $!"
# - what you meant.
```

Rule of thumb:

- Inside an expression, where you want the value to flow into a variable or another operator: use &&, ||, !, //.
- After a statement, where you want a side-effect (die, warn, return) to fire conditionally: use or, and, not.

! and not

Both negate. Both return the canonical 1 or "":

```
my $missing = ! $config{port};           # 1 if port is unset/0/"/"/0
↳ ""
return if not @items;                    # cleaner reading than `if !
↳ @items`
```

! binds high (row 5); not binds very low (row 22).

Compound forms

Three of the logical operators have compound assignment forms:

Compound	Means	Use
&&=	<code>\$x = \$x && \$y</code>	“narrow if currently true”
=	<code>\$x = \$x \$y</code>	“default if currently false”
//=	<code>\$x = \$x // \$y</code>	“default if currently undef”

```
$cache{$k} ||= compute($k);           # lazy populate
$cfg{port} //= 8080;                   # default that respects 0
$check    &&= validate($check);        # only if currently truthy
```

Tutorial cross-reference

The conceptual underpinnings of this page are covered in the boolean-logic tutorial:

- *Truthiness* - what Perl considers false.
- *Operators* - the operand-return rule, the precedence reasoning, and the ternary `?:` (which lives in *its own page* on this reference).
- *Truth tables* - all sixteen binary functions and how to spell each in Perl.
- *De Morgan* - reducing convoluted `unless` to clean `if`.
- *Functional completeness*
 - NAND/NOR.

See also

- *Bitwise* - the same boolean operators applied to integer bits in parallel.
- *Ternary* - `?:`, logical selection as an expression.
- *Numeric comparison* and *string comparison* - produce the boolean inputs that the operators on this page combine.
- *Error variables* - `$!` and `$@` are the values you typically put on the right-hand side of the `or die` idiom that this page’s word-form operators were made for.

Bitwise operators

The boolean operators applied to each pair of bits in parallel, plus the two shifts. A 32- or 64-bit integer is, from these operators’ point of view, a vector of independent one-bit values.

Operator	Operation	Per-bit rule
&	bitwise AND	each output bit = $a \wedge b$
	bitwise OR	each output bit = $a \vee b$
^	bitwise XOR	each output bit = $a \oplus b$
~	bitwise NOT	each output bit = $\neg a$ (unary)
<<	left shift	shift bits left, zero-fill from right
>>	right shift	shift bits right

Operands are coerced to integers (the floor toward zero, with warnings if a non-integer string is supplied under `use warnings`).

```
0xFF & 0x0F      # 0x0F    -- mask to low nibble
0x10 | 0x01      # 0x11    -- combine flag bits
0xFF ^ 0xAA      # 0x55    -- toggle alternating bits
~0               # -1 (or all-ones, depending on integer width)
1 << 4           # 16
32 >> 2          # 8
```

Setting, clearing, toggling, testing a flag

The four basic flag operations on a single bit:

```
use constant FLAG_VERBOSE => 0x01;
use constant FLAG_DRY_RUN => 0x02;
use constant FLAG_RECURSE => 0x04;
use constant FLAG_FORCE   => 0x08;

my $flags = 0;

$flags |= FLAG_VERBOSE;      # SET    -- OR with the bit
$flags |= FLAG_DRY_RUN;     # SET another
$flags &= ~FLAG_DRY_RUN;     # CLEAR  -- AND with the inverted bit
$flags ^= FLAG_VERBOSE;     # TOGGLE  -- XOR with the bit
my $on = $flags & FLAG_RECURSE; # TEST   -- AND, then test truthiness
```

Each is a one-bit application of a boolean operator: set is $\text{bit} \vee \text{flag}$, clear is $\text{bit} \wedge \neg \text{flag}$, toggle is $\text{bit} \oplus \text{flag}$, test is $\text{bit} \wedge \text{flag}$.

Compound assignment

All six binary bitwise operators have compound forms:

```
$bits &= $mask;      # AND-assign
$flags |= 0x01;     # OR-assign (set bit)
$x ^= $y;           # XOR-assign (toggle bits where $y is 1)
$x <<= 1;           # multiply-by-2 (with overflow caveats)
$x >>= 1;           # divide-by-2 (toward  $-\infty$  for signed values)
```

Shifts

<< shifts left; >> shifts right. The number of bit positions to shift is the right operand. Shifting by a negative amount, or by an amount equal to or greater than the integer's bit width, is implementation-defined - don't rely on it.

For unsigned arithmetic, left-shift is multiplication by $2^{** \$n}$ (modulo overflow):

```
$x << 1          # like $x * 2
$x << 8          # like $x * 256
```

For unsigned arithmetic, right-shift is integer division by $2^{** \$n}$:

```
$x >> 1          # like int($x / 2)   (for non-negative $x)
$x >> 8          # like int($x / 256)
```

For *signed* values, >> is implementation-defined - Perl does not guarantee arithmetic vs logical shift. Mask explicitly when sign matters:

```
($x >> 8) & 0xFF    # extract a byte at offset 8, regardless of
↳sign
```

Numeric vs bitwise mode

Perl's bitwise operators act on **integers** by default. With use feature 'bitwise' (or use v5.22+), the suffixed forms &., |., ^., ~. act on **strings** byte-by-byte:

```
use feature 'bitwise';
"\x80" |. "\x01"      # "\x81"  -- single-byte string OR
"abc" ^. "abc"        # "\0\0\0" -- bitwise XOR
```

The undecorated &, |, ^, ~ operators always do integer math, and string operands are coerced to numbers first. The suffixed .-forms are the way to ask for byte-string operations explicitly.

XOR-swap

XOR has two properties - $a \oplus a = 0$ and $a \oplus b \oplus b = a$ - that combine into the famous swap-without-temporary trick:

```
my ($a, $b) = (0xFEED, 0xBEEF);

$a ^= $b;          # a := a ⊕ b
$b ^= $a;          # b := b ⊕ (a ⊕ b) = a
$a ^= $b;          # a := (a ⊕ b) ⊕ a = b

# $a == 0xBEEF, $b == 0xFEED
```

Not faster than $(\$a, \$b) = (\$b, \$a)$ in Perl. Worth knowing because it appears in interview folklore and embedded code, and because it is a clean illustration of the XOR identity.

Common bit tricks

Patterns you will see in performance-sensitive code:

```
$x & ($x - 1)           # $x with its lowest set bit cleared
($x & ($x - 1)) == 0    # true when $x is a power of two (and non-
→zero)
1 << $n                 # the integer with only bit $n set
$x & (1 << $n)          # is bit $n set in $x?
$x | (1 << $n)          # set bit $n
$x & ~(1 << $n)         # clear bit $n
$x ^ (1 << $n)          # toggle bit $n
($x >> $n) & 0xFF       # extract a byte at offset $n
```

Each is an exercise in tracking what each bit does - pure boolean reasoning applied 32 (or 64) times in parallel.

Tutorial cross-reference

The boolean-logic tutorial has a longer treatment of the bitwise-as-parallel-boolean model and the application patterns above:

- *Boolean Logic - Applications*
 - bitwise operators as parallel boolean, flag manipulation, XOR-swap, common bit tricks.
- *Boolean Logic - Operators*
 - the boolean foundations the bitwise operators inherit.

See also

- *Logical* - the same boolean operators on whole values.
- *Precedence* - `&`, `|`, `^` sit between comparison and `&&`/`|`. Almost always parenthesise around them.
- `pack`, `unpack`, `vec` - perlfunc tools for bit-level work beyond what raw operators offer.

Range and flip-flop operators

Two operators with two completely different jobs depending on context: in **list context** they generate ranges, and in **scalar context** they are stateful “flip-flop” boolean expressions for matching line ranges in stream-processing.

Operator	List-context job	Scalar-context job
<code>..</code>	inclusive range	flip-flop (false-tested entry)
<code>... ..</code>	inclusive range	flip-flop (true-tested entry)

.. in list context - ranges

In list context, .. produces the inclusive sequence between its endpoints:

```

0..9           # (0,1,2,3,4,5,6,7,8,9)
'a'..'z'       # ('a','b',...,'y','z')      26 elements
'aa'..'zz'     # ('aa','ab',...,'zy','zz')  676 elements
'A'..'F'       # ('A','B','C','D','E','F')

for my $i (1..100) { ... }           # numeric for-loop
for my $name ('alpha'..'omega') { ... } # alphabetic,
↳magic-incremented

```

For numeric endpoints the result is the integer sequence (with strict ordering - 5..1 is the empty list, never reverse). For string endpoints the magic-increment from ++ is used, which gives you the natural alphabetic spread (and on 'aa' through 'zz' the expected 676 two-letter combinations).

A common Perl idiom:

```

my @hex = ('0'..'9', 'a'..'f');      # 16 hex digits as strings
my @rev = reverse 1..10;             # (10,9,8,...,1)
@arr[ 5..9 ]                         # array slice, indexes 5.
↳.9
@arr[ -3..-1 ]                      # last three elements

```

The all-integer range can be very large - 1..1_000_000_000 is a billion-element list. Perl 5 lazily generates loop ranges in for, so for my \$i (1..1_000_000_000) is safe, but my @x = 1..1_000_000_000 allocates the whole array. PetaPerl follows the same rule.

Reversed endpoints

5..1 is empty. There is no “reverse range” form; if you want descending, generate ascending and reverse:

```

1..5           # (1,2,3,4,5)
5..1           # ()          - empty, no warning
reverse 1..5    # (5,4,3,2,1)

```

Mixed-type endpoints

If either endpoint *looks* numeric, .. does numeric ranging:

```

'1'..'5'       # (1,2,3,4,5)          - numeric
'1'..'10'      # (1,2,3,4,5,6,7,8,9,10) - numeric

'a'..'5'       # ('a')                - magic-increment
↳from "a",     #                      stops because it
↳can't reach "5"

```

Make the type explicit to avoid confusion:

```
0+'1' .. 0+'5'           # numeric, no doubt
sprintf("%d", 1) .. sprintf("%d", 5)
```

In practice, write integer ranges with bare integer literals and character ranges with bare quoted-string letters; the magic-vs-arithmetic distinction follows automatically.

.. and ... in scalar context - flip-flops

Inside a boolean context (an `if`, a `while`, a `?:`, the right of `&&|||`), `..` becomes a *stateful* expression that remembers whether it is currently between a left-side match and a right-side match.

```
while (<$fh>) {
    if (/^BEGIN/ .. /^END/) {      # true from BEGIN line through END
    ↪line
        print;                    # print everything in between,
    ↪inclusive
    }
}
```

The semantics:

1. The flip-flop is initially **false**.
2. When the **left** condition (`/^BEGIN/`) becomes true, the flip-flop transitions to **true**, and the right side is evaluated immediately on the same iteration.
3. The flip-flop stays true on subsequent iterations until the **right** condition (`/^END/`) becomes true. On that iteration it returns true *and then* transitions back to false.
4. So both the BEGIN line and the END line are included.

When the flip-flop is true, it returns a *sequence number* (1 on the entering iteration, 2 on the next, ...). On the closing iteration it returns the count followed by `E0` - so the value is true (non-empty string, not the false `"0"`) but stringifies as the number `0` for cosmetic clarity. You can usually treat it as plain boolean.

`...` differs from `..` in one detail:

- `..` (two dots) - when the left side becomes true, the right side is checked **on the same iteration**. So the same line can open and close the range.
- `...` (three dots) - when the left side becomes true, the right side is **not** checked until the next iteration. Always a multi-line range.

```
while (<$fh>) {
    print if /<head>/ .. /\</head>/;    # one-line <head>...</head>
    ↪works
    print if /<head>/ ... /\</head>/;    # forces multi-line; same-
    ↪line ignored
}
```

When you'd use a flip-flop

The flip-flop shines in awk-style line-range filtering - pulling sections out of a log file, isolating a fenced code block from markdown, processing a configuration block delimited by keywords. It is famously *un*-loved by code reviewers because its statefulness is implicit.

Two practical alternatives that some prefer:

```
# Explicit state variable - equivalent semantics, easier to grep
→for:
my $in_block = 0;
while (<$fh>) {
    $in_block = 1 if /^BEGIN/;
    print if $in_block;
    $in_block = 0 if /^END/;
}

# A simple two-step iterator:
while (<$fh>) {
    if (/^BEGIN/ .. /^END/) { print }
}
```

The flip-flop wins on conciseness; the explicit form wins on clarity at the cost of three more lines. Pick by audience.

See also

- [reverse](#), [grep](#), [map](#) - list operators commonly composed with ranges.
- [Subscript](#) - array slices `@arr[1..3]` rely on range-in-list-context.
- [Precedence](#) - `..` and `...` sit at row 17, between `|||` and `?:`. Almost always parenthesise when combining with logic.

Regex binding operators

Two operators that connect a string with a regular expression operation. They do not perform the match themselves; they merely say “this expression’s input is that string”.

Operator	Reads as	Use
<code>=~</code>	matches	bind a regex op (<code>m//</code> , <code>s///</code> , <code>tr///</code>)
<code>!~</code>	doesn't match	same, with the boolean result negated

```
$str =~ /pattern/           # match: TRUE if $str contains
→a match
$str =~ s/foo/bar/         # substitution: returns count
→of changes
$str =~ tr/a-z/A-Z/        # transliteration: returns
→count
```

(continues on next page)

(continued from previous page)

```
$str !~ /pattern/           # match negated: TRUE if NO
↳match
```

Without an explicit binding, regex ops act on \$_:

```
$_ = "hello";
print "match\n" if /h/;      # implicit $_ binding
```

=~ and !~ are how you redirect that input to a different variable.

What they actually return

=~ returns whatever the regex op on the right would return:

- `m//` - boolean (true on match, false on no match) in scalar context; the captured groups in list context.
- `s///` - the number of substitutions performed (which is boolean-true when ≥ 1).
- `tr///` - the number of characters processed.

!~ returns the boolean negation, regardless of the underlying op. It is mostly used with `m//`:

```
print "no digit"   if $str !~ /\d/;
print "$n changes" if $str =~ s/foo/bar/g;
my @hits          = $str =~ /(\w+)/g;      # list context:
↳captures
```

Three op partners

=~ accepts three regex operations on its right:

- `m//` - match. The `m` is optional when the delimiters are slashes: `$s =~ /pattern/` and `$s =~ m{pattern}` both work.
- `s///` - substitution. Three pieces: pattern, replacement, flags. Returns the count of replacements.
- `tr///` (also spellable `y///`) - transliteration. Replaces characters one-for-one between two character sets. Returns the count of characters processed.

```
$line =~ /(\d+)/          # extract first run of digits
$line =~ s/^\s+//         # strip leading whitespace
$line =~ s/\s+/ /g        # collapse all whitespace to single
↳spaces
$line =~ tr/A-Z/a-z/      # ASCII lowercase
```

!~ only with m//

!~ is meaningful only with the *match* operation, since substitution and transliteration return *counts* and the “no-changes-made” case is a meaningful zero, not a “didn’t match” boolean. Perl will let you write `$s !~ s/.../.../`, but you almost never want it - the result `!s///` is “true if zero substitutions” which reads strangely. Stick to `!~ /.../`.

Lvalue vs rvalue

`=~` itself does not assign. It only routes the regex op on its right at the string on its left. The op *itself* may then mutate that string - `s///` and `tr///` do, `m//` does not - but the mutation comes from the op, not from `=~`:

```
my $s = "hello";
$s =~ s/l/L/g;      # mutates $s - now "heLlo"
my $n = $s =~ /(\w+)/; # does NOT mutate $s; $n is the boolean
                     ↪ result
```

The string on the left of `=~` must be modifiable for `s///` and `tr///`. A literal string or `$1` (a regex capture variable) will fail with “Modification of a read-only value attempted”:

```
"hello" =~ s/l/L/g;      # FATAL - literal is read-only
$1 =~ s/x/y/;           # FATAL - capture variable is read-only
```

Copy first if you need to modify a read-only source:

```
(my $copy = $1) =~ s/x/y/;      # idiom for "modify a copy of $1"
```

Precedence

`=~` and `!~` sit at row 6 of the *precedence* table - quite tight, between unary and the multiplicative operators. This is why you can write `$s =~ /foo/ && $t =~ /bar/` without parens around either match.

Tutorial cross-reference

The boolean-logic tutorial covers regex-as-set-algebra in its applications chapter - alternation as union, lookarounds as intersection and complement:

- *Boolean Logic - Applications* (the *Regular expressions: logic on sets of strings* section)

The full regex language reference lives in the regex guide:

- *Regular expressions guide*

See also

- `m//`, `s///`, `tr///`, `qr//`, `split` - the regex-consuming perlfunc forms.
- `$_` - the default input variable when no binding is given.
- *Regex match variables* - `$$`, `$1..$N`, `%+`, `@+`, `@-`: what every successful match populates.

Assignment operators

Plain assignment plus the compound forms - every binary operator that produces a value also has a compound assignment form that applies the operation in place.

Plain =

```
my $x = 10;           # scalar assignment
my @a = (1, 2, 3);    # list assignment
my ($x, $y, $z) = (1, 2, 3); # destructuring
my ($first, @rest) = @items; # one + remainder
my %h = (a => 1, b => 2);  # hash from key-value list
```

Assignment is **right-associative** - chains are evaluated right-to-left:

```
$a = $b = $c = 0;      # parses as $a = ($b = ($c = 0))
                        # all three become 0
```

The whole assignment expression has a value: in scalar context it is the value assigned; in list context it is the **list of elements assigned to**, in left-to-right order:

```
my $count = (my @items = qw(a b c)); # $count = 3
                                         # - number of elements assigned
                                         # to @items, in scalar context

while ((my $line = <$fh>) ne '') { ... } # uses the assignment as
                                         # both side-effect and value
```

List assignment in scalar context

Probably the most-used Perl idiom that nobody can quote the rule for:

```
my $n = () = $string =~ /\d/g;    # count digits in $string
```

Decoded:

1. `$string =~ /\d/g` in list context yields all digit captures.
2. `() = ...` is a list assignment to the empty list (assigns nothing, but evaluates the right-hand side in list context).
3. `$n = (...)` puts the *list assignment* in scalar context, which evaluates to the **number of elements** on the right.

Result: `$n` is the count of matches. There is no `scalar(grep ...)` or `count` built-in for this; the empty-list trick is the canonical idiom.

Compound assignment

Every binary operator that fits the pattern `OP=` exists. The mechanics are identical to `$x = $x OP $y`, *except* that `$x` is evaluated only once on the left - which matters when the lvalue has a side-effect (e.g. an `autoviv`, a function call):

```
$h->{counter}++;           # autoviv $h->{counter}, increment
↳increment
$h->{counter} += 1;         # same effect, but spell out the OP
↳the OP
```

Operator	Equivalent	Use
<code>+=</code>	<code>\$x = \$x + \$y</code>	accumulate
<code>-=</code>	<code>\$x = \$x - \$y</code>	subtract in place
<code>*=</code>	<code>\$x = \$x * \$y</code>	scale
<code>/=</code>	<code>\$x = \$x / \$y</code>	divide in place
<code>%=</code>	<code>\$x = \$x % \$y</code>	modulo
<code>**=</code>	<code>\$x = \$x ** \$y</code>	exponent in place
<code>.=</code>	<code>\$x = \$x . \$y</code>	append (string)
<code>x=</code>	<code>\$x = \$x x \$y</code>	repeat (string), or (<code>@</code>) <code>x= n</code>
<code>&=</code>	<code>\$x = \$x & \$y</code>	bitmask AND-equals
<code> =</code>	<code>\$x = \$x \$y</code>	bitmask OR-equals (set bits)
<code>^=</code>	<code>\$x = \$x ^ \$y</code>	bitmask XOR-equals (toggle)
<code><<=</code>	<code>\$x = \$x << \$y</code>	shift-left in place
<code>>>=</code>	<code>\$x = \$x >> \$y</code>	shift-right in place
<code>&&=</code>	<code>\$x = \$x && \$y</code>	“narrow if currently true”
<code> =</code>	<code>\$x = \$x \$y</code>	default if currently false
<code>//=</code>	<code>\$x = \$x // \$y</code>	default if currently undef

```
$count    += 5;           # arithmetic
$path     .= "/user";     # string append
$flags    |= FLAG_VERBOSE; # set a flag bit
$flags    &= ~FLAG_VERBOSE; # clear a flag bit
$cache    ||= load_data(); # lazy populate
$cfg{key} //= $default;   # default that respects 0 and "
↳"
$str      x= 3;           # repeat in place
```

Lvalues - what can appear on the left

The left operand of `=` (or any compound form) must be an **lvalue** - a location that can be written to. Perl’s lvalues include:

- A simple scalar variable: `$x`, `$obj->{field}`.
- An array or hash variable: `@arr`, `%h`.
- An array element or slice: `$arr[3]`, `@arr[1..3]`.
- A hash element or slice: `$h{key}`, `@h{qw(a b c)}`.

- A dereferenced container: `$$ref, @$aref, %{$h}`.
- The `local`-/`my`-declared form of any of the above.
- A *substring*: `substr($s, $offset, $len) = "..."`.
- A *ternary* whose branches are lvalues (see [ternary](#)).
- The pseudo-operators `pos`, `vec`, `keys` on tied hashes, certain `@_` slots, and a handful of specialised forms.

Most of the time you write an lvalue without thinking. The cases where you stop and check are exotic - `substr` as lvalue, a ternary as lvalue, an array slice as the target of a list assignment.

Multiple-target list assignment

The right-hand side is evaluated *first* in list context, then distributed across the lvalues on the left. This makes swap easy:

```
($a, $b) = ($b, $a);           # swap
($x, $y, $z) = ($y, $z, $x);    # rotate left
```

If the lists differ in length:

- More values on the right than slots on the left → extras silently discarded.
- More slots on the left than values on the right → unfilled slots become `undef`.

```
my ($a, $b) = (1, 2, 3, 4);      # $a=1, $b=2; 3 and 4 dropped
my ($a, $b, $c) = (1, 2);       # $a=1, $b=2, $c=undef
```

A trailing array or hash on the left slurps the remainder:

```
my ($head, @tail) = @items;      # $head = first element
                                # @tail = the rest
my ($head, %tail) = @items;      # if @items has key/value pairs
                                # after the first
```

Only **one** array or hash on the left side; otherwise it would not be unambiguous where one ends and the next begins.

Precedence

Assignment operators sit at row 19 - quite low, looser than comparison and `?:`, tighter only than the comma operators and the word-form logicals (`and`, `or`, `not`, `xor`). This is why `my $fh = open ... or die` works (or is even looser than `=`).

See also

- *Logical* - `||=`, `!=`, `&&=` are the workhorses of “default if missing” idioms.
- *Subscript* - most lvalue forms involve a subscript.

- *Comma* - = parses out from the surrounding , -list via precedence; understanding row 19 vs row 20 is what tells you when (a, b)=(c, d) distributes vs being a single list.

Ternary operator

One operator, three operands. Logical selection as an *expression* - the result has a value, can be assigned, returned, passed as an argument.

Form

```
COND ? THEN : ELSE
```

If COND is true, the value is THEN; otherwise it is ELSE. Only the chosen branch is evaluated:

```
my $label = $count == 1 ? "1 item" : "$count items";
return $ok ? \%result : undef;
my $abs    = $x < 0      ? -$x      : $x;
```

It is an expression, not a statement

The most common reason to reach for ?: instead of if/else is that you want a *value* - to assign, to return, to put inside another expression:

```
# verbose
my $kind;
if ($n == 1) { $kind = 'singular' }
else        { $kind = 'plural'   }

# idiomatic
my $kind = $n == 1 ? 'singular' : 'plural';
```

Chaining (right-associative)

A ? B : C ? D : E parses as A ? B : (C ? D : E) - the chain descends through the false branch:

```
my $sign = $x < 0 ? '-'
           : $x > 0 ? '+'
           :       '0';
```

The columnar layout makes the chain readable. Past two or three levels, switch to if/elsif/else - the chain stops being clearer than the alternative.

Ternary as lvalue

Both branches must be lvalues, and the chosen branch is the lvalue you assign to:

```
my ($pos, $neg) = (0, 0);
($x > 0 ? $pos : $neg) = $x;           # assigns to $pos or $neg

# Common case: pick a buffer to append to
($verbose ? $log : $debug) .= "$msg\n";
```

This pattern is rare in modern code - an explicit `if` with two straightforward assignments reads better. Worth knowing for when you encounter it.

What you don't write

`?:` is *not* a statement separator. These are wrong:

```
$x ? do_a() : do_b();                 # OK: do_a/do_b have side-
    ↪effects,                          # but the value is
                                     ↪discarded
$x > 0 ? print "yes\n" : print "no\n"; # WORKS, but reads
    ↪strangely
```

When the goal is a side-effect (rather than a value), use `if/else` or the postfix `if/unless`:

```
$x > 0 ? do_yes() : do_no();           # value-return phrasing
do_yes() if $x > 0;                   # side-effect phrasing
do_no() unless $x > 0;
```

With short-circuit logical operators

`?:` interacts with `&&`, `||`, `//` in ways that can confuse the eye. Almost always parenthesise:

```
my $x = $a || $b ? $c : $d;           # parses as ($a || $b) ? $c
    ↪: $d                               # because || binds tighter
                                     ↪than ?:
my $x = $a || ($b ? $c : $d);         # if you meant the other
    ↪thing,                             # parenthesise
```

Precedence

`?:` sits at row 18 - looser than the boolean operators (`&&`/`||`/`//`), tighter than assignment. This is why `my $x = COND ? A : B` works without parens around the right-hand side.

See also

- *Logical* - short-circuit && and ||, the alternative-without-if family.
- *Boolean Logic - Operators*
 - ?: in the context of all the boolean operators.
- *Assignment* - ?: as lvalue.

Subscript and slice operators

The bracket-and-brace family that picks elements out of arrays and hashes - and the deref-brace family that does the same through a reference.

Form	Selects	Container	Result type
<code>\$arr[i]</code>	one element	array	scalar
<code>\$hash{key}</code>	one element	hash	scalar
<code>@arr[i, j, k]</code>	many elements (slice)	array	list
<code>@hash{k1, k2}</code>	many elements (slice)	hash	list
<code>%arr[i, j, k]</code>	key/value slice	array	list (k,v...)
<code>%hash{k1, k2}</code>	key/value slice	hash	list (k,v...)

A few rules cover all twelve combinations.

The sigil signals the *result*, not the container

`@arr` is the array; `$arr[3]` is one of its elements. The leading sigil tells you what *kind of value* you are accessing - scalar (\$), list of values (@), key/value list (%):

```
my @arr = (10, 20, 30, 40, 50);

$arr[2]           # 30                - scalar (single element)
@arr[1, 3]        # (20, 40)         - list of values at those
↳ indices
%arr[1, 3]        # (1 => 20, 3 => 40) - list of (index, value)
↳ pairs
```

```
my %h = (a => 1, b => 2, c => 3);

$h{a}             # 1                - scalar
@h{'a', 'c'}      # (1, 3)          - list of values
%h{'a', 'c'}      # (a => 1, c => 3) - list of (key, value) pairs
```

The “%-slice” forms (`%arr[...]`, `%hash{...}`) are a Perl 5.20 addition that returns key/value pairs as a list. They are the right tool when you want a copy of *part of* a hash:

```
my %selected = %h{ qw(a c) };           # %selected = (a => 1, c =>
↳ 3)
```

Negative indices on arrays

Negative array indices count from the end:

```

$arr[-1]      # last element
$arr[-2]      # second-from-last
@arr[-3..-1]  # last three elements as a slice

```

Out-of-range positive indices read as undef. Out-of-range negative indices (further back than the start) raise a fatal runtime error:

```

my @a = (1, 2, 3);
$a[10]      # undef (with no warning)
$a[-10]     # FATAL: Modification of non-creatable array value
             # (the error wording reflects how the lookup is
             ↪ implemented)

```

Slice as lvalue

Both array and hash slices are valid assignment targets:

```

@arr[0, 2, 4] = ('A', 'C', 'E');      # set positions 0, 2, 4
@h{qw(a b c)} = (1, 2, 3);            # set three hash keys

(@arr[0, 1], @arr[1, 0]) = ...        # be careful: aliasing!

```

You can swap elements via slice assignment without a temporary:

```

@arr[$i, $j] = @arr[$j, $i];          # swap two array elements
@h{$key1, $key2} = @h{$key2, $key1};  # swap two hash values

```

Hash key autoquoting

Inside {} for hash subscripts, **bareword identifiers are auto-quoted**:

```

$h{name}     # same as $h{'name'}
$h{first_name} # same as $h{'first_name'}
$h{1_2}      # FATAL - not a valid bareword identifier
$h{"1_2"}    # quote it explicitly

```

This is one of the most-loved (and most-bug-causing) Perl conveniences. The bareword-or-not decision happens at parse time based on the literal token; constants and computed values do not benefit:

```

use constant KEY => 'name';
$h{KEY}          # WRONG - this is the bareword "KEY", not the
                 ↪ constant!
$h{KEY()}        # use empty parens to force function-call
                 ↪ evaluation
$h{+KEY}         # or unary +, the canonical workaround

```

The +KEY form is the most compact way to say “this is *not* a bareword, evaluate it as an expression”.

Array indices: integer-coerced, no autoquoting

[] indices are evaluated as expressions and coerced to integers (truncating toward zero):

```
$arr[1.7]      # $arr[1]      - 1.7 truncates to 1
$arr["3"]      # $arr[3]      - string coerces to integer
$arr[$#arr]    # last element ($#arr is last valid index)
```

The \$#arr syntax inside subscripts is idiomatic for “last”:

```
$arr[$#arr]    # last
@arr[0..$#arr] # all of @arr (verbose; just write @arr)
@arr[1..$#arr] # all except first
```

Mixing - multidimensional access

Perl 5 does not have native multi-dimensional arrays; nested data is built from references:

```
my @grid = ([1,2,3], [4,5,6], [7,8,9]);

$grid[1][2]      # 6                - chained subscripts on AoA
$grid[1]->[2]     # same - the arrow is optional between subscripts
$grid[$y][$x]
```

The arrow-between-subscripts is implied by Perl’s grammar: when two subscripts immediately follow each other on a deref chain, the -> can be omitted. See [arrow](#).

See also

- [Arrow](#) - ->[] and ->{} for going through a reference; the arrow-elision rule.
- [Assignment](#) - slice assignment; multi-target destructuring.
- [exists](#), [delete](#) - perlfunc forms that take a subscript expression.
- [keys](#), [values](#) - whole-container forms; the slice operators are the partial forms.
- [Arrays](#), [Hashes](#) - the data-types pages explain the sigil-vs-result rule that governs which subscript form to use.

Arrow operator

The single most heavily used operator in working Perl, after =. It does three things at three syntactic positions, and they all start by *dereferencing* something on the left.

Form	Means	Example
<code>\$ref->[i]</code>	element of an arrayref	<code>\$aref->[0]</code>
<code>\$ref->{k}</code>	element of a hashref	<code>\$href->{name}</code>
<code>\$ref->(...)</code>	call a coderef	<code>\$cref->(@args)</code>
<code>\$obj->meth</code>	method call on an object	<code>\$obj->print</code>
<code>\$obj->meth(@a)</code>	method call with arguments	<code>\$obj->set_x(42)</code>
<code>\$class->meth</code>	class method call	<code>Foo::Bar->new</code>

The first three forms are *dereferencing*. The last three are *method dispatch*. Both are spelled with `->`.

Dereferencing

A reference is a scalar that points at a container or a coderef. You cannot index into a reference directly with `[]` or `{}`; you must dereference first, and `->` is the canonical way:

```
my @arr = (10, 20, 30);
my $aref = \@arr;

$arr[0]           # 10 - direct array access
$$aref[0]         # 10 - dereference, then subscript (old syntax)
$aref->[0]         # 10 - dereference and subscript in one move
$aref->[-1]        # 30 - negative indices work the same way
```

```
my %h = (name => 'John');
my $href = \%h;

$h{name}          # 'John'
${$href}{name}    # 'John' - full deref + subscript
$href->{name}      # 'John' - same, idiomatic form
```

Coderef invocation

```
my $f = sub { $_[0] * 2 };

$f->(5)            # 10 - invoke the coderef with argument 5
&$f(5)            # same - older syntax, still works
```

The `->()` form is preferred in modern Perl. Empty argument list is `$f->()`.

Arrow elision between subscripts

When two subscripts directly follow each other on a deref chain, the **second arrow can be omitted**. The grammar implies it:

```

$grid[1]->[2]      # explicit
$grid[1][2]        # arrow elided - the only way most people write
→this

$tree->{children}[0]{name}  # parses with arrows between every
→pair                      # - the chain is read as a sequence
                           # of subscripts on the same deref path

```

The first arrow (between a scalar variable and the *first* subscript) is **required**. Only arrows *between consecutive subscripts* may be elided.

```

$aeref->[0]->[1]    # explicit
$aeref->[0][1]       # elided second arrow - fine
$aeref[0][1]        # WRONG - $aeref[0] reads from @aeref, not the
                   # arrayref in $aeref. Different variable!

```

The first-arrow rule is the source of subtle bugs when refactoring `$aeref->[0]` into a deeper expression - keep the leading arrow.

Method calls

When the right side of `->` is an *identifier* (or a parenthesised name expression), it is interpreted as a method call rather than a subscript:

```

$obj->print;          # call $obj->print()
$obj->name;            # accessor: returns $obj->name
$obj->set_name("Alice"); # mutator: sets and may return $obj
Foo::Bar->new(arg => 1); # class method call

$obj->$method_name(@args); # method name in a variable
$class->can('foo')         # can() - does method exist?

```

Method calls dispatch through the package the object is blessed into. The exact resolution path is governed by @ISA and the chosen MRO; see the *OOP guide* for the full story.

Indirect-object syntax - don't use it

Perl 5 historically allowed `print $fh "foo"` and `new Foo @args` without an arrow. These forms still parse but are context-sensitive in ways that surprise - the indirect-object syntax can resolve the `new` (or whatever) at the wrong time. The modern guidance is to **always use** `->` for method calls:

```

my $obj = Foo->new(@args); # correct
my $obj = new Foo @args;  # avoid (still
→parses)

print $fh "hello\n";      # OK - print is
→special
$fh->print("hello\n");    # also fine, prefer

```

(continues on next page)

(continued from previous page)

```

↪this                                     # for IO::File-style_
↪handles

```

Precedence

-> is row 2 of the [precedence](#) table - second only to terms and list operators. It binds tighter than every other operator, which is what you want: `$h->{a} + $h->{b}` parses as the sum of two hash lookups, not as a lookup of `{a + $h->{b}}`.

See also

- [Subscript](#) - direct (non-dereferenced) array/hash access; the arrow-elision rule.
- [References tutorial](#) - full treatment of references, including how \ builds them and how the deref alternatives (`@{...}`, `%{...}`, `${...}`) work.
- [OOP guide](#) - ->method and class dispatch in depth.
- [ref](#), [bless](#) - perlfunc tools that pair with the arrow operator's two roles.
- [References](#) - what \X produces and how the deref shapes (`@{...}`, `%{...}`, `${...}`) compose.

Comma operators

Two operators, both spelled with a comma - one of them spelled “fat comma” with `=>`. They are syntactically the same in most positions but differ on the left-hand-side bareword rule.

Operator	Reads as	What it does
,	comma	list separator (in list ctx); discard-and-yield-right (in scalar ctx)
=>	fat comma	same as ,, plus auto-quotes a bareword on the left

In list context - building lists

In list context the comma is the list-element separator:

```

my @items = (1, 2, 3, 4);           # 4-element list
my @flat  = (1, 2, (3, 4), 5);      # (1,2,3,4,5) - lists_
↪flatten
my @kv    = (a => 1, b => 2);        # 4-element list: a, 1,
↪b, 2
my %h     = (a => 1, b => 2);        # hash from key-value_
↪pairs

```

Lists in Perl are *flat* - there is no nested-list value. A list embedded in another list disappears: its elements simply become elements of the outer list. To get nested structure you need references (see the [References tutorial](#)):

```
my @flat    = (1, 2, (3, 4), 5);           # (1,2,3,4,5) - 5_
    ↪elements
my @nested  = (1, 2, [3, 4], 5);          # (1, 2, ARRAYREF, 5) - 4_
    ↪elements
```

In scalar context - comma is “discard, then return right”

When a comma expression is *forced* into scalar context - by an assignment to a scalar, by `scalar(...)`, by being the condition of an `if` - it evaluates each operand left-to-right for side-effects, then yields the value of the rightmost operand:

```
my $x = (1, 2, 3);                        # $x = 3 - but warn_
    ↪under                                # `use warnings`:
    ↪"Useless use of                      # a constant in void_
    ↪context"                            # for 1 and 2

my $r = (do_a(), do_b(), final_value());   # runs all three,_
    ↪returns                             # the last value
```

This usage is rare in modern Perl - the C-style `for(init, init; cond; step, step)` idiom is the main place you encounter it (and even there, parens around the comma sub-expression are advisable).

=> - the fat comma

`=>` does the same thing as `,` *plus* it auto-quotes the identifier-shaped bareword on its left:

```
my %h = (
    name => "John",                        # 'name' auto-quoted
    age  => 30,
    "key with spaces" => "...",           # quote required for_
    ↪non-bareword
);

# Equivalent without =>, requiring explicit quoting:
my %h = (
    'name', "John",
    'age', 30,
    'key with spaces', '...',
);
```

The auto-quote applies only when the left side is a **syntactically valid bareword identifier** - letters, digits, underscores, starting with a letter or underscore. Other shapes are

not auto-quoted:

```
( name      => 1 )      # 'name' auto-quoted - fine
( "name"    => 1 )      # already a string - fine
( $variable => 1 )      # $variable evaluated - fine
( 0+1       => 1 )      # numeric expression - fine
( -name     => 1 )      # WRONG - looks like negative bareword;
                        # parses as -('name') which is the
                        # numeric negation of a string. Quote_
→it.
( CONST     => 1 )      # 'CONST' auto-quoted - does NOT call
                        # the constant CONST!
( CONST()   => 1 )      # explicit empty parens - calls CONST
```

The interaction with constants is the most common surprise - see also the [subscript](#) page on hash key autoquoting, which has the same pattern.

Common idioms

```
# Function arguments - the comma is just the argument separator
some_function($arg1, $arg2, $arg3);

# Hash construction - fat-comma communicates "this is a key"
my %config = (
    host => 'localhost',
    port => 8080,
    user => $ENV{USER},
);

# List literal - interchangeable
my @months = (qw/Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec/);

# Trailing comma - allowed and idiomatic for diff-friendly multi-
→line literals
my @list = (
    "first",
    "second",
    "third",      # trailing comma is fine
);
```

The trailing comma is intentional: it lets you append, remove, or reorder list elements without touching surrounding lines, which shows up as cleaner diffs in version control.

Precedence

, and => sit at row 20 - looser than assignment, tighter only than the word-form logicals (and/or/not/xor) and list operators in their right-side position. This is why a list of assignments works:

```
my $a = 1, my $b = 2;      # WRONG: parses as (my $a = 1), my $b.
    => 2                    # - comma at outer level, not list.
    assignment.            # $a is 1, $b is 2, but the warning.
    is loud.
my ($a, $b) = (1, 2);      # the right way: list assignment
```

See also

- *Assignment* - list assignment relies on the comma's role in list context.
- *Subscript* - `$h{name}` autoquoting, the parallel case to `name => ...`
- *use constant* - the bareword/constant collision is the canonical fat-comma gotcha.
- *Context* - the list-vs-scalar story behind why `,` means two different things.

Precedence

Two pieces, in this order: the full precedence table, and the parse rule for *named unary operators* - built-in functions whose syntax binds at operator precedence rather than as full list-operator calls.

The full table

Operators tighter at the top, looser at the bottom. Same row = same precedence; combining two same-row operators uses associativity. none = non-associative (combining two of them without parens is an error or a parse-time warning).

Row	Associativity	Operators
1	left	terms, list operators (leftward)
2	left	->
3	none	++ --
4	right	**
5	right	! ~ \ unary + unary -
6	left	=~ !~
7	left	* / % x
8	left	+ - .
9	left	<< >>
10	none	named unary operators
11	none	< <= > >= lt le gt ge
12	none	== != <=> eq ne cmp
13	left	&
14	left	^
15	left	&&
16	left	//
17	none	
18	right	?:
19	right	= += -= *= /= etc.
20	left	, =>
21	none	list operators (rightward)
22	right	not
23	left	and
24	left	or xor

A few rules of thumb:

- **Comparison and logic.** Comparison (rows 11–12) is tighter than &&/|| (rows 15–16), so `$a < $b && $c > $d` works without inner parens.
- **Bitwise vs comparison.** `&`, `|`, `^` (rows 13–14) are *looser* than comparison. `$x & 0xFF == 0` parses as `$x & (0xFF == 0)`. Almost always parenthesise around bitwise.
- **Symbolic vs word logic.** `&&/||` are tight (rows 5, 15, 16). `and/or/not` are very loose (rows 22–24). Same logical meaning; different binding. The *logical* chapter shows where each is appropriate.
- **= is loose.** Row 19 - almost everything binds tighter than assignment. This is why `my $fh = open ... or die` works: `or` (row 24) is even looser than `=`, so the `open` finishes assigning before `or` decides.
- **List operators bookend the table.** Row 1 looking *leftward*, row 21 looking *rightward*. A list operator like `print` or `sort` slurps everything to its right, up to the first operator that binds looser than row 21 - which is just `not/and/or/xor`. The *named unary* section below contrasts this with the row-10 single-argument case.

When in doubt, write parentheses. They cost nothing and document intent.

Post and pre increment / decrement

Row 3 (`++` and `--`) is non-associative. The pre and post forms differ in *when* the side-effect happens:

- `++$x` - increment first, then yield the new value.
- `$x++` - yield the current value, then increment.

```
my $x = 5;
my $a = ++$x;      # $x is now 6, $a is 6
my $b = $x++;      # $a was 6, $x++ yields 6 then increments,
                  # so $b is 6 and $x is now 7
```

`++` on a string performs Perl's *magic increment*:

```
my $s = "aa";
$s++;      # "ab"
$s++;      # "ac"
# ...
$s = "az"; $s++;      # "ba" - carries like base-26
$s = "Aa"; $s++;      # "Ab" - case is preserved per-position
$s = "Az9"; $s++;     # "Ba0" - digits and letters carry
→independently
```

The magic-increment is what makes `'a' .. 'zz'` produce all 702 two-letter combinations in *range*.

Named unary operators

Row 10 contains a category called *named unary operators*. These are not symbolic operators - they are **built-in functions** that take exactly one argument and parse with operator-style precedence (between bitshifts and comparison) rather than as full list operators.

The practical consequence is one parsing rule:

A named unary operator grabs its **one** argument greedily, and anything that would need more than one argument is not part of it.

```
defined $x + 1      # parses as: ( defined($x) ) + 1
                   # - NOT defined($x + 1)
length $s > 3       # parses as: ( length($s) ) > 3
-e $f && -r _        # both file tests are named-unary;
                   # `&&` combines two boolean results
```

Compare with a list operator (`print`, `sort`, `push`, ...), which is at row 1 / 21 and slurps the rest of the expression:

```
print "n=", $n, "\n"  # print's argument list is "n=", $n, "\n"
                   # all three commas belong to print
```

If you want a named-unary built-in to take a more complex argument, parenthesise:


```
defined($x + 1)      # explicit: argument is the sum $x+1
length($s . "x")    # length of $s with "x" appended
```

The unambiguous form is always available. When in doubt, parenthesise.

The list of named unary built-ins

These built-ins parse at row-10 precedence. Each links to its [perlfunc](#) detail page - the parsing rule above is the only thing you need to know about them as *operators*; their *function* behaviour is in the [perlfunc](#) reference.

[defined](#), [exists](#), [ref](#), [scalar](#), [length](#), [uc](#), [lc](#), [ucfirst](#), [lcfirst](#), [chr](#), [ord](#), [hex](#), [oct](#), [int](#), [abs](#), [sqrt](#), [sin](#), [cos](#), [log](#), [exp](#), [rand](#), [srand](#), [alarm](#), [sleep](#), [caller](#), [wantarray](#), [chroot](#), [readlink](#), [umask](#), [lstat](#), [stat](#) (when given exactly one argument), plus all the **file test operators** ([-e](#), [-r](#), [-w](#), [-x](#), [-f](#), [-d](#), [-l](#), [-s](#), and the rest - see the *file test operators* section below).

The “operator” framing in classical Perl documentation is historical. In modern Perl terms, these are *functions whose argument-parsing places them between bitshift and comparison on the precedence table*. The naming is what it is; the behaviour is what matters.

File test operators

File tests are unary operators (named-unary precedence) that inspect a property of a file or filehandle. They begin with a hyphen and a single letter. All return boolean except [-s](#) (file size in bytes), [-M/-A/-C](#) (age in days, fractional), and the read-time tests [-T/-B](#) (boolean but slightly slower because they read the file).

Op	Test	Returns
-e	exists	boolean
-r	readable by effective UID	boolean
-w	writable by effective UID	boolean
-x	executable by effective UID	boolean
-o	owned by effective UID	boolean
-R	readable by real UID	boolean
-W	writable by real UID	boolean
-X	executable by real UID	boolean
-O	owned by real UID	boolean
-z	zero size	boolean
-s	non-zero size	size in bytes or false
-f	regular file	boolean
-d	directory	boolean
-l	symbolic link	boolean
-p	named pipe (FIFO)	boolean
-S	socket	boolean
-b	block special file	boolean
-c	character special file	boolean
-t	TTY (terminal)	boolean
-u	setuid bit	boolean
-g	setgid bit	boolean
-k	sticky bit	boolean
-T	text file (heuristic)	boolean
-B	binary file (heuristic)	boolean
-M	modification time (script-start age, days)	number
-A	access time (days)	number
-C	inode change time (days)	number

```

if (-e $file) { ... }           # exists
if (-f $path && -r $path) { ... } # regular file and readable
my $size = -s $file;           # size in bytes
if (-d $path) { ... }           # is a directory

```

Stacked file tests. `-f -w -r $file` is shorthand for `(-f $file) && (-w $file) && (-r $file)` - the chain re-uses the same target. The pseudo-handle `_` (single underscore) re-uses the *previous* file test's stat cache to avoid extra `stat()` syscalls:

```

if (-f $file && -r _) { ... }    # one stat() call, two checks

```

The trailing `_` is the standard idiom for any chain of file tests on the same path.

See also

- Every other page in this reference links into the rows of this table from its own *Precedence* section. The cross-reference is intentional - when you need to combine an operator with another, this table is the answer.
- [perlfunc index](#) - every name in the named-unary list is documented there.

- *Arithmetic* - + - * / % ** and the unary +/-.
- *String* - . (concatenation), x (repetition).
- *Numeric comparison* - < <= == >= > != <=>.
- *String comparison* - lt le eq ge gt ne cmp (lexicographic).
- *Logical* - && || // ! and or not xor and the short-circuit / operand-return rule.
- *Bitwise* - & | ^ ~ << >>.
- *Range* - .. and ... , with their list/scalar dual semantics.
- *Binding* - =~ and !~ connect strings with regex.
- *Assignment* - = and the compound forms (+= -= ||= /= etc.).
- *Ternary* - ?:.
- *Subscript* - [], {}, slices, deref braces.
- *Arrow* - -> (deref, method call).
- *Comma* - , and the fat comma =>.
- *Precedence* - the full precedence table and the named-unary parse rule that confuses everyone the first time.

Cross-cutting concepts

A few ideas appear in every chapter. Read these once and you will recognise them throughout the rest:

- **Context.** Operators evaluate their operands in a specific context (numeric, string, list, scalar, boolean). Many of the surprises in Perl arithmetic and comparison come from context coercion at the operand boundary, not from the operator itself.
- **Short-circuit.** &&, ||, //, and, or evaluate left-to-right and stop as soon as the result is determined. The whole expression's value is the last thing actually evaluated - which is one of its operands, not a normalised true/false. See *logical* and the *boolean-logic tutorial*.
- **Precedence.** Tells you how operators bind when parentheses do not. The full table is in *precedence* and every page links the rows it cares about.
- **Lvalue.** Most operators produce *rvalues* (values you can read). A few - assignment, deref, ternary in some shapes - produce *lvalues* (locations you can write to).

See also

- *Boolean Logic for Perl Programmers*
 - the conceptual companion to the *logical* and *bitwise* pages.
- *Regular expressions guide*
 - paired with *binding*.
- *perlfunc* - built-in functions, including every name listed in the *precedence* page's *named unary operators* section.

5.1.4 Data types

Perl has three fundamental data types - **scalars**, **arrays**, and **hashes** - plus a small number of supporting kinds (references, typeglobs, code) that are technically scalars but behave differently enough to deserve their own treatment. Every value the language manipulates is one of these.

This reference is split into one page per topic. Topics that share context (numeric vs string conversion, references vs typeglobs, context vs interpolation) live in the same page; topics that don't share context get their own page.

Choose a topic

Scalars

A scalar is the unit value of Perl. It holds one of: a number (integer or float), a string, a reference, or undef. The sigil is \$, in every position where the value is a single scalar - including when the value is one element of an array or hash:

```
$x           # the scalar named x
$arr[0]      # the first element of @arr - also a scalar
$h{key}      # one value from %h - also a scalar
```

A scalar is *not* statically typed. The same variable can hold a number now and a string a moment later. Perl converts between the two whenever an operator demands one shape or the other; see *numbers and strings*.

Truth, definedness, and undef

Two related but distinct questions about a scalar:

```
defined $x    # is there any value at all?
$x ? T : F    # is the value true?
```

A scalar is **false** in boolean context if it is one of:

```
undef        # the undefined value
0            # numeric zero
0.0          # also numeric zero
""           # empty string
"0"          # the literal string "0"
```

Everything else is true. Note "00", "0.0", "0 but true", and "0E0" are all *true* - they are non-empty strings that don't match the literal "0". The last two are common idioms for “this function returned successfully but there was nothing to count”:

```
sub touch_files {
    my $count = 0;
    $count++ for grep { -e $_ && utime undef, undef, $_ } @_;
    return $count || '0E0';    # true even on zero work, so callers
                              # can write `if (touch_files(...))`
}
```

defined is the only way to tell undef from the false-but-defined values. A common bug:

```
if ($cache{$key}) { ... }           # WRONG - false for empty/0/
↳undef
if (exists $cache{$key}) { ... }    # right - pure presence test
if (defined $cache{$key}) { ... }   # right - value-not-undef test
```

Use `defined` when the only thing you care about is “is there a value here at all”; `exists` for “is this key/index in the container” (the two differ for sparse arrays and deleted keys).

Dynamic typing

A scalar can change shape during execution:

```
my $x = 42;           # internally an integer
$x = "hello";         # now a string
$x = 3.14;            # now a float
$x = \@items;         # now a reference
```

Most code never needs to think about which internal shape a scalar currently has. The exceptions are JSON-serialisation libraries that care whether 1 was assigned as a number or a string, and the `Scalar::Util` introspection helpers (`looks_like_number`, `du-alvar`).

“Looks like a number”

When Perl converts a string to a number it stops at the first character that isn’t part of a valid numeric prefix:

```
"42"          + 1    # 43
"42abc"        + 1    # 43    - leading digits parsed
"abc42"        + 1    # 1     - no leading digits, treated as 0
" 42"         + 1    # 43    - leading whitespace allowed
"4_2"         + 1    # 5     - underscores in literals only, NOT in
↳strings
"0x1F"        + 1    # 1     - hex string is NOT auto-recognised;
↳hex() does
```

The mismatch between numeric *literals* (which honour `_` and `0x`) and numeric *strings* (which do not) is one of the most reported “why” questions in Perl. Use `hex` / `oct` when the input is a string in non-decimal form:

```
my $bytes = hex "1F";           # 31
my $perms = oct "0755";         # 493
my $perms = oct "0o755";        # 493 - explicit octal prefix
my $bytes = oct "0x1F";         # 31 - oct() accepts any prefix
```

Under use `warnings`, an arithmetic operation on a non-numeric string emits *Argument “...” isn’t numeric in addition (+)*. The operation still returns 0 - the warning, not a fatal error, is the signal.

Dual values

Some scalars carry both a string and a number form simultaneously, chosen by context. The classic example is `$!`:

```
$!           # in string context: "Permission denied"
$! + 0       # in numeric context: 13
```

You can build your own with `Scalar::Util::dualvar`:

```
use Scalar::Util qw(dualvar);
my $err = dualvar 13, 'Permission denied';

print "err is $err";           # "err is Permission denied"
print "err is " . ($err + 0);  # "err is 13"
```

Dual scalars are a niche tool. The common case where one matters is the `$!` `errno`; the general escape hatch is the `dualvar` helper.

“0 but true”

The literal string `"0 but true"` is the canonical Perl idiom for “a function whose normal return is a count, but the count is zero and the call succeeded”. It is true (non-empty, non-`"0"`) but numerically equals 0:

```
my $hits = "0 but true";

if ($hits) { print "succeeded\n"; } # prints
print $hits + 0;                     # 0
print "got $hits results\n";         # got 0 but true
↪results
```

The compiler also recognises `"0 but true"` specifically and skips the *isn't numeric* warning that any other “0 followed by garbage” string would emit.

See also

- *Numbers and strings* - how integer, float, and string forms convert into one another and where the corners are.
- *References* - `\X` produces a scalar that points to another value.
- *Context* - when a scalar gets one value vs many.
- `defined`, `undef` - test for and produce the undefined value.
- `scalar` - force scalar context on a list or array.
- `$_` - the implicit scalar for many built-ins.

Arrays

An array is an ordered, integer-indexed sequence of scalars. The sigil is @ for the whole array and for slices (which are lists); \$ for a single element (which is one scalar). The leading sigil tells you the *result type*, not the container.

```
my @days = ('Mon', 'Tue', 'Wed', 'Thu', 'Fri');

@days      # the whole array (5 elements)
$days[0]   # the first element - a single scalar
@days[1, 3] # a slice - a list of two scalars
$#days     # the last valid index - here, 4
scalar @days # the length - here, 5
```

Indices start at 0. Negative indices count from the end (\$days[-1] is 'Fri').

Length: \$#arr vs scalar @arr

Two related quantities, used for different jobs:

```
$#days      # last valid index      - 4
scalar @days # number of elements    - 5
$#days + 1  # equivalent to scalar @days
```

Assigning to \$#arr resizes the array. Shrinking destroys the elements past the new end; growing fills them with undef:

```
my @a = (1, 2, 3, 4, 5);
$a = 2;      # @a is now (1, 2, 3)          - length 3
$a = 5;      # @a is now (1, 2, 3, undef, undef, undef)
$a = -1;     # @a is now ()                 - empty
```

The @a = () form is the idiomatic empty-array assignment; \$#a = -1 is equivalent and occasionally clearer when the intent is “shrink to nothing without changing the variable’s identity”.

Element access

```
my $first = $days[0];      # 'Mon'
my $last  = $days[-1];     # 'Fri'
my $last  = $days[$#days]; # same; redundant but explicit
$days[2] = 'Wednesday';   # set
$days[10] = 'Sunday';      # auto-extends; $days[5..9] become
↪undef
```

Reading past the end returns undef (no warning - out-of-range read is a non-event). Writing past the end *grows* the array; the intervening slots become undef.

```
my @a;
$a[5] = 'fifth';
# @a is now (undef, undef, undef, undef, undef, 'fifth')
print scalar @a;      # 6
```

Reading a negative index further back than the start is a *fatal* error:

```
my @a = (1, 2, 3);
$a[-10];                                # FATAL: Modification of non-
    ↳creatable                           #      array value attempted
```

The whole-array operations

```
push    @arr, $value;                    # append at end                -
    ↳returns new length
my $v = pop @arr;                        # remove from end                -
    ↳returns removed element
my $v = shift @arr;                      # remove from start            -
    ↳returns removed element
unshift @arr, $value;                    # prepend                      -
    ↳returns new length

splice @arr, $offset, $len, @replacement; # general remove/insert
```

shift and unshift are $O(n)$ because they renumber every remaining element; push and pop are $O(1)$ amortised. For queue-shaped workloads where this matters, prefer push/shift on the *short* end or use a deque from `List::Util` (which has reductions, not a deque).

splice is the swiss-army knife: remove a stretch of elements, insert a stretch in their place, return what was removed:

```
my @arr = (1, 2, 3, 4, 5);
my @gone = splice @arr, 1, 2;             # @gone = (2, 3); @arr =
    ↳(1, 4, 5)
splice @arr, 1, 0, ('a', 'b');            # insert at offset 1, no
    ↳removal                               # @arr = (1, 'a', 'b', 4,
    ↳5)
```

A negative `$offset` counts from the end. A negative `$len` keeps that many trailing elements:

```
splice @arr, -1;                          # remove just the last element
splice @arr, 0, -2;                        # remove all but the last two
```

Slices

A slice extracts (or assigns to) several elements at once. Indices come from a list:

```
my @arr = ('a', 'b', 'c', 'd', 'e');

my @subset = @arr[0, 2, 4];               # ('a', 'c', 'e')
my @middle = @arr[1..3];                  # ('b', 'c', 'd')
```

(continues on next page)

(continued from previous page)

```

my @last3 = @arr[-3..-1];           # ('c', 'd', 'e')
my @rev   = @arr[reverse 0..$#arr]; # ('e', 'd', 'c', 'b', 'a')

@arr[1, 3] = ('B', 'D');             # set positions 1 and 3
@arr[1, 3] = @arr[3, 1];             # swap two elements (no temp)

```

A *key/value* slice with % returns index/value pairs (Perl 5.20+):

```

my @arr = ('a', 'b', 'c', 'd', 'e');
my %picked = %arr[1, 3];           # (1 => 'b', 3 => 'd')

```

This is the right shape when you need both the index and the value of selected elements. See *subscript* for the twelve-way matrix of slice forms.

List context vs scalar context

An array in list context yields its elements; in scalar context, it yields its length:

```

my @arr = (10, 20, 30);

my @copy = @arr;           # list context - three elements
my $length = @arr;         # scalar context - 3
print "got @arr\n";        # list context inside "" - "got
→ 10 20 30"
print "got " . @arr . "\n"; # scalar context (concat) - "got
→ 3"

```

The distinction is the most common Perl trap. Two rules of thumb:

- *Inside double-quoted strings*, arrays interpolate as their elements joined by \$" (a single space by default). See *interpolation*.
- *In a numeric or string operator context* (+, ., comparison), an array yields its length.

Force scalar context with *scalar* when the expression position is otherwise ambiguous:

```

print "got " . scalar(@arr) . " items\n"; # "got 3 items"
print "got @arr items\n";                 # "got 10 20 30 items"
→ - different!

```

List assignment and the count idiom

Assigning a list to an array makes the array exactly that long:

```

my @arr = (1, 2, 3);
@arr = (10, 20);           # @arr is now (10, 20)
@arr = ();                 # @arr is now empty

```

Assigning a list to a *list* of scalars unpacks (any extras are discarded; missing ones become undef):

```

my ($x, $y, $z) = (1, 2, 3);           # 1, 2, 3
my ($x, $y, $z) = (1, 2);             # 1, 2, undef
my ($x, $y)      = (1, 2, 3);         # 1, 2 - '3' silently
↳dropped

```

A list assignment in scalar context returns the number of elements on the *right* side. This is the source of the count idiom:

```

my $count = () = $string =~ /\d+/g;
# $string =~ /\d+/g - match in list context, returns all matches
# () = LIST         - list assignment to empty list
# $count = SCALAR   - that assignment in scalar context = count of
↳RHS

```

The `() = LIST` middle is the trick. Without it, `$count = $s =~ /\d+/g` gives a boolean instead of a count.

Array references

Arrays are not first-class values in the sense that they can be passed by name through a chain of function calls without flattening into a list. To pass an array around without flattening, take a **reference** to it:

```

my @items = (1, 2, 3);

my $aref = \@items;           # reference to the existing @items
my $anon = [1, 2, 3];         # anonymous array reference

$aref->[0]                     # access through the arrow - 1
@{$aref}                      # fully bracketed deref - list
↳(1, 2, 3)
@{$aref}                      # unbracketed deref - same as
↳above
${$aref}                      # last index of dereffed array
scalar @{$aref}               # length

```

See *references* for the full picture.

Real example: building a histogram of word lengths

```

my @words = qw(the quick brown fox jumps over the lazy dog);

my @hist;
$hist[length $_]++ for @words;

for my $len (1 .. $#hist) {
    next unless $hist[$len];
    printf "%2d-letter words: %d\n", $len, $hist[$len];
}
# 3-letter words: 4

```

(continues on next page)

(continued from previous page)

```
# 4-letter words: 2
# 5-letter words: 3
```

Note `$hist[length $_]++` autovivifies (creates) the slot when it's the first word of that length. Reading from `$hist[7]` later returns `undef`, which `++` happily increments - but next unless `$hist[$len]` skips the gap before printing.

See also

- *Hashes* - when the index is a string, not an integer.
- *References* - passing arrays around without flattening.
- *Context* - the list-vs-scalar story expanded.
- *Interpolation* - how arrays appear inside `" "`.
- *Subscript and slice operators* - the bracket-and-brace family in detail.
- *Range operator* `..` - the most common source of slice index lists.
- `push`, `pop`, `shift`, `unshift`, `splice` - the whole-array operations.
- `scalar` - force scalar context (`length`).
- `reverse`, `sort` - whole-array transformations.

Hashes

A hash is an unordered collection of key-value pairs where keys are strings and values are scalars. The sigil is `%` for the whole hash, `@` for value slices and key/value slices, and `$` for one value:

```
my %user = (
    name => 'John',
    age  => 30,
    email => 'john@example.com',
);

%user          # the whole hash (six elements: key1, val1, ...
↪)
$user{name}    # one value           - 'John'
@user{qw(name age)} # value slice    - ('John', 30)
%user{qw(name age)} # key/value slice - (name => 'John', age =>
↪30)
keys %user      # the list of keys
```

Hash keys are always coerced to strings. Storing under the integer key 42 and reading under the string key `"42"` retrieve the same slot - they are the same key.

Initialisation: pairs, fat comma, and %h = LIST

A hash is initialised from a list of even length, in key/value order:

```
my %h = ('a', 1, 'b', 2);           # legal but ugly
my %h = (a => 1, b => 2);           # idiomatic
```

The fat comma => is a comma that *also* auto-quotes a left-hand bareword that looks like an identifier. So `a => 1` is exactly `'a', 1`. Auto-quoting requires the bareword to be a simple identifier - `2.0 => 'x'` is parsed as the number 2, not the string `"2.0"`:

```
my %h = (a => 1);                   # ('a', 1)           - auto-quoted
my %h = ('a' => 1);                 # ('a', 1)           - same
my %h = (2.0 => 'x');               # (2, 'x')           - not ('2.0', 'x'
↳ ')!
my %h = ("2.0" => 'x');             # ('2.0', 'x')       - explicit quote
```

If a key appears more than once in the initialiser list, the **last occurrence wins**. The standard idiom for “merge with overrides” exploits that:

```
my %config = (%defaults, %overrides);
# %config has every default key, with %overrides values where they
↳ collide
```

Access, with the four sigil shapes

Reading and writing one slot is \$-sigil:

```
my $name = $user{name};           # read
$user{city} = 'NYC';              # write
$user{age}++;                     # arithmetic on a hash value
delete $user{email};              # remove the slot entirely
```

The other three subscript shapes match what you want back:

```
@user{qw(name age city)}          # ('John', 31, 'NYC')
↳ - values
%user{qw(name age)}                # (name => 'John', age => 31)
↳ - pairs
keys %user                         # ('name', 'age', 'city')
↳ - keys
values %user                       # ('John', 31, 'NYC')
↳ - values
```

See *subscript* for the whole twelve-way matrix and the bareword-auto-quote rule inside {}.

exists vs defined vs truthiness

Three distinct questions about a key:

```

exists $h{key}      # is the slot present at all?
defined $h{key}     # is the slot present AND non-undef?
                    $h{key}      # is the value true (non-empty, non-zero, non-
→ "0")?

```

These differ when:

```

$h{a} = undef;
$h{b} = 0;
delete $h{c};

exists $h{a}        # TRUE  - slot exists, value is undef
defined $h{a}       # FALSE - value is undef
                    $h{a}        # FALSE - undef is false

exists $h{b}        # TRUE
defined $h{b}       # TRUE
                    $h{b}        # FALSE - 0 is false

exists $h{c}        # FALSE - never set, or deleted
defined $h{c}       # FALSE
                    $h{c}        # FALSE

```

Pick by intent: *exists* for *presence*, *defined* for *value-not-undef*, *truthiness* for *meaningful value*. Reaching for `if ($h{key})` when *exists* was meant is a bug source - `0` and `" "` are valid values that the test rejects.

Iteration: keys, values, each, while (each)

```

for my $k (keys %user) {
    print "$k = $user{$k}\n";
}

for my $v (values %user) { ... }           # values only

while (my ($k, $v) = each %user) {         # keys + values, one_
→pair per call
    print "$k = $v\n";
}

```

`each` carries an iterator state on the hash itself; calling `keys` (or `values`) on the hash resets that iterator. Mixing the two produces hard-to-debug looping bugs:

```

while (my ($k, $v) = each %h) {
    if (some_condition($k)) {
        print "size: ", scalar keys %h, "\n";    # resets each()_
→iterator!
    }
}

```

(continues on next page)

(continued from previous page)

```
    # next iteration of while() starts over from the top
  }
}
```

Fix: either accumulate the list with keys once at the top, or avoid each entirely. Most hash iteration is best written with keys:

```
for my $k (sort keys %h) {           # bonus: defined ordering
    ...
}
```

Key ordering

Hash keys come out in **insertion-order-perturbed-by-hash-randomisation** order - i.e. there is no guaranteed order. Two runs of the same program with the same input may iterate keys in different orders. This is a security feature (it prevents algorithmic-complexity attacks against the hash function); it is also a recurring source of brittle tests:

```
my %h = (a => 1, b => 2, c => 3);
print "$_ " for keys %h;           # output order is undefined
```

If you need a stable order, sort:

```
print "$_ " for sort keys %h;       # alphabetical
print "$_ " for sort { $h{$a} <=> $h{$b} } keys %h; # by value
```

Hash references

A hash, like an array, flattens when passed through a list. To pass a hash without flattening, take a **reference**:

```
my %h = (a => 1, b => 2);

my $href = \%h;           # reference to the existing %h
my $anon = { x => 1, y => 2 }; # anonymous hash reference

$href->{a}                 # access through the arrow      - 1
${$href}{a}               # fully bracketed deref        - 1
keys %$href               # deref then keys
%{$href}                  # deref to flat key/value list
```

See *references* for the full picture.

Real example: counting and grouping

A frequency count is the textbook small example for hashes:

```
my @words = qw(apple banana apple cherry apple banana);
my %count;
$count{$_}++ for @words;
# %count = (apple => 3, banana => 2, cherry => 1)
```

Grouping is the next step up - building a hash whose values are arrayrefs:

```
my @people = (
    { name => 'Alice', dept => 'eng' },
    { name => 'Bob',   dept => 'sales' },
    { name => 'Carol', dept => 'eng' },
    { name => 'Dan',   dept => 'sales' },
);

my %by_dept;
push @{ $by_dept{$_->{dept}} }, $_->{name} for @people;
# %by_dept = (eng => ['Alice', 'Carol'], sales => ['Bob', 'Dan'])
```

The `push @{ $by_dept{$_->{dept}} }, ...` line autovivifies: when the `eng` key isn't there yet, the hash slot is created and an anonymous array goes into it, then the value is pushed. See [references](#) for the autoviv story.

Hash slice as multi-value get/set

```
my %config = (host => 'localhost', port => 80, debug => 0);

my ($h, $p) = @config{'host', 'port'};      # ('localhost', 80)

@config{qw(host port debug)} = ('elsewhere', 8080, 1); # set three
↪at once
```

Combine with the `%h{...}` slice for “extract a record”:

```
my %fields = %config{qw(host port)};
# %fields = (host => 'localhost', port => 80)
```

This is the canonical way to project a hash down to a subset of its keys. Without `%h{...}` you'd hand-roll it with a loop or with `map`.

See also

- [Arrays](#) - when the index is an integer.
- [References](#) - autovivification, hash-of-arrays, hash-of-hashes; the `$h->{a}{b}` shape.
- [Subscript and slice operators](#) - the bareword-auto-quote rule and the four slice shapes.
- `exists`, `defined`, `delete` - the presence/value/removal trio.
- `keys`, `values`, `each` - whole-hash iteration.

- `sort` - for ordered iteration.
- `tie` - make a hash backed by something other than the in-memory hash table (DBM file, lazy database, ...).

Context

Almost every operator in Perl evaluates its operands in a specific **context** - list, scalar, void, or boolean - and many operators return *different things* depending on the context they were called in. The same expression can mean different things in different positions:

```
my @arr = (10, 20, 30);

my @copy = @arr;           # list context      - three elements
my $len  = @arr;           # scalar context - 3
print @arr;                # list context      - "102030"
print "@arr";              # list context inside " " - "10 20 30"
print "" . @arr;           # scalar context (concat) - "3"
@arr;                      # void context - discarded; warns under -w
```

Context is the single biggest difference between Perl and most other languages, and the single most common source of “why doesn’t this do what I expect” surprises.

The four contexts

Context	What the operand sees	Imposed by
list	“give me all of your values”	@a = ..., (...), function args, print
scalar	“give me one value”	\$s = ..., if/while, ., +, comparison
void	“throw the result away”	a statement whose value isn’t used
boolean	“give me a true/false”	if, unless, while, &&, , !

Boolean context is technically a special case of scalar context - the operand is asked for one value, then that value is tested for truth. The two are not always the same: a wantarray-aware sub can detect boolean context separately (see below).

Imposing context

The left side of an assignment imposes context on the right:

```
my @x = func();           # func() is called in LIST context
my $x = func();           # func() is called in SCALAR context
my ($x, $y) = func();     # func() is called in LIST context
my ($x) = func();         # also LIST - note the parens
my $x = func();           # SCALAR - same chars, very different
```

The single most-tripped-over case is `my $x = func()` vs `my ($x) = func()`. Without parens, the assignment is a scalar; with them, it’s a 1-element list assignment. If `func()` returns three things in list context and “the count” in scalar context, the difference is between getting ‘first-thing’ and getting 3.

Operators impose context too. Numeric and string operators impose scalar context on their operands; print imposes list:

```
my @arr = (1, 2, 3);
my $n = @arr + 0;      # scalar context - 3 + 0 = 3
my $s = @arr . "";     # scalar context - "3"
print @arr;            # list context - prints "123"
```

And the boolean tests impose scalar (boolean) context:

```
if (@arr)      { ... } # true if @arr is non-empty
unless (%hash) { ... } # true if %hash is empty
while (<$fh>)  { ... } # context is BOOLEAN, but the diamond op
→is                                     # special: it reads ONE line and assigns
→to $_
```

What an array yields in scalar context

An **array** in scalar context yields its **length**:

```
my @arr = (1, 2, 3);
my $n = @arr;      # 3
```

A **list literal** in scalar context yields its **last element** (the comma operator):

```
my $x = (1, 2, 3);      # 3 - comma operator, last element
my $x = ('a', 'b', 'c'); # 'c'
```

These look identical and behave differently. The rule is: arrays have a length, list literals don't. A bare comma-list is the C comma operator and produces its last operand; an array variable in scalar position is asking "how many":

```
my @a = ('x', 'y', 'z');
my $count = @a;      # 3
my $last_el = $a[-1]; # 'z'
```

The scalar operator forces scalar context on whatever follows, which is most useful when the expression position is otherwise ambiguous:

```
print "got " . scalar(@arr) . " items\n"; # "got 3 items"
print "got " . @arr . " items\n";        # same - concat
→imposes scalar
print "got @arr items\n";                 # "got 1 2 3 items" -
→list inside ""
```

What a hash yields in scalar context

A non-empty hash in scalar context returns a true value (the count of keys, since Perl 5.25); an empty hash returns the false value 0. The shape that matters is the truth value:

```

if (%h) { ... }           # true if %h has any pairs
my $n = keys %h;          # explicit: count of keys
my $n = %h;               # also count of keys (Perl 5.25+); was a_
↳ debug ratio earlier

```

For older code targeting pre-5.25 perls you sometimes still see scalar keys %h; on PetaPerl (target perl 5.42) scalar %h returns the key count and is fine.

wantarray - write context-sensitive subs

A sub can ask “what context was I called in?” with wantarray:

```

sub items {
    return wantarray ? (1, 2, 3)      # list context
        : defined wantarray ? 3      # scalar context
        : do { warn "items() called in void context\n"; () }; #_
↳ void context
}

my @list = items();                # (1, 2, 3)
my $cnt  = items();                # 3
items();                           # void - warning fires

```

The three-way distinction is the only way to detect *void* from inside a sub. Use it sparingly: most subs return one shape and let the caller worry about context coercion. The places where wantarray earns its keep are:

- Functions that legitimately return either “the list” or “the count” - e.g. text-search routines.
- Diagnostic helpers that print a warning when their result was discarded.

A sub that returns different *meanings* (not just different shapes) based on wantarray is a maintenance hazard. Two named functions read better than one polymorphic one.

Pitfall: scalar(@arr) vs @arr in different positions

```

my @arr = (1, 2, 3);

# 1. Function arguments - LIST context
some_func(@arr);      # passes 1, 2, 3 as three arguments
some_func(scalar @arr); # passes 3 as a single argument

# 2. String concatenation - SCALAR context
"got " . @arr          # "got 3"

# 3. Inside double quotes - array interpolation, NOT scalar!
"got @arr"              # "got 1 2 3"

# 4. Boolean test - SCALAR (boolean) context
if (@arr) { ... }      # true iff @arr non-empty

```

(continues on next page)

(continued from previous page)

```
# 5. Comparison - SCALAR context on both sides
@arr == 3           # 3 == 3 is true
@arr eq "3"         # "3" eq "3" is true; this is rarely what
                    ↪ you want
```

The mismatch between case 2 (concat → scalar) and case 3 (inside " → list) is the single most-asked Perl context question. Inside "", @arr is array interpolation; outside "", the context of the surrounding expression decides.

Pitfall: list-flattening in function arguments

A function receives @_ as a single flat list of all its arguments - there is no way for the callee to tell where one array ended and the next began:

```
sub many {
    print "got ", scalar @_, " args\n";
}

my @a = (1, 2);
my @b = (3, 4, 5);

many(@a, @b);           # 5 args - both arrays flattened
many(\@a, \@b);         # 2 args - two array references
many(scalar @a, scalar @b); # 2 args - two integers (the lengths)
```

This is *the* reason references exist: to pass a non-flat shape through the list-flattening calling convention. Anything you want to keep its identity inside @_ must arrive as a reference.

Real example: a sub that returns count or list

```
sub digits_of {
    my ($s) = @_;
    return $s =~ /\d/g;    # in list context: all digit chars
                          # in scalar context: TRUE/FALSE (last
    ↪ match)
}

my @digits = digits_of("a1b2c3");    # ('1', '2', '3') - three
    ↪ matches
my $any     = digits_of("a1b2c3");    # 1
    ↪ boolean
```

To make digits_of return a *count* in scalar context, the right-hand side has to coerce explicitly:

```
sub digits_of {
    my ($s) = @_;
```

(continues on next page)

(continued from previous page)

```
my @d = $s =~ /\d/g;
return wantarray ? @d : scalar @d;
}

my @digits = digits_of("a1b2c3");      # ('1', '2', '3')
my $n      = digits_of("a1b2c3");      # 3
```

This is the canonical “return a list or its length” idiom. Note that without the `wantarray` switch, `return @d` would still give the count in scalar context (because of the array-in-scalar rule)

- but it’s clearer to be explicit.

See also

- [Arrays](#) - the array-vs-list-vs-scalar trichotomy expanded.
- [Hashes](#) - `%h` in scalar context.
- [Comma](#) - the operator that builds list literals and yields its last operand in scalar context.
- [wantarray](#) - the in-sub way to ask.
- [scalar](#) - force scalar context.
- [reverse](#) - context-sensitive: a list of scalars in list context, a single reversed string in scalar context.
- [References](#) - passing a non-flat shape through the flattening list-context arg convention.

Numbers and strings

A scalar can hold a number, a string, or both at once. Perl silently converts between the two whenever an operator demands one shape - `+` wants numbers, `.` wants strings, comparison operators have separate numeric and string forms. Understanding the conversion rules and where they bite is essential to writing predictable Perl.

Numeric literals

Integer literals come in five bases:

```
0                # zero
12345           # decimal
4_294_967_296    # underscores for legibility (decimal 4.29
↳billion)
0xff            # hex          - 255
0Xdead_beef     # hex with underscores
0b01_1011       # binary      - 27
0377            # octal       - 255 (leading 0, NOT followed by
↳[bBxX])
0o12_345        # octal, alternative spelling (5.33.5+)
```

Float literals require either a dot or an exponent:

```
3.14
0.5          # leading zero ok
.5           # leading dot ok
1e3          # 1000
1.5E-10      # exponent - uppercase or lowercase 'e'
```

Underscores are accepted between digits anywhere - 3.14_15_92, 1_000_000. Multiple consecutive underscores warn: 23__500 warns, 23_500 does not.

Hex and octal floats need the p (power-of-2) exponent:

```
0x1.999ap-4  # hex float
0b101.01p-1  # binary float
0o7.65p2     # octal float
```

The p exponent is *required* for non-decimal floats. Plain 0x1F.5 is a parse error.

String → number

When Perl converts a string to a number, it consumes the longest prefix that looks like a number and ignores the rest:

```
"42"          + 0      # 42
"42abc"       + 0      # 42      - leading digits parsed
"abc42"       + 0      # 0       - no leading digits
" \t 42"      + 0      # 42      - leading whitespace ok
"4_2"         + 0      # 4       - underscores in literals only, NOT
↳strings
"0xFF"        + 0      # 0       - hex prefix NOT auto-recognised
"3.14e2"      + 0      # 314     - scientific notation parsed
"Inf"         + 0      # Inf     - case-insensitive infinity
"NaN"         + 0      # NaN
```

The mismatch between numeric *literals* (which honour `_`, `0x`, `0b`, `0`) and numeric *strings* (which honour none of those) is the single most-tripped-over conversion rule. To convert a string that's in non-decimal form, use `hex` or `oct`:

```
my $bytes = hex "FF";          # 255
my $bytes = hex "0xFF";        # 255 - hex() ignores the 0x
my $perms = oct "0755";        # 493
my $perms = oct "0o755";       # 493 - explicit octal prefix
my $bytes = oct "0xFF";        # 255 - oct() recognises any prefix
```

Under `use warnings`, an arithmetic operation on a string that doesn't *start* with a number prefix emits *Argument "... isn't numeric in addition (+)*. The result is still 0; the warning is the signal. Writing tolerant code that accepts user input often means filtering with `Scalar::Util::looks_like_number` or a regex first.

Number → string

Perl converts a number to a string using the same logic as `sprintf("%g", $n)` for floats and the obvious decimal representation for integers:

```

my $i = 42;           "$i"    # "42"
my $f = 3.14;         "$f"    # "3.14"
my $z = 0.0;          "$z"    # "0"           - not "0.0"
my $b = 1e10;         "$b"    # "10000000000"
my $s = 1e20;         "$s"    # "1e+20"       - switches to scientific
→past ~15 digits
my $n = "Inf" + 0;    "$n"    # "Inf"

```

The default precision is enough digits to roundtrip *most* doubles. For exact control, format explicitly with `sprintf`:

```

sprintf "%.2f", 3.14159      # "3.14"
sprintf "%05d", 42          # "00042"
sprintf "%e", 1234567       # "1.234567e+06"

```

Concatenation forces strings; arithmetic forces numbers

```

"42" + "3"              # 45      - both coerced to numbers
42 . 3                  # "423"    - both coerced to strings
"3" x "5"               # "33333" - left operand string, right operand
→number

```

The `.` operator is the explicit-stringify operator; `+` is the explicit-numerify operator. A scalar that holds a number you want as a string can be concatenated with `"`:

```

my $s = "" . $n;        # idiomatic "stringify"
my $s = "$n";           # same; usually clearer

```

A scalar that holds a string you want as a number adds 0:

```

my $n = $s + 0;         # idiomatic "numify"

```

Numeric vs string comparison

Two parallel families of comparison operators with **different semantics**:

Numeric	String	Meaning
<code>==</code>	<code>eq</code>	equal
<code>!=</code>	<code>ne</code>	not equal
<code><</code>	<code>lt</code>	less
<code><=</code>	<code>le</code>	less or equal
<code>></code>	<code>gt</code>	greater
<code>>=</code>	<code>ge</code>	greater or equal
<code><=></code>	<code>cmp</code>	three-way compare

The sort order is dramatically different on values that look like numbers but are large:

```
"10" < "9"          # FALSE - 10 < 9 numerically
"10" lt "9"         # TRUE  - '1' < '9' lexicographically
```

Sort numbers numerically:

```
my @sorted = sort { $a <=> $b } @numbers;    # numeric
my @sorted = sort @strings;                  # default - string
my @sorted = sort { $a cmp $b } @strings;     # explicit string
```

Picking the wrong operator is the source of “my sort is wrong” bugs. See *numeric comparison* and *string comparison*.

Floating-point: precision, Inf, NaN

Most Perls use IEEE 754 double-precision floats - about 15–17 decimal digits of precision. The classic surprise:

```
0.1 + 0.2 == 0.3      # FALSE - neither side is exactly
↳representable         # 0.1 + 0.2 = 0.30000000000000004
```

For money or anything that *must* be exact, work in integers (cents instead of dollars) or use a bignum module:

```
use bignum;           # auto-promotes to arbitrary precision
0.1 + 0.2 == 0.3      # TRUE under bignum

use Math::BigInt;
my $big = Math::BigInt->new("10000000000000000000") + 1;
```

Inf and NaN are real values:

```
my $inf = 9 ** 9 ** 9;    # Inf - overflow
my $nan = $inf - $inf;    # NaN - undefined arithmetic
my $nan = "NaN" + 0;     # NaN - accepted from string

$nan == $nan              # FALSE - NaN never equals anything
$nan != $nan              # TRUE  - including itself
```

Inf and NaN are case-insensitive on input and have several spellings (Infinity, 1.#INF, ...); on output Perl normalises to the short forms.

Note that perl5 generates *fatal* errors for `1/0` and `sqrt(-1)`; those don’t produce NaN. They are exceptional, not arithmetic.

Locale-affected number parsing

Under `use locale` plus an active POSIX locale, the decimal point is whatever the locale says. This affects parsing and stringifying of floats:

```
use locale;
use POSIX qw(setlocale LC_NUMERIC);
setlocale LC_NUMERIC, 'de_DE.UTF-8';    # German uses ',' as decimal

my $x = 3.14;
print "$x\n";                          # "3,14" - locale
    ↪ formatting
"3,14" + 0                             # 3.14 - locale parsing
```

This is rarely what you want for data interchange. Most code that reads or writes structured data should explicitly *not* be in the `use locale` scope, so that `.` is always the decimal point. Use locale formatting only at the boundary where output is for a human reader.

“Looks like a number”

The rules for whether a string is “numeric enough” are not simple to inline; use `Scalar::Util::looks_like_number`:

```
use Scalar::Util qw(looks_like_number);

looks_like_number("42")           # true
looks_like_number("3.14e10")      # true
looks_like_number("Inf")          # true
looks_like_number("NaN")          # true
looks_like_number("0xFF")         # FALSE - hex strings are not
    ↪ "numeric"
looks_like_number("")             # FALSE - empty string
looks_like_number(undef)          # FALSE
looks_like_number("3.14abc")      # FALSE - trailing junk
```

This is the validator function - strict, no leading garbage, no trailing garbage, no hex/oct prefixes. It matches the conditions under which Perl will use a number’s cached numeric form rather than re-parsing on every arithmetic op.

See also

- *Scalars* - the dual-form / dualvar story; truth values.
- *Numeric comparison* - `==`, `<=>`.
- *String comparison* - `eq`, `cmp`.
- *Arithmetic* - `+`, `-`, `*`, `/`, `%`, `**`.
- *String* - `.` (concat), `x` (repetition).
- `hex`, `oct` - convert non-decimal numeric strings.
- `int`, `abs`, `sprintf` - explicit numeric conversion and formatting.
- `Scalar::Util` - `looks_like_number`, `dualvar`.

References

A reference is a scalar that *points to* another value. References are how Perl builds nested data structures - arrays of arrays, hash of hashes, hash of arrays of hashes - because the basic container types (arrays, hashes) flatten when they appear in lists, and a reference is a single scalar that survives the flattening.

```
my @arr = (1, 2, 3);
my $ref = \@arr;           # take a reference to @arr
$$ref[0]                   # full deref form      - 1
$ref->[0]                   # arrow shorthand     - 1
@$ref                      # whole-array deref   - (1, 2, 3)
```

A reference is itself a scalar. It can be passed through any list-flattening calling convention without losing its identity, stored in arrays and hashes, returned from subs, weakened, serialised - anywhere a scalar can go.

Things you can reference

Seven things take a reference:

```
\$scalar      # SCALAR ref
\@array       # ARRAY ref
\%hash        # HASH ref
\&subroutine  # CODE ref
\*glob        # GLOB ref
\\$scalar     # REF (a reference to a reference)
\\$file_handle # IO ref (reference to a filehandle)
```

Plus the **anonymous** constructors, which build the value and return a reference in one step:

```
my $aref = [1, 2, 3];           # anonymous array
my $href = { name => 'Alice', age => 30 }; # anonymous hash
my $cref = sub { $_[0] * 2 };   # anonymous sub
```

The bracket-vs-paren distinction is essential: (1, 2, 3) is a **list**, [1, 2, 3] is an **anonymous array reference**. The first flattens; the second does not.

Dereferencing

Four spellings, in increasing order of how often they appear in modern code:

```
${$sref}      # full sigil-deref form
$$sref        # short sigil-deref form (only if name is simple)
$->simple)
$sref->{key}   # arrow form (postfix; for arrays and hashes)
$sref->[0]     # same, for arrays
```

The -> arrow form is the idiomatic spelling for nested access:

```
$config->{database}{host}           # arrow elision between
→subscripts
$config->{database}->{host}         # explicit; same meaning
${$config->{database}}{host}       # full form; the same again
```

Between two consecutive subscripts the `->` is **optional** (the “arrow elision” rule); at the head of a chain it is required:

```
$ref->[0]                           # required arrow at the head
$ref->[0][1]                         # second arrow elided
$ref->[0]->[1]                      # equivalent
[1,2,3]->[0]                       # required arrow on an anonymous ref
```

Whole-container deref:

```
@$aref                             # the whole array
%$href                             # the whole hash, as flat key/value
→list
&$cref(@args)                     # call the sub
$cref->(@args)                    # idiomatic call form
```

Use the curly-braced form `@{ EXPR } / %{ EXPR }` when the deref target is an expression more complex than a bare scalar variable:

```
my @found = @{ $config->{users} }; # deref a chain - needs braces
```

See *arrow* for the operator and *subscript* for the bracket family.

Precedence of prefix deref versus subscript

The five prefix dereference sigils `$ @ * % &` bind **tighter** than the postfix subscript brackets `[]` and `{}`. So these two are exactly the same expression:

```
$$aref[2][2]                       # confusing, but valid
$aref->[2][2]                       # clear; identical meaning
```

`$$aref[2][2]` dereferences `$aref` *first*, then subscripts the result. It is **not** the C reading `*a[i]`, where the subscript would apply before the dereference. Perl has no construct that subscripts before dereferencing at that position; if you want “the thing the *i*-th element points at”, spell it out left to right with a postfix deref:

```
$aref->[$i]->$*
```

The arrow form is the readable choice for nested access; the leading-sigil form is a trap precisely because it reads as the C precedence and means the opposite. See *precedence* for the full operator table.

What ref returns

`ref` reports the kind of thing a reference points to:

```

ref \$scalar      # 'SCALAR'
ref \@array      # 'ARRAY'
ref \%hash       # 'HASH'
ref \&sub        # 'CODE'
ref \*FH         # 'GLOB'
ref \\$scalar    # 'REF'      - a ref-to-ref
ref \$io_handle  # 'IO'      - wrapper around a file handle

ref [1, 2, 3]    # 'ARRAY'
ref { a => 1 }   # 'HASH'
ref sub {}       # 'CODE'

ref bless {}, 'Foo' # 'Foo'    - blessed; class name not 'HASH'
ref 42           # ''        - not a reference at all
ref undef        # ''

```

For blessed objects, `ref` returns the class. To distinguish a blessed hash-ref from a plain hash-ref, use `Scalar::Util::reftype` (which always returns the underlying container shape regardless of `bless`) or `isa` (the inheritance-aware check).

Postfix dereference (@*, %*, \$*)

Perl 5.20+ added a postfix syntax that mirrors the arrow chain:

```

$ref->@*          # equivalent to @$ref
$ref->%*          # equivalent to %$ref
$ref->$*          # equivalent to $$ref

$ref->@[1, 3]      # array slice through ref - equivalent to @{$ref}[1,3]
$ref->%[1, 3]     # key/value slice
$ref->@{qw(a b)}  # hash slice

```

The postfix forms read left-to-right with the rest of an arrow chain, which is often clearer when the deref target is itself a chain:

```

$config->{users}->@*      # all users - chains naturally
@{$config->{users}}      # same; reads right-to-left

```

Pick the form that reads better in context. Postfix is the clearer choice in chains; the leading-sigil form is shorter for plain scalar refs.

Autovivification

Reading or writing through a deref into a slot that doesn't exist yet auto-creates the intermediate structure:

```
my %h;
$h{a}{b}{c} = 1;
# %h is now (a => { b => { c => 1 } }) - three nested hashes built
```

The mechanism: when the assignment target is `$h{a}{b}{c}`, Perl sees `$h{a}` is undef, the next subscript is `{b}` (a hash subscript), so it makes `$h{a}` a fresh hash ref. Then `$h{a}{b}` is undef, next subscript is `{c}` (also hash), so make it a fresh hash ref. Then assign 1 to the leaf.

The same happens for array subscripts:

```
my %by_dept;
push @{ $by_dept{eng} }, 'Alice';
# $by_dept{eng} did not exist; the push autoviv'd it as an arrayref
```

Autoviv on the **left** of an assignment (or a push/pop/... mutator) is what makes this work. Autoviv on the **right** of an expression is more dangerous - *reading* `$h{a}{b}{c}` when `$h{a}` is undef *also* autovivifies, populating the hash as a side effect of inspection:

```
my %h;
exists $h{a}{b};           # accidentally creates $h{a} = {} as a_
    ↪ hash!
                           # %h is now (a => {})
```

The fix when you genuinely just want to test: chain exists:

```
exists $h{a} && exists $h{a}{b};           # short-circuits if $h{a}_
    ↪ absent
```

This is the most-bitten autoviv trap. Tests of nested existence must short-circuit, or they leave hash slots behind.

Hash of arrays - the canonical example

```
my @people = (
    { name => 'Alice', dept => 'eng' },
    { name => 'Bob',   dept => 'sales' },
    { name => 'Carol', dept => 'eng' },
    { name => 'Dan',   dept => 'sales' },
);

# Group by department:
my %by_dept;
for my $p (@people) {
    push @{ $by_dept{ $p->{dept} } }, $p->{name};
}
# %by_dept = (
#     eng => ['Alice', 'Carol'],
#     sales => ['Bob', 'Dan'],
# )
```

(continues on next page)

(continued from previous page)

```
# Iterate:
for my $dept (sort keys %by_dept) {
    print "$dept: @{ $by_dept{$dept} }\n";
}
# eng: Alice Carol
# sales: Bob Dan
```

The `@{ $by_dept{...} }` is the deref-with-braces form. It's necessary because the deref target is an expression (`$by_dept{$dept}`), not a bare variable.

The same shape with the postfix form:

```
print "$dept: ", join(' ', $by_dept{$dept}->@*), "\n";
```

Pick whichever you find clearer; both are idiomatic in modern Perl.

Weak references and circular structures

A reference holds the referent alive (refcount-bumped) until the reference itself goes out of scope. This becomes a problem when two structures point at each other:

```
my $parent = { name => 'p' };
my $child  = { name => 'c', parent => $parent };
push @{ $parent->{children} }, $child;

# parent refers to child via the children arrayref
# child refers back to parent via the parent slot
# When both go out of scope, neither's refcount can reach 0 - leak.
```

The fix is a **weak reference** - a reference that doesn't bump the refcount. The `$child->{parent}` slot is the one that should be weakened (the back-pointer):

```
use Scalar::Util qw(weaken);

my $parent = { name => 'p' };
my $child  = { name => 'c', parent => $parent };
weaken $child->{parent};           # break the cycle
push @{ $parent->{children} }, $child;
```

After weakening, `$child->{parent}` reads as the parent while it exists, and silently becomes undef when the parent is freed. The canonical place for `weaken` is on every back-pointer in a tree structure where children point back at their parent.

The `Scalar::Util::isweak` predicate tests whether a given reference is weakened.

Reference equality

Two references are equal (numerically `==`, stringwise `eq`) if and only if they refer to the **same** underlying value:

```
my @a = (1, 2, 3);
my $r1 = \@a;
my $r2 = \@a;
my $r3 = [1, 2, 3];      # different anonymous array

$r1 == $r2               # TRUE - same address
$r1 == $r3               # FALSE - different array, even though
↳ contents match
```

Numeric comparison of refs effectively compares their machine address. Two refs with the same content but different identities compare unequal. Deep-equal needs a helper from `Test::More` or a hand-rolled walker.

A reference stringifies to something like `ARRAY(0x55ab12cd)` - class name plus hex address. Don't rely on the format; use it only as a debug print.

See also

- *Arrays, Hashes* - the two containers most references point at.
- *Typeglobs* - `*name`, the symbol-table entry that references-to-globs let you alias.
- *Arrow operator* - `->`, the deref shorthand.
- *Subscript* - `[]` and `{}` through a ref.
- *ref* - what kind of thing does this point at.
- *bless* - make a reference into an object.
- *Scalar::Util* - `weaken`, `isweak`, `reftype`, `blessed`, `looks_like_number`.
- *References tutorial* - step-by-step companion to this terse reference.
- *Weak refs tutorial* - the cycle problem in depth.

Typeglobs

A typeglob is a single name that holds *every* variable type with that name in a package - the scalar `$foo`, the array `@foo`, the hash `%foo`, the sub `&foo`, the filehandle `foo`, and the format `foo` all share one symbol-table entry, and `*foo` is the way to talk about that entry as a value. The sigil is `*`, deliberately chosen because the asterisk reads as “all”.

```
*foo                # the typeglob - all of $foo, @foo, %foo, &foo,
↳ etc.
*foo{SCALAR}        # ref to $foo
*foo{ARRAY}         # ref to @foo
*foo{HASH}          # ref to %foo
*foo{CODE}          # ref to &foo
*foo{IO}            # the filehandle, if foo is one
*foo{NAME}          # the name "foo" - a string
*foo{PACKAGE}       # the package name - a string
```

In modern Perl, typeglobs come up rarely. References cover most of what they used to be needed for. The remaining legitimate uses are package aliasing, filehandle creation in older idioms, and a few introspection hooks.

What a typeglob does

The single most useful sentence about typeglobs:

Assigning to a typeglob makes one name mean another name.

```
*foo = *bar;           # $foo, @foo, %foo, &foo, FH foo all alias $bar,
↳ @bar, %bar, &bar, FH bar
```

After that assignment, modifying `$foo` modifies `$bar`, calling `foo()` calls `bar()`, and so on for every type. The two names are now genuinely the same storage - not a copy.

Assigning a *reference* to a typeglob aliases only that one type:

```
our @items;
*shortcut = \@items;    # only @shortcut aliases @items; $shortcut
↳ and %shortcut unchanged
```

This is how the `Exporter` module installs imported subs into the caller's package: `*caller_package::name = \&exporter_package::name`. Each imported function gets one slot of one glob assigned, so the rest of the destination package's variables under that name are untouched.

`*name = \&sub` - the standard import idiom

```
package My::Util;
sub greet { "Hello, $_[0]!" }

package main;
*greet = \&My::Util::greet;    # import greet into main::
print greet("world");          # "Hello, world!"
```

This is the mechanism that `use` invokes via `Exporter::import` - the importing package gets a glob assignment for each requested name. `Exporter` does it; you can do it by hand for cases `Exporter` doesn't fit.

Local glob aliases

`local *name = \&$thing` is a temporary alias that auto-restores when the enclosing scope exits:

```
our $config = "default";

sub run_with_config {
    my ($alt_config, $code) = @_;
    local *config = \&$alt_config;    # only inside this sub
    $code->();                          # any read of $config
```

(continues on next page)

(continued from previous page)

```

    ↪reads $alt_config
}

run_with_config("test", sub {
    print "config is $config\n";           # "config is test"
});

print "config is $config\n";             # "config is default" -
    ↪restored

```

This is a heavyweight mechanism that has largely been replaced by plain `local $config = $alt_config`; for scalars and explicit parameter-passing for everything else. The case where it still matters is when you need to alias a *filehandle* across the call of arbitrary user code (see below).

Filehandle typeglobs

Before Perl 5.6's "open my \$fh" idiom, filehandles had to be named by package globals or passed as typeglobs:

```

# Old style - bareword filehandle in a glob slot
open FH, '<', $path or die $!;
while (<FH>) { ... }
close FH;

# Pass to a sub:
sub print_to { my $fh = shift; print $fh "stuff\n" }
print_to(\*FH);           # ref-to-glob

# Modern style:
open my $fh, '<', $path or die $!;
while (<$fh>) { ... }
close $fh;
print_to($fh);           # plain scalar ref to IO

```

The modern style is preferred everywhere new code is written. The typeglob-filehandle form survives in three places:

1. **The standard handles** - STDIN, STDOUT, STDERR are bareword filehandles backed by the typeglobs *STDIN, *STDOUT, *STDERR. You see these in error messages and reset code:

```

print STDERR "error\n";
*STDOUT = *STDERR;           # redirect all STDOUT to STDERR for
    ↪the rest of the program

```

2. **Returning newly-created filehandles** - before lexical filehandles, the way to return a fresh handle from a sub was to `local *FH` and return *FH:


```

sub new_handle {
    my $path = shift;
    local *FH;
    open FH, $path or return;
    return *FH;           # the glob outlives the local
    ↪because it was returned
}

```

Modern equivalent:

```

sub new_handle {
    my $path = shift;
    open my $fh, $path or return;
    return $fh;
}

```

3. **Selecting a different default output handle** - select returns and accepts a file-handle, in glob-or-string form:

```

my $old = select(STDERR);    # subsequent print without
    ↪explicit fh writes to STDERR
print "errors-only output\n";
select($old);                # restore

```

***foo{THING} - the slot accessors**

*foo{THING} returns a reference to one slot of the glob, or undef if that slot is empty:

```

*foo{SCALAR}      # \ $foo  if $foo has been touched
*foo{ARRAY}       # \@foo   if @foo has been touched
*foo{HASH}        # \%foo   if %foo has been touched
*foo{CODE}        # \&foo   if &foo defined
*foo{IO}          # the IO handle, or undef

```

The *foo{IO} form is the one place you regularly see THING syntax in modern code. It's how you ask “is this scalar holding a filehandle”:

```

sub is_handle {
    my $thing = shift;
    return defined ref($thing) && (
        ref($thing) eq 'GLOB'
        || (ref(\$thing) eq 'GLOB' && defined *$thing{IO})
    );
}

```

The standard introspection helper that wraps this is `Scalar::Util::openhandle`.

*foo{SCALAR} has a quirk: it always returns *something* - the package's \$foo slot is created on demand. The other accessors return undef when the slot is empty.

What replaces typeglobs

For the patterns where typeglobs used to be the only answer, the modern alternatives are:

Old typeglob pattern	Modern replacement
Pass a filehandle to a sub	Lexical FH: open my \$fh, ...
Return a filehandle from a sub	Lexical FH: return \$fh;
Alias @there as @here	our @here; *here = \@there; (still typeglob, no replacement)
Pass an array by reference	\@arr - pass the reference
Install an imported sub	*name = \&Source::name (still typeglob) - or use Exporter
Localised global alias	local \$var = ... for plain scalars

The `*here = \@there` package-aliasing pattern is the one that typeglobs still own outright - there's no “alias one package variable to another” mechanism that doesn't go through a glob. Inside Exporter-using modules that mostly happens automatically.

See also

- [References](#) - the modern alternative to globs for most “indirect access” patterns.
- [Scalars](#) - the dynamic-typing story. Globs are a separate type: `ref \*FH` returns 'GLOB'.
- `local` - the safe way to install a temporary glob alias.
- `select` - the selected-output-handle function that takes glob or string handle names.
- `Scalar::Util` - `openhandle`, `reftype`, blessed for runtime introspection.
- `Symbol` - generate fresh anonymous globs and qualify package names. (No local reference page yet.)

Interpolation

Double-quoted strings, here-docs, regex patterns, the `s/.../.../` replacement, and `qq//` all *interpolate*: they expand variables and certain escape sequences in place. Single-quoted strings, `q//`, and the LHS pattern of `tr/.../.../` do not. Knowing which constructs interpolate and exactly which forms get expanded is one of the most useful pieces of Perl knowledge.

```
my $name = "Alice";
my @nums = (1, 2, 3);

"hello, $name"           # "hello, Alice"
'hello, $name'          # 'hello, $name' - single quotes don't
↪ interpolate
```

(continues on next page)

(continued from previous page)

```
"@nums in a string"      # "1 2 3 in a string"
"\$name is $name"        # "$name is Alice"
```

What interpolates

Inside "" (and the other double-quotish contexts), Perl expands exactly two shapes of variable:

```
$name          # scalar - the simple form
@list          # array - joined by $" (one space by default)
```

Plus the bracket-and-brace subscript forms that compose on top of those:

```
$arr[0]        # one element of @arr
$h{key}        # one value from %h
$ref->[0]       # via array ref
$ref->{key}     # via hash ref
@arr[1, 2]     # array slice
@h{qw(a b)}    # hash slice
```

Note that **whole hashes do not interpolate**:

```
"hash is %h"      # "hash is %h" - literal '%' is fine, '%h' is NOT
↳magical
```

If you want a hash dumped into a string, format it explicitly with `Data::Dumper` or a manual join.

What does *not* interpolate

```
&sub          # function calls don't interpolate
sub_with_args($x) # neither do calls with arguments
$h->method     # method calls don't interpolate
$obj->{a} + 1   # arithmetic doesn't interpolate
$arr[0+1]      # this DOES interpolate as $arr[0+1] - subscripts
↳allow expressions

"\u\L$name"    # case-modifying escapes - these DO interpolate
"\x{2603}"     # numeric escapes - DO interpolate (a snowman)
```

The rule of thumb: **variables and their subscripts interpolate; function/method calls and arithmetic on top of them do not**. When you want an expression interpolated, see the `@{[ ... ]}` and `${\ ... }` constructs below.

The `${name}` braces

Curly braces around a variable name disambiguate where the name ends:

```
my $who = "Larry";

"${who}speak"          # "Larryspeak"
"$whospeak"           # "" with warning: $whospeak is
↳uninitialised
"${who}::sub"          # "Larry::sub"
"$who::sub"           # the variable $who::sub from package who -
↳bug
```

Without the braces, Perl is greedy: `$whospeak` reads as one identifier, `$who::sub` reads as a package-qualified name. The braces force the parser to stop at `}`.

This is not the same brace-form as a hash subscript:

```
"${key}"               # hash element                - what most
↳people mean
"${h}{key}"            # hash element in a different parse path -
↳same result
"${h{key}}"            # also the same
"${\(\$h{key})}"       # the absurd "interpolate an expression"
↳form
```

The `${name}` braces are mostly used for the disambiguation case (name followed by alphanumerics that would otherwise be eaten), not for hash access.

The `@{...}` and `${...}` deref-with-braces forms

When you want to *dereference* something inside a string, the brace form lets the deref expression be arbitrary:

```
my $config = { users => ['Alice', 'Bob'] };

"users: @{ $config->{users} }"    # "users: Alice Bob"
"users: $config->{users}"         # "users: ARRAY(0x...)" - wrong;
↳whole arrayref stringified
"users: @$config->{users}"        # parse error / wrong
```

The `@{ EXPR }` form deref's `EXPR` as an array. The braces are **required** because the deref target is an expression, not a plain variable. `@$config->{users}` doesn't parse the way intended

- `@$config` would deref `$config` as an array, and `->{users}` is a hash-element access on what's left.

The same for `${ EXPR }` (scalar deref) and `%{ EXPR }` (the hash deref, which returns a flat list, rarely useful in interpolation).

`@{ [ expr ] }` - interpolate any list expression

To interpolate the result of an arbitrary expression, wrap it in `@{ [ ... ] }`:

```

my $name = "World";
print "hello, @ {[ uc $name ] }!\n";           # "hello, WORLD!"

print "found @ {[ scalar @hits ] } matches\n"; # interpolate a_
↪count

print "users: @ {[ join ' , ' , @users ] }\n";  # interpolate a_
↪join

```

The mechanism: `[ ... ]` builds an anonymous array reference whose contents are the expression's list-context value; `@{ ... }` then deref's that arrayref back into a list, which interpolates as a list does (joined by `$`).

This is the canonical workaround for “I want a function call inside a double-quoted string”. The alternative is concatenation:

```

print "hello, " . uc($name) . "!\n";           # equivalent

```

The `@{ [ ... ] }` form is denser when there are multiple interpolations; concatenation is clearer when there's just one. In templates with many fields, `printf` or `sprintf` is usually clearer still:

```

printf "found %d matches\n", scalar @hits;

```

Use `@{ [ ... ] }` when you genuinely want a Perl expression in the middle of a string and the alternatives are uglier.

`${ \ expr }` - interpolate a scalar expression

The scalar counterpart of `@{ [ ... ] }`:

```

print "answer: ${ \ 6 * 7 } \n";               # "answer: 42"
print "ts: ${ \ scalar localtime } \n";       # "ts: Mon Apr 27_
↪..."

```

`\ EXPR` makes a reference to the scalar value of `EXPR`; `${ ... }` deref's it. The result is one value, interpolated as a scalar.

For most cases the `@{ [ ... ] }` form is more uniform - it works for both scalar and list contexts. `${ \ ... }` is occasionally clearer when you specifically want scalar context:

```

"got @ {[ items() ] } items"      # items() called in LIST context
"got ${ \ items() } items"       # items() called in SCALAR context

```

Hash-subscript ambiguity in patterns

Inside a regex pattern (which is double-quotish), the parser has to guess whether `$foo[bar]` means “interpolate `$foo[bar]` (one element of `@foo`)” or “interpolate `$foo` and then match a character class `[bar]`”:

```
my @foo = ('hi');
$x =~ /$foo[abc]/           # @foo exists - parsed as
    ↳ $foo[abc] (one element)
$x =~ /$foo[abc]/           # @foo doesn't exist - parsed as
    ↳ $foo + char class
```

The disambiguation is the brace form:

```
$x =~ /${foo}[abc]/         # $foo + character class [abc]
$x =~ /$foo[5]/             # element 5 of @foo
$x =~ /\Q$foo\E[abc]/       # $foo (literal) + character class
```

The `\Q... \E` quoter disables metacharacter interpretation inside its scope, which is the cleanest fix when the variable is a user input.

Heredocs

Heredocs are double-quotish unless the marker is single-quoted:

```
my $msg = <<EOM;
Hello, $name!
The hash is %h.
EOM
# "Hello, Alice!\nThe hash is %h.\n" - $name interpolated, %h
    ↳ literal

my $msg = <<'EOM';
Hello, $name!
EOM
# "Hello, $name!\n" - single-quoted marker disables interpolation
```

Indented heredocs (Perl 5.26+) ignore leading whitespace up to the shortest run on any line:

```
my $msg = <<~EOM;
    Hello, $name!
    EOM
# "Hello, Alice!\n" - leading "    " stripped from every line
```

The `~` form is preferred in modern code; the un-indented form is ugly inside loops and method bodies.

Real example: building HTML safely

```
my @items = ('Apples', 'Bananas', 'Cherries');

my $html = <<~HTML;
    <ul>
    @[{ map { "    <li>$_</li>" } @items }]
```

(continues on next page)

(continued from previous page)

```
</ul>
HTML
```

The map returns a list of `<li>...</li>` strings; `@{[ ... ]}` interpolates them; the implicit `$` is a single space, giving:

```
<ul>
  <li>Apples</li>    <li>Bananas</li>    <li>Cherries</li>
</ul>
```

To get newlines between the items, set `$` locally:

```
my $html = do {
  local $ = "\n";
  <<~HTML;
    <ul>
      @{[ map { "  <li>$_</li>" } @items ]}
    </ul>
    HTML
};
```

Or join explicitly:

```
my $li = join "\n", map { "  <li>$_</li>" } @items;
my $html = "<ul>\n$li\n</ul>\n";
```

The explicit join is usually clearer for non-trivial templates. The `@{[ ... ]}` form earns its place when the template is mostly literal text with one or two embedded computations.

What the special variable `$` controls

The list separator in interpolation:

```
my @x = (1, 2, 3);
print "x = @x\n";           # "x = 1 2 3" - default $
is " "

local $ = ", ";
print "x = @x\n";           # "x = 1, 2, 3"

local $ = "\n ";
print "items:\n @x\n";      # "items:\n 1\n 2\n 3"
```

`$` is the canonical way to control how arrays interpolate without rewriting the string. See *filehandle and printing state* for the family of separator variables.

See also

- *Scalars, Arrays, Hashes* - the shapes that interpolation expands.
- *References* - the `@{ EXPR }` deref form and what the `\` in `${ \ ... }` does.
- `$"` - the list separator for array interpolation.
- `$,, $\\, $/` - the related separator variables for print and input.
- `sprintf, printf`
 - the explicit format-string alternative when the template is mostly punctuation.
- `qq, q, qw` - the quote-like operators with custom delimiters.
- *Scalars* - `$x`. Numbers, strings, references; `undef`; truth and definedness; the dual numeric/ string nature of scalars.
- *Arrays* - `@arr`. Element access, `$#arr`, negative indices, slices, list assignment, the `@arr / $arr[i] / @arr[i, j]` sigil rule.
- *Hashes* - `%h`. Key-value pairs, the four flavours of subscript, keys/values/each, key ordering, hash slices and key/value slices.
- *Context* - list, scalar, void, boolean. How each operator and assignment imposes context, what `wantarray` reports, the `scalar()` operator, and the recurring traps with `,` versus list construction.
- *Numbers and strings* - automatic conversion both ways, integer vs float, `Inf` and `NaN`, numeric literal forms, “0 but true”, `dualvar`, locale and precision pitfalls.
- *References* - `\X`, the seven things you can reference, the four ways to deref, auto-vivification, weak references, circular structures, `ref` and `Scalar::Util`.
- *Typeglobs* - `*name`. The symbol-table entry, aliasing, `*foo{THING}`, when modern Perl still needs them and when references are the right answer instead.
- *Interpolation* - what double-quoted strings, here-docs, and regex patterns interpolate; the `${name}` braces; the `@{[ expr ]}` and `${ \ expr }` baroque constructs; ambiguities with `[` and `{` after a sigil.

Cross-cutting concepts

A handful of ideas appear on every page below. Read these once and the rest will feel familiar:

- **Sigils mark the *result*, not the container.** `@arr[1, 3]` is not “two arrays”; the `@` says “the result is a list”. The variable named `arr` is the same in `@arr`, `$arr[0]`, `@arr[0..3]`, and `%arr[0..3]`.
- **Context coerces.** A scalar evaluated in list context becomes a one-element list; a list evaluated in scalar context yields its **last** element (comma operator), but an *array* in scalar context yields its **length**. The two cases are not the same; see *context*.
- **References are scalars.** A reference to anything fits in a scalar variable, which is what makes nested data structures possible. The deref syntax (`@$ref`, `$ref->[0]`, `${$ref}{key}`) is the inverse of `\`.

- **undef is a value, not the absence of one.** A scalar can hold the value undef; testing defined `$x` is how you ask. An array can have undef elements without being empty.
- **Strings and numbers convert silently.** `"42" + 1` is 43, `42 . "x"` is `"42x"`. The conversion is always defined; see *numbers and strings* for what counts as “looks like a number” and where the corners are.

See also

- *perlop* - the operators that read and produce these values (arithmetic, comparison, subscript, range, arrow, assignment).
- *perlvar* - the special variables Perl populates with state from the last operation, including `$_`, `@_`, the match variables, and the I/O-shaping `$/`, `$\`, `$,`, `$"`.
- *perlfunc* - the built-in functions that operate on these types: `ref`, `scalar`, `wantarray`, `defined`, `keys`, `values`, `each`, `exists`, `delete`, `bless`.
- *References tutorial* - step-by-step introduction to the reference forms covered tersely in *references*.

5.1.5 Subroutines

A subroutine is a named (or anonymous) block of code that takes a list of arguments, runs in a fresh lexical scope, and returns a list of values. Subroutines are the unit of reuse in Perl, the unit of dispatch for object-oriented code, and - through closures

- the unit of stateful encapsulation. Almost everything else in the language is built on top of them.

PetaPerl implements the full Perl 5.42 subroutine model: the classic `@_`-based calling convention, modern signatures, the prototype mechanism that controls call-site parsing, lexical and package scoping (`my`, `our`, `local`, `state`), closures, the attribute system (`:lvalue`, `:method`, `:prototype( . . . )`), recursion with `goto &sub` tail calls, and the special compile-time and run-time blocks (`BEGIN`, `END`, `INIT`, `CHECK`, `UNITCHECK`).

This reference is split into one page per topic. The `@_` aliasing rule lives with arguments, the closure-over-loop-var pitfall lives with scoping, the parser-level effects of prototypes live with prototypes - so each page can be read and returned to without sweeping context.

Choose a topic

Declaration

Every sub is one of three things: a **named** sub bound to a package, an **anonymous** sub stored in a code reference, or a **lexical** sub bound to a `my`-like slot in the current scope. The declaration form picks one; everything else (prototype, signature, attributes, body) decorates it.

Named subs

```
sub greet {  
    my ($name) = @_;  
    return "Hello, $name!";  
}  
  
greet("Alice");           # "Hello, Alice!"
```

The name `greet` is bound in the current package’s symbol table under the ampersand & slot - the same slot the runtime looks up when you call `greet(...)` or `\&greet`. Named subs are visible to any code that runs after the sub statement is compiled; in practice “compiled” means “the file is parsed up to the declaration”, which is almost always what you want.

A named sub can also be declared in two parts - a forward declaration and a definition:

```
sub later;                # promise: a sub named 'later' will  
→exist  
sub later {               # fulfilment  
    return 42;  
}
```

The forward declaration is mainly useful when prototypes need to take effect for recursive calls (the prototype must be visible *before* the recursive call is parsed) or when you want to suppress “Bareword not allowed” warnings for code that calls the sub before its definition.

Calling a sub by name

```
greet("Alice");           # most common form  
greet "Alice";            # parens optional once predeclared  
&greet("Alice");         # legacy: disables prototype  
→checking  
&greet;                  # legacy: pass current @_ as the  
→call's @_  

```

The bare-& form (`&greet` with no parens) is a holdover from perl4 with one remaining legitimate use: passing the current `@_` straight through without copying. See [recursion](#) for `goto &greet`, the tail-call variant.

Anonymous subs

```
my $double = sub {  
    my ($x) = @_;  
    return $x * 2;  
};  
  
$double->(7);             # 14
```

The `sub { ... }` expression evaluates to a code reference. It captures any lexical

(my/state) variables from its enclosing scope - see [scoping](#) - and is invoked through `->()` or with the legacy `&$double()` / `&{$double}()` forms.

Anonymous subs are how every Perl callback works:

```
my @sorted = sort { $a <=> $b } @list;
my @big    = grep { $_ > 100 } @list;
my @str    = map { "<$_>" } @list;
```

They are also the building block for closures, dispatch tables, and mock objects. A dispatch table:

```
my %op = (
    add => sub { $_[0] + $_[1] },
    sub => sub { $_[0] - $_[1] },
    mul => sub { $_[0] * $_[1] },
);

my $result = $op{$cmd}->($x, $y);
```

Code references

A code reference is a first-class scalar that holds a sub. You get one from `sub { ... }`, from `\&named_sub`, or from a method lookup with `can`:

```
my $cref = \&greet;           # ref to named sub
my $cref = sub { ... };       # anonymous
my $cref = $obj->can('method'); # method lookup, returns code ref
    or undef

$cref->("Alice");             # invocation
&$cref("Alice");             # legacy invocation
```

Inside a code ref, the sub keeps its identity: `\&greet` is always the same ref for the lifetime of the package's `&greet`. Anonymous subs are distinct refs every time the `sub { ... }` expression is evaluated - that is what makes closure factories work.

Lexical subs

A lexical sub is declared with `my sub` (and the corresponding `state sub` and `our sub`):

```
sub outer {
    my sub helper {
        my ($x) = @_;
        return $x * 2;
    }

    return helper(21);           # 42
}

helper(1);                      # error: undefined subroutine
                                # (helper isn't visible here)
```

`my sub` is most useful inside long functions where you want a private helper without polluting the package namespace. It is not heavily used in the wild, but it is the proper tool when you want true encapsulation. See [scoping](#) for state and our variants.

Predeclaration

A predeclaration is a `sub NAME; statement` (no body, no braces). Three reasons to write one:

```
# 1. Allow the sub to be called without parens before it's defined
sub later;
later;                                # parses as later()

# 2. Make a prototype visible before the body is reached
sub mypush (\@@);
sub mypush (\@@) { ... }

# 3. Document an exported API at the top of the file
sub greet;
sub farewell;
sub welcome;
# ... actual definitions further down
```

In modern code reasons (1) and (3) are mostly stylistic. Reason (2) is a real requirement when you use prototypes; see [prototypes](#).

What you cannot do

- You cannot redefine a built-in by `sub print { ... }` and have Perl quietly use yours. Built-in overriding goes through `CORE::GLOBAL::print` and is documented under [attributes](#) and the [prototype](#) detail page.
- You cannot have two named subs with the same name in the same package without a “Subroutine NAME redefined” warning. Either rename or genuinely intend the redefinition (and silence the warning with `no warnings 'redefine';`).
- You cannot give a forward declaration and a body with *different* prototypes. The two declarations must agree.

See also

- [Arguments and @_](#) - what the body sees once the call is made.
- [Scoping](#) - what `my`, `our`, `local`, and `state` do to the variables inside the body.
- [sub](#) - the keyword’s perlfunc page.
- [prototype](#) - runtime introspection of a sub’s prototype, and the bridge to overriding built-ins.
- [Arrow operator](#) - `->( )` invocation of code references.

Arguments and @_

When a sub is called the classic way (no signature), Perl flattens the entire argument list into a single array and exposes it inside the body as `@_`. The elements of that array are not copies - they are **aliases** to the caller's expressions. This single rule is the source of most of Perl's calling-convention surprises and most of its calling-convention power.

The aliasing rule

```
sub bump {
    $_[0]++;           # mutates the caller's variable
}

my $x = 10;
bump($x);
say $x;               # 11
```

Modifying `$_[0]` writes through the alias to whatever the caller passed. That is by design - it is how `chomp(@lines)` manages to strip newlines from every element of `@lines` rather than from a copy of `@lines`.

The aliasing extends to literals, with predictable consequences:

```
bump(10);             # error: Modification of a read-
↳only                 # value attempted
```

A literal `10` is a constant. The alias points at it; the mutation attempt dies.

The unpacking idiom

Because the aliasing is rarely what the body wants, almost every classic-style sub starts with one of these two patterns:

```
# Named arguments (most common)
sub greet {
    my ($name, $greeting) = @_; # COPY into private lexicals
    $greeting //= 'Hello';
    return "$greeting, $name!";
}

# Method dispatch
sub method_call {
    my $self = shift;           # COPY first arg, leave rest in @_
    my %args = @_;              # remaining args as a hash
    ...
}
```

Once unpacked, the body works with private copies and cannot accidentally mutate the caller's state. `shift` with no argument inside a sub defaults to `@_`, which is why `my $self = shift;` is everywhere in OO Perl.

Legitimate uses of aliasing

Aliasing is the right tool when you actually want to mutate caller state:

```
# in-place trim
sub trim_inplace {
    for (@_) {
        s/^\s+//;
        s/\s+$//;
    }
}

my @data = (" hello ", " world\n");
trim_inplace(@data);
# @data is now ("hello", "world")
```

The `for (@_)` loop aliases `$_` to each element of `@_`, which is itself an alias to each element of `@data`. The `s///` modifications go through both alias hops to the original strings. Without the aliasing, this same shape of code would be twice as much work.

The same mechanism powers `chomp(@lines)`, `chop(@buf)`, and any “work over a list and modify in place” function you write yourself. State the intent in the function’s name (`_inplace`, `mutate_`, similar) so callers are not surprised.

Named arguments

Perl has no built-in named-parameter syntax in the classic form, but the `=>` (fat comma) plus a hash assignment gives the same effect:

```
sub configure {
    my %opt = @_;                # ('host' => 'localhost', 'port' =>
    ↪8080)
    $opt{host} //= 'localhost';
    $opt{port} //= 80;
    ...
}

configure( host => 'example.com', port => 8080 );
```

The fat comma is identical to a regular comma in semantics but quotes the bareword on its left, so `host => ...` is shorthand for `'host' => ...`. See [, for the operator itself](#).

For mandatory-plus-optional named args, the standard shape is:

```
sub render {
    my ($template, %opt) = @_;  # one positional, rest as named
    $opt{escape} //= 1;
    ...
}

render('greeting.tt', escape => 0, locale => 'de');
```

For the modern named-parameter idiom built on signatures, see [signatures](#).

shift, pop, unshift, push on @_

@_ is a real array. You can shift from the front, pop from the back, slice it, modify it. None of these change the caller's state - only writing through \$_[N] does.

```

sub describe {
    my $what = shift;           # remove first arg from @_
    my @rest = @_;              # copy what's left
    ...
}

sub method {
    my $self = shift;
    my @args = @_;
    $self->dispatch(@args);
}

```

A handy consequence: if you want to *re-pack* the remaining args to forward to another sub, just write @_ - that is exactly what goto &other does without the explicit pass. See *recursion* for tail-call forwarding.

wantarray and the calling context

Inside a sub, wantarray tells you the context the call was made in:

wantarray	Context	Common shape
1 (true)	list	return a list
0 (false)	scalar	return a single scalar
undef	void	return early; the result will be discarded

```

sub items {
    return unless defined wantarray;    # void: nothing to compute
    my @result = compute_items();
    return wantarray ? @result : scalar @result;
}

```

@_ says *what* came in; wantarray says *what is expected back*. They are the two halves of the calling-convention picture. See *lvalue and context* for the full discussion.

Common pitfalls

- **my (\$x) = @_ vs my \$x = @_.** The first is list context: \$x gets the first argument. The second is scalar context: \$x gets the *count* of arguments. The parentheses on the left are the difference; mistakes here are common.

```

sub one_arg {
    my ($x) = @_;           # $x = first arg
}

```

(continues on next page)

(continued from previous page)

```
sub count_args {
    my $n = @_;           # $n = number of args
}
```

- **Modifying `$_[N]` “by accident”.** Functions like `chomp`, `chop`, `tr///`, `s///` operate on their argument *in place*. `chomp($_[0])` mutates the caller’s string. If that’s not what you want, copy first.
- **Flattening of arrays and hashes.** Arrays and hashes passed as arguments lose their identity at the call site - they all become elements of the single flat `@_`. To pass an array without flattening, pass a reference:

```
takes_array(\@arr);           # one ref argument
sub takes_array {
    my ($aref) = @_;
    push @$aref, 'extra';
}
```

- **`@_` after `goto &sub`.** The current `@_` is passed through to the target. Anything you’ve already shifted off is gone from the perspective of the target. This is a feature, not a bug - see [recursion](#).

See also

- `@_` - the variable’s perlvar entry, with the aliasing rule and the loop-context discussion.
- `shift`, `pop` - default to `@_` inside a sub.
- [Signatures](#) - the modern alternative for named-and-positional parameter handling.
- [Return values](#) - what the body sends back.
- [Comma operator](#) - `=>` for named-arg construction.

Return values

A sub returns either by reaching its last statement, by hitting an explicit `return`, or by being `goto`-ed somewhere else (see [recursion](#)). The returned list is **flattened** the same way the argument list is - arrays and hashes lose their identity unless you return references - and is then evaluated in the caller’s context.

Implicit return

If control falls off the end of the body, the value of the last *expression* statement is returned:

```
sub square {
    my ($x) = @_;
    $x * $x           # implicit return
}
```

(continues on next page)

(continued from previous page)

```
say square(5);                                # 25
```

Implicit return is idiomatic for short, one-expression subs. It is *not* idiomatic for multi-branch subs, where readers benefit from seeing every exit explicitly marked with `return`.

If the last statement is a control-flow construct (a `for`, `while`, `if` without an `else`, ...), the returned value is unspecified. The empty sub returns the empty list.

Explicit return

```
sub max {
    my ($a, $b) = @_;
    return $a if $a >= $b;
    return $b;
}
```

Explicit `return` exits the sub immediately and supplies the return value. Multiple early returns are the standard shape for guarded code:

```
sub fetch {
    my ($id) = @_;
    return unless defined $id;      # void/empty
    return undef if $id eq '';      # explicit undef in scalar
    ...
}
```

Note the difference between `return;` and `return undef;`:

- `return;` returns the empty list in list context, `undef` in scalar context, and nothing in void context. It is the context-correct way to say “no result”.
- `return undef;` returns the *list* (`undef`) in list context, which has length 1 - so a caller writing `if (my @r = my_sub()) { ... }` will see a one-element list and the `if` will fire. This is almost always wrong.

The rule of thumb: write `return;` to indicate failure or absence; write `return undef;` only when the caller is guaranteed to be in scalar context.

List flattening on return

The same flattening that applies to arguments applies to return values:

```
sub two_lists {
    my @a = (1, 2);
    my @b = (3, 4);
    return (@a, @b);                # returns (1, 2, 3, 4) - one flat
    ↪ list
}
```

(continues on next page)

(continued from previous page)

```
my ($x, $y) = two_lists();      # $x = 1, $y = 2 ; rest discarded
my @all      = two_lists();      # @all = (1, 2, 3, 4)
my (@p, @q) = two_lists();      # @p = (1, 2, 3, 4), @q = ()
```

The last line is the classic surprise: `@p` greedily slurps the whole list because list-assignment to a flat array on the LHS has no way to know where one logical group ends and the next begins. To return two arrays distinctly, return references:

```
sub two_lists {
    return (\@a, \@b);
}

my ($pref, $qref) = two_lists();
```

Context-sensitive return

```
sub items {
    if (wantarray) {
        return (1, 2, 3);      # list context
    }
    elsif (defined wantarray) {
        return 3;               # scalar context: count
    }
    else {
        return;                 # void: nothing to compute
    }
}

my @list = items();            # (1, 2, 3)
my $cnt  = items();            # 3
items();                        # discarded
```

The “void early return” branch is a real optimisation pattern - expensive computations should not run when the caller throws the result away.

For the full discussion of `wantarray` and how it composes with `:lvalue`, see *[lvalue and context](#)*.

Returning hashes and arrays

Returning a `@array` or `%hash` directly works only because of flattening:

```
sub config {
    return (host => 'localhost', port => 80);
}

my %c = config();              # %c is the hash
my @c = config();              # @c = ('host', 'localhost', 'port'
    ↪ , 80)
```

For larger structures, return references:

```
sub config {
    my %c = (host => 'localhost', port => 80);
    return \%c;                # one ref, no flattening
}

my $c = config();
say $c->{host};
```

References are also the only way to return *multiple* aggregates without the caller having to know their sizes.

return inside an eval

return inside a `do {}` or `eval {}` block returns from the *enclosing sub*, not from the block:

```
sub safe {
    my $result = eval {
        return 'inside';        # this returns from `safe`, not
    } from eval                 # never reached
    return $result;
}
```

To exit just the eval, use a value ('inside' as the last expression of the block) - eval returns the value of its last expression like any other block.

Returning from BEGIN/END

BEGIN, END, INIT, CHECK, and UNITCHECK blocks are subs in the technical sense but their return value is discarded. Don't bother writing `return ...` inside them.

See also

- [return](#) - the keyword's perlfunc page.
- [wantarray](#) - what your caller wants back.
- [Lvalue subs and context](#) - the full picture for context-sensitive subs.
- [Arguments and @_](#) - the symmetric story for what comes in.
- [die, warn](#) - the alternatives to “return failure”.

Scoping: my, our, local, state

Inside a sub, four declarators control where a variable lives, how long it lives, and who can see it. Pick the wrong one and the sub leaks state, fails to capture properly in a closure, or quietly clobbers something the caller relied on.

Declarator	Scope	Lifetime	Typical use
<code>my</code>	lexical (the block)	until the block ends	the default; private locals
<code>our</code>	lexical alias to package	as long as the package	shared package globals with a short name
<code>local</code>	dynamic (the call tree)	until the block ends	temporarily change a special variable
<code>state</code>	lexical (the block)	program lifetime	per-sub persistent state, init once

my - lexical locals

`my` is the everyday declarator:

```
sub greet {  
    my ($name) = @_;  
    my $message = "Hello, $name!";  
    return $message;  
}
```

The variables exist for the duration of the call, are private to the body, and are stored in the sub's *pad* (one slot per declared lexical). They are **not** visible to callers, callees, or eval'd code that doesn't textually nest inside the sub.

`my` variables are the only ones that close over correctly. Closures capture *cells* in the surrounding *pad*, and only `my` (and `state`) variables get *pad* cells.

our - package globals with a lexical alias

```
package Counter;  
  
sub increment {  
    our $value;                # alias to $Counter::value  
    $value++;  
    return $value;  
}
```

`our $x` declares a *lexical alias* in the current scope to the package variable `$Pkg::x`. The variable itself is global. If two packages both `our`-declare `$VERSION`, they each get their own - `our` prefixes with the current package name.

Use `our` when:

- You genuinely want a package-level shared variable.
- You want to silence use `strict 'vars'` complaints about a variable that *should* be a package global (`$VERSION`, `@ISA`, `@EXPORT`, `%ENV` aliases, ...).

Do **not** use `our` to share state between subs in the same file. That's what `my` at file scope is for:

```
package Counter;

my $value = 0;                                # private to this package's file
sub increment { ++$value }
sub get      { $value }
```

local - dynamically-scoped temporary

`local` does *not* declare a new variable. It saves the current value of an existing (typically global) variable, sets a new value, and restores the old value when the enclosing block exits - even on exception:

```
sub slurp {
    my ($path) = @_;
    open my $fh, '<', $path or die "open $path: $!";
    local $/;                                # slurp mode for this sub only
    return <$fh>;
}
```

Without the `local`, the caller's `$/` would be permanently clobbered. With it, the change is confined to the dynamic extent of `slurp` - including any subs `slurp` happens to call.

`local` is the right tool for almost every special-variable override (`$/`, `$\`, `$,`, `$_`, `$@`, signal handlers in `%SIG`). It is rarely the right tool for ordinary user variables - those want `my`.

For the full register, see [local](#) and the [error-state](#) and [I/O](#) variable pages, which list which specials want local-isation in which patterns.

state - initialise once, keep forever

```
use feature 'state';

sub counter {
    state $n = 0;                            # initialised once, ever
    return ++$n;
}

counter();                                  # 1
counter();                                  # 2
counter();                                  # 3
```

`state` variables are lexicals that survive between calls. The initialiser runs the first time control reaches the declaration; subsequent calls skip it.

`state` is the right tool for:

- Per-sub caches and counters where the cost of constructing the data dominates and you don't want a closure factory.
- One-time setup that depends on values not available at compile time:

```
sub compiled_re {
    my ($pattern) = @_;
    state %cache;
    $cache{$pattern} //= qr/$pattern/;
    return $cache{$pattern};
}
```

(Note: this caches the *first* pattern only with a literal state of a single regex; the hash form generalises it.)

Closures: capturing surrounding my

An anonymous sub captures every my (or state) variable from its surrounding scope that it textually references:

```
sub make_adder {
    my ($n) = @_;
    return sub {
        my ($x) = @_;
        return $x + $n;           # captures $n
    };
}

my $add5 = make_adder(5);
my $add9 = make_adder(9);

$add5->(3);                       # 8
$add9->(3);                       # 12
```

Each call to make_adder creates a fresh \$n, and the returned closure references its own \$n. Two calls produce two independent counters / adders / state machines.

The closure-over-loop-variable surprise

```
my @subs;
for my $i (1, 2, 3) {
    push @subs, sub { $i };
}

print $_->(), "\n" for @subs;
# 1
# 2
# 3    - fresh $i each iteration: each closure sees its own
```

This is the *correct* behaviour with for my \$i: every iteration declares a fresh \$i, each closure captures its own.

The buggy shape is when \$i is declared *outside* the loop:

```

my $i;
my @subs;
for $i (1, 2, 3) {                                # NO `my` - same $i every iteration
    push @subs, sub { $i };
}

print $_->(), "\n" for @subs;
# 3
# 3
# 3 - all three closures share the one $i, and its final value is 3
→3

```

Always declare loop variables with `my`. Closure capture *follows the variable*, not the value, and “same variable” vs “different variable” is decided at the `my` site.

Capturing our and package globals

our aliases live for the whole package, so a closure capturing `$Counter::value` shares state with every other piece of code that touches `$Counter::value`. That’s almost always not what you want - closures over package globals are a sign that the design wants a `my` instead.

Pad lifetime

Every active sub call has a **pad**, a slot per declared lexical. The pad is what `my` writes into and what closures hold a reference to. While a closure is alive, every pad it captures is alive - even if the originating call has long returned. This is how `make_adder`’s `$n` survives the call to `make_adder` itself.

See also

- `my`, `our`, `local`, `state` - perlfunc detail pages for each declarator.
- `@_` - the one variable that’s *always* present in a sub regardless of declarators.
- *Declaration* - anonymous subs and the lexical `my` sub form.
- *Recursion* - closure-based trampolines and `__SUB__`.
- *Special variables* - almost every special wants `local`, not `my`.

Prototypes

A prototype is a piece of metadata attached to a sub that controls **how the parser parses calls to that sub**. It does not control the runtime - it does not check argument counts at runtime, it does not enforce argument types, it does not affect method calls. Its job is to make a user-defined sub *parse* like a built-in.

That narrow purpose is the one thing to remember about prototypes. Everything else follows.

Declaration

Two equivalent forms:

```
sub mypush (\@@) {                # short form
    my $aref = shift;
    push @$aref, @_;
}

sub mypush :prototype(\@@) {      # attribute form
    ...
}
```

The attribute form is mandatory when *signatures* are enabled in the same scope, because the parenthesised list after the sub name is then claimed by the signature.

What prototypes actually do

A prototype rewrites the parser’s expectations at the call site:

```
sub mygrep (&@) {
    my $code = shift;
    my @result;
    for (@_) { push @result, $_ if &$code }
    return @result;
}

mygrep { /foo/ } @items;           # parses correctly thanks to (&@)
                                   # without the prototype, this is
                                   # a syntax error
```

The (&@) prototype tells the parser: “the first argument is a block (no sub keyword needed, no comma after it), the rest is a list.” That’s exactly how built-in grep, map, sort parse.

This is the **only** place prototypes are useful: building DSL-shaped subs that look like built-ins.

Prototype characters

Char	Meaning
\$	scalar - forces scalar context on the argument
@	list - eats all remaining arguments, list context
%	hash - like @ (even-length list)
&	block / code ref - first position lets you omit sub
*	bareword, scalar, typeglob, or globref
\\$	scalar reference - caller must pass \$var form (auto-\-applied)
\@	array reference - caller must pass @arr form
\%	hash reference - caller must pass %h form
\&	code reference - caller must pass &fn form
\[\$@%&*&]	reference to any of the bracketed kinds
+	scalar OR \@/\% (auto-detect array/hash literal)
;	separates mandatory from optional
_	scalar; if omitted, defaults to \$_
()	no arguments - and triggers the constant-fold optimisation

The (\@) vs (@) difference

This is the whole reason backslash prototypes exist.

```

sub takes_array (\@) {                # backslash: pass-by-reference
    my ($aref) = @_;                 # @_ holds [array_ref]
    push @$aref, 'extra';
}

sub takes_list (@) {                  # no backslash: pass-by-value
    →(flattened)
    my @copy = @_;                   # @_ holds the flat list
    push @copy, 'extra';              # nothing in caller is affected
}

my @arr = (1, 2, 3);

takes_array(@arr);                    # caller writes @arr - parser
→inserts \@arr                        # @arr is now (1, 2, 3, 'extra')

takes_list(@arr);                     # caller writes @arr - flattens to
→a list                               # @arr is unchanged

```

The (\@) prototype lets you *write* the call as `takes_array(@arr)` - looking just like `push @arr, ...` - while the runtime actually receives an array reference. That's exactly what built-in `push`, `unshift`, `splice`, and `pop` look like to the user.

The () constant-fold prototype

A sub with the empty prototype () and a single-expression body that the optimiser can fold to a constant becomes inline-able:

```
sub PI () { 3.14159265358979 }
sub MAX () { 100 }

my $area = PI * $r ** 2;           # PI is inlined as 3.14159...
```

This is what use constant produces:

```
use constant {
    PI => 3.14159265358979,
    MAX => 100,
};
```

Constant-folded subs are zero-cost at runtime. They do require the prototype to be () exactly (no arguments accepted), and the body must be a single expression that the optimiser can reduce. Adding return defeats the inlining:

```
sub PI () { return 3.14159; }      # NOT inlined: explicit return
```

What prototypes do *not* do

- They do **not** check argument counts at runtime. Calling a (\$) sub with three arguments is a *parse-time* error, but if you bypass the parser (via &fn(...), code refs, goto &fn), there is no runtime check.
- They do **not** affect method calls. Method dispatch is resolved at runtime through @ISA, by which point parsing is long over. Prototypes on methods are silently ignored.
- They do **not** affect calls through code references. The call site sees a scalar, not a name; no name, no prototype lookup.
- They are **not** part of the sub’s identity. \&named_sub loses prototype knowledge.

When you should use prototypes

You are writing a sub that looks like a built-in. Examples:

- A list-eating function with a block argument ((&@)): mygrep, myfirst, try { ... } catch { ... }.
- An “inplace” helper that mutates an array in place ((\@) or (\@@)).
- A constant-like sub (() for inlining).

That’s the list. If you’re not building a built-in look-alike, do not reach for prototypes - use a signature, or just unpack @_.

When you should not use prototypes

- For “type checking”. Prototypes are not a type system. `sub f ($) { ... }` does not say “takes one scalar”; it says “takes one expression in scalar context”. Calling `f(@arr)` passes the *count* of `@arr`, not the array itself, with no warning.
- For run-time argument-count enforcement. Use signatures (which *do* enforce arity at runtime) or write the check by hand.
- On methods. They are silently ignored.

Forward declarations and recursion

A recursive call sees the prototype only if the sub was predeclared:

```
sub fact ($) ;                                # forward declaration with
↳prototype
sub fact ($) {
    my ($n) = @_;
    return $n <= 1 ? 1 : $n * fact($n - 1);    # parsed with ($) in
↳mind
}
```

Without the forward declaration, the recursive `fact($n - 1)` inside the body is parsed before the `( $ )` prototype is finalised, so the prototype does not apply to that call.

Inspecting a prototype at runtime

`prototype` returns a sub’s prototype string (or `undef` if there is none):

```
prototype(\&push);                            # '@@' - yes, even built-ins have
↳one
prototype('CORE::push');                      # '@@'
prototype(\&my_sub);                          # whatever you declared
```

See also

- `prototype` - runtime introspection.
- *Signatures* - the modern alternative for arity and named-parameter handling. They coexist with prototypes but have different goals.
- `sub` - the keyword’s `perlfunc` page.
- *Declaration* - where prototypes attach in the declaration syntax.
- *Attributes* - `:prototype( ... )` is the attribute form.

Signatures

A signature is a parenthesised parameter list that goes immediately after the sub name (and after any attributes), in place of - or in addition to - the body’s manual `@_` unpacking. Signatures introduce **lexical** parameter variables, perform **arity checking** at call time, and support default values and slurpy parameters.

Signatures stabilised in Perl 5.36; in 5.42 they are a production feature available with one of:

```
use feature 'signatures';           # explicit feature import
use v5.36;                          # implicit (signatures + other 5.36_
  ↪ features)
```

PetaPerl supports the full 5.42 signature syntax.

Basic shape

```
use feature 'signatures';

sub add ($x, $y) {                  # two mandatory positional_
  ↪ parameters
    return $x + $y;
}

add(2, 3);                          # 5
add(2);                             # error: Too few arguments
add(2, 3, 4);                       # error: Too many arguments
```

Each \$name in the signature declares a my lexical that receives the corresponding argument. Arity is checked at the call: the body never runs if the count is wrong.

This is the most concrete improvement over the classic form:

```
# Classic - silently accepts wrong arity
sub add {
  my ($x, $y) = @_;
  return $x + $y;
}

add(2);                             # $y is undef; later arithmetic_
  ↪ warns
add(2, 3, 4);                       # third arg silently ignored
```

Defaults

```
sub greet ($name, $greeting = 'Hello') {
  return "$greeting, $name!";
}

greet('Alice');                     # "Hello, Alice!"
greet('Alice', 'Hi');               # "Hi, Alice!"
```

The default is an arbitrary expression evaluated each time the parameter is omitted (so state and per-call computation both work). It can reference earlier parameters:

```

sub log_event ($message, $level = 'INFO', $prefix = "[ $level]") {
    print "$prefix $message\n";
}

log_event('starting');           # [INFO] starting
log_event('failure', 'ERROR');   # [ERROR] failure
log_event('done', 'OK', '>>');   # >> done

```

A default = undef is exactly equivalent to “no default” for arity (the parameter is still optional) but documents the intent. Use `//=`-style guards inside the body for “undef means use my fallback” semantics.

Optional vs mandatory

Parameters are mandatory by default. A parameter becomes optional by:

- Giving it a default value (`$x = 42`).
- Putting it after another optional parameter (transitive).
- Being followed by a slurpy `@arr` or `%hash`.

You cannot have a mandatory parameter after an optional one (except via slurpy):

```

sub bad ($x = 1, $y) { ... }      # error: Mandatory parameter
    ↳ follows optional

```

Slurpy parameters

The last parameter may be a slurpy `@list` or `%hash`. It eats all remaining arguments:

```

sub join_all ($sep, @parts) {
    return join $sep, @parts;
}

join_all(',', 'a', 'b', 'c');    # "a, b, c"

sub configure ($name, %opts) {
    %opts{verbose} //= 0;
    ...
}

configure('demo', verbose => 1, port => 8080);

```

The slurpy hash also fails if the remaining list has odd length:

```

configure('demo', 'verbose');    # error: Odd-sized list given

```

This is the single most useful pattern signatures unlock: a named-parameter API with one positional plus a `%opts` slurp.

Anonymous parameters

A \$ (or @, %) with no name is a positional placeholder:

```
sub second ($, $x) { return $x }

second('ignored', 'kept');      # "kept"
```

This is mildly useful when adapting a callback to an interface that supplies more arguments than you care about.

Combining defaults and slurpy

```
sub render ($template, $cache = 1, %opt) {
    ...
}

render('greeting.tt');           # cache=1, %opt empty
render('greeting.tt', 0);        # cache=0, %opt empty
render('greeting.tt', 1, locale => 'de'); # all three
render('greeting.tt', locale => 'de'); # ERROR: 'locale'
    ↳ becomes $cache                                # then 'de' is the lone
    ↳ slurp arg                                     # (odd-length hash) -
    ↳ diagnose carefully
```

That last case is the gotcha: a positional optional parameter followed by a slurpy hash is ambiguous with named-arg-only calls. Either drop the optional positional and put everything in the hash, or always pass the optional positional explicitly.

Migration from @_ unpacking

The mechanical translation:

```
# Before
sub make_user {
    my ($name, $email, %extra) = @_;
    ...
}

# After
use feature 'signatures';
sub make_user ($name, $email, %extra) {
    ...
}
```

You get arity checking and named lexicals for free. The body is unchanged.

The places where you cannot mechanically translate:

- The body uses the *count* of arguments dynamically (if (@_ >= 3)) - signatures fix arity at parse time, so rework into an optional parameter with a default.

- The body modifies `$_[0]` for the aliasing effect - keep the classic form, since signatures introduce *copies*, breaking the alias.
- The body uses `goto &other` and wants to forward the original `@_` - `@_` is still populated under signatures in 5.42, but this is implementation-defined and may change. Keep the classic form for tail-call forwarding.

Signatures and prototypes are orthogonal

Signatures and *prototypes* solve different problems:

Mechanism	Affects	When checked	Useful for
signature	the body's view	runtime	arity, named params
prototype	how the parser parses	compile-time	DSL look-alikes

A sub can have both, with the prototype written as an attribute:

```
use feature 'signatures';
sub mygrep :prototype(&@) ($code, @list) {
    grep { $code->($_) } @list;
}

mygrep { /foo/ } @items;           # prototype lets us drop `sub`
                                   # signature gives us $code and
                                   ↪ @list
```

The prototype shapes the call site; the signature shapes the body. They do not duplicate each other.

The single biggest reason to migrate

Catching wrong-arity calls at the moment they happen - not several lines later when you finally do `$y + 1` and trip a warning about an undef value. Signatures turn a class of subtle bugs into a clear error message.

See also

- *Arguments and @_* - the classic mechanism that signatures coexist with.
- *Prototypes* - different mechanism, often confused with signatures.
- *sub* - the keyword's perlfunc page.
- *Declaration* - where signatures sit in the declaration syntax.
- *Scoping* - signature parameters are my-style lexicals.

Recursion

Subs may call themselves, each other, or - with a bit of ceremony - replace their own call frame and pass control elsewhere without growing the stack. This page covers the four mechanisms: ordinary recursion, mutual recursion, goto &fn tail calls, and __SUB__ for anonymous self-reference. It also covers the depth limit you will hit if you ignore them.

Plain recursion

```
sub factorial {  
    my ($n) = @_;  
    return 1 if $n <= 1;  
    return $n * factorial($n - 1);  
}  
  
factorial(10);                                # 3628800
```

Each recursive call adds a new frame to Perl's call stack. The stack costs memory and a fixed per-call overhead; for shallow recursion this is irrelevant, for deep recursion it is the bottleneck.

Perl emits a “Deep recursion on subroutine X” warning when a sub recurses past 100 levels deep (compile-time threshold, PERL_SUB_DEPTH_WARN). The warning is in the recursion category - no warnings 'recursion' silences it. The only hard limit is the OS stack, typically 8 MiB by default, controllable with `ulimit -s`.

For deep, structurally-recursive workloads, prefer one of:

- An explicit work queue (iterative reformulation).
- Tail recursion via `goto &fn` (see below).
- A trampoline (see below).

Mutual recursion

Two subs that call each other:

```
sub is_even {  
    my ($n) = @_;  
    return 1 if $n == 0;  
    return is_odd($n - 1);  
}  
  
sub is_odd {  
    my ($n) = @_;  
    return 0 if $n == 0;  
    return is_even($n - 1);  
}  
  
is_even(10);                                # 1
```


Mutual recursion has no special treatment from the runtime - it is just two subs happening to call each other. The same depth limit applies.

If `is_even` is defined first and refers to `is_odd` before `is_odd` exists, you'll see a “Bareword not allowed” or “Called undefined subroutine” depending on the call shape. Pre-declare to fix:

```
sub is_odd;                                # forward declaration
sub is_even { ... is_odd($n-1) }
sub is_odd { ... is_even($n-1) }
```

goto &fn - tail-call forwarding

The `goto &SUBROUTINE` form **replaces the current frame** with a call to another sub. The current frame is gone - it is not on the stack - so the call does not grow the stack:

```
sub api_v2 {
    my @args = @_;
    push @args, deprecated => 1;
    @_ = @args;
    goto &api_v1;                        # tail-call: api_v1 sees the
    ↪modified @_
}
```

This is the right tool when:

- A wrapper sub needs to forward to a real implementation without leaving a stack frame behind. Decorators, logging shims, AUTOLOAD dispatchers.
- A recursive sub can express its recursion as “do some work, then become this other call”. Classic tail recursion:

```
sub factorial_tail {
    my ($n, $acc) = @_;
    $acc //= 1;
    return $acc if $n <= 1;
    @_ = ($n - 1, $n * $acc);
    goto &factorial_tail;
}

factorial_tail(20);                      # 2432902008176640000, no stack
    ↪growth
```

The mechanics: `@_` becomes the called sub's argument array, the current frame is unwound, control jumps. The caller of the *current* sub will see the result of the `goto`'d-to sub, never the one it called.

`goto &fn` is the **only** form of `goto` that performs a tail call. The label-based form (`goto LABEL`) is unrelated and far more restricted.

See [goto](#) for the full keyword description and the shape constraints.

__SUB__ - anonymous self-reference

```
use feature 'current_sub';           # or use v5.16

my $factorial = sub {
    my ($n) = @_;
    return 1 if $n <= 1;
    return $n * __SUB__->($n - 1);
};

$factorial->(5);                      # 120
```

__SUB__ evaluates to a code reference to the currently-running sub. It is the only way to recurse into an anonymous sub without storing it in a named variable first (and without leaking a strong reference to it from inside itself, the classic memory-leak shape).

For anonymous subs that do not need to recurse, prefer the plain anonymous form. __SUB__ exists for the recursion case specifically.

Trampolines

When you want bounded-stack recursion but `goto &fn` is awkward (because the recursion is mutual, or branches into many sub-cases), the trampoline pattern works:

```
sub trampoline {
    my $next = shift;
    while (ref $next eq 'CODE') {
        $next = $next->();
    }
    return $next;
}

sub even {
    my ($n) = @_;
    return $n == 0 ? 1 : sub { odd($n - 1) };
}

sub odd {
    my ($n) = @_;
    return $n == 0 ? 0 : sub { even($n - 1) };
}

trampoline(even(10000));              # 1, no stack growth
```

Each “recursive” call returns a closure to be invoked next, rather than calling itself directly. The trampoline drives the closures in a loop. The stack stays at depth 2 forever.

Trampolines are heavier than `goto &fn` (one closure allocation per “call”), but they generalise to arbitrary control flow - including continuations and CPS-style code. For production Perl, `goto &fn` is the usual choice; trampolines appear when the structure of the recursion makes `goto` clumsy.

The depth limit, in practice

```

sub recurse {
    my ($n) = @_;
    return $n if $n <= 0;
    return recurse($n - 1);
}

recurse(2000);          # warning: Deep recursion on _
↳ subroutine          # (default limit 1000)

```

The “Deep recursion” warning is informational, not fatal - it fires once per sub. You will run out of actual stack space well before any practical limit on most platforms. If you see this warning, the answer is almost always to refactor (iterative loop, goto &fn, or trampoline), not to raise the limit.

If you have already verified the recursion is bounded and you just want to suppress the noise, scope the warning category:

```
no warnings 'recursion';
```

This silences the warning entirely; it does not change the OS stack limit, which is what eventually terminates a runaway recursion regardless of warnings.

See also

- [goto](#) - the keyword behind goto &fn.
- [caller](#) - works through goto'd frames with the same numbering as ordinary frames.
- [Arguments and @_](#) - @_ semantics under goto &fn.
- [Declaration](#) - anonymous subs, the substrate for __SUB__ and trampolines.
- [Scoping](#) - closures, the substrate for trampolines.

Lvalue subs and context

Two related topics that come up wherever a sub needs to behave differently depending on how it is being used: **:lvalue** subs (those usable on the left-hand side of an assignment) and **wantarray**-driven dispatch on the caller's context.

Why context matters in subs

Every Perl expression evaluates in a context - list, scalar (with sub-flavours: numeric, string, boolean), or void - that's imposed by what surrounds it. A sub call inherits the context of its position:

```

my @r = my_sub();      # list context
my $r = my_sub();      # scalar context
my_sub();              # void context

```

(continues on next page)

(continued from previous page)

```
print my_sub();           # list context (print's args are
→LIST)
my $n = () = my_sub();    # scalar of (a list assignment) -
→element count
```

For most subs, this is invisible - they return the same data either way and Perl's automatic context conversion does the right thing. For some subs the work is genuinely different depending on context, and those subs benefit from explicitly inspecting `wantarray`.

wantarray: the three-way switch

```
sub items {
  if (wantarray) {
    return (1, 2, 3);           # list context
  }
  elsif (defined wantarray) {
    return 3;                   # scalar context
  }
  else {
    return;                     # void context - caller
→will                           # discard whatever we
→return                          return
  }
}
```

The three return values of `wantarray`:

Return value	Caller's context	Practical meaning
1 (true)	list	return all the things
0 (false)	scalar	return one thing (e.g. count)
undef	void	nobody cares; consider returning early

`defined wantarray` is the test for “scalar or list” (anything non-void).

The void-context shortcut

When work is expensive and the caller threw the result away, return early:

```
sub expensive_query {
  return unless defined wantarray;  # nothing to compute
  my @rows = $db->select(...);      # actually do the work
  return wantarray ? @rows : scalar @rows;
}

expensive_query();                 # void: returns
→immediately
```

(continues on next page)

(continued from previous page)

```
my @r = expensive_query();           # list: full result
my $n = expensive_query();           # scalar: count
```

This pattern is the single best reason to consult `wantarray`: turning a “always do the work” sub into a “do the work only if the caller wants it” sub.

When *not* to use `wantarray`

- For convenience overloading. A sub that returns a hashref in scalar context and a hash in list context is a maintenance hazard - readers can’t tell from the call site which they’re getting. Pick one.
- For subs that are obviously list-y or scalar-y. A sub named `count_users` should return a number; readers do not expect it to behave differently in list context.

:lvalue subs

An `:lvalue` sub is one whose return value is itself assignable. The body must end in an expression that has a memory location - typically a `my` lexical or an aggregate element:

```
sub editable :lvalue {
    my $self = shift;
    $self->{counter};           # last expression is an lvalue
    ↪lvalue
}

my $obj = { counter => 0 };
bless $obj, 'Counter';

$obj->editable = 42;           # writes through to $obj->{counter}
    ↪{counter}
```

The body’s last expression is what the assignment writes to. If that expression is not an lvalue (`return $x + 1`, `return scalar @list`), the assignment is illegal and you’ll get a clear error.

`:lvalue` is rarely the right tool. The only places it pays for itself:

- Tied variables and tied-like wrappers, where the sub is mediating access to a real lvalue underneath.
- DSLs that genuinely want assignment-syntax for setting (eg. `$config->host = 'localhost'`);).

For “set a property on an object”, a plain setter is clearer:

```
sub set_counter {
    my ($self, $value) = @_;
    $self->{counter} = $value;
    return $self;              # method chaining
}
```

(continues on next page)

(continued from previous page)

```
$obj->set_counter(42); # explicit, no surprises
```

:lvalue and the body's structure

Because the body's last expression is special, the rules are restrictive:

```
sub good :lvalue { $self->{x} } # OK - last expr is 
  ↪lvalue

sub also_good :lvalue {
  my @scratch = ...; # work
  $self->{x}; # last expr is lvalue
}

sub bad :lvalue {
  return $self->{x}; # NOT OK - explicit 
  ↪return # disables :lvalue
}

sub also_bad :lvalue {
  if ($cond) { $self->{x} }
  else { $self->{y} } # an `if` block is not an 
  ↪lvalue
}
```

The “no explicit return” rule is the same one that makes constant-fold prototypes work; both rely on the body being a single value-producing expression at the end.

For the modern alternative - a setter that returns \$self for chaining - see the *Object-Oriented Programming guide*.

Combining :lvalue with context

An :lvalue sub cannot easily branch on wantarray, because the lvalue version of the call doesn't quite fit the “returns a value” model. If you want both behaviours from one sub, accept that you are in DSL territory and document the contract carefully - or split into two subs.

Context-affecting operators

A few operators force a specific context on their argument; the sub will see that context regardless of where the surrounding expression is going:

```
my @r = scalar my_sub(); # scalar() forces scalar context on 
  ↪my_sub
my $n = () = my_sub(); # the inner () = ... is list 
  ↪context;
```

(continues on next page)

(continued from previous page)

```

# outer assignment to $n is scalar_
↳ of that

```

The `() = LIST` idiom is the standard way to force list context on a sub call and then read the count. See [assignment](#) for the operator-level discussion.

See also

- [wantarray](#) - the keyword.
- [Return values](#) - the rest of the return-value picture (without the lvalue twist).
- [Attributes](#) - `:lvalue` is one of several built-in attributes.
- [Assignment operator](#) - what the caller's `=` actually does on either side of an `:lvalue` call.
- [scalar](#) - explicit context coercion.

Attributes

A sub attribute is a colon-prefixed annotation between the sub's name (or signature) and its body. Attributes mark a sub as having some special property - being a method, being an lvalue, having a prototype, being constant, being subject to some third-party mechanism - that the runtime, the parser, or a user-written attribute handler can act on.

```

sub greet :method ($name)      { ... }
sub editable :lvalue           { ... }
sub mypush :prototype(\@@)     { ... }
sub PI :const                  { 3.14159 }
sub handler :Memoize           { ... } # third-party_
↳ attribute

```

The four attribute steps

To follow how an attribute affects a sub, distinguish four phases:

1. **Parse.** The compiler reads `:foo(args)` and breaks it into a name (`foo`) and an optional parameter string (`args`).
2. **Register.** The compiler invokes either a built-in handler (for `:lvalue`, `:method`, `:prototype(...)`) or `MODIFY_CODE_ATTRIBUTES` from the relevant package (for user-defined attributes).
3. **Store.** Built-in attributes set a flag on the CV (the compiled sub object). User attributes are stored wherever the handler chooses.
4. **Query.** Anyone can later ask `attributes::get(\&sub)` for the attribute list. Built-in attributes are also queryable through more direct APIs (`prototype`, `is_lvalue`, ...).

The four phases are why attributes are heterogeneous in behaviour: built-in attributes are runtime concepts; user attributes are whatever code in `MODIFY_CODE_ATTRIBUTES` decides to do with them.

Built-in attributes

:lvalue

Marks the sub as usable on the left-hand side of an assignment. The body's last expression must be an lvalue. See *lvalue and context*.

```
sub editable :lvalue { $self->{x} }  
  
$obj->editable = 42;           # writes through to $self->{x}
```

:method

Hint to the parser and to introspection that this sub is meant to be called as a method. The flag is mostly informational; it also suppresses prototype effects (since methods don't honour prototypes anyway):

```
package Counter;  
  
sub increment :method {  
    my $self = shift;  
    $self->{n}++;  
}
```

:method does **not** automatically use `parent` or `set @ISA`; it does not change dispatch. It is a documentation / intent marker that the runtime also uses to skip prototype processing.

:prototype(...)

The attribute form of a *prototype*, required when *signatures* are enabled in the same scope:

```
use feature 'signatures';  
sub mygrep :prototype(&@) ($code, @list) {  
    grep { $code->($_) } @list;  
}
```

The body's parameter list is the signature; the attribute carries the prototype. Without the attribute form, the parser cannot tell which parenthesised list is which.

:const

Promises the runtime that the sub's body produces a constant result, enabling more aggressive inlining than the bare `()` prototype gives:

```
sub PI :const { 3.14159265358979 }
```

:const was introduced for use by `use constant` and similar tools. Most code should reach for `use constant` rather than applying `:const` directly.

Third-party attributes

Any package can register attribute handlers for code refs by implementing `MODIFY_CODE_ATTRIBUTES`. The classic example is `Attribute::Handlers`, which provides a more declarative API on top of the raw mechanism. A user attribute looks identical to a built-in at the call site:

```
use Some::Memoize;           # provides ':Memoize' attribute

sub expensive :Memoize {
    ...
}
```

The package providing the attribute decides what `:Memoize` means. Common third-party attributes:

- `:Memoize` - automatically cache return values (from `Memoize`).
- `:Logged` - wrap the sub with logging entry/exit (from various AOP-style modules).
- `:Test`, `:Benchmark` - testing-framework markers.

When you see a colon-attribute that isn't on the built-in list above, find the package supplying it.

How attributes are written

Several attributes can be chained. The ordering on disk is the ordering they're processed in:

```
sub fancy :method :prototype(\@) :Memoize {
    ...
}
```

Attributes attach to *both* a forward declaration and the definition; if you write both, the attribute lists must agree:

```
sub greet :method;           # forward declaration with :method
sub greet :method ($name) {  # body, same attribute
    ...
}
```

A signature, when present, comes after attributes:

```
sub greet :method ($name) {  # OK
    ...
}

sub greet ($name) :method {  # error: attributes must precede
  ↪signature
    ...
}
```

What attributes do *not* do

- They are not type annotations. `:Int` is not a built-in; attempts to read it that way will silently do nothing unless some package has registered a handler that means the same thing.
- They are not metadata for the test framework unless the test framework registered a handler for them. Sprinkling `:tested` on subs does nothing without code on the other side reading the annotation.
- They are not optimisation directives. `:fast` does nothing.

Querying a sub's attributes

```
use attributes;  
  
sub demo :method :prototype($) :Memoize { ... }  
  
my @attrs = attributes::get(\&demo);  
# returns: ('method', 'prototype($)', 'Memoize')
```

This is the introspection equivalent of `prototype` - it answers “what annotations does this sub carry?” rather than “what does this sub look like to the parser?”.

See also

- *Declaration* - where the `:attr` syntax fits in the declaration grammar.
- *Prototypes* - `:prototype(...)` is one attribute among several.
- *Lvalue and context* - the meaning of `:lvalue`.
- `prototype` - runtime introspection of one specific built-in attribute.
- *Object-Oriented Programming* - `:method` and the broader OO context.

Phasers: BEGIN, UNITCHECK, CHECK, INIT, END

Five specially named blocks let code hook the transitions a program makes between being compiled and being run. They look like subroutines but are not: you can write several of each, none has a callable name, and each one fires automatically at its own moment.

```
BEGIN      { ... }  # during compilation, as soon as defined  
UNITCHECK  { ... }  # right after this compilation unit is compiled  
CHECK      { ... }  # after the whole program is compiled  
INIT       { ... }  # just before the runtime begins  
END        { ... }  # just before the interpreter exits
```

The sub prefix (`sub BEGIN { ... }`) is accepted but adds nothing and is discouraged. These are blocks, not named subs; you cannot call them, only let them fire.

When each one runs

A Perl program lives in two phases: a **compile phase** that parses the source into an internal form, and a **run phase** that executes it. The phasers mark the boundary.

- BEGIN runs the instant its closing brace is parsed, before the rest of the file is even read. This is how `use` works: a `use` is a BEGIN-wrapped `require` plus `import`, so the imported names exist for the code that follows. Once a BEGIN has run, it is undefined and its memory reclaimed.
- UNITCHECK runs as soon as the **compilation unit** that contains it has finished compiling. A unit is the main program, a module loaded by `require`, a string `eval`, a `do FILE`, or code compiled by the `(?{ })` regex construct. Because units finish compiling at different times, UNITCHECK is the only phaser whose timing tracks the individual file rather than the program as a whole.
- CHECK runs once, after the **initial** compile phase of the whole program ends and before the run phase starts.
- INIT runs once, just before the run phase begins, after all CHECK blocks.
- END runs once, as late as possible: after the program has finished (including after a `die`), just before the interpreter exits.

BEGIN and UNITCHECK are tied to *parsing*, not to the interpreter phase, so they can fire during any phase: a string `eval` at run time still compiles its body, and any BEGIN or UNITCHECK inside it runs then and there. CHECK and INIT, by contrast, mark the one-time compile/run boundary of the main program.

Multiplicity and execution order

You may write any number of each block. They all run, in a defined order:

- BEGIN: first in, first out (FIFO) as encountered during compilation. The first BEGIN in source order runs first.
- UNITCHECK: last in, first out (LIFO) within each unit, after that unit compiles.
- CHECK: LIFO, after all compilation.
- INIT: FIFO, just before the runtime.
- END: LIFO at exit. The last END defined is the first to run.

The instructive case is a file that mixes ordinary statements with all five block kinds. The blocks do not run where they are written; they run at their phase, in their per-phase order, while the ordinary statements run in between at run time. Conceptually, for a single file:

1. BEGIN blocks	-- FIFO, during compilation
2. UNITCHECK blocks	-- LIFO, after this unit compiles
3. CHECK blocks	-- LIFO, after all compilation
4. INIT blocks	-- FIFO, just before runtime
5. ordinary code	-- run phase, in source order
6. END blocks	-- LIFO, at exit

So BEGIN and UNITCHECK run before any CHECK; CHECK runs before INIT; INIT runs before the first line of ordinary code; and every END runs after the last. Within BEGIN and INIT, earlier-written blocks run earlier; within UNITCHECK, CHECK, and END, later-written blocks run earlier.

END blocks in detail

END is the cleanup phaser. It runs LIFO so that teardown unwinds in the reverse of setup, the way nested resources expect.

Inside an END block, `$?` holds the value the program is about to pass to `exit`. You may read it to learn how the program is ending, and you may assign to it to change the exit code:

```
END {  
    $? = 0 if $? == 42;      # rewrite one exit code on the way out  
}
```

Beware of clobbering `$?` by accident. Running anything via `system` inside an END block overwrites `$?` with the child's status, which then becomes the program's exit code unless you save and restore it.

END blocks do **not** run in several cases:

- Under the `-c` switch (compile-only syntax check). Main code does not run, and neither do END blocks.
- When compilation fails. A program that does not compile never reaches the point where END blocks would fire.
- When the process replaces itself with `exec`. The image is gone; there is nothing left to run the block.
- When the process is killed by a fatal signal you do not trap. Catch the signal yourself (via `%SIG`) if you need cleanup to run.

An END block created inside a string `eval` is **not** run when that `eval` finishes. It is registered like any other END block of its package and runs in LIFO order just before the interpreter exits.

`${^GLOBAL_PHASE}` and the phases

The read-only variable `${^GLOBAL_PHASE}` names the interpreter's current phase. Its values let code that runs in more than one phase find out where it is. During a CHECK block it reads CHECK; during INIT, INIT; during ordinary run-time code, RUN; during an END block, END. See `${^GLOBAL_PHASE}` for the full list of values and the phase timeline.

defer blocks

A `defer` block is END's small-scale cousin. It runs when the enclosing **block scope** exits, for any reason, rather than when the whole program exits. Reach for `defer` to pair acquisition with release inside a sub or loop; reach for END for process-wide teardown that must happen once, at the very end.

See also

- *BEGIN and use* - use is a BEGIN-wrapped load-and-import; the most common reason to care about compile-phase timing.
- `${^GLOBAL_PHASE}` - read the current phase from inside any block to learn where you are.
- *defer* - block-scoped cleanup, the right tool when END's program-wide reach is too broad.
- `$?` - the exit status an END block can read and rewrite.
- `%SIG` - trap the signals that would otherwise bypass END blocks entirely.
- *exit* - what END blocks run just before; the value it carries is the `$?` an END block sees.
- *Declaration* - `sub NAME { ... }`, anonymous subs, forward declarations, code references, the `&` sigil and what predeclaration buys you.
- *Arguments and @_* - the aliasing rule, the unpacking idiom, named-arg conventions, shift defaults, the `chomp(@lines)` legitimate use of aliasing, and the `$_[0] = ... trap`.
- *Return values* - explicit return, implicit last-expression, list-flattening on return, void-context shortcuts, and the difference between `return;` and `return undef;`.
- *Scoping: my, our, local, state* - pad lifetime, closures, the closure-over-loop-variable surprise, when `local` is the right tool, and what `state` actually means.
- *Prototypes* - what they do (call-site parsing), what they don't (runtime checks), the `(\@)` vs `(@)` difference, the `()` constant-fold prototype, and why most modern code shouldn't reach for them.
- *Signatures* - the modern syntax from 5.20+ stable in 5.36, with positional, optional, and slurpy parameters; defaults; named parameters via slurpy hash; and how signatures interact with (and don't replace) prototypes.
- *Recursion* - depth limit, mutual recursion, `goto &fn` for tail calls, `__SUB__` for anonymous self-reference, trampolines.
- *Lvalue subs and context* - `:lvalue` semantics, `wantarray`, the void-scalar-list three-way switch, and the standard "return early in void context" pattern.
- *Attributes* - `:method`, `:lvalue`, `:prototype(...)`, `:const`, third-party attributes, and how the four attribute-related steps (parse, register, store, query) compose.
- *Phasers* - the BEGIN, UNITCHECK, CHECK, INIT, and END blocks: when each fires relative to the compile/run boundary, their FIFO/LIFO ordering, `$?` and `${^GLOBAL_PHASE}` inside END, and the `defer` block.

Defining a sub vs. calling a sub

The two halves of the subroutine system are mostly orthogonal:

- **Definition** is a declaration plus a body. The declaration fixes the name (or makes the sub anonymous), the optional prototype or signature, the optional attributes. The body is a block of statements with its own pad.
- **Call** turns a list of expressions into the new pad's `@_` (or signature variables), runs the body, and collects whatever the body returned in the caller's context.

Almost every “subtle” subroutine question is a question about *one* side or the other. Prototypes affect calls only, not definitions; signatures affect what the body sees, not how the caller writes the call; attributes affect how the *defined* sub is treated by the runtime.

The classic vs. modern split

Perl carries two parallel argument-passing mechanisms:

```
# Classic: @_ aliasing, manual unpacking
sub add {
    my ($a, $b) = @_;
    return $a + $b;
}

# Modern: signatures, named lexicals, arity checking
use feature 'signatures';
sub add ($a, $b) {
    return $a + $b;
}
```

Both are fully supported. Signatures are not a replacement for `@_`; they are an additional, more declarative entry into the same calling convention. Inside a signature-using sub, `@_` is still populated (this is implementation-defined and discouraged to rely on, but it is the case in 5.42).

The split between classic and modern is the *single most useful piece of orientation* when reading other people's Perl. Most older code uses the classic form; most newer code uses signatures; both will be in front of you on any non-trivial codebase.

See also

- [perlop](#) - operators that interact with subs: `->` for method dispatch and code-ref invocation, `,` (and the fat comma) for named-arg construction, `=` for the list-assignment used in the `my (...) = @_` idiom.
- [perlvar](#) - special variables a sub will read or write: `@_`, `$_`, `$@`, `$AUTOLOAD`.
- [perlfunc](#) - the verbs of the subroutine system: `sub`, `return`, `caller`, `wantarray`, `prototype`, `goto`, `my`, `our`, `state`, `local`.
- *Object-Oriented Programming* - subs as methods, the role of `bless`, dispatch via `@ISA`, the modern use `experimental 'class'` syntax.

5.1.6 Locale handling

A *locale* is the set of cultural conventions a program follows when it sorts text, changes case, formats numbers and money, names the days of the week, or writes an error message. The same array of words sorts one way under a Swedish locale and another under a German one; the number twelve-and-a-half prints as 12.5 in one country and 12,5 in another. Locale handling is how Perl lets a single program adapt to those conventions at runtime instead of hard-coding one country's rules.

By default PetaPerl ignores locale entirely: every built-in behaves the same regardless of the user's environment, using the C/POSIX conventions (ASCII collation, . as the decimal point, English month names). You opt in to locale-awareness, one lexical scope at a time, with the `use locale` pragma. This page covers what that pragma turns on, which operations it affects, how to set and inspect the active locale, and where locale and Unicode interact.

What a locale is

The system C library divides cultural conventions into independent *categories*, each named `LC_` followed by the area it governs. Sorting (`LC_COLLATE`) is separate from case rules (`LC_CTYPE`), which is separate from number formatting (`LC_NUMERIC`), and so on. A locale is a named bundle of settings - `de_DE.UTF-8`, `sv_SE.UTF-8`, `C` - and switching to it can change all categories at once or just the ones you select. The categories PetaPerl honours are listed in the category table below.

The `use locale` pragma

Locale-sensitive behaviour is **off by default** and **lexically scoped**. `use locale` enables it from that point to the end of the enclosing block (or file); `no locale` turns it back off for the remainder of its own scope. Nothing outside the lexical scope is affected - a module that does not say `use locale` keeps the default C-locale behaviour even when called from code that did.

```
use locale;           # locale-aware from here down
my @sorted = sort @words; # collates per LC_COLLATE
{
    no locale;        # back to codepoint order in this block
    my @plain = sort @words;
}
```

Without the pragma, `sort`, `lc`, `uc`, `printf` and the rest use fixed C-locale rules no matter what the environment says. This is a deliberate safety default: a program produces the same output on every machine until it explicitly asks to follow the local conventions.

The pragma sets a per-statement hint, exactly as upstream Perl does, so a single file can mix locale-aware and locale-blind regions freely. The setting travels with the lexical scope, not with the call stack.

Locale categories

Each category controls a distinct group of operations. `use locale` activates all of them at once; you choose *which* locale is active per category through `setlocale`. The “affected operations” column links to the reference pages where the behaviour is documented in full - that is where the locale-specific rules for each built-in live.

Category	Governs	Affected operations
LC_COLLATE	String sort order and comparison	sort , the <code>lt le gt ge cmp</code> string comparisons
LC_CTYPE	Character classification and case mapping	lc , uc , lcfirst , ucfirst , <code>\w \d \s</code> and <code>\b</code> in regex
LC_NUMERIC	Decimal point and digit grouping	printf , sprintf , number-to-string conversion
LC_MONETARY	Currency formatting conventions	localeconv (inspection only; no built-in formats money for you)
LC_TIME	Day and month names, date layout	<code>strftime</code> via POSIX , langinfo
LC_MESSAGES	Language of system error strings	<code>\$!</code> text, langinfo message strings

LC_ALL is not a category of its own but a shorthand that sets every category at once.

Note one asymmetry that trips people up: LC_NUMERIC changes how numbers are *formatted for output*, but Perl’s own source code and its numeric *parsing* always use `.` as the decimal separator. A script that reads `12,5` from a German-locale file still needs to translate the comma itself before treating it as a number - the locale does not make `"12,5" + 0` yield `12.5`. See [numbers and strings](#).

Setting the locale

`use locale` says *follow the locale*; it does not say *which* one. That choice comes from the environment by default, and you can query or override it through the POSIX module.

`setlocale` selects the active locale for a category. The idiomatic call honours the user’s environment (LANG, LC_*):

```
use POSIX qw(setlocale LC_ALL LC_COLLATE);
setlocale(LC_ALL, "");                    # adopt the environment's
→ locale
my $cur = setlocale(LC_COLLATE);          # query the active collation
setlocale(LC_COLLATE, "sv_SE.UTF-8");    # force one explicitly
```

`localeconv` returns a hash describing the active LC_NUMERIC and LC_MONETARY conventions - the decimal point, thousands separator, currency symbol, and the rest - so a program can format values itself when no built-in does it directly.

I18N: Langinfo exposes `langinfo`, which reads individual items out of the current locale: the localized name of a weekday or month (DAY_1, MON_1), the date format string, the radix character, the currency symbol. It is the narrow, item-at-a-time companion to `localeconv`.

Locale and Unicode

Locale handling and Unicode are two different answers to “what is a character”, and they interact in ways worth understanding before you combine them. The full treatment - how use `utf8`, UTF-8 I/O layers, and locale `LC_CTYPE` rules compose, and the traps that arise when they disagree - lives in the Unicode tutorial. Start with *Unicode strings and encodings* and the *pitfalls chapter*; the *regex-and-properties chapter* covers how `\w` and the character classes behave under each model.

Locale leaks and safety

A locale is process-global state in the C library underneath, so a `setlocale` call made anywhere changes the locale everywhere in the process - across module boundaries, not just the scope that called it. Two consequences follow:

- **use locale is safe to scope; setlocale is not.** The pragma is lexical and leaks nothing. A `setlocale` call is a global side effect - if a library sets `LC_NUMERIC` to a comma-decimal locale and forgets to restore it, every later `printf` in the program is affected. Save and restore around a temporary change:

```
my $saved = setlocale(LC_NUMERIC);
setlocale(LC_NUMERIC, "de_DE.UTF-8");
# ... locale-sensitive work ...
setlocale(LC_NUMERIC, $saved);
```

- **Untrusted locale names are untrusted input.** Passing an environment-supplied string straight to `setlocale` lets the environment steer your program’s formatting. In a security-sensitive context, validate the name or pin the locale explicitly rather than calling `setlocale(LC_ALL, "")`.

Parallel execution and locale

PetaPerl auto-parallelizes some numeric loops across worker threads. This never affects locale-sensitive code, by construction. Operations whose results depend on the current locale - sort with the default collation, `LC_COLLATE` string comparison, `lc/uc/fc` case folding, and `LC_NUMERIC` number formatting - are never JIT-compiled or auto-parallelized; they always execute sequentially in the interpreter against the single process-wide locale. Auto-parallelization only ever runs numeric loop bodies, so no locale-sensitive computation is dispatched across worker threads. You do not need to guard locale state when writing parallelizable code: the parallel path simply never touches it.

Environment variables

When a program calls `setlocale(LC_ALL, "")`, the C library reads the locale from the environment in a fixed order of precedence: `LC_ALL` overrides every individual `LC_*` variable, each `LC_*` variable sets its own category, and `LANG` supplies the default for any category not otherwise named. These variables live in `%ENV` like any other environment value, so a program can inspect or adjust them before the `setlocale` call that consumes them.

Variable	Effect
LC_ALL	Overrides all categories
LC_COLLATE, LC_CTYPE, ...	Sets one named category
LANG	Default for any category not set above

Differences from upstream

- **Linux only, UTF-8 native.** PetaPerl runs only on Linux against the system C library's locales. The platform-specific locale machinery upstream Perl carries for VMS, Windows, and Android - the Win32::locale shims, the VMS category quirks, the “locales are not available on this platform” fallbacks - has no counterpart here and is simply absent. There is one locale model: the POSIX one your glibc provides.
- **LC_NUMERIC and source parsing.** As upstream, the decimal separator used when Perl *parses* numeric literals and string-to-number conversions is always `.`, regardless of locale; only *formatting* output respects LC_NUMERIC. PetaPerl makes no exception to this.
- **Auto-parallelization never touches locale-sensitive code.** PetaPerl auto-parallelizes some numeric loops across worker threads, a capability upstream Perl does not have. Because locale-dependent operations (collation, case folding, number formatting) are never JIT-compiled or parallelized, they always run sequentially against the single process-wide locale - so the feature introduces no locale-safety concern of its own. See [Parallel execution and locale](#) above.

See also

- [perlfunc](#) - the built-ins whose behaviour locale changes: `sort`, `lc`, `uc`, `lcfirst`, `ucfirst`, `sprintf`, `printf`.
- [perlop](#) - the string comparison operators (`lt`, `cmp`, ...) that collate under LC_COLLATE.
- [perlvar](#) - `%ENV` holds the `LANG/LC_*` variables; `$!` carries locale-translated error text.
- [POSIX](#) - `setlocale` and `localeconv`, the interface for choosing and inspecting the active locale.
- [I18N::Langinfo](#) - `langinfo`, for reading one localized item (month name, radix character) at a time.
- [Unicode strings and encodings](#)
 - how the Unicode character model relates to locale LC_CTYPE.

5.1.7 Diagnostic messages

When a program goes wrong, Perl tells you about it. Some diagnostics stop the program (die-style fatal errors); others print a line to `STDERR` and let it run on (warnings). This reference lists the diagnostics PetaPerl emits, grouped by where they come from, with the cause and the fix for each.

Every entry shows the **message verbatim** in a code span - the exact text the interpreter prints, with the placeholders (%s, %d, a named variable) shown the way they appear in the output. Search this page for the literal words you saw on STDERR and you will land on the entry.

Reading an entry

Each entry opens with the message text, then a short classification badge in parentheses, then the cause and the fix:

``Illegal division by zero`` **(F)** You divided by zero. Guard the divisor.

The badge follows the same letter code Perl has always used:

Code	Meaning
W	A warning - printed only when warnings are enabled. Optionally restricted to a category (shown as (W category)).
D	A deprecation - a warning that a feature is on the way out.
S	A severe warning - on by default, but still only a warning.
F	A fatal error - the program dies unless the error is trapped (eval, try).
P	An internal error (a panic) - should never reach a user. If you see one, it is a bug in pperl.
X	A very fatal error - the program exits immediately.
A	An alien error - produced by something other than Perl itself.

Enabling and silencing warnings

Warnings (the **W**, **D**, and **S** classes) are controlled by the warnings pragma. Without it, only the severe (**S**) warnings print:

```
use warnings;           # turn on every warning
no warnings;           # turn them all off (lexically)
no warnings 'uninitialized'; # turn off one category
use warnings FATAL => 'all'; # promote warnings to fatal errors
```

The category named in a (W category) badge is the string you pass to no warnings to silence just that diagnostic. A single \$SIG{__WARN__} handler can intercept every warning at runtime; \$SIG{__DIE__} does the same for fatal errors. See %SIG.

A note on portability

PetaPerl runs only on Linux and ships no XS layer. The platform diagnostic families a portable perl carries - VMS status codes, Win32 GetLastError text, “unimplemented on this architecture” notices, XS-loader failures - therefore do not arise here. Where upstream Perl documents a diagnostic that only fires on another operating system or through XS, PetaPerl simply never emits it.

The diagnostics

General fatal errors

Runtime dies that are neither regex-compilation nor compile-time syntax errors: method resolution, inheritance, subroutine calls, attributes, and module-level usage errors. All are fatal (F) and trappable with `eval` or `try`.

Subroutine calls

``Undefined subroutine &%s called``

(F) A call named a subroutine that does not exist at call time. The fully-qualified name is shown. Causes: a typo in the name, a missing use of the module that defines it, or calling before the sub is declared in a context where that matters. The runtime form ``Undefined subroutine &NAME called at FILE line N.`` adds the call site.

Method resolution and inheritance

``Can't use anonymous symbol table for method lookup``

(F) Method dispatch reached a package with no named symbol table (an anonymous stash). A blessed reference must point into a real, named package for method calls to resolve.

``Recursive inheritance detected in package '%s'``

(F) A class's @ISA chain leads back to itself - a cycle. Method resolution cannot proceed. Break the loop in the inheritance graph.

``Inconsistent hierarchy during C3 merge of class '%s'``

(F) Under the C3 method-resolution order, the linearization of this class's parents has no consistent ordering. Two parents disagree on the order of a shared ancestor. Restructure the inheritance so the parent orderings are compatible, or switch the class back to the default depth-first MRO.

``Invalid mro name: '%s'``

(F) A use `mro '...'` named an order that does not exist. The two valid names are ``dfs`` (the default depth-first order) and ``c3``.

Attributes

``Unterminated attribute parameter in attribute list``

(F) An attribute with a parenthesized parameter - `:Foo(bar` - was not closed. Balance the parentheses in the attribute list.

Module usage errors

These come from native modules when called incorrectly. They are fatal because the call cannot proceed.

``Usage: POSIX::sprintf(pattern, args...)``

(F) A POSIX function was called with the wrong arguments. The message names the correct calling form.

``IO::Socket::%s``

(F) An `IO::Socket` operation failed; the message names the specific call.

See also

- `@ISA` - the inheritance array whose contents the MRO errors are about.
- `mro` - selecting and inspecting the method-resolution order.
- `bless` - associates a reference with the package whose methods the anonymous-symbol-table error is about.
- *Value and type errors* - “Not a subroutine reference” and the other what-kind-of-value diagnostics.

Regex compilation errors

Every diagnostic on this page comes from the pattern compiler. A regular expression is compiled before it runs, and a malformed pattern dies at that point - before the match is ever attempted. The errors are all fatal (F); trap them with `eval` if you build patterns from untrusted input:

```
my $re = eval { qr/$user_pattern/ };  
die "bad pattern: $@" unless $re;
```

Upstream Perl appends a `<-- HERE` in `m/.../` suffix that points at the offending position. PetaPerl reports the same errors; the exact position-marker formatting may differ.

Common structural errors

``Unmatched ) in regex``

(F) A `)` in the pattern has no matching `(`. Usually a typo or an unescaped literal parenthesis. To match a literal `)`, escape it as `\)` or put it in a character class `[]`.

``Quantifier follows nothing in regex``

(F) A quantifier (`*`, `+`, `?`, `{n,m}`) appeared with nothing in front of it to repeat - for example `/*abc/` or `/(?:)+/` on an empty group. Quantify a literal `*` by escaping it: `/\*/`.

`Unterminated group`

(F) A (opened a group that the pattern never closes. Balance the parentheses.

`Unterminated character class`

(F) A [opened a character class with no closing]. To match a literal], place it first in the class ([]x) or escape it ([\]x).

`Unterminated comment group`

(F) A (?#...) comment group was opened but not closed with).

`Unterminated conditional pattern`

(F) A (? (condition) yes | no) construct was left unclosed.

`Unterminated group name`

(F) A named-group construct ((?<name>...), \k<name>) ran to the end of the pattern without closing its name delimiter.

`Unrecognized group type`

(F) The characters after (? do not form a group type PetaPerl knows. Check the (?...) spelling against the regex reference.

`Unexpected end of character class`

(F) The pattern ended in the middle of a [...] class.

`Unexpected end of escape sequence`

(F) A backslash escape was cut off by the end of the pattern.

Character-class and range errors

`Invalid [ ] range "%s" in regex`

(F) A character range inside [...] runs backwards (high to low), as in [z-a]. Order the endpoints low to high, or escape the - to match it literally.

`Invalid range endpoint in character class`

(F) One end of a - range is not a single character (for example a multi-character escape used as an endpoint).

`POSIX class [:%s:] unknown`

(F) A `[ :name: ]` POSIX class names a class that does not exist. The valid names are `alpha`, `digit`, `alnum`, `space`, `upper`, `lower`, `punct`, `print`, `graph`, `cntrl`, `xdigit`, `blank`, `word`, and `ascii`. A bare `[ :alpha: ]` outside another `[ ... ]` is also an error - POSIX classes only work *inside* a character class: `[ [ :alpha: ] ]`.

`Empty POSIX character class name`

(F) A `[ : : ]` with nothing between the colons.

Control-character and numeric escapes**`Missing control char`**

(F) A `\c` escape was not followed by the character it should control, as in a pattern ending in `\c`.

`Missing control character after \c`

(F) Same cause as above, from a different point in the compiler: `\c` needs one following character.

`Invalid control character '\c%s'`

(F) The character after `\c` is not one `\c` accepts.

`Invalid octal escape`

(F) A `\0/\o` octal escape is malformed.

`Invalid octal digit in \o{}`

(F) A digit inside `\o{ ... }` is not 0–7.

`Expected '{' after \o`

(F) `\o` must be followed by `{`, as in `\o{17}`.

`Expected '}' in \o{}` and `Invalid Unicode code point in \o{}`

(F) A `\o{ ... }` escape is unterminated or names a code point outside the valid range.

`Invalid hex escape` / `Invalid hex digit` / `Expected '}' in hex escape`

(F) A `\x` escape is malformed: a non-hex digit, or an unterminated `\x{ ... }`.

\g, \k, and group references

`\g0 is not allowed` and `\g{0} is not allowed`

(F) Group 0 is the whole match and cannot be back-referenced. Reference a real capture group (\g1, \g{name}).

`Invalid group number in \g` / `Invalid group number in \g{}`

(F) The number after \g is not a valid group reference.

`Expected digit or '{' after \g` / `Expected '}' after \g{`

(F) \g must be followed by a digit, a sign, or {name}.

`Reference to nonexistent group number %d`

(F) A \g/\1-style back-reference names a group number the pattern does not contain.

`Reference to nonexistent group (relative offset %d)`

(F) A relative back-reference (\g{-2}) points past the groups that exist.

`Reference to nonexistent named group '%s'`

(F) A \k<name> or \g{name} references a named group that was never defined.

`Reference to nonexistent or invalid group`

(F) A group reference could not be resolved to any group.

`Expected '<' or '\'' after \k` / `Expected '}' after \k{`

(F) A named back-reference \k must be written \k<name>, \k'name', or \k{name}.

Named groups

`Empty group name`

(F) A named group (?<> . . .) or reference has empty name brackets.

`Group name must not start with a digit`

(F) Group names follow identifier rules; they may not begin with a digit.

\N and Unicode escapes

`\N in character class requires braces`

(F) Inside [. . .], \N must be written \N{. . .}; the bare “any non-newline” form is not allowed in a class.

`\N{NAME} not supported in character class`

(F) A named-character `\N{NAME}` escape is not permitted inside a character class.

`Expected '}' after \N{ / `Empty \N{U+} escape`

(F) A `\N{...}` escape is unterminated or `\N{U+}` has no hex digits.

`Invalid hex in \N{U+...} / `Invalid Unicode code point in \N{U+...}`

(F) The hex inside `\N{U+...}` is malformed or names a code point outside the Unicode range.

`Invalid Unicode code point`

(F) A code-point escape names a value above the Unicode maximum (`0x10FFFF`).

`Empty Unicode property name` / `Unknown Unicode property name`

(F) A `\p{...}` / `\P{...}` property escape is empty or names a property PetaPerl does not recognize.

(?...) extended constructs

`Invalid flag after (?^ / `Invalid flag character`

(F) A flag-setting group `(?flags)` or `(?^flags:...)` contains a character that is not a valid pattern modifier.

`Expected '{' after (??`

(F) A `(??{ code })` postponed-subpattern construct is malformed.

`Expected '<', '=', or '>' after (?P`

(F) The Python-style `(?P<name>...)` / `(?P=name)` / `(?P>name)` construct is malformed.

`Expected DEFINE after (?(` / `Expected ')' after DEFINE`

(F) A `(?(DEFINE) ...)` block is malformed.

`Switch condition not recognized in regex` / `Invalid lookahead condition`

(F) The condition in a `(?(...) ...)` conditional is not a recognized form (a group number, a lookahead, R, or DEFINE).

`Expected ')' after condition` family

(F) A conditional construct's condition was not closed with `)`. The variants - ``Expected ')' after (? (R...))``, ``Expected ')' after code block condition``, ``Expected ')' after lookahead condition``, ``Expected ')' after lookbehind condition``, ``Expected '=' or '!' after (? (<` - all report the same shape of error at different condition types.`

Quantifier braces

`Expected '}' after quantifier` / `Expected '}' or ',' in quantifier`

(F) A `{n,m}` quantifier is unterminated or contains an unexpected character.

Lookbehind

`Variable length lookbehind not implemented`

(F) A lookbehind `(?<=...)` / `(?<!=...)` whose contents can match a variable number of characters. PetaPerl, like upstream Perl, requires fixed-length lookbehind. Rewrite the lookbehind to a fixed width, or use `\K` to mark a keep-point instead.

Verbs

`Unterminated verb`

(F) A control verb `(*VERB)` / `(*VERB:ARG)` was not closed.

Catch-all

`Regex compilation error at position %d: %s` / `Regular expression error`

(F) A position-tagged compile error for cases the more specific messages above do not cover. The trailing text names the underlying problem.

`Unterminated string in %s`

(F) A delimited construct inside the pattern ran to the end of input without its closing delimiter.

See also

- `qr` - precompile a pattern so a malformed one fails once, at compile time, instead of on every match.
- `m` - the match operator; the same compiler rejects a malformed pattern here.
- `s` - the substitution operator, whose pattern half is compiled by the same engine.
- *Warnings* - the non-fatal regex diagnostics (Useless use of `\E`, Invalid character in group name).

Compile-time syntax errors

Errors the parser raises before the program runs. A compile-time error stops the whole program: nothing executes, because the source could not be turned into runnable ops. Unlike runtime dies, these are not trappable with `eval BLOCK` - but a string `eval` of source that fails to compile reports them through `$@`:

```
my $code = '1 +';
eval $code;
warn "compile failed: $@" if $@;
```

Subroutine resolution at compile time

`Undefined subroutine &%s`

(F) The parser resolved a call to a named subroutine that does not exist at compile time. This is the compile-time companion to the runtime `Undefined subroutine &%s called` error: the difference is *when* the missing sub is noticed. Make sure the defining module is used, and that the name is spelled correctly and fully qualified where the package matters.

See also

- *General fatal errors* - the runtime form of the undefined-subroutine error.
- *Subroutines* - declaration, prototypes, and the rules that decide when a call resolves.
- `use` - pulls in the module that defines the subroutine, at compile time.

Warnings

Non-fatal diagnostics. A warning prints one line to `STDERR` and the program runs on. The severe (**S**) warnings print even without the warnings pragma; the plain **W** warnings print only when warnings are enabled for their category:

```
use warnings;                # all categories on
no warnings 'recursion';     # silence just deep-recursion
use warnings FATAL => 'utf8'; # make utf8 warnings die instead
```

The category in a (**W** category) badge is exactly the string you pass to `no warnings`.

Deep recursion

`Deep recursion on subroutine "%s"`

(W recursion) A subroutine has called itself (directly or through a chain) more than a hundred levels deep. This is usually a runaway recursion - a base case that never fires - rather than genuinely deep work. The message names the subroutine.

```
use warnings;
```

(continues on next page)

(continued from previous page)

```
sub countdown {  
    my $n = shift;  
    countdown($n + 1);           # bug: should be $n - 1, never_  
    ↪ stops  
}  
countdown(1);  
# Deep recursion on subroutine "main::countdown" at script line 5.
```

Fix the base case. If the recursion really is meant to run deep, silence the category for that scope:

```
no warnings 'recursion';
```

Wide characters

`Wide character in %s`

(S) A string containing a character above codepoint 255 was sent to a byte-oriented operation - most often print to a filehandle with no `:encoding` layer. Perl emits the bytes of the internal UTF-8 form and warns, because the output stream was never told it would carry Unicode.

```
use warnings;  
print "café\x{2603}\n";           # Wide character in print at ..._  
↪ line 2.
```

The fix is to tell the stream its encoding:

```
binmode STDOUT, ':encoding(UTF-8)';  
print "café\x{2603}\n";           # no warning; correct UTF-8 bytes
```

This is a severe **(S)** warning: it prints even without `use warnings`, because silently emitting raw internal bytes is almost always a bug.

Useless constructs

`Useless use of \E`

(W regexp) An `\E` in a pattern or string closes a `\U`, `\L`, `\Q` case/quote span that was never opened. The `\E` has nothing to end. Remove it, or add the opening `\U/\L/\Q` it was meant to close.

Group names

`Invalid character in group name`

(W) A regex named group (`(?<name>...)`) contains a character that is not allowed in a group name. Group names follow identifier rules - letters, digits, and underscore, not starting with a digit. The pattern still compiles, treating the construct leniently, but the name is suspect.

Encoding

``Cannot find encoding "%s" ``

(S) A `:encoding(...)` layer or an `Encode` call named an encoding that is not available. Check the spelling against the encodings `Encode` supports (UTF-8, Latin-1, ASCII, and the rest).

Catch-all

``Warning: something's wrong ``

(W) A generic warning the codegen raises for a condition that has no more specific message. The surrounding context names what triggered it.

See also

- `warnings` - the pragma that enables, scopes, and categorizes every **W** warning.
- `$SIG{__WARN__}` - intercept warnings at runtime instead of letting them reach `STDERR`.
- *Regex compilation errors* - the fatal regex diagnostics, as opposed to the warning-level ones here.
- `Encode` - the encoding names behind the “Cannot find encoding” warning.

Value and type errors

These diagnostics fire when a value is not the kind of thing the operation needs: a plain scalar where a reference was expected, a zero where a divisor was expected, a read-only value where a writable one was expected. All are fatal (**F**) unless noted.

Arithmetic

``Illegal division by zero ``

(F) The right-hand side of `/` or `%` evaluated to zero. Numeric division and modulus have no result at zero, so Perl dies rather than returning infinity or NaN. Guard the divisor:

```
my $rate = $count ? $total / $count : 0;
```

``Illegal modulus zero ``

(F) The right operand of `%` was zero. Same cause and same fix as division by zero, for the modulus operator.

References

``Not a reference ``

(F) A dereference (`@$x`, `$$x`, `$$x`, `$x->[0]`, `$x->{k}`) was attempted on a value that is not a reference. Most often `$x` is `undef` or a plain string. Check it first:

```
die "expected an arrayref" unless ref $x eq 'ARRAY';
```

`Not a subroutine reference`

(F) A value used where a code reference was required (for example the second argument to a higher-order function) is not a CODE ref. ``set_subname: not a code reference`` and ``set_prototype: not a code reference`` are the same error reported by `Sub::Util`, and ``PadWalker: cv is not a code reference`` is the `PadWalker` form.

Restricted hashes

A hash locked with `Hash::Util`'s `lock_keys` rejects access to keys outside its allowed set.

`Attempt to access disallowed key '%s' in a restricted hash`

(F) You read or wrote a key that the restricted hash does not permit. Either add the key to the allowed set, or unlock the hash.

`Attempt to delete readonly key '%s' from a restricted hash`

(F) You tried to delete a key that the hash has locked against deletion.

Read-only values

`Modification of a read-only value attempted`

(F) An assignment or in-place operation targeted something that cannot be changed: a literal, a constant, a value passed by the caller and made read-only, or a `ReadOnly/const` value. A common surprise is modifying `$_` inside a `map/grep` block when the source list elements are read-only.

Filehandles

`Bad filehandle: %s`

(F) An I/O operation was handed something that is not a usable filehandle - an undefined value, a closed handle, or a bareword that names no open handle. Open the handle (and check the result) before using it:

```
open my $fh, '<', $path or die "open $path: $!";
```

Memory

`Out of memory`

(F) An allocation failed. Usually the program asked for an implausibly large string, array, or hash - check the size you are requesting before assuming the machine is at fault.

See also

- [References](#) - what a reference is and how dereferencing works; the background for the “Not a reference” errors.
- [ref](#) - test what kind of reference a value holds before you dereference it.
- [Hash::Util](#) - `lock_keys`, `unlock_keys`, and the rest of the restricted-hash machinery.
- [General fatal errors](#) - method-resolution and inheritance errors that are also about what a value is.

Internal errors

A **P** diagnostic is a panic: a check that should never fail in correct operation did fail. These messages are not something your Perl code can cause through ordinary mistakes - they signal a bug in PetaPerl itself. If you see one, it is worth a bug report. See [Disclaimer & bug reports](#).

There is nothing to fix on the Perl side. Capture the message, the input that triggered it, and the `ppperl` version, and file it.

``Unknown error``

(P) A fallback message for an error path that produced no specific diagnostic - most often surfaced from a native module (such as the GMP bignum backend) when an underlying library returned a failure code with no text. Treat it as an internal error and report it with the operation that triggered it.

See also

- [Disclaimer & bug reports](#) - how to file a `ppperl` bug, and what to include.
- [ppperl-specific diagnostics](#) - other messages with no upstream Perl counterpart.

ppperl-specific diagnostics

Messages PetaPerl emits that have no counterpart in upstream Perl. Most concern the daemon mode - `ppperl`’s persistent-process feature, which has no analogue in a stock perl - and a few native modules with `ppperl`-specific behavior.

Daemon mode

`ppperl` can run a script as a resident daemon and dispatch requests to it, which avoids paying interpreter startup on every call. The daemon diagnostics report problems with that channel.

`Daemon already running for this script (socket: %s)`

(F)

A `--daemon` start was requested for a script that already has a running daemon. The socket path is shown. Stop the existing daemon first, or connect to it instead of starting a new one.

`Cannot connect to daemon (%s): %s`

(F)

A client could not reach the daemon socket. The message continues with one of two hints depending on context - Start with: `ppperl --daemon <script>` when no daemon appears to be running, or `Is the daemon running?` when the socket exists but the connection failed. Start the daemon, or check that the socket path matches.

Native modules

`ERRNO hash is read only!`

(F)

Code tried to assign to `%! (the Errno symbolic-error hash)` or to one of its keys. That hash is populated by the runtime and reflects the current `errno`; it is not writable. Read it (`!{ENOENT}`) rather than assigning to it.

`Hash has key '%s' which is not in the new key set`

(F)

From `Hash::Util`: a `lock_keys` call supplied an explicit key set that does not include a key the hash already contains. Locking cannot proceed because that existing key would become inaccessible. Either include the existing key in the new set, or delete it before locking.

`Time::HiRes::usleep(%s): negative time not invented yet`

(F)

A `Time::HiRes` sleep function was given a negative duration. Time runs forward; pass a non-negative number of microseconds (or nanoseconds for `nanosleep`). The `nanosleep` form reports the same with its own name.

`%s version %s required--this is only version %s`

(F)

A `use Module VERSION` (or an explicit version check) demanded a newer version of a module than the one installed. The message names the module, the version requested, and the version present. Install or load the required version.

See also

- *Internal errors* - the panics that mean a pperl bug rather than a usage error.
- *Disclaimer & bug reports* - how pperl diverges from stock perl, and where to report problems.
- *Errno* - the module behind `%!` , the read-only symbolic-errno hash.
- *Hash::Util* - `lock_keys` and the restricted-hash machinery.
- *General fatal errors* - method resolution, references, arithmetic, restricted hashes, filehandles, and the rest of the runtime dies that are not regex- or parser-specific.
- *Regex compilation errors* - everything the pattern compiler rejects, from `Un-matched )` to malformed `\g`, `\N`, `\o`, and `(? . . )` constructs.
- *Compile-time syntax errors* - errors the parser and codegen raise before the program runs.
- *Warnings* - the non-fatal diagnostics: deep recursion, wide characters, useless constructs.
- *Value and type errors* - diagnostics about what a value is and is not: not a reference, not a code reference, division by zero, restricted-hash access.
- *Internal errors* - the panics that signal a bug in pperl itself.
- *ppperl-specific diagnostics* - messages PetaPerl emits that have no upstream Perl counterpart.

5.1.8 Security and taint mode

Security in a Perl program is mostly about not trusting data that came from outside the program: command-line arguments, the environment, file contents, network input. The dangerous step is not reading that data but *acting* on it. A filename that arrives from a web form is harmless until it reaches `open`; an environment variable is harmless until it steers a system call. This page describes the mechanism PetaPerl gives you to make that boundary explicit: taint mode. It explains what taint mode is, when it turns on, what it marks as untrusted, and how data is cleaned.

Taint mode

PetaPerl runs the perl5 5.42 core, so taint mode is the real perl5 taint machinery, not an approximation. Every check perl5 performs, PetaPerl performs.

Taint mode turns on automatically when PetaPerl detects that it is running with differing real and effective user or group IDs, which is how a `setuid` (Unix mode `04000`) or `setgid` (mode `02000`) script is run. You can also enable it explicitly with the `-T` command-line switch. `-T` is strongly recommended for any program run on behalf of someone else, a CGI script being the classic case. Once taint mode is on, it stays on for the rest of the program; it cannot be turned off from inside.

The switch must appear early enough that the interpreter sees it before it starts parsing your program. On the `#!` line of a script this is automatic. On the command line, put it among the first switches. The related `-t` switch enables the same checks but downgrades the fatal errors to warnings, which is useful when retrofitting taint mode onto

an existing program: you see every place that *would* fail without the program dying at the first one.

The read-only variable `${^TAINT}` reports the current state:

<code>\${^TAINT}</code>	Meaning
1	Taint mode on, fatal (-T)
-1	Taint checks on, warn only (-t)
0	Taint mode off

What gets tainted

When taint mode is on, any value that came from outside the program is marked tainted, and that mark propagates through every expression the value touches. The tainted sources are:

- command-line arguments (`@ARGV`) and the environment (`%ENV`);
- all file input;
- the results of `readdir` and `readlink`, the variable read by `shmread`, the messages from `msgrcv`, and the password, `gcov`, and shell fields returned by the `getpw*` calls;
- locale data (see `perllocale`).

Tainted data may not be used, directly or indirectly, in any operation that starts a sub-shell or that modifies a file, directory, or process. Using it in such an operation is fatal, with a message like `Insecure dependency in %s` or `Insecure $ENV{PATH}`. There are three exceptions:

- arguments to `print` and `syswrite` are **not** checked;
- symbolic method calls (`$obj->$method(@args)`) and symbolic subroutine references (`$foo->(@args)`) are **not** checked, so validate those names yourself if they can come from outside;
- hash keys are **never** tainted.

For efficiency, taint is tracked conservatively: if any part of an expression is tainted, the whole expression is treated as tainted, even where the tainted value does not actually influence the result. Taint attaches to individual scalar values, so some elements of an array or hash can be tainted while others are not.

```
my $arg  = shift;           # tainted (from @ARGV)
my $hid  = $arg . 'bar';    # also tainted
my $line = <STDIN>;         # tainted (file input)
my $data = 'abc';          # not tainted

system "echo $arg";         # dies: Insecure dependency
system "/bin/echo", $arg;   # dies: tainted argument
system "echo $data";       # dies until PATH is cleaned (see
↪below)
```

The one structural exception to whole-expression tainting is the ternary `?:` operator. Because

```
my $result = $tainted ? "untainted" : "also untainted";
```

is just a conditional choice between two constants, `$result` is not tainted even though the condition is.

Laundering and detecting tainted data

The **only** way to clear the taint on a value is to extract a substring of it through a regular-expression capture. PetaPerl assumes that if you bother to write a pattern and pull out `$1`, you have thought about what you are willing to accept. Untaint by validating against a whitelist of acceptable characters, never by trying to strip out the bad ones:

```
if ($data =~ /^([-@\w.]+$)/) {
    $data = $1;                # $data is now untainted
} else {
    die "rejected suspicious input";
}
```

This is reasonably safe because `\w`, a dot, a dash, and an at sign do not carry special meaning to a shell. A pattern like `/(.+)/` would “launder” the value while accepting anything, which defeats the purpose. Be deliberate about the pattern.

There is a locale caveat. Under `use locale` the set of characters that `\w` matches is itself drawn from the locale, which is external data PetaPerl does not trust, so a `\w` match under `use locale` does **not** untaint. If you need to launder with `\w` in a locale-aware program, put `no locale` ahead of the match in the same block.

To test whether a value is tainted, use `tainted` from the `Scalar::Util` module. Under `-T` it correctly reports the taint state of its argument.

Cleaning up %ENV and \$ENV{PATH}

A tainted `PATH` is the most common surprise. Because PetaPerl cannot know that a program it runs will not itself consult `PATH`, it refuses to start a sub-process while `$ENV{PATH}` is tainted, with `Insecure $ENV{PATH}`. Every directory in the path must be absolute and writable only by its owner and group; a writable directory yields `Insecure directory in %s`. Note that an empty path component is treated as `.` (the current directory) and also triggers the message.

`PATH` is not the only offender. Because a shell may consult `IFS`, `CDPATH`, `ENV`, and `BASH_ENV`, PetaPerl requires those to be empty or untainted before it will start a sub-process, reporting `Insecure $ENV{%s}` otherwise. Set a known-good `PATH` and delete the shell variables early:

```
$ENV{PATH} = '/bin:/usr/bin';
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};
```

When you must open a file or pipe from a privileged (`setuid` or `setgid`) program, the safe pattern is to fork a child, have the child drop its privileges back to the real user and group, and let the unprivileged child do the work and pass the result back over a pipe:

```
use English;
die "can't fork: $!" unless defined(my $pid = open(my $kid, "-|"));
if ($pid) {                                     # parent
    while (<$kid>) {
        # consume the child's output
    }
    close $kid;
} else {                                       # child
    my @priv = ($EUID, $EGID);
    my $real_uid = $UID;
    my $real_gid = $GID;
    $EUID = $UID;
    $EGID = $GID;
    $UID = $real_uid;                         # drop privileges
    $GID = $real_gid;
    ($EUID, $EGID) = @priv;                   # make sure they are gone
    die "can't drop privileges"
        unless $UID == $EUID && $GID == $EGID;
    $ENV{PATH} = '/bin:/usr/bin';
    exec 'myprog', 'arg1', 'arg2'
        or die "can't exec myprog: $!";
}
```

Note that `exec` is called with a list, never a single string, so no shell is involved to re-expand the arguments. This is the safest way to invoke an external command. The same fork-and-drop strategy works for `glob` and backticks, which (unlike `system` and `exec`) offer no list-form calling convention of their own.

Taint mode and @INC

Under `-T`, PetaPerl ignores the `PERL5LIB`, `PERLLIB`, and `PERL_USE_UNSAFE_INC` environment variables. They are ignored because they are easy to set without a user noticing, which would let the environment inject library directories silently. You can still adjust the module search path with the visible `-I` switch or the `lib` pragma (`-Mlib=/path`), because those appear plainly on the command line.

The current directory `.` is **not** in the default `@INC`, taint mode or not. (`PERL_USE_UNSAFE_INC=1` would re-add it, but that variable is itself ignored under `-T`, so `.` stays out of `@INC` for any tainted program.)

Set-id scripts and the shebang race

PetaPerl detects the `setuid`/`setgid` condition (real user or group ID differing from the effective one) and enables taint mode accordingly; the “while running `setuid`” / “while running `setgid`” wording in the insecurity messages comes from that detection. PetaPerl does **not**, however, ship a `setuid` `perl` install or any set-id helper. If you need a privileged Perl program, the supported approach on Linux is a small dedicated `setuid` wrapper that sanitises the environment and then execs the interpreter on a fixed, non-writable script path, rather than relying on a `setuid` script directly. The classic shebang race, where the kernel and the interpreter open the script file at two different

moments and an attacker swaps the file in between, is a property of kernel set-id-script handling, not of the interpreter, and the wrapper approach sidesteps it.

Algorithmic complexity attacks

Some internal algorithms can be driven to consume disproportionate time or memory by carefully chosen input, which is a denial-of-service vector independent of taint mode. The threat classes:

- **Hash collisions.** An attacker who can predict the hash seed can craft keys that all land in one bucket, degrading hash operations. pperl randomises the hash seed at process start and randomises the visible order of `keys`, `values`, and `each` per hash. Two environment variables override these defaults for debugging: `PERL_HASH_SEED` pins the seed (which controls storage, not the presented order) and `PERL_PERTURB_KEYS` controls traversal-order perturbation; the `Hash::Util` function `hash_traversal_mask` does the same per hash. These overrides weaken the protection and are not advised in production. PetaPerl has never guaranteed any hash key ordering; do not rely on it, and do not use the pseudo-random ordering as a shuffle.
- **Catastrophic regex backtracking.** The regular-expression engine is an NFA and can take time or space that grows steeply when a pattern can match an input in many ways. A hostile input matched against a vulnerable pattern can exhaust memory or CPU. Bound the input size and prefer patterns that cannot backtrack pathologically.
- **Sort.** `sort` uses mergesort, whose worst case is bounded, so it cannot be driven into quadratic behaviour by adversarial input.

Tied hashes may carry their own ordering and their own complexity characteristics.

Differences from upstream

- **Linux only.** PetaPerl runs only on Linux. The VMS, Windows, and other-platform set-id and security discussion that upstream Perl carries has no counterpart here and is simply absent.
- **No “protection” tooling.** PetaPerl has no bytecode compiler and no source-filter obfuscation machinery, so the upstream advice about shipping compiled or filtered source does not apply. Obscuring source was never a security measure in any case.
- **Safe compartment not available yet.** The Safe module (and the Opcode module it builds on) is not available in PetaPerl yet; loading it today fails with dynamic loading not available in this perl. Until it is, do not rely on the “run untrusted code in a restricted compartment” approach: treat untrusted *code* (as opposed to untrusted *data*) as something to run in a separate, OS-level sandbox or process with dropped privileges, not inside the interpreter.

See also

- `perlvar` - `${^TAINT}`, `%ENV`, and the real/effective ID variables `$<` `$>` `$(< $>)` that decide whether taint mode auto-enables.
- `perllocale` - why locale data is untrusted and how `use locale` interacts with `\w` laundering.

- `perlop` - the `?:` operator and its exception to whole-expression tainting.
- *Command-line switches* - the `-T`, `-t`, and `-I` switches in context.
- *Command-line options* - how `ppperl` parses its switches, including the runtime selector.
- `Scalar::Util` - tainted, for probing the taint state of a value.
- `Hash::Util` - `hash_traversal_mask`, for per-hash control of traversal-order randomisation.

5.2 Modules

Built-in modules, by top-level namespace. Badges show the feature tier and any extra Cargo features required — see *Disclaimer & bug reports* for how to check which features your `ppperl` binary has.

- `B` — Inspect a running Perl program's own optree, symbol table, and compiled subs.
- `Clone` — Make a deep, independent copy of any Perl value - hashes, arrays, scalars, references,
- `Config` — Look up the build-time configuration of the running Perl interpreter.
- `Cwd` — Report the current working directory and resolve paths to their
- `Data::[ Dumper ]`
- `Devel::[ Peek ]`
- `Digest::[ MD5, SHA ]`
- `Encode` — Convert between Perl character strings and bytes in any named encoding -
- `Errno` — Symbolic names for the `errno(3)` values your system uses - `EINTR`, `EAGAIN`,
- `Fcntl` — Symbolic constants for `fcntl(2)`, `flock(2)`, `seek(2)`, and file-mode
- `File::[ Glob, Map, Temp ]`
- `Filter::[ Util ]`
- `Hash::[ Util ]`
- `I18N::[ Langinfo ]`
- `IO` — Load the core IO modules - `IO::Handle`, `IO::File`, `IO::Seekable`, `IO::Pipe`, `IO::Socket`, `IO::Dir` - in one use.
- `List::[ Util ]`
- `MIME::[ Base64, QuotedPrint ]`
- `Math::[ GMP ]`
- `PDL` — PDL - the Perl Data Language: fast, vectorised, multidimensional arrays for scientific and bulk numeric work.
:: `[ Bad, Basic, Constants, Core, FFT, Lite, LiteF, Math, MatrixOps, Ops, Primitive, Slices, Types, Ufunc ]`

- `POSIX` — The POSIX (IEEE Std 1003.1) C standard library, exposed to Perl.
- `PadWalker` — Reach into another subroutine's my and our variables - read them,
- `PerlIO` :: [`encoding`, `mmap`, `via`]
- `Peta` :: [`FFI`]
- `SDBM_File` — Tie a Perl hash to an on-disk SDBM database so keys and values persist between runs.
- `Scalar` :: [`Util`]
- `Socket` — BSD socket constants and address pack/unpack helpers that feed Perl's
- `Storable` — Serialise arbitrary Perl data structures to bytes, reconstruct them later, or deep-clone them in memory.
- `Sys` :: [`Hostname`]
- `Term` :: [`ReadLine`]
- `Time` :: [`HiRes`]
- `attributes` — The machinery behind ``sub foo : lvalue method { ...`
- `mro` — Method Resolution Order - how Perl decides which parent class's method a call inherits.
- `re` — The `re` pragma and regular-expression introspection module.
- `version` — Turn a version string into an object you can compare, print, and introspect.

5.2.1 B

Inspect a running Perl program's own optree, symbol table, and compiled subs.

B is the compiler back-end interface. It hands a script introspective access to the data structures the interpreter already holds: the root of the main optree, every compiled sub's CV, the symbol table (stash) graph, and the SV/AV/HV/GV objects those structures point at. Subclasses such as `B::OP`, `B::CV`, `B::GV`, `B::SV` (and its type-specific children) wrap those pointers as blessed Perl objects with accessor methods.

Common callers are back-end modules like `B::Deparse`, `B::Concise`, and `B::Xref`; debugging tooling that walks the optree; and scripts that introspect the symbol table to generate code or diagnostics.

Functions

Entry points

`svref_2object`

Wrap a reference's referent as a `B::*` object.

opnumber

Return the numeric op-type for an op name, or -1 if unknown.

ppname

Return the pp__{pp}-prefixed name for a numeric op-type.

Synopsis

```
my $name = B::ppname($op->type);    # e.g. "pp_print"
```

Inverse of B::opnumber. Returns undef if the number is out of range.

sv_undef

Return a B::SPECIAL wrapping perl's shared PL_sv_undef.

sv_yes

Return a B::SPECIAL wrapping perl's shared PL_sv_yes.

sv_no

Return a B::SPECIAL wrapping perl's shared PL_sv_no.

main_root

Return a B::OP for the root of the program's main optree.

main_start

Return a B::OP for the first op executed in the main program.

Synopsis

```
my $start = B::main_start;
```

Where B::main_root gives the syntactic root, B::main_start gives the head of the linked list perl follows at run time via op_next.

cast_I32

Truncate \$i to a signed 32-bit integer.

Synopsis

```
B::cast_I32(0x1_0000_0001);    # 1 (low 32 bits, sign-extended)
```

Exists so Perl back-ends can reproduce perl's own I32 truncation when they need to emit exact values for an op field typed I32 in C.

amagic_generation

Return the overload-magic generation counter (always 0 in modern perl).

address

Return the memory address of an SV as an integer.

Synopsis

```
my $addr = B::address(\$x);
```

Mostly useful as an identity key: two `B::*` objects wrapping the same underlying SV will report the same address.

threadsv_names

Return the list of thread-local special-variable names (empty).

parents

Return a reference to `@B::parents`.

Walking the optree

walkoptree

Recursively walk an optree, calling `$method` on each op.

walkoptree_debug

Get or set the walkoptree debug flag.

String and hash helpers

cstring

Return `$sv`'s contents as a double-quoted C-style string literal.

perlstring

Return `$sv`'s contents as a double-quoted Perl string literal.

Synopsis

```
B::perlstring('say $x@y');    # "say \$x\@y"
```

Like `B::cstring`, but also escapes `$` and `@` so the result can be pasted back into Perl source without interpolation surprises.

cchar

Return the first character of `$sv` as a single-quoted C character literal.

hash

Return perl's hash of a string, formatted as `0xNNNNNNNNN`.

B::SV methods

sv_refcnt

Return the reference count of the wrapped SV.

sv_flags

Return the raw SvFLAGS word of the wrapped SV.

sv_svttype

Return the SV type as a numeric SVt_* code.

object_2svref

Return a Perl reference back to the SV wrapped by a `B : *` object.

B::IV methods

iv_sviv

Return the integer value of the wrapped SV, stringifying if necessary.

iv_ivx

Return the IV slot of the wrapped SV without coercion.

iv_uvz

Return the UV slot of the wrapped SV without coercion.

B::NV methods

nv_svnv

Return the floating-point value of the wrapped SV, coercing if needed.

nv_nv

Return the NV slot of the wrapped SV without coercion.

B::PV methods**pv_pv**

Return the string contents of a PV SV, preserving SVf_UTF8.

pv_pvx

Return the PV buffer contents as a C-style NUL-terminated string.

pv_cur

Return the current length in bytes of the PV buffer (SvCUR).

pv_len

Return the allocated buffer size in bytes (SvLEN).

B::CV methods**cv_gv**

Return the B::GV that a named CV belongs to.

cv_stash

Return the B::HV for the package the CV was compiled in.

cv_file

Return the source filename the CV was compiled from.

Synopsis

```
print $cv->FILE;      # e.g. "MyModule.pm"
```

Returns an empty string for XSUBs that do not carry a file record.

cv_root

Return the root op of the CV's optree, or B::NULL for an XSUB.

Synopsis

```
my $root = $cv->ROOT;
```

XSUBs carry compiled C code and have no optree, so ROOT returns a B::NULL. Test with \$cv->CvFLAGS & B::CVf_ISXSUB first if it matters.

cv_start

Return the first op executed in the CV's body, or B : :NULL for an XSUB.

cv_depth

Return the current recursion depth of the sub.

cv_cvflags

Return the raw CvFLAGS word of the CV.

cv_name_hek

Return the CV's name when it was stored as a HEK rather than via a GV.

cv_outside

Return the lexically enclosing CV (CvOUTSIDE).

cv_padlist

Return the CV's padlist as a B : :PADLIST, or B : :NULL for an XSUB.

cv_outside_seq

Return the COP sequence number at which the CV was closed over.

B::GV methods

gv_name

Return the GV's unqualified name.

gv_safename

Return the GV's name with control characters rendered as ^X.

gv_file

Return the source file that first introduced the GV.

Synopsis

```
print $gv->FILE;      # e.g. "MyModule.pm"
```

The filename perl recorded when the glob was created or first accessed as an lvalue. An empty string for globs with no recorded origin.

gv_stash

Return the B : :HV for the stash that owns the GV.

gv_sv

Return the scalar slot of the GV as a B : :SV.

gv_av

Return the array slot of the GV as a B : :AV.

gv_hv

Return the hash slot of the GV as a B : :HV.

gv_cv

Return the code slot of the GV as a B : :CV.

gv_egv

Return the “effective” GV (GvEGV) - for aliased globs, the canonical one.

gv_refcnt

Return the refcount of the GV’s GP (slot-bundle) record.

gv_line

Return the source line where the GV was first introduced.

gv_isgv_with_gp

Return true if the SV is a GV that has a GP slot-bundle attached.

gv_is_empty

Return true if the GV has no slot-bundle (GP) at all.

gv_flags_wrapper

Return the raw SvFLAGS word of the GV (same as B : :SV : :FLAGS).

gv_gvflags

Return the GV-specific flag bits.

gv_svtype

Return the SV type code (same as `B::SV::SvTYPE`).

B::HV methods

hv_name

Return the package name of a stash HV.

B::OP methods

op_next

Return the next op in execution order (`op_next`).

op_sibling

Return the next syntactic sibling under the same parent.

op_targ

Return the op's pad target index (`op_targ`).

op_flags

Return the op's shared flag byte (`op_flags`).

op_private

Return the op's private flag byte (`op_private`).

op_first

Return the first child of a UNOP (or any op with `OPf_KIDS`).

op_name

Return the short name for the op type (e.g. "print", "aelem").

op_desc

Return the human-readable description of the op type.

Synopsis

```
print $op->desc;      # e.g. "array element"
```

Longer and more natural than `B::OP::name`. Used in diagnostic messages such as “Can’t modify X in Y”.

op_ppaddr

Return a C-source-style reference to the op's pp function.

op_type

Return the numeric op-type code.

op_opt

Return the op_opt bit: non-zero once the peephole optimiser has run.

op_moresib

Return the op_moresib bit: true if the op has another sibling after it.

op_parent

Return the syntactic parent of the op.

op_children

Return the number of children in a LISTOP.

Synopsis

```
print $listop->children;    # integer count
```

Counted by walking from first through sibling. Trivially 0 for a LISTOP with no kids; most real LISTOPs have at least one.

Interpreter globals

intrpvar_main_cv

Return a B: :CV for the implicit CV of the main program.

intrpvar_inc_gv

Return a B: :GV for *INC, the module-loading search list.

intrpvar_defstash

Return a B: :HV for %main::, the default stash.

intrpvar_curstash

Return a B: :HV for the currently active compile-time stash.

intrpvar_warnhook

Return a B: :SV for the current \$SIG{__WARN__} handler.

intrpvar_diehook

Return a B: :SV for the current \$SIG{__DIE__} handler.

intrpvar_initav

Return a B: :AV of the INIT blocks queued for execution.

intrpvar_checkav

Return a B: :AV of CHECK blocks queued for execution.

intrpvar_unitcheckav

Return a B: :AV of UNITCHECK blocks queued for execution.

intrpvar_beginav

Return a B: :AV of BEGIN blocks already executed.

intrpvar_endav

Return a B: :AV of END blocks queued to run at interpreter exit.

svref_2object

Wrap a reference's referent as a B: : * object.

Synopsis

```
my $obj = B::svref_2object(\&some_sub);  
my $cv  = B::svref_2object(\&main::foo);  # returns B::CV  
my $av  = B::svref_2object(\@ARGV);      # returns B::AV
```

Given a Perl reference, return a blessed B: : * object that wraps the underlying SV, AV, HV, CV, GV, etc. Croaks with argument `is not a reference` if the input is not a reference. This is the usual entry point into B: most introspection starts from a reference the caller already has.

opnumber

Return the numeric op-type for an op name, or -1 if unknown.

Synopsis

```
my $n = B::opnumber("print");    # numeric type
my $n = B::opnumber("pp_print");  # leading "pp_" is tolerated
```

Accepts either the bare name or the pp_-prefixed form. The returned integer is an index into perl's global op-name / op-desc tables and is the value B::OP::type produces.

main_root

Return a B::OP for the root of the program's main optree.

Synopsis

```
my $root = B::main_root;
$root->name;    # typically "leave"
```

The root op is the top of the tree that B::Deparse and friends walk. Paired with B::main_start, which gives the entry point in execution order.

walkoptree

Recursively walk an optree, calling \$method on each op.

Synopsis

```
B::walkoptree(B::main_root, "visit");
```

Each op is blessed into its B::*OP class and then \$method is dispatched on it. Children are visited after the parent. Back-end classes such as B::Concise define a visit method on their own package and call walkoptree to drive tree-traversal without writing the walk themselves.

walkoptree_debug

Get or set the walkoptree debug flag.

Synopsis

```
my $prev = B::walkoptree_debug(1);    # enable
B::walkoptree_debug($prev);           # restore
```

When enabled, B::walkoptree also calls a walkoptree_debug method on each node before the regular visit method, which lets a back-end print diagnostics as it walks. Returns the previous flag value.

cstring

Return \$sv's contents as a double-quoted C-style string literal.

Synopsis

```
B::cstring("hi\\tthere");    # "hi\\tthere"
B::cstring(undef);           # 0
```

Unprintable bytes become `\n`, `\t`, `\r`, `\a`, `\b`, `\f`, `\v`, or octal escapes. An `undef` SV renders as the literal `0` (no quotes), which is the convention `B::Deparse` depends on. Used by back-ends that emit C-level source.

cchar

Return the first character of \$sv as a single-quoted C character literal.

Synopsis

```
B::cchar("A");              # 'A'
B::cchar("\n");              # '\n'
```

Non-printable bytes are escaped the same way as in `B::cstring`. Only the first byte is inspected.

hash

Return perl's hash of a string, formatted as `0xNNNNNNNN`.

Synopsis

```
my $h = B::hash("foo");     # e.g. "0x2ad53e4d"
```

Uses the same hash function the interpreter uses for hash keys, so the value is useful for reasoning about bucket placement and hash collisions in a live hash, not as a general-purpose checksum.

sv_refcnt

Return the reference count of the wrapped SV.

Synopsis

```
my $sv = B::svref_2object(\%x);
print $sv->REFCNT;
```

Calling `REFCNT` does not itself alter the count the `B::*` wrapper reports (the wrapper holds a weak handle, not a strong reference).

sv_svtype

Return the SV type as a numeric SVt_* code.

Synopsis

```
if ($sv->SvTYPE == B::SVt_PVCV) { ... }
```

Compare against B::SVt_NULL, B::SVt_IV, B::SVt_PVCV, etc. The blessed class of the wrapper already encodes the type at a coarser granularity; SvTYPE is the precise value.

object_2svref

Return a Perl reference back to the SV wrapped by a B::* object.

Synopsis

```
my $b = B::svref_2object(\$x);
my $ref = $b->object_2svref; # same as \$x
```

Inverse of B::svref_2object. Useful when a back-end has descended into a sub-structure via B::* methods and needs a plain Perl reference to the SV it's now looking at.

pv_pv

Return the string contents of a PV SV, preserving SVf_UTF8.

Synopsis

```
my $pv = B::svref_2object("hello");
print $pv->PV;
```

Returns the full SvCUR bytes, so embedded \0 are kept. An SV with no PV slot returns an empty string.

cv_gv

Return the B::GV that a named CV belongs to.

Synopsis

```
my $cv = B::svref_2object(&main::foo);
my $gv = $cv->GV; # B::GV for *main::foo
print $gv->NAME; # "foo"
```

Anonymous and lexical subs may return a B::SPECIAL or a sentinel rather than a real B::GV; check with isa before calling NAME.

cv_name_hek

Return the CV's name when it was stored as a HEK rather than via a GV.

Synopsis

```
my $name = $cv->NAME_HEK;      # undef unless CVf_NAMED is set
```

Modern perl stores some sub names directly in a hash-key slot on the CV (cheaper than a full GV). If the CV does not use that slot, returns undef; callers generally fall back to `$cv->GV->NAME`.

gv_name

Return the GV's unqualified name.

Synopsis

```
my $gv = B::svref_2object(\*main::foo);  
print $gv->NAME;      # "foo"
```

The name alone, without the stash prefix. For `*main::foo` this is "foo"; combine with `B::GV::STASH->NAME` to recover the full name.

gv_safename

Return the GV's name with control characters rendered as `^X`.

Synopsis

```
B::svref_2object(\*{"main::\cA"})->SAFENAME;    # "^A"
```

The single-character control names (`^A`, `^D`, `^?`) appear in the stash with raw control bytes in their name; `SAFENAME` renders them in the caret form normally used in Perl source.

gv_sv

Return the scalar slot of the GV as a `B::SV`.

Synopsis

```
our $x = 42;  
my $gv = B::svref_2object(\*main::x);  
print $gv->SV->SvIV;      # 42
```

Equivalent to `*foo{SCALAR}` in Perl, but wrapped as a `B::SV`.

hv_name

Return the package name of a stash HV.

Synopsis

```
my $stash = B::svref_2object(\%main:);
print $stash->NAME;      # "main"
```

Meaningful only on stash hashes (those accessible as %Pkg: :). A plain user hash returns an empty string.

op_ppaddr

Return a C-source-style reference to the op's pp function.

Synopsis

```
print $op->ppaddr;      # e.g. "PL_ppaddr[OP_PRINT]"
```

Intended for back-ends that emit C source (such as B: :C), which need to refer to pp functions by their slot in the PL_ppaddr dispatch table.

5.2.2 Clone

Make a deep, independent copy of any Perl value - hashes, arrays, scalars, references, blessed objects, tied variables, and weak references.

clone walks a data structure recursively and returns a fresh copy. Shared references inside the input become shared references inside the copy (no accidental duplication), circular structures are reproduced without infinite recursion, weak references come out still weak, and tied variables keep their tie. Modifying the clone never touches the original.

For pure-Perl deep copies, Clone is typically faster than Storable::dclone on shallow to medium-depth structures (roughly three levels or fewer).

Synopsis

```
use Clone qw(clone);

my $copy = clone($original);      # function form

package Foo;
use parent 'Clone';
my $other = $obj->clone;           # method form (inherited)
```

Functions

Cloning

clone

Return a deep, independent copy of a Perl value.

clone

Return a deep, independent copy of a Perl value.

Synopsis

```
use Clone qw(clone);

my $copy = clone($thing);           # function form
my $copy = clone($thing, 5);        # limit recursion depth
my $copy = $obj->clone;              # method form (class inherits
↳ from Clone)
```

What you get back

A value of the same kind as the input. The copy is fully independent: assigning into \$copy - at any nesting level - never changes \$thing.

- Shared references inside the input are de-duplicated: if two slots pointed at the same referent, the corresponding two slots in the copy point at *one* fresh clone of that referent, not two separate clones.
- Circular structures are reproduced with their cycles intact.
- Weak references (see `Scalar::Util::weaken`) come out weak in the copy.
- Tied variables keep their tie; the tie's internal object is itself cloned.
- Blessed objects are reblessed into the same package as the original.

Examples

Basic hash clone - the copy is independent:

```
use Clone qw(clone);
my $h = { name => 'Alice', tags => [qw(admin user)] };
my $c = clone($h);
push @{$c->{tags}}, 'guest';
print "@{ $h->{tags} }\n";           # admin user
print "@{ $c->{tags} }\n";           # admin user guest
```

Shared references stay shared inside the clone:

```

my $shared = { count => 0 };
my $pair   = { a => $shared, b => $shared };
my $copy   = clone($pair);
$copy->{a}{count} = 42;
print $copy->{b}{count};           # 42  (a and b still point at one_
    ↪ hash)
print $pair->{a}{count};           # 0   (original untouched)

```

Circular structures:

```

my $a = { name => 'A' };
my $b = { name => 'B', peer => $a };
$a->{peer} = $b;           # cycle
my $copy = clone($a);
print $copy->{peer}{peer}{name}; # A  (cycle reproduced)

```

Edge cases

- undef clones to undef.
- A non-reference scalar (42, "hi") clones by value - you get a fresh scalar with the same contents.
- Code references are shared, not duplicated: the clone points at the same sub.
- Filehandles and IO objects are cloned but share the underlying file descriptor and file position.
- The depth argument caps recursion. `clone($thing, 0)` returns `$thing` unchanged (refcount bump, no copy). A negative or omitted depth means “unlimited”; the runtime still guards against stack overflow by falling back to an iterative cloner once `rdepth` passes 4000.

Differences from upstream

Fully compatible with upstream Clone 0.50.

See also

- `Storable::dclone` - serializer-based deep copy; typically faster for structures 4+ levels deep, slower for shallow ones.
- `Scalar::Util::weaken` - create the weak references that `clone` preserves.
- `Clone::PP` - pure-Perl implementation of the same interface.

5.2.3 Config

Look up the build-time configuration of the running Perl interpreter.

`Config` exposes the values recorded when Perl itself was compiled - platform, compiler flags, install paths, feature toggles. Scripts consult it to make portability decisions, locate installed libraries, or report interpreter characteristics.

Two surfaces are provided:

- The tied hash %Config, accessed element-wise as \$Config{key}. Lookup, existence, and iteration go through the tie hooks. The hash is read-only; STORE, DELETE, and CLEAR croak.
- Programmatic queries: config_re, config_sh, config_vars, myconfig, compile_date, bincompat_options, non_bincompat_options, local_patches, header_files, and the perl -V backend_V.

Under pperl, every value reflects pperl's own build, not any upstream perl's. Callers using %Config entries to drive portability decisions (compiler name, archname, install prefix, feature defines) get answers that match the interpreter they are actually running.

Functions

Hash access

fetch

Tied-hash read: return the value stored under a key.

exists

Tied-hash membership test: does a key exist in %Config?

store

Tied-hash write: always croaks; %Config is read-only.

delete

Tied-hash delete: always croaks; %Config is read-only.

Synopsis

```
delete $Config{archname};    # dies
```

What you get back

Nothing. The call raises Config is read-only.

Differences from upstream

Fully compatible with upstream.

clear

Tied-hash clear: always croaks; %Config is read-only.

Synopsis

```
%Config = ();                # dies
```


What you get back

Nothing. The call raises `Config` is read-only.

Differences from upstream

Fully compatible with upstream.

`firstkey`

Tied-hash iterator: reset and return the first key.

`nextkey`

Tied-hash iterator: return the next key after FIRSTKEY.

Querying

`myconfig`

Return a human-readable summary of the interpreter's build.

`config_sh`

Return the full configuration as one shell-assignment block.

`config_vars`

Print selected configuration entries to STDOUT.

`config_re`

Grep configuration entries whose key matches a pattern.

Introspection

`compile_date`

Return a one-line description of when and with what the interpreter was built.

`local_patches`

Return the list of local patches applied to the interpreter.

`bincompat_options`

Return the compile-time options that affect binary compatibility.

non_bincompat_options

Return compile-time options that do not affect binary compatibility.

header_files

Return the canonical list of Perl C header filenames.

_v

Print the full perl -V report to STDOUT.

myconfig

Return a human-readable summary of the interpreter's build.

Synopsis

```
use Config;  
print Config::myconfig();
```

What you get back

A single multi-line string, suitable for printing verbatim. Groups the values under Platform, Compiler, and Linker and Libraries headings, matching the shape expected by tools that scrape perl -V-style output.

Examples

```
use Config;  
my $summary = Config::myconfig();  
print $summary if $summary =~ /osname=linux/;
```

Edge cases

- Returns a fresh scalar on every call; cache it if used repeatedly in a hot path.

Differences from upstream

- Platform, compiler, and library values describe pperl's build (Rust toolchain, pperl version line) rather than a C-compiled perl. The layout and key names match upstream so existing parsers keep working.

config_sh

Return the full configuration as one shell-assignment block.

Synopsis

```
use Config;
my $sh = Config::config_sh();
```

What you get back

A single scalar containing one line per entry in the form `key='value'`, newline-terminated. The output is stable across calls and suitable for feeding into a shell `eval` or a line-based parser.

Examples

```
use Config;
for my $line (split /\^/, Config::config_sh()) {
    print $line if $line =~ /\^archname=/;
}
```

Edge cases

- Entries whose value is undef render as `key= ' '`, matching how the tied hash reports them via `EXISTS` (true) and `FETCH` (undef).

Differences from upstream

Fully compatible with upstream.

config_vars

Print selected configuration entries to STDOUT.

Synopsis

```
use Config;
Config::config_vars(qw(archname osname ivsize));
```

What you get back

Nothing. Each named key is written to STDOUT on its own line as `key='value'`; . Unknown keys render with an empty value.

Examples

```
use Config;
Config::config_vars('archname');           # archname='x86_64-linux
↪ ' ;
Config::config_vars('archname', 'osname'); # two lines
```

Edge cases

- Empty key names are skipped silently.
- Output goes straight to the OS STDOUT; redirecting with `local *STDOUT` in Perl does not affect it.

Differences from upstream

Fully compatible with upstream.

fetch

Tied-hash read: return the value stored under a key.

Synopsis

```
my $arch = $Config{archname};    # calls FETCH
```

What you get back

The string value for the given key, or `undef` if the key is either absent or recorded as `undef` (for example `d_pseudofork`).

Examples

```
use Config;
print $Config{osname};      # linux
print $Config{archname};    # x86_64-linux
```

Edge cases

- An empty string key returns `undef`.
- Missing keys and keys whose recorded value is `undef` are indistinguishable through `FETCH` alone; use `EXISTS` to tell them apart.

Differences from upstream

Fully compatible with upstream.

exists

Tied-hash membership test: does a key exist in `%Config`?

Synopsis

```
if (exists $Config{useithreads}) { ... }
```

What you get back

A true value when the key is part of the configuration (even if the recorded value is undef), false otherwise.

Examples

```
use Config;
print exists $Config{d_fork}      ? "yes" : "no";  # yes
print exists $Config{nonesuch}    ? "yes" : "no";  # no
print exists $Config{useithreads} ? "yes" : "no";  # yes (value is_
↪undef)
```

Edge cases

- Pair EXISTS with FETCH to distinguish “key is absent” from “key is present but undef”.

Differences from upstream

Fully compatible with upstream.

store

Tied-hash write: always croaks; %Config is read-only.

Synopsis

```
$Config{archname} = 'foo';  # dies
```

What you get back

Nothing. The call raises Config is read-only and unwinds to the nearest eval.

Examples

```
eval { $Config{archname} = 'bogus' };
print $@;                      # Config is read-only
```

Edge cases

- The croak fires regardless of the key or value; there is no per-key override.

Differences from upstream

Fully compatible with upstream.

firstkey

Tied-hash iterator: reset and return the first key.

Synopsis

```
my @keys = keys %Config;           # triggers FIRSTKEY + NEXTKEY
while (my ($k, $v) = each %Config) { ... }
```

What you get back

The first key in the internal iteration order, or undef if the hash is empty.

Edge cases

- Iteration order is unspecified; never rely on it.
- Calling FIRSTKEY resets a NEXTKEY walk in progress.

Differences from upstream

Fully compatible with upstream.

nextkey

Tied-hash iterator: return the next key after FIRSTKEY.

Synopsis

```
while (my ($k, $v) = each %Config) { ... }
```

What you get back

The next key in the current iteration, or undef when the walk is exhausted.

Edge cases

- Calling NEXTKEY without a prior FIRSTKEY starts from wherever the hash's internal iterator happens to be.

Differences from upstream

Fully compatible with upstream.

config_re

Grep configuration entries whose key matches a pattern.

Synopsis

```
use Config;
my @hits = Config::config_re('^cc');
```

What you get back

A list of strings in key='value' form, one per matching entry. The pattern is applied to the key portion only; values are included verbatim in the returned strings.

Examples

```
use Config;
my @arch = Config::config_re('arch'); # archname, archlibexp, ...
my @d    = Config::config_re('^d_');  # every d_* feature flag
```

Edge cases

- An empty pattern returns no results.
- Non-UTF-8 pattern bytes return no results.

Differences from upstream

- Patterns containing only literal characters match as substrings against the key. Patterns that use regex metacharacters (^, \$, ., *, +, ?, |, (, [, \)) are treated as regexes with a substring fallback. The common idioms ('^cc', 'arch', '^d_') behave the same as upstream; exotic lookarounds may differ. Covered by t/81-xs-native/Config/*.t.

compile_date

Return a one-line description of when and with what the interpreter was built.

Synopsis

```
use Config;
print Config::compile_date(), "\n";
```

What you get back

A scalar string such as `Compiled with rustc 1.xx (...)`. The exact format is stable enough to log but not intended for machine parsing; prefer `%Config` entries for structured fields.

Differences from upstream

- Describes the Rust toolchain that built pperl rather than a C compiler invocation date. Callers that only print the string are unaffected; callers that parse it for a specific date layout need to adapt.

local_patches

Return the list of local patches applied to the interpreter.

Synopsis

```
use Config;
my @patches = Config::local_patches();
```

What you get back

A list of strings, one per local patch. The list is empty when the interpreter was built from pristine sources.

Examples

```
use Config;
print "vanilla build\n" unless Config::local_patches();
```

Differences from upstream

- Always returns an empty list under pperl. pperl is not a patched fork of upstream perl, so there is no patch trail to report.

bincompat_options

Return the compile-time options that affect binary compatibility.

Synopsis

```
use Config;
my @opts = Config::bincompat_options();
```


What you get back

A list of option names (USE_64_BIT_INT, USE_PERLIO, and so on) representing decisions baked into the interpreter that would break loading of incompatibly-built XS binaries.

Examples

```
use Config;
print "64-bit ints\n"
    if grep { $_ eq 'USE_64_BIT_INT' } Config::bincompat_options();
```

Differences from upstream

- The list reports the options that truthfully apply to pperl's build. Upstream-specific switches that do not exist in pperl are omitted rather than forged.

non_bincompat_options

Return compile-time options that do not affect binary compatibility.

Synopsis

```
use Config;
my @opts = Config::non_bincompat_options();
```

What you get back

A list of option names (PERL_COPY_ON_WRITE, USE_LOCALE, and so on) describing behavioural switches that do not change the ABI. Useful for feature-probing scripts that want to know whether a particular semantics is enabled without caring about XS compatibility.

Differences from upstream

- Reports only the switches that meaningfully apply to pperl's build.

header_files

Return the canonical list of Perl C header filenames.

Synopsis

```
use Config;
my @hdrs = Config::header_files();
```

What you get back

A fixed list of bare header filenames (`perl.h`, `sv.h`, `op.h`, and so on) - no paths, no extensions other than `.h`. The call exists so tooling that probes for XS build prerequisites keeps working.

Edge cases

- The filenames name headers from the upstream perl C sources. `ppperl` does not ship them as separate files; callers that read the headers from disk will fail to find them.

Differences from upstream

- `ppperl` does not have a C-level public header layout, so the list is informational only. It matches upstream's roster so scripts that merely inspect the names (without opening the files) behave identically.

`_V`

Print the full `perl -V` report to STDOUT.

Synopsis

```
use Config;
Config::_V();
```

What you get back

Nothing. Side effects only: a multi-section dump covering the `myconfig` summary, compile-time options (sorted), the host OS, the compile-date line, every `PERL */PPERL *` entry in `%ENV`, and the contents of `@INC`.

Examples

```
## Equivalent of the command-line `ppperl -V`:

use Config;
Config::_V();
```

Edge cases

- Output goes to the OS STDOUT directly; Perl-level STDOUT redirection with `local *STDOUT` does not affect it.
- `@INC` entries are written verbatim, including any non-UTF-8 bytes.

Differences from upstream

- Lines describing the compiler and compile date reflect pperl’s Rust-based build.
- %ENV probe includes PPERL* in addition to PERL* so pperl-aware environments are visible in the dump.

5.2.4 Cwd

Report the current working directory and resolve paths to their absolute, symlink-free form. This module provides the four “where am I” functions (`cwd`, `getcwd`, `fastcwd`, `fastgetcwd`), the three “make this path absolute” functions (`abs_path`, `realpath`, `fast_abs_path`), and a `chdir` that keeps `$ENV{PWD}` in sync.

Two groups of tradeoffs matter when choosing between them.

Current directory. `getcwd` is the most portable and safe call - it asks the kernel via the `POSIX getcwd(3)` interface and never leaves the process anywhere surprising. `fastcwd` walks the tree with repeated `chdir(‘..’)` and is faster on deep paths, but if a parent directory disappears mid-walk (or is unreadable) it returns `undef` and may leave the process in a different directory than it started. `cwd` picks the most natural form for the platform; on Linux it is effectively `getcwd`. `fastgetcwd` is a synonym for `cwd`.

Path resolution. `abs_path` canonicalises a path, resolving symlinks and `./..` components. `realpath` is its POSIX-style alias - same function, different name. `fast_abs_path` is the faster variant that drops the “non-existent leaf” fallback: it returns `undef` the moment any component fails to resolve, where `abs_path` will still return a usable path when only the final leaf is missing.

`chdir` changes directory and, on success, writes the new absolute path to `$ENV{PWD}` so child processes inherit a consistent view.

Functions

Current directory

`cwd`

Return the absolute path of the process’s current working directory.

`chdir`

Change the process’s current working directory, and update `$ENV{PWD}` to match.

Path resolution

`abs_path`

Canonicalise a path: resolve symlinks, `..`, and `./..`, and return the absolute form.

fast_abs_path

Canonicalise a path strictly: resolve symlinks and `./` `..`, or return `undef` if any component is missing.

Other Functions

fsu_canonpath

`File::Spec->canonpath($path)` - logical cleanup of a path.

fsu_fn_canonpath

`File::Spec::Functions::canonpath($path)` - functional interface.

fsu_catdir

`File::Spec->catdir(@dirs)` - join directory names into a path.

fsu_fn_catdir

`File::Spec::Functions::catdir(@dirs)` - functional interface.

fsu_catfile

`File::Spec->catfile(@dirs, $file)` - join directories and a filename.

fsu_fn_catfile

`File::Spec::Functions::catfile(@dirs, $file)` - functional interface.

cwd

Return the absolute path of the process's current working directory.

Synopsis

```
use Cwd;
my $dir = getcwd();
my $dir = cwd();
my $dir = fastcwd();
my $dir = fastgetcwd();
```

What you get back

A plain Perl string: the absolute path, no trailing newline, no trailing slash (except when you really are at the root `/`). On failure every variant returns `undef` and sets `$!`.

Examples

Print where the script is running from:

```
use Cwd;
print getcwd(), "\n";      # e.g. /home/alice/project
```

Save and restore the working directory around a block that chdirs:

```
use Cwd;
my $saved = getcwd();
chdir '/tmp' or die $!;

## ... do work in /tmp ...

chdir $saved or die $!;
```

Compare two paths by their canonical form:

```
use Cwd;
chdir '/var/log' or die $!;
print getcwd() eq '/var/log' ? "yes\n" : "no\n";    # yes
```

Edge cases

- If the current directory has been removed by another process, every variant returns undef with \$! set to ENOENT.
- Symlinks in the path are already resolved by the kernel's `getcwd(3)`; the returned string is the real path.
- All four names are provided by default when you use `Cwd` with no import list.

Differences from upstream

- `cwd`, `getcwd`, `fastcwd`, and `fastgetcwd` all call the kernel's `getcwd(3)` directly. Upstream `fastcwd` walks the tree with repeated `chdir('..')` and can leave the process in a different directory on failure; pperl's version never `chdirs` and therefore cannot strand the process. The “Unstable directory path” die from upstream does not occur.

See also

- `abs_path` - canonicalise an arbitrary path, not just the CWD
- `chdir` - change directory and keep `$ENV{PWD}` synchronised
- `realpath` - POSIX-style alias for `abs_path`

abs_path

Canonicalise a path: resolve symlinks, ., and .., and return the absolute form.

Synopsis

```
use Cwd qw(abs_path realpath);
my $abs = abs_path($path);
my $abs = realpath($path);      # same function, POSIX alias
my $abs = abs_path();           # defaults to current directory
```

What you get back

A plain Perl string: the absolute, symlink-free path. On unresolvable input returns undef and sets \$!. If the leaf component does not exist but its parent does, the parent is canonicalised and the leaf is appended verbatim - useful for computing where a file *would* be written.

Examples

Resolve a relative path against the current directory:

```
use Cwd qw(abs_path);
chdir '/home/alice' or die $!;
print abs_path('project/Makefile'), "\n";
# /home/alice/project/Makefile
```

Follow a symlink to its target:

```
use Cwd qw(abs_path);
symlink '/etc/hostname', '/tmp/link';
print abs_path('/tmp/link'), "\n";      # /etc/hostname
```

Compute the absolute path a file would have before creating it:

```
use Cwd qw(abs_path);
print abs_path('output.new'), "\n";
# e.g. /home/alice/project/output.new - even if output.new
# does not yet exist, as long as the directory does
```

Use realpath when the surrounding code expects the POSIX name:

```
use Cwd qw(realpath);
print realpath('.'), "\n";              # absolute path of current dir
```

Edge cases

- Empty string or no argument - treated as ..

- Leaf missing, parent present - returns parent's canonical path joined with the leaf name. `fast_abs_path` returns `undef` in this situation; use it when “doesn't exist” should be an error.
- Both leaf and parent missing - returns `undef`.
- Symlink loop - returns `undef` and sets `$!` to `ELOOP`.

Differences from upstream

- The leaf-fallback form is always available; upstream provides it only via `_perl_abs_path` when the XS path is not loaded.
- `realpath` is wired to the same implementation as `abs_path`, matching upstream's `*realpath = \&abs_path` alias.

See also

- `fast_abs_path` - same canonicalisation, no leaf fallback
- `getcwd` - absolute path of the current directory only
- `chdir` - change directory before resolving relative paths

fast_abs_path

Canonicalise a path strictly: resolve symlinks and `./` `..`, or return `undef` if any component is missing.

Synopsis

```
use Cwd qw(fast_abs_path);
my $abs = fast_abs_path($path);
my $abs = fast_abs_path();      # defaults to current directory
```

What you get back

A plain Perl string with the absolute, symlink-free path, or `undef` if any part of the path cannot be resolved (including the leaf). Unlike `abs_path`, there is no parent-plus-leaf fallback - a missing final component is treated as an error.

Examples

Verify that a path exists and get its canonical form in one step:

```
use Cwd qw(fast_abs_path);
my $abs = fast_abs_path('/etc/passwd');
defined $abs or die "not found: $!";
print $abs, "\n";          # /etc/passwd
```

Reject non-existent files up front:

```
use Cwd qw(fast_abs_path);
print defined fast_abs_path('/no/such/file') ? "ok" : "missing";
# missing
```

Resolve the current directory quickly:

```
use Cwd qw(fast_abs_path);
print fast_abs_path(), "\n";      # equivalent to getcwd()
```

Edge cases

- Empty string or no argument - treated as ..
- Any missing component (including the leaf) - returns undef. Use `abs_path` when a missing leaf should still produce a path.
- Symlink loop - returns undef and sets `$!` to `ELOOP`.

Differences from upstream

- Upstream's `fast_abs_path` localises `$ENV{PWD}`, `chdir`s into the target, and reads the new CWD back. `pperl` calls `canonicalize(3)` directly - the result is the same, but the process working directory is never disturbed, so concurrent code that observes CWD sees no perturbation.

See also

- `abs_path` - same resolution, with a parent-plus-leaf fallback
- `realpath` - POSIX-style alias for `abs_path`
- `getcwd` - absolute path of the current directory only

chdir

Change the process's current working directory, and update `$ENV{PWD}` to match.

Synopsis

```
use Cwd qw(chdir);
chdir($path) or die $!;
```

What you get back

1 on success, 0 on failure. `$!` is set when the call fails so you can distinguish errors.

Examples

Change directory and keep `$ENV{PWD}` consistent for child processes:

```
use Cwd qw(chdir);
chdir('/tmp') or die "chdir: $!";
system('printenv', 'PWD');      # /tmp
```

Guard against a missing target:

```
use Cwd qw(chdir);
unless (chdir('/does/not/exist')) {
    warn "cannot enter directory: $!";
}
```

Switch directories around a block and return:

```
use Cwd qw(getcwd chdir);
my $saved = getcwd();
chdir('/var/log') or die $!;

## ... work in /var/log ...

chdir($saved) or die $!;
```

Edge cases

- Called with no argument - returns `0` (unlike the core `chdir` builtin, which `chdirs` to `$ENV{HOME}`). Pass the target explicitly.
- Target exists but is not a directory - returns `0` with `$!` set to `ENOTDIR`.
- `$ENV{PWD}` is updated only on success, through the `%ENV` tied hash, so `setenv(3)` is called as well.

Differences from upstream

- Upstream's `Cwd::chdir` normalises slashes in the argument before calling the builtin. `ppperl` passes the path through unchanged and relies on the kernel to handle duplicate separators, which it does.
- No-argument form returns `0`; upstream treats it as `chdir('')` which is equivalent to `chdir` to `HOME`. Scripts that depend on the implicit-`HOME` form should pass `$ENV{HOME}` explicitly.

See also

- `getcwd` - absolute path of the current directory
- `abs_path` - canonicalise a path without changing directory

5.2.5 Data

Namespace containing the following modules:

- `Data::Dumper` — Stringify Perl data structures into valid Perl code - feed the result back

`Data::Dumper`

Stringify Perl data structures into valid Perl code - feed the result back through eval and you rebuild the original structure.

Two calling styles cover almost everything:

- **Functional:** `Dumper(\%h, \@a)` - one call, one string, uses the package-global configuration variables (`$Data::Dumper::Indent` etc.).
- **Object-oriented:** `Data::Dumper->new([\%h, \@a], ['hash', 'array'])` returns a configurable dumper object. Chain accessors to tune output, then call `->Dump` to produce the string.

The main knobs:

- `$Indent` - 0 compact, 1 line-wrapped, 2 line-wrapped with alignment (the default and the most readable).
- `$Purity` - when true, round-trip fidelity for self-referential and cyclic structures at the cost of extra fixup statements.
- `$Sortkeys` - 1 for alphabetical, or a coderef returning the key order per hash.
- `$Deepcopy` - break shared references and emit each one as a copy.
- `$Terse` - omit the `$VAR1 =` prefix; emits a bare expression.
- `$Useqq` - quote strings with `"` and visible escapes (`\n`, `\t`, `\x{...}`) instead of `'...'`.
- `$Sparseseen` - skip populating the internal “seen” hash for values that the dumper can prove are never revisited. Trades memory for a small amount of safety on bizarre inputs.

Cycles and shared references are tracked through an internal *seen* table keyed by referent address; a second visit prints as a backref like `$VAR1->[0]` instead of recursing forever.

Most of the user-facing surface (`new`, `Dump`, `Reset`, `Seen`, `Values`, `Names`, all configuration accessors, `import`) lives in `Dumper.pm` and loads from disk; this file provides the two routines that must be written in Rust: the heavy-lifting serializer `Dumpxs` and the vstring magic probe `_vstring`.

Functions

Core API

`dumpxs`

Serialize one or more values into valid Perl code, fast.

Introspection

vstring

Return the raw bytes of a v-string, or undef if the argument is not a v-string.

dumpxs

Serialize one or more values into valid Perl code, fast.

Dumpxs is the XS counterpart of Dump: same inputs, same output, implemented without the pure-Perl recursion overhead. Dump itself delegates here unless `$Data::Dumper::Useperl` is true.

Synopsis

```
use Data::Dumper;
print Data::Dumper->new([\%h, \@a], ['hash', 'array'])->Dumpxs;
print Data::Dumper::Dumpxs('Data::Dumper', [\%h], ['hash']);
```

What you get back

In scalar context, a single string containing one `$VAR = ... ;` statement per input value, separated by `$self->{sep}` (default `"\n"`). In list context, one string per input - useful when you want to post-process each dump separately.

With `Terse(1)`, the `$VAR =` prefix and trailing `;` are dropped and you get a bare Perl expression suitable for embedding.

Examples

Named output, default formatting:

```
my %h = (a => 1, b => [2, 3]);
print Data::Dumper->new([\%h], ['h'])->Dumpxs;

## $h = {
##   'a' => 1,
##   'b' => [2, 3]
## };
```

Compact, sorted, double-quoted:

```
print Data::Dumper->new([\%h])
      ->Indent(0)->Sortkeys(1)->Useqq(1)->Dumpxs;

## $VAR1 = {"a" => 1,"b" => [2,3]};
```

Circular structure - the cycle is emitted as a backref, not an infinite expansion:

```
my @a = (1, 2);
push @a, \@a;
print Data::Dumper->new([ \@a ])->Dumpxs;

## $VAR1 = [1, 2, $VAR1];
```

List context gives you one string per argument:

```
my ($s1, $s2) = Data::Dumper->new([ \%h, \@a ])->Dumpxs;
```

Edge cases

- First argument is not a blessed hashref: Dumpxs will call new() internally using the remaining arguments, so Data::Dumper::Dumpxs('Data::Dumper', \@vals, \@names) works.
- maxrecurse exceeded: croaks with "Recursion limit of N exceeded" after finishing the current top-level value so partial output is not lost from the caller's perspective.
- maxdepth exceeded: prints [...] or {...} at the cut-off point rather than croaking.
- CODE references: emitted as sub { "DUMMY" } unless Deparse(1) is set.
- Regexp references: emitted as qr/.../.
- Sortkeys as a coderef: accepted but currently treated as a truthy value, i.e. alphabetical sort. The coderef is not invoked.

Differences from upstream

- Sortkeys as a coderef: upstream calls the coderef per hash to obtain a custom key order; pperl treats any truthy Sortkeys as alphabetical. Scripts relying on coderef ordering need to pre-sort their data or call Dump with Useperl(1).
- Deparse(1) emits sub { "DUMMY" } instead of the deparsed source. Scripts that depend on round-tripping closures need Useperl(1).
- Purity(1) is accepted and stored but does not yet emit the post-dump fixup statements upstream produces for self-referential structures; the backref notation works but arbitrary in-place assignments to reach an already-seen inner value are not emitted.

See also

- Dump - the pure-Perl fallback; identical output, slower.
- Dumper - the procedural one-liner wrapping new(...) ->Dumpxs.
- Seen - pre-seed the seen-table to name pre-existing references.
- Reset - clear the seen-table between unrelated Dumpxs calls.

5.2.6 Devel

Namespace containing the following modules:

- `Devel::Peek` — Inspect the internal representation of any Perl scalar - a debugging tool for working with SVs.

Devel::Peek

Inspect the internal representation of any Perl scalar - a debugging tool for working with SVs.

`Devel::Peek` is the tool you reach for when the question is “*what does Perl actually think this value is?*”. Typical uses:

- chasing a refcount leak - something holds a reference longer than you expect, and you want to see REFCNT climb;
- confirming that a value came back as the type you meant (IV vs PV vs PVMG, a blessed object, a tied scalar);
- looking at the flags on an SV (POK, IOK, ROK, UTF8, TEMP, OBJECT) to diagnose dual-var, stringification, or tainting surprises;
- peeking inside arrays, hashes, and references to see how Perl has laid them out and which slots contain what.

Synopsis

```
use Devel::Peek;

my $x = 42;
Dump($x);
```

A minimal `Dump($x)` for a plain integer looks like this on STDERR:

```
SV = IV(0xbc818) at 0xbe9a8
  REFCNT = 1
  FLAGS = (IOK,pIOK)
  IV = 42
```

Every line is meaningful. `SV = IV(...)` names the body type. REFCNT is what you stare at when tracking leaks. FLAGS tells you which of the IV / NV / PV slots below it are live; IOK means the IV field is current, POK means the PV field is, ROK means the scalar is a reference and there is another `SV = ...` block nested below it. Arrays dump with FILL (last index) and MAX (capacity) and then their elements as `Elt No. N` sub-dumps. Hashes dump with KEYS, bucket occupancy, and one sub-dump per entry.

The entry points:

- `Dump($sv, $depth=4)` - dump one value. `@array` and `%hash` dump the aggregate itself rather than the first element; other arguments are evaluated in scalar rvalue context.
- `DumpArray($depth, @values)` - dump several values in one call, each tagged with its position. Useful when inspecting a function’s return list.

- `SvREFCNT(\ $sv)` - return the refcount of the referent of a reference (the outer reference itself is subtracted). Pairs well with `Dump` when you want the number without scraping it out of a dump.
- `DumpWithOP($sv, $depth=4)` - same as `Dump` but also walks into compiled ops when the scalar carries one (a PVCV, for example). Equivalent to setting `$Devel::Peek::dump_ops` around a `Dump` call.
- `DumpProg()` - dump the program's main op tree. Aimed at people debugging the compiler, not at regular scripts.
- `DeadCode()` - walk the SV arena for scalars frozen into inactive CVs. A deep-internals tool; most users never touch it.
- `CvGV($cv)` - return the GV associated with a code reference, so you can recover a sub's package-qualified name.

The `mstat` family

`mstat`, `fill_mstats`, `mstats_fillhash`, and `mstats2hash` report on Perl's malloc arena - bucket counts, free/used chunks, sbrk activity. They only produce real numbers when Perl was built with its bundled malloc (rare in modern builds). Under a standard build, `mstat` prints `perl not compiled with MYMALLOC` and the hash-filling variants croak with the same message. This is the upstream behaviour, not a pperl limitation; scripts that guard on it already handle the non-MYMALLOC case.

The `runops` and `:opd knobs`

use `Devel::Peek ' :opd=st'` switches the interpreter to its debugging dispatch loop and turns on per-op trace flags (s stack, t trace, P profile). `runops_debug()` is the runtime toggle for the same switch; `debug_flags()` reads and writes the `^D` letter mask directly. Reach for these when you want to see which ops execute and in what order - they are the programmable equivalent of `perl -Dst`.

All output from `Dump`, `DumpArray`, `DumpProg`, and `mstat` goes to `STDERR`. The global `$Devel::Peek::pv_limit` caps how many characters of each PV are printed; `0` means no limit.

Functions

Scalar inspection

`Dump`

Dump one scalar's internal representation to `STDERR`. The usual entry point for `Devel::Peek`; reaches for this first when you need to see what Perl has under the hood.

`DumpArray`

Dump several values in one call, each tagged with its index. Useful when inspecting a function's return list or an argument list without wrapping every element in its own `Dump`.

SvREFCNT

Return the reference count of a referent. `SvREFCNT(\$x)` answers “*how many live references to \$x exist right now?*” - the usual tool for tracking a suspected leak without pulling the number out of a full Dump.

DeadCode

Report scalars “frozen” into inactive code references - a deep-internals probe for leaks inside closed-over CVs.

DumpWithOP

Dump a scalar and, if it carries a compiled op tree, the tree itself. Convenient when inspecting a CV (code reference) or any scalar whose interest is partly in its attached ops.

DumpProg

Dump the program’s main op tree. A compiler-debugging tool, not a value-inspection tool.

CvGV

Recover the GV (typeglob) associated with a code reference. `CvGV($cv)` is how you go from an anonymous-looking sub ref back to the name Perl filed it under - useful when inspecting caller frames, dispatch tables, or closures assigned to named slots.

Memory stats

fill_mstats

Fill a scalar with a machine-readable snapshot of Perl’s malloc state. `fill_mstats($buf)` is the cheap member of the mstat family: pair it with `mstats2hash` later when you actually want to read the numbers.

mstats_fillhash

Populate a hash with Perl’s malloc statistics. `mstats_fillhash(%hash)` fills %hash with fields like `nbuckets`, `total`, `totfree`, `sbrks`, plus the per-bucket `free / used / mem_size / available_size` array references. See `perldebguts` for the full field list.

mstats2hash

Decode a `fill_mstats` buffer into a hash. `mstats2hash($buf, %hash)` is the readout step for the buffered form: cheap `fill_mstats` inside a hot loop, one `mstats2hash` pass at the end to turn the buffer into inspectable fields.

mstat

Print a one-line memory snapshot to STDERR. The classic marker-in-a-log tool - sprinkle `mstat "Point 5"` across a program to see how arena size moves between points.

Dump

Dump one scalar's internal representation to STDERR. The usual entry point for `Devel::Peek` reaches for this first when you need to see what Perl has under the hood.

`Dump($sv, $depth=4)`. `$depth` caps both recursion into nested references and the number of array/hash elements shown at each level; pass a larger value to see past the default cutoff. When called on `@array` or `%hash` directly, the aggregate itself is dumped (not its first element). Any other argument is evaluated in scalar `rvalue` context.

The dump layout follows the shape shown in the module summary: an `SV = <type>(body)` at `<head>` line, then `REFCNT`, `FLAGS`, and whichever of the `IV / NV / PV / RV / array / hash` slots are populated. Reference dumps nest a child `SV = ...` block inside the parent.

Two package variables tune the output:

- `$Devel::Peek::pv_limit` - cap on characters printed per PV; `0` lifts the cap entirely.
- `$Devel::Peek::dump_ops` - when true, CV-like scalars also print their op tree (see `DumpWithOP` for the scoped form).

Returns nothing; all output is on STDERR.

DumpArray

Dump several values in one call, each tagged with its index. Useful when inspecting a function's return list or an argument list without wrapping every element in its own `Dump`.

`DumpArray($depth, @values)`. The first argument is the recursion depth (same meaning as `Dump`'s second argument); the remaining arguments are the values to dump. Each value is preceded by a header line of the form `Elt No. N 0x<address>`, where `N` counts from zero. After the header the element is rendered exactly as `Dump` would render it.

Respects `$Devel::Peek::pv_limit` and `$Devel::Peek::dump_ops` the same way `Dump` does. Returns nothing; output goes to STDERR.

SvREFCNT

Return the reference count of a referent. `SvREFCNT(\ $x)` answers “*how many live references to \$x exist right now?*” - the usual tool for tracking a suspected leak without pulling the number out of a full `Dump`.

The argument must be a reference (prototype `\[$@%&*]`). The referent's refcount is returned, minus one so that the temporary reference created by the `\` operator does not skew the answer. Passing anything that is not a reference croaks with `Usage: Devel::Peek::SvREFCNT(SCALAR)`.

Typical pattern: snapshot the count before and after a suspected leak site and look for growth that does not drop back.

DeadCode

Report scalars “frozen” into inactive code references - a deep-internals probe for leaks inside closed-over CVs.

`DeadCode()` walks Perl’s SV arena looking for PVCV bodies whose pads still hold live scalars even though the sub is no longer reachable, and returns the findings as an array reference. Most scripts never need this; it is a tool for people chasing exotic leaks that `Dump` and `SvREFCNT` cannot locate.

Returns an array reference. An empty result means “nothing matched the arena-walk heuristic” - which is the expected outcome under `ppperl`, whose allocator layout differs from the arena the upstream walk was written against.

DumpProg

Dump the program’s main op tree. A compiler-debugging tool, not a value-inspection tool.

`DumpProg()` writes the current value of `PL_dumpindent` as a warning and then, if `PL_main_root` is populated, walks the tree rooted there. Aimed at people working on Perl’s own compiler pipeline; regular scripts have no reason to call it.

Under `ppperl` the op-tree walk reports only the indent header, since `PL_main_root` is not wired through the XS surface yet.

mstat

Print a one-line memory snapshot to `STDERR`. The classic marker-in-a-log tool - sprinkle `mstat "Point 5"` across a program to see how arena size moves between points.

`mstat($marker="Devel::Peek::mstat: ")`. The marker prefixes the output so you can match lines to call sites when the log gets noisy. Output format is documented under `perldebguts/Using $ENV{PERL_DEBUG_MSTATS}`.

Only produces real numbers when Perl was built with its bundled `malloc`. On a standard build the line reads `<marker>: perl not compiled with MYMALLOC`, which is useful as a pass-through breadcrumb even when the numbers are unavailable.

5.2.7 Digest

Namespace containing the following modules:

- `Digest::MD5` — Compute MD5-128 message digests - fixed 128-bit fingerprints of
- `Digest::SHA` — Compute SHA-1 and SHA-2 family message digests: SHA-1, SHA-224, SHA-256,

Digest::MD5

Compute MD5-128 message digests - fixed 128-bit fingerprints of arbitrary byte strings per RFC 1321.

`Digest::MD5` offers two idioms. Pick whichever fits the shape of the data you have at hand.

Functional one-shot

When the whole message is already in memory as a single string (or a few pieces you can list as arguments), the functional form is the shortest path:

```
use Digest::MD5 qw(md5 md5_hex md5_base64);

my $bin = md5($data);           # 16 raw bytes
my $hex = md5_hex($data);       # 32 lowercase hex chars
my $b64 = md5_base64($data);    # 22 base64 chars, no padding
```

All three take any number of arguments and concatenate them before hashing, so `md5("a", "b", "c")` equals `md5("abc")`.

OO incremental

When the message arrives in pieces - streaming bytes, a file read chunk by chunk, or data assembled across multiple code paths - build a context object, feed it, and ask for the digest at the end:

```
use Digest::MD5;

my $ctx = Digest::MD5->new;
$ctx->add($chunk1);
$ctx->add($chunk2, $chunk3);
$ctx->addfile($fh);
my $hex = $ctx->hexdigest;      # also resets the context
```

`add` returns the object, so calls chain: `$ctx->add("a")->add("b")`. Calling any of the `digest` / `hexdigest` / `b64digest` methods is destructive - the context is reset to an empty state afterwards, so the same object is immediately reusable. To peek at the digest without resetting, clone first: `$ctx->clone->hexdigest`.

Output encodings

Every function and method comes in three encodings of the same 128-bit digest:

- **Raw binary** (`md5`, `digest`) - 16 bytes. Suitable for packing into fixed-width fields or feeding into another binary protocol.
- **Hex** (`md5_hex`, `hexdigest`) - 32 lowercase characters from 0-9a-f. This is the form you usually see in ETag headers and checksum files.
- **Base64** (`md5_base64`, `b64digest`) - 22 characters from the standard alphabet A-Z a-z 0-9 + /. The result is *not* padded to a multiple of 4; append "==" yourself if an interop partner expects padding.

File-handle ingestion

`addfile` reads an open file handle until EOF and feeds every byte into the context without buffering the whole thing in memory, so it scales to files of arbitrary size. Make sure the handle is in binmode first - MD5 is defined over bytes, not decoded text.

Security context

MD5 is **cryptographically broken for collision resistance**: attackers can construct two different messages with the same digest. Do not use it for new digital signatures, certificate fingerprints, password hashing, or any setting where an adversary chooses the input. For those, reach for `Digest::SHA` (SHA-256 or better) or a purpose-built password hash.

MD5 remains perfectly adequate - and widely used - for non-adversarial integrity checks: ETags, cache keys, change detection in build systems, deduplication of trusted data, and checksums over storage where the threat model is accidental corruption rather than forgery.

Bytes only - no wide characters

MD5 hashes bytes. Feeding a string containing code points above U+00FF croaks with `Wide character in subroutine entry`. To hash a Unicode string, decide on an encoding first and pass the encoded bytes: `md5_hex(encode_utf8($str))`.

Functions

Functional digests

`md5_bin`

One-shot MD5 digest, returned as 16 raw bytes.

`md5_hex`

One-shot MD5 digest, returned as a 32-character lowercase hex string.

`md5_b64`

One-shot MD5 digest, returned as a 22-character base64 string.

OO lifecycle

`new`

Create a fresh MD5 context object, or reset an existing one to empty.

clone

Make an independent copy of the current digest state.

destroy

Release the native MD5 state buffer when a context object is reaped.

reset

Clear an existing context so it behaves like a freshly created one.

OO input

add_bits

Add a bit-oriented quantum of data - but only in multiples of 8 bits.

add

Feed one or more pieces of data into the MD5 context.

addfile

Read an open file handle to EOF and feed every byte into the context.

OO output

digest

Finalise the context and return the 16-byte raw binary digest.

hexdigest

Finalise the context and return the digest as 32 lowercase hex chars.

b64digest

Finalise the context and return the digest as a 22-char base64 string.

Utility

context

Save or restore the raw internal MD5 state - for advanced use only.

new

Create a fresh MD5 context object, or reset an existing one to empty.

Synopsis

```
my $ctx = Digest::MD5->new;    # class method - new empty context
$ctx->new;                      # instance method - reset in place
```

What you get back

A blessed `Digest::MD5` reference whose internal state is the MD5 initial vector - i.e. the digest of the empty string is `md5("")`. Called on an existing instance, `new` resets that instance and returns it; no second object is allocated. The common idiom `Digest::MD5->new->add($data)->hexdigest` relies on this chaining.

Examples

Fresh context, then feed and finalise:

```
my $ctx = Digest::MD5->new;
$ctx->add("hello");
print $ctx->hexdigest;    # 5d41402abc4b2a76b9719d911017c592
```

Reset an existing context instead of allocating a new one:

```
$ctx->add("round 1")->hexdigest;    # digest + auto-reset
$ctx->new;                          # explicit reset (idiomatic no-
  ↳op here)
$ctx->add("round 2");
```

Chain straight through in one expression:

```
my $hex = Digest::MD5->new->add("foo", "bar")->hexdigest;
```

Edge cases

- Called on a subclass name, `new` blesses into that subclass, so `MyDigest->new` returns a `MyDigest` object whose internals are a `Digest::MD5` context.
- Called as an instance method, the same object is returned - the caller's variable is unchanged.

Differences from upstream

Fully compatible with upstream `Digest::MD5` 2.58.

See also

- `clone` - copy the current context instead of resetting it
- `reset` - explicit alias for the instance-method form of `new`
- `add` - feed data into the context you just created
- `hexdigest` - common companion call for one-shot chained use

clone

Make an independent copy of the current digest state.

Synopsis

```
my $copy = $ctx->clone;
```

What you get back

A new `Digest::MD5` object whose internal state is a byte-for-byte copy of the original - same accumulated length, same four state words, same partial-block buffer. The two contexts are independent afterwards: further add calls on one do not affect the other. The class of the returned object matches the class of `$ctx`, so cloning a subclass instance yields a subclass instance.

The primary use is peeking at an intermediate digest without destroying the ongoing computation. Because `digest` / `hexdigest` / `b64digest` reset the context, the pattern `$ctx->clone->hexdigest` is how you extract a snapshot and keep hashing.

Examples

Per-line running digest over a stream:

```
my $ctx = Digest::MD5->new;
while (my $line = <$fh>) {
    $ctx->add($line);
    print "after line $.: ", $ctx->clone->hexdigest, "\n";
}
my $final = $ctx->hexdigest;    # digest of the whole stream
```

Fork a digest to feed two suffixes:

```
my $base = Digest::MD5->new;
$base->add("common prefix ");

my $a = $base->clone->add("suffix A")->hexdigest;
my $b = $base->clone->add("suffix B")->hexdigest;
```

Edge cases

- Cloning immediately after `new` yields a context equivalent to a fresh `Digest::MD5->new`.
- Cloning an object whose state was restored via context copies that restored state, not a fresh one.

Differences from upstream

Fully compatible with upstream `Digest::MD5 2.58`.

See also

- `new` - create a fresh empty context from scratch
- `digest` - the destructive read that makes `clone` worth having
- `context` - save/restore raw state if you need persistence
- `add` - continue feeding data into the clone or the original

destroy

Release the native MD5 state buffer when a context object is reaped.

Synopsis

```
## Called automatically - you do not invoke DESTROY yourself.  
undef $ctx;           # triggers DESTROY on the last reference
```

What you get back

Nothing. `DESTROY` is Perl's object finaliser hook; it runs when the last reference to a `Digest::MD5` object goes away, frees the `MD5_CTX` buffer that was allocated alongside the object, and returns no value.

You almost never call this directly. It is documented here so readers doing memory audits or writing subclasses know that the module owns a raw buffer behind the blessed reference and that the buffer is reclaimed on normal refcount-driven destruction.

Examples

Destruction happens when the last reference drops, either at scope exit or via explicit `undef`:

```
{  
    my $ctx = Digest::MD5->new;  
    $ctx->add("data");  
} # DESTROY runs here
```

Explicit drop mid-scope:

```
my $ctx = Digest::MD5->new;
$ctx->add("data");
undef $ctx;      # DESTROY runs now, context is freed
```

Edge cases

- Tolerant of malformed input: if the object does not carry the expected magic, DESTROY returns silently rather than croaking. This matches upstream and is important because DESTROY can be called during global destruction in arbitrary order.
- Running on an already-destroyed context (possible during global destruction cycles) is a no-op.

Differences from upstream

Fully compatible with upstream Digest::MD5 2.58.

See also

- new - the other end of the lifecycle
- clone - each clone carries its own buffer and its own DESTROY
- reset - avoid destruction entirely by reusing the context

reset

Clear an existing context so it behaves like a freshly created one.

Synopsis

```
$ctx->reset;          # discard accumulated data
$ctx->reset->add($data); # reset and keep going in one line
```

What you get back

The same object, returned so calls chain. After reset, the internal state is identical to Digest::MD5->new: accumulated length is zero, the four state words hold the MD5 initial vector, and the partial-block buffer is empty. No new allocation happens

- the existing MD5_CTX is overwritten in place.

reset is an alias for the instance-method form of new, exposed as a separate method for readability. Use whichever reads better at the call site.

Examples

Reuse a context across independent digests:

```
my $ctx = Digest::MD5->new;
my $h1 = $ctx->add("first")->hexdigest;    # digest + implicit
    ↪ reset
my $h2 = $ctx->add("second")->hexdigest;    # starts fresh
```

Explicit reset after a partial computation you want to throw away:

```
$ctx->add("oops, wrong data");
$ctx->reset;
$ctx->add($correct_data);
print $ctx->hexdigest;
```

Edge cases

- Calling `reset` on a fresh context is a harmless no-op.
- Reset does not change the class of the object, so subclasses remain blessed into the subclass after a reset.

Differences from upstream

Fully compatible with upstream `Digest::MD5 2.58`.

See also

- `new` - same effect when called as an instance method
- `digest` - implicitly resets after returning the digest
- `add` - the usual next step after `reset`

add_bits

Add a bit-oriented quantum of data - but only in multiples of 8 bits.

Synopsis

```
$ctx->add_bits($bitstring);    # "01101000..." form
$ctx->add_bits($data, $nbits);  # byte-string + bit count
```

What you get back

The same context object, for chaining. MD5 is a byte-oriented algorithm, so `add_bits` exists purely for interface compatibility with other `Digest::*` modules that *do* hash at bit granularity. It accepts exactly the same two calling forms they do, but requires the bit count to be a whole multiple of 8 and croaks otherwise.

Unless you are writing code that must work across multiple digest backends, use `add` directly - it is clearer and has no restrictions.

Examples

Bit-string form - a string of '0' and '1' characters:

```
$ctx->add_bits("0110100001101001");    # feeds bytes "hi"
```

Byte-string + bit count - takes the first `$nbits` / 8 bytes:

```
$ctx->add_bits("hello world", 40);        # feeds "hello"
```

Chain alongside `add`:

```
my $hex = Digest::MD5->new
    ->add_bits("0110100001101001")    # "hi"
    ->add(" there")
    ->hexdigest;
```

Edge cases

- Croaks with "Number of bits must be multiple of 8" when the bit string's length is not a multiple of 8.
- Croaks with "Number of bits must be multiple of 8 in `add_bits`" when the explicit `$nbits` argument is not a multiple of 8.
- In the two-argument form, `$nbits` larger than `8 * length($data)` silently feeds nothing - the call is a no-op.

Differences from upstream

Fully compatible with upstream `Digest::MD5` 2.58.

See also

- `add` - the direct byte-level input method
- `addfile` - stream bytes from a file handle
- `new` - typical predecessor in a chain
- `hexdigest` - typical successor in a chain

add

Feed one or more pieces of data into the MD5 context.

Synopsis

```
$ctx->add($data);
$ctx->add($a, $b, $c);           # same as add($a.$b.$c)
$ctx->add("a")->add("b")->add("c"); # chained equivalent
```

What you get back

The same context object, so add calls chain and compose with hexdigest / digest / b64digest. Every argument is flattened into a byte stream in order; the digest depends only on the concatenation, not on how you split it across calls or arguments.

Examples

One-shot digest of a concatenated message:

```
my $hex = Digest::MD5->new->add("hello world")->hexdigest;

## 5eb63bbbe01eeed093cb22bb8f5acdc3
```

Stream a file line by line:

```
open(my $fh, '<:raw', $path) or die $!;
my $ctx = Digest::MD5->new;
while (my $line = <$fh>) {
    $ctx->add($line);
}
print $ctx->hexdigest, "\n";
```

Multiple arguments are equivalent to concatenation - all four of these produce the same digest:

```
Digest::MD5->new->add("a")->add("b")->add("c")->hexdigest;
Digest::MD5->new->add("a", "b", "c")->hexdigest;
Digest::MD5->new->add("abc")->hexdigest;
md5_hex("abc");
```

Edge cases

- **Wide characters.** Arguments containing code points above U+00FF croak with Wide character in subroutine entry. Encode first: `$ctx->add(encode_utf8($str))`.
- **Empty strings** are accepted and change nothing.
- **No arguments** (`$ctx->add` with no extra args) is a no-op and returns the context.
- **undef arguments** stringify to the empty string; warnings depend on the caller's use warnings state.

Differences from upstream

Fully compatible with upstream Digest::MD5 2.58.

See also

- `add_bits` - interface-compatibility wrapper for bit-oriented input
- `addfile` - feed an entire file handle without buffering
- `new` - typical predecessor
- `hexdigest` - typical successor; auto-resets the context
- `clone` - snapshot the intermediate state without stopping add

addfile

Read an open file handle to EOF and feed every byte into the context.

Synopsis

```
open(my $fh, '<:raw', $path) or die $!;  
$ctx->addfile($fh);
```

What you get back

The same context object, for chaining. `addfile` reads the handle in 4 KiB blocks and feeds them into MD5 without loading the whole file into memory, so it handles files of any size. Once the call returns, the handle is positioned at EOF; the caller is responsible for closing it.

Put the handle in binmode (or open with `:raw`) before calling - MD5 hashes bytes, and a text-mode handle on Windows or one with a decoding layer will change what bytes reach the digest.

Examples

Digest a file in one chain:

```
open(my $fh, '<:raw', "/etc/passwd") or die $!;  
print Digest::MD5->new->addfile($fh)->hexdigest, "\n";  
close $fh;
```

Combine file contents with extra data:

```
my $ctx = Digest::MD5->new;  
$ctx->add($salt);  
$ctx->addfile($fh);  
$ctx->add($pepper);  
my $hex = $ctx->hexdigest;
```

Skip the object entirely for the one-file common case:

```
## Same result, shorter:

my $hex = Digest::MD5->new->addfile($fh)->hexdigest;
```

Edge cases

- Croaks with "No filehandle passed" if the argument is not a file handle (including when it is undef or an unopened glob).
- Croaks with "Reading from filehandle failed" if read returns an I/O error; the context's state after such a failure is unspecified - discard or reset it.
- Reading an empty handle (EOF immediately) is a no-op and returns the context unchanged.
- Handles decoded as UTF-8 still work, but you are hashing the raw *decoded* bytes the PerlIO layer returns, which is rarely what you want. Prefer binmode.

Differences from upstream

Fully compatible with upstream Digest::MD5 2.58.

See also

- add - feed in-memory data instead of a file
- add_bits - bit-oriented input for cross-digest compatibility
- new - typical predecessor when digesting one file
- hexdigest - typical successor for a single-file chain

digest

Finalise the context and return the 16-byte raw binary digest.

Synopsis

```
my $bin = $ctx->digest;           # exactly 16 bytes
```

What you get back

A 16-byte Perl string holding the raw digest. This is the most compact representation - handy for binary protocols or packing into fixed-width records - but it typically contains non-printable bytes, so do not paste it into log lines without encoding.

digest is destructive. After it returns, the context is automatically reset to the empty state and is immediately ready to digest a new message. To read the digest without destroying the state, use `$ctx->clone->digest` instead.

Examples

Raw bytes, then convert to hex by hand:

```
my $bin = Digest::MD5->new->add("abc")->digest;
printf "%v02x\n", $bin;    # 90.01.50.98.3c.d2.4f.b0.d6.96.3f.7d.28.
                             ↪e1.7f.72
```

Pack into a fixed-width binary record:

```
my $record = pack("A16 a16", $filename, $ctx->digest);
```

Peek at an intermediate digest without resetting:

```
my $snapshot = $ctx->clone->digest;    # $ctx keeps going
$ctx->add($more);
```

Edge cases

- Called on a context with no data added, returns the MD5 of the empty string (d41d8cd98f00b204e9800998ecf8427e as hex).
- Resets *before* returning, so calling `digest` twice in a row without adding data between the calls gives the empty-string digest the second time.

Differences from upstream

Fully compatible with upstream `Digest::MD5 2.58`.

See also

- `hexdigest` - same digest, formatted as 32 hex characters
- `b64digest` - same digest, base64-encoded
- `md5` - one-shot functional form
- `clone` - take a snapshot digest without resetting the context

hexdigest

Finalise the context and return the digest as 32 lowercase hex chars.

Synopsis

```
my $hex = $ctx->hexdigest;    # 32 chars, [0-9a-f]
```

What you get back

A 32-character ASCII string containing only 0-9 and a-f. This is the form you want for ETags, md5sum-style checksum files, log output, and anywhere humans will read the digest. No 0x prefix, no separators, lowercase.

hexdigest is **destructive**: the context is reset to empty after the digest is produced. Clone first if you need to keep going.

Examples

Common one-liner:

```
my $hex = Digest::MD5->new->add("hello")->hexdigest;
## 5d41402abc4b2a76b9719d911017c592
```

Digest the same context twice by cloning:

```
$ctx->add("stream so far");
my $checkpoint = $ctx->clone->hexdigest;
$ctx->add("more data");
my $final      = $ctx->hexdigest;
```

Build an md5sum-compatible line:

```
open(my $fh, '<:raw', $path) or die $!;
printf "%s  %s\n", Digest::MD5->new->addfile($fh)->hexdigest, $path;
```

Edge cases

- Empty context → d41d8cd98f00b204e9800998ecf8427e (the MD5 of the empty string).
- The result is always lowercase; if an external format demands uppercase, call uc on it.
- Resets the context, so \$ctx->hexdigest twice without intervening add calls gives the empty-string digest the second time.

Differences from upstream

Fully compatible with upstream Digest::MD5 2.58.

See also

- digest - same content as 16 raw bytes
- b64digest - shorter base64 encoding of the same digest
- md5_hex - one-shot functional form
- clone - preserve the context before finalising

b64digest

Finalise the context and return the digest as a 22-char base64 string.

Synopsis

```
my $b64 = $ctx->b64digest;    # 22 chars, no padding
```

What you get back

A 22-character string using the standard base64 alphabet A-Z a-z 0-9 + /. The result is **not** padded with =; the full base64 encoding of 16 bytes would be 24 characters, but the tail is fixed so the padding is redundant. Append "==" yourself if an interop partner strictly requires multiples of 4.

Base64 gives you a compact, one-line-friendly identifier that is roughly 30% shorter than hex while still printable. Common in cache keys, URL query parameters, and condensed log formats.

b64digest is **destructive**: the context is reset after use.

Examples

Compact digest for a cache key:

```
my $key = Digest::MD5->new->add($url)->b64digest;
```

Pad to a multiple of 4 for strict base64 consumers:

```
my $padded = Digest::MD5->new->add($data)->b64digest . "==";
```

Peek without consuming:

```
my $snapshot = $ctx->clone->b64digest;
```

Edge cases

- Empty context → 1B2M2Y8AsgTpgAmY7PhCfg (the MD5 of the empty string in base64).
- Contains / and +, so the result is **not** safe to drop into a URL path without further encoding - use URL-safe base64 for that (substitute - for + and _ for /).
- Resets the context after returning.

Differences from upstream

Fully compatible with upstream Digest::MD5 2.58.

See also

- digest - raw bytes
- hexdigest - hex-encoded digest (longer but URL-safe)
- md5_base64 - one-shot functional form
- clone - preserve the context before finalising

context

Save or restore the raw internal MD5 state - for advanced use only.

Synopsis

```
my @state = $ctx->context;           # get
$ctx->context($blocks, $state_bytes, $partial); # restore
```

What you get back

Called with no extra arguments, returns a list that fully captures the current digest state:

1. `$blocks` - number of complete 64-byte blocks processed so far, as an unsigned integer.
2. `$state_bytes` - 16 raw bytes holding the four MD5 state words in little-endian order.
3. `$partial` - present only when some bytes have been added that do not yet fill a full block. Up to 63 bytes of unprocessed data.

Called with those same three values, restores the context to the captured state and returns the object itself. This is the only way to persist a digest across process boundaries - serialise the three values, hand them off, then restore on the other side and continue adding.

For almost every use case, `clone` is a cleaner tool. Reach for `context` only when the state needs to survive something `clone` cannot cross, such as a database round-trip or an out-of-band storage step.

Examples

Round-trip a partial digest through storage:

```
my $ctx = Digest::MD5->new;
$ctx->add($first_half);
my @saved = $ctx->context;           # freeze state

## ... later, possibly in another process ...

my $ctx = Digest::MD5->new;
$ctx->context(@saved);               # thaw state
$ctx->add($second_half);
print $ctx->hexdigest;              # same as hashing whole thing
```

Serialise for storage:

```
use Storable qw(freeze thaw);
my $blob = freeze([$ctx->context]); # three values in, one blob
    ↳out
```

(continues on next page)

(continued from previous page)

```
## ...  
  
my $ctx = Digest::MD5->new;  
$ctx->context(@{ thaw($blob) });
```

Edge cases

- The third (\$partial) value is only returned when `length($partial) > 0`. Be prepared for a two-element list when the input fits exactly into whole blocks.
- Passing a \$state_bytes argument shorter than 16 bytes silently leaves the state words untouched. Do not rely on this - pass exactly what context gave you.
- context is the only method whose interface is genuinely raw. Its output is sensitive to the algorithm, not to a documented wire format; do not pretend values produced by one digest class are interchangeable with another.

Differences from upstream

Fully compatible with upstream Digest::MD5 2.58.

See also

- clone - the usual way to snapshot state in-process
- digest - finalise rather than persist
- new - fresh context for use with context as a restore target
- add - continue hashing after a restore

md5_bin

One-shot MD5 digest, returned as 16 raw bytes.

Synopsis

```
use Digest::MD5 qw(md5);  
my $bin = md5($data);           # 16 bytes  
my $bin = md5($a, $b, $c);      # md5($a.$b.$c)
```

What you get back

A 16-byte Perl string holding the raw MD5 digest. All arguments are concatenated (in order) and the result is hashed, so `md5("a", "b", "c")` is the same as `md5("abc")`. The function does not allocate an object; it runs through a stack-local context, which makes it the cheapest way to hash something you already hold in memory.

Use `md5_hex` or `md5_base64` when you need a printable form.

Examples

Digest a string to raw bytes:

```
use Digest::MD5 qw(md5);
my $bin = md5("hello world");      # 16 bytes
printf "%v02x\n", $bin;

## 5e.b6.3b.bb.e0.1e.ee.d0.93.cb.22.bb.8f.5a.cd.c3
```

Concatenation is implicit:

```
my $a = md5("foo", "bar");
my $b = md5("foobar");

## $a eq $b
```

Pack into a binary record:

```
my $blob = pack("N a16", length($payload), md5($payload));
```

Edge cases

- `md5()` with no arguments returns the MD5 of the empty string (d41d8cd98f00b204e9800998ecf8427e in hex).
- Wide-character arguments (code points above U+00FF) croak with Wide character in subroutine entry. Encode first: `md5(encode_utf8($str))`.
- Called as a class or instance method (e.g. `Digest::MD5->md5(...)`), it warns about probable misuse but still returns a digest based on all arguments, including the invocant. Prefer the plain function call.

Differences from upstream

Fully compatible with upstream `Digest::MD5` 2.58.

See also

- `md5_hex` - same digest, as 32 lowercase hex characters
- `md5_base64` - same digest, base64-encoded
- `digest` - OO form; reuse one context across many inputs
- `addfile` - for data too large to hold in memory at once

md5_hex

One-shot MD5 digest, returned as a 32-character lowercase hex string.

Synopsis

```
use Digest::MD5 qw(md5_hex);
my $hex = md5_hex($data);           # 32 chars
my $hex = md5_hex($a, $b, $c);      # md5_hex($a.$b.$c)
```

What you get back

A 32-character ASCII string containing only 0-9 and a-f. This is the encoding used by md5sum, HTTP ETag headers, and virtually every human-readable checksum format. All arguments are concatenated before hashing.

Examples

The canonical one-liner:

```
use Digest::MD5 qw(md5_hex);
print md5_hex("foobarbaz"), "\n";

## 6df23dc03f9b54cc38a0fc1483df6e21
```

Compose a cache key from structured data:

```
my $key = md5_hex($user_id, "|", $query, "|", $locale);
```

Digest bytes from a multi-byte string via UTF-8 encoding:

```
use Encode qw(encode_utf8);
my $hex = md5_hex(encode_utf8("abc\x{300}"));

## 8c2d46911f3f5a326455f0ed7a8ed3b3
```

Edge cases

- `md5_hex()` with no arguments returns `d41d8cd98f00b204e9800998ecf8427e`.
- Wide-character arguments croak; see `md5` for the fix.
- Result is always lowercase. Call `uc` if a tool requires uppercase hex.

Differences from upstream

Fully compatible with upstream `Digest::MD5 2.58`.

See also

- `md5` - same digest as raw bytes
- `md5_base64` - same digest, shorter base64 form
- `hexdigest` - OO form over a reusable context
- `add` - feed data incrementally when you cannot fit it in memory

md5_b64

One-shot MD5 digest, returned as a 22-character base64 string.

Synopsis

```
use Digest::MD5 qw(md5_base64);
my $b64 = md5_base64($data);      # 22 chars, no padding
```

What you get back

A 22-character string from the standard base64 alphabet A-Z a-z 0-9 + /. The tail is **not** padded to a multiple of 4 - the redundancy would encode nothing - so if an interop partner demands strict base64, append "==" yourself.

Base64 is about 30% shorter than hex and still fits on one line in a log or URL query parameter. The price is that it contains / and +, which are not safe in URL paths without further escaping.

Examples

Compact digest for a cache key or tag:

```
use Digest::MD5 qw(md5_base64);
my $tag = md5_base64($payload);

## 22-character identifier, e.g. "rL0Y20zC+Fzt72VPzMsk2A"
```

Pad for strict base64 parsers:

```
my $padded = md5_base64($payload) . "==";
```

Store alongside the content in a manifest:

```
printf "%s %s\n", md5_base64($bytes), $name;
```

Edge cases

- `md5_base64()` with no arguments returns `1B2M2Y8AsgTpgAmY7PhCfg`.
- Wide-character arguments croak; encode first.
- The + and / characters require percent-encoding if the result appears in a URL path. Consider a URL-safe base64 transform (`tr|+ /| - _|`) when serving digests in URLs.

Differences from upstream

Fully compatible with upstream `Digest::MD5` 2.58.

See also

- md5 - raw-byte form
- md5_hex - hex form (longer, URL-safer)
- b64digest - OO form over a reusable context
- add - feed data incrementally for larger inputs

Digest::SHA

Compute SHA-1 and SHA-2 family message digests: SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256, with optional HMAC keying.

Two call styles are available. The **functional interface** (`sha256_hex`, `hmac_sha512_base64`, and friends) is a one-shot: hand it the data, get the digest back. The **object interface** (`Digest::SHA->new(256)`, `->add`, `->digest`) lets you feed data incrementally - from streams, files, or a mix of sources - and finalize when you are ready.

Every digest can be rendered in three encodings: raw binary (fixed-length bytes), lowercase hex, or unpadded Base64. Choose the encoding by the function suffix (`_hex`, `_base64`, or none for raw). HMAC variants use the same suffixes and live under the `hmac_*` prefix.

File input is supported through `addfile`, which reads a filehandle or filename and feeds its contents straight into the context. A bits helper (`add_bits`) lets you hash arbitrary-length bit strings, not just whole bytes. Context state is serialisable with `getstate/putstate(text)` and `_getstate/_putstate(packed binary)`, so a long-running hash can be suspended and resumed across processes.

Pick **SHA-256** by default for new work: it is fast on 32-bit and 64-bit CPUs alike and has the widest interoperability. Prefer **SHA-512** (or its truncated SHA-512/256 cousin) on 64-bit hardware when you want more performance headroom or a larger internal state; SHA-512 often outperforms SHA-256 there despite the larger output. Avoid **SHA-1** for anything collision-sensitive - it survives only for legacy interoperability.

Functions

Functional digests

`sha_functional`

Compute a SHA digest of the concatenated arguments in one call.

OO lifecycle

`newSHA`

Construct a fresh `Digest::SHA` object for the given algorithm number.

new

Construct or reinitialise a `Digest::SHA` object.

clone

Copy a `Digest::SHA` object, producing an independent duplicate.

destroy

Release the native context attached to a `Digest::SHA` object.

reset

Rewind a `Digest::SHA` object, optionally switching algorithm.

OO input

add

Feed byte data into a `Digest::SHA` object incrementally.

add_bits

Feed an arbitrary number of bits into a `Digest::SHA` object.

addfilebin

Feed the raw byte contents of a filehandle into a `Digest::SHA` object.

addfileuniv

Feed a filehandle's contents into a `Digest::SHA` object, reading through the PerlIO text layer.

addfile

Feed a filehandle or a file by name into a `Digest::SHA` object.

OO output

digest

Finalise the running hash and return the raw binary digest.

hexdigest

Finalise the running hash and return the digest as a lowercase hex string.

b64digest

Finalise the running hash and return the digest as an unpadded Base64 string.

algorithm

Return the algorithm code of a `Digest::SHA` object.

hashsize

Return the digest length of a `Digest::SHA` object in bits.

HMAC

hmac_functional

Compute a keyed HMAC digest of the data arguments in one call.

State export

getstate

Export a `Digest::SHA` object's state as a packed binary string.

putstate

Restore a `Digest::SHA` object's state from a packed binary string.

getstate_text

Export a `Digest::SHA` object's state as a human-readable text string.

putstate_text

Construct or restore a `Digest::SHA` object from a text state string.

dump

Write the current `Digest::SHA` state to a file or STDOUT in text form.

load

Construct a `Digest::SHA` object from a text state file written by dump.

sha_functional

Compute a SHA digest of the concatenated arguments in one call.

This is the dispatcher behind every functional hash entry point - all 21 shapes (7 algorithms × 3 encodings) route through it. The Perl-level name selects both the algorithm and the output encoding:

- sha1, sha224, sha256, sha384, sha512, sha512224, sha512256 return the **raw binary** digest (a fixed-length byte string).
- sha1_hex, sha224_hex, ... return the **lowercase hex** digest.
- sha1_base64, sha224_base64, ... return the **unpadded Base64** digest.

All arguments are concatenated byte-wise before hashing, so sha256_hex("foo", "bar") is identical to sha256_hex("foobar").

Synopsis

```
use Digest::SHA qw(sha256_hex sha512_base64);
my $hex = sha256_hex($data);
my $b64 = sha512_base64(@chunks);
```

What you get back

A plain scalar. For the raw-binary variants it is a byte string of fixed length (20 bytes for SHA-1, 28 for SHA-224, 32 for SHA-256, 48 for SHA-384, 64 for SHA-512). For _hex variants it is twice that length. For _base64 variants it is the unpadded Base64 encoding (no trailing = characters).

Examples

```
use Digest::SHA qw(sha256_hex);
print sha256_hex("abc");

## e.g. ↵
↵ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad
```

```
use Digest::SHA qw(sha1_hex);
my $digest = sha1_hex("The quick brown fox ", "jumps over the lazy ↵
↵dog");

## identical to sha1_hex("The quick brown fox jumps over the lazy ↵
↵dog")
```

```
use Digest::SHA qw(sha512_base64);
my $tag = sha512_base64($payload);

## 86-character unpadded Base64 string
```

```
use Digest::SHA qw(sha256);
my $raw = sha256("");      # 32 raw bytes
print length $raw;        # 32
```

Edge cases

- No arguments: hashes the empty string. `sha256_hex()` returns the well-known `e3b0c442...` empty-input digest.
- Wide characters (codepoint above 255) croak with `Wide character in subroutine entry - encode to UTF-8 first if you want to hash text above Latin-1.`

Differences from upstream

Fully compatible with upstream `Digest::SHA 6.04`.

See also

- `hmac_sha256_hex` - keyed variant for message authentication.
- `Digest::SHA->new(256)` - incremental interface for large or streamed input.
- `sha256` - raw-binary counterpart of `sha256_hex`.

hmac_functional

Compute a keyed HMAC digest of the data arguments in one call.

Shared dispatcher for every `hmac_sha*` one-shot: 21 Perl-level names (7 algorithms $\times$ 3 encodings) all land here. The name selects both the underlying SHA algorithm and the output encoding:

- `hmac_sha1`, `hmac_sha256`, ... return the **raw binary** MAC.
- `hmac_sha1_hex`, `hmac_sha256_hex`, ... return the **lowercase hex** MAC.
- `hmac_sha1_base64`, `hmac_sha256_base64`, ... return the **unpadded Base64** MAC.

Argument shape matches upstream: the **last** argument is the key, every preceding argument is message data (concatenated byte-wise). With a single argument the key is that argument and the message is empty. With no arguments both key and message are empty.

Synopsis

```
use Digest::SHA qw(hmac_sha256_hex hmac_sha1_base64);
my $tag = hmac_sha256_hex($message, $key);
my $b64 = hmac_sha1_base64(@chunks, $key);
```

What you get back

A plain scalar. The raw-binary form has the fixed length of the chosen algorithm's digest (20 bytes for SHA-1, 32 for SHA-256, 64 for SHA-512, and so on). The `_hex` form is twice that length; the `_base64` form is the unpadded Base64 encoding.

Examples

```
use Digest::SHA qw(hmac_sha256_hex);
my $mac = hmac_sha256_hex("Hi There", "\x0b" x 20);

## b0344c61d8db38535ca8afceaf0bf12b881dc200c9833da726e9376c2e32cff7
```

```
use Digest::SHA qw(hmac_sha1_hex);

## Concatenated message: "Hello, " + "world!"

my $mac = hmac_sha1_hex("Hello, ", "world!", $shared_secret);
```

```
use Digest::SHA qw(hmac_sha512_base64);
my $cookie_tag = hmac_sha512_base64($cookie_body, $server_secret);
```

```
use Digest::SHA qw(hmac_sha256_hex);
my $tag = hmac_sha256_hex($key);    # one arg: key only, empty_
    ↪message
```

Edge cases

- Single-argument calls treat the sole argument as the key and hash an empty message. This matches upstream behaviour and is rarely what you want - pass message and key explicitly.
- Keys longer than the algorithm's block size are hashed down to digest size before use (standard HMAC key reduction).
- Wide characters in either key or data croak with Wide character in subroutine entry.

Differences from upstream

Fully compatible with upstream Digest::SHA 6.04.

See also

- sha256_hex - unkeyed one-shot digest for integrity-only needs.
- Digest::SHA->new(256)->add(...)->hmac_hex(\$key) - no OO HMAC exists; if you need incremental HMAC use Digest::HMAC_SHA1 or feed a keyed construction manually.
- hmac_sha1_hex - SHA-1 variant; prefer SHA-256 or larger for new code.

newSHA

Construct a fresh Digest::SHA object for the given algorithm number.

Low-level constructor that takes an algorithm code directly. Most code calls new instead; newSHA is the path that the XS layer exposes and that new eventually delegates to. Valid

algorithm codes are 1, 224, 256, 384, 512, 512224, and 512256. Any other value returns `undef` instead of croaking.

Synopsis

```
my $sha = Digest::SHA::newSHA("Digest::SHA", 256);
```

What you get back

A blessed `Digest::SHA` object, ready to accept `add`, `addfile`, or `digest` calls. Returns `undef` on an unknown algorithm code.

Examples

```
my $sha = Digest::SHA::newSHA("Digest::SHA", 512);
$sha->add("hello");
print $sha->hexdigest;
```

```
my $bad = Digest::SHA::newSHA("Digest::SHA", 999);
print defined $bad ? "ok" : "undef"; # undef
```

Edge cases

- Missing algorithm or missing class name croaks with a usage error.
- Unknown algorithm code returns `undef`; check the return value before using it.

Differences from upstream

Fully compatible with upstream `Digest::SHA` 6.04.

See also

- `new` - higher-level constructor that accepts strings like "SHA-256".
- `clone` - copy an existing context instead of starting over.
- `reset` - reinitialise an existing object, optionally switching algorithm.

new

Construct or reinitialise a `Digest::SHA` object.

The primary object constructor. Works as both a class method (`Digest::SHA->new(256)`) and an instance method (`$sha->new(512)`), and accepts the algorithm in any of the shapes `Digest::SHA` recognises: an integer like 256, a string like "SHA-256", "sha256", or "512/224". When called without an algorithm argument it defaults to SHA-1.

As an instance method, `new` reuses the existing object: supplying a new algorithm resets state and switches to that algorithm; omitting it simply rewinds the state (same as `reset`).

Synopsis

```
my $sha = Digest::SHA->new(256);
my $sha = Digest::SHA->new("SHA-512/256");
$sha->new;                                # rewind, keep algorithm
$sha->new(384);                            # rewind, switch to SHA-384
```

What you get back

A blessed `Digest::SHA` object, or `undef` when the algorithm argument cannot be parsed as a supported algorithm.

Examples

```
my $sha = Digest::SHA->new("sha256");
$sha->add("The quick brown fox ", "jumps over the lazy dog");
print $sha->hexdigest;

## d7a8fbb307d7809469ca9abcb0082e4f8d5651e46d3cdb762d02d0bf37c9e592
```

```
my $sha = Digest::SHA->new(512);
$sha->add_bits("01011010");           # hash exactly 8 bits
```

```
my $sha = Digest::SHA->new(256);
$sha->add("first");
$sha->new(512);                        # rewind and switch
$sha->add("second");                  # now hashing with SHA-512
```

Edge cases

- No algorithm argument defaults to SHA-1.
- An unparseable algorithm returns `undef`; check the return value before chaining `->add`.
- Called on an existing object, `new` mutates that object in place and returns it - it does not allocate a second context.

Differences from upstream

Fully compatible with upstream `Digest::SHA 6.04`.

See also

- `reset` - explicit rewind, same semantics as the no-argument instance form.
- `clone` - copy the current state into a fresh object.
- `newSHA` - lower-level XS constructor taking an integer algorithm code.

clone

Copy a `Digest::SHA` object, producing an independent duplicate.

Returns a new object with the same algorithm and the same internal state

- including any data already absorbed via `add` or `addfile`. The copy and the original are independent from that moment on: hashing more data into one does not affect the other.

Useful when you want several digests that share a common prefix, for example hashing a large buffer followed by a per-record suffix.

Synopsis

```
my $base = Digest::SHA->new(256);
$base->add($prefix);
my $copy = $base->clone;
```

What you get back

A fresh blessed `Digest::SHA` object with state copied from the receiver. Returns `undef` if called on something that is not a valid `Digest::SHA` object.

Examples

```
my $base = Digest::SHA->new(256);
$base->add("common-prefix:");
for my $id (1 .. 3) {
    my $h = $base->clone;
    $h->add($id);
    print $h->hexdigest, "\n";    # three distinct digests
}
```

```
my $sha = Digest::SHA->new(512);
$sha->add("abc");
my $copy = $sha->clone;
print $copy->hexdigest eq $sha->hexdigest ? "same\n" : "diff\n"; #
↪same
```

Edge cases

- The copy is a snapshot: subsequent add on the receiver does not propagate into it.
- `clone` on a `non-Digest::SHA` scalar returns `undef` rather than croaking.

Differences from upstream

Fully compatible with upstream `Digest::SHA 6.04`.

See also

- `new` - start a fresh context instead of branching from an existing one.
- `reset` - reuse the same object for the next digest.
- `getstate` / `putstate` - snapshot to a string you can send across processes.

destroy

Release the native context attached to a `Digest::SHA` object.

The Perl destructor. Invoked automatically when the last reference to a `Digest::SHA` object goes out of scope - you do not normally call it yourself. It frees the heap-allocated hashing context behind the blessed reference.

Synopsis

```
{  
    my $sha = Digest::SHA->new(256);  
    # ... use $sha ...  
} # DESTROY runs here when $sha leaves scope
```

What you get back

Nothing. `DESTROY` returns to Perl with no value.

Examples

```
my $sha = Digest::SHA->new(256);  
undef $sha;    # triggers DESTROY
```

```
## Double-free safety: explicit DESTROY followed by scope exit is  
→harmless.
```

```
my $sha = Digest::SHA->new(256);  
$sha->DESTROY;
```

```
## automatic DESTROY on scope exit is a no-op after the explicit  
→call
```

Edge cases

- Calling DESTROY on an object that is not a blessed `Digest::SHA` reference is silently ignored.
- After DESTROY, the object's internal pointer is zeroed, so repeated calls are safe.

Differences from upstream

Fully compatible with upstream `Digest::SHA` 6.04.

See also

- `new` - counterpart: allocates the context that DESTROY releases.
- `clone` - each clone also owns a context and is destroyed independently.

reset

Rewind a `Digest::SHA` object, optionally switching algorithm.

Clears buffered data and restores the initial hash state so the same object can be reused for a fresh digest. With no argument the algorithm is preserved; with an algorithm argument the object is rewound **and** retargeted at that algorithm. Accepts the same algorithm shapes as `new` (integer code or string like "SHA-256").

Synopsis

```
$sha->reset;           # rewind, keep algorithm
$sha->reset(512);       # rewind and switch to SHA-512
$sha->reset("SHA-384"); # string form
```

What you get back

The receiver on success (chainable), or `undef` if an algorithm argument was supplied that could not be parsed.

Examples

```
my $sha = Digest::SHA->new(256);
$sha->add("first");
my $a = $sha->hexdigest;      # finalises and auto-resets already
$sha->reset;                  # defensive rewind before reuse
$sha->add("second");
my $b = $sha->hexdigest;
```

```
my $sha = Digest::SHA->new(256);
$sha->reset(512)->add("hello");
print $sha->algorithm;        # 512
```



```
my $sha = Digest::SHA->new(256);
my $ok = $sha->reset("bogus"); # undef
```

Edge cases

- Called on a non-Digest::SHA object, returns undef.
- After the built-in auto-reset inside digest, calling reset explicitly is harmless.

Differences from upstream

Fully compatible with upstream Digest::SHA 6.04.

See also

- new - when called on an instance, functionally equivalent to reset.
- digest - auto-rewinds after finalisation, so explicit reset is only needed when you abandon a computation partway.
- clone - if you want a copy of the pre-reset state, clone first.

add

Feed byte data into a Digest::SHA object incrementally.

Appends each argument to the running hash in order. Use add when the data does not fit comfortably in memory, when you are assembling the message from many sources, or when you want to branch the computation via clone. Calling add(\$a, \$b, \$c) is equivalent to calling add(\$a)->add(\$b)->add(\$c) or to calling add(\$a . \$b . \$c).

Arguments are treated as byte strings via SvPVbyte semantics: raw bytes pass straight through, UTF-8 strings are downgraded to Latin-1 bytes, and wide characters (codepoint above 255) croak.

Synopsis

```
$sha->add($data);
$sha->add(@chunks);
$sha->add($a)->add($b)->add($c);
```

What you get back

The receiver, so calls chain.

Examples

```
my $sha = Digest::SHA->new(256);
$sha->add("abc");
```

(continues on next page)

(continued from previous page)

```
print $sha->hexdigest;

## ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad
```

```
my $sha = Digest::SHA->new(512);
while (defined(my $line = <$fh>)) {
    $sha->add($line);
}
print $sha->hexdigest;
```

```
my $sha = Digest::SHA->new(256);
$sha->add("a", "b", "c");

## identical to $sha->add("abc")
```

Edge cases

- Empty strings are silently absorbed with no effect on the hash.
- A wide character in any argument croaks with `Wide character in subroutine entry`; encode to UTF-8 explicitly before `add` if you want to hash non-Latin-1 text.
- Called on an object that is not a `Digest::SHA`, returns `undef`.

Differences from upstream

Fully compatible with upstream `Digest::SHA 6.04`.

See also

- `addfile` - feed a filehandle or filename directly, avoiding a read loop.
- `add_bits` - hash a sub-byte number of bits.
- `digest` - finalise and return the hash once all data is fed.

add_bits

Feed an arbitrary number of bits into a `Digest::SHA` object.

Unlike `add`, this method is not restricted to whole bytes. It accepts two call forms:

- **One-argument form** `add_bits($bitstr)` hashes a string of '0' and '1' characters - the hash takes one bit per character, so "01011010" adds exactly 8 bits of input.
- **Two-argument form** `add_bits($data, $numbits)` hashes the first `$numbits` bits of the packed binary string `$data`. If `$numbits` exceeds `8 * length($data)` it is capped at that value.

Most of the SHA standard is defined on bit-strings rather than byte-strings; this is the entry point for the bit-exact test vectors in FIPS 180-4.

Synopsis

```
$sha->add_bits("11001010");      # 8 bits from ASCII 0/1 string
$sha->add_bits($packed, 13);      # first 13 bits of $packed
```

What you get back

The receiver, for chaining.

Examples

```
my $sha = Digest::SHA->new(256);
$sha->add_bits("01011010" x 8);  # 64 bits
print $sha->hexdigest;
```

```
## FIPS 180-4 1-bit test vector for SHA-1

my $sha = Digest::SHA->new(1);
$sha->add_bits("0");
print $sha->hexdigest;

## bb6b3e18f0115b9f9ededbb2a9cbfd2fe4c1a1a0
```

```
my $sha = Digest::SHA->new(256);
$sha->add_bits("\xab\xcd", 12);  # first 12 bits of 0xabcd
```

Edge cases

- The one-argument form ignores any character that is not '0' or '1'; an empty or all-non-bit string is a no-op.
- In the two-argument form, passing a negative or zero bit count leaves the hash state untouched.
- Called on a non-Digest::SHA object, returns undef.

Differences from upstream

Fully compatible with upstream Digest::SHA 6.04.

See also

- `add` - standard byte-aligned input path; use it unless you truly have sub-byte data.
- `digest` - finalises the running hash once input is complete.
- `addfile` - byte-oriented file ingestion.

addfilebin

Feed the raw byte contents of a filehandle into a `Digest::SHA` object.

The binary-mode backend behind `addfile`. Reads until end of file with no newline translation and no character interpretation - exactly what you want when hashing a file whose bytes must be preserved literally (archives, images, compiled artefacts).

Normally you use `addfile` with an explicit "b" mode; `_addfilebin` is the leading-underscore XS hook, exposed here for completeness.

Synopsis

```
open my $fh, "<:raw", "archive.tar.gz" or die $!;
$sha->_addfilebin($fh);
```

What you get back

The receiver, for chaining.

Examples

```
open my $fh, "<:raw", "data.bin" or die $!;
my $sha = Digest::SHA->new(256);
$sha->_addfilebin($fh);
print $sha->hexdigest;
```

```
my $sha = Digest::SHA->new(512);
$sha->_addfilebin("path/to/file"); # also accepts filenames
```

Edge cases

- A bad filehandle or unreadable filename croaks with `Bad filehandle: ....`
- An empty file leaves the hash state untouched.
- Called on a non-`Digest::SHA` object, returns `undef`.

Differences from upstream

Fully compatible with upstream `Digest::SHA 6.04`.

See also

- `addfile` - the public, mode-switching wrapper; prefer it in new code.
- `_addfileuniv` - universal-newline counterpart.
- `add` - string-oriented input for data already in memory.

addfileuniv

Feed a filehandle's contents into a `Digest::SHA` object, reading through the PerlIO text layer.

The universal-newline backend behind `addfile`. On platforms where Perl performs CR/LF conversion on text streams, this mode reads the file through that conversion so the hash is platform-agnostic. On Linux the PerlIO text layer is a no-op, so on this platform `_addfileuniv` and `_addfilebin` produce identical digests.

Prefer the public `addfile($fh, "U")` or `addfile($fh, "p")` wrapper in application code.

Synopsis

```
open my $fh, "<", "notes.txt" or die $!;
$sha->_addfileuniv($fh);
```

What you get back

The receiver, for chaining.

Examples

```
my $sha = Digest::SHA->new(256);
open my $fh, "<", "README" or die $!;
$sha->_addfileuniv($fh);
print $sha->hexdigest;
```

```
## Portable digest of a text file shared between Unix and Windows:

my $sha = Digest::SHA->new(256);
$sha->_addfileuniv("config.ini");
```

Edge cases

- On Linux, newline translation is inactive; the digest matches `_addfilebin` on the same file.
- A bad filehandle or unreadable filename croaks with `Bad filehandle: ....`
- Called on a non-`Digest::SHA` object, returns `undef`.

Differences from upstream

Fully compatible with upstream `Digest::SHA` 6.04.

See also

- `addfile` - mode-switching wrapper; prefer it in application code.
- `_addfilebin` - byte-exact binary counterpart.
- `add_bits` - sub-byte input path for bit-oriented hashing.

addfile

Feed a filehandle or a file by name into a `Digest::SHA` object.

The public file-ingestion entry point. Accepts either an open filehandle or a filename string, plus an optional mode character that controls how bytes are read:

- `"b"` (default): raw binary read. Use this for anything that is not human-readable text, or when you need bit-exact file digests.
- `"U"` or `"p"`: read through the PerlIO text layer. On platforms that perform CR/LF translation this yields platform-independent digests of text files; on Linux it reads identically to `"b"`.
- `"0"`: BITS mode. Interpret every `'0'` and `'1'` character in the file as a single bit and feed those bits into the hash. Anything else in the file is ignored.

Synopsis

```
$sha->addfile("data.bin");           # filename, binary
$sha->addfile($fh);                   # handle, binary
$sha->addfile($fh, "U");              # universal newlines
$sha->addfile("bits.txt", "0");       # ASCII bit stream
```

What you get back

The receiver, for chaining.

Examples

```
my $sha = Digest::SHA->new(256);
$sha->addfile("archive.tar.gz");
print $sha->hexdigest;
```

```
## Cross-platform digest of a source file
```

```
my $sha = Digest::SHA->new(256);
$sha->addfile("src/main.c", "U");
```

```
## BITS mode: hash the bit pattern "01011010" repeated
```

```
open my $fh, "<", "bitstream.txt" or die $!;
my $sha = Digest::SHA->new(1);
$sha->addfile($fh, "0");
```

```
my $sha = Digest::SHA->new(256);
open my $fh, "<:raw", "photo.jpg" or die $!;
$sha->addfile($fh);
```

Edge cases

- Missing or unreadable filename, or a closed filehandle, croaks with `Bad filehandle: ....`
- Unknown mode characters fall back to binary.
- Empty files leave the hash state untouched.
- Called on a non-Digest::SHA object, returns `undef`.

Differences from upstream

Fully compatible with upstream Digest::SHA 6.04.

See also

- `add` - use when the data is already in a Perl scalar.
- `add_bits` - for bit-aligned input strings.
- `digest` - finalise and return the digest after the file has been consumed.

digest

Finalise the running hash and return the raw binary digest.

Completes the SHA computation on whatever bytes have been fed in so far (via `add`, `addfile`, `add_bits`) and returns the result as a byte string. After `digest` returns, the object is automatically reset, so you can feed a new message into the same object without calling `reset` explicitly.

The digest is returned as raw bytes - the fixed length of the chosen algorithm (20 bytes for SHA-1, 32 for SHA-256, 64 for SHA-512, and so on). If you want a printable form, use `hexdigest` or `b64digest` instead of converting yourself.

Synopsis

```
my $bytes = $sha->digest;
```

What you get back

A scalar holding the raw digest bytes. `length` equals `hashsize / 8`.

Examples

```
my $sha = Digest::SHA->new(256);
$sha->add("abc");
my $raw = $sha->digest;
print length $raw;          # 32
```

```
my $sha = Digest::SHA->new(1);
$sha->add("hello");
my $a = $sha->digest;
$sha->add("world");          # fresh message after auto-reset
my $b = $sha->digest;
```

```
use Digest::SHA;
my $raw = Digest::SHA->new(256)->add($data)->digest;
```

Edge cases

- Called on an object with no input fed, returns the digest of the empty string.
- The auto-reset is unconditional: do not assume the hash state is preserved after digest returns.
- Called on a non-Digest::SHA object, returns undef.

Differences from upstream

Fully compatible with upstream Digest::SHA 6.04.

See also

- hexdigest - same digest, lowercase hex string.
- b64digest - same digest, unpadded Base64 string.
- clone - take a snapshot before finalising if you want to keep hashing.

hexdigest

Finalise the running hash and return the digest as a lowercase hex string.

Identical to `digest` except for encoding: the result is a string of lowercase hexadecimal characters, twice the length of the raw digest (40 chars for SHA-1, 64 for SHA-256, 128 for SHA-512, etc.). As with `digest`, the object is automatically reset after the call.

Synopsis

```
my $hex = $sha->hexdigest;
```


What you get back

A string of [0-9a-f] characters of length `hashsize / 4`.

Examples

```
my $sha = Digest::SHA->new(256);
$sha->add("abc");
print $sha->hexdigest;

## ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad
```

```
my $sha = Digest::SHA->new(1);
print $sha->hexdigest;

## da39a3ee5e6b4b0d3255bfef95601890afd80709    (empty-input SHA-1)
```

```
use Digest::SHA;
print Digest::SHA->new(512)->add("The quick brown fox")->hexdigest,
  "\n";
```

Edge cases

- Auto-resets the object, same as `digest`.
- Called on an object with no input fed, returns the hex digest of the empty string (da39a3ee... for SHA-1, e3b0c442... for SHA-256).
- Called on a non-Digest::SHA object, returns `undef`.

Differences from upstream

Fully compatible with upstream Digest::SHA 6.04.

See also

- `digest` - raw-byte counterpart.
- `b64digest` - unpadded Base64 counterpart.
- `sha256_hex` - functional one-shot with the same output shape.

b64digest

Finalise the running hash and return the digest as an unpadded Base64 string.

Identical to `digest` except that the result is the standard Base64 encoding of the digest **without** trailing = padding. The length depends on algorithm: 27 characters for SHA-1, 43 for SHA-256, 86 for SHA-512, etc. As with `digest` and `hexdigest`, the object is reset after the call.

Synopsis

```
my $b64 = $sha->b64digest;
```

What you get back

A Base64 string using the standard alphabet, with no = padding.

Examples

```
my $sha = Digest::SHA->new(256);
$sha->add("abc");
print $sha->b64digest;

## ungWv48Bz+pBQUDeXa4iI7ADYa0WF3qctBD/YfIAFa0
```

```
## Compact integrity tag for a message body:

my $tag = Digest::SHA->new(256)->add($body)->b64digest;
```

```
use Digest::SHA;
print Digest::SHA->new(512)->add("hello")->b64digest, "\n";
```

Edge cases

- Auto-resets the object.
- Returned string never contains = - reattach padding yourself if you need to decode against a strict Base64 parser that requires it.
- Called on a non-Digest::SHA object, returns undef.

Differences from upstream

Fully compatible with upstream Digest::SHA 6.04.

See also

- digest - raw-byte counterpart.
- hexdigest - hexadecimal counterpart.
- sha256_base64 - functional one-shot with the same output shape.

algorithm

Return the algorithm code of a Digest::SHA object.

The code is the same integer you can pass to new: 1, 224, 256, 384, 512, 512224, or 512256. Useful when a function receives a Digest::SHA object from elsewhere and needs to act differently based on which variant it is.

Synopsis

```
my $code = $sha->algorithm;
```

What you get back

An integer in the set {1, 224, 256, 384, 512, 512224, 512256}.

Examples

```
my $sha = Digest::SHA->new("SHA-256");  
print $sha->algorithm;    # 256
```

```
my $sha = Digest::SHA->new("SHA-512/224");  
print $sha->algorithm;    # 512224
```

```
## Dispatch on algorithm:  
  
my $code = $sha->algorithm;  
my $name = $code == 1 ? "SHA-1" : "SHA-$code";
```

Edge cases

- Called on a non-Digest::SHA object, returns undef.

Differences from upstream

Fully compatible with upstream Digest::SHA 6.04.

See also

- `hashsize` - digest length in bits for this algorithm.
- `new` - accepts the values returned here as its algorithm argument.

hashsize

Return the digest length of a Digest::SHA object in bits.

For a given algorithm the hashsize is a constant: 160 for SHA-1, 224 for SHA-224, 256 for SHA-256 and SHA-512/256, 384 for SHA-384, 512 for SHA-512. Divide by 8 to get the byte length of the output from digest.

Synopsis

```
my $bits = $sha->hashsize;
```

What you get back

An integer, always a multiple of 8.

Examples

```
my $sha = Digest::SHA->new(256);
print $sha->hashsize;           # 256
print $sha->hashsize / 8;       # 32 (bytes per digest)
```

```
my $sha = Digest::SHA->new("SHA-512/224");
print $sha->hashsize;           # 224
```

```
## Preallocate output buffer for $sha->digest:

my $buf = "\0" x ($sha->hashsize / 8);
```

Edge cases

- Called on a non-Digest::SHA object, returns undef.

Differences from upstream

Fully compatible with upstream Digest::SHA 6.04.

See also

- `algorithm` - the integer algorithm code for this object.
- `digest` - output whose length is `hashsize / 8` bytes.

getstate

Export a Digest::SHA object's state as a packed binary string.

Returns the full hashing state - the intermediate hash values, buffered partial block, block counter, and message length counters - as a binary scalar. Pair with `_putstate` to restore the state later, for example when resuming a long hash across processes or persisting it to disk.

Prefer the text-based `getstate` / `putstate` pair for human-readable output or for interchange across endianness. The binary form here is native-byte-order and tightly packed.

Synopsis

```
my $blob = $sha->_getstate;
```

What you get back

A binary string of fixed length: 116 bytes for SHA-1/224/256, 212 bytes for SHA-384/512/512-224/512-256.

Examples

```

my $sha = Digest::SHA->new(256);
$sha->add("hello ");
my $snapshot = $sha->_getstate;

my $other = Digest::SHA->new(256);
$other->_putstate($snapshot);
$other->add("world");
print $other->hexdigest;           # same as full "hello world"
↪hash

```

```

## Persist a half-finished hash to disk:

open my $fh, ">", "hash.state" or die $!;
binmode $fh;
print $fh $sha->_getstate;

```

Edge cases

- The output is binary and includes null bytes; never interpret it as text.
- Called on a non-Digest::SHA object, returns undef.

Differences from upstream

Fully compatible with upstream Digest::SHA 6.04.

See also

- `_putstate` - restore state produced by this method.
- `getstate` - human-readable text variant.
- `dump / load` - higher-level convenience wrappers around text state.

putstate

Restore a Digest::SHA object's state from a packed binary string.

The inverse of `_getstate`. Validates the length of the supplied blob against the receiver's algorithm and rewrites the internal hash state in place. After a successful `_putstate`, feeding more data with `add` continues the hash as if no break had occurred.

The receiver's algorithm is **not** changed by this call; it must already match the algorithm the blob was produced under.

Synopsis

```
$sha->_putstate($blob);
```

What you get back

The receiver on success, undef on failure (invalid length or bad object).

Examples

```
my $a = Digest::SHA->new(256);
$a->add("first half");
my $blob = $a->_getstate;

my $b = Digest::SHA->new(256);
$b->_putstate($blob) or die "restore failed";
$b->add("second half");
print $b->hexdigest;
```

```
## Round-trip through a file:

open my $in, "<", "hash.state" or die $!;
binmode $in;
my $blob = do { local $/; <$in> };
my $sha = Digest::SHA->new(256);
$sha->_putstate($blob);
```

Edge cases

- Blob length must match the receiver's algorithm (116 vs 212 bytes); a mismatch returns undef.
- Called on a non-Digest::SHA object, returns undef.

Differences from upstream

Fully compatible with upstream Digest::SHA 6.04.

See also

- `_getstate` - produces the blob consumed here.
- `putstate` - text-based restore that also creates new objects.
- `load` - text-based file-loading counterpart.

getstate_text

Export a Digest::SHA object's state as a human-readable text string.

Produces a multi-line plain-text dump of the current hashing state, including algorithm number, hash registers, pending block, and message length counters. The text form is portable across machines of different endianness and lends itself to inspection, diffing, and manual editing.

Pair with putstate to reconstruct the state later.

Synopsis

```
my $text = $sha->getstate;
```

What you get back

A newline-terminated ASCII string.

Examples

```
my $sha = Digest::SHA->new(256);
$sha->add("hello");
print $sha->getstate;

## alg:256

## H:...

## block:...

## blockcnt:40

## lenhh:0 lenhl:0 lenlh:0 lenll:40
```

```
## Resume on another machine:

my $text = $sha->getstate;
...
my $resumed = Digest::SHA->putstate($text);
$resumed->add($more_data);
```

Edge cases

- The text format is stable across Digest::SHA versions and endianness.
- Called on a non-Digest::SHA object, returns undef.

Differences from upstream

Fully compatible with upstream `Digest::SHA 6.04`.

See also

- `putstate` - inverse: parses text and constructs or updates an object.
- `dump` - write the text state to a file or `STDOUT`.
- `_getstate` - binary counterpart for same-machine use.

putstate_text

Construct or restore a `Digest::SHA` object from a text state string.

Accepts the text produced by `getstate` or `dump` and either:

- As a **class method** (`Digest::SHA->putstate($text)`), returns a new object with algorithm and state taken from the string.
- As an **instance method** (`$sha->putstate($text)`), rewrites the receiver in place, switching algorithm if the string asks for a different one.

The algorithm is read from the text, so - unlike `_putstate` - you do not need to preconfigure the object with the correct algorithm.

Synopsis

```
my $sha = Digest::SHA->putstate($text);
$sha->putstate($text);
```

What you get back

A `Digest::SHA` object on success, `undef` if the string could not be parsed.

Examples

```
my $text = $other->getstate;
my $sha  = Digest::SHA->putstate($text);
$sha->add($remaining_data);
```

```
## Update an existing object from a state string:
```

```
my $sha = Digest::SHA->new(256);
$sha->putstate($text);
```

```
## Invalid input returns undef:
```

```
my $sha = Digest::SHA->putstate("garbage");
print defined $sha ? "ok" : "undef"; # undef
```


Edge cases

- Unrecognised or truncated text returns undef rather than croaking.
- When used as an instance method, the receiver's algorithm is overwritten by whatever the text specifies.

Differences from upstream

Fully compatible with upstream `Digest::SHA 6.04`.

See also

- `getstate` - produces the strings this method consumes.
- `load` - convenience wrapper that reads the text from a file.
- `_putstate` - binary-state counterpart for same-machine use.

dump

Write the current `Digest::SHA` state to a file or STDOUT in text form.

With a filename argument, writes the text produced by `getstate` to that file, truncating it if it exists. Without an argument, prints the same text to STDOUT. The companion `load` method reads it back.

Synopsis

```
$sha->dump("hash.state");  
$sha->dump;                # to STDOUT
```

What you get back

1 on success (truthy), undef on I/O failure.

Examples

```
my $sha = Digest::SHA->new(256);  
$sha->add("partial");  
$sha->dump("checkpoint.state");  
  
## ... later, perhaps in another process ...  
  
my $resumed = Digest::SHA->load("checkpoint.state");  
$resumed->add("tail");  
print $resumed->hexdigest;
```

```
## Debug-dump to the terminal:  
  
$sha->dump;
```

```
## Check the return value for I/O errors:

defined $sha->dump("/nonexistent/path/file")
    or warn "could not write state";
```

Edge cases

- An unwritable path returns undef rather than croaking.
- Called on a non-Digest::SHA object, returns undef.

Differences from upstream

Fully compatible with upstream Digest::SHA 6.04.

See also

- load - counterpart that reads the file dump produces.
- getstate - underlying text-state producer.
- putstate - text-state ingester.

load

Construct a Digest::SHA object from a text state file written by dump.

Reads the file, parses its contents as a text state string, and returns a new object with the algorithm and internal state recorded in the file. Typically called as a class method, but works as an instance method too (the receiver is ignored; a fresh object is always returned).

Synopsis

```
my $sha = Digest::SHA->load("hash.state");
```

What you get back

A Digest::SHA object on success, undef if the file could not be read or the contents could not be parsed.

Examples

```
my $sha = Digest::SHA->new(256);
$sha->add("one");
$sha->dump("hash.state");

my $resumed = Digest::SHA->load("hash.state");
$resumed->add("two");
```

(continues on next page)

(continued from previous page)

```
print $resumed->hexdigest;

## same as Digest::SHA->new(256)->add("one")->add("two")->hexdigest
```

```
## Missing file returns undef rather than croaking:
```

```
my $sha = Digest::SHA->load("/does/not/exist");
print defined $sha ? "ok" : "undef"; # undef
```

```
## Works as an instance method too (but still returns a fresh
↳ object):
```

```
my $dummy = Digest::SHA->new(1);
my $sha = $dummy->load("hash.state");
```

Edge cases

- I/O failure or a parse error returns undef.
- The returned object's algorithm comes from the file, not from any receiver - the receiver is essentially a namespace hook.

Differences from upstream

Fully compatible with upstream Digest::SHA 6.04.

See also

- dump - writes the file load reads.
- putstate - parses the text state directly from a string.
- _putstate - binary-state counterpart.

5.2.8 Encode

Convert between Perl character strings and bytes in any named encoding - UTF-8, UTF-16, Latin-1, CP1252, Shift_JIS, EUC-JP, and every other IANA-registered character set.

Encode works in two directions. `decode($name, $bytes)` takes raw bytes in the named encoding and returns a Perl character string (the `SVf_UTF8` flag is set on the result). `encode($name, $string)` goes the other way: it takes a character string and returns raw bytes in the target encoding. Keep the two operations mentally distinct - strings are sequences of Unicode codepoints, bytes are what you read from and write to files, sockets, and pipes.

Because UTF-8 is the internal form Perl uses for character strings, it has a fast path: `encode_utf8` and `decode_utf8` skip the full encoding machinery and just flip or validate the `SVf_UTF8` flag. Three low-level helpers let you poke at that flag directly: `is_utf8` queries it, `_utf8_on` forces it on, `_utf8_off` forces it off. Use those only when you

know what you are doing - they change how Perl interprets the bytes already in the scalar without touching the bytes themselves.

Every conversion takes an optional `$check` bitmask that controls what happens when a character cannot be represented in the target encoding. The predefined values are `FB_DEFAULT` (substitute with `?` or the encoding's replacement character), `FB_CROAK` (die), `FB_QUIET` (stop and return the converted prefix), `FB_WARN` (warn and substitute), `FB_HTMLCREF` (substitute with `&#NNNN;`), `FB_XMLCREF` (substitute with `&#xHHHH;`), and `FB_PERLQQ` (substitute with `\x{HHHH}`). `LEAVE_SRC`, `STOP_AT_PARTIAL`, `PERLQQ`, `WARN_ON_ERR`, and `ONLY_PRAGMA_WARNINGS` are the raw bits you OR together to build custom check values.

Encoding names are resolved through a registry. `find_encoding($name)` returns a blessed encoding object you can call methods on; `resolve_alias($name)` returns the canonical name as a string; `encodings()` lists every name the registry knows about.

`from_to($octets, $from, $to)` is a one-shot in-place conversion useful when all you want is to reencode a byte string - for example rewriting a file body from Latin-1 to UTF-8 without unpacking it into characters first. It handles BOM-tagged and MIME-tagged inputs when paired with `find_mime_encoding`.

Functions

Encode/decode

`native_encode`

Turn a character string into a byte string in the named encoding.

`native_decode`

Turn a byte string in the named encoding into a Perl character string.

`native_encode_utf8`

Fast path for encoding a string to UTF-8 bytes.

`native_decode_utf8`

Fast path for decoding UTF-8 bytes to a Perl character string.

UTF-8 flags

`native_is_utf8`

Return true if the scalar carries the `SVf_UTF8` flag.

`native_utf8_on`

Force `SVf_UTF8` on in place, without touching the underlying bytes.

native_utf8_off

Force SVf_UTF8 off in place, without touching the underlying bytes.

Encoding registry

native_find_encoding

Look up an encoding by name and return an object you can call methods on.

native_resolve_alias

Return the canonical encoding name for an alias, or undef if unknown.

native_encodings

Return the list of encoding names the registry knows about.

native_obj_encode

Method form of encode on an encoding object.

native_obj_decode

Method form of decode on an encoding object.

native_obj_name

Return the canonical name of an encoding object as a string.

native_obj_renew

Return a fresh encoding object (effectively a no-op returning \$self).

native_obj_perlio_ok

Return true if the encoding is safe to stack as a PerlIO layer.

MIME/XML helpers

native_fb_htmlcref

CHECK value 520 - replace unencodable characters with HTML decimal character references (&#NNNN;).

native_fb_xmlcref

CHECK value 1032 - replace unencodable characters with XML hexadecimal character references (&#xHHHH;).

Conversion

`native_from_to`

Reencode a byte string in place from one encoding to another.

Utilities

`native_fb_default`

CHECK value 0 - substitute unencodable characters with the encoding's default replacement (usually ? or U+FFFD).

`native_fb_croak`

CHECK value 1 - die on the first unencodable character or invalid byte sequence.

`native_fb_quiet`

CHECK value 4 - stop at the first unencodable character and return the encoded prefix; truncate the input to what was not consumed (method form only).

`native_fb_warn`

CHECK value 6 - warn and substitute on unencodable input.

`native_fb_perlqq`

CHECK value 264 - replace unencodable characters with Perl `\x{HHHH}` escape sequences.

`native_leave_src`

CHECK bit 8 - when OR'd into \$check, keeps the input scalar untouched; the default is to consume its successfully-encoded prefix.

`native_stop_at_partial`

CHECK bit 2048 - stop at a partial trailing multi-byte sequence rather than reporting it as an error. Useful for streaming decoders.

`native_perlqq`

CHECK bit 256 - the raw bit behind FB_PERLQQ; OR it into your own check mask for `\x{HHHH}` substitution.

native_warn_on_err

CHECK bit 2 - emit a warning on encoding errors. Combined with a substitution bit to build custom fallback behaviour.

native_only_pragma_warnings

CHECK bit 16 - emit encoding warnings only when the caller has use warnings 'utf8' (or equivalent) active, rather than unconditionally.

native_encode

Turn a character string into a byte string in the named encoding.

Synopsis

```
my $bytes = encode($encoding, $string);
my $bytes = encode($encoding, $string, $check);
```

What you get back

A scalar holding raw bytes. The SVf_UTF8 flag on the result is always off: this is the form you write to files, sockets, and pipes. The input \$string is treated as a sequence of Unicode codepoints regardless of how it is internally represented.

If \$encoding is unknown, encode croaks with Unknown encoding '...'.

The optional \$check argument controls what happens when a character cannot be represented in the target encoding:

- FB_DEFAULT (0, the default) - substitute with ? or the encoding's replacement character.
- FB_CROAK - die on the first unencodable character.
- FB_QUIET - stop at the first unencodable character and return the encoded prefix. In the method form, the consumed prefix is also removed from \$string.
- FB_WARN - warn and substitute.
- FB_PERLQQ - substitute with \x{HHHH}.
- FB_HTMLCREF - substitute with &#NNNN;.
- FB_XMLCREF - substitute with &#xHHHH;.

Examples

Encode a string to UTF-8 bytes for writing to a file:

```
my $bytes = encode('UTF-8', "caf\x{e9}");

## $bytes is "caf\xc3\xa9" - 4 bytes, no SVf_UTF8
```

Encode to Latin-1, losing characters that don't fit:

```
my $bytes = encode('iso-8859-1', "\x{20ac}"); # Euro sign

## $bytes is "?" - U+20AC has no Latin-1 byte
```

Die if anything can't be encoded:

```
use Encode qw(encode FB_CROAK);
my $bytes = encode('ascii', "caf\x{e9}", FB_CROAK);

## dies: "\x{e9}" does not map to ascii
```

Edge cases

- undef input returns an empty byte string.
- Input without SVf_UTF8 is treated as Latin-1 bytes and reencoded as such.
- Encoding "null" passes input bytes through unchanged.

Differences from upstream

Fully compatible with upstream for ASCII, Latin-1, CP1252, the ISO-8859 family, and UTF-8. Shift_JIS, EUC-JP, and other multi-byte encodings are not yet registered in the static table and fall back to the Latin-1 identity mapping. Covered by `t/81-xs-native/Encode/040-encode-utf8-latin1.t` and `t/81-xs-native/Encode/060-check-parameter.t`.

See also

- `decode` - the inverse, bytes to string.
- `encode_utf8` - the UTF-8-only fast path.
- `from_to` - reencode in place without materialising a character string in between.

native_decode

Turn a byte string in the named encoding into a Perl character string.

Synopsis

```
my $string = decode($encoding, $bytes);
my $string = decode($encoding, $bytes, $check);
```

What you get back

A scalar holding a Perl character string - the SVf_UTF8 flag is set on the result. Each element of that string is one Unicode codepoint; indexing with `substr` and counting with `length` returns characters, not bytes.

If `$encoding` is unknown, `decode` croaks with `Unknown encoding '...'`.

The optional `$check` argument controls what happens when the bytes are not valid in the source encoding:

- `FB_DEFAULT` (0, the default) - substitute invalid sequences with U+FFFD (the Unicode replacement character).
- `FB_CROAK` - die on the first invalid byte.
- `FB_QUIET` - decode the valid prefix and stop. In the method form, the consumed prefix is also removed from `$bytes`.
- `FB_WARN` - warn and substitute with U+FFFD.

Examples

Decode UTF-8 bytes read from a file:

```
my $string = decode('UTF-8', "caf\xc3\xa9");
## length($string) == 4, fourth char is U+00E9
```

Decode CP1252 bytes, turning Windows “smart quotes” into Unicode:

```
my $string = decode('cp1252', "\x93hi\x94");
## $string is "\x{201c}hi\x{201d}"
```

Die on malformed UTF-8:

```
use Encode qw(decode FB_CROAK);
my $string = decode('UTF-8', "\xc3\x28", FB_CROAK);
## dies: utf8 "\xC3" does not map to Unicode
```

Edge cases

- `undef` input returns an empty character string.
- Encoding `"null"` passes input bytes through unchanged.
- A byte string that is already valid UTF-8 is re-tagged with `SVf_UTF8` without re-allocating.

Differences from upstream

Fully compatible with upstream for ASCII, Latin-1, CP1252, the ISO-8859 family, and UTF-8. Covered by `t/81-xs-native/Encode/010-basic.t` and `t/81-xs-native/Encode/090-decode-inplace.t`.

See also

- `encode` - the inverse, string to bytes.
- `decode_utf8` - the UTF-8-only fast path.
- `from_to` - reencode in place without materialising a character string in between.

native_encode_utf8

Fast path for encoding a string to UTF-8 bytes.

Synopsis

```
my $bytes = encode_utf8($string);
```

Equivalent to `encode('UTF-8', $string)` but skips the encoding registry lookup. Prefer this when you know UTF-8 is what you want.

What you get back

A byte string (no `SVf_UTF8`). If `$string` already had the `SVf_UTF8` flag, the bytes are copied as-is and the flag is cleared on the copy. If `$string` was a raw byte string (Latin-1), its bytes are first upgraded to their UTF-8 multi-byte form.

Examples

```
my $bytes = encode_utf8("caf\x{e9}");  
  
## $bytes is "caf\xc3\xa9"  
  
my $bytes = encode_utf8("\xe9");    # raw Latin-1 input  
  
## $bytes is "\xc3\xa9"
```

Edge cases

- `undef` returns an empty byte string.
- Idempotent only when applied to a character string; applying it twice mojibakes the result.

Differences from upstream

Fully compatible with upstream.

See also

- `decode_utf8` - the inverse.
- `encode` - the general form.
- `is_utf8` - check whether a scalar already carries the flag.

native_decode_utf8

Fast path for decoding UTF-8 bytes to a Perl character string.

Synopsis

```
my $string = decode_utf8($bytes);
my $string = decode_utf8($bytes, $check);
```

Equivalent to `decode('UTF-8', $bytes)` but skips the encoding registry lookup. Input is fully validated as UTF-8; invalid sequences are replaced with U+FFFD unless `$check` asks for a different outcome.

What you get back

A character string with the `SVf_UTF8` flag set. `length` on the result counts characters, not bytes.

Examples

```
my $string = decode_utf8("caf\xc3\xa9");
## length($string) == 4

use Encode qw(decode_utf8 FB_CROAK);
my $string = decode_utf8("\xc3\x28", FB_CROAK);

## dies on the invalid sequence
```

Edge cases

- `undef` returns an empty character string.
- Input that is already valid UTF-8 is copied and flagged; no byte-level conversion happens.

Differences from upstream

Fully compatible with upstream. Covered by `t/81-xs-native/Encode/010-basic.t`.

See also

- `encode_utf8` - the inverse.
- `decode` - the general form.
- `is_utf8` - check whether a scalar already carries the flag.

native_find_encoding

Look up an encoding by name and return an object you can call methods on.

Synopsis

```
my $enc = find_encoding($name);
my $bytes = $enc->encode($string);
my $string = $enc->decode($bytes);
my $canon = $enc->name;
```

What you get back

A blessed object. UTF-8 family encodings return an `Encode::utf8` object; everything else returns an `Encode::XS` object. If the name cannot be resolved, `find_encoding` returns `undef`.

The object exposes four methods: `encode`, `decode`, `name` (returns the canonical string), and `perlio_ok` (true for UTF-8 family, false otherwise). `renew` is also accepted and returns the same object.

Name resolution is case-insensitive and treats `-` and `_` interchangeably. “UTF8”, “utf-8”, and “utf_8” all resolve to the same object.

Examples

Cache an encoding object and reuse it in a loop:

```
my $enc = find_encoding('UTF-8');
while (my $line = <$fh>) {
    my $string = $enc->decode($line);
    # ... work with $string ...
}
```

Query by alias:

```
my $enc = find_encoding('latin1');
print $enc->name;    # iso-8859-1
```

Edge cases

- Unknown names return `undef` after falling back to `Encode::Alias::find_alias` for dynamically registered aliases (e.g. "locale").
- The object is a blessed hashref with a single `Name` key; code that pokes inside works but is not portable across `Encode` implementations.

Differences from upstream

The returned object carries only the `Name` key rather than the full encoding table upstream `Encode::XS` stores. The four documented methods behave identically; direct field access does not. Covered by `t/81-xs-native/Encode/020-find-encoding.t` and `t/81-xs-native/Encode/070-encoding-object.t`.

See also

- `resolve_alias` - get just the canonical name string.
- `encodings` - list every name the registry knows.
- `encode`, `decode` - the function forms, if you don't want to hold an object.

native_is_utf8

Return true if the scalar carries the `SVf_UTF8` flag.

Synopsis

```
my $flagged = is_utf8($string);
my $flagged = is_utf8($string, $check);
```

With one argument, `is_utf8` reports the flag as-is. With a true `$check`, the bytes are also validated as UTF-8; a flagged scalar whose bytes are not valid UTF-8 returns false.

This is a low-level diagnostic. Regular Perl code should not need it - the flag is an internal implementation detail, and the right place to think about "is this text" is at I/O boundaries with `encode/decode`.

Differences from upstream

Fully compatible with upstream. Covered by `t/81-xs-native/Encode/080-utf8-flags.t`.

native_utf8_on

Force `SVf_UTF8` on in place, without touching the underlying bytes.

Returns the previous flag state as a boolean. Use this only when you know the bytes really are well-formed UTF-8 - flipping the flag on garbage produces mojibake that will surface later as double-encoding or Wide character warnings.

The leading underscore in the name is a warning: this is an internals-poking operation, not part of the everyday API.

native_encodings

Return the list of encoding names the registry knows about.

Synopsis

```
my @all = encodings();
my @utf = encodings('utf');      # names starting with "utf"
my @all = encodings(':all');     # same as no argument
```

Names are sorted and deduplicated; each canonical name appears exactly once regardless of how many aliases it has. Use `resolve_alias` or `find_encoding` to turn any of these into a canonical name or an encoding object.

native_from_to

Reencode a byte string in place from one encoding to another.

Synopsis

```
from_to($octets, $from, $to);
from_to($octets, $from, $to, $check);
```

What you get back

The number of characters in the reencoded result, or `undef` if the conversion could not complete. `$octets` itself is modified in place - after the call it holds raw bytes in the `$to` encoding and has no `SVf_UTF8` flag.

Think of `from_to` as a bundled decode / encode pair where the intermediate character string is never exposed to Perl. That is sometimes what you want (less allocation, no flag juggling) and sometimes a trap (no way to intervene between the two halves).

Examples

Convert a Latin-1 HTTP body to UTF-8 before writing:

```
my $body = read_file($path);          # raw bytes
from_to($body, 'iso-8859-1', 'UTF-8');
write_file($out, $body);
```

Unknown encodings croak - same as `encode` / `decode`.

Edge cases

- undef input returns undef and leaves \$octets untouched.
- \$check is forwarded to both the decode and encode halves.

Differences from upstream

Fully compatible with upstream for the registered encodings.

native_obj_encode

Method form of encode on an encoding object.

Synopsis

```
my $enc = find_encoding('UTF-8');
my $bytes = $enc->encode($string);
my $bytes = $enc->encode($string, $check);
```

Behaves like the function form of encode with the encoding name already bound. When \$check is non-zero, the portion of \$string that was successfully consumed is removed from it, mirroring the behaviour upstream documents for \$check != 0.

native_obj_decode

Method form of decode on an encoding object.

Synopsis

```
my $enc = find_encoding('UTF-8');
my $string = $enc->decode($bytes);
my $string = $enc->decode($bytes, $check);
```

Behaves like the function form of decode with the encoding name already bound. When \$check is non-zero, the portion of \$bytes that was successfully consumed is removed from it, so a partial trailing character can be prepended to the next chunk in a streaming decoder.

5.2.9 Errno

Symbolic names for the errno(3) values your system uses - EINTR, EAGAIN, ENOENT, EACCES, EEXIST, EIO, ENOTDIR, EISDIR, EBUSY, EPIPE, ECONNRESET, ECONNREFUSED, ETIMEDOUT, and every other errno name the host libc defines.

Most scripts reach for Errno to compare \$! against a specific error after a failed system call, or to probe %! for the same purpose without touching \$! by name.

Synopsis

```
use Errno qw(EINTR EAGAIN);

if ($! == EINTR) { retry() }      # function form
if ($!{EAGAIN}) { backoff() }    # hash form via %!
```

The numeric values come from the libc the pperl binary was built against (Linux glibc or musl), so scripts see exactly the errno values the kernel will actually raise on this host. The :POSIX tag groups the 74 POSIX-standard names for bulk import.

Functions

Predicate helpers

tiehash

Install the tie behind %! so lookups like \$!{EINTR} start working.

fetch

Read \$!{NAME}: gives the errno value when the last system error matches that name, 0 when the name is a known errno but does not match, and the empty string when the name is not an errno at all.

exists

Answer exists \$!{NAME}: true when the name is a known errno on this host, false otherwise. The current value of \$! is not consulted.

firstkey

Start iteration over %! - the first key that keys %! or each %! returns.

nextkey

Continue iteration over %! - the errno name that follows the previous one in keys %! or each %!, or undef at the end.

store

Refuse any attempt to write, clear, or delete entries in %!. Assigning to \$!{ENOENT}, clearing %!, or deleting a key all raise ERRNO hash is read only!.

tie_it

Internal startup hook that activates the %! tie. Not part of the user-facing API; scripts use %! directly and never call this.

5.2.10 Fcntl

Symbolic constants for `fcntl(2)`, `flock(2)`, `seek(2)`, and file-mode bits - everything you need to call `sysopen`, `fcntl`, `flock`, `seek`, and to interpret the mode word returned by `stat`.

Import individual names, or a named group via one of the export tags: `:DEFAULT` (open and file-descriptor flags), `:flock` (advisory locks), `:seek` (whence positions), `:mode` (mode bits and `S_IS*` predicates), `:Fcompat` (BSD-style `flock` aliases).

Open flags (`sysopen`)

- `O_RDONLY`, `O_WRONLY`, `O_RDWR` - access mode.
- `O_CREAT` - create the file if it does not exist.
- `O_EXCL` - with `O_CREAT`, fail if the file already exists.
- `O_TRUNC` - truncate to zero length on open.
- `O_APPEND` - every write seeks to end of file first.
- `O_NONBLOCK` - open without blocking; I/O returns `EAGAIN` instead.
- `O_NOCTTY`, `O_NOFOLLOW`, `O_BINARY` - additional platform flags (`O_BINARY` is a no-op on Linux).

File-descriptor control (`fcntl`)

- `F_DUPFD` - duplicate a descriptor.
- `F_GETFD`, `F_SETFD` - read / set the close-on-exec flag `FD_CLOEXEC`.
- `F_GETFL`, `F_SETFL` - read / set the open-file status flags.

Lock types (`flock`)

- `LOCK_SH` - shared lock.
- `LOCK_EX` - exclusive lock.
- `LOCK_UN` - release lock.
- `LOCK_NB` - OR with the above to request a non-blocking attempt.

Seek whence

- `SEEK_SET` - from start of file.
- `SEEK_CUR` - from current position.
- `SEEK_END` - from end of file.

File mode bits (`stat`)

- Type mask and values: `S_IFMT`, `S_IFREG`, `S_IFDIR`, `S_IFCHR`, `S_IFBLK`, `S_IFIFO`, `S_IFLNK`, `S_IFSOCK`.

- Permission triplets: `S_IRWXU`, `S_IRWXG`, `S_IRWXO` and the per-bit `S_IRUSR` / `S_IWUSR` / `S_IXUSR` (and the GRP, OTH variants).
- Setuid / setgid / sticky bits: `S_ISUID`, `S_ISGID`, `S_ISVTX`.

Mode test helpers

Function-form predicates over a stat-mode value: `S_ISREG`, `S_ISDIR`, `S_ISCHR`, `S_ISBLK`, `S_ISFIFO`, `S_ISLNK`, `S_ISSOCK`, plus `S_IFMT` (extract file-type bits) and `S_IMODE` (extract permission bits).

Performance

Every constant in this module is resolved at compile time - using `O_CREAT` in an expression costs exactly what writing the integer literal would cost. No per-call lookup.

Functions

Mode test helpers

`s_ifmt`

Extract the file-type bits from a stat-mode value.

`s_imode`

Extract the permission bits from a stat-mode value.

`s_isreg`

True if a stat-mode value names a regular file.

`s_isdir`

True if a stat-mode value names a directory.

`s_ischr`

True if a stat-mode value names a character-special device.

`s_isblk`

True if a stat-mode value names a block-special device.

`s_isfifo`

True if a stat-mode value names a named pipe (FIFO).

s_islnk

True if a stat-mode value names a symbolic link.

s_issock

True if a stat-mode value names a socket.

s_ifmt

Extract the file-type bits from a stat-mode value.

Synopsis

```
my $type = S_IFMT($mode);  
my $mask = S_IFMT();           # the mask constant itself
```

Called with an argument, returns `$mode & S_IFMT` - the bits that classify the file (regular, directory, symlink, ...). Called with no argument, returns the mask constant, which is how Perl code commonly compares against `S_IFREG`, `S_IFDIR`, and friends.

Examples

```
use Fcntl qw(:mode);  
my $mode = (stat $path)[2];  
print "is a directory\n" if S_IFMT($mode) == S_IFDIR;
```

s_ismode

Extract the permission bits from a stat-mode value.

Synopsis

```
my $perm = S_IMODE($mode);
```

Returns `$mode & 07777` - the nine permission bits plus the setuid, setgid, and sticky bits. Strips the file-type bits that `S_IFMT` would keep. With no argument, returns 0.

Examples

```
use Fcntl qw(:mode);  
my $mode = (stat $path)[2];  
printf "perms = %04o\n", S_IMODE($mode);  # perms = 0644
```

s_isreg

True if a stat-mode value names a regular file.

Synopsis

```
my $mode = (stat $path)[2];
print "plain file\n" if S_ISREG($mode);
```

Returns 1 when `$mode & S_IFMT == S_IFREG`, 0 otherwise. With no argument, returns 0.

s_isdir

True if a stat-mode value names a directory.

Synopsis

```
my $mode = (stat $path)[2];
print "directory\n" if S_ISDIR($mode);
```

Returns 1 when `$mode & S_IFMT == S_IFDIR`, 0 otherwise. With no argument, returns 0.

s_ischr

True if a stat-mode value names a character-special device.

Synopsis

```
my $mode = (stat "/dev/tty")[2];
print "char device\n" if S_ISCHR($mode);
```

Returns 1 when `$mode & S_IFMT == S_IFCHR`, 0 otherwise. With no argument, returns 0.

s_isblk

True if a stat-mode value names a block-special device.

Synopsis

```
my $mode = (stat "/dev/sda")[2];
print "block device\n" if S_ISBLK($mode);
```

Returns 1 when `$mode & S_IFMT == S_IFBLK`, 0 otherwise. With no argument, returns 0.

s_isfifo

True if a stat-mode value names a named pipe (FIFO).

Synopsis

```
my $mode = (stat $path)[2];
print "fifo\n" if S_ISFIFO($mode);
```

Returns 1 when `$mode & S_IFMT == S_IFIFO`, 0 otherwise. With no argument, returns 0.

s_islnk

True if a stat-mode value names a symbolic link.

Synopsis

```
my $mode = (lstat $path)[2];
print "symlink\n" if S_ISLNK($mode);
```

Returns 1 when `$mode & S_IFMT == S_IFLNK`, 0 otherwise. With no argument, returns 0. Use `lstat` rather than `stat`: plain `stat` follows symlinks and reports the target's mode.

s_issock

True if a stat-mode value names a socket.

Synopsis

```
my $mode = (stat $path)[2];
print "socket\n" if S_ISSOCK($mode);
```

Returns 1 when `$mode & S_IFMT == S_IFSOCK`, 0 otherwise. With no argument, returns 0.

5.2.11 File

Namespace containing the following modules:

- `File::Glob` — POSIX pathname globbing - expand shell-style wildcard patterns into lists of filenames.
- `File::Map` — Memory-mapped scalars: make a file look like an ordinary string.
- `File::Temp` — Create temporary files and directories with race-safe naming and

File::Glob

POSIX pathname globbing - expand shell-style wildcard patterns into lists of filenames.

`File::Glob` is what backs Perl's built-in glob operator and its angle-bracket form `<*.pl>`. Give it a pattern with shell metacharacters (`*`, `?`, `[...]`, `{a,b,c}`, `~user`) and you get back the list of paths on disk that match. The same machinery is exposed directly as `bsd_glob` when you want explicit control over flags.

The mental model is simple: a pattern goes in, a list of real filenames comes out. Behaviour is tuned by a bitmask of `GLOB_*` flags - tilde expansion, brace expansion, case folding, sort order, and what happens when nothing matches are all flag-controlled. Every flag has a named constant (`GLOB_TILDE`, `GLOB_BRACE`, `GLOB_NOCASE`, `GLOB_NOMAGIC`, `GLOB_CSH`, etc.) and they OR together.

Three call styles, same engine:

```
use File::Glob ':bsd_glob';
my @pl = <*.pl>;           # angle-bracket form
my @c  = bsd_glob('*.ch'); # function form,
                             # default flags
my @hd = bsd_glob('~gnat', GLOB_TILDE | GLOB_ERR);
```

The default flag set is `GLOB_CSH` - brace expansion, tilde expansion, backslash quoting, alphabetical sort, and the `GLOB_NOMAGIC` fallback that returns the literal pattern when it has no metacharacters and doesn't match. Two import tags nudge the default: `:nocase` turns `GLOB_NOCASE` on globally, `:case` turns it off.

Functions

Globbing

`bsd_glob`

Expand a shell-style pattern and return the list of matching paths.

`csch_glob`

Expand a pattern using the default flag set - the engine behind Perl's built-in glob.

`glob_error`

Report the error status of the most recent `bsd_glob` call.

`bsd_glob`

Expand a shell-style pattern and return the list of matching paths.

Synopsis

```
my @list      = bsd_glob('*.ch');
my @homes     = bsd_glob('~gnat', GLOB_TILDE | GLOB_ERR);
my @latest    = bsd_glob('src/*.pl', GLOB_NOSORT);
```

What you get back

A list of filenames, possibly empty. In scalar context you get the count of matches. Filenames are returned as plain strings in the order determined by the flags (alphabetical under the default GLOB_CSH, unsorted under GLOB_NOSORT).

When the pattern matches nothing and neither GLOB_NOCHECK nor GLOB_NOMAGIC is in effect, the list is empty. With GLOB_NOCHECK the original pattern is returned as the sole element; with GLOB_NOMAGIC the pattern is returned only when it contains no metacharacters.

Examples

```
use File::Glob ':bsd_glob';
my @perl = bsd_glob('*.pl');           # ('a.pl', 'b.pl', ...)
```

```
## Brace expansion - default flags include GLOB_BRACE

my @src = bsd_glob('{lib,t}/*.pm');    # files under lib/ and t/
```

```
## Tilde expansion to another user's home

my ($home) = bsd_glob('~gnat', GLOB_TILDE);
```

```
## Case-insensitive match

my @readmes = bsd_glob('readme*', GLOB_NOCASE);
```

Edge cases

- No second argument: flags come from `$File::Glob::DEFAULT_FLAGS` (initialised to GLOB_CSH).
- GLOB_APPEND, GLOB_DOOFFS, GLOB_ALTDIRFUNC, GLOB_MAGCHAR are masked out of the caller's flags: they require C-level state pperl does not expose.
- A pattern containing an embedded NUL byte returns the empty list.
- Missing pattern (zero-argument call) returns the empty list.

Differences from upstream

- `bsd_glob_override` (installed as the caller's `glob` by use `File::Glob ' :bsd_glob'`) is an alias for `bsd_glob` and does not implement the scalar-context iterator state machine upstream provides. In list context the behaviour matches; in scalar context upstream yields one name per call until exhausted, whereas `ppperl` returns the match count. Covered by `t/81-xs-native/File/Glob/*.t`.
- Filenames are not tainted. `ppperl` does not implement taint mode.

See also

- `csch_glob` - the form Perl's built-in `glob` uses internally.
- `GLOB_ERROR` - non-zero after a traversal error.
- `GLOB_CSH` - the default flag bundle.
- `GLOB_NOCHECK` - return the pattern itself when nothing matches.

csch_glob

Expand a pattern using the default flag set - the engine behind Perl's built-in `glob`.

Synopsis

```
use File::Glob qw(csch_glob);
my @files = csch_glob('*.pl');
```

What you get back

A list of matching filenames. Flags always come from `$File::Glob::DEFAULT_FLAGS` (initially `GLOB_CSH`); `csch_glob` takes no flag argument. Sort order, brace expansion, tilde expansion, and the rest follow whatever `DEFAULT_FLAGS` currently holds.

You almost certainly want `bsd_glob` or the angle-bracket operator `<...>` instead. `csch_glob` exists because Perl's built-in `glob` dispatches through it; calling it directly is documented as “don't unless you know what you're doing.”

Examples

```
use File::Glob qw(csch_glob);
my @matches = csch_glob('*.txt');           # uses DEFAULT_FLAGS
```

```
## Flip case sensitivity globally, then call
```

```
use File::Glob qw(csch_glob :nocase);
my @any = csch_glob('ReadMe*');             # matches README, readme, ↵
↵ReadMe, ...
```


Edge cases

- Called with no arguments: returns the empty list.
- Unlike upstream, does not split the pattern on whitespace into multiple sub-patterns; the whole string is treated as one pattern.
- Does not maintain per-op iterator state, so it cannot be driven one-result-at-a-time in scalar context.

Differences from upstream

- No whitespace-splitting of patterns. Upstream `csch_glob` tokenises `"a* b*"` into two patterns; `ppperl` treats it as a single literal pattern. Use `bsd_glob(' {a*,b*}')` if you need multiple patterns in one call.
- No scalar-context iterator. The per-op result cache that lets upstream's `while (my $f = <*.c>)` yield one name per iteration is not implemented - scalar context returns the match count. Covered by `t/81-xs-native/File/Glob/*.t`.

See also

- `bsd_glob` - the public, flag-accepting equivalent; prefer it.
- `GLOB_CSH` - the flag set `DEFAULT_FLAGS` is seeded with.
- `:nocase` / `:case` - import tags that toggle `GLOB_NOCASE` in `DEFAULT_FLAGS`.

glob_error

Report the error status of the most recent `bsd_glob` call.

Synopsis

```
my @files = bsd_glob('~nobody', GLOB_TILDE | GLOB_ERR);
if (GLOB_ERROR) {
    warn "glob failed: $!";
}
```

What you get back

An integer. Zero means the last `bsd_glob` call completed without error. A non-zero value signals that an error interrupted traversal: `GLOB_NOSPACE` for allocation failure, `GLOB_ABEND` for a stopped traversal. When non-zero, `$!` carries the underlying `errno`.

Examples

```
use File::Glob ':bsd_glob';
my @out = bsd_glob('/root/*', GLOB_ERR); # root-only dir
print "error\n" if GLOB_ERROR;           # usually prints under_
↪non-root
```

Edge cases

- Reading GLOB_ERROR without a preceding `bsd_glob` call returns 0.
- Reading it after a successful call returns 0.

Differences from upstream

- Always returns 0. `ppperl` does not thread per-call glob error state through the interpreter, so the value is a stub. Scripts that gate on GLOB_ERROR will always take the no-error branch. Use `$!` directly if you need to detect filesystem errors during globbing. Covered by `t/81-xs-native/File/Glob/*.t`.

See also

- `bsd_glob` - the call whose outcome this function reports.
- GLOB_ERR - the flag that makes `bsd_glob` stop (rather than skip) on unreadable directories.
- GLOB_NOSPACE, GLOB_ABEND - the two non-zero values this would return upstream.

File::Map

Memory-mapped scalars: make a file look like an ordinary string.

`File::Map` attaches a file - or a fresh anonymous region of memory - to a Perl scalar. Reads and writes to the scalar go straight to the kernel's mapping: the file appears in your variable as a string, and changes propagate back without you ever calling `read` or `write`. A multi-gigabyte file costs almost nothing to open this way, because pages are faulted in lazily as you touch them.

Once a scalar is mapped it behaves like a regular string for the operations that matter: `substr`, regex match and substitution, `index`, `length`, and slicing. Direct assignment (`$map = "..."`) is caught and fixed up so you never corrupt the mapping, though `substr` is the recommended way to write. The mapping is released automatically when the scalar goes out of scope; `unmap` is available for explicit control.

Synopsis

```
use File::Map qw(map_file map_anonymous unmap);

map_file my $map, 'data.bin', '+<';    # read/write
$map =~ s/foo/bar/g;                  # edits go to disk
substr $map, 0, 4, "HEAD";             # in-place write

map_anonymous my $buf, 4096, 'shared'; # scratch region
unmap $map;                            # optional
```

Choosing a mapping function

- `map_file` - give a filename and a mode, get a mapped scalar. The common case.
- `map_handle` - same idea, but you already have an open filehandle.
- `map_anonymous` - no file; a scratch region shared with forked children or private to the process.
- `sys_map` - low-level wrapper around raw `mmap(2)` flags for when the convenience functions don't fit.

Managing a live mapping

- `sync` - flush pending writes back to disk.
- `advise` - hint the kernel about access patterns ("sequential", "random", "will-need", ...).
- `protect` - change read/write protection at runtime.
- `remap` - resize an existing mapping (Linux only).
- `unmap` - drop the mapping explicitly.

Functions

Mapping

`map_file`

Open a file by name and attach it to a scalar as a memory map.

`map_handle`

Attach an already-open filehandle to a scalar as a memory map.

`map_anonymous`

Allocate a fresh memory region backed by no file and attach it to a scalar.

`sys_map`

Low-level mapping with raw `mmap(2)` protection and flag constants.

`remap`

Resize an existing mapping in place.

Unmapping

`unmap`

Release a mapping explicitly before the scalar goes out of scope.

Synchronisation

sync

Flush pending writes from a mapped scalar back to disk.

Advice

advise

Tell the kernel how the mapped region will be accessed.

protect

Change the read/write/execute protection of a mapped region.

map_file

Open a file by name and attach it to a scalar as a memory map.

Synopsis

```
map_file my $map, $filename;           # read-only
map_file my $map, $filename, '+<';     # read/write
map_file my $map, $filename, '<', $offset, $length;
```

Arguments

- `$lvalue` - scalar to receive the mapping. Any previous value is discarded. The variable must not already be mapped.
- `$filename` - path to the file. The file is opened with a `:raw` layer added automatically, so encoding and newline translation never interfere with the mapping.
- `$mode` - defaults to `<`. Accepted values are `<`, `+<`, `>`, and `+>`, with the same meaning as `open`.
- `$offset` - byte position at which the mapping starts. Defaults to 0.
- `$length` - length of the mapping in bytes. Defaults to `-s($filename) - $offset`, i.e. the rest of the file.

Examples

Read an entire file into a scalar without loading it eagerly:

```
map_file my $text, 'big.log';
print "lines: ", scalar((() = $text =~ /\n/g), "\n");
```

Patch bytes inside a binary file:

```
map_file my $map, 'image.bin', '+<';
substr $map, 0, 4, "MZ\0\0";
```

Map a window into a large file:

```
map_file my $chunk, 'archive.tar', '<', 1024 * 1024, 4096;
```

Edge cases

- Opening a zero-length file succeeds but produces a mapping that cannot be written to; writes are silently truncated to the empty string.
- \$offset does not need to be page-aligned; the module adjusts internally.
- A filehandle with a non-binary PerlIO layer is rejected.
- Going out of scope unmaps the variable; an explicit unmap is rarely needed.

Differences from upstream

Fully compatible with upstream File::Map 0.71.

See also

- map_handle - same operation starting from an open filehandle.
- map_anonymous - allocate a scratch region with no backing file.
- sys_map - drop down to raw mmap(2) flags.
- unmap - release a mapping explicitly before scope exit.
- sync - flush writes to disk without waiting for scope exit.

map_handle

Attach an already-open filehandle to a scalar as a memory map.

Synopsis

```
open my $fh, '<:raw', $filename;
map_handle my $map, $fh;
map_handle my $map, $fh, '+<', $offset, $length;
```

Arguments

- \$lvalue - scalar to receive the mapping.
- \$filehandle - a real filehandle. Scalar-string handles and tied handles are rejected.
- \$mode - defaults to '<'. Same accepted values as map_file.
- \$offset, \$length - same meaning and defaults as map_file.

Examples

Map the tail of a log file:

```
open my $fh, '<:raw', '/var/log/syslog' or die $!;
my $size = -s $fh;
map_handle my $tail, $fh, '<', $size - 4096, 4096;
```

Edge cases

- The filehandle must be binary. Encoding layers like `:utf8` or `:crlf` cause `map_handle` to reject the handle.
- The handle must refer to a regular file, block device, or character device. Pipes and sockets cannot be mapped.
- The handle stays valid after the call; closing it does not invalidate the mapping.

Differences from upstream

Fully compatible with upstream `File::Map` 0.71.

See also

- `map_file` - take a filename instead of a filehandle.
- `sys_map` - pass raw `mmap(2)` flags directly.
- `unmap` - release the mapping before scope exit.

map_anonymous

Allocate a fresh memory region backed by no file and attach it to a scalar.

Synopsis

```
map_anonymous my $buf, 4096;           # shared (default)
map_anonymous my $buf, 4096, 'shared';  # visible to forked_
↳children
map_anonymous my $buf, 1 << 20, 'private'; # process-local
```

Arguments

- `$lvalue` - scalar to receive the mapping.
- `$length` - size of the region in bytes. Must be greater than zero.
- `$type` - `'shared'` or `'private'`. A shared region is inherited by child processes created with `fork`; a private region is not. Defaults to `'shared'`.

Examples

Share a buffer with a forked child:

```
map_anonymous my $counter, 16, 'shared';
substr $counter, 0, 16, pack('Q', 0) . pack('Q', 0);
if (fork() == 0) {
    substr $counter, 0, 8, pack('Q', 42);
    exit;
}
```

Allocate a private scratch region:

```
map_anonymous my $scratch, 1_000_000, 'private';
substr $scratch, 0, 5, "hello";
```

Edge cases

- Zero-length anonymous maps are rejected.
- Any \$type other than 'shared' or 'private' raises an exception.

Differences from upstream

Fully compatible with upstream File::Map 0.71.

See also

- map_file - map a file instead of raw memory.
- sys_map - take raw mmap(2) flags.
- remap - grow or shrink a private anonymous mapping on Linux.

sys_map

Low-level mapping with raw mmap(2) protection and flag constants.

Synopsis

```
use File::Map qw(sys_map :constants);

sys_map my $map, $length, PROT_READ | PROT_WRITE,
        MAP_SHARED, $fh, 0;

sys_map my $buf, 4096, PROT_READ | PROT_WRITE,
        MAP_PRIVATE | MAP_ANONYMOUS;
```

Arguments

- `$lvalue` - scalar to receive the mapping.
- `$length` - size in bytes.
- `$protection` - bitwise-or of `PROT_NONE`, `PROT_READ`, `PROT_WRITE`, `PROT_EXEC`.
- `$flags` - bitwise-or of `MAP_SHARED`, `MAP_PRIVATE`, `MAP_ANONYMOUS`, `MAP_ANON`, `MAP_FILE`.
- `$filehandle` - omitted or `undef` when `MAP_ANONYMOUS` is set.
- `$offset` - byte offset into the file. Defaults to `0`.

When to reach for it

Use `sys_map` only when `map_file`, `map_handle`, or `map_anonymous` cannot express what you need - for example, to request `PROT_EXEC` on an executable page, or to combine flags that the convenience functions do not expose. If the `mmap(2)` man page does not feel familiar, `map_file` or `map_anonymous` is almost certainly the better choice.

Differences from upstream

Fully compatible with upstream `File::Map` 0.71.

See also

- `map_file`, `map_handle`, `map_anonymous` - the high-level alternatives.
- `protect` - change protection after the mapping exists.

sync

Flush pending writes from a mapped scalar back to disk.

Synopsis

```
sync $map;           # synchronous - wait for the kernel to finish
sync $map, 0;        # asynchronous - queue the flush and return
```

Arguments

- `$lvalue` - a currently mapped scalar.
- `$synchronous` - defaults to `true`. When `true`, `sync` does not return until the kernel reports the data on stable storage. When `false`, the flush is only queued.

The mapping is always flushed automatically when the scalar goes out of scope, so explicit calls to `sync` are only needed when you want other processes, or a crash-safe on-disk snapshot, to see the changes before then.

Edge cases

- Calling `sync` on an unmapped scalar raises an exception.
- Calling `sync` on an empty mapping is a no-op.

Differences from upstream

Fully compatible with upstream `File::Map` 0.71.

See also

- `unmap` - release the mapping (also triggers a flush).
- `advise` - influence the kernel's write-back strategy.

remap

Resize an existing mapping in place.

Synopsis

```
remap $map, $new_size;
```

Arguments

- `$lvalue` - a currently mapped scalar.
- `$new_size` - new length in bytes. Must be greater than zero.

When it applies

`remap` is Linux-specific. It wraps `mremap(2)` with `MREMAP_MAYMOVE`, so the scalar may end up pointing to a different virtual address after the call.

- For a file-backed mapping, the underlying file must be large enough to cover `$new_size`, or accessing the extra pages will produce a bus fault.
- Anonymous shared mappings cannot be remapped.

Use with caution on production data.

Differences from upstream

Fully compatible with upstream `File::Map` 0.71.

See also

- `map_file` - pick the right size up front if you can.
- `unmap` - drop the mapping entirely instead of resizing.

unmap

Release a mapping explicitly before the scalar goes out of scope.

Synopsis

```
unmap $map;
```

When to call it

Most programs never need to. A mapped scalar is unmapped automatically when it is freed - at scope exit for my variables, at the end of the program for globals - and the kernel flushes any pending writes as part of that. Call `unmap` explicitly when you want to release the memory (and close the mapping of the file) before the natural lifetime of the scalar ends.

After `unmap`, the scalar becomes an ordinary empty string and can be reused for any purpose, including being mapped again.

Edge cases

- Calling `unmap` on a scalar that is not mapped raises an exception.

Differences from upstream

Fully compatible with upstream `File::Map` 0.71.

See also

- `sync` - flush writes without releasing the mapping.

advise

Tell the kernel how the mapped region will be accessed.

Synopsis

```
advise $map, 'sequential';  
advise $map, 'willneed';
```

Arguments

- `$lvalue` - a currently mapped scalar.
- `$advice` - a string naming an access pattern.

The portable values are:

- `'normal'` - no particular pattern (the default).
- `'random'` - access order is unpredictable.

- 'sequential' - read start to end once.
- 'willneed' - predefault pages that will be needed soon.
- 'dontneed' - release pages that will not be needed soon.

On Linux additional values are accepted: 'remove', 'dontfork', 'dofork', 'mergeable', 'unmergeable', 'free'. Unknown values are silently ignored.

Examples

Hint that a large log file will be scanned top to bottom:

```
map_file my $log, 'huge.log';
advise $log, 'sequential';
while ($log =~ /^(ERROR:.*)/mg) { ... }
```

Edge cases

- Advising an unmapped scalar raises an exception.
- Advising an empty mapping is a no-op.
- advise is a hint to the kernel; it may do nothing on some systems.

Differences from upstream

Fully compatible with upstream File::Map 0.71.

See also

- protect - change read/write permissions rather than access hints.
- sync - control flush timing explicitly.

protect

Change the read/write/execute protection of a mapped region.

Synopsis

```
protect $map, '<';           # read-only
protect $map, '+<';         # read/write
protect $map, PROT_READ;    # constants also work
```

Arguments

- \$lvalue - a currently mapped scalar.
- \$mode - either a string in the same format as open ('<', '+<', '>', '+>'), or a bitmask built from the PROT_* constants.

If the new protection includes write permission the scalar becomes writable; otherwise it is marked read-only and direct assignment raises an exception.

Examples

Load a file read/write, patch it, then lock it down:

```
map_file my $map, 'data.bin', '+<';
substr $map, 0, 4, "HEAD";
sync $map;
protect $map, '<';           # prevent further writes
```

Edge cases

- Protecting an unmapped scalar raises an exception.
- Protecting an empty mapping updates the scalar's read-only flag but performs no kernel call.

Differences from upstream

Fully compatible with upstream `File::Map` 0.71.

See also

- `sys_map` - set protection when creating the mapping.
- `advise` - hint access patterns rather than enforce permissions.

File::Temp

Create temporary files and directories with race-safe naming and automatic cleanup.

`File::Temp` hands you a fresh, uniquely-named file (or directory) in one step: the name is picked, the file is created, and the filehandle is returned together - so no other process can slip in between naming and opening. On destruction, temporary files and directories are removed automatically unless you ask to keep them.

Two idioms are supported, and they share the same options:

- **Object-oriented** - `File::Temp->new` returns a blessed filehandle object. The object stringifies to its filename and unlinks the file when it goes out of scope. `File::Temp->newdir` is the directory counterpart and cleans up its tree on destruction.
- **Functional** - `tempfile` returns `($fh, $filename)`, and `tempdir` returns a directory path. Both accept the same `DIR`, `SUFFIX`, `TEMPLATE`, `UNLINK`, `CLEANUP`, `PERMS`, and `TMPDIR` options as the OO forms.

Templates use trailing `X` characters (at least four) which are replaced with random name-safe bytes at creation time; a `SUFFIX` appended to the template is preserved across the random fill.

Functions

Temporary files

tempfile

Create a temporary file and return an open filehandle paired with its filename.

new

Construct a `File::Temp` object wrapping a newly created temporary file.

filename

Return the on-disk path associated with a `File::Temp` object.

stringify

Overload handler for `""` - stringify a `File::Temp` object to its filename.

numify

Overload handler for `0+` - numify a `File::Temp` object to its reference address.

tmpnam

POSIX-style `tmpnam` - in scalar context a candidate filename in the system temp directory; in list context a created file with its filehandle.

tmpfile

POSIX-style `tmpfile` - open an anonymous temporary file and return the filehandle. The file is unlinked immediately.

mkstemp

Create a file from a template and return a filehandle together with its pathname.

mkstemp

Create a file from a template with a literal suffix preserved at the end of the name.

Temporary directories

tempdir

Create a uniquely named temporary directory and return its path.

newdir

Construct a `File::Temp::Dir` object wrapping a newly created temporary directory.

mkdtemp

Create a directory from a template and return its path.

dir_dirname

Return the on-disk path for a `File::Temp::Dir` object.

dir_stringify

Overload handler for `"` - stringify a `File::Temp::Dir` to its pathname.

dir_numify

Overload handler for `0+` - numify a `File::Temp::Dir` object to its `refaddr`.

What you get back

The integer `refaddr` of the underlying reference, used for identity comparison via `==`.

Differences from upstream

Fully compatible with upstream `File::Temp` 0.2312.

See also

- `STRINGIFY` - companion overload for string contexts.

Cleanup control

unlink_on_destroy

Toggle or query whether a `File::Temp` object unlinks its file when destroyed.

destroy

Destructor for `File::Temp` objects - removes the underlying file when `UNLINK` is set and `$File::Temp::KEEP_ALL` is false.

cleanup

Remove every temp file and directory currently registered for cleanup, immediately, and return 1.

unlink0

Remove a temporary file while its filehandle is still open.

unlink1

Unlink a temporary file. Upstream closes \$fh first; here the filehandle is left open and closed by its usual refcount path.

safe_level

Report the current safety level used by File::Temp checks.

top_system_uid

Return the highest uid treated as a system account for ancestor ownership checks.

cmpstat

Check whether a filehandle and a pathname refer to the same file - used to guard against race-based substitution attacks.

dir_unlink_on_destroy

Toggle or query whether a File::Temp::Dir object removes its directory tree on destruction.

dir_destroy

Destructor for File::Temp::Dir objects - removes the directory tree when CLEANUP is true, the current pid matches the creator, and \$File::Temp::KEEP_ALL is false.

Template handling

tempnam

Return a candidate filename under \$dir starting with \$prefix - does not create the file.

mktemp

Fill in the trailing Xs of a template and return the resulting filename - does not create the file.

tempfile

Create a temporary file and return an open filehandle paired with its filename.

Synopsis

```
my ($fh, $filename) = tempfile();
my ($fh, $filename) = tempfile($template, DIR => $dir, SUFFIX => '.
    ↳dat');
my $fh                = tempfile();                # scalar context,
    ↳file auto-unlinked
```

What you get back

In list context, a two-element list (`$fh`, `$filename`): `$fh` is an open read/write filehandle on the new file and `$filename` is the path on disk. In scalar context, only the filehandle is returned and the file is unlinked immediately - the fd keeps the storage alive until `$fh` is closed.

Options

- `DIR => $dir` - directory to create the file in.
- `TMPDIR => 1` - place the file under the system temp directory.
- `SUFFIX => $ext` - appended to the template; preserved across random fill.
- `UNLINK => 1` - register the file for deletion at interpreter exit.
- `OPEN => 0` - only pick a name; return (`undef`, `$filename`).
- `PERMS => 0o600` - explicit file permissions (default `0600`).

A leading template argument (odd arg count) is accepted and must end in at least four X characters.

Examples

```
use File::Temp qw/ tempfile /;
my ($fh, $name) = tempfile();
print $fh "hello\n";           # writes to $name
```

```
my ($fh, $name) = tempfile('dataXXXXXX', DIR => '/var/tmp', SUFFIX_
↳=> '.log');

## $name looks like /var/tmp/dataAb3c9F.log
```

```
my ($fh, $name) = tempfile('reportXXXXXX', UNLINK => 1);

## file is removed when the process exits
```

Edge cases

- Called as `File::Temp->tempfile` (method form): croaks.
- Template with fewer than four trailing X characters: croaks.
- Parent directory missing: croaks with a descriptive message.

Differences from upstream

Fully compatible with upstream `File::Temp` 0.2312.

See also

- `tempdir` - directory counterpart with the same option vocabulary.
- `File::Temp->new` - OO form that unlinks on object destruction.
- `mkstemp` - low-level template-only variant without keyword options.
- `tmpfile` - scalar-only form guaranteeing immediate unlink.

tempdir

Create a uniquely named temporary directory and return its path.

Synopsis

```
my $dir = tempdir();
my $dir = tempdir($template, DIR => $parent, CLEANUP => 1);
my $dir = tempdir(CLEANUP => 1);
```

What you get back

A plain string - the full path of the new directory, created with mode `0700`. The directory is not removed automatically unless `CLEANUP => 1` is passed.

Options

- `DIR => $parent` - parent directory for the new temp dir.
- `TMPDIR => 1` - place the directory under the system temp dir.
- `TEMPLATE => $t` - template with at least four trailing X.
- `CLEANUP => 1` - remove the directory (and its contents) at exit.

Examples

```
use File::Temp qw/ tempdir /;
my $dir = tempdir(CLEANUP => 1);
open my $fh, '>', "$dir/notes.txt";

## $dir and its contents are removed at process exit
```

```
my $dir = tempdir('buildXXXXXX', DIR => '/var/tmp');

## directory remains after the process exits (no CLEANUP)
```

Edge cases

- Called as `File::Temp->tempdir` (method form): croaks.
- Parent directory missing or not a directory: croaks.
- Without `CLEANUP`, the directory persists - the caller owns it.

Differences from upstream

Fully compatible with upstream `File::Temp` 0.2312.

See also

- `File::Temp->newdir` - OO form whose object removes the tree on destruction.
- `mkdtemp` - template-only low-level variant.
- `tempfile` - file counterpart with the same option vocabulary.
- `cleanup` - trigger registered cleanups manually.

new

Construct a `File::Temp` object wrapping a newly created temporary file.

Synopsis

```
my $tmp = File::Temp->new;
my $tmp = File::Temp->new(TEMPLATE => 'dataXXXXXX', DIR => '/var/tmp
↪', SUFFIX => '.dat');
```

What you get back

A blessed object that acts as an open read/write filehandle. It stringifies to its filename and numifies to `refaddr`. The underlying file is unlinked when the object is destroyed, unless you opt out with `UNLINK => 0`.

Options

Same as `tempfile`, with two differences: `UNLINK` defaults to true and `OPEN` is always on. Pass `TEMPLATE => 'nameXXXXXX'` to control the filename pattern.

Examples

```
use File::Temp;
my $tmp = File::Temp->new(SUFFIX => '.json');
print $tmp qq({"ok":1}\n);
my $path = $tmp->filename;           # /tmp/XXXXXXX.json
```

```
my $tmp = File::Temp->new(UNLINK => 0);    # keep the file after_
    ↪exit
```

Edge cases

- OPEN => 0 is silently ignored; the file is always opened.
- Object destruction while \$File::Temp::KEEP_ALL is true skips unlink.

Differences from upstream

Fully compatible with upstream File::Temp 0.2312.

See also

- tempfile - functional counterpart returning (\$fh, \$filename).
- File::Temp->newdir - directory constructor.
- filename - retrieve the pathname without stringifying.
- unlink_on_destroy - toggle the unlink-on-destruction flag.

newdir

Construct a File::Temp::Dir object wrapping a newly created temporary directory.

Synopsis

```
my $dir = File::Temp->newdir;
my $dir = File::Temp->newdir('buildXXXXXX', DIR => '/var/tmp',
    ↪CLEANUP => 0);
```

What you get back

A blessed File::Temp::Dir object. It stringifies to the directory path, and its destructor removes the directory tree when CLEANUP is true (the default) and the current process matches the creating pid.

Options

Same as tempdir except CLEANUP defaults to true. The object records the creator pid so child processes do not accidentally delete a parent's directory on exit.

Examples

```
my $dir = File::Temp->newdir;
open my $fh, '>', "$dir/scratch";           # $dir stringifies

## $dir and its contents are removed when $dir goes out of scope
```

```
my $dir = File::Temp->newdir(CLEANUP => 0); # keep the tree
```

Edge cases

- Forked children do not clean up the parent's directory.
- `$File::Temp::KEEP_ALL = 1` suppresses removal on destruction.

Differences from upstream

Fully compatible with upstream `File::Temp` 0.2312.

See also

- `tempdir` - functional counterpart returning a plain path string.
- `dirname` - method on the returned object returning the path.
- `unlink_on_destroy` - toggle cleanup on a per-object basis.
- `File::Temp->new` - file counterpart.

filename

Return the on-disk path associated with a `File::Temp` object.

Synopsis

```
my $path = $tmp->filename;
```

What you get back

The filename string the object was opened against, or `undef` if the argument is not a recognised `File::Temp` handle.

Examples

```
my $tmp = File::Temp->new;  
print "writing to ", $tmp->filename, "\n";
```

Differences from upstream

Fully compatible with upstream `File::Temp` 0.2312.

See also

- `File::Temp->new` - constructs the objects this method queries.
- `dirname` - the analogous accessor on `File::Temp::Dir`.

stringify

Overload handler for "" - stringify a File::Temp object to its filename.

Synopsis

```
my $tmp = File::Temp->new;  
print "wrote to $tmp\n";           # calls STRINGIFY
```

What you get back

The object's pathname as a string, or the empty string if the filename is not known.

Differences from upstream

Fully compatible with upstream File::Temp 0.2312.

See also

- filename - explicit accessor with the same result.
- NUMIFY - companion overload used for numeric comparisons.

numify

Overload handler for 0+ - numify a File::Temp object to its reference address.

Synopsis

```
my $tmp = File::Temp->new;  
print 0 + $tmp, "\n";           # the refaddr  
$tmp == $other_handle          # compares by identity
```

What you get back

The integer refaddr of the underlying reference - the same value Scalar::Util::refaddr would return.

Differences from upstream

Fully compatible with upstream File::Temp 0.2312.

See also

- STRINGIFY - companion overload used for string contexts.

unlink_on_destroy

Toggle or query whether a `File::Temp` object unlinks its file when destroyed.

Synopsis

```
my $flag = $tmp->unlink_on_destroy;      # getter
$tmp->unlink_on_destroy(0);               # keep the file
$tmp->unlink_on_destroy(1);               # remove on destruction
```

What you get back

The new (or current) flag value as an integer: 1 if the file will be removed when the object is destroyed, 0 otherwise.

Examples

```
my $tmp = File::Temp->new;
$tmp->unlink_on_destroy(0);

## $tmp->filename survives after $tmp goes out of scope
```

Edge cases

- Called on something that is not a `File::Temp` object: returns 1 without touching state.
- Setting to a true value on an object whose file is no longer tracked does not re-register it.

Differences from upstream

- The flag is stored in an internal map rather than the blessed glob's hash slot. Code that pokes at `%{*fh}` directly will not observe the flag; use the method. Covered by `t/81-xs-native/File-Temp/*.t`.

See also

- `File::Temp->new` - constructor whose default is `UNLINK => 1`.
- `cleanup` - force-drain all registered cleanups now.
- `unlink0` - verified unlink used internally by scalar-context tempfile.

destroy

Destructor for `File::Temp` objects - removes the underlying file when `UNLINK` is set and `$File::Temp::KEEP_ALL` is false.

Synopsis

```
{
    my $tmp = File::Temp->new; # UNLINK => 1 by default
} # DESTROY runs here; file is unlinked
```

What you get back

Nothing - this is a destructor invoked by the runtime when the last reference drops. It is rare to call it directly.

Edge cases

- `$File::Temp::KEEP_ALL = 1` suppresses all file removal.
- Files not registered with cleanup tracking are left on disk.

Differences from upstream

Fully compatible with upstream `File::Temp` 0.2312.

See also

- `unlink_on_destroy` - per-object opt-out for the removal behavior.
- `cleanup` - explicit cleanup of all registered temp entries.

tmpnam

POSIX-style `tmpnam` - in scalar context a candidate filename in the system temp directory; in list context a created file with its filehandle.

Synopsis

```
my $name      = tmpnam();           # scalar: just a name
my ($fh, $name) = tmpnam();         # list: creates the file
```

What you get back

In scalar context a filename string in the system temp directory that does not exist at the moment of the call - there is still a race between picking it and opening it, so prefer `tempfile`. In list context, a freshly created file is returned along with its filehandle.

Examples

```
use File::Temp qw/ tmpnam /;
my ($fh, $name) = tmpnam();           # race-free file
    ↪ creation
```

Edge cases

- Scalar context does not create a file - callers must open the name themselves and accept the race.

Differences from upstream

Fully compatible with upstream `File::Temp` 0.2312.

See also

- `tempfile` - the recommended race-safe replacement.
- `tmpfile` - file is opened and unlinked immediately.
- `mkstemp` - template-based creation.

tmpfile

POSIX-style `tmpfile` - open an anonymous temporary file and return the filehandle. The file is unlinked immediately.

Synopsis

```
my $fh = tmpfile();
print $fh "scratch\n";
seek $fh, 0, 0;
```

What you get back

An open read/write filehandle reference. There is no filename to hand out: the file is removed from the directory immediately after opening and exists only as long as `$fh` is open.

Examples

```
my $fh = tmpfile();
print $fh "line 1\n";
seek $fh, 0, 0;
my $line = <$fh>                                # "line 1\n"
```

Edge cases

- On filesystems that refuse to unlink open files the file stays on disk until `$fh` is closed.

Differences from upstream

Fully compatible with upstream `File::Temp` 0.2312.

See also

- `tempfile` - gives you both filehandle and filename.
- `tmpnam` - returns only a name (or `($fh, $name)` in list context).

tempnam

Return a candidate filename under `$dir` starting with `$prefix` - does not create the file.

Synopsis

```
my $name = tempnam('/var/tmp', 'cache_');
```

What you get back

A full path string `"$dir/${prefix}XXXXXXXX"` with the Xs replaced by random characters. The caller is responsible for opening the file, and there is a race between name selection and open.

Examples

```
use File::Temp qw/ tempnam /;
my $name = tempnam('/tmp', 'sess-');
open my $fh, '>', $name or die;
```

Edge cases

- Wrong arg count: croaks with a usage message.
- Like `mktemp`, unsafe against attackers - prefer `tempfile`.

Differences from upstream

Fully compatible with upstream `File::Temp` 0.2312.

See also

- `tempfile` - race-safe replacement returning both filehandle and name.
- `mktemp` - similar idea with a caller-supplied template.

mktemp

Fill in the trailing Xs of a template and return the resulting filename - does not create the file.

Synopsis

```
my $name = mktemp('/tmp/dataXXXXXX');
```

What you get back

A candidate filename string with random characters substituted for the trailing Xs. No file is created; the caller must open it themselves and accept the race between naming and opening.

Examples

```
use File::Temp qw/ mktemp /;
my $name = mktemp('/var/tmp/snapXXXXXXXX');
```

Edge cases

- Template with fewer than four trailing Xs: croaks.
- Wrong arg count: croaks with a usage message.
- Unsafe against racing attackers - prefer tempfile.

Differences from upstream

Fully compatible with upstream File::Temp 0.2312.

See also

- tempfile - race-safe replacement.
- mkstemp - same idea, but also creates and opens the file.
- tempnam - dir + prefix variant.

mkstemp

Create a file from a template and return a filehandle together with its pathname.

Synopsis

```
my ($fh, $path) = mkstemp('/tmp/dataXXXXXX');
my $fh          = mkstemp('/tmp/dataXXXXXX');    # scalar context
```

What you get back

In list context (`$fh`, `$path`); in scalar context just the filehandle. The template must end in at least four `X` characters, which are replaced atomically with random bytes.

Examples

```
use File::Temp qw/ mkstemp /;
my ($fh, $path) = mkstemp('/var/tmp/workXXXXXX');
print $fh "payload\n";
```

Edge cases

- Template with fewer than four trailing `X`s: croaks.
- Wrong arg count: croaks with a usage message.

Differences from upstream

Fully compatible with upstream `File::Temp` 0.2312.

See also

- `tempfile` - higher-level variant accepting `DIR`, `SUFFIX`, `UNLINK`.
- `mkstemp`s - like `mkstemp` but keeps a literal suffix on the name.
- `mkdtemp` - directory counterpart.

mkstemp

Create a file from a template with a literal suffix preserved at the end of the name.

Synopsis

```
my ($fh, $path) = mkstemp('/tmp/dataXXXXXX', '.log');
## $path looks like /tmp/dataAb3c9F.log
```

What you get back

In list context (`$fh`, `$path`); in scalar context just the filehandle. The suffix is appended to the template before filling, and the `X`s before it are the ones replaced.

Examples

```
use File::Temp qw/ mkstemp /;
my ($fh, $path) = mkstemp('/var/tmp/rpt-XXXXXX', '.json');
```

Edge cases

- The suffix is a literal string, not a length count.
- Template with fewer than four trailing Xs before the suffix: croaks.
- Wrong arg count: croaks with a usage message.

Differences from upstream

Fully compatible with upstream `File::Temp` 0.2312.

See also

- `mkstemp` - same idea, no suffix handling.
- `tempfile` - use `SUFFIX => ...` for a named-option form.

`mkdtemp`

Create a directory from a template and return its path.

Synopsis

```
my $dir = mkdtemp('/tmp/buildXXXXXX');
```

What you get back

The pathname of the directory created with mode `0700`. The directory is not registered for cleanup - it remains until the caller removes it or arranges for cleanup themselves.

Examples

```
use File::Temp qw/ mkdtemp /;
my $dir = mkdtemp('/var/tmp/stageXXXXXX');

## caller owns $dir; remove with rmtree / File::Path
```

Edge cases

- Template with fewer than four trailing Xs: croaks.
- Wrong arg count: croaks with a usage message.

Differences from upstream

Fully compatible with upstream `File::Temp` 0.2312.

See also

- `tempdir` - higher-level variant accepting `CLEANUP => 1`.
- `File::Temp->newdir` - OO form that removes the tree automatically.
- `mkstemp` - file counterpart.

cleanup

Remove every temp file and directory currently registered for cleanup, immediately, and return 1.

Synopsis

```
File::Temp::cleanup();
```

What you get back

The integer 1. The call drains the internal cleanup registry, so anything tagged `UNLINK => 1` or `CLEANUP => 1` is removed in the reverse order it was registered.

Examples

```
use File::Temp;
my $dir = File::Temp::tempdir(CLEANUP => 1);

## ...

File::Temp::cleanup();           # remove $dir now
```

Edge cases

- An implicit call is made automatically at interpreter exit via the `END` queue, so manual invocation is rarely needed.
- Entries not flagged for cleanup are skipped.

Differences from upstream

Fully compatible with upstream `File::Temp` 0.2312.

See also

- `unlink_on_destroy` - toggle cleanup tracking on a single object.
- `unlink0` / `unlink1` - targeted file removal used by tempfile internals.

unlink0

Remove a temporary file while its filehandle is still open.

Synopsis

```
unlink0($fh, $path) or die "unlink0: $!";
```

What you get back

Integer 1 on success. The directory entry is removed; reads and writes on \$fh continue to work, and the blocks are released when \$fh is finally closed.

Examples

```
use File::Temp qw/ tempfile unlink0 /;
my ($fh, $path) = tempfile();
unlink0($fh, $path);

## $path is gone, but $fh is still usable
```

Edge cases

- If \$File::Temp::KEEP_ALL is true the function returns 1 without unlinking.
- Fewer than two arguments: no-op, returns 1.

Differences from upstream

- Upstream performs a full stat/fstat comparison before unlinking to detect races and symlink tricks. This implementation unlinks by path without that check. Callers who need the stat-equality guarantee should validate the handle themselves. Covered by t/81-xs-native/File-Temp/*.t.

See also

- unlink1 - close-then-unlink variant.
- cmpstat - the stat comparison used by upstream unlink0.
- tempfile - uses this internally in scalar context.

unlink1

Unlink a temporary file. Upstream closes \$fh first; here the filehandle is left open and closed by its usual refcount path.

Synopsis

```
unlink1($fh, $path) or die "unlink1: $!";
```

What you get back

Integer 1 on success.

Examples

```
use File::Temp qw/ tempfile unlink1 /;
my ($fh, $path) = tempfile();

## ... use $fh ...

unlink1($fh, $path);
```

Edge cases

- `$File::Temp::KEEP_ALL = 1` suppresses removal.
- Fewer than two arguments: no-op, returns 1.

Differences from upstream

- Upstream closes `$fh` explicitly before unlinking. This implementation relies on perl5 refcount-driven close instead, so callers wanting an immediate close should do it themselves before the call. Covered by `t/81-xs-native/File-Temp/*.t`.

See also

- `unlink0` - unlink-while-open variant.
- `cleanup` - bulk removal of registered entries.

safe_level

Report the current safety level used by `File::Temp` checks.

Synopsis

```
my $level = File::Temp::safe_level();
File::Temp->safe_level(File::Temp::STANDARD);
```

What you get back

The integer `0`, which corresponds to `STANDARD`. The setter form is accepted but has no effect.

Differences from upstream

- Upstream supports STANDARD, MEDIUM, and HIGH; the latter two add sticky-bit and ancestor-ownership checks that are not implemented here. Requests to raise the level are silently clamped to STANDARD. Covered by `t/81-xs-native/File-Temp/*.t`.

See also

- `top_system_uid` - related stub used by the HIGH-level checks.
- `cmpstat` - stat comparison performed independently of `safe_level`.

`top_system_uid`

Return the highest uid treated as a system account for ancestor ownership checks.

Synopsis

```
my $uid = File::Temp::top_system_uid();
```

What you get back

The integer 0 (root).

Differences from upstream

- Upstream defaults this value to 10; pperl returns 0 because the elevated safety levels that consume it are not implemented. Covered by `t/81-xs-native/File-Temp/*.t`.

See also

- `safe_level` - the setting this value participates in.

`cmpstat`

Check whether a filehandle and a pathname refer to the same file - used to guard against race-based substitution attacks.

Synopsis

```
cmpstat($fh, $path) or die "fh/path diverged";
```

What you get back

A truthy value if the check passes, 0 otherwise. A passing result is a necessary precondition for safe `unlink0`-style operations.

Differences from upstream

- Upstream compares every field of `stat` and `fstat`. This implementation only verifies that `$path` exists as a regular file; the detailed comparison is not performed. Callers that rely on it for security should add their own verification. Covered by `t/81-xs-native/File-Temp/*.t`.

See also

- `unlink0` - wraps `cmpstat` in the full `unlink-while-open` flow.
- `safe_level` - controls how aggressively these checks are applied.

`dir_dirname`

Return the on-disk path for a `File::Temp::Dir` object.

Synopsis

```
my $path = $dir->dirname;
```

What you get back

The directory pathname as a string, or `undef` if called on something that is not a `File::Temp::Dir`.

Examples

```
my $dir = File::Temp->newdir;  
my $p   = $dir->dirname;           # same as "$dir"
```

Differences from upstream

Fully compatible with upstream `File::Temp` 0.2312.

See also

- `File::Temp->newdir` - constructor that returns objects answering this method.
- `filename` - the analogous accessor on `File::Temp`.

`dir_stringify`

Overload handler for `" "` - stringify a `File::Temp::Dir` to its pathname.

Synopsis

```
my $dir = File::Temp->newdir;  
print "dir is $dir\n";           # calls STRINGIFY
```

What you get back

The directory pathname, identical to `$dir->dirname`.

Differences from upstream

Fully compatible with upstream `File::Temp` 0.2312.

See also

- `dirname` - explicit accessor with the same result.

`dir_unlink_on_destroy`

Toggle or query whether a `File::Temp::Dir` object removes its directory tree on destruction.

Synopsis

```
my $flag = $dir->unlink_on_destroy;    # getter  
$dir->unlink_on_destroy(0);           # keep the tree
```

What you get back

The new (or current) cleanup flag as an integer.

Examples

```
my $dir = File::Temp->newdir;  
$dir->unlink_on_destroy(0);  
  
## $dir's path survives after $dir goes out of scope
```

Edge cases

- Called on something that is not a `File::Temp::Dir`: returns `0`.
- Writes to the object's `CLEANUP` hash slot; reads by other code see the updated value.

Differences from upstream

Fully compatible with upstream `File::Temp` 0.2312.

See also

- `File::Temp->newdir` - constructor whose default is `CLEANUP => 1`.
- `cleanup` - force-drain all registered cleanups now.

`dir_destroy`

Destructor for `File::Temp::Dir` objects - removes the directory tree when `CLEANUP` is true, the current pid matches the creator, and `$File::Temp::KEEP_ALL` is false.

Synopsis

```
{  
    my $dir = File::Temp->newdir;    # CLEANUP => 1 by default  
} # DESTROY runs here; tree is removed
```

What you get back

Nothing - invoked by the runtime when the last reference drops.

Edge cases

- A child process that inherits the object does not remove the directory on exit; only the creator pid does.
- `$File::Temp::KEEP_ALL = 1` suppresses removal.

Differences from upstream

Fully compatible with upstream `File::Temp` 0.2312.

See also

- `unlink_on_destroy` - per-object opt-out for the removal behavior.
- `cleanup` - explicit cleanup of all registered temp entries.

5.2.12 Filter

Namespace containing the following modules:

- `Filter::Util`

Filter::Util

Namespace containing the following modules:

- `Filter::Util::Call` — Framework for writing Perl *source filters* - modules that rewrite program

Filter::Util::Call

Framework for writing Perl *source filters* - modules that rewrite program text on the fly as the tokenizer reads it.

A source filter is a normal Perl module that registers a callback with the compiler. Each time the tokenizer needs more source, it calls back into the filter, which pulls raw text from the next filter in the chain (or the file), mangles it, and hands the result back. The compiler never sees the original bytes - it sees whatever the filter produces. This is the mechanism behind `Switch`, `Filter::Simple`, `PDL::NiceSlice`, `Smart::Comments`, and similar compile-time rewriters.

The filter lifecycle

A filter goes through three phases. Each phase corresponds to one of the primary functions exported by this module:

- **Install** - the filter's `import` is called when Perl processes use `MyFilter`. `import` calls `filter_add` to register itself with the compiler. From this point on the compiler will route every read through the filter.
- **Read** - each time the compiler wants more source, it invokes the filter's `filter` method (or anonymous sub). The filter calls `filter_read` (line mode) or `filter_read_exact` (block mode) to pull raw text from the chain below, transforms it in `$_`, and returns a status: positive for "more data", `0` for EOF, negative for error.
- **Remove** - when the filter decides it is done (for example on seeing an end-marker in the source), it calls `filter_del`. The compiler then reads directly from whatever filter - or file - sat below it in the chain.

The two filter shapes

A filter is either a **method filter** (a blessed object whose class provides a `filter` method) or a **closure filter** (an anonymous sub passed directly to `filter_add`). The two shapes carry context differently - method filters stash state on the object, closure filters close over lexicals - but the read/return contract is identical in both cases.

Shared conventions

- `$_` is the filter's scratch buffer. It is cleared before the filter's `filter` is called; `filter_read` appends to it; the transformed contents of `$_` on return are what the compiler sees.
- The status code is three-valued: `> 0` success, `= 0` EOF, `< 0` error. Check it after every read.

- Filters only run at compile time. Calling `filter_add` outside a use-triggered import (for example at runtime) does nothing useful, because there is no active parser to register with.

Skeleton

```
package MyFilter;
use Filter::Util::Call;

sub import {
    filter_add(bless []);      # method filter
}

sub filter {
    my ($self) = @_;
    my $status = filter_read();
    s/old/new/g if $status > 0; # rewrite $_
    $status;
}
1;
```

Functions

Filter lifecycle

`filter_del`

Remove the calling filter from the active chain. Subsequent reads bypass it and hit the next filter down (or the source file).

`filter_add`

Install a source filter for the current package. Called from the filter module's import.

Input handling

`filter_read`

Pull the next line of source - or up to a given number of bytes - from the filter chain into `$_`.

`filter_read_exact`

Pull exactly `$size` bytes from the filter chain into `$_`, looping over short reads until the request is satisfied or the stream ends.

Internal dispatch

`real_import`

Low-level installer that wires a pre-blessed object or code ref into the tokenizer. Called by `filter_add` after it has determined the filter shape; not intended for direct user calls.

`filter_read`

Pull the next line of source - or up to a given number of bytes - from the filter chain into `$_`.

Synopsis

```
my $status = filter_read();      # line mode
my $status = filter_read($size); # block mode, up to $size bytes
```

What you get back

An integer status, three-valued:

- `> 0` - more source was appended to `$_`.
- `= 0` - EOF: no more source is available from anything below this filter in the chain.
- `< 0` - a read error occurred.

The data is appended to `$_`, not returned. The caller's filter body then transforms `$_` and hands it back to the compiler by returning the same status.

In block mode, `filter_read($size)` may return fewer than `$size` bytes - short reads are normal. Use `filter_read_exact` when the filter needs a guaranteed block size.

Examples

A minimal filter that rewrites Joe to Jim line by line:

```
sub filter {
    my $status = filter_read();
    s/Joe/Jim/g if $status > 0;
    $status;
}
```

Block mode, processing a chunk at a time:

```
sub filter {
    my $status = filter_read(4096);
    s/secret/*****/g if $status > 0;
    $status;
}
```

Edge cases

- Calling `filter_read` from outside a filter's body (for example at module top level) has no useful effect - no parser is active to read from.
- On EOF, `$_` may still contain buffered data from an earlier partial read; the caller decides whether to flush it.

Differences from upstream

Fully compatible with upstream `Filter::Util::Call` 1.65.

See also

- `filter_read_exact` - same, but insists on a full block.
- `filter_add` - installs the filter that `filter_read` reads under.
- `filter_del` - stops the current filter so subsequent reads go straight to the chain below.

`filter_read_exact`

Pull exactly `$size` bytes from the filter chain into `$_`, looping over short reads until the request is satisfied or the stream ends.

Synopsis

```
my $status = filter_read_exact($size);
```

What you get back

An integer status using the same convention as `filter_read`:

- `> 0` - a full `$size`-byte block was read into `$_`.
- `= 0` - EOF reached with no pending data, or EOF reached with a partial block (the partial block is still in `$_`, and the status returned to the caller is 1 in that specific case to signal “there is data to process”). See the note below.
- `< 0` - a read error occurred.

`filter_read_exact` keeps calling `filter_read` internally until it has accumulated `$size` bytes or the chain reports EOF / error. It is the right choice for filters that operate on fixed-size frames (for example, a binary decoder).

Examples

Read 16-byte records until end of file:

```
sub filter {  
    my $status = filter_read_exact(16);
```

(continues on next page)

(continued from previous page)

```
decode_record($_) if $status > 0;
$status;
}
```

Edge cases

- `$size` must be strictly positive. Calling `filter_read_exact(0)` or with a negative argument croaks with "filter_read_exact: size parameter must be > 0".
- On EOF with a partially-filled `$_`, the returned status is 1, not 0 - this lets the filter process the tail of the stream before the next call reports true EOF.

Differences from upstream

Fully compatible with upstream `Filter::Util::Call` 1.65.

See also

- `filter_read` - the short-read variant; cheaper when the filter can tolerate partial reads.
- `filter_add` - every filter that calls `filter_read_exact` must first register itself with `filter_add`.

filter_del

Remove the calling filter from the active chain. Subsequent reads bypass it and hit the next filter down (or the source file).

Synopsis

```
filter_del();
```

What you get back

Nothing useful - `filter_del` is a statement, not an expression.

It does not stop the current filter invocation: the caller finishes its current call and returns a status to the compiler normally. Starting with the *next* read, the compiler skips this filter and reads from whatever sat below it in the chain.

Examples

Disable the filter as soon as a sentinel line is seen, so the compiler processes the rest of the file untouched:


```

sub filter {
    my $status = filter_read();
    if ($status > 0 && /^__END_FILTER__$/) {
        filter_del();
        $_ = '';      # drop the sentinel line itself
    }
    $status;
}

```

Edge cases

- Calling `filter_del` when no filter is active is a no-op.
- `filter_del` marks the filter inactive but does not free its per-filter state; the tokenizer releases that state when it tears down the parser at end of compilation.

Differences from upstream

Fully compatible with upstream `Filter::Util::Call 1.65`.

See also

- `filter_add` - installs the filter that `filter_del` later removes.
- `filter_read` - normally called right before `filter_del` in the sentinel branch.

real_import

Low-level installer that wires a pre-blessed object or code ref into the tokenizer. Called by `filter_add` after it has determined the filter shape; not intended for direct user calls.

Synopsis

```

Filter::Util::Call::real_import($object_or_coderef, $package, $is_
    ↪coderef);

```

What you get back

Nothing. `real_import` installs the filter as a side effect: the tokenizer now routes every read through it until `filter_del` is called or compilation of the enclosing file finishes.

The three arguments carry the state that the tokenizer will hand back to every filter invocation:

- `$object_or_coderef` - the blessed filter object (method filter) or the anonymous sub (closure filter).
- `$package` - the module name, used in diagnostic messages such as "Filter::Util::Call - MyFilter::filter returned N values, 1 was expected".

- `$is_coderef` - 1 if the first argument is a code ref (the closure-filter shape), 0 if it is a blessed object (the method-filter shape).

Edge cases

- If `real_import` runs outside the compiler (for example at runtime), registration silently succeeds but the filter is never invoked - there is no parser to drive it.
- The module-name argument is copied into a C-string buffer that lives for the lifetime of the process; filter registrations are intentionally not reclaimed.

Differences from upstream

Fully compatible with upstream `Filter::Util::Call` 1.65.

See also

- `filter_add` - the high-level entry point that prepares arguments and calls `real_import`.
- `unimport` - the inverse operation, exposed for no `MyFilter` idioms.

`filter_add`

Install a source filter for the current package. Called from the filter module's `import`.

Synopsis

```
filter_add($object);      # method filter: $object is blessed state
filter_add(sub { ... }); # closure filter: anonymous sub
```

What you get back

Nothing. `filter_add` registers the filter as a side effect; the compiler now routes every subsequent read through it until the filter calls `filter_del` or compilation of the enclosing file finishes.

The shape of the argument picks the filter style:

- A CODE reference - closure filter. The sub is called for every compiler read; state lives in lexicals captured by the closure.
- Anything else - method filter. If the argument is a plain scalar or an unblessed ARRAY/HASH reference, `filter_add` blesses a reference to it into the caller's package, so the caller's filter method receives it as `$self`. An already-blessed reference is used as-is.

Examples

A closure filter installed from import:

```

sub import {
    filter_add(sub {
        my $status = filter_read();
        tr/a-z/A-Z/ if $status > 0;
        $status;
    });
}

```

A method filter that carries a substitution count as its state:

```

sub import {
    my $count = 0;
    filter_add(\$count);      # becomes $self in filter()
}
sub filter {
    my ($self) = @_;
    my $status = filter_read();
    $$self++ if s/Joe/Jim/g && $status > 0;
    $status;
}

```

Edge cases

- `filter_add` looks up the caller's package through `caller`, so it must be called from the filter module's `import` (or a sub called directly by it). Invoking it from a deeply nested helper will bless the state object into the wrong package.
- Source filters only act during compilation. Calling `filter_add` at runtime (outside any `use/BEGIN` chain) has no effect.
- Passing neither a blessed reference nor a code reference is legal: `filter_add` wraps and blesses the value for you.

Differences from upstream

Fully compatible with upstream `Filter::Util::Call` 1.65.

See also

- `filter_read` - the read primitive every filter body calls.
- `filter_del` - stop the filter from inside its own body.
- `Filter::Simple` - a higher-level wrapper around this module, preferred for new filters that do not need the fine-grained read control.

5.2.13 Hash

Namespace containing the following modules:

- `Hash::Util` — General-utility hash subroutines - lock and unlock keys or values, inspect bucket layout, and seed the hash function.

Hash::Util

General-utility hash subroutines - lock and unlock keys or values, inspect bucket layout, and seed the hash function.

A *restricted* hash is an ordinary hash that rejects new keys and, optionally, refuses writes to its values. You build one by calling `lock_keys` on a populated hash; from that point on a write to any key not already present croaks. Locking individual values with `lock_value`, or the whole hash with `lock_hash`, makes the existing values immutable as well. Every lock has a matching unlock.

Alongside the locking primitives the module exposes a handful of introspection helpers: `bucket_ratio`, `num_buckets`, `used_buckets`, `bucket_info`, `bucket_array`, and the summary reports `bucket_stats` and `bucket_stats_formatted`. They report how the hash is laid out internally and are useful when tuning the size of very large hashes.

`hash_seed` and `hash_value` expose the hash function itself - the random per-process seed, and the integer hash of a given string under that seed (or a caller-supplied seed). Both are sensitive: handing them to untrusted code enables algorithmic-complexity attacks against any hash in the program.

Every ref-suffixed variant (`lock_ref_keys`, `lock_hashref`, `hashref_locked`, `legal_ref_keys`, `hidden_ref_keys`, etc.) does the same work as its plain counterpart but takes an already-taken hash reference rather than relying on the `%` prototype.

Synopsis

```
use Hash::Util qw(lock_keys unlock_keys lock_value hash_locked_
↳hidden_keys);

my %h = (foo => 1, bar => 2);
lock_keys(%h);                # fix the keyset
${foo} = 10;                  # ok
eval { ${baz} = 99 };          # croaks - attempt to access_
↳disallowed key

lock_value(%h, 'foo');         # make ${foo} immutable too
print hash_locked(%h) ? "yes" : "no"; # yes

unlock_keys(%h);              # lift the keyset restriction
```

Security

hash_seed, hash_value, and bucket_array reveal enough about the hash function to enable collision attacks. Do not expose their output to code you do not trust.

Functions

Hash locking

lock_keys

Restrict a hash to its current keys, or to an explicit list.

unlock_keys

Lift the keyset restriction imposed by lock_keys.

lock_ref_keys

Hash-reference variant of lock_keys.

unlock_ref_keys

Hash-reference variant of unlock_keys.

Synopsis

```
unlock_ref_keys($href);
```

Lifts the keyset restriction on the referenced hash. See unlock_keys for the full story on placeholders and values that remain locked.

lock_ref_keys_plus

Lock the hash to the union of its current keys and a given extra list.

lock_value

Make the value under a given key immutable.

unlock_value

Undo a previous lock_value, making the value at \$key writable again.

Synopsis

```
unlock_value(%hash, $key);
```

Does nothing (silently) if the key is absent or was never locked. Returns a reference to the hash.

lock_ref_value

Hash-reference variant of lock_value.

Synopsis

```
lock_ref_value($href, $key);
```

Same behaviour as lock_value but takes a hash reference.

unlock_ref_value

Hash-reference variant of unlock_value.

Synopsis

```
unlock_ref_value($href, $key);
```

Same behaviour as unlock_value but takes a hash reference.

lock_hash

Lock the whole hash: both its keyset and every value.

unlock_hash

Undo a previous lock_hash, making every key and value writable again.

Synopsis

```
unlock_hash(%hash);
```

Clears the read-only flag on each existing value and lifts the keyset restriction. Returns a reference to the hash.

lock_hashref

Hash-reference variant of lock_hash.

Synopsis

```
lock_hashref($href);
```

Same behaviour as lock_hash but takes a hash reference.

unlock_hashref

Hash-reference variant of unlock_hash.

Synopsis

```
unlock_hashref($href);
```

Same behaviour as unlock_hash but takes a hash reference.

lock_hash_recurse

Lock a hash and every nested hash it references, transitively.

unlock_hash_recurse

Undo a previous lock_hash_recurse, unlocking every nested hash it touched.

Synopsis

```
unlock_hash_recurse(%hash);
```

Traverses the same subset of references as lock_hash_recurse - hash-valued entries only. Returns a reference to the hash.

lock_hashref_recurse

Hash-reference variant of lock_hash_recurse.

Synopsis

```
lock_hashref_recurse($href);
```

Same behaviour as lock_hash_recurse but takes a hash reference.

unlock_hashref_recurse

Hash-reference variant of unlock_hash_recurse.

Synopsis

```
unlock_hashref_recurse($href);
```

Same behaviour as unlock_hash_recurse but takes a hash reference.

hash_locked

Report whether the hash's keyset is currently locked.

Synopsis

```
hash_locked(%hash) and print "restricted\n";
```

Returns 1 when lock_keys (or an equivalent) has been applied to the hash, 0 otherwise. Value-level locks alone do not count - this only reports the keyset restriction.

hash_unlocked

Report whether the hash's keyset is *not* currently locked.

Synopsis

```
hash_unlocked(%hash) and print "open\n";
```

The logical inverse of hash_locked: returns 1 on an unlocked hash, 0 on a locked one. Handy in conditions where testing for the open state reads more naturally than negating the locked one.

hashref_locked

Hash-reference variant of hash_locked.

Synopsis

```
hashref_locked($href) and print "restricted\n";
```

Same behaviour as hash_locked but takes a hash reference.

hashref_unlocked

Hash-reference variant of hash_unlocked.

Synopsis

```
hashref_unlocked($href) and print "open\n";
```

Same behaviour as hash_unlocked but takes a hash reference.

hidden_keys

List the “hidden” placeholder keys of a restricted hash - legal keys that currently have no value assigned.

hidden_ref_keys

Hash-reference variant of hidden_keys.

Synopsis

```
my @hidden = hidden_ref_keys($href);
```

Same behaviour as hidden_keys but takes a hash reference.

legal_keys

List every key the restricted hash *is allowed to carry* - visible keys plus hidden placeholders.

Synopsis

```
my @legal = legal_keys(%hash);
```

On an unrestricted hash this is equivalent to `keys %hash`. On a restricted hash it is the visible keys (`keys %hash`) merged with whatever `hidden_keys` would report.

legal_ref_keys

Hash-reference variant of legal_keys.

Synopsis

```
my @legal = legal_ref_keys($href);
```

Same behaviour as legal_keys but takes a hash reference.

all_keys

Split a restricted hash's keys into visible and hidden buckets, writing the results into the caller's arrays.

hv_store

Install an *alias* from `$hash{$key}` to an existing scalar, instead of copying its value in the usual way.

clear_placeholders

Strip all hidden placeholder entries from a hash.

Synopsis

```
Hash::Util::clear_placeholders(%hash);
```

Intended for internal use by `Hash::Util` itself when preparing a hash for a new lock cycle. The leading underscore marks it as unsupported for application code - its behaviour may change without notice.

Bucket introspection

bucket_ratio

Return a "used/total" string describing how many of the hash's internal buckets hold at least one key.

num_buckets

Return the total number of buckets the hash holds (or would hold if its bucket array were allocated).

Synopsis

```
my $n = num_buckets(%hash);
```

A freshly created hash may report a non-zero bucket count even before its bucket array has been materialised - Perl allocates buckets lazily. The value is always a power of two.

used_buckets

Return the count of buckets that currently hold at least one key.

Synopsis

```
my $used = used_buckets(%hash);
```

The value is computed from scratch on every call by walking the bucket array - there is no cache. Avoid calling it inside a hot loop on a large hash.

bucket_info

Return a flat list summarising the hash's bucket layout.

bucket_array

Return a packed array-of-arrays describing which key lives in which bucket.

hash_traversal_mask

Read or set the per-hash bucket-traversal mask used by keys, values, and each.

bucket_stats

Return a rich statistical summary of the hash's bucket layout.

bucket_stats_formatted

Return a multi-line human-readable report of `bucket_stats`, complete with ASCII bar-charts.

Hash function

hash_seed

Return the per-process random bytes that seed the hash function.

hash_value

Return the integer hash value of a string under the current (or a supplied) seed.

hash_seed

Return the per-process random bytes that seed the hash function.

Synopsis

```
my $seed = hash_seed();
```

The result is a byte string whose length depends on the hash algorithm built into the interpreter - commonly 4 bytes for the classic algorithms and 16 bytes for SipHash. Two interpreters started from the same program see different values; a single run sees the same value throughout its life.

The seed is sensitive. Anyone who learns it can construct keys that collide in your hashes and mount an algorithmic-complexity attack. Keep it out of logs and error messages.

hash_value

Return the integer hash value of a string under the current (or a supplied) seed.

Synopsis

```
my $h = hash_value($string); my $h = hash_value($string, $seed);
```

With one argument, hashes `$string` with the interpreter's per-process seed - the same seed reported by `hash_seed`. With a second argument, hashes it with `$seed` instead, letting you compute what the hash value *would* be under a different seed. The custom seed must be at least as long as `hash_seed` produces; a shorter `$seed` croaks with `seed len must be at least N long only got M bytes`.

The one-argument value is stable for the lifetime of a single interpreter and unstable across runs, Perl versions, build options, and architectures. Do not persist it.

Like `hash_seed`, the result is sensitive information that can be used to engineer collisions. Treat it accordingly.

bucket_ratio

Return a "used/total" string describing how many of the hash's internal buckets hold at least one key.

Synopsis

```
my $r = bucket_ratio(%hash); # e.g. "5/8"
```

Behaves like the pre-5.25 `scalar(%hash)`: for a tied hash it calls the `SCALAR` tie method; for a normal empty hash it returns `0`; otherwise it reports the count of non-empty buckets followed by a slash followed by the total bucket count.

Useful when eyeballing load factor on a large hash. For a structured report reach for `bucket_stats` or `bucket_stats_formatted` instead.

bucket_info

Return a flat list summarising the hash's bucket layout.

Synopsis

```
my ($keys, $buckets, $used, @length_counts) = bucket_info(%hash);
```

The first three values are the number of keys, the total bucket count, and the number of used buckets. Each `$length_counts[$n]` that follows is the number of buckets holding exactly `$n` keys, so `$length_counts[0]` is the empty-bucket count, `$length_counts[1]` the count of single-key buckets, and so on.

Returns the empty list for a non-hash argument or a magical hash. When the hash's bucket array has not been allocated yet, only the three leading values are returned.

bucket_array

Return a packed array-of-arrays describing which key lives in which bucket.

Synopsis

```
my $array = bucket_array(%hash);
```

Each element of the returned array reference is either an integer n - meaning n consecutive empty buckets at that position - or an array reference listing the keys stored in a single bucket. Placeholder keys are reported as normal keys. The order within a bucket is Perl's insert-first chain order and changes across rehashes.

Intended for debugging and diagnostics only. The layout reveals enough about the hash function to support collision attacks, so do not leak the output to untrusted code.

Returns nothing when the argument is not a normal hash reference or has no bucket array yet.

hash_traversal_mask

Read or set the per-hash bucket-traversal mask used by keys, values, and each.

Synopsis

```
my $mask = hash_traversal_mask(%hash); hash_traversal_mask(%hash, 1234);
```

Every hash carries a U32 mask that is XOR'd with each bucket index during iteration, so two hashes with the same contents still visit their keys in different orders. Called with one argument, the function returns the current mask. Called with two, it sets the mask - pinning the iteration order of that particular hash for reproducible test output.

Pinning the mask does not make two *different* hashes iterate identically: keys that collide into the same bucket may still appear in different orders because chain order depends on insertion history.

Currently returns 0 in pperl and ignores any set value; the pperl hash table does not expose a per-hash traversal mask.

bucket_stats

Return a rich statistical summary of the hash's bucket layout.

Synopsis

```
my ($keys, $buckets, $used, $quality, $utilization_ratio, $collision_pct, $mean, $stddev, @length_counts) = bucket_stats(%hash);
```

Extends `bucket_info` with derived quantities: `$quality` is the Red-Dragon quality score (ideal 1.0, acceptable around 0.95–1.05, worse as the number grows), `$utilization_ratio` is the fraction of used buckets, `$collision_pct` is the fraction of keys sharing a bucket with at least one other key, and `$mean` and `$stddev` describe the bucket-chain-length distribution over used buckets.

For an empty hash, only `$keys`, `$buckets`, and `$used` are returned - the statistical block is skipped.

bucket_stats_formatted

Return a multi-line human-readable report of `bucket_stats`, complete with ASCII bar-charts.

Synopsis

```
print bucket_stats_formatted(%hash);
```

The first paragraph of the report gives the summary line (key count, used buckets, total buckets, quality score translated into “Good”, “Poor”, or “Bad”), followed by utilisation and collision percentages and the chain-length mean and standard deviation. Two bar-charts follow: one showing the distribution of bucket chain lengths (how costly a missed lookup is), and one showing how many keys sit at each position in their chain (how costly a found lookup is).

Useful when deciding whether to pre-size a large hash with `keys(%h) = N` to cut collisions.

lock_keys

Restrict a hash to its current keys, or to an explicit list.

Synopsis

```
lock_keys(%hash); # pin the current keyset
lock_keys(%hash, @keys); # pin to the given keyset
```

With one argument, subsequent attempts to assign to a key not already in `%hash` croak with Attempt to access disallowed key `'...'` in a restricted hash. With more arguments, the extra keys specify the allowed set; existing keys that do not appear in that set make the call croak with Hash has key `'...'` which is not in the new key set. Keys in the allowed set that are not yet present are accepted later (their slots start as hidden placeholders).

`delete` and `exists` still work on a locked hash and do not change the legal keyset - a deleted key can be reassigned, but a brand-new key still cannot. Returns a reference to the hash.

A locked hash cannot be blessed; bless before locking if you need both.

unlock_keys

Lift the keyset restriction imposed by `lock_keys`.

Synopsis

```
unlock_keys(%hash);
```

Any keys locked with `lock_value` or `lock_hash` remain locked - this undoes only the keyset restriction. Hidden placeholder keys introduced while the hash was locked stay in place; use `hidden_keys` to see them and ordinary assignment to bring them into the visible keyset.

Returns a reference to the hash.

lock_ref_keys

Hash-reference variant of `lock_keys`.

Synopsis

```
lock_ref_keys($href); lock_ref_keys($href, @keys);
```

Exactly the same effect as `lock_keys`, but takes an already-taken hash reference rather than relying on the `%` prototype. Useful when the hash only exists as a reference - for example, an anonymous hash or a field on an object.

lock_ref_keys_plus

Lock the hash to the union of its current keys and a given extra list.

Synopsis

```
lock_ref_keys_plus($href, @additional_keys); lock_keys_plus(%hash, @additional_keys); # plain-hash form
```

Equivalent to `lock_keys($hash, @additional_keys, keys %hash)` without having to list the existing keys yourself. Extra keys that are not yet present in the hash become hidden placeholders - they are legal to assign to later but do not yet appear in keys.

Returns a reference to the hash.

lock_value

Make the value under a given key immutable.

Synopsis

```
lock_value(%hash, $key);
```

Any subsequent attempt to overwrite `$hash{$key}` croaks with `Modification of a read-only value attempted`. The key itself remains deletable unless `lock_keys` has also been applied - once the key is deleted, a later assignment creates a fresh (writable) value. Values not already present are silently ignored.

Returns a reference to the hash.

lock_hash

Lock the whole hash: both its keyset and every value.

Synopsis

```
lock_hash(%hash);
```

Equivalent to `lock_keys(%hash)` followed by `lock_value` on each existing key. After the call, no key may be added and no value overwritten; `delete` still works but leaves the slot as a hidden placeholder.

Returns a reference to the hash.

lock_hash_recurse

Lock a hash and every nested hash it references, transitively.

Synopsis

```
lock_hash_recurse(%hash);
```

Recurse only through hash-valued entries - a hash-of-hashes is locked throughout, but a hash-of-arrays-of-hashes has only the outermost hash locked because the recursion does not descend into arrays. Reach for `Data::Visitor` or write your own walker if you need deeper traversal.

Returns a reference to the hash.

hidden_keys

List the “hidden” placeholder keys of a restricted hash - legal keys that currently have no value assigned.

Synopsis

```
my @hidden = hidden_keys(%hash);
```

A hidden key is one that is allowed by the hash’s keyset but fails both `exists` and `defined` - typically the ghost left behind after a `delete` on a locked hash, or a key reserved in advance by `lock_keys_plus`. Assigning to a hidden key brings it into the visible keyset.

Returns the empty list on an unrestricted hash. The feature is experimental and closely tied to the current restricted-hash implementation; do not build durable contracts on top of it.

all_keys

Split a restricted hash’s keys into visible and hidden buckets, writing the results into the caller’s arrays.

Synopsis

`all_keys(%hash, @visible, @hidden);`

Clears both arrays first, then fills `@visible` with every key that passes `exists` and `@hidden` with every legal-but-unassigned placeholder key. Returns a reference to the hash.

On an unrestricted hash `@hidden` comes back empty and `@visible` matches keys `%hash`. Like `hidden_keys`, the behaviour is experimental and depends on the current restricted-hash implementation.

hv_store

Install an *alias* from `$hash{$key}` to an existing scalar, instead of copying its value in the usual way.

Synopsis

```
my $sv = 0; hv_store(%hash, $key, $sv) or die "store failed"; $hash{$key} = 1; print $sv;
# prints 1 - $sv and $hash{$key} share one scalar
```

After the call, `$sv` and `$hash{$key}` refer to the same SV: a later assignment through either name is visible through the other. Returns a true value on success, a false value if the store was rejected (for example, on a magical hash that refused the entry).

This is a low-level primitive with narrow uses; ordinary code should assign through the hash normally. Aliasing is fragile once the hash is duplicated, serialised, or passed through `Storable`.

5.2.14 I18N

Namespace containing the following modules:

- `I18N::Langinfo` — Query locale information - country, language, and cultural conventions from the current locale.

I18N::Langinfo

Query locale information - country, language, and cultural conventions from the current locale.

Exports `langinfo()` and `nl_langinfo` constants. Self-contained - operates on P5Sv via P5Interp.

Functions

Other Functions

langinfo

XS wrapper for `langinfo($code)` - calls `libc nl_langinfo()`.

5.2.15 IO

Load the core IO modules - IO::Handle, IO::File, IO::Seekable, IO::Pipe, IO::Socket, IO::Dir - in one use.

This is the bootstrap module that IO::Handle, IO::File, IO::Seekable, IO::Dir, IO::Socket, IO::Pipe all depend on. Without it, require IO fails with “dynamic loading not available”.

Functions are registered in their respective packages (IO::Seekable::, IO::File::, IO::Handle::, IO::Socket::) - NOT in IO:: itself.

The BOOT section installs numeric constants into IO::Poll:: and IO::Handle:: stashes via newCONSTSUB.

Functions

Other Functions

io_import

IO.xs: MODULE = IO - import is a no-op at the XS level. The Perl-level IO.pm does eval "require IO::\$_" for each arg.

fgetpos

IO.xs: 149-173 fgetpos(handle) PerlIO path: RETVAL = newSV(0); PerlIO_getpos(handle, RETVAL) On failure or null handle: returns &PL_sv_undef.

fsetpos

IO.xs: 176-201 fsetpos(handle, pos) Returns 0 on success, -1 on failure.

new_tmpfile

IO.xs: 205-230 new_tmpfile(packname = “IO::File”) Creates a PerlIO tmpfile, wraps in a new GV, do_open with “+>&”, blesses the RV into packname. Returns blessed ref or undef.

io_blocking

IO.xs: 267-281 io_blocking(handle, blk=-1) XS wrapper: calls io_blocking_impl, returns IV or undef.

ungetc

IO.xs: 286-325 ungetc(handle, c) Simplified: we handle the PerlIO path only (no stdio fallback). skipped: UTF-8 multi-byte ungetc path (UVCHR_IS_INVARIANT / Perl_PerlIO_isutf8 / uvchr_to_utf8_flags) - would need full UTF-8 encoding machinery. For now we handle the single-byte invariant case which covers ASCII and the non-utf8-handle path.

ferror

`IO.xs:327-346 ferror(handle)` Returns `Perl_PerlIO_error(in) || (out && in != out && Perl_PerlIO_error(out))`.

clearerr

`IO.xs:348-373 clearerr(handle)` Clears error on both input and output streams. Returns 0 or -1.

untaint

`IO.xs:375-394 untaint(handle)` No-op in our runtime (taint not supported). Returns 0 if handle valid, -1 otherwise. `IO.xs` sets `IoFLAGS(io) |= IOf_UNTAINT` - we just return 0.

fflush

`IO.xs:396-411 fflush(handle)` Returns 0 on success, -1 on error.

setbuf

`IO.xs:413-426 setbuf(handle, ...)` Non-`PERLIO_IS_STDIO` path: `not_here("IO::Handle::setbuf")` With `PerlIO` (our case): if `buf` is undef, call `Perl_PerlIO_setlinebuf`. `IO.xs` only implements this for `PERLIO_IS_STDIO`; for `PerlIO` it calls `not_here`. We implement the `Perl_PerlIO_setlinebuf` path when `buf` is undef (common usage), and croak for the `PERLIO_IS_STDIO` `setbuf` case.

setvbuf

`IO.xs:428-458 setvbuf(handle, buf, type, size)` Only available under `PERLIO_IS_STDIO && _IOFBF && HAS_SETVBUF`. We are `PerlIO` (not `PERLIO_IS_STDIO`), so this croaks.

fsync

`IO.xs:462-492 fsync(arg)` Calls `fsync(Perl_PerlIO_fileno(handle))`.

getlines

`IO.xs:520-565 getlines(...)` / `getline` / `gets` These call `pp_readline` internally. For our implementation: `getlines (ix=0)`: reads all lines in list context, croaks in scalar context. `getline/gets (ix=1,2)`: reads one line in scalar context.

getlines_common

Common implementation for `getlines/getline/gets`. `ix=0` → `getlines` (list context, croak if scalar) `ix=1` → `getline/gets` (scalar context)

socketmark

`IO.xs:567-602 socketmark(sock)` Uses `SIOCATMARK` `ioctl` on Linux (`HAS_SOCKETMARK` path uses `socketmark(3)`, but `ioctl` is the fallback and what Linux actually does).

getlines

`IO.xs:520-565 getlines(...)` / `getline` / `gets` These call `pp_readline` internally. For our implementation: `getlines (ix=0)`: reads all lines in list context, croaks in scalar context. `getline/gets (ix=1,2)`: reads one line in scalar context.

`IO.xs` constructs a fake UNOP with `op_ppaddr=PL_ppaddr[OP_READLINE]` and calls `CALLRUNOPS`. This is deeply tied to perl5 internals.

DEVIATION: We implement `getlines/getline` via `call_method("readline")` rather than constructing a fake op and calling `CALLRUNOPS`. This achieves the same observable behavior (reads lines from the filehandle) without requiring access to `PL_ppaddr` and the ability to construct ops at runtime. The C code does this hack specifically because Perl-level implementation couldn't properly propagate lexical hints to `pp_readline` - but we're calling through the C runtime which handles this natively.

getlines_common

Common implementation for `getlines/getline/gets`. `ix=0` → `getlines` (list context, croak if scalar) `ix=1` → `getline/gets` (scalar context)

`IO.xs:520-565` constructs a fake UNOP with `op_ppaddr=PL_ppaddr[OP_READLINE]` and calls `CALLRUNOPS`. This is deeply tied to perl5's internal op machinery.

DEVIATION: We use `eval_pv` to read via `<$fh>`. The `IO.xs` XS versions exist to work around a lexical hints propagation issue with perl5's pure-Perl `getline/getlines` in `IO::Handle.pm`. Since our C runtime handles hints natively, this workaround achieves the same result.

5.2.16 List

Namespace containing the following modules:

- `List::Util` — The standard library of list reductions and scanners.

List::Util

The standard library of list reductions and scanners.

`List::Util` is the first stop when you need to collapse a list to a single answer (`sum`, `max`, `product`, `reduce`), find or test elements by predicate (`first`, `any`, `all`, `none`, `notall`), strip duplicates (`uniq`, `uniqnum`, `uniqint`, `uniqstr`), or reshape an even-sized key/value list (`pairs`, `pairkeys`, `pairmap`, `pairgrep`). It also covers slicing (`head`, `tail`), randomisation (`shuffle`, `sample`), and lockstep iteration over multiple arrays (`zip`, `mesh`, and their `_shortest` / `_longest` variants).

The block-taking reductions - `reduce`, `reductions`, `pairmap`, `pairgrep`, `pairfirst` - expose the current pair or accumulator through `$a` and `$b`. The per-element scanners - `first`, `any`, `all`, `none`, `notall` - expose the current element through `$_`.

This module also re-exports `Sub::Util` (`subname`, `set_subname`, `set_prototype`); in upstream perl5 these live in the same XS source file and pperl mirrors that arrangement.

Functions

Reductions

sum

Add up every value in the list and return the numerical total.

sum0

Add up every value in the list, returning 0 for an empty list.

min

Return the numerically smallest value in the list.

max

Return the numerically largest value in the list.

product

Multiply every value in the list and return the numerical product.

minstr

Return the lexically smallest value in the list.

maxstr

Return the lexically largest value in the list.

reduce

Collapse a list to a single value by repeated pairwise combination.

reductions

Like `reduce`, but also return every intermediate accumulator value.

Scanners

first

Return the first list element for which a block returns true.

any

True if the block returns true for at least one element.

all

True if the block returns true for every element (including empty list).

none

True if the block returns true for no element (including empty list).

notall

True if the block returns false for at least one element.

Pair handling

pairs

Group an even-sized list into key/value pair objects.

unpairs

Flatten a list of two-element arrayrefs into a key/value list.

pairkeys

Extract just the keys from an even-sized key/value list.

pairvalues

Extract just the values from an even-sized key/value list.

pairmap

Run a block for each pair and concatenate the results.

pairgrep

Keep the pairs for which a block returns true.

pairfirst

Return the first pair for which a block returns true.

Selection

head

Return the first \$size elements of a list.

tail

Return the last \$size elements of a list.

uniq

Remove subsequent duplicates by string equality, preserving order.

uniqnum

Remove subsequent duplicates by numerical equality, preserving order.

uniqint

Remove subsequent duplicates by integer equality, preserving order.

Misc

shuffle

Return the input list in a uniformly random order.

sample

Pick \$count distinct elements from the list at random.

mesh

Interleave multiple arrays, padding shorter ones with undef.

mesh_shortest

Interleave multiple arrays, stopping at the shortest.

zip

Walk multiple arrays in lockstep, returning one arrayref per row.

zip_shortest

Walk multiple arrays in lockstep, stopping at the shortest.

sub_util_subname

Return the fully qualified name of a subroutine reference.

sub_util_set_subname

Assign a name to a subroutine reference in place.

sub_util_set_prototype

Set or clear the prototype on a subroutine reference.

sum

Add up every value in the list and return the numerical total.

Synopsis

```
my $total = sum @values;  
my $total = sum 1..10;           # 55
```

What you get back

A single number - an integer if every argument summed as an integer, a float otherwise. Integer arithmetic wraps on overflow rather than promoting, matching upstream. For an empty list, sum returns undef for backwards compatibility; if that is awkward use sum0 instead.

Examples

```
my $n = sum 3, 9, 12;           # 24  
my $n = sum @bar, @baz;         # concatenated lists  
my $n = sum();                  # undef  
my $n = sum map { length } @s;  # total length of all strings
```

Edge cases

- Empty list returns undef. Use sum0 for a 0 default.
- Mixing ints and floats promotes the running total to a float.
- Non-numeric strings are coerced via the usual numification.

Differences from upstream

Fully compatible with upstream.

See also

- `sum0` - same as `sum`, but returns `0` for an empty list.
- `product` - multiplies rather than adds.
- `reduce` - the generic reduction when `sum` is not specific enough.

`sum0`

Add up every value in the list, returning `0` for an empty list.

Synopsis

```
my $total = sum0 @values;  
my $total = sum0 ();           # 0, not undef
```

What you get back

The same number `sum` would produce, except that an empty list yields `0` instead of `undef`. This is almost always what callers want when adding up a list that may legitimately be empty.

Examples

```
my $n = sum0 1..10;           # 55  
my $n = sum0 ();              # 0  
my $n = sum0 grep { $_ > 0 } @x; # total of positives, 0 if none
```

Differences from upstream

Fully compatible with upstream.

See also

- `sum` - the original; returns `undef` on empty input.
- `product` - identity element for multiplication (returns `1` on empty).

`min`

Return the numerically smallest value in the list.

Synopsis

```
my $lo = min @values;  
my $lo = min 3, 9, 12;      # 3
```


What you get back

A single number. If every argument is an integer the result is returned as an integer; otherwise it is returned as a float. An empty list yields undef.

Examples

```
my $n = min 1..10;           # 1
my $n = min @bar, @baz;      # smallest across concatenated
→lists
my $n = min();               # undef
```

Edge cases

- Empty list returns undef.
- Strings are compared numerically (use minstr for lexical order).

Differences from upstream

Fully compatible with upstream.

See also

- max - the numerical maximum counterpart.
- minstr - string-wise minimum (lt semantics).
- reduce - custom minima (e.g. by a key function).

max

Return the numerically largest value in the list.

Synopsis

```
my $hi = max @values;
my $hi = max 3, 9, 12;      # 12
```

What you get back

A single number. If every argument is an integer the result is returned as an integer; otherwise it is returned as a float. An empty list yields undef.

Examples

```
my $n = max 1..10;           # 10
my $n = max @bar, @baz;      # largest across concatenated lists
my $n = max();               # undef
```

Edge cases

- Empty list returns undef.
- Strings are compared numerically (use `maxstr` for lexical order).

Differences from upstream

Fully compatible with upstream.

See also

- `min` - the numerical minimum counterpart.
- `maxstr` - string-wise maximum (gt semantics).
- `reduce` - custom maxima (e.g. by a key function).

product

Multiply every value in the list and return the numerical product.

Synopsis

```
my $p = product @values;  
my $p = product 1..10;           # 3628800
```

What you get back

A single number. Integer overflow is promoted to floating point transparently, so the result is an integer as long as the exact product fits in an IV. An empty list yields 1 (the multiplicative identity).

Examples

```
my $n = product 3, 9, 12;         # 324  
my $n = product();               # 1  
my $n = product map { $_->weight } @items;
```

Differences from upstream

Fully compatible with upstream.

See also

- `sum` - additive counterpart.
- `sum0` - additive counterpart that mirrors `product`'s identity-element behaviour on empty input.
- `reduce` - the generic reduction when `product` is not specific enough.

head

Return the first `$size` elements of a list.

Synopsis

```
my @first = head $size, @list;
my @first = head 2, qw( foo bar baz );    # ('foo', 'bar')
```

What you get back

A list of at most `$size` elements, taken from the start of the input. If `$size` is negative, `head` returns all but the last `-$size` elements - useful for dropping a tail.

Examples

```
my @r = head 2, qw( foo bar baz );    # ('foo', 'bar')
my @r = head -2, qw( foo bar baz );    # ('foo')
my @r = head 5, qw( a b c );           # ('a', 'b', 'c') - fewer than
→$size
```

Edge cases

- `$size >= @list` returns the whole list.
- `$size == 0` or `-$size >= @list` returns the empty list.

Differences from upstream

Fully compatible with upstream.

See also

- `tail` - mirror operation, taking from the end.
- `sample` - random selection without positional bias.

tail

Return the last `$size` elements of a list.

Synopsis

```
my @last = tail $size, @list;
my @last = tail 2, qw( foo bar baz );    # ('bar', 'baz')
```

What you get back

A list of at most `$size` elements, taken from the end of the input. If `$size` is negative, `tail` returns all but the first `-$size` elements - useful for dropping a head.

Examples

```
my @r = tail 2, qw( foo bar baz ); # ('bar', 'baz')
my @r = tail -2, qw( foo bar baz ); # ('baz')
my @r = tail 5, qw( a b c ); # ('a', 'b', 'c') - fewer than
↪$size
```

Differences from upstream

Fully compatible with upstream.

See also

- `head` - mirror operation, taking from the start.
- `sample` - random selection without positional bias.

shuffle

Return the input list in a uniformly random order.

Synopsis

```
my @mixed = shuffle @values;
my @deck = shuffle 0..51; # a shuffled deck
```

What you get back

A new list with the same elements as the input, permuted by a Fisher-Yates shuffle. Each permutation is equally likely given a well-distributed random source. The source is the global `$List::Util::RAND`, or perl's built-in `rand()` when `$RAND` is undefined.

Examples

```
my @picked = shuffle @candidates;
my ($winner) = shuffle @entries; # pick one at random
local $List::Util::RAND = sub { 0.5 }; # deterministic for tests
```

Differences from upstream

Fully compatible with upstream.

See also

- `sample` - take a fixed count without replacement.
- `head` - deterministic prefix selection.

sample

Pick `$count` distinct elements from the list at random.

Synopsis

```
my @picks = sample $count, @values;  
my @three = sample 3, 1..100;
```

What you get back

A list of at most `$count` elements drawn without replacement. If `$count` is larger than the input, every element is returned in a random order - i.e. the function degrades to shuffle. The random source is `$List::Util::RAND` if set, otherwise perl's built-in `rand()`.

Examples

```
my @preview = sample 5, @articles;  
my ($winner) = sample 1, @entries;  
my @all      = sample 999, qw( a b c ); # ('c','a','b') say
```

Edge cases

- `$count <= 0` returns the empty list.
- Fewer inputs than `$count` returns all inputs, shuffled.

Differences from upstream

Fully compatible with upstream.

See also

- `shuffle` - permute the whole list.
- `head` - deterministic prefix (no randomness).

uniq

Remove subsequent duplicates by string equality, preserving order.

Registered under both `uniq` and `uniqstr`. Both treat elements as strings: two values are the same when their stringifications are byte-for-byte identical. The first occurrence of any given value wins; later duplicates are dropped.

Synopsis

```
my @subset = uniq    @values;  
my @subset = uniqstr @values;  
my $count  = uniq    @values;    # scalar context - count only
```

What you get back

In list context, the unique elements in first-seen order. In scalar context, the count of unique elements. `undef` compares equal to the empty string under `uniqstr` but is preserved as `undef` in the output by `uniq`.

Examples

```
my @u = uniq    qw( a b a c b );    # ('a', 'b', 'c')  
my @u = uniqstr 'a', 'A';           # ('a', 'A') - case matters  
my $n = uniq    qw( a b a c b );    # 3
```

Differences from upstream

Fully compatible with upstream.

See also

- `uniqnum` - compare as numbers rather than as strings.
- `uniqint` - compare as integers after truncation.

uniqnum

Remove subsequent duplicates by numerical equality, preserving order.

Synopsis

```
my @subset = uniqnum @values;  
my $count  = uniqnum @values;    # scalar context - count only
```

What you get back

In list context, the unique numerical values in first-seen order. In scalar context, the count. `undef` is treated as `0` and coerced to `0` in the output (matching upstream's numerical handling). `+0.0` and `-0.0` are treated as equal. Every NaN is treated as a duplicate of every other NaN, despite `NaN != NaN`.

Examples

```
my @u = unqnum 1, 1.0, '1', 2;    # (1, 2)
my @u = unqnum 0, -0.0;          # (0)
my $n = unqnum 1, 2, 3, 2, 1;    # 3
```

Differences from upstream

Fully compatible with upstream.

See also

- `uniq` - string-equality deduplication.
- `uniqint` - integer-equality deduplication (ignores fractions).

uniqint

Remove subsequent duplicates by integer equality, preserving order.

Synopsis

```
my @subset = uniqint @values;
my $count  = uniqint @values;    # scalar context - count only
```

What you get back

In list context, the unique integer values in first-seen order - every input is truncated to its integer part before comparison, and the returned list itself contains integers (not the original SVs). `undef` is treated as `0`.

Examples

```
my @u = uniqint 1.1, 1.9, 2.0;    # (1, 2)
my @u = uniqint '3', 3, 3.7;      # (3)
my $n = uniqint 1, 2, 2, 3;       # 3
```

Differences from upstream

Fully compatible with upstream.

See also

- `unqnum` - full numerical (float) equality.
- `uniq` - string-equality deduplication.

minstr

Return the lexicographically smallest value in the list.

Synopsis

```
my $s = minstr @values;  
my $s = minstr 'A'..'Z';           # 'A'
```

What you get back

The first (original) list element whose stringification compares less than every other element's stringification under `lt` (byte-wise comparison). Empty list yields `undef`.

Examples

```
my $s = minstr 'hello', 'world';    # 'hello'  
my $s = minstr qw( foo bar baz );  # 'bar'  
my $s = minstr ();                  # undef
```

Differences from upstream

Fully compatible with upstream.

See also

- `maxstr` - string-wise maximum.
- `min` - numerical minimum.

maxstr

Return the lexicographically largest value in the list.

Synopsis

```
my $s = maxstr @values;  
my $s = maxstr 'A'..'Z';           # 'Z'
```

What you get back

The first (original) list element whose stringification compares greater than every other element's stringification under `gt` (byte-wise comparison). Empty list yields `undef`.

Examples

```
my $s = maxstr 'hello', 'world';    # 'world'
my $s = maxstr qw( foo bar baz );  # 'foo'
my $s = maxstr ();                  # undef
```

Differences from upstream

Fully compatible with upstream.

See also

- minstr - string-wise minimum.
- max - numerical maximum.

pairs

Group an even-sized list into key/value pair objects.

Synopsis

```
my @pairs = pairs @kvlist;
for my $p ( pairs %hash ) {
    my ($k, $v) = @$p;           # or $p->key / $p->value
}
```

What you get back

A list of two-element blessed array references (class `List::Util::_Pair`). Each pair responds to `->key`, `->value`, and `->TO_JSON`, and can also be dereferenced as a plain arrayref. For an odd-sized input, the final pair's value is `undef`.

Examples

```
for my $p ( pairs one => 1, two => 2 ) {
    print $p->key, '=', $p->value, "\n";
}
```

```
## one=1
```

```
## two=2
```

```
my @sorted = sort { $a->key cmp $b->key } pairs %h;
```

Differences from upstream

Fully compatible with upstream.

See also

- `unpairs` - inverse: flatten an arrayref list back to key/value.
- `pairkeys` / `pairvalues` - extract just one side.
- `pairmap` / `pairgrep` - transform or filter pairs in place.

unpairs

Flatten a list of two-element arrayrefs into a key/value list.

Synopsis

```
my @kvlist = unpairs @pairs;  
my %hash   = unpairs @pairs;
```

What you get back

A flat list with exactly two elements per input arrayref: `[0]` becomes the key, `[1]` becomes the value. Missing elements and non-arrayref inputs pad with `undef` so that the result is always even-sized.

Examples

```
my @kv = unpairs [a => 1], [b => 2];    # ('a', 1, 'b', 2)  
my %h  = unpairs pairs %other;          # round-trip  
my @kv = unpairs sort { $a->[0] cmp $b->[0] } pairs %h;
```

Differences from upstream

Fully compatible with upstream.

See also

- `pairs` - the inverse direction.

pairkeys

Extract just the keys from an even-sized key/value list.

Synopsis

```
my @keys = pairkeys @kvlist;
my @keys = pairkeys %hash;
```

What you get back

A list of every even-indexed element (indices 0, 2, 4, ...). Equivalent to but more efficient than

```
pairmap { $a } @kvlist
```

Examples

```
my @k = pairkeys one => 1, two => 2;    # ('one', 'two')
my @k = pairkeys %ENV;                  # names of every env var
```

Differences from upstream

Fully compatible with upstream.

See also

- `pairvalues` - the value-side counterpart.
- `pairs` - keep keys and values bound together.

pairvalues

Extract just the values from an even-sized key/value list.

Synopsis

```
my @values = pairvalues @kvlist;
my @values = pairvalues %hash;
```

What you get back

A list of every odd-indexed element (indices 1, 3, 5, ...). If the input has an orphan key at the end, its corresponding value is `undef`. Equivalent to but more efficient than

```
pairmap { $b } @kvlist
```

Examples

```
my @v = pairvalues one => 1, two => 2;    # (1, 2)
my @v = pairvalues %ENV;                  # values of every env var
```

Differences from upstream

Fully compatible with upstream.

See also

- `pairkeys` - the key-side counterpart.
- `pairs` - keep keys and values bound together.

pairmap

Run a block for each pair and concatenate the results.

Synopsis

```
my @out    = pairmap { BLOCK } @kvlist;  
my $count = pairmap { BLOCK } @kvlist;    # scalar context
```

For every pair the block is called in list context with `$a` set to the key and `$b` set to the value; its results are appended to the output list in order.

What you get back

In list context, the concatenation of everything every invocation of the block returned. In scalar context, the number of items that would have been returned in list context. `$a` and `$b` are aliased to the original list elements - modifications in the block are visible to the caller.

Examples

```
my @lines = pairmap { "The key $a has value $b" } %hash;  
my @flat  = pairmap { ( $a, uc $b ) } %h;    # values upper-cased  
my @pairs = pairmap { [ $a, $b ] } %h;      # equivalent to `pairs`
```

Differences from upstream

Fully compatible with upstream.

See also

- `pairgrep` - filter pairs by a predicate.
- `pairfirst` - stop at the first matching pair.
- `pairs` - bundle pairs into blessed arrayrefs.

pairgrep

Keep the pairs for which a block returns true.

Synopsis

```
my @kept = pairgrep { BLOCK } @kvlist;
my $count = pairgrep { BLOCK } @kvlist; # scalar context
```

For every pair the block is called in scalar context with \$a set to the key and \$b set to the value. Pairs for which the block returns true are included in the result; others are dropped.

What you get back

In list context, an even-sized list: the kept keys and values interleaved. In scalar context, the number of pairs (not elements) that matched - half of what the list-context length would be. \$a and \$b are aliased to the original list elements.

Examples

```
my %big = pairgrep { $b > 100 } %totals;
my @uppers = pairgrep { $a =~ /^[A-Z]/ } %h;
my $n_big = pairgrep { $b > 100 } %totals; # scalar: pair count
```

Differences from upstream

Fully compatible with upstream.

See also

- pairmap - transform pairs instead of filtering.
- pairfirst - find the first matching pair and stop.

pairfirst

Return the first pair for which a block returns true.

Synopsis

```
my ($k, $v) = pairfirst { BLOCK } @kvlist;
my $found = pairfirst { BLOCK } @kvlist; # scalar context
```

For each pair the block is called in scalar context with \$a set to the key and \$b set to the value. pairfirst stops at the first pair for which the block returns true.

What you get back

In list context, the matching (`$key`, `$value`) pair, or an empty list if nothing matched. In scalar context, a boolean indicating whether any pair matched. `$a` and `$b` are aliased to the original list elements.

Examples

```
my ($k, $v) = pairfirst { $a =~ /^upper_/ } %config;
if ( pairfirst { $b > 1_000 } %totals ) {
    warn "at least one big entry";
}
```

Differences from upstream

Fully compatible with upstream.

See also

- `first` - non-pair equivalent, uses `$_`.
- `pairgrep` - collect every matching pair.

mesh

Interleave multiple arrays, padding shorter ones with `undef`.

Registered under both `mesh` and `mesh_longest`. Both forms stop at the length of the longest input and fill missing elements with `undef`.

Synopsis

```
my @out = mesh      \@a, \@b, \@c;
my @out = mesh_longest \@a, \@b, \@c;
my %h   = mesh \@keys, \@values;
```

What you get back

A flat list containing elements at position `0` from each input array, then position `1`, and so on, up to the longest input. Missing positions are filled with `undef`.

Examples

```
my @r = mesh [1..3], ['a'..'c'];           # (1,'a', 2,'b', 3,'c')
my @r = mesh [1..3], ['a'..'b'];           # (1,'a', 2,'b', 3,undef)
my %h = mesh \@keys, \@values;             # build a hash
```

Edge cases

- An input that is not an arrayref contributes only undefs.
- Empty input list returns an empty list.

Differences from upstream

Fully compatible with upstream.

See also

- `mesh_shortest` - stop at the shortest input.
- `zip` - same lockstep walk, but yields one arrayref per row.

mesh_shortest

Interleave multiple arrays, stopping at the shortest.

Synopsis

```
my @out = mesh_shortest \@a, \@b, \@c;
```

What you get back

A flat list containing elements at position 0 from each input, then 1, and so on - up to the length of the shortest input. No undef padding.

Examples

```
my @r = mesh_shortest [1..3], ['a','b'];    # (1,'a', 2,'b')
my @r = mesh_shortest [1..3], [];           # ()
```

Differences from upstream

Fully compatible with upstream.

See also

- `mesh` / `mesh_longest` - pad to longest instead.
- `zip_shortest` - same lockstep walk, but yields one arrayref per row.

zip

Walk multiple arrays in lockstep, returning one arrayref per row.

Registered under both `zip` and `zip_longest`. Both forms stop at the length of the longest input and fill missing positions with undef.

Synopsis

```
my @rows = zip      \@a, \@b, \@c;  
my @rows = zip_longest \@a, \@b, \@c;  
foreach ( zip \@xs, \@ys ) {  
    my ($x, $y) = @$_;  
}
```

What you get back

A list of arrayrefs, one per row. Row *i* contains the *i*-th element of each input array in order. Shorter inputs contribute undef beyond their own length.

Examples

```
my @r = zip [1..3], ['a'..'c']; # ([1,'a'], [2,'b'], [3,'c'])  
my @r = zip [1..3], ['a'..'b']; # ([1,'a'], [2,'b'], [3,undef])
```

Differences from upstream

Fully compatible with upstream.

See also

- `zip_shortest` - stop at the shortest input.
- `mesh` - flatten the rows instead of wrapping in arrayrefs.

zip_shortest

Walk multiple arrays in lockstep, stopping at the shortest.

Synopsis

```
my @rows = zip_shortest \@a, \@b, \@c;
```

What you get back

A list of arrayrefs, one per row. Row *i* contains the *i*-th element of each input array - produced only while every input still has an *i*-th element. No undef padding.

Examples

```
my @r = zip_shortest [1..3], ['a','b']; # ([1,'a'], [2,'b'])  
my @r = zip_shortest [1..3], [];       # ()
```


Differences from upstream

Fully compatible with upstream.

See also

- `zip / zip_longest` - pad to longest instead.
- `mesh_shortest` - flatten the rows instead of wrapping in arrayrefs.

reduce

Collapse a list to a single value by repeated pairwise combination.

Synopsis

```
my $result = reduce { BLOCK } @list;
my $sum    = reduce { $a + $b } 1..10;      # 55
my $max    = reduce { $a > $b ? $a : $b } @xs;
```

The block is called in scalar context with `$a` set to the running accumulator and `$b` set to the next element. On the first call, `$a` and `$b` are the first two elements; on each subsequent call, `$a` is whatever the previous call returned.

What you get back

The value of the last block invocation. If the list is empty, `undef` is returned. If the list contains exactly one element, that element is returned and the block is not called.

Examples

```
my $concat = reduce { "$a,$b" } 'a'..'c';    # 'a,b,c'
my $best   = reduce { $a->score > $b->score ? $a : $b } @players;
my $total  = reduce { $a + length $b } 0, @strings; # sum of
↳ lengths
```

```
## Guarantee a numeric identity so an empty @values doesn't yield
↳ undef:

my $sum = reduce { $a + $b } 0, @values;
```

Differences from upstream

Fully compatible with upstream.

See also

- `reductions` - like `reduce`, but also keeps the intermediate values.
- `sum` / `product` / `min` / `max` - specialised, faster forms.
- `first` / `any` / `all` - element-test reductions over `$_`.

reductions

Like `reduce`, but also return every intermediate accumulator value.

Synopsis

```
my @steps = reductions { BLOCK } @list;  
my @running = reductions { $a + $b } 1..4;    # (1, 3, 6, 10)
```

The block is called in the same way as for `reduce`: `$a` is the accumulator, `$b` is the next element. `reductions` differs in that it captures the accumulator at every step rather than just the final value.

What you get back

A list whose first element is the first input value and whose subsequent elements are each block invocation's return value in order. The last element equals what `reduce` would have returned for the same block and list.

Examples

```
reduce      { "$a-$b" } 'a'..'d';    # 'a-b-c-d'  
reductions { "$a-$b" } 'a'..'d';    # ('a', 'a-b', 'a-b-c', 'a-b-c-  
→d')
```

```
my @running_sum = reductions { $a + $b } @values;
```

Differences from upstream

Fully compatible with upstream.

See also

- `reduce` - when only the final result is needed.

first

Return the first list element for which a block returns true.

Synopsis

```
my $val = first { BLOCK } @list;  
my $d   = first { defined } @list;  
my $big = first { $_ > 100 } @values;
```

For each element `first` sets `$_` to that element and calls the block in scalar context; the first element for which the block returns true is returned. Remaining elements are not examined.

What you get back

The matching element (the original value, not the block's return). If no element matches - or the list is empty - returns `undef`.

Examples

```
my $first_defined = first { defined $_ } @list;  
my $first_big    = first { $_ > $threshold } @measurements;  
my $first_admin  = first { $_->is_admin } @users;
```

Differences from upstream

Fully compatible with upstream.

See also

- `any` - boolean “at least one matches” test.
- `grep` - collect every matching element.
- `pairfirst` - pair-oriented counterpart.

any

True if the block returns true for at least one element.

Synopsis

```
my $bool = any { BLOCK } @list;  
if ( any { length > 10 } @strings ) { ... }
```

For each element `any` sets `$_` to that element and calls the block in scalar context. It returns true as soon as the block returns true for any element, without examining the rest. For an empty list it returns false.

What you get back

1 if any element satisfied the block, 0 otherwise.

Examples

```
warn "has empty"    if any { $_ eq '' } @values;  
die "has admin"    if any { $_->is_admin } @users;  
my $ok = any { /^--help$/ } @ARGV;
```

Differences from upstream

Fully compatible with upstream.

See also

- `all` - require every element to match.
- `none` - require no element to match.
- `first` - return the matching element itself.

all

True if the block returns true for every element (including empty list).

Synopsis

```
my $bool = all { BLOCK } @list;  
if ( all { defined } @values ) { ... }
```

For each element `all` sets `$_` to that element and calls the block in scalar context. It returns false as soon as the block returns false for any element. An empty list yields true (the conventional “vacuous truth”).

What you get back

1 if every element satisfied the block, 0 otherwise.

Examples

```
die "found empty string" unless all { length } @strings;  
my $ready = all { $_->is_ready } @workers;  
my $yes   = all { $_ > 0 } ();           # 1 - vacuous truth
```

Differences from upstream

Fully compatible with upstream.

See also

- `any` - require at least one match.
- `notall` - inverse: at least one failure.
- `none` - inverse: zero matches.

none

True if the block returns true for no element (including empty list).

Synopsis

```
my $bool = none { BLOCK } @list;  
if ( none { $_ < 0 } @measurements ) { ... }
```

For each element `none` sets `$_` to that element and calls the block in scalar context. It returns false as soon as the block returns true for any element. An empty list yields true.

What you get back

1 if every element failed the block, 0 otherwise.

Examples

```
die "negatives present" unless none { $_ < 0 } @measurements;  
my $clean = none { /ERROR/ } @log_lines;  
my $none = none { 1 } (); # 1 - nothing to fail
```

Differences from upstream

Fully compatible with upstream.

See also

- `any` - inverse: at least one match.
- `all` - require every element to match.
- `notall` - at least one failure.

notall

True if the block returns false for at least one element.

Synopsis

```
my $bool = notall { BLOCK } @list;
warn "some missing" if notall { defined } @values;
```

For each element `notall` sets `$_` to that element and calls the block in scalar context. It returns true as soon as the block returns false for any element. An empty list yields false.

What you get back

1 if at least one element failed the block, 0 otherwise.

Examples

```
my $incomplete = notall { defined } @row_cells;
warn "stale" if notall { $_->is_fresh } @entries;
my $false = notall { 1 } ();                # 0 - nothing to fail
```

Differences from upstream

Fully compatible with upstream.

See also

- `all` - inverse: require every element to match.
- `any` - at least one match.
- `none` - zero matches.

sub_util_subname

Return the fully qualified name of a subroutine reference.

Synopsis

```
use Sub::Util qw(subname);
my $name = subname \&some_sub;           # 'Pkg::some_sub'
my $name = subname sub { 1 };             # 'main::__ANON__'
```

What you get back

A string of the form `"Package::name"`. Anonymous subroutines produce `"Package::__ANON__"`. If the subroutine has been renamed with `set_subname`, the new name is returned. Passing something that is not a code reference croaks with `Not a subroutine reference`.

Examples

```
package My::Pkg;
sub hello { 1 }
print Sub::Util::subname(\&hello);    # 'My::Pkg::hello'
```

Differences from upstream

Fully compatible with upstream.

See also

- `set_subname` - rename a code reference (useful for stack traces).

sub_util_set_subname

Assign a name to a subroutine reference in place.

Synopsis

```
use Sub::Util qw(set_subname);
my $code = set_subname 'My::Pkg::name', sub { ... };
```

What you get back

The same code reference, now reporting `$name` under `subname`, `caller`, and stack traces. If `$name` contains `::` or a quote-style separator (`'`), the left part becomes the package and the right part the short name. An unqualified name is attached to the current compilation package.

Examples

```
my $sub = set_subname 'My::Thing::handler', sub { process(@_) };
print Sub::Util::subname($sub);    # 'My::Thing::handler'
```

```
## Naming closures so stack traces are readable:
```

```
for my $op (qw( add sub mul )) {
    $dispatch{$op} = set_subname "op_$op", sub { ... };
}
```

Edge cases

- Croaks with `Not a subroutine reference` for non-CODE arguments.
- Croaks with `set_subname: not enough arguments on missing args`.

Differences from upstream

Fully compatible with upstream.

See also

- `subname` - read back the fully qualified name.
- `set_prototype` - adjust the prototype of a code reference.

sub_util_set_prototype

Set or clear the prototype on a subroutine reference.

Synopsis

```
use Sub::Util qw(set_prototype);
my $code = set_prototype '$$', \&adder;
my $code = set_prototype undef, \&adder;    # clear the prototype
```

What you get back

The same code reference, with its prototype replaced by `$proto` (a string like `'$$'`, `'&@'`, `'\@'`) or cleared when `$proto` is `undef`.

Examples

```
set_prototype '$$', \&my_add;    # force two scalar args
set_prototype '&@', \&wrapper;    # block followed by a list
set_prototype undef, \&free_form; # no prototype
```

Edge cases

- Croaks with `set_prototype`: not a reference for scalars.
- Croaks with `set_prototype`: not a subroutine reference for non-CODE refs.
- Croaks with `set_prototype`: not enough arguments on missing args.

Differences from upstream

Fully compatible with upstream.

See also

- `set_subname` - rename a code reference.
- `prototype` - perl built-in that reads a prototype.

5.2.17 MIME

Namespace containing the following modules:

- `MIME::Base64` — Encode and decode base64 strings - the standard MIME flavour plus the URL-safe variant and cheap length helpers.
- `MIME::QuotedPrint` — Quoted-printable encoding and decoding as specified by RFC 2045.

MIME::Base64

Encode and decode base64 strings - the standard MIME flavour plus the URL-safe variant and cheap length helpers.

`MIME::Base64` turns arbitrary byte strings into the 65-character ASCII subset defined by RFC 2045 (`[A-Za-z0-9+/=]`) and back. Most callers reach for it for one of a handful of jobs:

- writing MIME-style bodies where encoded output must be wrapped at 76 characters per line (the default `encode_base64` form);
- producing a single unwrapped token for an HTTP header, a credential, or a config file - `encode_base64($bytes, "")`;
- building URL-safe tokens (JWT payloads, signed cookies, opaque identifiers) where `+` and `/` would need percent-escaping - `encode_base64url`;
- sizing a buffer without paying the cost of the full encode or decode - `encoded_base64_length` and `decoded_base64_length`.

Line-length wrapping

`encode_base64` breaks the output into lines of **at most 76 characters**, each terminated by the caller's end-of-line sequence (default `"\n"`). Pass `"` as the second argument to suppress wrapping entirely; pass `"\r\n"` for CRLF output suitable for raw MIME transport. The URL-safe encoder never wraps and never pads.

Decode leniency

`decode_base64` is deliberately forgiving. Any character outside the 65-character alphabet is silently skipped, so embedded whitespace, line breaks, and MIME boilerplate pass through harmlessly. Decoding stops at the first = padding group - data after padding is never consumed. The URL-safe decoder accepts input with or without padding and translates `-/_` back to `+//` before running the same state machine.

Who calls this

Most real-world consumers are `Authen::*` (HTTP Basic auth), `Email::MIME` and friends (message bodies), JWT/JOSE libraries (via the URL-safe variant), and anything that needs to round-trip binary through a text channel. `encode_base64` and `decode_base64` are exported by default; the URL-safe and length helpers are exported on request.

Functions

Standard encoding

`encode_base64`

Encode a byte string as standard RFC 2045 base64, optionally wrapping at 76 characters.

`decode_base64`

Decode a base64 string back to the original byte sequence, tolerating stray whitespace and line breaks.

URL-safe variant

`encode_base64url`

Encode bytes using the URL- and filename-safe base64 alphabet, with no padding and no line breaks.

`decode_base64url`

Decode a URL-safe base64 string, accepting input with or without trailing padding.

Length helpers

`encoded_base64_length`

Return the length `encode_base64` would produce, without doing the encoding.

`decoded_base64_length`

Return the length `decode_base64` would produce, without doing the decoding.

`encode_base64`

Encode a byte string as standard RFC 2045 base64, optionally wrapping at 76 characters.

Synopsis

```
use MIME::Base64;
my $encoded = encode_base64($bytes);      # wrapped at 76, "\n"-
    ↪terminated
my $oneline = encode_base64($bytes, "");  # no wrapping, no
    ↪trailing EOL
my $crlf     = encode_base64($bytes, "\r\n"); # MIME-style CRLF
    ↪wrapping
```

What you get back

A plain (non-UTF8) ASCII string containing only characters from [A-Za-z0-9+/=] plus the caller's end-of-line sequence. With the default "\n" terminator, every 76-character line - including the final short one - ends with "\n". Pass "" to get a single unbroken token with no trailing newline; useful for HTTP headers, credential strings, and anywhere a single atomic value is expected. The output always pads to a multiple of four characters with =.

Examples

```
encode_base64("Aladdin:open sesame");  
  
## "QWxhZGRpbjpvcGVuIHNlc2FtZQ==\n"  
  
encode_base64("Aladdin:open sesame", "");  
  
## "QWxhZGRpbjpvcGVuIHNlc2FtZQ=="  
  
encode_base64("\x00\x00\x00");  
  
## "AAAA\n"
```

Edge cases

- Empty input returns the empty string.
- A single character (or any input whose length is not a multiple of 3) is padded with = so the output length is a multiple of four.
- Binary bytes are encoded as-is; there is no byte-order or encoding transformation.
- Input with code points above 255 is a programming error in the caller; base64 is defined over octets only. Encode to UTF-8 first if the source is a text string.

Differences from upstream

Fully compatible with upstream MIME : : Base64 3.16.

See also

- `decode_base64` - the inverse operation.
- `encode_base64url` - the URL-safe variant, no padding, no wrapping.
- `encoded_base64_length` - size the output without encoding.

`decode_base64`

Decode a base64 string back to the original byte sequence, tolerating stray whitespace and line breaks.

Synopsis

```
use MIME::Base64;  
my $bytes = decode_base64($encoded);
```

What you get back

A byte string carrying the decoded octets. The result is binary; treat it as bytes, not text. If the input was the output of `encode_base64`, the result is byte-for-byte what was encoded.

The decoder silently skips any character outside the 65-character base64 subset, so wrapped lines, CRLFs, leading/trailing whitespace, and MIME header remnants are handled without special preprocessing. Decoding stops at the first = padding group; data appearing after padding is ignored.

Examples

```
decode_base64("QWxhZGRpbjpvYGVuIHNlc2FtZQ==");  
## "Aladdin:open sesame"  
  
decode_base64("QWxh\nZGRp\nbjpv\ncGVu\nIHNl\nc2Ft\nZQ==\n");  
## "Aladdin:open sesame"    (wrapped input decodes identically)  
  
decode_base64("AAAA");  
## "\x00\x00\x00"
```

Edge cases

- Empty input returns the empty string.
- Input of all-whitespace returns the empty string.
- A truncated final group (1 valid character) produces no output for that group; 2 valid characters produce 1 byte, 3 produce 2 bytes.
- Characters after the first = padding group are discarded.
- Non-alphabet characters anywhere in the input are silently skipped, including control characters and punctuation.

Differences from upstream

Fully compatible with upstream `MIME::Base64` 3.16.

See also

- `encode_base64` - the inverse operation.
- `decode_base64url` - the URL-safe variant, accepts `-/_` and missing padding.
- `decoded_base64_length` - size the output without decoding.

encode_base64url

Encode bytes using the URL- and filename-safe base64 alphabet, with no padding and no line breaks.

Synopsis

```
use MIME::Base64 qw(encode_base64url);
my $token = encode_base64url($bytes);
```

What you get back

A single unbroken ASCII string using the alphabet `[A-Za-z0-9-_]`. Trailing = padding is stripped. The result is safe to drop into URLs, path segments, cookie values, and JWT components without percent-escaping. The output is always one line - never wrapped.

Examples

```
encode_base64url("Aladdin:open sesame");

## "QWxhZGRpbjpvcGVuIHNlc2FtZQ"

encode_base64url("\xFB\xFF");

## "-_8" (note the '-' and '_' where '+' and '/'
→would appear)

encode_base64url("");

## ""
```

Edge cases

- Empty input returns the empty string.
- Bytes that would encode to `+` under standard base64 become `-`; bytes that would encode to `/` become `_`.
- No `=` appears in the output under any input length.

Differences from upstream

Fully compatible with upstream MIME::Base64 3.16.

See also

- `decode_base64url` - the inverse operation.
- `encode_base64` - standard variant with wrapping and padding.

decode_base64url

Decode a URL-safe base64 string, accepting input with or without trailing padding.

Synopsis

```
use MIME::Base64 qw(decode_base64url);
my $bytes = decode_base64url($token);
```

What you get back

The decoded byte string. `-` and `_` in the input are mapped back to `+` and `/` before decoding, and any missing trailing padding is reconstructed. The decoder inherits `decode_base64`'s tolerance: characters outside the alphabet are silently skipped, and decoding stops at the first padding group.

Examples

```
decode_base64url("QWxhZGRpbjpvcGVuIHNlc2FtZQ");
## "Aladdin:open sesame"

decode_base64url("-_8");
## "\xFB\xFF"

decode_base64url("QWxhZGRpbjpvcGVuIHNlc2FtZQ=="); # padding also_
→OK
## "Aladdin:open sesame"
```

Edge cases

- Empty input returns the empty string.
- Input with stray `+` or `/` is accepted (those are valid under standard base64); the URL-safe decoder is a superset of the standard decoder for input characters.
- Truncated input (length not a multiple of four; no padding) is padded internally and decoded; the last partial group follows the same “1 char → 0 bytes, 2 → 1, 3 → 2” rule as `decode_base64`.

Differences from upstream

Fully compatible with upstream MIME::Base64 3.16.

See also

- `encode_base64url` - the inverse operation.
- `decode_base64` - standard variant.

encoded_base64_length

Return the length `encode_base64` would produce, without doing the encoding.

Synopsis

```
use MIME::Base64 qw(encoded_base64_length);
my $n = encoded_base64_length($bytes);      # wrapping with "\n"
my $m = encoded_base64_length($bytes, "");  # no wrapping
my $k = encoded_base64_length($bytes, "\r\n"); # CRLF wrapping
```

What you get back

An integer: the exact number of bytes `encode_base64($bytes, $eol)` would produce, including every end-of-line sequence. Equivalent to `length(encode_base64($bytes, $eol))` but avoids allocating and building the encoded string - preferred when you need to pre-size a buffer or report progress on a large payload.

Examples

```
encoded_base64_length("");           # 0
encoded_base64_length("A");          # 5  ("QQ==\n")
encoded_base64_length("A", "");      # 4  ("QQ==")
encoded_base64_length("A" x 57);      # 77  (one full 76-char_
↪line + "\n")
encoded_base64_length("A" x 57, "\r\n"); # 78  (one full 76-char_
↪line + "\r\n")
```

Edge cases

- Empty input returns 0 regardless of the `$eol` argument.
- The formula is `ceil(n/3) * 4 + lines * length($eol)` where `lines = ceil(base_len / 76)`.
- An undefined or missing `$eol` counts as `"\n"` (length 1) to match `encode_base64`'s default.

Differences from upstream

Fully compatible with upstream MIME::Base64 3.16.

See also

- `encode_base64` - the function whose output length is being computed.
- `decoded_base64_length` - the companion predictor for the decoder.

decoded_base64_length

Return the length `decode_base64` would produce, without doing the decoding.

Synopsis

```
use MIME::Base64 qw(decoded_base64_length);  
my $n = decoded_base64_length($encoded);
```

What you get back

An integer: the exact number of bytes `decode_base64($encoded)` would produce. Equivalent to `length(decode_base64($encoded))` but avoids building the decoded string. Useful when validating that a payload fits a size limit, or pre-sizing a destination buffer.

The same leniency rules as `decode_base64` apply: stray whitespace and non-alphabet characters are ignored, and the count stops at the first = padding group.

Examples

```
decoded_base64_length(""); # 0  
decoded_base64_length("QQ=="); # 1  
decoded_base64_length("QWxhZGRpbjpvcGVuIHNlc2FtZQ=="); # 19  
decoded_base64_length("QWxh\nZGRp\nbjpv\ncGVu\n"); # 12 (newlines  
→ ignored)
```

Edge cases

- Empty input, or input with no alphabet characters, returns 0.
- A lone trailing character that would decode to no output (a group of exactly 1 valid character) contributes 0 bytes.
- Characters after the first = are not counted.

Differences from upstream

Fully compatible with upstream MIME : : Base64 3.16.

See also

- `decode_base64` - the function whose output length is being computed.
- `encoded_base64_length` - the companion predictor for the encoder.

MIME::QuotedPrint

Quoted-printable encoding and decoding as specified by RFC 2045.

Quoted-printable is the encoding of choice for mail bodies that are *mostly* ASCII with the occasional non-ASCII byte - long passages of plain text stay readable, and only the unsafe bytes turn into `=XX` hex triplets. For content that is truly binary, prefer base64 via MIME : : Base64.

The encoder keeps lines at most 76 characters long by inserting soft line breaks (`=` at end of line, then the `$eol` sequence). Pass a `$binmode` flag when you want `\n` itself to be encoded - useful when the decoded output must be byte-for-byte identical regardless of the receiver's line-ending conventions. An `$eol` of `" "` disables both soft breaks and `\n` rewriting.

The decoder always produces `\n`-terminated output, even when the input used `\r\n` line endings.

Functions

Other Functions

`encode_qp`

Encode a string as quoted-printable.

`decode_qp`

Decode a quoted-printable string back to plain bytes.

`encode_qp`

Encode a string as quoted-printable.

Synopsis

```
$encoded = encode_qp($str);  
$encoded = encode_qp($str, $eol);  
$encoded = encode_qp($str, $eol, $binmode);
```

What you get back

A scalar holding the quoted-printable encoding of `$str`. Printable ASCII bytes (plus `\t`) pass through unchanged; every other byte, and `=` itself, becomes a `=XX` triplet with uppercase hex digits. Lines are kept at most 76 characters by inserting soft breaks: a trailing `=` followed by the `$eol` sequence.

`$eol` defaults to `"\n"`. Pass `"\015\012"` to produce output suitable for direct transmission over the wire. Pass `" "` to suppress soft breaks and force binary-mode behavior for `\n`.

`$binmode`, when true, encodes `\n` and `\r` as `=0A/ =0D` rather than passing them through as line terminators. Use this when the decoded form must match the input byte for byte on every platform.

Examples

```
use MIME::QuotedPrint;
print encode_qp("Hello, world!\n");      # Hello, world!\n
print encode_qp("caf\xe9\n");            # caf=E9\n
print encode_qp("\x00\x01\x02", "", 1);  # =00=01=02
print encode_qp("x" x 80 . "\n");        # wraps at col 75 with =\n
↳ soft break
```

Edge cases

- Empty `$str` returns the empty string.
- `$eol` of `" "` disables soft line breaks and encodes every `\n`.
- Trailing spaces and tabs on a line are always encoded so they survive transports that strip trailing whitespace.

Differences from upstream

Fully compatible with upstream `MIME::QuotedPrint 3.16`.

See also

- `decode_qp` - reverse the encoding.
- `MIME::Base64::encode_base64` - better choice for fully binary data.

decode_qp

Decode a quoted-printable string back to plain bytes.

Synopsis

```
$decoded = decode_qp($encoded);
```

What you get back

A scalar holding the decoded byte string. Every =XX triplet is replaced by the byte whose hex value is XX; soft line breaks (a = immediately before \r\n or \n) are removed.

Line terminators are normalized: whether the input used \n or \r\n, each line in the result ends with \n.

Examples

```

use MIME::QuotedPrint;
print decode_qp("Hello, world!\n");      # Hello, world!\n
print decode_qp("caf=E9\n");             # café (Latin-1 byte E9)
print decode_qp("line =\r\ncontinued");  # line continued
print decode_qp("=00=01=02");            # three bytes: 00 01 02

```

Edge cases

- A stray = not followed by two hex digits or a line break is passed through literally, matching the upstream decoder.
- Trailing whitespace on a decoded line is stripped, matching the encoder's soft-break contract.
- Lowercase hex digits are accepted even though the encoder only emits uppercase.

Differences from upstream

Fully compatible with upstream MIME::QuotedPrint 3.16.

See also

- `encode_qp` - produce the encoded form.
- `MIME::Base64::decode_base64` - the counterpart for binary payloads.

5.2.18 Math

Namespace containing the following modules:

- `Math::GMP` — Arbitrary-precision integer arithmetic backed by GNU libgmp.

Math::GMP

Arbitrary-precision integer arithmetic backed by GNU libgmp.

`Math::GMP` lets you work with integers of unlimited size. The interpreter's native integer tops out somewhere around 2^{63} ; pass anything bigger through a `Math::GMP` object and every arithmetic operator keeps working, up to the memory you have.

Build a value with `Math::GMP->new`, then use it like a normal number - the module overloads `+` `-` `*` `/` `%` `**` `<=>` `==` `&` `|` `^` `<<` `>>` so `$x * $y` and `$x ** 1000` Do The Right Thing. Only the final stringification crosses back into ordinary Perl:

```
use Math::GMP;
my $x = Math::GMP->new('2');
my $huge = $x ** 1024;                # a 309-digit integer
print "$huge\n";
```

A `Math::GMP` value is a blessed reference whose underlying scalar holds a pointer to a GMP `mpz_t`. Passing one to `ref` returns `"Math::GMP"`; passing it to a stringifying or numifying operator goes through the overloaded `op_stringify` / `op_numify`.

The `:constant` pragma extends the overloading to bare integer literals in your source:

```
use Math::GMP qw(:constant);
my $n = 2 ** (256 * 1024);           # evaluated in GMP, not
    ↪ doubles
```

Without `:constant` the exponent `256 * 1024` would be computed as a C double and overflow to infinity before `Math::GMP` ever sees it.

Division by zero

`libgmp` raises `SIGFPE` on division by zero instead of returning a Perl-level error. On Perl 5.36 and later you can trap it with a `$SIG{FPE}` handler; on older Perls use `POSIX::sigaction`. Returning from the handler re-raises the signal, so use it for diagnostics (`Carp::confess`) rather than recovery.

Functions

Construction

`const_integer`

Wrap an integer literal in a `Math::GMP` object.

`new_from_scalar`

Build a `Math::GMP` object from a decimal/auto-detected string.

`new_from_scalar_with_base`

Build a `Math::GMP` object from a string in the given base.

`destroy`

Release the `mpz_t` backing a `Math::GMP` object.

`gmp_copy`

Return an independent copy of `$x`.

Conversion

stringify

Return the decimal string representation of a `Math::GMP` value.

get_str_gmp

Return the string representation of `$x` in the given base.

sizeinbase_gmp

Return the number of digits `$x` would take in the given base.

uintify

Return the low bits of `$x` as a plain unsigned Perl integer.

intify

Return `$x` as a plain signed Perl integer.

Arithmetic

add_ui_gmp

Add an unsigned integer to `$x` in place.

op_add

Overloaded `+` - return `$m + $n` as a new `Math::GMP`.

op_sub

Overloaded `-` - return `$m - $n` as a new `Math::GMP`.

op_mul

Overloaded `*` - return `$m * $n` as a new `Math::GMP`.

bmulf

Return `$x` multiplied by a floating-point value, truncated to an integer.

op_div

Overloaded `/` - integer division `int($m / $n)`, truncated toward zero.

bdiv

Return the quotient and remainder of m / n as a two-element list.

op_mod

Overloaded % - remainder of m / n , sign follows the dividend.

op_pow

Overloaded ** - return m raised to the integer power n .

Comparison

op_spaceship

Overloaded $<=>$ - signed comparison returning -1, 0, or 1.

op_eq

Overloaded $==$ - return 1 when m equals n , else 0.

Bitwise

mul_2exp_gmp

Return x shifted left by e bits ($x * 2^{**e}$).

Synopsis

```
Math::GMP->new(0b1011)->mul_2exp_gmp(4);    # 0b10110000
```

Equivalent to $x \ll e$ on a `Math::GMP` value; the operator form goes through `blshift`. e is a plain non-negative integer.

div_2exp_gmp

Return x shifted right by e bits (`int( $x / 2^{**e}$ )`).

Synopsis

```
Math::GMP->new(200)->div_2exp_gmp(2);        # 50
```

Truncates toward zero, matching `libgmp`'s `mpz_tdiv_q_2exp`. Use `brshift` for the overloaded-operator spelling.

mod_2exp_gmp

Return the low cnt bits of x ($x \bmod 2^{**cnt}$).

Synopsis

```
Math::GMP->new(0b10001011)->mod_2exp_gmp(4);    # 0b1011
```

Does not modify \$x. The sign of the result matches \$x - this is truncated (tdiv) rather than Euclidean (mmod) semantics.

band

Bit-wise AND of \$m and \$n.

Synopsis

```
Math::GMP->new(6)->band(3);    # 0b110 & 0b011 = 0b010 = 2
```

Overload hook for &. The optional third argument (swap) is accepted and ignored.

bxor

Bit-wise XOR of \$m and \$n.

bior

Bit-wise inclusive OR of \$m and \$n.

blshift

Return \$m << \$n - left shift by \$n bits.

brshift

Return \$m >> \$n - right shift by \$n bits, truncating toward zero.

Synopsis

```
Math::GMP->new(0b11001)->brshift(3);    # 0b11 = 3
```

Overload hook for >>. Uses truncated division, so negative values shift toward zero rather than toward minus infinity.

gmp_tstbit

Return bit \$n of \$x (0-indexed, from the least significant end).

Number theory

powm_gmp

Compute \$base ** \$exp mod \$mod without building the full power.

mmod_gmp

Return the non-negative remainder of a / b .

bmodinv

Return the modular inverse of $x \bmod y$.

legendre

Return the Legendre symbol (m / n) as -1, 0, or 1.

jacobi

Return the Jacobi symbol (m / n) as -1, 0, or 1.

probab_prime

Probabilistic primality test: return 0, 1, or 2.

bgcd

Return the greatest common divisor of m and n .

blcm

Return the least common multiple of m and n .

fibonacci

Return the n -th Fibonacci number.

bfac

Return $n!$ - the factorial of a non-negative integer.

bnok

Return the binomial coefficient $n \text{ choose } k$.

Roots and powers

broot

Return the integer n -th root of x , truncated toward zero.

brootrem

Return the integer n -th root of x and the shortfall.

bsqrt

Return the integer square root of x .

bsqrtrem

Return the integer square root of x and the shortfall.

Synopsis

```
my ($r, $q) = Math::GMP->new(7)->bsqrtrem;    # 2, 3
**because 2**2 + 3 == 7**
```

The returned list satisfies $r^2 + q = x$.

is_perfect_power

Return 1 when x is a perfect power a^b with $b > 1$.

is_perfect_square

Return 1 when x is the square of an integer, else 0.

Synopsis

```
Math::GMP->new(49)->is_perfect_square;    # 1
Math::GMP->new(50)->is_perfect_square;    # 0
```

Overload glue

nomethod

Fallback stub for overloaded operations that have no handler.

op_stringify

Overloaded "" - stringify a `Math::GMP` as its decimal digits.

op_numify

Overloaded 0+ - convert a `Math::GMP` to a plain Perl number.

op_bool

Overloaded bool - true when \$x is non-zero.

Library info

gmp_lib_version

Return the version of libgmp currently loaded into the process.

Other Functions

new

Build a Math::GMP object from a string or integer scalar.

new

Build a Math::GMP object from a string or integer scalar.

Synopsis

```
my $x = Math::GMP->new(123);
my $y = Math::GMP->new('123456789012345678901234567890');
my $z = Math::GMP->new('abcd', 36);           # base 36
```

What you get back

A blessed reference of class Math::GMP. You can feed it to any overloaded operator, or to the named methods on this page.

The first argument is read as a string. A leading + is stripped, and embedded spaces and underscores are removed - so '1_000_000' and ' 1 000 000 ' both parse. When no base is given the string must match /^-?[\dxA-Fa-f]+\$/, which admits the usual 0x-prefixed hex form; anything else croaks with Argument to Math::GMP->new is not a string representing an integer. With an explicit base (2–36) the usual GMP digit-set applies.

Edge cases

- Missing or false first argument → Math::GMP of 0.
- Floating-point input is not special-cased here; you get whatever libgmp's parser makes of the stringified value.
- Base 0 (the default) means “guess from the prefix”: 0x → hex, 0 → octal, otherwise decimal.

gmp_lib_version

Return the version of libgmp currently loaded into the process.

Synopsis

```
my $v = Math::GMP::_gmp_lib_version();  
print "running libgmp $v\n";    # e.g. "running libgmp 6.3.0"
```

What you get back

A dotted-decimal string like "6.3.0". Feed it to `version->parse` if you need numeric comparison.

Differences from upstream

Upstream returns a `vstring`. This implementation returns a plain string; both compare the same way under `version->parse` and under `string ge / le`.

get_str_gmp

Return the string representation of `$x` in the given base.

Synopsis

```
my $x = Math::GMP->new(3 * 7 + 2 * 7**2 + 6 * 7**3);  
print $x->get_str_gmp(7);    # "6230"  
print $x->get_str_gmp(16);   # hex
```

Base is 2–36 for lowercase digits or -2 through -36 for uppercase, matching libgmp's `mpz_get_str` convention.

sizeinbase_gmp

Return the number of digits `$x` would take in the given base.

Synopsis

```
Math::GMP->new(10**100)->sizeinbase_gmp(10);    # 101  
Math::GMP->new(255)->sizeinbase_gmp(2);        # 8
```

The count may be off by one on the high side - this is libgmp's documented behaviour for `mpz_sizeinbase`, which trades accuracy for speed. Use it for buffer sizing, not for exact digit counts.

intify

Return `$x` as a plain signed Perl integer.

Synopsis

```
my $n = Math::GMP->new(-42)->intify;    # -42
```

Truncates when `$x` does not fit in a platform signed long. For values outside that range use `get_str_gmp` or `stringify` to preserve precision.

add_ui_gmp

Add an unsigned integer to `$x` in place.

Synopsis

```
my $x = Math::GMP->new(1_000_000);
$x->add_ui_gmp(7);
print $x;                # 1000007
```

Mutates `$x` and returns nothing. The second argument must be a non-negative plain Perl integer. Useful in tight counting loops where allocating a new `Math::GMP` per += would dominate the cost.

powm_gmp

Compute `$base ** $exp mod $mod` without building the full power.

Synopsis

```
my $base = Math::GMP->new(157);
my $exp  = Math::GMP->new(100);
my $mod  = Math::GMP->new(5013);
my $r    = $base->powm_gmp($exp, $mod);    # 157**100 mod 5013
```

The workhorse of RSA-style modular exponentiation: fast enough for thousand-bit exponents, and never materialises the intermediate `$base ** $exp`. Behaviour is undefined when `$mod` is 0.

mmod_gmp

Return the non-negative remainder of `$a / $b`.

Synopsis

```
Math::GMP->new(-7)->mmod_gmp(3);    # 2, not -1
```

Unlike `%` on `Math::GMP` (which uses truncated division and follows the sign of the dividend), `mmod_gmp` always returns a result in $[0, \text{abs}(\$b))$. This matches the mathematical “mod” operation used in number-theory work.

op_numify

Overloaded `0+` - convert a `Math::GMP` to a plain Perl number.

Invoked when a `Math::GMP` is used in numeric context outside of the overloaded arithmetic operators (for example, passed to a function that does `$arg + 0`). Precision is limited to what an IV or UV can hold; for large values the top bits are lost. Call `intify` or `uintify` if you want to be explicit about the cast, or `stringify` if you want to keep all the digits.

bmulf

Return `$x` multiplied by a floating-point value, truncated to an integer.

Synopsis

```
my $x = Math::GMP->new(3)->bpow(100);
my $y = $x->bmulf(1.5);           # floor(3**100 * 1.5)
```

The double `$float` is converted to a GMP float, multiplied, then truncated toward zero. Internal precision is raised just enough to hold both operands exactly; the default mpf precision is restored before returning.

Edge cases

- `$float == 0.0` → a `Math::GMP` of 0.
- Negative `$float` → result has the expected sign.
- NaN / infinity → result is implementation-defined; do not rely on it.

bdiv

Return the quotient and remainder of `$m / $n` as a two-element list.

Synopsis

```
my $x = Math::GMP->new(7);
my ($q, $r) = $x->bdiv(3);      # $q = 2, $r = 1
```

Division truncates toward zero, and the remainder has the same sign as the dividend - the `tdiv_qr` convention. For non-negative remainders, combine with `mmod_gmp`.

bmodinv

Return the modular inverse of $x \bmod y$.

Synopsis

```
my $x = Math::GMP->new(5);  
my $inv = $x->bmodinv(7);      # 3, because 5 * 3 == 1 (mod 7)
```

When x has no inverse modulo y (i.e. $\gcd(x, y) \neq 1$), returns 0 . Behaviour is undefined when y is 0 .

Edge cases

- `bmodinv` returning 0 is ambiguous: 0 is never a valid inverse, so treat it as “no inverse exists”.

legendre

Return the Legendre symbol (m / n) as -1 , 0 , or 1 .

Synopsis

```
Math::GMP->new(6)->legendre(Math::GMP->new(11));  # -1, 0, or 1
```

Defined only when n is an odd prime. When n is not an odd prime the result is currently 0 , but do not rely on that - upstream documents this as “may change in the future”.

probab_prime

Probabilistic primality test: return 0 , 1 , or 2 .

Synopsis

```
my $x = Math::GMP->new(7);  
my $v = $x->probab_prime(10);  # 2 - definitely prime
```

What you get back

- 0 - definitely composite.
- 1 - probably prime (no witness found in $\$reps$ rounds).
- 2 - definitely prime (small numbers, or `libgmp` proved it).

$\$reps$ is the Miller-Rabin round count; higher values cost more but tighten the confidence on a 1 answer. A round count of 25 is a common default for cryptographic work.

bgcd

Return the greatest common divisor of \$m and \$n.

Synopsis

```
my $x = Math::GMP->new(6);  
print $x->bgcd(4);           # 2
```

Also registered as gcd. Result is always non-negative; bgcd(0, 0) is 0.

blcm

Return the least common multiple of \$m and \$n.

Synopsis

```
my $x = Math::GMP->new(6);  
print $x->blcm(4);           # 12
```

Also registered as lcm. Result is always non-negative; blcm(\$x, 0) is 0.

fibonacci

Return the \$n-th Fibonacci number.

Synopsis

```
my $f = Math::GMP::fibonacci(100); # 354224848179261915075
```

Called as a function, not a method. \$n is a plain Perl non-negative integer. libgmp's recursive doubling algorithm handles indices in the millions in fractions of a second.

blshift

Return \$m << \$n - left shift by \$n bits.

Synopsis

```
Math::GMP->new(0b11)->blshift(4); # 0b110000 = 48
```

Overload hook for <<. \$n is taken modulo the word size of an unsigned long because it is passed through mpz_get_ui; if you are shifting by bit-widths that large, reconsider what you are doing.

bfac

Return $n!$ - the factorial of a non-negative integer.

Synopsis

```
my $x = Math::GMP->new(5);  
print $x->bfac;           # 1*2*3*4*5 = 120
```

The receiver is read as an unsigned long - factorials larger than that range are out of reach of a single call. In practice this is rarely limiting: libgmp struggles with memory long before LONG_MAX enters the picture.

bnok

Return the binomial coefficient n choose k .

Synopsis

```
my $x = Math::GMP->new(5);  
print $x->bnok(2);        # 10
```

Equivalent to $n! / (k! * (n - k)!)$, but computed without building the factorials - suitable for counting problems where n is large.

gmp_copy

Return an independent copy of x .

Because `Math::GMP` values are references to heap-allocated `mpz_t` structures, two Perl variables can share the same underlying integer. For the overloaded arithmetic operators that does not matter - they always allocate a fresh result. If you are about to call an in-place mutator (currently just `add_ui_gmp`) and want to keep the original intact, copy first.

gmp_tstbit

Return bit n of x (0-indexed, from the least significant end).

Synopsis

```
my $x = Math::GMP->new(0b1010);  
print $x->gmp_tstbit(1);    # 1  
print $x->gmp_tstbit(2);    # 0
```

For negative values, libgmp uses a two's-complement view. Bits above the most significant set bit of a positive value read as 0.

broot

Return the integer n -th root of x , truncated toward zero.

Synopsis

```
my $x = Math::GMP->new(100);  
print $x->broot(3);           # 4, since 4**3 = 64 and 5**3 = 125
```

n must be a positive integer. For odd n negative values of x are fine; for even n they croak inside libgmp.

brootrem

Return the integer n -th root of x and the shortfall.

Synopsis

```
my $x = Math::GMP->new(100);  
my ($root, $rem) = $x->brootrem(3);    # 4, 36  
  
## because 4**3 + 36 == 100
```

The returned list satisfies $\text{\$root} ** n + \text{\$rem} == x$ exactly. Use when you need both pieces in one pass - calling broot and computing the remainder yourself doubles the work.

Differences from upstream

Replicates upstream's explicit workaround for libgmp versions earlier than 5.1.0, where mpz_rootrem returned wrong results for negative arguments with odd n . Modern libgmp builds skip the workaround automatically.

bsqrt

Return the integer square root of x .

Synopsis

```
Math::GMP->new(6)->bsqrt;    # 2 - int(sqrt(6))  
Math::GMP->new(49)->bsqrt;    # 7
```

Equivalent to broot(2) but noticeably faster. Croaks on negative arguments inside libgmp.

is_perfect_power

Return 1 when \$x is a perfect power \$a ** \$b with \$b > 1.

Synopsis

```
Math::GMP->new(27)->is_perfect_power;    # 1 (3**3)
Math::GMP->new(28)->is_perfect_power;    # 0
```

Returns 0 otherwise. Detects any exponent - squares, cubes, fifth powers, and so on.

5.2.19 PDL

PDL - the Perl Data Language: fast, vectorised, multidimensional arrays for scientific and bulk numeric work.

use PDL boots Core, Ops, Ufunc, Math, Primitive, Slices, Basic and exports the standard PDL functions into the caller's namespace.

Option B Architecture

Every PDL object in Perl stores a `*mut c_pdl` (the C-repr struct from `core_vtable`) directly in `PERL_MAGIC_ext` on the blessed SV. This mirrors how upstream PDL does it (`pdl_SetSV_PDL / pdl_SvPDLV` from `pdlcore.c`).

No `rust_pdl::Pdl` intermediary. No copy bridge. The same pointer that lives in the SV is passed directly to `pdl_run_*` functions.

Child submodules declared here (leaf-with-children pattern, same as `crate::native::Peta::FFI`) because PDL has both its own `P5_ENTRY` and 14 sub-namespace modules. Rust does not allow `#[path]` combined with an inline block.

Modules

- `PDL::Bad` — `PDL::Bad` - bad value (missing data) support.
- `PDL::Basic` — `PDL::Basic` - basic constructors and utility functions.
- `PDL::Constants` — `PDL::Constants` - mathematical and physical constants.
- `PDL::Core` — `PDL::Core` - constructors, accessors, type system, DESTROY.
- `PDL::FFT` — `PDL::FFT` - Fast Fourier Transform operations.
- `PDL::Lite` — `PDL::Lite` - minimum PDL module OO loader.
- `PDL::LiteF` — `PDL::LiteF` - `PDL::Lite` + `Basic` + `Primitive`.
- `PDL::Math` — `PDL::Math` - mathematical functions (sin, cos, exp, log, etc.).
- `PDL::MatrixOps` — `PDL::MatrixOps` - matrix operations (det, inv, eigens, etc.).
- `PDL::Ops` — `PDL::Ops` - element-wise arithmetic, comparison, and logic operations.
- `PDL::Primitive` — `PDL::Primitive` - fundamental operations (inner, outer, matmult, etc.).

- `PDL::Slices` — `PDL::Slices` - indexing, slicing, and data rearrangement.
- `PDL::Types` — `PDL::Types` - PDL type system (byte, short, long, float, double, etc.).
- `PDL::Ufunc` — Unary reductions and aggregations over PDL arrays.

Functions

Other Functions

`type_convert_func`

Shared XS function for all type constructors (byte, short, float, etc.)

`PDL::Bad`

`PDL::Bad` - bad value (missing data) support.

Synopsis

```
use PDL;
my $a = pdl [1, 2, 3];
$a->badflag(1);
$a->set(1, $a->badvalue);
print $a->nbad, "\n";      # 1
print $a->ngood, "\n";    # 2
```

Functions

Other Functions

`setvaltobad`

`setvaltobad($pdl, $val)` - set elements equal to `$val` as bad. Upstream Bad. pd:625. Same signature as `setbadtoval`: `pdl_run_setvaltobad(a, b, value)`. Engine restored 2026-04-19. Honours `PDL_INPLACE` so `$x->inplace->setvaltobad(1)` mutates `$x`.

`setbadtoval`

`setbadtoval($pdl, $val)` - replace bad values with `$val`.

`badmask`

`badmask($pdl, $replacement)` - replace bad values with `$replacement`.

`setbadat`

`setbadat($pdl, @indices)` - set element at given indices to the bad value and enable the badflag.

setinfetobad

setinfetobad(\$pdl) - set Inf/-Inf elements to bad value, enable badflag. Returns a new pdl (or modifies inplace if inplace flag set).

setnonfinitetobad

setnonfinitetobad(\$pdl) - set NaN and Inf/-Inf elements to bad value, enable badflag. Returns a new pdl (or modifies inplace if inplace flag set). Combines setnantobad and setinfetobad behavior.

isfinite

isfinite(\$pdl) - returns a PDL of 1s for finite values, 0s for NaN/Inf/bad. Mirrors PDL::Bad::isfinite.

PDL::Basic

PDL::Basic - basic constructors and utility functions.

Provides sequence generators (sequence, xvals, yvals, zvals, axisvals), coordinate helpers (ndcoords), and hist.

These are mostly pure Perl in the original PDL::Basic.pm. We provide native implementations here for performance and correctness.

Synopsis

```
use PDL;
my $a = sequence(10);      # 0..9
my $x = xvals(5, 5);
my $y = yvals(5, 5);
my $z = zvals(3, 3, 3);
my $h = hist($a, 0, 10, 2);
```

Functions

Other Functions

sequence

sequence(@dims) – create an ndarray filled with 0, 1, 2, ..., N-1.

xvals

xvals(@dims)

yvals

yvals(@dims)

zvals

zvals(@dims)

axisvals

axisvals(\$pdl_or_dims, \$axis)

hist

hist(\$pdl, \$min?, \$max?, \$step?)

ndcoords

ndcoords(@dims)

xlinvals

xlinvals(\$pdl, \$min, \$max)

ylinvals

ylinvals(\$pdl, \$min, \$max)

zlinvals

zlinvals(\$pdl, \$min, \$max)

xlogvals

xlogvals(\$pdl, \$min, \$max)

ylogvals

ylogvals(\$pdl, \$min, \$max)

zlogvals

zlogvals(\$pdl, \$min, \$max)

rvals

rvals(@dims) - radial distance values. Creates a PDL where each element is the Euclidean distance from the center. For dims (W, H): center = (W/2, H/2), rvals[x,y] = $\sqrt{(x-cx)^2 + (y-cy)^2}$. Options (last arg hashref): {center=>[x,y,...], squared=>1}

random

`grandom(@dims)` - Gaussian random numbers using Box-Muller transform.
`random([type], $nx, $ny, $nz, ...)` - uniform random PDL in [0, 1). Mirrors `PDL::Basic::random` (`Primitive.pd` wrapper).

sequence

`sequence(@dims)` – create an ndarray filled with `0, 1, 2, ..., N-1`.

The returned PDL has shape `@dims` and, stored in row-major order, contains the arithmetic progression `0 .. prod(@dims)-1`. Element type defaults to double; an optional leading `PDL::Type` blessed argument requests another type (e.g. long, byte).

Also supported as an instance method: `$pdl->sequence` returns a new PDL with the same dims and element type as `$pdl`, filled with the progression `0 .. nelem-1`.

Signature

```
$w = sequence(@dims);           # double, shape @dims
$w = sequence($type, @dims);    # typed form
$w = sequence($template_pdl);   # shape and type borrowed
$w = $template_pdl->sequence;    # instance-method form
```

Broadcasting

Not applicable: `sequence` is a constructor. The fill pattern always depends solely on the flat element index.

Bad values

The result is freshly allocated with `BADFLAG` clear; it contains no bad cells. If a template PDL is passed, only its dimensions and element type are inherited – the badflag is not copied.

Examples

```
use PDL;
print sequence(5);           # [0 1 2 3 4]
print sequence(3, 2);        # [[0 1 2] [3 4 5]]

## Typed form

my $b = sequence(byte, 4);    # 1-byte elements, values [0 1 2 3]
print $b->type;               # "byte"

## Instance-method form: reshape/retype from existing PDL

my $tmpl = zeroes(long, 2, 3);
my $seq  = $tmpl->sequence;    # same dims (2,3), type long, values
```

(continues on next page)

(continued from previous page)

```

→0..5

## Empty-dim edge case

print sequence(0)->nelem;      # 0 (empty PDL, but valid)

```

Errors

Croaks with "PDL::sequence: allocation failed" if the backing `pdl_from_f64` allocation returns null (e.g. out-of-memory, or an overflowing dim product).

Performance notes

The progression is built in a `Vec<f64>` before being handed to `pdl_from_f64`. For very large PDLs this temporarily doubles the memory footprint versus an in-place fill; an in-place `axisvals2($pdl->flat->inplace, 0, ...)` path (as in the pure-Perl upstream) is a future optimisation.

Upstream

Mirrors `PDL::sequence` in `lib/PDL/Basic.pm`, which is itself pure Perl wrapping `_construct + axisvals2`.

PDL::Constants

`PDL::Constants` - mathematical and physical constants.

Mirrors `PDL::Constants.pm`: exports numeric constants as constant subs.

Synopsis

```

use PDL::Constants qw(PI E);
my $circle = 2 * PI * $r;
my $growth = E ** $x;

```

PDL::Core

`PDL::Core` - constructors, accessors, type system, DESTROY.

Option B architecture: every PDL SV stores a `*mut c_pdl` in `PERL_MAGIC_ext`. All functions operate directly on `c_pdl` pointers via `core_vtable` - no `rust_pdl`.

Functions

Other Functions

`pdl_new`

`new($class?, @data)`

null

`null()` - create a null (typeless, dimensionless) PDL placeholder.

zeroes

`zeroes($class?, TYPE?, @dims)` - create a zero-filled PDL.

ones

`ones($class?, TYPE?, @dims)` - create a PDL filled with 1.

destroy

No-op destructor; magic vtable `svt_free` handles `pdl_destroy`.

get_dateref

`get_dateref($pdl)` - return a SCALAR ref to a byte buffer aliasing the pdl's data. Mirrors `PDL::Core::get_dateref` from `Core.xs:1141-1156`.

upd_data

`upd_data($pdl, $keep_datasv=0)` - Perl wrote into the dateref; propagate back. Mirrors `PDL::Core::upd_data` from `Core.xs:1158-1182`.

update_data_from

`update_data_from($pdl, $sv)` - overwrite pdl buffer from an SV of bytes. Mirrors `PDL::Core::update_data_from` from `Core.xs:1184-1197`.

shape

`shape($pdl)` - return the ndarray's dim-vector as a 1-D PDL of `indx`.

ndims

`ndims($pdl)` - return the number of dimensions. Also aliased as `getndims()`. Mirrors `Core.xs:getndims` - calls `pdl_make_physdims` first.

nelem

`nelem($pdl)` - return the total number of elements. Upstream `Core.xs:nelem` does `make_physvaffine` (not just `physdims`) for reasons unclear - we mirror exactly.

at

`$pdl->at(@pos)` – return a single element of an ndarray as a Perl scalar.

set

`set($pdl, @indices, $value)` - set the element at the given indices.

list

`list($pdl)` - return all elements as a flat Perl list. Upstream `Core.xs:list` calls `pdl_make_physvaffine` before iterating.

unpdl

`unpdl($pdl)` - return nested arrayrefs matching the dim structure. e.g. `pdl([[1,2],[3,4]])->unpdl` returns `[[1,2],[3,4]]`. Mirrors `Core.xs:unpdl` - calls `pdl_make_physvaffine` first.

type

`type($pdl)` - return the PDL datatype as a string. `type($pdl)` - returns a `PDL::Type` blessed arrayref `[$enum]` Mirrors upstream: returns `PDL::Type` object that stringifies to lowercase name.

convert

`convert($pdl, $typecode)` - convert to a different datatype.

inplace

`inplace($pdl)` - set the inplace flag. Croaks if called as a class method (`PDL->inplace`) instead of on an object.

is_inplace

`is_inplace($pdl)` - check whether the inplace flag is set.

badflag

`badflag($pdl, $flag?)` - get or set the bad-value flag.

string

`string($pdl)` - return the PDL's string representation.

barf

`barf(@msg)` - croak with the given error message.

copy

`copy($pdl)` - return a deep copy via `pdl_hard_copy`.

sclr

`sclr($pdl)` - return the first element as a plain Perl scalar.

flowing

`flowing($pdl)` - no-op stub, returns self. `flowing($pdl)` - enable forward-flow dataflow on this pdl. Sets `PDL_DATAFLOW_F` on the state flags. Upstream `Core.pm` semantics: trans created from this pdl retain their link so parent changes cascade to children.

squeeze

`squeeze($pdl)` - remove size-1 dimensions.

info

`info($pdl)` - return a diagnostic string.

sever

`sever($pdl)` - break dataflow (no-op for physical pdls), return self.

cat

`cat($class?, @pdl)` - concatenate PDLs along a new trailing dimension.

dog

`dog($pdl)` - split along last dimension into a list of sub-PDLs.

empty

`empty($class?, TYPE?)` - create an empty PDL with dimension [0].

dim

`dim($pdl, $n)` - return the size of dimension `$n`.

dimincs

`dimincs($pdl)` - return the list of stride increments.

isnull

`isnull($pdl)` - return 1 if the PDL is a null placeholder.

isempty

`isempty($pdl)` - return 1 if the PDL has zero elements.

is_readonly

`is_readonly($pdl)` - return whether the PDL is marked read-only.

set_readonly_on

`readonly($pdl)` - SET the read-only flag and return self. Upstream PDL: `$pdl->readonly` always sets the flag (no args needed). Distinct from `is_readonly()` which only reads the flag.

set_readonly

`set_readonly($pdl, $flag)` - set or clear the read-only flag.

make_physical

`make_physical($pdl)` - no-op for physical pdls, return self.

check_badflag

`nbad($pdl)` - count of bad values. Uses `effective_badvalue_f64` to mirror upstream's `has_badvalue` ? `pdl->badvalue` : `PDL.bvals[dtype]` lookup (`pdlcore.c:631 BADTRANS` macro). `check_badflag($pdl)` - clear the badflag if no bad values are present. Mirrors PDL : : `check_badflag` (`Bad.pm:262-266`). Returns the final badflag.

ngood

`ngood($pdl)` - count of good (non-bad) values.

nbadover_core

`nbadover($pdl)` - per-row bad-value count (reduces along first dimension).

ngoodover_core

`ngoodover($pdl)` - per-row good-value count.

set_badvalue

`set_badvalue($pdl, $val)` - set the bad-value sentinel.

get_badvalue

`get_badvalue($pdl)` - return the current bad-value sentinel.

set_datatype

`set_datatype($pdl, $typecode)` - set the ndarray's datatype. Mirrors Core.xs:997-1001 → `pdlapi.c:1022 pdl_set_datatype`. For a NOMYDIMS (fresh-from-null) pdl, just assigns the datatype; otherwise converts the existing data. Used by `PDL::IO::Storable`'s thaw path.

set_donttouchdata

`set_donttouchdata($pdl, $size=-1)` - mark data as externally owned. Mirrors Core.xs:481-489. Sets `PDL_DONTTOUCHDATA|PDL_ALLOCATED` and optionally overrides `nbytes`. Used by `set_data_by_file_map` so that `pdl_destroy` won't free() the mmap'd region.

setdims

`setdims($pdl, \@dims)` - set the ndarray's dimensions in-place. Mirrors Core.xs:1300-1305 → `pdl_setdims()`. The `dims` argument is an arrayref of integers.

set_sv_to_null_pdl

`set_sv_to_null_pdl($sv)` - connect an existing SV to a freshly-allocated empty pdl. Mirrors Core.xs:1073-1082 → `pdlcore.c:34 pdl_SetSV_PDL`.

howbig

`howbig($typecode)` - size in bytes of a PDL type, by integer type code.

get_datatype

`get_datatype($pdl)` - return the datatype as integer type code.

orig_badvalue

`orig_badvalue` - class method on `PDL::Type`. Returns hardcoded `pdl.h` default. Call form: `byte->orig_badvalue` returns `u8::MAX` as IV/NV.

badvalue

badvalue - dual dispatch mirror of PDL::badvalue (Bad.pm:177-194). PDL instance: `$pdl->badvalue([$new])` → get/set per-PDL `(*pdl).badvalue` PDL::Type: `byte->badvalue([$new])` → get/set global per-type table

approx

approx(\$a, \$b, \$tol?) - element-wise approximate equality. Returns a PDL of 0/1 (default tolerance 1e-6).

inf

inf() - return positive infinity as a 0-d Double PDL.

nan

nan() - return NaN as a 0-d Double PDL.

r2c

r2C(\$pdl) - convert real to complex (stub: croak).

i2c

i2C(\$pdl) - convert imaginary to complex (stub: croak).

broadcast

broadcast(\$pdl, \$dim?) - return a broadcasted view. Broadcasting happens automatically in PDL binary operations, so this is a no-op that returns self. Mirrors PDL::Core::broadcast.

nbytes

nbytes(\$pdl) - return the number of bytes in the pdl's data buffer.

unbroadcast

unbroadcast(\$pdl, \$n?) - remove up to \$n trailing size-1 dimensions. With no \$n argument, removes all trailing size-1 dims. Returns a new PDL with the reduced dimensionality.

sethdr

sethdr(\$pdl, \$hashref) - attach a header hashref to the PDL.

gethdr

gethdr(\$pdl) - retrieve the header hashref from the PDL.

hdr

hdr(\$pdl, \$hashref?) - get or set header (combined accessor).

hdrcpy

hdrcpy(\$pdl, \$flag?) - get/set whether header is copied on operations. Stores the flag as a package variable in %PDL : : _hdrcpy keyed by SV address.

fhdr

fhdr(\$pdl) - “force header” = get header, creating an empty one if none exists.

new_or_inplace

new_or_inplace(\$pdl) - if inplace flag is set, return the pdl itself (clearing the inplace flag); otherwise return a hard copy. Mirrors PDL : : Core : : new_or_inplace.

dup

dup(\$pdl, \$dim, \$n) - duplicate along dimension \$dim, \$n times. Output dim[\$dim] = input dim[\$dim] * \$n. Data is repeated.

dupn

dupN(\$pdl, @counts) - duplicate along each dimension by the corresponding count.

datasv_refcount

datasv_refcount(\$pdl) - return SvREFCNT of the pdl’s datasv slot.

allocated

allocated(\$pdl) - return 1 iff PDL_ALLOCATED is set on state flags.

at

\$pdl->at(@pos) – return a single element of an ndarray as a Perl scalar.

@pos is a coordinate list whose length matches the number of dimensions of \$pdl. Indices may be negative (counted from the far end, so -1 is the last element along that axis). Missing trailing indices default to 0; extra trailing indices past \$pdl->ndims are ignored.

Before the lookup, the PDL is made physical-or-affine (pdl_make_physvaffine), and if any upstream transform has been invalidated (PDL_ANYCHANGED) a full pdl_make_physical is run so the data we read is current. The returned SV has the

same Perl-level numeric type as the underlying PDL cell (IV for integer types, NV for floats); complex-typed PDLs yield a `PDL::Complex::Overloads` object upstream – the connector currently returns the real part via `pdl_anyval_to_f64`.

Signature

```
$z = at($pdl, @pos);      # function form
$z = $pdl->at(@pos);      # method form
```

Broadcasting

`at` does not broadcast. It is a scalar-valued accessor: supply exactly one coordinate tuple.

Bad values

If `$pdl` has the `BADFLAG` set and the fetched cell is the bad sentinel, `at` returns the string "BAD" (matches upstream `at_bad_c` and test 170-bad.t #54). If `BADFLAG` is clear the raw numeric value is returned regardless of content.

Examples

```
use PDL;
my $x = sequence(3, 4);      # 3x4, values 0..11
print $x->at(1, 2);          # 7
print $x->at(-1, -1);        # 11 (last element via negative
    ↪ indexing)

## BAD value propagation

my $b = pdl([1, 2, 3]);
$b->badflag(1);
$b->inplace->setbadat(1);
print $b->at(1);             # "BAD"
```

Errors

Croaks with "Position out of range" when:

- `$pdl` is a null PDL (NOMYDIMS, e.g. fresh from `PDL->null`);
- `$pdl` has zero elements (e.g. `pdl([ ])`);
- any supplied index, after negative-index resolution, falls outside `[0, dim_size)` for its axis.

Also croaks via `croak_c_pdl_error` if the engine's `pdl_make_physvaffine` / `pdl_make_physical` reports an error.

Upstream

Mirrors `PDL::at` in `lib/PDL/Core.pm`, which delegates to the XS routine `at_bad_c` defined in `Core.xs:at_c`. Our flat-index computation is done connector-side via `pdl_indices_to_flat` rather than going back through `pdl_at` for each call.

flowing

`flowing($pdl)` - no-op stub, returns self. `flowing($pdl)` - enable forward-flow dataflow on this pdl. Sets `PDL_DATAFLOW_F` on the state flags. Upstream `Core.pm` semantics: trans created from this pdl retain their link so parent changes cascade to children.

NOTE (Phase A status): we mark the flag and the slice/affine forward-flow path works (set $\rightarrow$ `pdl_mark_children_dirty`). Full trans-retention for non-affine transforms (e.g. `$a + $a`) is the remaining Phase A work - without it, `$pb = $pa + $pa; $pa ->set(...)` does not refresh `$pb` since the `+` trans was freed after its one-shot execution.

set_sv_to_null_pdl

`set_sv_to_null_pdl($sv)` - connect an existing SV to a freshly-allocated empty pdl. Mirrors `Core.xs:1073-1082` $\rightarrow$ `pdlcore.c:34 pdl_SetSV_PDL`.

Storable's thaw path receives a blessed-scalar-ref stub from Storable core and needs to reshape it into a real PDL object. Upstream (`Core.xs:1077-1082`):

```
pdl *it = pdl_pdlnew();
sv_setiv(SvRV(sv), PTR2IV(it)); // inner becomes PTR-IV
it->sv = SvRV(sv);              // link back
pdl_SetSV_PDL(sv, it);          // inner has it->sv set, so
                                // pdl_SetSV_PDL takes the
                                // `else` branch: create a
                                // new RV pointing at it->sv,
                                // SvAMAGIC_on it,
                                // sv_setsv(sv, newref),
                                // SvREFCNT_dec(newref).
```

howbig

`howbig($typecode)` - size in bytes of a PDL type, by integer type code.

Mirrors upstream `PDL::Core::howbig` (PP-generated from `Core.pd:pp_def('howbig', GenericTypes => \@all, Pars => 'int a(); int [o]b();', ...)`). Free function, callable as `PDL::Core::howbig(10)` and re-exported via `*PDL::howbig`. `PDL::Types.pm:556` also calls this to back the `$type->howbig` method.

PDL::FFT

PDL::FFT - Fast Fourier Transform operations.

Synopsis

```
use PDL;
use PDL::FFT;
my $re = sequence(8);
my $im = zeroes(8);
fft($re, $im);
ifft($re, $im);
```

Functions

Other Functions

fft

fft(\$re, \$im)

ifft

ifft(\$re, \$im)

realfft

realfft(\$re)

realifft

realifft(\$re, \$im)

PDL::Lite

PDL::Lite - minimum PDL module OO loader.

Mirrors PDL/Lite.pm: boots Core + Ops + Primitive + Ufunc + Basic + Slices + Bad; exports only the “always exported” symbols from PDL::Core (pdl, piddle, null, barf) - not the full convenience set. Upstream line: our @EXPORT = qw(piddle pdl null barf);

PDL::LiteF

PDL::LiteF - PDL::Lite + Basic + Primitive.

Mirrors PDL::LiteF.pm: boots everything PDL::Lite does, plus Basic and Primitive for full function coverage.

PDL::Math

PDL::Math - mathematical functions (sin, cos, exp, log, etc.).

Synopsis

```
use PDL;
my $a = pdl [0, 1, 2, 3];
my $s = sin($a);
my $c = cos($a);
my $e = exp($a);
```

Provides element-wise transcendental, hyperbolic, rounding, error, and Bessel functions operating on PDL ndarrays. Each function accepts a single PDL (or plain scalar) and returns a new PDL with the operation applied element-wise.

Functions

Other Functions

polyval

polyval(\$c, \$x) - evaluate polynomial with coeffs c at x. Binop: (c, x) → y. Signature from Math.pd upstream.

polyroots

polyroots(\$cr, \$ci) - roots of a polynomial with complex coefficients. Signature: (cr, ci) → (rr, ri). Returns two PDLs in list context.

pow

pow(\$a, \$b) → \$a ** \$b, element-wise. Binop wrapper around pdl_run_pow. Mirrors Ops.rs's binop_xs_c2rust pattern but lives here because pow is declared in Math.pd upstream.

PDL::MatrixOps

PDL::MatrixOps - matrix operations (det, inv, eigens, etc.).

Synopsis

```
use PDL;
use PDL::MatrixOps;
my $m = pdl [[1,2],[3,4]];
my $d = det($m);
my $inv = inv($m);
```

Functions

Other Functions

identity

`identity($n_or_pdl)` - create an NxN identity matrix. If given a PDL, uses its first dim; if given a scalar, uses it directly.

trace

`trace($matrix)` - sum of diagonal elements of a square matrix

norm

`norm($matrix)` - Frobenius norm of a matrix

det

`det($matrix)` - determinant of a square matrix (via LU decomposition). Supports broadcasting: if input has dims [N,N,...], computes determinant for each NxN slice, returning a PDL of shape [...].

inv

`inv($matrix [, {s=>1}])` - inverse of a square matrix (via Gauss-Jordan elimination). With {s=>1} option, returns undef instead of croaking on singular matrix.

lu_decomp

`lu_decomp($matrix)` - LU decomposition, returns (\$LU, \$perm, \$sign)

lu_backsub

`lu_backsub($LU, $perm, $b)` - solve $Ax=b$ from LU decomposition

simq

`simq($a, $b)` - solve simultaneous linear equations $Ax=b$

eigens_sym

`eigens_sym($matrix)` - eigenvalues and eigenvectors of a real symmetric matrix. Uses the Jacobi eigenvalue algorithm. Returns (\$eigenvectors, \$eigenvalues).

eigens

`eigens($matrix)` - eigenvalues and eigenvectors of a general real matrix. For symmetric matrices delegates to `eigens_sym` logic. For non-symmetric, uses QR algorithm (simplified). Returns (\$eigenvectors, \$eigenvalues).

stretcher

`stretcher($diag_pdl)` - create diagonal matrix(es) from a PDL. 1D input [a,b] → 2x2 diagonal. 2D input [N,M] → NxNxM broadcasted diagonals.

svd

`svd($matrix)` - Singular Value Decomposition. Returns (\$U, \$S, \$V) where `$matrix = $U * diag($S) * $V^T`. Uses one-sided Jacobi SVD algorithm.

PDL::Ops

PDL::Ops - element-wise arithmetic, comparison, and logic operations.

Mirrors PDL::Ops (generated from Ops.pd in upstream PDL). Binary ops take two PDL args plus an optional swap flag (from operator overloading). Unary ops take a single PDL arg.

Synopsis

```
use PDL;
my $a = pdl [1, 2, 3];
my $b = pdl [4, 5, 6];
my $c = $a + $b;           # plus
my $d = $a * $b;           # mult
my $e = $a ** $b;          # power
my $cmp = $a > $b;         # gt (returns 0/1 piddle)
my $s = sqrt($a);          # element-wise sqrt
my $n = -$a;               # neg
```

Functions

Other Functions

neg

`neg($pdl)`

string_overload

`xs_string_overload($pdl)`

matmult_overload

`xs_matmult_overload($pdl, $other, $swap)`

numify_overload

`xs_numify_overload($pdl)`

assgn

`xs_assgn($dest, $src, $swap)`

incr_overload

`xs_incr_overload($pdl)`

decr_overload

`xs_decr_overload($pdl)`

bool_overload

`xs_bool_overload($pdl)`

copy_overload

`xs_copy_overload($pdl)`

nomethod

`xs_nomethod()`

bool_overload

`xs_bool_overload($pdl)`

Boolean-context overload (`if $pdl, while $pdl, etc.`). Mirrors the `bool => closure` in `Core.pm: 838-845` of upstream PDL:

- null PDL → false
- multielement PDL → croak “multielement ndarray in conditional ...”
- bad-valued 1-elem → croak “bad value ndarray in conditional ...”
- 1-elem → truthiness of the single value

PDL::Primitive

PDL::Primitive - fundamental operations (inner, outer, matmult, etc.).

Functions

Other Functions

inner

`inner($a, $b)` – inner (dot) product along the first dimension.

matmult

`matmult($a, $b [, $c])` - matrix multiplication.

append

`append($a, $b)` - concatenate two PDLs along dimension 0. Handles null PDL inputs: `append(null, null) → empty`, `append(null, x) → x`, etc.

interpol

`interpol($xi, $x, $y)` - linear interpolation. For each query point in `$xi`, find the corresponding y-value in the reference dataset (`$x, $y`). Mirrors upstream `PDL::Primitive::interpol(Primitive.pd:3010)` which delegates to `interpolate($xi, $x, $y, $yi)`.

indexND

`indexND($data, $indices)` - index into data using N-D coordinate columns.

axisvalues

`axisvalues($pdl)` - fill PDL with index values along dim 0. Mirrors `PDL::Primitive pp_def 'axisvalues' Code => 'loop(n) %{ $a() = n; %}'` Supports inplace: `axisvalues($pdl->inplace)` modifies `$pdl` directly.

indadd

`indadd($input, $ind, $sum)` – broadcasting index-add.

uniqind

`uniqind($pdl)` - return flat indices of unique elements. Mirrors `PDL::uniqind` from `Primitive.pm:727-756`.

fibonacci

`fibonacci($n_or_pdl)` – Fibonacci sequence constructor.

inner

`inner($a, $b)` – inner (dot) product along the first dimension.

Given two ndarrays with matching first-axis length `n`, computes `c = sum_i a_i * b_i` as a 0-dim PDL. For 1-D inputs this is the standard dot product; for higher-rank inputs the contraction is over axis 0 and the remaining axes broadcast.

Signature

```
$c = inner($a, $b);      # function form
$c = $a->inner($b);      # method form
```

Upstream also accepts a 3-arg form `inner($a, $b, $c)` writing into an output PDL `$c`; that form is not yet wired through the connector shim – supply only two arguments.

Broadcasting

Upstream `inner` has signature `(a(n); b(n); [o]c())` and broadcasts over all axes beyond the contracted first axis. The connector shim currently implements the 1-D / whole-array case: it multiplies `$a * $b` element-wise (via the engine's `pdl_run_mult`, which *does* broadcast) and then reduces the product with `sumover`. For genuinely 1-D inputs this matches upstream; for multi-axis inputs the result is a full reduction over *all* axes, not the per-row inner product upstream gives. See the performance note below.

Bad values

Bad-value handling is inherited from `pdl_run_mult` and `pdl_run_sumover`: products containing a BAD operand propagate BAD, then `sumover` skips BAD cells. If every product is BAD the result PDL is BAD-flagged; otherwise it is clean. Matches the `=for bad` stanza in upstream `PDL::Primitive::inner`.

Examples

```
use PDL;
my $a = pdl([1, 2, 3]);
my $b = pdl([4, 5, 6]);
print inner($a, $b);      # 32    (1*4 + 2*5 + 3*6)
print $a->inner($b);      # 32    (method form)

## Scalar-on-vector also works (scalar broadcasts to each element)

print inner(pdl([1, 2, 3]), 2); # 12    (1*2 + 2*2 + 3*2)
```

Errors

Croaks via `croak_c_pdl_error` if either the `mult` or the `sumover` engine step returns an error (e.g. dimension mismatch inside `mult`). Returns `undef` when either input cannot be coerced to a PDL or scalar.

Performance notes

Connector-side shim pending engine delivery. PDL's PP-generated `inner` routine (`Primitive_pp_inner`) was not picked up by the `c2rust` transpile pass (the `readdata` codegen template is not supported), so until the engine ships a native `pdl_run_inner` we emulate it in two steps – one `mult` allocation and one `sumover` reduction. Cost is one full pass over the product buffer plus its allocation; no SIMD fused multiply-add is used. This will be replaced by a direct call once the engine delivers the op.

Upstream

Mirrors `PDL::Primitive::inner` in `lib/PDL/Primitive.pm` (generated from `lib/PDL/Primitive.pd`); upstream signature `(a(n); b(n); [o]c())`.

indadd

`indadd($input, $ind, $sum)` – broadcasting index-add.

Scatter-accumulates values from `$input` into `$sum` at positions given by `$ind`: `sum[ind[n]] += input[n]`. Mirrors upstream `PDL::indadd(Primitive.pd pp_def 'indadd', Pars => 'input(n); indx ind(n); [io] sum(m)')`.

Delegates to the engine's `pdl_run_indadd(input, ind, sum)`. The engine handles type dispatch, broadcasting, bad-value propagation (bad `ind` croaks; bad input marks the target element bad), and out-of-range index detection.

uniqind

`uniqind($pdl)` - return flat indices of unique elements. Mirrors `PDL::uniqind` from `Primitive.pm:727-756`.

Semantics:

1. Flatten input.
2. NaN indices go at the end of the result (NaN is never equal to itself).
3. Among non-NaN values: sort by value (stable → first-occurrence wins), find positions where sorted values change (rotate-minus-one diff), map those back through the sort permutation to original indices.
4. Empty input → returns input unchanged.

fibonacci

`fibonacci($n_or_pdl)` – Fibonacci sequence constructor.

Mirrors upstream `PDL::fibonacci(Primitive.pd pp_def 'fibonacci', Pars => 'i(n); [o]x(n)')`. The engine op `pdl_run_fibonacci(i, x)` writes `x[k] = x1+x2` with `x1=1, x2=0` initially and the standard carry `{ x2=x1; x1=x[k] }` from `k>0`, producing `1, 1, 2, 3, 5, 8, 13, ...`

Two call forms, mirroring upstream's `PDL::fibonacci` wrapper:

- `fibonacci(N)` - scalar integer → N-element pdl
- `$pdl->fibonacci` - fills `$pdl` in-place (inplace flag)

The connector prepares the `i` input pdl (a sequence of counter values, shape-match the desired output) and lets the engine do the actual recurrence.

PDL::Slices

PDL::Slices - indexing, slicing, and data rearrangement.

Synopsis

```
use PDL;  
my $a = sequence(10);  
my $s = $a->slice("2:5");  
my $i = $a->index(pdl [1, 3, 5]);  
my $d = $a->dice([2, 4, 6]);
```

Functions

Other Functions

slice

`$pdl->slice($spec)` – extract a rectangular slice of an ndarray.

xchg

`xchg($pdl, $dim1, $dim2)`

mv

`mv($pdl, $from, $to)`

reorder

`reorder($pdl, @dims)`

reshape

`reshape($pdl, @new_dims)`

flat

`flat($pdl)`

diagonal

`diagonal($pdl, $dim1, $dim2)`

dice

`dice($pdl, @index_lists)`

dice_axis

`dice_axis($pdl, $dim, $indices_pdl)`

uniq

`uniq($pdl)`

uniquevec

`uniquevec($pdl)`

hclip

`hclip($pdl, $max)`

lclip

`lclip($pdl, $min)`

clump

`clump($pdl, $n)`

dummy

`dummy($pdl, $dim, $size?)`

lags

`lags($pdl, $nthdim, $step, $nlags)`

splitdim

`splitdim($pdl, $dim, $size)`

rotate

`rotate($pdl, $shift)`

mslice

`mslice($pdl, @args)` - slice variant where each arg is a scalar (start, truncated to int) or [start, stop, inc?] arrayref. Mirrors `Core.pm:2545 *PDL::mslice = \&PDL::slice` combined with the numeric-arg parsing in `PDL::NiceSlice's mslice` translation.

range

`range($source, $index, $size?, $boundary?)` - extract elements (or rectangular sub-cubes with `$size`) at N-D coordinates given by `$index`.

make_physdims

`make_physdims($pdl)` - no-op stub. Upstream triggers deferred trans evaluation; our slices materialize eagerly, so nothing to do except return self.

transpose_entry

Public forwarder for `PDL::t` (MatrixOps alias).

transpose

`transpose($pdl)` - transpose a 2D matrix (swap dims 0 and 1). Same as `$pdl->xchg(0,1)`.

rle

`rle($pdl)` - run-length encoding. Returns `($lengths, $values)` - two PDLs: run lengths and corresponding values. For 2D+ input, operates over the first dimension (broadcasting).

rld

`rld($lengths, $values)` - run-length decoding (inverse of `rle`). Returns a PDL reconstructed from run lengths and values.

ins

`ins($target, $source, @pos)` - insert `$source` into `$target` at position `@pos`. Copies source data into target starting at the given coordinates. Modifies `$target` in-place, returns `$target`.

sec

`sec($pdl, $x1, $x2, $y1, $y2, ...)` - extract a rectangular section. Deprecated upstream; equivalent to `$pdl->slice("$x1:$x2,$y1:$y2,...")`. Takes pairs of (start, end) indices for each dimension.

slice

`$pdl->slice($spec)` - extract a rectangular slice of an ndarray.

`slice` is the main dimension-manipulation primitive in PDL. The `spec` argument describes, axis by axis, which elements of `$pdl` to keep. The resulting child PDL is connected to its parent by dataflow: it normally does not copy storage, and writes into the child propagate back into the parent.

Signature

```
$child = slice($parent, $spec);      # function form
$child = $parent->slice($spec);      # method form
$child = $parent->slice("1:3", "*2"); # one string per dim
```

Slice spec syntax

Per-axis specifiers, comma-separated if packed into a single string:

Spec	Meaning
" " / : / X	keep axis as-is
n	a single index (keeps the axis, length 1)
(n)	a single index, <i>squish</i> (axis removed)
a:b	inclusive range a..b
a:b:s	range with stride s (negative reverses)
-n	count from the end: -1 is the last element
*n	insert a dummy (broadcast) axis of size n

Fewer specs than \$parent->ndims leaves the trailing axes intact.

Broadcasting

slice itself does not broadcast – it structurally rewrites axes. The *n spec *creates* a new broadcast axis of size n with stride 0, i.e. a cheap virtual repeat useful as an input to later broadcasting ops.

Bad values

BADFLAG and bad-value storage are preserved from the parent: a slice of a BAD-flagged PDL is itself BAD-flagged, and the bad sentinel value is inherited. No bad-specific checks happen inside slice.

Examples

```
use PDL;
my $x = sequence(10);           # [0 1 2 3 4 5 6 7 8 9]
print $x->slice("2:5");          # [2 3 4 5]
print $x->slice("0:-1:2");       # [0 2 4 6 8] (stride 2)
print $x->slice("-1:0");         # [9 8 7 6 5 4 3 2 1 0]
    ↪reversed

my $m = sequence(4, 3);         # 4 cols, 3 rows
print $m->slice(":",1);          # row 1: [4 5 6 7]
print $m->slice("(2),:");        # col 2, axis squished: [2 6
    ↪10]
```

(continues on next page)

(continued from previous page)

```
## Dataflow: writes propagate

my $row = $m->slice(":",1);
$row .= -1;
print $m;                                # row 1 of $m is now all -1

## Dummy dim for broadcasting a row-vector against a matrix

my $v = pdl([10, 20, 30, 40]);
my $mat = $v->slice(":",*3);              # 4x3 view, each row equal to
↪$v
```

Errors

Croaks with:

- "slice: cannot slice a null PDL" when \$parent has PDL_NOMYDIMS set.
- "PDL::slice: slice index 'a:b' out of bounds for dim N of size M" when an index, after negative-index resolution, does not land in [0, dim_size) (test 180-slicing-full.t #11).
- whatever parse_slice_spec reports for malformed spec strings.

Upstream

Mirrors PDL::Slices::slice in lib/PDL/Slices.pm (generated from lib/PDL/Slices.pd), which calls the XS pdl_slice_args_parse helper from pdlutil.c. Parsing happens connector-side in parse_slice_spec; the engine is invoked for the actual view construction.

PDL::Types

PDL::Types - PDL type system (byte, short, long, float, double, etc.).

Synopsis

```
use PDL::Types;
my $type = PDL::Type->new(6);    # double
print $type->symbol, "\n";        # PDL_D
print $type->realctype, "\n";    # double
my $a = zeroes(double, 10);
```

Mirrors PDL::Types.pm: exports type-name subs (e.g. byte, double) that return PDL type codes, plus uppercase aliases (PDL_B, PDL_D, etc.). Type codes match PdlDatatype discriminant values exactly.

Functions

Other Functions

type_stringify

PDL::Type stringify: returns lowercase type name. PDL::Type is bless [\$enum], 'PDL::Type'.

type_new

PDL::Type->new(\$code_or_name) - constructor. Handles: new(\$class, \$code), new(\$class, "double"), new(\$class, \$existing_type). If the argument is already a PDL::Type blessed ref, pass it through.

types_list

PDL::Types::types() - return the list of all registered PDL::Type objects, ordered by numval. Mirrors PDL::Types.pm: my @CACHED_TYPES = map bless([\$_->{numval}, \$_], 'PDL::Type'), @HASHES; sub types { @CACHED_TYPES } Heavily used by Test::PDL (line 417) to iterate over available types when building per-type comparison subs; without it, every upstream test that uses Test::PDL fails at compile time.

type_eq

PDL::Type eq comparison - stringify both sides and compare

type_ne

PDL::Type ne comparison

type_spaceship

PDL::Type <=> numeric comparison by enum/numval. Mirrors upstream PDL::Types.pm:578: "<=>" => sub { \$_[2]? \$_[1][0] <=> \$_[0][0] : \$_[0][0] <=> \$_[1][0] }, where \$_[2] is the swap flag.

type_cmp

PDL::Type cmp string comparison. Mirrors Types.pm:575-577.

type_nomethod

PDL::Type overload marker (nomethod)

type_numify

PDL::Type numification (0+) - returns the type code integer

type_howbig

`PDL::Type::howbig` - returns byte size of the type.

type_enum

`enum($type)` - numval, the slot-0 integer. Types.pm:531.

type_badvalue

`PDL::Type::badvalue` - get or set the bad value for the type.

type_orig_badvalue

`PDL::Type::orig_badvalue` - returns the ORIGINAL (hardcoded) bad value, ignoring any user-set override. Mirrors Bad.pd's *default_badvalue_int* (pd1.h's *PDL_BADVAL** constants). Note: this is NOT the same as `xs_type_badvalue`, which reflects mutations made via `byte->badvalue($v)`.

type_codederef

`PDL::Type &{} overload` - allows calling a `PDL::Type` as a code reference. `byte->()` returns 'byte' (the type name string).

PDL::Ufunc

Unary reductions and aggregations over PDL arrays. This module holds every operation that collapses one or more dimensions of a piddle down to a summary - sums, products, means, extrema, percentiles, sort-based reductions, and cumulative variants.

These are the first functions most users reach for. They behave the same on 1-D arrays as on higher-rank arrays, and they all share a consistent story about bad values, type promotion, and when they reduce everything vs. just one axis.

Synopsis

```
use PDL;
my $a = pdl [1, 2, 3, 4, 5];
print $a->sum, "\n";      # 15
print $a->avg, "\n";      # 3
print $a->min, "\n";      # 1
print $a->max, "\n";      # 5
```

Functions

Full-array reductions

sum

Add up every element of a PDL and return the total as a 0-dimensional PDL.

Other Functions

qsortvec

`qsortvec($pdl)` - sort rows of a 2D pdl lexicographically, return sorted data.

qsortveci

`qsortveci($pdl)` - sort rows of a 2D pdl lexicographically, return index piddle.

minimum_n_ind

`minimum_n_ind($pdl, $n)` - 3-arg c2rust: (a, c, m_size)

maximum_n_ind

`maximum_n_ind($pdl, $n)` - 3-arg c2rust: (a, c, m_size)

pctover

`pctover($pdl, $pct)` - 3-arg c2rust: (a, p, b) where p is pct as a scalar pdl
`pctover($pdl, $percentile)` - Excel/NIST linear-interpolation percentile. Mirrors `Ufunc.pd pp_def('pctover')` algorithm: $\text{rank} = 1 + p \cdot (n-1)$; interp between `sorted[floor(rank)-1]` and `sorted[ceil(rank)-1]`. The c2rust version produces rounding errors on the boundary ($17.0000000007 \neq 17$); this pure-Rust path avoids the double→double→double chain and is bit-exact when the interpolation happens on aligned indices.

all

`all($pdl)` - 1 if all elements nonzero, via full andover reduction

any

`any($pdl)` - 1 if any element nonzero, via full orover reduction

minmax

`minmax($pdl)` - return a 2-element list: (min_value, max_value). Iterates all elements via `pdl_at_flat`, tracking min and max.

pct

`pct($pdl, $p)` - percentile, delegates to `pctover` then extracts scalar. Returns a 0-d PDL scalar.

oddpct

`oddpct($pdl, $p)` - odd percentile (nearest-rank method). Sort the data, return element at index $\text{floor}((n-1) * p)$. Returns a 0-d PDL scalar.

stats

`stats($pdl)` - compute 7 summary statistics. Returns (\$mean, \$prms, \$median, \$min, \$max, \$adev, \$rms).

sum

Add up every element of a PDL and return the total as a 0-dimensional PDL.

Calling it

```
my $s = sum($pdl);    # function form
my $s = $pdl->sum;    # method form
```

Both forms accept any PDL, regardless of shape. The input is flattened and every element is added; nothing is broadcast. For per-row or per-column sums, use `sumover` (see *See also* below).

What you get back

A 0-dimensional PDL - think of it as “a PDL holding one number”. It prints as a single number, compares numerically, and works inside PDL arithmetic. If you want a plain Perl scalar, pull it out with `sclr`:

```
my $n = $pdl->sum->sclr;    # ordinary Perl number
```

The element type of the result is promoted so the total can’t silently overflow:

Input type	Result type
signed integer	indx
unsigned integer	ulonglong
float / double	same
complex	same

This matches the “int+” promotion rule of the underlying engine. If you specifically want a double-precision result from an integer input, call `dsum` instead.

Examples

Basic sum of a 1-D piddle:

```
my $x = pdl(1, 2, 3, 4, 5);
print $x->sum;           # 15
```

Sum of a 2-D piddle - every element, not row- or column-wise:

```
my $m = sequence(3, 4);    # 3x4 matrix, values 0..11
print $m->sum;              # 66
```

Floating-point accumulation follows IEEE-754 rules, so adding many small numbers to a large one can lose precision. For a more accurate sum of many floats, sort ascending before summing or use `dsum` to accumulate in double precision:

```
my $x = float(1e8, 1, 1, 1);
print $x->sum;           # 100000000    - the three 1s got lost
print $x->dsum;          # 100000003    - accumulates in double
```

Bad values

If the input has its bad-value flag set, elements marked BAD are skipped:

```
my $b = pdl(1, 2, 3);
$b->badflag(1);
$b->inplace->setbadat(1); # mark the middle element BAD
print $b->sum;           # 4
```

If every element is BAD, the returned 0-dim PDL is itself marked BAD and stringifies as "BAD".

Edge cases

- **Empty PDL** (any dim of size 0) - returns 0, marked BAD if the input has its bad-value flag set.
- **undef argument** - returns a PDL holding 0. (Upstream PDL croaks here; see *Differences from upstream PDL* below.)
- **Non-PDL scalar** (plain number or arrayref) - promoted to a PDL first, then summed. `sum(5)` returns 5; `sum([1,2,3])` returns 6.

Differences from upstream PDL

- `sum(undef)` returns 0 rather than croaking with `Can't call method "flat" on an undefined value`. Scripts that rely on the croak for error detection need an explicit defined check. Covered by `t/81-xs-native/PDL/030-reductions.t`.

Otherwise behavior matches upstream PDL 2.103.

See also

- `sumover` - sum along the first dimension only
- `dsum` - same as `sum`, accumulating in double precision
- `avg` - arithmetic mean of all elements
- `prod` - product of all elements
- `stats` - mean, variance, min, max, median in one call

5.2.20 POSIX

The POSIX (IEEE Std 1003.1) C standard library, exposed to Perl.

Reach for POSIX when you need something the Perl core does not give you directly: a `strftime` format, `setlocale` to switch numeric conventions, `sigaction` for signal handling, wait-status macros like `WIFEXITED` / `WEXITSTATUS`, `mkfifo`, or any of the hundreds of `E*` / `O*` / `F*` / `S*` constants from `<errno.h>`, `<fcntl.h>`, and friends.

The module groups its surface into the categories below; each one becomes a section on the rendered POSIX page. Every category is backed by `libc` calls, so behaviour matches the host's C library exactly (`glibc` / `musl` on Linux).

Functions

Math

`abs`

Integer absolute value of `$n`.

`frexp`

Split a floating-point number into mantissa and exponent.

`modf`

Split a number into integer and fractional parts, both as doubles.

`ldexp`

Compute `$x * 2**$exp` with a single rounding step.

Synopsis

```
my $y = POSIX::ldexp(1.5, 10);    # 1536
```

Differences from upstream

Fully compatible with upstream POSIX.

`scalbn`

Compute `$x * FLT_RADIX**$n` efficiently. On IEEE-754 systems `scalbn` is equivalent to `ldexp`.

Differences from upstream

Fully compatible with upstream POSIX.

fma

Fused multiply-add: $x * y + z$ with a single rounding.

nan

Build a quiet NaN, optionally with a payload string.

remquo

Floating-point remainder with partial quotient.

jn

Bessel function of the first kind, integer order n .

yn

Bessel function of the second kind, integer order n .

Float classification

fpclassify

Classify a floating-point number.

ilogb

Exponent of x as an integer, $\text{trunc}(\log_2(|x|))$.

Edge cases

- $\text{ilogb}(0)$ returns FP_ILOGB0 (typically INT_MIN).
- $\text{ilogb}(\text{NaN})$ returns FP_ILOGBNAN .
- $\text{ilogb}(\text{inf})$ returns INT_MAX .

Differences from upstream

Fully compatible with upstream POSIX.

lrint

Round x to the nearest integer, respecting the current rounding mode, and return it as an integer.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `lround` - always rounds half-away-from-zero regardless of mode.
- `fegetround` / `fesetround` - query or change the rounding mode.

lround

Round x half-away-from-zero to the nearest integer.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `lrint` - rounds according to the current FP rounding mode.

fegetround

Current floating-point rounding direction.

What you get back

An integer matching one of the `FE_TONEAREST`, `FE_DOWNWARD`, `FE_UPWARD`, `FE_TOWARDZERO` constants.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `fesetround` - change the mode.

fesetround

Change the floating-point rounding direction.

Wait status

wexitstatus

Exit code of a child that terminated normally.

wifexited

True if the child terminated normally (via `exit` or return from `main`).

Differences from upstream

Fully compatible with upstream POSIX.

wifsignaled

True if the child was killed by a signal.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `WTERMSIG` - the signal number, when this is true.

wifstopped

True if the child is stopped (usable with `waitpid(..., WUNTRACED)`).

Differences from upstream

Fully compatible with upstream POSIX.

wstopsig

Signal number that stopped the child, meaningful only when `WIFSTOPPED($status)` is true.

Differences from upstream

Fully compatible with upstream POSIX.

wtermsig

Signal number that killed the child, meaningful only when `WIFSIGNALED($status)` is true.

Differences from upstream

Fully compatible with upstream POSIX.

Strings and printing

strtod

Parse a double from the start of a string, locale-aware.

strtol

Parse a signed integer from the start of a string, in a given base.

strtoul

Parse an unsigned integer from the start of a string, in a given base.

strcoll

Compare two strings using the current `LC_COLLATE` ordering.

What you get back

A negative, zero, or positive integer, suitable for use as a `cmp`-like sort return. The ordering depends on the locale set via `setlocale(LC_COLLATE, ...)`.

Differences from upstream

Fully compatible with upstream POSIX.

strerror

Message string for an errno value.

strstr

Byte offset of the first occurrence of \$needle in \$haystack, or -1 if not found.

Edge cases

- Empty \$needle: returns 0 (matches the empty substring at the start), matching C strstr.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- Built-in index - the Perl-native way to do this.

sprintf

Format a string POSIX-style, identical to built-in sprintf.

System and process

getpid

Process ID of the current process.

Differences from upstream

Fully compatible with upstream POSIX.

getppid

Process ID of the parent process.

Differences from upstream

Fully compatible with upstream POSIX.

getuid

Real user ID of the current process.

Differences from upstream

Fully compatible with upstream POSIX.

geteuid

Effective user ID of the current process.

Differences from upstream

Fully compatible with upstream POSIX.

getgid

Real group ID of the current process.

Differences from upstream

Fully compatible with upstream POSIX.

getegid

Effective group ID of the current process.

Differences from upstream

Fully compatible with upstream POSIX.

setuid

Change the real user ID of the current process.

What you get back

1 on success, 0 on failure (and \$! is set). Permission is required; only privileged processes can switch to arbitrary UIDs.

Differences from upstream

Fully compatible with upstream POSIX.

setgid

Change the real group ID of the current process.

What you get back

1 on success, 0 on failure.

Differences from upstream

Fully compatible with upstream POSIX.

setpgid

Put process \$pid into process group \$pgid.

setsid

Create a new session and become its leader.

What you get back

The new session ID on success, -1 on failure. The caller must not already be a process group leader.

Differences from upstream

Fully compatible with upstream POSIX.

uname

Kernel and machine identification as a five-element list.

sysconf

Runtime system configuration value for the given `_SC_*` name.

getpgrp

Process group ID of the current process.

Differences from upstream

Fully compatible with upstream POSIX.

getlogin

Login name of the user running the process, or `undef` if not available (e.g. no controlling terminal).

Differences from upstream

Fully compatible with upstream POSIX.

getenv

Value of environment variable `$name`, or `undef` if it is unset.

Synopsis

```
my $path = POSIX::getenv( "PATH" );
```

Differences from upstream

Fully compatible with upstream POSIX. Prefer `$ENV{$name}` in ordinary Perl code.

exit

Terminate the process immediately with the given status, skipping END blocks, destructors, and stdio flushing.

Synopsis

```
POSIX::_exit(0);
```

Differences from upstream

Fully compatible with upstream POSIX. Note: exported as `_exit`, not `exit`, to avoid colliding with the core built-in.

abort

Abort the process abnormally (typically producing a core dump).

Differences from upstream

Fully compatible with upstream POSIX.

nice

Adjust the current process's scheduling niceness by `$incr`.

pause

Suspend the process until any signal is delivered.

What you get back

Always -1 (with `$!` set to `EINTR`) once a signal handler returns - pause has no “success” exit path.

Differences from upstream

Fully compatible with upstream POSIX.

sleep

Suspend the process for `$secs` seconds, or until a signal arrives.

What you get back

The number of seconds left unslept - 0 for a full sleep, or a positive remainder when a signal cut the sleep short.

Differences from upstream

Fully compatible with upstream POSIX.

File and I/O

isatty

True if `$fd` refers to a terminal.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `ttyname` - the device path for a terminal fd.

ttyname

Device path of the terminal attached to `$fd`, or `undef` if none.

Differences from upstream

Fully compatible with upstream POSIX.

ctermid

Path of the controlling terminal.

What you get back

On most systems, the string `"/dev/tty"`.

Differences from upstream

Fully compatible with upstream POSIX.

getcwd

Current working directory as an absolute path.

What you get back

The path as a byte string, or `undef` if the call fails (for example, if the process has no read permission on some ancestor directory).

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `Cwd::getcwd` - the recommended Perl-level interface.

access

Check the caller's permissions on a path.

lchown

Change the owner/group of a symlink itself (without following it).

Synopsis

```
POSIX::lchown($uid, $gid, $path);
```

What you get back

1 on success, 0 on failure.

Differences from upstream

Fully compatible with upstream POSIX.

mkfifo

Create a FIFO (named pipe) at `$path` with permission bits `$mode`.

What you get back

1 on success, 0 on failure.

Differences from upstream

Fully compatible with upstream POSIX.

remove

Delete a file or empty directory at \$path.

What you get back

1 on success, 0 on failure. Works on both files and empty directories (dispatching to unlink / rmdir internally).

Differences from upstream

Fully compatible with upstream POSIX.

rename

Rename a file from \$old to \$new, atomically when on the same filesystem.

What you get back

1 on success, 0 on failure.

Differences from upstream

Fully compatible with upstream POSIX.

open

Open a file by path, returning a raw file descriptor.

close

Close a raw file descriptor.

What you get back

"0 but true" on success, undef on failure. A negative \$fd is rejected with \$! set to EBADF.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- open, dup, dup2 - where fds come from.

dup

Duplicate a file descriptor, returning the lowest-numbered unused fd.

What you get back

The new fd as an integer on success, undef on failure. A negative \$fd is rejected with \$! set to EBADF.

Differences from upstream

Fully compatible with upstream POSIX.

dup2

Duplicate \$fd1 onto \$fd2, closing \$fd2 first if necessary.

lseek

Reposition the read/write offset of an open fd.

pipe

Create an anonymous pipe and return its two fds.

read

Read up to \$nbytes bytes from \$fd into \$buffer.

write

Write bytes from \$buffer to \$fd.

Time

strftime

Format a broken-down time as a string using a format template.

mktime

Convert a broken-down local time into epoch seconds.

difftime

Difference between two epoch times, \$t1 - \$t2, as a double.

Differences from upstream

Fully compatible with upstream POSIX.

tzset

Refresh the process's timezone information from the TZ environment variable.

Differences from upstream

Fully compatible with upstream POSIX.

tzname

Short names of the current standard and daylight timezones.

clock

Approximate processor time used by the program, in `CLOCKS_PER_SEC` units.

What you get back

An integer count of clock ticks. Divide by `POSIX::CLOCKS_PER_SEC` for seconds. On Linux this counts user+system CPU time of the whole process.

Differences from upstream

Fully compatible with upstream POSIX.

times

Raw wall + CPU times of the process and its children, as a five-element list.

asctime

Canonical English timestamp string from a broken-down time.

ctime

Canonical English timestamp string for an epoch time, in local time.

Locale

setlocale

Query or change the program's locale setting.

localeconv

Numeric and monetary formatting rules for the active locale.

Character classification

toupper

Locale-aware uppercase of one byte value.

What you get back

The uppercased byte as an integer, or the input unchanged if it has no uppercase form in the current `LC_CTYPE` locale.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- Core `uc` - uppercase a whole string, Unicode-aware by default.

tolower

Locale-aware lowercase of one byte value.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- Core `lc` - lowercase a whole string.

Errno and path limits

errno

Current value of the C `errno` variable, as an integer.

What you get back

The integer value. Compare against symbolic constants like `ENOENT`, `EINVAL`, etc., which this module also exports.

Differences from upstream

Fully compatible with upstream POSIX. In ordinary Perl code prefer `$!` (which also stringifies to `strerror(errno)`).

See also

- `strerror` - message string for an `errno` number.

fpathconf

Runtime limit for a configuration variable on an open fd.

pathconf

Runtime limit for a configuration variable on a pathname.

What you get back

The limit value, or `-1` if unsupported/unlimited.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `fpathconf` - same query on an already-open fd.

Terminal control

tcdrain

Block until all queued output on `$fd` has been transmitted.

What you get back

"0 but true" on success, undef on failure. A negative \$fd sets \$! to EBADF.

Differences from upstream

Fully compatible with upstream POSIX.

tcflow

Suspend or resume transmission or reception on terminal \$fd.

tcflush

Discard buffered input, output, or both on terminal \$fd.

tcsendbreak

Send a continuous break condition on terminal \$fd for roughly \$duration tenths of a second (0 means the implementation default).

What you get back

"0 but true" on success, undef on failure. Negative \$duration sets \$! to EINVAL.

Differences from upstream

Fully compatible with upstream POSIX.

tcgetpgrp

Process group ID of the foreground process group for terminal \$fd.

What you get back

The pgid as an integer, or -1 on error (with \$! set).

Differences from upstream

Fully compatible with upstream POSIX.

tcsetpgrp

Make \$pgrp the foreground process group of terminal \$fd.

What you get back

"0 but true" on success, undef on failure.

Differences from upstream

Fully compatible with upstream POSIX.

termios_new

Create a fresh POSIX::Termios object, all fields zeroed.

termios_getattr

Fill the POSIX::Termios object with the current settings of \$fd.

termios_setattr

Apply the POSIX::Termios settings to \$fd.

termios_getispeed

Input baud rate encoded in the termios structure.

What you get back

The encoded speed as an integer (compare against B9600, B115200, etc., which this module also exports).

Differences from upstream

Fully compatible with upstream POSIX.

termios_getospeed

Output baud rate encoded in the termios structure.

Differences from upstream

Fully compatible with upstream POSIX.

termios_setispeed

Encode an input baud rate into the termios structure.

What you get back

"0 but true" on success, undef on failure. The change is not applied to the terminal until you call setattr.

Differences from upstream

Fully compatible with upstream POSIX.

termios_setospeed

Encode an output baud rate into the termios structure.

Differences from upstream

Fully compatible with upstream POSIX.

termios_getcc

Value of a special control character in the termios structure.

`termios_setcc`

Set a special control character in the termios structure.

Edge cases

- Indices beyond NCCS croak with `Bad setcc subscript`.
- The change is not applied until you call `setattr`.

Differences from upstream

Fully compatible with upstream POSIX.

Signals

`sigset_new`

Create a `POSIX::SigSet` object, optionally pre-populated with the given signal numbers.

`sigset_addset`

Add signal `$sig` to the set.

What you get back

`"0 but true"` on success, `undef` on failure.

Differences from upstream

Fully compatible with upstream POSIX.

`sigset_delset`

Remove signal `$sig` from the set.

What you get back

`"0 but true"` on success, `undef` on failure.

Differences from upstream

Fully compatible with upstream POSIX.

`sigset_emptyset`

Remove every signal from the set.

Differences from upstream

Fully compatible with upstream POSIX.

`sigset_fillset`

Add every signal to the set.

Differences from upstream

Fully compatible with upstream POSIX.

sigset_ismember

Test whether signal `$sig` is in the set.

What you get back

1 if the signal is a member, 0 if it is not, -1 on error.

Differences from upstream

Fully compatible with upstream POSIX.

sigprocmask

Change, query, or both the set of signals blocked for this process.

sigpending

Fill `$sigset` with the signals currently pending for delivery.

What you get back

"0 but true" on success, undef on failure.

Differences from upstream

Fully compatible with upstream POSIX.

sigsuspend

Temporarily replace the signal mask with `$sigset` and suspend the process until any non-blocked signal arrives.

What you get back

Always undef (with `$_` set to `EINTR`) once a signal handler has run - there is no success exit path.

Differences from upstream

Fully compatible with upstream POSIX.

sigaction

Install or inspect a signal handler for signal `$sig`.

Constants

constant

Look up a POSIX constant by name.

abs

Integer absolute value of \$n.

Synopsis

```
my $n = POSIX::abs(-7);    # 7
```

What you get back

An integer scalar. Unlike the core `abs` built-in, the argument is coerced to IV before taking the absolute value, so floating-point inputs are truncated.

Differences from upstream

Fully compatible with upstream POSIX.

frexp

Split a floating-point number into mantissa and exponent.

Synopsis

```
my ($frac, $exp) = POSIX::frexp($x);  
## $x == $frac * 2**$exp, with 0.5 <= |$frac| < 1 (or $frac == 0).
```

What you get back

A two-element list: the fractional part as a double and the exponent as an integer.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `ldexp` - the inverse operation: reassemble \$x from \$frac and \$exp.
- `modf` - split into integer and fractional parts instead.

modf

Split a number into integer and fractional parts, both as doubles.

Synopsis

```
my ($frac, $int) = POSIX::modf(3.75); # (0.75, 3.0)
```

What you get back

A two-element list (\$frac, \$int). Both parts carry the sign of the input.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- frexp - split into mantissa and exponent instead.

fma

Fused multiply-add: $x * y + z$ with a single rounding.

Synopsis

```
my $r = POSIX::fma($x, $y, $z);
```

What you get back

A double. Because there is only one rounding step, the result can be more accurate than the naive two-step expression, especially for catastrophic-cancellation patterns like $a * b - c * d$ rewritten as `fma($a, $b, -($c * $d))`.

Differences from upstream

Fully compatible with upstream POSIX.

nan

Build a quiet NaN, optionally with a payload string.

Synopsis

```
my $q = POSIX::nan(); # quiet NaN
my $p = POSIX::nan("0x42"); # NaN with payload where supported
```

Edge cases

- An empty or missing argument yields a plain quiet NaN.
- Payload strings the C library cannot parse fall back to a plain quiet NaN silently.

Differences from upstream

Fully compatible with upstream POSIX.

remquo

Floating-point remainder with partial quotient.

Synopsis

```
my ($rem, $quo) = POSIX::remquo($x, $y);
```

What you get back

A two-element list: the IEEE remainder and the low bits of the quotient as an integer (typically at least three bits, enough for argument reduction of trigonometric functions).

Differences from upstream

Fully compatible with upstream POSIX.

jn

Bessel function of the first kind, integer order \$n.

Synopsis

```
my $y = POSIX::jn($n, $x);
```

Differences from upstream

Fully compatible with upstream POSIX.

See also

- j0, j1 - fixed-order variants for \$n == 0 and \$n == 1.
- yn - Bessel function of the second kind.

yn

Bessel function of the second kind, integer order \$n.

Synopsis

```
my $y = POSIX::yn($n, $x);
```

Edge cases

- yn(\$n, 0) returns negative infinity (the function has a singularity at zero).

Differences from upstream

Fully compatible with upstream POSIX.

See also

- y0, y1 - fixed-order variants.
- jn - Bessel function of the first kind.

fpclassify

Classify a floating-point number.

Synopsis

```
use POSIX qw(fpclassify FP_NAN FP_INFINITE FP_ZERO FP_SUBNORMAL FP_
↳NORMAL);
my $k = fpclassify($x);
```

What you get back

An integer matching one of the FP_* constants. Compare against the symbolic names, not a hardcoded number - the values are libc implementation details.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- isnan, isinf, isfinite, isnormal - narrower single-class tests.

fesetround

Change the floating-point rounding direction.

Synopsis

```
use POSIX qw(fesetround FE_DOWNWARD);
fesetround(FE_DOWNWARD);
```

What you get back

0 on success, non-zero if the requested mode is not supported.

Differences from upstream

Fully compatible with upstream POSIX.

wexitstatus

Exit code of a child that terminated normally.

Synopsis

```
system($cmd);
my $code = POSIX::WEXITSTATUS($?);
```

What you get back

The low 8 bits of the child's `exit()` argument. Only meaningful when `WIFEXITED($status)` is true.

Differences from upstream

Fully compatible with upstream POSIX.

strtod

Parse a double from the start of a string, locale-aware.

Synopsis

```
use POSIX qw(strtod);
my ($num, $unparsed) = strtod("3.14abc"); # (3.14, 3)
```


What you get back

A two-element list: the parsed value and the number of bytes that did not form part of the number. When `$unparsed == 0`, the whole string was consumed.

Edge cases

- Input that doesn't start with a number: returns `(0, length)`.
- Overflow: returns `HUGE_VAL` and sets `$!` to `ERANGE`.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `strtol`, `strtoul` - integer variants.

strtol

Parse a signed integer from the start of a string, in a given base.

Synopsis

```
use POSIX qw(strtol);
my ($n, $unparsed) = strtol("0x2a rest", 0); # (42, 5)
```

What you get back

A two-element list: the parsed integer and the number of bytes that were not consumed. Base `0` auto-detects from prefix (`0x`, `0b`, leading `0`); otherwise `$base` must be 2-36.

Edge cases

- Invalid base: returns `(undef, undef)` and sets `$!` to `EINVAL`.
- Overflow: returns `LONG_MIN / LONG_MAX` and sets `$!` to `ERANGE`.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `strtoul` - unsigned variant.
- `strtod` - floating-point variant.

strtoul

Parse an unsigned integer from the start of a string, in a given base.

Synopsis

```
use POSIX qw(strtoul);  
my ($n, $unparsed) = strtoul("255 end", 10);
```

Edge cases

- Invalid base: returns (undef, undef) and sets \$! to EINVAL.
- Negative input: parsed and two's-complement-cast to unsigned, matching C strtoul.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- strtol - signed variant.

strerror

Message string for an errno value.

Synopsis

```
use POSIX qw(strerror);  
print strerror($!);
```

What you get back

A byte string in the current locale, like "No such file or directory". For unknown error numbers, libc returns an implementation-defined placeholder.

Differences from upstream

Fully compatible with upstream POSIX.

sprintf

Format a string POSIX-style, identical to built-in sprintf.

Synopsis

```
use POSIX qw(sprintf);
my $line = sprintf("%-10s %6.2f", $name, $price);
```

What you get back

A single scalar holding the formatted string. Conversion specifiers, flags, width, precision, and length modifiers are whatever your built-in `sprintf` supports.

Edge cases

- Called with no arguments: croaks with Usage: `POSIX::sprintf(pattern, args...)`.
- Too few arguments for the format: unfilled specifiers expand to the empty string, matching the core `sprintf`.

Differences from upstream

Fully compatible with upstream POSIX.

setpgid

Put process `$pid` into process group `$pgid`.

Synopsis

```
POSIX::setpgid($pid, $pgid);
```

What you get back

1 on success, 0 on failure. A `$pid` of 0 means the calling process; a `$pgid` of 0 means use `$pid` as the new group ID.

Differences from upstream

Fully compatible with upstream POSIX.

uname

Kernel and machine identification as a five-element list.

Synopsis

```
use POSIX qw(uname);
my ($sysname, $nodename, $release, $version, $machine) = uname();
```

What you get back

Five strings in order: sysname (e.g. "Linux"), nodename (hostname), release (kernel release), version (kernel version string), and machine (architecture, e.g. "x86_64").

Edge cases

- On uname failure, returns an empty list.

Differences from upstream

Fully compatible with upstream POSIX.

sysconf

Runtime system configuration value for the given `_SC_*` name.

Synopsis

```
use POSIX qw(sysconf _SC_OPEN_MAX _SC_PAGESIZE);
my $max_fd = sysconf(_SC_OPEN_MAX);
```

What you get back

The configured value, or -1 if the option is not supported (in which case `$!` may or may not be set - distinguish by checking `$!` before and after).

Differences from upstream

Fully compatible with upstream POSIX.

access

Check the caller's permissions on a path.

Synopsis

```
use POSIX qw(access R_OK W_OK);
print "readable\n" if access($path, R_OK);
```

What you get back

The string `"0 but true"` when the access check succeeds, or `undef` when it fails (with `$!` set). The `"0 but true"` return matches core Perl's convention for system calls whose success value is 0.

Edge cases

- Check uses real (not effective) UID/GID, so is vulnerable to TOCTOU races; prefer trying the operation and handling failure.

Differences from upstream

Fully compatible with upstream POSIX.

open

Open a file by path, returning a raw file descriptor.

Synopsis

```
use POSIX qw(open O_RDONLY O_CREAT O_WRONLY);
my $fd = open($path, O_RDONLY)
    or die "open: $!";
```

What you get back

A non-negative integer fd on success, undef on failure (with \$! set). This is the raw system fd, not a Perl filehandle - pair it with POSIX::close or convert via IO::Handle::fdopen.

Differences from upstream

Fully compatible with upstream POSIX. Third argument \$mode defaults to 0666 when opening with O_CREAT and no mode given.

See also

- close, dup, lseek, read, write - the rest of the raw fd API.
- Core open - the Perl-native filehandle interface.

dup2

Duplicate \$fd1 onto \$fd2, closing \$fd2 first if necessary.

Synopsis

```
POSIX::dup2($fd, 1); # redirect stdout to $fd
```

What you get back

The target fd on success, undef on failure. Either argument being negative is rejected with \$! set to EBADF.

Differences from upstream

Fully compatible with upstream POSIX.

lseek

Reposition the read/write offset of an open fd.

Synopsis

```
use POSIX qw(lseek SEEK_SET SEEK_CUR SEEK_END);
my $pos = lseek($fd, 0, SEEK_CUR);    # current offset
```

What you get back

The new absolute offset on success, -1 on failure (with \$! set).

Differences from upstream

Fully compatible with upstream POSIX.

pipe

Create an anonymous pipe and return its two fds.

Synopsis

```
use POSIX qw(pipe);
my ($rd, $wr) = pipe();
```

What you get back

A two-element list (\$read_fd, \$write_fd) on success. On failure, a single undef is returned and \$! is set.

Differences from upstream

Fully compatible with upstream POSIX.

read

Read up to \$nbytes bytes from \$fd into \$buffer.

Synopsis

```
use POSIX qw(read);
my $buf = "";
my $n = read($fd, $buf, 4096);
```

What you get back

On success, the number of bytes read (or "0 but true" for EOF); on failure, undef (with \$! set). The data is written into \$buffer in place - truncated or grown to match the actual read length.

Edge cases

- Partial reads are normal on pipes, sockets, and terminals; always check the return length.

Differences from upstream

Fully compatible with upstream POSIX.

write

Write bytes from \$buffer to \$fd.

Synopsis

```
POSIX::write($fd, $data, length($data));
```

What you get back

The number of bytes actually written. A value less than \$nbytes is not an error - common for pipes and sockets - and the caller must loop to write the rest.

Differences from upstream

Fully compatible with upstream POSIX.

strftime

Format a broken-down time as a string using a format template.

Synopsis

```
use POSIX qw(strftime);  
my @tm = localtime;  
my $str = strftime("%Y-%m-%d %H:%M:%S", @tm);
```

What you get back

A byte string formatted according to the template. Argument order matches localtime / gmtime: second, minute, hour, mday, mon (0-11), year (years since 1900), wday, yday, isdst - the last three optional and defaulting to -1.

Edge cases

- Fewer than seven arguments (format plus the first six tm fields): returns the empty string.
- Output truncated at 1024 bytes.
- Month is 0-11 and year is years-since-1900, the struct tm convention, *not* the 1-12 / 4-digit-year human convention.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- mktime - go the other way: broken-down time → epoch seconds.
- asctime, ctime - simpler canned-format alternatives.

mktime

Convert a broken-down local time into epoch seconds.

Synopsis

```
use POSIX qw(mktime);  
my $t = mktime($sec, $min, $hour, $mday, $mon, $year - 1900);
```

What you get back

The epoch seconds as an integer on success. The sentinel value "0 but true" is returned when the epoch genuinely is 0 (so you can distinguish it from failure). undef is returned on failure (when the input cannot be represented).

Edge cases

- The tm fields are normalised in place by libc, so values outside the usual ranges (e.g. mon = 13) are accepted and carry into adjacent fields.
- wday and yday are ignored on input; isdst defaults to -1 (let libc figure it out) when omitted.

Differences from upstream

Fully compatible with upstream POSIX.

tzname

Short names of the current standard and daylight timezones.

Synopsis

```
use POSIX qw(tzname);  
my ($std, $dst) = tzname();
```

What you get back

A two-element list. Both strings are derived from TZ - the call also implicitly refreshes the timezone cache.

Differences from upstream

Fully compatible with upstream POSIX.

times

Raw wall + CPU times of the process and its children, as a five-element list.

Synopsis

```
my ($real, $user, $sys, $cuser, $csys) = POSIX::times();
```

What you get back

Five integer tick counts, in `sysconf(_SC_CLK_TCK)` units - raw, not divided by the ticks-per-second constant. Divide if you want seconds.

Differences from upstream

Fully compatible with upstream POSIX.

asctime

Canonical English timestamp string from a broken-down time.

Synopsis

```
use POSIX qw(asctime);  
my @tm = localtime;  
print asctime(@tm);    # "Wed Apr 22 14:30:00 2026\n"
```

What you get back

A 25-byte string **including the trailing newline**, or `undef` if nine arguments are not supplied.

Differences from upstream

Fully compatible with upstream POSIX. The trailing newline is preserved as-is; do not assume this function trims it.

ctime

Canonical English timestamp string for an epoch time, in local time.

Synopsis

```
use POSIX qw(ctime);
print ctime(time);   # "Wed Apr 22 14:30:00 2026\n"
```

What you get back

A 25-byte string including the trailing newline, or `undef` on conversion failure.

Differences from upstream

Fully compatible with upstream POSIX. The trailing newline is preserved.

See also

- `asctime` - same formatting, but from a broken-down time list.

nice

Adjust the current process's scheduling niceness by `$incr`.

Synopsis

```
POSIX::nice(5);   # lower priority
```

What you get back

The new niceness value. When the new value genuinely is `0`, returns the string `"0 but true"` to distinguish it from error. On error, returns `undef` and sets `$!`.

Edge cases

- Positive increments lower priority; only root can pass negative values to raise priority.

Differences from upstream

Fully compatible with upstream POSIX.

setlocale

Query or change the program's locale setting.

Synopsis

```
use POSIX qw(setlocale LC_ALL LC_NUMERIC);
setlocale(LC_ALL, "");           # honour environment
my $cur = setlocale(LC_NUMERIC); # query current
```

What you get back

The resulting locale name as a string, or undef if the category/locale combination is invalid. Calling with one argument queries; calling with two sets. Passing "" as the new locale asks libc to pick from LC_* / LANG environment variables.

Edge cases

- Changing a category can affect formatting of numbers, collation, date strings, and error messages throughout the process.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- localeconv - inspect the active locale's numeric and monetary conventions.

localeconv

Numeric and monetary formatting rules for the active locale.

Synopsis

```
use POSIX qw(localeconv);
my $lc = localeconv();
print "Decimal: $lc->{decimal_point}\n";
```

What you get back

A hashref whose keys are the struct lconv field names - decimal_point, thousands_sep, grouping, int_curr_symbol, currency_symbol, mon_decimal_point, mon_thousands_sep, mon_grouping, positive_sign, negative_sign, and the monetary char fields (int_frac_digits, frac_digits, p_cs_precedes, p_sep_by_space, n_cs_precedes, n_sep_by_space, p_sign_posn, n_sign_posn).

Edge cases

- Char fields equal to CHAR_MAX mean “not available in this locale”; all numeric char fields are returned unconditionally.
- Returns undef if libc localeconv () fails (rare).

Differences from upstream

Fully compatible with upstream POSIX.

See also

- setlocale - pick which locale those rules come from.

fpathconf

Runtime limit for a configuration variable on an open fd.

Synopsis

```
use POSIX qw(fpathconf _PC_NAME_MAX);  
my $max = fpathconf($fd, _PC_NAME_MAX);
```

What you get back

The limit value, or -1 if the variable is unsupported or unlimited (check \$! to distinguish).

Differences from upstream

Fully compatible with upstream POSIX.

constant

Look up a POSIX constant by name.

Synopsis

```
my $enoent = POSIX::constant("ENOENT");
```

What you get back

The integer or floating-point value of the named constant, or undef if the name is not known.

Differences from upstream

Fully compatible with upstream POSIX. Ordinary code should import the constant by name rather than going through this accessor - it exists mainly for tooling and introspection.

tcflow

Suspend or resume transmission or reception on terminal \$fd.

Synopsis

```
use POSIX qw(tcflow TC00FF TCOON);  
tcflow($fd, TC00FF); # pause output
```

What you get back

"0 but true" on success, undef on failure. Negative \$action sets \$! to EINVAL.

Differences from upstream

Fully compatible with upstream POSIX.

tcflush

Discard buffered input, output, or both on terminal \$fd.

Synopsis

```
use POSIX qw(tcflush TCIFLUSH TCOFLUSH TCIOFLUSH);  
tcflush($fd, TCIOFLUSH); # drop pending input and output
```

What you get back

"0 but true" on success, undef on failure. Negative \$queue_selector sets \$! to EINVAL.

Differences from upstream

Fully compatible with upstream POSIX.

termios_new

Create a fresh `POSIX::Termios` object, all fields zeroed.

Synopsis

```
use POSIX;  
my $t = POSIX::Termios->new;  
$t->getattr(fileno(STDIN));
```

What you get back

A blessed `POSIX::Termios` reference. To fill it with the current terminal settings, follow up with `$t->getattr($fd)`.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `getattr`, `setattr` - populate and apply terminal settings.

termios_getattr

Fill the `POSIX::Termios` object with the current settings of `$fd`.

Synopsis

```
$t->getattr(fileno(STDIN));
```

What you get back

"0 but true" on success, undef on failure. `$fd` defaults to 0 (stdin).

Differences from upstream

Fully compatible with upstream POSIX.

termios_setattr

Apply the `POSIX::Termios` settings to `$fd`.

Synopsis

```
use POSIX qw(TCSANOW TCSADRAIN);
$t->setattr(fileneno(STDIN), TCSANOW);
```

What you get back

"0 but true" on success, undef on failure. \$fd defaults to 0, \$optional_actions defaults to TCSANOW (apply immediately).

Edge cases

- Negative \$optional_actions sets \$! to EINVAL.

Differences from upstream

Fully compatible with upstream POSIX.

termios_getcc

Value of a special control character in the termios structure.

Synopsis

```
use POSIX qw(VINTR VEOF);
my $intr = $t->getcc(VINTR);    # Ctrl-C byte
```

Edge cases

- Indices beyond the NCCS bound croak with Bad getcc subscript.

Differences from upstream

Fully compatible with upstream POSIX.

sigset_new

Create a POSIX::SigSet object, optionally pre-populated with the given signal numbers.

Synopsis

```
use POSIX qw(SIGINT SIGTERM);
my $set = POSIX::SigSet->new(SIGINT, SIGTERM);
```

What you get back

A blessed `POSIX::SigSet` reference, with `sigemptyset` applied first and then each argument added via `sigaddset`.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `addset`, `delset`, `emptyset`, `fillset`, `ismember` - mutate or query the set.
- `sigprocmask` - install a `SigSet` as the process mask.

sigprocmask

Change, query, or both the set of signals blocked for this process.

Synopsis

```
use POSIX qw(sigprocmask SIG_BLOCK SIG_UNBLOCK SIG_SETMASK);
my $mask = POSIX::SigSet->new(SIGINT);
sigprocmask(SIG_BLOCK, $mask);
```

What you get back

"0 but true" on success, undef on failure. When `$oldsigset` is supplied, it is filled with the mask in effect before the call.

Edge cases

- Passing a non-`POSIX::SigSet` reference for either set croaks with `sigset is not of type POSIX::SigSet/oldsigset is not of type POSIX::SigSet`.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `sigpending`, `sigsuspend`, `sigaction` - the rest of the POSIX signal API.

sigaction

Install or inspect a signal handler for signal `$sig`.

Synopsis

```
use POSIX qw(sigaction SIGINT);
my $action = POSIX::SigAction->new("IGNORE");
my $old = POSIX::SigAction->new;
sigaction(SIGINT, $action, $old);
```

What you get back

"0 but true" on success, undef on failure. When \$oldaction is supplied, it is populated with the previous HANDLER, MASK, FLAGS, and SAFE fields.

Edge cases

- Negative \$sig croaks with Negative signals are not allowed.
- An action hash without a HANDLER key croaks with Can't supply an action without a HANDLER.

Differences from upstream

- Coderef handlers are not yet wired to the perl5 callback bridge, so passing one silently falls through and leaves the current disposition in place. Upstream would invoke the Perl code on signal delivery.

See also

- sigprocmask - change the blocked-signal mask.
- POSIX::SigSet - build the MASK field of an action.

5.2.21 PadWalker

Reach into another subroutine's my and our variables - read them, change them, find out their names - from anywhere on the call stack.

PadWalker is the tool of choice for debuggers, introspection utilities, and any code that needs to look at variables it did not declare. It is what Perl's own built-in debugger uses, and it is the modern replacement for Devel::LexAlias when you want to alias or substitute a lexical in a caller's pad.

Every function that walks the stack takes a \$level argument interpreted just like the argument to caller:

- 0 - the current scope.
- 1 - the scope of whoever called the current sub.
- 2 - two frames up, and so on.

File scope is a valid destination: asking for a \$level deeper than the call stack supports raises Not nested deeply enough.

Returned hashes key on the variable name *including* its sigil - so `$x` appears under the key `'$x'`, `@items` under `'@items'`. Values are always references to the live variable, so assigning through the reference updates the original.

Functions

Lexical inspection

`peek_my`

Return every my variable in scope at a given call-stack depth as a hash of name-to-reference pairs.

`peek_our`

Return every our variable in scope at a given call-stack depth as a hash of name-to-reference pairs.

`peek_sub`

Return the my variables declared inside a given subroutine, whether or not the sub is currently on the call stack.

`closed_over`

Return the lexicals that a subroutine closes over - the variables it references but did not itself declare.

`var_name`

Given a reference to a variable, return the name that variable has in a specified subroutine's pad - or `undef` if it is not there.

Lexical modification

`set_closed_over`

Replace the lexicals that a subroutine closes over with a new set of references.

`peek_my`

Return every my variable in scope at a given call-stack depth as a hash of name-to-reference pairs.

Synopsis

```
use PadWalker qw(peek_my);
my $h = peek_my($level);
```

What you get back

A hash reference. Each key is a variable name *with* its sigil - '\$x', '@list', '%env'. Each value is a reference to the live variable, so dereferencing and assigning through it changes the original. our variables are not included; use peek_our for those.

Examples

Read a my variable in the current scope:

```
my $x = 12;
my $h = peek_my(0);
print ${ $h->{'$x'} };      # 12
```

Modify the caller's my variable:

```
sub bump_x {
    my $h = peek_my(1);
    ${ $h->{'$x'} }++;
}
my $x = 5;
bump_x();
print $x;                  # 6
```

A my declared deeper than the target frame is not in the returned hash - only variables visible at the call site come back.

Edge cases

- \$level defaults to 0 if omitted.
- \$level deeper than the call stack croaks with Not nested deeply enough.
- Negative \$level is not supported and croaks.
- Variables declared inside an eval BLOCK between the target frame and the call site are folded in at the sequence number of the call - they become visible alongside the target's own lexicals.

Differences from upstream

Fully compatible with upstream PadWalker 2.5.

See also

- peek_our - same idea for our variables.
- peek_sub - inspect a CV's pad directly, without walking frames.
- var_name - given a reference, find the name it had in a pad.

peek_our

Return every our variable in scope at a given call-stack depth as a hash of name-to-reference pairs.

Synopsis

```
use PadWalker qw(peek_our);
my $h = peek_our($level);
```

What you get back

A hash reference keyed on sigiled variable names, with values that are references to the package variables those our declarations alias. Dereferencing reads from the package slot; assigning through the reference writes back to it.

Examples

```
package Main;
our $config = 'ok';
my $h = peek_our(0);
print ${ $h->{'$config'} }; # ok
```

Inspect a caller's our variables:

```
sub show_our {
    my $h = peek_our(1);
    print join ", ", sort keys %$h;
}
```

Edge cases

- \$level works exactly as in peek_my - 0 for the current scope, 1 for the caller, and so on.
- our declarations are resolved through the package stash in effect at the declaration site, not the calling site.
- A package with an our name but no value yet appears in the hash with a reference to undef.

Differences from upstream

Fully compatible with upstream PadWalker 2.5.

See also

- peek_my - the sibling for my variables.
- var_name - resolve a reference back to its declared name.

peek_sub

Return the my variables declared inside a given subroutine, whether or not the sub is currently on the call stack.

Synopsis

```
use PadWalker qw(peek_sub);
my $h = peek_sub(\&some_sub);
```

What you get back

A hash reference keyed on sigiled variable names. The references point at whatever values the pad currently holds. If the sub is not active, those values will typically be undef for scalars or empty for aggregates - the sub has not yet run to populate them. If the sub is active, the references point at the live values of the most recent invocation.

Examples

```
my $x = "Hello!";
my $r = peek_sub(sub { $x }->{'$x'});
print $$r;           # Hello!
```

Inspect the pad of a named sub:

```
sub counter { my $n = 0; sub { $n++ } }
my $h = peek_sub(\&counter);
print join ", ", sort keys %$h;    # $n
```

Edge cases

- If the sub declares several my variables with the same name, only the last one is returned. There is no way to distinguish shadowed entries; this is a known upstream limitation.
- XS subs and subs without a padlist croak with PadWalker: cv has no padlist.
- Passing something that is not a code reference croaks with PadWalker: cv is not a code reference.

Differences from upstream

Fully compatible with upstream PadWalker 2.5.

See also

- `closed_over` - the strict subset of `peek_sub` covering only variables the sub closes over.
- `peek_my` - walk the live call stack instead of a named sub.

- `var_name` - name-from-reference lookup against the same pad.

closed_over

Return the lexicals that a subroutine closes over - the variables it references but did not itself declare.

Synopsis

```
use PadWalker qw(closed_over);
my $h = closed_over(\&some_sub);
my ($h, $targs) = closed_over(\&some_sub);
```

What you get back

In scalar context, a hash reference keyed on sigiled variable names with references to the enclosing variables as values. In list context, that hash reference plus a second hash reference keyed on the sub's pad slot indices (as numbers) - useful for tools that need to match up pad positions with values.

Only captured lexicals appear; my variables declared inside the sub itself and our variables are both excluded.

Examples

Classic closure:

```
my $n = 41;
my $adder = sub { $n + $_[0] };
my $h = closed_over($adder);
${ $h->{'$n'} } = 100;
print $adder->(1);           # 101
```

List form for serialisers that need pad indices:

```
my ($names, $slots) = closed_over($sub);
for my $idx (sort { $a <=> $b } keys %$slots) {
    print "slot $idx -> ", ${ $slots->{$idx} }, "\n";
}
```

Edge cases

- A sub that captures nothing returns an empty hash reference.
- XS subs and subs with no padlist return empty hashes rather than croaking.
- Passing a bare CV name (without `\&`) works the same as passing a code reference - the XS layer dereferences either form.

Differences from upstream

Fully compatible with upstream PadWalker 2.5.

See also

- `set_closed_over` - substitute the captured variables wholesale.
- `peek_sub` - full pad, not just the captures.
- `var_name` - given one of the returned references, find its name.

set_closed_over

Replace the lexicals that a subroutine closes over with a new set of references.

Synopsis

```
use PadWalker qw(set_closed_over);
set_closed_over(\&some_sub, \%replacements);
```

What you get back

Nothing useful - the function is called for its side effect. For every name in `%replacements` that matches one of the sub's captured pad slots, the corresponding slot is replaced by the referent of `$replacements{$name}`.

Examples

Swap a closure's captured variable for a different scalar:

```
my $n = 1;
my $say = sub { print "$n\n" };
my $other = 42;
set_closed_over($say, { '$n' => \ $other });
$say->();           # 42
```

Round-trip through `closed_over` is the idiomatic pattern:

```
my $refs = closed_over($sub);
$refs->{'$x'} = \my $fresh;
set_closed_over($sub, $refs);
```

Edge cases

- Names in `%replacements` that the sub does not close over are silently ignored.
- The reftype of each replacement must match the original - scalar for scalar, array for array, and so on. A mismatch croaks with `Incorrect reftype for variable NAME (got X expected Y)`.

- A non-reference value in %replacements croaks with `The variable for NAME is not a reference.`
- XS subs and subs with no padlist are no-ops.

Differences from upstream

Fully compatible with upstream PadWalker 2.5.

See also

- `closed_over` - read the captures this function writes.
- `peek_sub` - broader inspection, read-only.

var_name

Given a reference to a variable, return the name that variable has in a specified subroutine's pad - or undef if it is not there.

Synopsis

```
use PadWalker qw(var_name);
my $name = var_name($level, \ $var);
my $name = var_name(\&sub, \ $var);
```

What you get back

A string - the sigiled pad name, e.g. `'$foo'` or `'@items'`. If the reference does not correspond to any pad slot in the target sub, returns undef.

The first argument selects the sub to search: a non-negative integer is interpreted the same as the argument to `peek_my` (walk the call stack), a code reference is searched directly.

Examples

Look up the name of a local variable by reference:

```
my $foo;
print var_name(0, \ $foo);      # $foo
```

Wrap it for use from helper subs:

```
sub my_name {
    return var_name(1, shift);
}
my $bar;
print my_name(\ $bar);          # $bar
```


Edge cases

- References to globals, package variables, or anonymous SVs return undef - only pad entries are searched.
- If several pad slots point at the same SV, the last one wins (same caveat as peek_sub).
- The second argument must be a reference; passing a plain value croaks with Usage: PadWalker::var_name(sub, var_ref).

Differences from upstream

Fully compatible with upstream PadWalker 2.5.

See also

- peek_my - the inverse direction, names to references.
- peek_sub - all names in a sub's pad regardless of value.
- closed_over - captured lexicals only.

5.2.22 PerlIO

Namespace containing the following modules:

- `PerlIO::encoding` — Open a filehandle with a transparent character-set encoding filter.
- `PerlIO::mmap` — Memory-mapped IO layer opened as `<:mmap`, with a `read()/write()` fallback when `mmap` fails.
- `PerlIO::via` — Write a PerlIO layer in Perl: open with `:via(MyClass)` and I/O dispatches to your class.

PerlIO::encoding

Open a filehandle with a transparent character-set encoding filter.

This is a PerlIO layer implementation. It registers a PerlIO layer vtable (`PerlIO_encode`) that perl5's PerlIO system calls for I/O operations on handles opened with `:encoding( . . )`.

The layer works by overloading “fill” and “flush” methods of `PerlIOBuf`:

- “fill” reads bytes from the layer below, calls `Encode->decode()` to convert to UTF-8, and presents the result to layers above.
- “flush” takes UTF-8 data from the write buffer, calls `Encode->encode()` to convert to the target encoding, and writes to the layer below.

`encoding.xs` struct `PerlIOEncode` extends `PerlIOBuf` with: `bufsv` - SV buffer seen by layers above `dataSV` - data read from layer below (pending bytes) `enc` - the `Encode` encoding object `chk` - `CHECK` flag for encode/decode calls `flags` - `NEEDS_LINES` flag in `EncodeCall` - re-entrancy guard

The BOOT section:

1. Loads Encode module if not already loaded (via `load_module`)
2. Fetches `Encode:::STOP_AT_PARTIAL` and `Encode:::LEAVE_SRC` constants
3. Registers the “encoding” PerlIO layer via `PerlIO_define_layer`

Functions

Other Functions

pop_result

Pop a single SV result off the Perl stack (equivalent of SPAGAIN; POPs). Returns a p5api SV*.

PerlIO::mmap

Memory-mapped IO layer opened as `<:mmap`, with a `read()/write()` fallback when `mmap` fails.

This is a PerlIO layer that memory-maps files for reading instead of using `read()` syscalls. It falls back to `PerlIOBuf` if `mmap` fails.

The layer extends `PerlIOBuf` with: `mptr` - mapped address (from `mmap()`) `len` - mapped length `bbuf` - saved malloced buffer if map fails

The BOOT section registers the “mmap” PerlIO layer via `PerlIO_define_layer`.

`mmap.xs` vtable (`PerlIO_mmap`, lines 258-287): `Pushed` = `PerlIOBuf_pushed` - line 263 `Popped` = `PerlIOBuf_popped` - line 264 `Open` = `PerlIOBuf_open` - line 265 `Binmode` = `PerlIOBase_binmode` - line 266 `Getarg` = `NULL` - line 267 `Filen` = `PerlIOBase_fileno` - line 268 `Dup` = `PerlIOMmap_dup` - line 269 `Read` = `PerlIOBuf_read` - line 270 `Unread` = `PerlIOMmap_unread` - line 271 `Write` = `PerlIOMmap_write` - line 272 `Seek` = `PerlIOBuf_seek` - line 273 `Tell` = `PerlIOBuf_tell` - line 274 `Close` = `PerlIOBuf_close` - line 275 `Flush` = `PerlIOMmap_flush` - line 276 `Fill` = `PerlIOMmap_fill` - line 277 `Eof` = `PerlIOBase_eof` - line 278 `Error` = `PerlIOBase_error` - line 279 `Clearerr` = `PerlIOBase_clearerr` - line 280 `Setlinebuf` = `PerlIOBase_setlinebuf` - line 281 `Get_base` = `PerlIOMmap_get_base` - line 282 `Get_bufsiz` = `PerlIOBuf_bufsiz` - line 283 `Get_ptr` = `PerlIOBuf_get_ptr` - line 284 `Get_cnt` = `PerlIOBuf_get_cnt` - line 285 `Set_ptrcnt` = `PerlIOBuf_set_ptrcnt` - line 286

PerlIO::via

Write a PerlIO layer in Perl: open with `:via(MyClass)` and I/O dispatches to your class.

This is a PerlIO layer implementation that delegates I/O operations to a Perl class. When a handle is opened with `:via(MyClass)`, each PerlIO vtable method (`read`, `write`, `fill`, `flush`, etc.) calls the corresponding Perl method on the class object (`MyClass->WRITE`, `MyClass->READ`, etc.).

`via.xs` struct `PerlIOVia` extends `_PerlIO` (base) with: `stash` - HV* for the class package obj - SV* the blessed object (or class name) `var` - SV* buffer for FILL results `cnt` - SSize_t byte count remaining in `var` `io` - IO* for the filehandle to layer below `fh` - SV* reference to the GV wrapping `io` `PUSHED.UTF8` - CV* cached method lookups (one per `via` method)

The `MYMethod(x)` macro expands to `"METHODNAME"`, `&s->x` - passing both the string name and a pointer to the cached CV* slot.

The BOOT section registers the “via” PerlIO layer via `PerlIO_define_layer`.

Functions

Other Functions

`pop_result`

Pop a single SV result off the Perl stack (equivalent of SPAGAIN; POPs). Returns a `p5api SV*`.

5.2.23 Peta

Namespace containing the following modules:

- `Peta::FFI` — Call C library functions from Perl without writing XS or a binding crate.

`Peta::FFI`

Call C library functions from Perl without writing XS or a binding crate.

`Peta::FFI` is the dynamic FFI layer `ppperl` ships for reaching into shared libraries at runtime. You open a library with `dlopen`, call any of its exported functions through `call` by giving a type signature string, and release the handle with `dlclose`. A fourth helper, `scan`, enumerates the shared libraries the dynamic linker already knows about, which is useful for finding the right `soname` before you open it.

Reach for this module when:

- The function you need lives in a system library (`libm`, `libc`, `libuuid`, `libcrypto`) and writing a full XS binding is more ceremony than the problem deserves.
- You already know the C prototype and want a one-liner, not a build step.
- You are prototyping against a third-party `.so` and will decide later whether to graduate to a proper binding.

It is **not** a replacement for XS when you need callbacks from C back into Perl, struct layouts beyond scalars, or automatic header parsing. For those, write an XS module.

The two sibling modules `Peta::FFI::Libc` and `Peta::FFI::UUID` are pre-baked convenience wrappers built on top of this layer - they handle the `dlopen` / `call` / `dlclose` dance for you for a fixed set of functions. If your need is covered by one of them, prefer it over hand-rolled FFI.

Signature strings

Every `call` takes a signature string of the form `(args)ret`, where each position is one single-character type code:

Code	C type	Perl side
v	void	return only; 1 is returned
i	int	integer
l	long	integer
L	unsigned long / size_t	integer
d	double	float
f	float	float
p	const char *	input string
P	mutable buffer	output buffer, auto-allocated

A signature with no arguments is written `()ret`, e.g. `()i` for an int-returning no-arg function. `P` allocates a zeroed buffer big enough for the passed scalar and hands the pointer to the callee - typical use is for functions like `uuid_generate` that write into a caller-supplied buffer.

Synopsis

```
use Peta::FFI;

my $libm = Peta::FFI::dlopen("libm.so.6");
my $root = Peta::FFI::call($libm, "sqrt", "(d)d", 2.0);
Peta::FFI::dlclose($libm);
print $root, "\n";          # 1.4142135623731
```

Modules

- [Peta::FFI::Cairo](#) — Native reimplementaion of the Perl Cairo module against `libcairo.so.2`.
- [Peta::FFI::Libc](#) — Call common `libc` functions from Perl without writing a single line of FFI glue.
- [Peta::FFI::UUID](#) — Generate and parse RFC 4122 UUIDs via the system's `libuuid` library.

Functions

Library loading

`ffi_dlopen`

Open a shared library and return a handle suitable for passing to `call`.

`ffi_dlclose`

Release a library handle previously returned by `dlopen`.

Function calls

`ffi_call`

Invoke a C function from an opened library and return its result.

Symbol introspection

`ffi_scan`

List the shared libraries the dynamic linker already knows about.

Peta::FFI::Cairo

Native reimplement of the Perl Cairo module against `libcairo.so.2`.

The upstream Cairo CPAN distribution (v1.109) is a thin XS shim over the cairo 2d graphics C library. This module reproduces its observable Perl behavior (the “phenotype”) by calling into `libcairo.so.2` directly through `dlopen/dlsym`, without a build-time C binding.

Although the Rust source lives under `Peta::FFI::Cairo`, the Perl-visible packages are the real Cairo, `Cairo::Context`, `Cairo::Surface`, `Cairo::ImageSurface`, etc. - so ported cairo tests and existing Perl code see the same API they would from the XS module.

Architecture: per-package plug-in modules

`mod.rs` is the FOUNDATION: the shared `dlopen` handle, the `string<->int` enum conversion tables, the blessing/refcount contract, the struct-as-hashref marshallers, the status-to-`croak` helper, and the base `Cairo::version/capability` subs. Each Perl-visible object package (`Cairo::Context`, `Cairo::Surface`, `Cairo::Pattern`, ...) lives in its OWN sibling file `Cairo/<Pkg>.rs` and plugs into `boot()` through a NARROW, STABLE contract so that the seven package files can be authored in PARALLEL without touching shared state.

Crucially, each package resolves its OWN cairo symbols locally (via `cairo_dlsym` over the shared `cairo_handle`), so a package agent never edits a central symbol table. The foundation resolves only the two base-namespace symbols (`cairo_version`, `cairo_version_string`).

THE PER-PACKAGE PLUG-IN CONTRACT (read this before authoring a package)

Every `Cairo/<Pkg>.rs` file must expose EXACTLY these two items and nothing else that `mod.rs` depends on:

```
// The fully-qualified Perl sub names this package installs, paired
// with their XS thunks. `boot()` newXS-registers every entry. Names
// carry the REAL Cairo namespace (e.g. "Cairo::Context::create").
pub(crate) const XS_FUNCTIONS:
    &[(&str, unsafe extern "C" fn(*mut CV))] = &[ /* ... */ ];
```

(continues on next page)

(continued from previous page)

```
// @ISA wiring for this package's classes (set_isa calls), or empty.
// Called once by boot() after all XS subs are registered.
pub(crate) unsafe fn install_isa() { /* ... */ }
```

A package resolves the cairo C symbols it needs itself, e.g.:

```
use super::{cairo_dlsym, cairo_t};
struct Syms { create: unsafe extern "C" fn(*mut cairo_surface_t) ->
    ↪ *mut cairo_t }
static SYMS: OnceLock<Syms> = OnceLock::new();
unsafe fn syms() -> &'static Syms {
    SYMS.get_or_init(|| Syms {
        create: cairo_dlsym(b"cairo_create\0").expect("cairo_create
    ↪"),
    })
}
```

It draws every foundation helper it needs from `super::` (all the `pub(crate)` items below): the opaque `cairo_*_t` types, `bless_ptr`, `extract_ptr`, `bless_surface_noinc`, `nv_hashref`, `check_status`, `set_isa`, `read_doubles`, `cairo_handle`, `cairo_dlsym`, and every `EnumTable` static (`FORMAT`, `STATUS`, `OPERATOR`, ...).

Refcount contract (census section 4)

Every cairo object is a blessed scalar ref whose referent's IV holds the raw `cairo_*_t*`. Constructors (`create*`) return an object with a reference already owned (+1), so we bless the pointer directly. Borrowed getters (`get_source`, `get_target`, `pop_group`) must take an extra reference (`cairo_*_reference`) before blessing. Every `DESTROY` drops exactly one reference via `cairo_*_destroy`.

Modules

- `Peta::FFI::Cairo::Context` — `Cairo::Context` package plug-in.
- `Peta::FFI::Cairo::Font` — `Cairo::Font` family plug-in: `FontFace`, `ToyFontFace`, `ScaledFont`,
- `Peta::FFI::Cairo::Matrix` — `Cairo::Matrix` package plug-in.
- `Peta::FFI::Cairo::Path` — `Cairo::Path` package plug-in - the tied-array live view onto `cairo_path_t*`.
- `Peta::FFI::Cairo::Pattern` — `Cairo::Pattern` family plug-in (`CairoPattern.xs` phenotype).
- `Peta::FFI::Cairo::Region` — `Cairo::Region` package plug-in.
- `Peta::FFI::Cairo::Surface` — `Cairo::Surface` family package plug-in.

Functions

Other Functions

`lib_version`

`Cairo::LIB_VERSION` / `Cairo::version` / `Cairo::lib_version` -> the compile-time-encoded cairo version integer of the loaded library. Note: the Perl-level `Cairo::VERSION` dispatcher (2-arg -> version check, else -> `LIB_VERSION`) lives in the `.pm`; here we provide the underlying `LIB_VERSION` that it and the tests call.

`version_string`

`Cairo::version_string` / `Cairo::lib_version_string` -> the cairo version as a string (e.g. "1.18.4").

`version_encode`

`Cairo::VERSION_ENCODE` / `Cairo::LIB_VERSION_ENCODE` (major,minor,micro) -> encoded int, mirroring `CAIRO_VERSION_ENCODE`. Called as either a function `Cairo::VERSION_ENCODE(1,2,0)` or class method `Cairo->VERSION_ENCODE(1,2,0)`; the class-method form prepends the invocant, so we read the LAST three numeric args.

`has_png`

`Cairo::HAS_PNG_FUNCTIONS` -> whether the loaded `libcairo` exposes PNG output. Probed by `dlsym` presence (census runtime-env note), not a compile-time `ifdef`. The symbol itself lives in the Surface package, so we probe it directly rather than through a central table field.

Peta::FFI::Cairo::Context

`Cairo::Context` package plug-in.

The drawing context (`cairo_t`). This module owns the context `cairo` symbols, resolving them itself via `super::cairo_dlsym` over the shared handle so the foundation's symbol table never carries context entries.

RefCount contract (census section 4): `create` returns a +1 context blessed directly; `DESTROY` drops exactly one reference via `cairo_destroy`. Borrowed getters (`get_source`, `get_target`, `get_group_target`, `get_font_face`) return objects the caller does not own, so we `cairo_*_reference` them before blessing; `pop_group` already returns +1 (matching `newSVCairoPattern_noinc`) so it is blessed directly.

Cross-package ABI assumptions (FLAGGED for review)

Two getters here manufacture objects that live in SIBLING packages, so this file makes allocator/refcount assumptions about how those packages' `DESTROY` will free them. Both are documented at their call sites:

- `get_matrix` / `get_font_matrix` return a `Cairo::Matrix`. Upstream `cairo_perl_copy_matrix` does `New(0, m, 1, cairo_matrix_t); *m =`

src and `Cairo::Matrix::DESTROY` does `Safefree(m)`. We mirror that with a raw `libc::malloc` of `sizeof::<CairoMatrixRepr>()` (6 f64 = 48 bytes), write the matrix into it via `cairo_get_matrix`, and bless the pointer into `Cairo::Matrix`. The `Matrix` package's `DESTROY` MUST free it with the matching `libc::free`. If `Matrix` uses a different allocator, this is a mismatch - hence the shared-repr definition below.

- `get_font_options` returns a `Cairo::FontOptions` allocated with `cairo_font_options_create()` (a +1 cairo object). The `Font` package's `Cairo::FontOptions::DESTROY` MUST call `cairo_font_options_destroy`.

Functions

Other Functions

`context_create`

`Cairo::Context->create($surface) -> blessed Cairo::Context`. Mirrors `cairo_create` with the `_noinc` contract; `ST(0)` is the class (dropped).

`context_destroy`

`$cr->DESTROY -> cairo_destroy`, dropping the wrapper's one reference.

`context_status`

`$cr->status -> status nickname` (enum-string return).

`pop_group`

`$cr->pop_group -> the group as a pattern`. `cairo_pop_group` already returns +1 (newSV-CairoPattern_noinc), so we bless directly, type-dispatched.

`get_source`

`$cr->get_source -> the current source pattern` (borrowed; reference + type-dispatched bless).

`set_dash`

`$cr->set_dash($offset, @dashes)`. Variadic: mirrors the XS which mallocs items - 2 doubles from `ST(2..)` and passes (dashes, ndash, offset).

`get_dash`

`$cr->get_dash -> list($offset, @dashes)`. Mirrors the PPCODE XS: `cairo_get_dash_count` then `cairo_get_dash` into a buffer, push offset first.

has_current_point

`$cr->has_current_point -> bool (>=1.6). dlsym-gated: absent -> croak.`

get_current_point

`$cr->get_current_point -> ($x, $y) OUTLIST.`

in_clip

`$cr->in_clip(x, y) -> bool (>=1.10). dlsym-gated via context_hit.`

select_font_face

`$cr->select_font_face($family, $slant, $weight). utf8 family + 2 enums.`

get_font_options

`$cr->get_font_options -> Cairo::FontOptions.` Allocs a fresh options object (cairo_font_options_create, +1 owned), fills it, blesses into the Font package's FontOptions (whose DESTROY calls cairo_font_options_destroy). See module ABI note.

get_scaled_font

`$cr->get_scaled_font -> Cairo::ScaledFont (>=1.4).` This is a borrowed getter in the XS (ref-INC via newSVCairoScaledFont); we take a reference through cairo_scaled_font_reference if present, then bless into the nominal Cairo::ScaledFont (no subclass dispatch for scaled fonts).

show_text

`$cr->show_text($utf8).`

text_path

`$cr->text_path($utf8).`

get_font_face

`$cr->get_font_face -> Cairo::FontFace subclass (borrowed; ref + dispatch).`

font_extents

`$cr->font_extents -> hashref {ascent,descent,height,max_x_advance, max_y_advance}.` Keys fixed by newSVCairoFontExtents.

text_extents

`$cr->text_extents($utf8) -> hashref {x_bearing,y_bearing,width,height, x_advance,y_advance}`. Keys fixed by `newSVCairoTextExtents`.

get_target

`$cr->get_target -> the target surface (borrowed; ref + type-dispatch)`.

get_group_target

`$cr->get_group_target -> the current group's target surface (>=1.2)`.

tag_begin

`$cr->tag_begin($tag_name, $attributes) (>=1.16)`. dlsym-gated.

tag_end

`$cr->tag_end($tag_name) (>=1.16)`. dlsym-gated.

show_glyphs

`$cr->show_glyphs(@glyphs) (Cairo.xs)`. Variadic LIST of glyph hashrefs; marshal each with `SvCairoGlyph`, then `cairo_show_glyphs`. Void return.

glyph_path

`$cr->glyph_path(@glyphs) (Cairo.xs)`. Variadic LIST; `cairo_glyph_path`. Void return.

glyph_extents

`$cr->glyph_extents(@glyphs) (Cairo.xs) -> text_extents hashref {x_bearing,y_bearing,width,height,x_advance,y_advance}`. Variadic LIST.

show_text_glyphs

`$cr->show_text_glyphs($utf8, \@glyphs, \@clusters, $cluster_flags) (Cairo.xs, >=1.8)`. utf8 string + glyph arrayref + cluster arrayref + text-cluster-flags. The XS allocates the glyph/cluster arrays with cairo's own `cairo_glyph_allocate` / `cairo_text_cluster_allocate` and frees them with the matching `_free` after the call, so we do the same. Void return.

copy_path

`Cairo::Context::copy_path(Cairo.xs:893) -> cairo_copy_path -> a tied Cairo::Path live view. cairo returns a +1 path; the tie's DESTROY frees it via cairo_path_destroy, so we wrap it directly (newSVCairoPath).`

copy_path_flat

Cairo::Context::copy_path_flat (Cairo.xs:895): same wrapping, flattened path (curves approximated by line segments).

append_path

Cairo::Context::append_path (Cairo.xs:897): take a Cairo::Path (or a plain arrayref-of-hashes) and cairo_append_path it. SvCairoPath recovers the C pointer from tie magic, or marshals an arrayref via path_from_array; the latter is a temp we must free once cairo has copied it (cairo acts on the path immediately, per CairoPath.xs:129-132).

copy_clip_rectangle_list

Cairo::Context::copy_clip_rectangle_list (Cairo.xs:641-651): return the current clip as a list of {x,y,width,height} (nv) hashrefs. Mirrors the XS PPCODE: check list->status (croak on error), push each rectangle, then cairo_rectangle_list_destroy. Not a tie - a plain list return (census 2.2). Requires libcairo >= 1.4 (dlsym-gated).

Peta::FFI::Cairo::Font

Cairo::Font family plug-in: FontFace, ToyFontFace, ScaledFont, FontOptions, and the (deferred) FtFontFace.

This module owns the font cairo symbols (cairo_font_face_*, cairo_toy_font_face_*, cairo_scaled_font_*, cairo_font_options_*), resolving them itself via super::cairo_dlsym over the shared handle so the foundation's symbol table never carries font entries. It mirrors CairoFont.xs (census sections 2.7 / 2.8).

Cross-package refcount / ABI contract (Context depends on this)

The Context and Surface packages already bless cairo objects into OUR packages and rely on OUR DESTROY to balance the reference they took. Getting these three destructors wrong leaks or double-frees:

- Context/Surface get_font_options allocate via cairo_font_options_create() (+1 owned) and bless into Cairo::FontOptions. So Cairo::FontOptions::DESTROY MUST call cairo_font_options_destroy.
- Context get_scaled_font blesses a Cairo::ScaledFont AFTER taking a reference (cairo_scaled_font_reference). So Cairo::ScaledFont::DESTROY MUST call cairo_scaled_font_destroy.
- Context get_font_face blesses a type-dispatched FontFace subclass after cairo_font_face_reference. So Cairo::FontFace::DESTROY MUST call cairo_font_face_destroy (inherited by ToyFontFace/FtFontFace via @ISA, so registering it once on the base is sufficient).

RefCount contract (census section 4)

FontFace/ScaledFont/FontOptions are cairo-refcounted. Our constructors (ToyFontFace::create, ScaledFont::create, FontOptions::create) return +1 and bless the pointer directly. Borrowed getters (ScaledFont::get_font_face) cairo_*_reference before blessing. Every DESTROY drops exactly one reference. FontFace blessing is TYPE-DISPATCHED via cairo_font_face_get_type (0->ToyFontFace, 1->FtFontFace, else FontFace).

Matrix / FontOptions cross-package allocation (see Context ABI note)

ScaledFont::get_font_matrix/get_ctm/get_scale_matrix return a Cairo::Matrix, which upstream is a libc::malloced cairo_matrix_t (6 f64 = 48 bytes) freed by Cairo::Matrix::DESTROY's libc::free; we mirror that exactly. get_font_options allocates via cairo_font_options_create() and blesses into Cairo::FontOptions.

Functions

Other Functions

face_status

\$face->status -> status nickname (enum-string return).

face_get_type

\$face->get_type -> font-type nickname (>=1.2).

face_destroy

\$face->DESTROY -> cairo_font_face_destroy, dropping the wrapper's one reference. Inherited by ToyFontFace/FtFontFace via @ISA. Balances the reference Context's get_font_face took (see module ABI contract).

toy_create

Cairo::ToyFontFace->create(\$family, \$slant, \$weight) -> blessed Cairo::ToyFontFace. cairo_toy_font_face_create returns +1; bless directly. ST(0) is the class (dropped). Mirrors the XS C_ARGS (family, slant, weight).

toy_get_family

\$face->get_family -> the font family (UTF-8 string).

toy_get_slant

\$face->get_slant -> slant nickname.

toy_get_weight

`$face->get_weight -> weight nickname.`

sf_create

`Cairo::ScaledFont->create($face, $font_matrix, $ctm, $options) -> blessed Cairo::ScaledFont. cairo_scaled_font_create returns +1; bless directly. ST(0) is the class. Mirrors the XS C_ARGS (font_face, font_matrix, ctm, options).`

sf_status

`$sf->status -> status nickname.`

sf_get_type

`$sf->get_type -> font-type nickname (>=1.2).`

sf_extents

`$sf->extents -> font_extents hashref {ascent,descent,height,max_x_advance,max_y_advance}.`

sf_text_extents

`$sf->text_extents($utf8) -> text_extents hashref {x_bearing,y_bearing,width,height,x_advance,y_advance} (>=1.2).`

sf_get_font_face

`$sf->get_font_face -> Cairo::FontFace subclass (>=1.2, borrowed: reference + type-dispatched bless).`

sf_get_font_matrix

`$sf->get_font_matrix -> Cairo::Matrix (>=1.2, out-param copy).`

sf_get_ctm

`$sf->get_ctm -> Cairo::Matrix (>=1.2, out-param copy).`

sf_get_scale_matrix

`$sf->get_scale_matrix -> Cairo::Matrix (>=1.8, out-param copy).`

sf_get_font_options

`$sf->get_font_options` -> Cairo::FontOptions (>=1.2). Allocs a fresh options object (cairo_font_options_create, +1 owned), fills it, blesses into Cairo::FontOptions (whose DESTROY calls cairo_font_options_destroy).

sf_destroy

`$sf->DESTROY` -> cairo_scaled_font_destroy, dropping the wrapper's one reference. Balances the reference Context's get_scaled_font took (see ABI note).

sf_glyph_extents

`$sf->glyph_extents(@glyphs)` (CairoFont.xs) -> text_extents hashref {x_bearing,y_bearing,width,height,x_advance,y_advance}. Variadic glyph LIST, same prologue as the Context variant.

sf_text_to_glyphs

`$sf->text_to_glyphs($x, $y, $utf8)` (CairoFont.xs, >=1.8). Returns a LIST: (status_nickname, glyphs_ref, clusters_ref, cluster_flags_ref). The status SV is ALWAYS pushed; the three data refs are pushed only when status is SUCCESS (the PPCODE mirrors this exactly). cairo ALLOCATES the glyph and cluster arrays; we build the Perl hashref arrays from them, then free the C arrays with `cairo_glyph_free` / `cairo_text_cluster_free`.

opts_create

Cairo::FontOptions->create -> blessed Cairo::FontOptions. `cairo_font_options_create` takes void (C_ARGS void) and returns +1.

opts_status

`$opts->status` -> status nickname.

opts_merge

`$opts->merge($other)`.

opts_equal

`$opts->equal($other)` -> bool.

opts_hash

`$opts->hash` -> UV (cairo_font_options_hash returns unsigned long).

opts_destroy

\$opts->DESTROY -> cairo_font_options_destroy, dropping the wrapper's one reference. Balances the +1 from Context/Surface get_font_options and our own create (see module ABI contract).

opts_set_color_mode

\$opts->set_color_mode(\$mode) (>=1.18). Enum: default/no-color/color.

opts_get_color_mode

\$opts->get_color_mode (>=1.18) -> mode nickname.

opts_set_color_palette

\$opts->set_color_palette(\$index) (>=1.18). UV index (CPAL palette).

opts_get_color_palette

\$opts->get_color_palette (>=1.18) -> UV palette index.

opts_set_custom_palette_color

\$opts->set_custom_palette_color(\$index, \$r, \$g, \$b, \$a) (>=1.18).

opts_get_custom_palette_color

\$opts->get_custom_palette_color(\$index) (>=1.18) -> list (status, r, g, b, a). The C signature returns a status and writes r/g/b/a out; we surface the status nickname first (a missing entry yields e.g. 'invalid-index'), then the four color components.

ft_create

Deferred by DEPENDENCY, not by effort. Per CairoFt.xs, this method takes a blessed Font::FreeType::Face object and extracts a raw FT_Face from its IV (SvIV(SvRV(face))), then calls cairo_ft_font_face_create_for_ft_face. That FT_Face can only exist if Font::FreeType produced it - and Font::FreeType is itself an XS-over-libfreetype module that pperl does not yet provide (absent from perl5-modules/ and the system perl). There is no faithful way to accept a Font::FreeType::Face when nothing can create one. Unblocking this requires porting Font::FreeType as its own native module (a libfreetype binding); until then the phenotype has no valid input. The FtFontFace->FontFace @ISA is still installed so isa/can are correct.

Peta::FFI::Cairo::Matrix

Cairo::Matrix package plug-in.

The affine transform value type (cairo_matrix_t). Unlike every other cairo object, a matrix is NOT cairo-refcounted: upstream CairoMatrix.xs models it as a blessed pointer

to a HEAP `cairo_matrix_t` allocated with Perl's `New` (`== malloc`) and released with `SafeFree` (`== free`) in `DESTROY`. `cairo_perl_copy_matrix` does `New(0, dst, 1, cairo_matrix_t)` then field-copies. All matrix-producing methods return a fresh copy.

CROSS-PACKAGE ABI CONTRACT (review-critical)

`Cairo::Context`'s `get_matrix` / `get_font_matrix` allocate a matrix with `libc::malloc(sizeof::<CairoMatrixRepr>())` (6 f64 = 48 bytes), let `cairo` fill it, and bless the raw pointer into `Cairo::Matrix` - handing it to THIS package to eventually free. Therefore:

- our constructors MUST allocate the referent with `libc::malloc(48)` (NOT `Box`, NOT `Vec`) so every `Cairo::Matrix` pointer is uniformly `libc::free`-compatible regardless of who created it;
- our `DESTROY` MUST `libc::free` the pointer. A `sizeof::<CairoMatrix>() == 48` assertion in the unit tests guards the shared layout against drift. Any mismatch here double-frees or leaks the matrices `Context` manufactures.

This module owns the `cairo_matrix_*` symbols, resolving them itself via `super::cairo_dlsym` over the shared handle. The matrix API is stable since `cairo 1.0`, so there is no version gating.

Functions

Other Functions

`init`

`Cairo::Matrix->init($xx, $yx, $xy, $yy, $x0, $y0) -> blessed matrix`. Mirrors the XS: `cairo_matrix_init` into a fresh `malloc`'d struct, bless.

`init_identity`

`Cairo::Matrix->init_identity -> blessed identity matrix`.

`init_translate`

`Cairo::Matrix->init_translate($tx, $ty) -> blessed translation matrix`.

`init_scale`

`Cairo::Matrix->init_scale($sx, $sy) -> blessed scale matrix`.

`init_rotate`

`Cairo::Matrix->init_rotate($radians) -> blessed rotation matrix`.

translate

`$matrix->translate($tx, $ty) -> void`; `cairo_matrix_translate` in place.

scale

`$matrix->scale($sx, $sy) -> void`; `cairo_matrix_scale` in place.

rotate

`$matrix->rotate($radians) -> void`; `cairo_matrix_rotate` in place.

invert

`$matrix->invert -> status` nickname. `cairo_matrix_invert` mutates the matrix in place and returns a `cairo_status_t`; we surface the nickname (mirrors the XS `cairo_status_t` return type -> `cairo_status_to_sv`).

multiply

`$a->multiply($b) -> a` NEW blessed `Cairo::Matrix` = $a \times b$.

transform_distance

`$matrix->transform_distance($dx, $dy) -> ($dx', $dy')`. IN_OUTLIST: the matrix's linear part (no translation) transforms the distance vector.

transform_point

`$matrix->transform_point($x, $y) -> ($x', $y')`. IN_OUTLIST: the full affine transform (including translation) applied to the point.

destroy

`$matrix->DESTROY -> libc::free` the referent. Mirrors the XS `SafeFree(matrix)`; there is NO `cairo` call (a matrix is not refcounted). The pointer was allocated with `libc::malloc` either here (`init/multiply`) or by `Context::get_matrix` - both use the same 48-byte allocator, so this free is always valid. See the module ABI contract.

Peta::FFI::Cairo::Path

`Cairo::Path` package plug-in - the tied-array live view onto `cairo_path_t*`.

`Cairo::Path` is not a plain blessed pointer like the other `cairo` objects: it is a **tied ARRAY of tied HASHes of tied ARRAYs** presenting a live view onto a C `cairo_path_t*`. This file is a 1:1 mirror of upstream `CairoPath.xs`; read that alongside this for the semantics. The four Perl-visible packages map to the XS's four MODULE blocks:

- `Cairo::Path` (tied ARRAY) - one element per path command.
- `Cairo::Path::Data` (tied HASH) - keys “type” / “points” for a command.

- `Cairo::Path::Points`(tied ARRAY) - the points of one command.
- `Cairo::Path::Point` (tied 2-ARRAY)- the (x, y) of one point.

Magic representation (mirrors `create_tie / cairo_perl_mg_get`)

`create_tie` (XS lines 41-69) attaches TWO magics to the referent AV/HV:

1. `PERL_MAGIC_tied` ('P', obj = the blessed RV) so Perl routes element access through this package's `FETCH/STORE/...` methods.
2. `PERL_MAGIC_ext` ('~', mg_ptr = the C object pointer; namlen = 0), with `mg->mg_private = MY_MAGIC_SIG (0xCAFE)` marking it as ours.

The C pointer carried in the ext slot IS the element/point offset: the top-level tie carries `cairo_path_t*`; each `Cairo::Path::Data / ::Points` tie carries the `cairo_path_data_t*` pointing at that command's header; each `Cairo::Path::Point` tie carries the `cairo_path_data_t*` pointing at that specific point union. There is no separate index stored - the pointer already threads the location, exactly as the XS does (e.g. `newSVCairoPathPoint(&data[index + 1])`, XS line 427/444).

Recovery mirrors `cairo_perl_mg_get` (XS lines 29-37): given the invocant blessed RV `sv`, walk `SvRV(sv)`'s magic chain for a `PERL_MAGIC_ext` slot whose `mg_private == 0xCAFE`, and read `mg_ptr`.

VERSION SCOPE

Path is stable cairo API; there is no version gating and no post-1.109 superset here.

Functions

Other Functions

`path_destroy`

`$path->DESTROY -> cairo_path_destroy` (XS:283-295). Recover the pointer through the ext-magic slot (via `mg_get`, matching `SvCairoPath`); if present, destroy it. The 5.6.x `mg_ptr-unset` branch is not mirrored (ppperl targets 5.42+, XS lines 289-293 are guarded out).

`path_fetchsize`

`Cairo::Path::FETCHSIZE` (XS:297-305): number of path commands. Walk `data[]` stepping by `header.length`, counting each command.

`path_fetch`

`Cairo::Path::FETCH` (XS:307-320): the *i*-th command as a tied `Cairo::Path::Data`. Walk by `header.length`, counting to `index`; return the tie carrying that command's `cairo_path_data_t*`, else `undef`.

data_fetch

Cairo::Path::Data::FETCH (XS:337-351): key “type” -> nickname; “points” -> a tied Cairo::Path::Points; anything else croaks.

data_store

Cairo::Path::Data::STORE (XS:353-367): only “points” is writable. Rewrite this command’s coordinates from the arrayref value (keeping the existing header.type), and return a fresh tied Points view (matching the XS RETVAL).

data_exists

Cairo::Path::Data::EXISTS (XS:369-379): true for “type”/“points” only.

data_firstkey

Cairo::Path::Data::FIRSTKEY (XS:381-385): always “type”.

data_nextkey

Cairo::Path::Data::NEXTKEY (XS:387-395): “type” -> “points”, else undef.

points_fetchsize

Cairo::Path::Points::FETCHSIZE (XS:412-419): n_points for this command.

points_fetch

Cairo::Path::Points::FETCH (XS:421-432): the i-th point of this command as a tied Cairo::Path::Point carrying &data[index + 1], else undef.

points_store

Cairo::Path::Points::STORE (XS:434-454): write x/y of the i-th point back into the C buffer from the [x,y] arrayref value; return a tied Point view.

point_fetchsize

Cairo::Path::Point::FETCHSIZE (XS:471-475): always 2.

point_fetch

Cairo::Path::Point::FETCH (XS:477-494): index 0 -> x, 1 -> y, read straight from the C buffer’s point union; else undef.

point_store

Cairo::Path::Point::STORE (XS:496-513): the LIVE write-back. index 0 sets point.x, 1 sets point.y - straight into the C buffer - and returns the new value as an NV (matching newSVnv(data->point.x = value)).

Peta::FFI::Cairo::Pattern

Cairo::Pattern family plug-in (CairoPattern.xs phenotype).

Covers the base Cairo::Pattern plus its five concrete subclasses: Cairo::SolidPattern, Cairo::SurfacePattern, Cairo::Gradient (abstract - only add_color_stop_* / get_color_stops), and the two gradient leaves Cairo::LinearGradient / Cairo::RadialGradient. This module owns the cairo_pattern_* symbols, resolving them itself via super::cairo_dlsym over the shared handle so the foundation's table never carries pattern entries.

RefCount contract (census section 4)

A pattern is cairo-refcounted. Every constructor here (create_rgb/create_rgba/create_for_surface/create_linear/radial) returns a +1 pattern (the XS cairo_pattern_t_noinc return type), so we bless the pointer directly. DESTROY drops exactly one reference via cairo_pattern_destroy.

Type-dispatched blessing (census section 4)

Constructors do NOT bless into the nominal package. The XS cairo_pattern_to_sv calls cairo_pattern_get_type and picks the concrete subclass. bless_pattern_noinc below replicates that dispatch (0->SolidPattern, 1->SurfacePattern, 2->LinearGradient, 3->RadialGradient, else Pattern), byte-identical to the private helper in Context.rs. create_rgb returning a solid pattern therefore comes back blessed Cairo::SolidPattern, matching upstream.

Cross-package ABI: get_matrix -> Cairo::Matrix (review-critical)

get_matrix mirrors cairo_perl_copy_matrix: allocate a fresh cairo_matrix_t on the heap, let cairo fill it, bless into Cairo::Matrix. We allocate with libc::malloc(size_of:<CairoMatrix>()) (6 f64 = 48 bytes) EXACTLY as Context.rs's get_matrix and Matrix.rs's constructors do, so the resulting pointer is uniformly libc::free-compatible with Cairo::Matrix::DESTROY. Any other allocator here would double-free or leak.

get_surface -> type-dispatched Cairo::Surface (borrowed)

get_surface returns a surface the pattern still owns (borrowed getter). Per the refcount contract we cairo_surface_reference it before blessing, then type-dispatch on cairo_surface_get_type - the same table as the foundation's bless_surface_noinc, replicated locally so this file stays independent of the Surface package.

Superset: gradient dithering (cairo 1.18)

`set_dither/get_dither(cairo_dither_t)` have no 1.109 XS original; the census VERSION SCOPE mandates binding them into the Pattern package as a clean superset addition (the gradient-banding fix). The symbols are dlsym-gated: if the loaded libcairo predates 1.18 they resolve to None and the method croaks “requires cairo >= 1.18”.

Functions

Other Functions

`pattern_destroy`

`$pattern->DESTROY` -> `cairo_pattern_destroy`, dropping the wrapper’s one reference.

`set_matrix`

`$pattern->set_matrix($matrix)` -> `cairo_pattern_set_matrix`. ST(1) is a Cairo::Matrix (blessed pointer to a `cairo_matrix_t`).

`get_matrix`

`$pattern->get_matrix` -> a fresh Cairo::Matrix. Mirrors `cairo_perl_copy_matrix`: heap-alloc a `cairo_matrix_t` with `libc::malloc` (matching Cairo::Matrix::DESTROY’s `libc::free`), let cairo fill it, bless. See the module ABI note.

`status`

`$pattern->status` -> status nickname (enum-string return).

`set_dither`

`$pattern->set_dither($dither)` (cairo 1.18 superset). dlsym-gated: croak if the loaded libcairo predates 1.18 (symbol absent).

`get_dither`

`$pattern->get_dither` -> dither nickname (cairo 1.18 superset).

`create_rgb`

`Cairo::SolidPattern->create_rgb($r,$g,$b)` -> +1 pattern, type-dispatch blessed. ST(0) is the class (dropped); the 3 doubles follow.

`create_rgba`

`Cairo::SolidPattern->create_rgba($r,$g,$b,$a)` -> +1 pattern.

get_rgba

`$pattern->get_rgba -> list ($r,$g,$b,$a)` (cairo >= 1.4). PPCODE list-return with a status-check (CAIRO_PERL_CHECK_STATUS -> croak).

surface_pattern_create

`Cairo::SurfacePattern->create($surface) -> +1 pattern` (cairo_pattern_create_for_surface), type-dispatch blessed. ST(0) is the class (dropped), ST(1) is the surface.

get_surface

`$pattern->get_surface -> the pattern's surface` (cairo >= 1.4). Borrowed getter: reference + type-dispatch bless. Status-check per the XS.

add_color_stop_rgb

`$gradient->add_color_stop_rgb($offset,$r,$g,$b)`.

add_color_stop_rgba

`$gradient->add_color_stop_rgba($offset,$r,$g,$b,$a)`.

get_color_stops

`$gradient->get_color_stops -> list of 5-elem arrayrefs [offset,r,g,b,a]` (cairo >= 1.4). Mirrors the PPCODE loop: `get_color_stop_count` then `get_color_stop_rgba` per index, status-checked.

linear_create

`Cairo::LinearGradient->create($x0,$y0,$x1,$y1) -> +1 pattern`.

get_points

`$linear->get_points -> list ($x0,$y0,$x1,$y1)` (cairo >= 1.4).

radial_create

`Cairo::RadialGradient->create($cx0,$cy0,$r0,$cx1,$cy1,$r1) -> +1 pattern`.

get_circles

`$radial->get_circles -> list of 6 doubles ($x0,$y0,$r0,$x1,$y1,$r1)` (cairo >= 1.4).

Peta::FFI::Cairo::Region

Cairo::Region package plug-in.

A region (`cairo_region_t`) is a set of integer-aligned rectangles. The ENTIRE Region API is `cairo >= 1.10` (`CairoRegion.xs` is guarded by `CAIRO_VERSION >= 1.10`), so every method here is `dlsym`-gated: if `cairo_region_create` is absent the constructor croaks with a version hint. The system `cairo` (1.18.4) has all of it, so on this host every symbol resolves.

This module owns the region `cairo` symbols, resolving them itself via `super::cairo_dlsym` over the shared handle so the foundation's symbol table never carries region entries.

RefCount contract (census section 4): a region is a `cairo-refcounted` object. `cairo_region_create*` all return `+1`, so `create` blesses the pointer directly into `Cairo::Region`; `DESTROY` drops exactly one reference via `cairo_region_destroy`.

rectangle_int hashref convention (Cairo.xs newSVCairoRectangleInt /

SvCairoRectangleInt)

`cairo_rectangle_int_t` is four C ints `{x, y, width, height}`. On the Perl side it is a plain (unblessed) hashref with exactly those keys, whose values are INTEGERS (`newSViv`, not `newSVnv`). `super::nv_hashref` builds float hashrefs (extents/rectangle), so it does NOT fit here; we use a local `iv_hashref` helper (`newHV + hv_store` of `newSViv`) for the int case. Reading a rectangle mirrors `SvCairoRectangleInt`: the struct is zeroed, then each of the four keys is read with `SvIV` only if present and defined (`alloc_temp` gives a zeroed struct, so unset/undef keys stay 0).

Functions

Other Functions

region_create

`Cairo::Region->create(...)`. Variadic ctor dispatching on arg count, mirroring `CairoRegion.xs cairo_region_create(class, ...)`: `items == 1` (class only) -> `cairo_region_create()` `items == 2` -> `cairo_region_create_rectangle($rect)` `items > 2` -> `cairo_region_create_rectangles(@rects)` All three `cairo` ctors return `+1`, so we bless the pointer directly.

region_destroy

`$region->DESTROY` -> `cairo_region_destroy`, dropping the one reference.

region_status

`$region->status` -> `status nickname` (enum-string return).

region_get_extents

`$region->get_extents -> {x,y,width,height} hashref.` Mirrors the XS PREINIT `rect`
`+ cairo_region_get_extents(region, &rect) -> RETVAL.`

region_num_rectangles

`$region->num_rectangles -> int.`

region_get_rectangle

`$region->get_rectangle($nth) -> {x,y,width,height} hashref.`

region_is_empty

`$region->is_empty -> bool.` `cairo_bool_t` is an int (0/1) surfaced as `T_UV`.

region_contains_point

`$region->contains_point($x, $y) -> bool.`

region_contains_rectangle

`$region->contains_rectangle($rect) -> overlap nickname (in/out/part).`

region_equal

`$region->equal($other) -> bool.`

region_translate

`$region->translate($dx, $dy) (void).`

Peta::FFI::Cairo::Surface

`Cairo::Surface` family package plug-in.

One file per the seven-file fan-out plan covers the whole surface family, because every concrete surface (`Cairo::ImageSurface`, `Cairo::PdfSurface`, `Cairo::PsSurface`, `Cairo::SvgSurface`, `Cairo::RecordingSurface`) is a `Cairo::Surface` subclass blessed by the SAME type-dispatch table (`cairo_surface_get_type`), and they share the surface-family cairo symbols. This module resolves those symbols itself via `super::cairo_dlsym` over the shared handle, so the foundation's symbol table never carries surface entries and parallel package agents never contend on it.

RefCount contract (census section 4): every `create*` constructor and `create_similar / create_for_rectangle` return a +1 surface, blessed directly (`_noinc`). `DESTROY` drops exactly one reference via `cairo_surface_destroy`. Enum-returning methods (`status`, `get_type`, `get_content`, `get_format`) surface the cairo int as its string nickname.

Version scope (census top block): the 1.109 phenotype is the floor; we bind it 1:1 against the loaded libcairo 1.18.4. Backend-specific ctors and the

=1.16 PDF metadata/outline methods are dlsym-gated: on a feature-stripped libcairo the symbol resolves to None and the method croaks with a targeted “built without X support” message rather than dereferencing null.

Deferred (later dedicated pass, census section 5 hard-parts 1 and 4): the stream ctors and `write_to_png_stream / create_from_png_stream` need a Perl write/read callback re-entered from inside libcairo through a C trampoline; `set_mime_data / get_mime_data` cross an SV lifetime into libcairo. These are registered as croaking stubs so the API surface is present but flags the unimplemented path explicitly.

Functions

Other Functions

`image_surface_create`

`Cairo::ImageSurface->create($format, $width, $height) -> blessed surface.` Mirrors `cairo_image_surface_create`; `C_ARGS` drops the leading class. The constructor owns +1, so we type-dispatch `bless` without an extra reference.

`image_surface_create_for_data`

`Cairo::ImageSurface->create_for_data($data, $format, $width, $height, $stride).` Mirrors `cairo_image_surface_create_for_data`: the raw bytes are borrowed by cairo (not copied), so the SV must outlive the surface - matching the XS, which passes the SV's PV pointer straight through.

`image_get_data`

`$image_surface->get_data -> SV string of height*stride bytes, or undef.` Mirrors the XS which reads height and stride to size the SV.

`image_get_format`

`$image_surface->get_format -> format nickname (enum-string return).`

`image_create_from_png`

`Cairo::ImageSurface->create_from_png($filename) -> blessed surface.` PNG is a build option; the symbol is dlsym-gated (census runtime-env note).

`surface_destroy`

`$surface->DESTROY -> cairo_surface_destroy`, dropping the one reference this wrapper owns (census section 4).

surface_status

`$surface->status` -> status nickname (enum-string return).

surface_get_type

`$surface->get_type` -> surface-type nickname (enum-string return).

surface_get_content

`$surface->get_content` -> content nickname (enum-string return).

surface_write_to_png

`$surface->write_to_png($filename)` -> status nickname. PNG is dlsym-gated.

surface_create_similar

`create_similar` in its dual class/instance form (CairoSurface.xs L313):
`Cairo::Surface->create_similar($other, $content, $w, $h)` (items==5)
`$other->create_similar($content, $w, $h)` (items==4) The extra leading class shifts the arg offset by one. `content` is an enum string; the result is a +1 surface blessed type-dispatched.

surface_set_device_offset

`$surface->set_device_offset($x, $y)`.

surface_get_device_offset

`$surface->get_device_offset` -> (`$x_offset`, `$y_offset`) (OUTLIST list).

surface_set_fallback_resolution

`$surface->set_fallback_resolution($x_ppi, $y_ppi)`.

surface_get_fallback_resolution

`$surface->get_fallback_resolution` -> (`$x_ppi`, `$y_ppi`) (OUTLIST list).

surface_get_font_options

`$surface->get_font_options` -> blessed `Cairo::FontOptions`. Mirrors the XS CODE block: allocate a fresh options with `cairo_font_options_create`, fill it via `cairo_surface_get_font_options`, then bless the +1 pointer directly (font options are their own object owned by the Font package).

surface_mark_dirty_rectangle

`$surface->mark_dirty_rectangle($x, $y, $width, $height)` (four ints).

surface_has_show_text_glyphs

`$surface->has_show_text_glyphs -> bool` (0/1).

surface_create_for_rectangle

`Cairo::Surface->create_for_rectangle($target, $x, $y, $width, $height)`. Mirrors the XS class, target, x, y, w, h with C_ARGS dropping class; the result is a +1 surface blessed type-dispatched.

surface_supports_mime_type

`$surface->supports_mime_type($mime_type) -> bool` (0/1).

pdf_restrict_to_version

`$pdf->restrict_to_version($version)` (enum string arg).

pdf_set_metadata

`$pdf->set_metadata($metadata, $utf8)` (≥ 1.16). metadata is an enum string.

pdf_set_page_label

`$pdf->set_page_label($utf8)` (≥ 1.16).

pdf_set_thumbnail_size

`$pdf->set_thumbnail_size($width, $height)` (≥ 1.16).

pdf_add_outline

`$pdf->add_outline($parent_id, $utf8, $link_attris, $flags) -> int` (≥ 1.16). flags is the outline-flags enum as an integer bitmask (not a nickname in the XS - it is a plain int typemap for this method).

pdf_outline_root

`Cairo::PdfSurface::OUTLINE_ROOT -> 0` (CAIRO_PDF_OUTLINE_ROOT constant).

ps_dsc_comment

`$ps->dsc_comment($comment)`.

ps_restrict_to_level

`$ps->restrict_to_level($level)` (enum string arg).

ps_set_eps

`$ps->set_eps($bool)`.

ps_get_eps

`$ps->get_eps -> bool` (0/1).

svg_restrict_to_version

`$svg->restrict_to_version($version)` (enum string arg).

recording_create

`Cairo::RecordingSurface->create($content, $extents_or_undef)`. Mirrors the XS class, `content`, `cairo_rectangle_t_ornull *extents`: `content` is an enum string; `extents` is either an unblessed `{x,y,width,height}` hashref (a bounded recording surface) or `undef/absent` (unbounded, NULL rectangle).

recording_ink_extents

`$recording->ink_extents -> ($x0, $y0, $width, $height)` (OUTLIST list).

recording_get_extents

`$recording->get_extents -> {x,y,width,height}` hashref, or `undef` when the surface is unbounded. Mirrors the XS: the `cairo_bool_t` return gates whether the filled rectangle is returned at all.

format_stride_for_width

`Cairo::Format::stride_for_width($format, $width) -> int`. Mirrors `cairo_format_stride_for_width`; `format` is an enum string. Called as a plain function (not a method), so the `format` is `ST(0)`.

write_to_png_stream

`$surface->write_to_png_stream($closure [, $data]) -> status nickname`. One-shot `write_func`: the closure lives only for the call, so we free it immediately afterwards (no `set_user_data`). Mirrors `CairoSurface.xs cairo_surface_write_to_png_stream`. PNG is `dlsym`-gated.

`create_from_png_stream`

`Cairo::ImageSurface->create_from_png_stream($closure [, $data])` -> blessed surface. `read_func` ctor; +1 surface blessed type-dispatched. The closure is consumed only during the ctor call, so free it immediately after. PNG gated.

`set_mime_data`

`$surface->set_mime_data($mime_type, $data)` -> status nickname (>=1.10). Mirrors `CairoSurface.xs`: `SvREFCNT_inc` the passed `$data` SV (NOT a copy), hand its raw bytes to cairo, and register `data_destroy` (`SvREFCNT_dec`) so cairo owns exactly one reference until it drops the buffer. The SV's PV must stay valid for the surface's life, which the retained reference guarantees.

`get_mime_data`

`$surface->get_mime_data($mime_type)` -> SV string, or undef if the surface carries no data for that MIME type (>=1.10). cairo writes an out double- pointer + length; the XS builds `newSVpvn(data, length)`. We return undef when cairo reports a null data pointer (no attached blob) rather than an empty string, since a genuine zero-length blob is indistinguishable and undef is the more useful phenotype for “absent”.

Peta::FFI::Libc

Call common libc functions from Perl without writing a single line of FFI glue.

Pre-baked libc bindings (Layer 1). Each function is an `unsafe extern "C" fn(*mut CV)` XS wrapper that calls libc directly via the `libc` crate.

Peta::FFI::UUID

Generate and parse RFC 4122 UUIDs via the system's `libuuid` library.

`dlopen`-based `libuuid` bindings. `boot()` attempts the `dlopen`; if `UUID_FUNCS` is `None` afterwards, we register stub XS functions that croak with install instructions.

`ffi_dlopen`

Open a shared library and return a handle suitable for passing to `call`.

Synopsis

```
my $lib = Peta::FFI::dlopen("libm.so.6");
my $lib = Peta::FFI::dlopen("/usr/lib/libuuid.so.1"); # absolute_
↳path
```

What you get back

An opaque integer handle. Treat it as a token: pass it to `call` and later to `dlclose`, but do not do arithmetic on it. Store it in a lexical and keep it alive as long as you want to call into the library.

The argument is either a bare soname (e.g. `"libm.so.6"`) that the dynamic linker resolves through the usual `LD_LIBRARY_PATH` / `ld.so.cache` machinery, or an absolute path to an `.so` file. Both `RTLD_NOW` and `RTLD_GLOBAL` are set - symbols are resolved eagerly and become visible to subsequently loaded libraries.

Edge cases

- **Library not found** - croaks with the message reported by `dlerror`, e.g. `Peta::FFI::dlopen: libnope.so: cannot open shared object file: No such file or directory.`
- **Name contains a null byte** - croaks with `Peta::FFI::dlopen: invalid library name (contains null byte).`
- **No argument** - croaks with `Peta::FFI::dlopen: requires library name.`
- **Calling `dlopen` repeatedly with the same name** - returns the same (reference-counted) handle each time. Pair every successful `dlopen` with a matching `dlclose` if you want the library to eventually unload.

ffi_call

Invoke a C function from an opened library and return its result.

Synopsis

```
my $n = Peta::FFI::call($lib, "strlen", "(p)L", "hello");      # 5
my $r = Peta::FFI::call($lib, "sqrt", "(d)d", 2.0);           # 1.4142...
my $s = Peta::FFI::call($lib, "getenv", "(p)p", "PATH");      # /usr/bin:...
Peta::FFI::call($lib, "srand", "(i)v", 42);                   #
void return
```

Arguments

- `$lib` - handle returned by `dlopen`.
- `$func` - symbol name to look up in the library.
- `$sig` - signature string (`args`) `ret`; see the module-level type-code table.
- the remaining arguments - one value per type code in `args`, coerced to the matching C type before the call.

What you get back

A single scalar whose shape depends on the return code:

- `i / l / L` - integer.
- `d / f` - float.
- `p` - the returned `const char *` copied into a Perl string; undef if the C function returned NULL.
- `v` - 1, so the call chains naturally in boolean contexts.
- `P` - pointer returned as a string (same rules as `p`).

Output buffers (P)

When an argument position is `P`, `call` allocates a zeroed buffer sized from the scalar you pass in. If the scalar is a string, its byte length (minimum 16) is used; otherwise the numeric value of the scalar (minimum 16) is taken as the byte size. The callee receives a pointer to that buffer. The buffer is freed after the call returns - if you need to read bytes the C function wrote into it, wrap the call in a helper module such as `Peta::FFI::UUID` that knows how to recover the buffer contents.

Edge cases

- **Wrong argument count** - croaks with `Peta::FFI::call FNAME: expected N args, got M`.
- **Bad signature string** - croaks with `FFI signature error: ... describing the offending character or missing )`.
- **Symbol not found** - croaks with `Peta::FFI::call: symbol not found: FNAME or the message reported by dLError`.
- **Incorrect type codes** - calling a function with a signature that does not match its real prototype is undefined behaviour. Check the C header before writing the signature.
- **String argument contains a null byte** - croaks with `Peta::FFI::call: CString error: ....`

ffi_dlclose

Release a library handle previously returned by `dlopen`.

Synopsis

```
Peta::FFI::dlclose($lib);
```

What you get back

Always 1. The return value exists so the call composes cleanly in expressions; inspecting it tells you nothing about success.

Edge cases

- **No argument** - croaks with `Peta::FFI::dlclose: requires library handle`.
- **Zero or null handle** - silently ignored. Passing `0` is safe and commonly used as an “already closed” sentinel.
- **Double close** - passing a handle that has already been released is undefined behaviour at the `dlclose` level; keep a single owner per handle or null out your copy after closing.
- **Outstanding calls** - closing a library while a pointer or string it returned is still in use is undefined behaviour. Copy everything you need out of the library first.

ffi_scan

List the shared libraries the dynamic linker already knows about.

Synopsis

```
my $libs = Peta::FFI::scan;  
print $libs->{"libm.so.6"}, "\n";           # /usr/lib/libm.so.6  
my @names = grep { /^libssl/ } keys %$libs;
```

What you get back

A hashref mapping each soname to the absolute path of its backing file. The data comes from `ldconfig -p`, so it reflects the system cache, not the full filesystem - a library not on the linker’s search path will not appear here even if it exists on disk.

Use this to discover the right name to feed `dlopen`. When the same soname appears more than once in the cache (e.g. 32-bit and 64-bit copies), the first entry wins.

Edge cases

- **ldconfig not installed or not on \$PATH** - returns an empty hashref rather than croaking; callers should treat an empty result as “no information available”.
- **Duplicate sonames** - only the first path is kept. If you need every match, read `ldconfig -p` yourself.
- **Non-Linux systems** - `ldconfig` is Linux-specific; on other platforms this function will typically return an empty hashref.

5.2.24 SDBM_File

Tie a Perl hash to an on-disk SDBM database so keys and values persist between runs.

SDBM_File is a **tied-hash interface**: you never call FETCH, STORE, DELETE, or EXISTS directly. You bind a hash to a database file with tie, and from then on any ordinary hash syntax - `$h{key}`, `delete $h{key}`, `exists $h{key}`, `keys %h`, `each %h` - is routed by Perl's tie machinery to the uppercase hook methods defined here.

The database itself lives in two files alongside each other, `NAME.dir` and `NAME.pag`, that persist between program runs. Neither `untie` nor program exit removes them - clean-up is the caller's responsibility.

Synopsis

```

use Fcntl;
use SDBM_File;

tie my %h, 'SDBM_File', 'mydb', O_RDWR | O_CREAT, 0666
    or die "tie failed: $!";
$h{greeting} = 'hello';
print $h{greeting};      # hello
untie %h;

```

Persistence

Tying creates (or opens) two files named after the `$basename` argument: `$basename.dir` and `$basename.pag`. Values written through the tied hash land in these files immediately and survive the process. `untie` closes the handle but leaves the files in place for the next run. Use `unlink` if you want them gone.

The six-argument form of `tie` lets you name the two files independently when the `.dir` / `.pag` naming convention does not fit - typically with `File::Temp` handles.

Size limits

SDBM is a fixed-page format. The combined length of a key and its value must not exceed **1008 bytes** (the PAIRMAX constant). A STORE for a too-large pair croaks with `sdbm store returned -1, errno 22, key "..."` and the pair is not written. The two filename-extension constants `PAGFEXT` and `DIRFEXT` are exported for code that needs to locate the backing files.

Security

Do not open SDBM files from untrusted sources. The on-disk layout was designed for speed, not integrity, and a maliciously crafted file can drive the reader into undefined behavior.

Functions

Tie hooks

tiehash

Open an SDBM database and return a blessed handle. Invoked by Perl when the user writes `tie %h, 'SDBM_File', $basename, $flags, $mode`.

destroy

Close the SDBM database and release the handle. Invoked when the tied hash goes out of scope, is untie'd, or is garbage-collected.

fetch

Retrieve the value stored under a given key. Invoked by Perl's tie machinery when the user writes `$h{key}`.

store

Write a key-value pair to the database. Invoked by Perl's tie machinery when the user writes `$h{key} = $value`.

delete

Remove a key from the database. Invoked by Perl's tie machinery when the user writes `delete $h{key}`.

exists

Report whether a key is present in the database. Invoked by Perl's tie machinery when the user writes `exists $h{key}`.

firstkey

Return the first key for an iteration over the tied hash. Invoked by Perl at the start of `keys %h`, `values %h`, or `each %h`, and whenever iteration is reset.

nextkey

Return the next key in an ongoing iteration. Invoked by Perl after FIRSTKEY for every subsequent step of `keys`, `values`, or `each`.

error

Report whether the database has a sticky I/O-error flag set. Called as a method on the tied object: `(tie %h)->error`.

clearerr

Clear the sticky I/O-error flag on the database. Called as a method on the tied object: (tied %h)->sdbm_clearerr.

Filters

filter_fetch_key

Install a callback that rewrites keys on the way out of the database. Called as a method on the tied object: (tied %h)->filter_fetch_key(sub { ... }).

filter_store_key

Install a callback that rewrites keys on the way into the database. Called as a method on the tied object: (tied %h)->filter_store_key(sub { ... }).

filter_fetch_value

Install a callback that rewrites values on the way out of the database. Called as a method on the tied object: (tied %h)->filter_fetch_value(sub { ... }).

filter_store_value

Install a callback that rewrites values on the way into the database. Called as a method on the tied object: (tied %h)->filter_store_value(sub { ... }).

tiehash

Open an SDBM database and return a blessed handle. Invoked by Perl when the user writes tie %h, 'SDBM_File', \$basename, \$flags, \$mode.

The five-argument form opens \$basename.dir and \$basename.pag using sdbm_open. The six-argument form (tie %h, 'SDBM_File', \$dirfile, \$flags, \$mode, \$pagfile) names the two files independently and uses sdbm_prep - useful with File::Temp handles.

\$flags takes the usual Fcntl constants: O_RDONLY, O_WRONLY, O_RDWR, optionally OR'd with O_CREAT to create a missing database. \$mode is the permission bits applied (after umask) to newly created files; 0666 is the usual choice.

Returns a blessed SDBM_File reference on success, or undef if the underlying open fails (with \$! set to the reason).

store

Write a key-value pair to the database. Invoked by Perl's tie machinery when the user writes \$h{key} = \$value.

The combined length of key and value must not exceed 1008 bytes (the PAIRMAX constant). Oversized pairs are rejected: STORE croaks with sdbm store returned -1, errno 22, key "... " and nothing is written.

An attempt to write to a read-only database croaks with No write permission to sdbm file.

Undef values are stored as empty strings, matching perl5's silent-coerce behavior for tied hashes.

5.2.25 Scalar

Namespace containing the following modules:

- `Scalar::Util` — General-utility subroutines for inspecting and tagging scalar values.

Scalar::Util

General-utility subroutines for inspecting and tagging scalar values.

Functions

Other Functions

`reftype`

`reftype($ref)` - returns type string without blessing.

`tainted`

`tainted($sv)` - mirrors `Scalar-Util-1.68/Util.xs tainted()`. Returns true iff the argument carries `PERL_MAGIC_taint` (set by perl5's taint-source machinery when running under `-T`). The earlier stub hardcoded false under the false assumption that pperl has no taint mode; we do - `${^TAINT}` is live and sources (`%ENV`, `@ARGV`, `$0`, file reads) propagate `SvTAINTED` normally.

`set_prototype`

`set_prototype(\&sub, $proto)` - `Scalar::Util` calling convention (`&$`). perl5 `ListUtil.xs set_prototype()` (`Sub::Util` section): `cv = SvRV(code); SvTYPE(cv) == SVt_PVCV` or croak. `SvPOK(proto) → sv_copypv(cv, proto)` else `SvPOK_off(cv)` `Scalar::Util` wrapper reverses args: `(&code, $proto) → set_prototype($proto, $code)`. We implement the `Scalar::Util` convention directly: `arg0=coderef, arg1=proto`.

5.2.26 Socket

BSD socket constants and address pack/unpack helpers that feed Perl's built-in `socket`, `bind`, `connect`, `accept`, `send`, `recv`, `setsockopt`, and `shutdown`.

Typical flow: import a constant, pack it into a `sockaddr`, hand the result to the built-in.

```
use Socket;  
socket(my $sock, PF_INET, SOCK_STREAM, IPPROTO_TCP) or die $!;
```

(continues on next page)

(continued from previous page)

```
my $addr = pack_sockaddr_in(80, inet_aton("127.0.0.1"));
connect($sock, $addr) or die $!;
```

What lives here

- **Address families:** AF_INET, AF_INET6, AF_UNIX, AF_LOCAL, AF_UNSPEC, plus the matching PF_* protocol families.
- **Socket types:** SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, SOCK_SEQPACKET.
- **Well-known addresses:** INADDR_ANY, INADDR_LOOPBACK, INADDR_BROADCAST, INADDR_NONE, IN6ADDR_ANY, IN6ADDR_LOOPBACK (packed binary, ready for pack_sockaddr_in / pack_sockaddr_in6).
- **Protocol numbers:** IPPROTO_IP, IPPROTO_TCP, IPPROTO_UDP, IPPROTO_IPV6, IPPROTO_ICMP, IPPROTO_RAW.
- **Socket-option levels and names:** SOL_SOCKET, SO_REUSEADDR, SO_KEEPALIVE, SO_BROADCAST, SO_LINGER, SO_SNDBUF, SO_RCVBUF, SO_ERROR, SO_TYPE, and the rest of the SO_* family.
- **TCP options:** TCP_NODELAY.
- **IP options:** IP_TOS, IP_TTL, IP_MULTICAST_TTL, IP_MULTICAST_LOOP, IP_ADD_MEMBERSHIP, IP_DROP_MEMBERSHIP, plus the IPV6_* counterparts.
- **Message flags:** MSG_OOB, MSG_PEEK, MSG_DONTWAIT, MSG_WAITALL, MSG_NOSIGNAL.
- **Shutdown modes:** SHUT_RD, SHUT_WR, SHUT_RDWR.
- **Address packing:** pack_sockaddr_in, unpack_sockaddr_in, pack_sockaddr_in6, unpack_sockaddr_in6, pack_sockaddr_un, unpack_sockaddr_un, plus the dual-mode sockaddr_in / sockaddr_in6 shortcuts and the sockaddr_family extractor.
- **Address parsing:** inet_aton, inet_ntoa, inet_pton, inet_ntop.
- **Name resolution:** getaddrinfo, getnameinfo, plus the AI_*, NI_*, and EAI_* flag and error constants.

Functions

Address packing

pack_sockaddr_in

Build a packed sockaddr_in from a port number and a 4-byte IPv4 address, ready to hand to bind or connect.

unpack_sockaddr_in

Split a packed sockaddr_in back into its port and 4-byte address.

pack_sockaddr_in6

Build a packed `sockaddr_in6` from a port, a 16-byte IPv6 address, and optional scope and flow-info fields.

unpack_sockaddr_in6

Split a packed `sockaddr_in6` into port, 16-byte address, scope id, and flow-info.

pack_sockaddr_un

Build a packed `sockaddr_un` from a filesystem path, for use with Unix-domain sockets.

unpack_sockaddr_un

Extract the filesystem path from a packed `sockaddr_un`.

sockaddr_family

Read the address-family byte from any packed `sockaddr` without committing to one unpacking shape.

sockaddr_in

Dual-mode shortcut: with two arguments it behaves like `pack_sockaddr_in`; with one argument it behaves like `unpack_sockaddr_in`.

sockaddr_in6

Dual-mode IPv6 shortcut: with two or more arguments it behaves like `pack_sockaddr_in6`; with a single argument it behaves like `unpack_sockaddr_in6`.

Address parsing

inet_aton

Turn a dotted-quad string or a hostname into a 4-byte packed IPv4 address, or `undef` if the name cannot be resolved.

inet_ntoa

Turn a 4-byte packed IPv4 address back into a dotted-quad string.

inet_pton

Parse an address string into its packed binary form for either IPv4 or IPv6, selected by the first argument.

inet_ntop

Turn a packed IPv4 or IPv6 address back into its printable string form, selected by the first argument.

Name resolution

getaddrinfo

Resolve a host name and/or service name into a list of ready-to-use address records, honouring IPv4/IPv6 and protocol preferences carried through a hints hash.

getnameinfo

Turn a packed sockaddr into a host name and service name, with optional flags to force numeric output or demand a successful DNS lookup.

inet_aton

Turn a dotted-quad string or a hostname into a 4-byte packed IPv4 address, or undef if the name cannot be resolved.

Synopsis

```
my $packed = inet_aton("127.0.0.1");  
my $packed = inet_aton("localhost");
```

What you get back

A 4-byte string in network byte order suitable as the second argument to `pack_sockaddr_in`, or undef on failure. To see the bytes as a human-readable address, pass the result through `inet_ntoa`.

Examples

```
my $loop = inet_aton("127.0.0.1");      # 4-byte string  
print join(".", unpack("C4", $loop));  # 127.0.0.1
```

```
my $sin = pack_sockaddr_in(80, inet_aton("example.com"));  
## pass $sin to connect()
```

Edge cases

- A literal IPv4 address is parsed directly - no DNS lookup.
- A non-resolvable name returns undef.
- IPv6 addresses are rejected here; use `inet_pton(AF_INET6, ...)`.

Differences from upstream

Fully compatible with upstream Socket.

See also

- `inet_ntoa` - reverse direction, packed bytes to dotted-quad string.
- `inet_pton` - same idea but parameterised by address family.
- `pack_sockaddr_in` - the immediate consumer of the packed address.
- `getaddrinfo` - richer resolver when you also need a port or IPv6.

`inet_ntoa`

Turn a 4-byte packed IPv4 address back into a dotted-quad string.

Synopsis

```
my $str = inet_ntoa($packed_ipv4);
```

What you get back

A string like "192.0.2.1", or undef if the input is shorter than 4 bytes.

Examples

```
print inet_ntoa(INADDR_LOOPBACK);    # 127.0.0.1
```

```
my ($port, $ip) = unpack_sockaddr_in($peer);
printf "%s:%d\n", inet_ntoa($ip), $port;
```

Edge cases

- Input shorter than 4 bytes returns undef.
- Input longer than 4 bytes: the extra bytes are ignored.

Differences from upstream

Fully compatible with upstream Socket.

See also

- `inet_aton` - reverse direction, string to packed bytes.
- `inet_ntop` - same idea for IPv4 or IPv6, parameterised by family.
- `unpack_sockaddr_in` - usually the producer of the packed address.

inet_pton

Parse an address string into its packed binary form for either IPv4 or IPv6, selected by the first argument.

Synopsis

```
my $v4 = inet_pton(AF_INET, "192.0.2.1");
my $v6 = inet_pton(AF_INET6, "2001:db8::1");
```

What you get back

A 4-byte string for AF_INET or a 16-byte string for AF_INET6, in network byte order. Returns undef when the string is not a valid address for the requested family.

Examples

```
my $addr = inet_pton(AF_INET6, "::1");
my $sin6 = pack_sockaddr_in6(443, $addr);
```

```
defined(inet_pton(AF_INET, $input))
  or die "not a valid IPv4 address: $input\n";
```

Edge cases

- Unsupported family returns undef rather than croaking.
- Unlike inet_aton, there is no DNS fallback - hostnames are rejected.
- Scoped IPv6 literals like "fe80::1%eth0" are not accepted; use getaddrinfo to carry the zone index through.

Differences from upstream

Fully compatible with upstream Socket.

See also

- inet_ntop - reverse direction, packed bytes to string.
- inet_aton - IPv4-only convenience with DNS fallback.
- pack_sockaddr_in6 - the usual consumer of an IPv6 packed address.

inet_ntop

Turn a packed IPv4 or IPv6 address back into its printable string form, selected by the first argument.

Synopsis

```
my $s = inet_ntop(AF_INET, $packed4);  
my $s = inet_ntop(AF_INET6, $packed6);
```

What you get back

A dotted-quad string for AF_INET, a canonical colon-hex string (with :: compression) for AF_INET6, or undef if the input is too short or the family is unsupported.

Examples

```
print inet_ntop(AF_INET6, IN6ADDR_LOOPBACK); # ::1
```

```
my ($port, $addr, $scope, $flow) = unpack_sockaddr_in6($peer);  
printf "[%s]:%d\n", inet_ntop(AF_INET6, $addr), $port;
```

Edge cases

- Input shorter than the family's address length returns undef.
- Any family other than AF_INET or AF_INET6 returns undef.

Differences from upstream

Fully compatible with upstream Socket.

See also

- inet_pton - reverse direction, string to packed bytes.
- inet_ntoa - IPv4-only shortcut with no family argument.

pack_sockaddr_in

Build a packed sockaddr_in from a port number and a 4-byte IPv4 address, ready to hand to bind or connect.

Synopsis

```
my $sin = pack_sockaddr_in($port, $packed_ipv4);
```

What you get back

A fixed-size byte string whose layout matches the platform's struct sockaddr_in: address family (AF_INET), port in network byte order, the raw 4 address bytes, and trailing padding. The result is opaque from Perl's perspective - treat it as a cookie for the socket built-ins.

Examples

```
my $addr = pack_sockaddr_in(80, inet_aton("example.com"));
connect($sock, $addr) or die $!;
```

```
## Bind to every local interface on a chosen port.

bind($sock, pack_sockaddr_in(8080, INADDR_ANY)) or die $!;
```

Edge cases

- Port is truncated to 16 bits; values above 65535 wrap silently.
- An address string shorter than 4 bytes leaves the address field zeroed (i.e. 0.0.0.0).
- The address argument is a **packed** 4-byte string, not a dotted-quad literal. Pass `inet_aton(...)` or `INADDR_*`.

Differences from upstream

Fully compatible with upstream Socket.

See also

- `unpack_sockaddr_in` - reverse direction.
- `sockaddr_in` - dual-mode shortcut that dispatches on arg count.
- `pack_sockaddr_in6` - IPv6 counterpart.
- `inet_aton` - usual source of the address argument.

unpack_sockaddr_in

Split a packed `sockaddr_in` back into its port and 4-byte address.

Synopsis

```
my ($port, $packed_ipv4) = unpack_sockaddr_in($sockaddr);
```

What you get back

A two-element list: the port as an integer, the address as a 4-byte packed string. Feed the second through `inet_ntoa` to see it as dotted-quad. When the input is shorter than a `sockaddr_in`, both elements come back as `undef`.

Examples

```
my $peer = getpeername($sock);
my ($port, $ip) = unpack_sockaddr_in($peer);
printf "connected from %s:%d\n", inet_ntoa($ip), $port;
```

Edge cases

- Input shorter than `sockaddr_in` yields `(undef, undef)` rather than croaking - check the first return value before using it.
- The address family byte in the input is **not** validated; the caller is expected to have an IPv4 `sockaddr` in hand.

Differences from upstream

Fully compatible with upstream `Socket`.

See also

- `pack_sockaddr_in` - reverse direction.
- `sockaddr_family` - read just the family byte without unpacking.
- `unpack_sockaddr_in6` - IPv6 counterpart.
- `inet_ntoa` - turn the returned packed address into a string.

pack_sockaddr_in6

Build a packed `sockaddr_in6` from a port, a 16-byte IPv6 address, and optional scope and flow-info fields.

Synopsis

```
my $sin6 = pack_sockaddr_in6($port, $packed_ipv6);
my $sin6 = pack_sockaddr_in6($port, $packed_ipv6, $scope_id,
    ↪$flowinfo);
```

What you get back

A fixed-size byte string matching the platform's `struct sockaddr_in6`, ready for `bind` or `connect`.

Examples

```
my $addr = inet_pton(AF_INET6, "::1");
my $sin6 = pack_sockaddr_in6(443, $addr);
connect($sock, $sin6) or die $!;
```

```
## Link-local address with scope index for eth0.

my $scope = scalar getsockopt(...); # or from if_nametoindex
my $sin6 = pack_sockaddr_in6(5353, inet_pton(AF_INET6, "fe80::1"),
    ↪$scope);
```

Edge cases

- \$scope_id and \$flowinfo default to 0 when omitted.
- An address argument shorter than 16 bytes leaves the address field zeroed (i.e. ::).
- Port and flow-info are truncated to 16 and 32 bits respectively.

Differences from upstream

Fully compatible with upstream Socket.

See also

- unpack_sockaddr_in6 - reverse direction.
- sockaddr_in6 - dual-mode shortcut that dispatches on arg count.
- pack_sockaddr_in - IPv4 counterpart.
- inet_pton - usual source of the address argument.

unpack_sockaddr_in6

Split a packed sockaddr_in6 into port, 16-byte address, scope id, and flow-info.

Synopsis

```
my ($port, $addr, $scope_id, $flowinfo) = unpack_sockaddr_in6(
    ↪$sockaddr);
```

What you get back

A four-element list. When the input is shorter than a sockaddr_in6, all four elements come back as undef.

Examples

```
my ($port, $addr, $scope, $flow) = unpack_sockaddr_in6(getpeername(
    ↪$sock));
printf "[%s%%%d]:%d\n", inet_ntop(AF_INET6, $addr), $scope, $port;
```

Edge cases

- Scope id and flow-info are often 0 for non-link-local traffic.
- Input shorter than `sockaddr_in6` yields four undefs.
- The family byte is not validated - pass an IPv6 `sockaddr`.

Differences from upstream

Fully compatible with upstream `Socket`.

See also

- `pack_sockaddr_in6` - reverse direction.
- `unpack_sockaddr_in` - IPv4 counterpart.
- `inet_ntop` - turn the returned packed address into a string.

`pack_sockaddr_un`

Build a packed `sockaddr_un` from a filesystem path, for use with Unix-domain sockets.

Synopsis

```
my $sun = pack_sockaddr_un($path);
```

What you get back

A fixed-size byte string matching the platform's `struct sockaddr_un`, ready for bind or connect against an `AF_UNIX` / `AF_LOCAL` socket.

Examples

```
socket(my $sock, AF_UNIX, SOCK_STREAM, 0) or die $!;  
connect($sock, pack_sockaddr_un("/tmp/app.sock")) or die $!;
```

```
## Abstract Linux namespace: first byte is NUL.
```

```
my $sun = pack_sockaddr_un("\0app.mysvc");
```

Edge cases

- Paths longer than 107 bytes are truncated to fit `sun_path`.
- No explicit trailing NUL is appended by the caller; the struct is zero-initialised and the path is copied in.

Differences from upstream

Fully compatible with upstream Socket.

See also

- `unpack_sockaddr_un` - reverse direction.
- `sockaddr_family` - read the family byte from an unknown sockaddr.

`unpack_sockaddr_un`

Extract the filesystem path from a packed `sockaddr_un`.

Synopsis

```
my ($path) = unpack_sockaddr_un($sockaddr);
```

What you get back

The `sun_path` field as a string, trimmed at the first NUL byte, or `undef` if the input is too short to be a `sockaddr_un`.

Examples

```
my ($path) = unpack_sockaddr_un(getsockname($sock));  
print "listening on $path\n";
```

Edge cases

- Abstract Linux-namespace sockets have a leading NUL; the result is truncated at that first NUL, so callers that need the full abstract name must unpack the raw bytes themselves.
- Input shorter than 3 bytes yields `undef`.

Differences from upstream

Fully compatible with upstream Socket.

See also

- `pack_sockaddr_un` - reverse direction.

`sockaddr_family`

Read the address-family byte from any packed sockaddr without committing to one unpacking shape.

Synopsis

```
my $family = sockaddr_family($sockaddr);
```

What you get back

An integer equal to one of AF_INET, AF_INET6, AF_UNIX, etc., or undef if the input is shorter than 2 bytes.

Examples

```
my $sa = getpeername($sock);
if (sockaddr_family($sa) == AF_INET6) {
    my ($port, $addr) = unpack_sockaddr_in6($sa);
    ...
} else {
    my ($port, $addr) = unpack_sockaddr_in($sa);
    ...
}
```

Edge cases

- The family byte is read in native byte order - this matches how every pack_sockaddr_* writes it.
- Input shorter than 2 bytes returns undef.

Differences from upstream

Fully compatible with upstream Socket.

See also

- unpack_sockaddr_in, unpack_sockaddr_in6, unpack_sockaddr_un
 - the right unpacker depends on the family returned here.

sockaddr_in

Dual-mode shortcut: with two arguments it behaves like pack_sockaddr_in; with one argument it behaves like unpack_sockaddr_in.

Synopsis

```
my $sin          = sockaddr_in($port, $packed_ipv4);
my ($port, $addr) = sockaddr_in($sockaddr);
```


What you get back

The same values as the underlying `pack_sockaddr_in` or `unpack_sockaddr_in`, depending on argument count.

Examples

```
## Packing
```

```
connect($sock, sockaddr_in(80, inet_aton("127.0.0.1"))) or die $!;
```

```
## Unpacking
```

```
my ($port, $ip) = sockaddr_in(getpeername($sock));
```

Edge cases

- The dispatch is on argument count, not argument type. Calling with a single two-element list-in-scalar oddity will still take the unpack branch.
- Prefer the explicit `pack_sockaddr_in` / `unpack_sockaddr_in` in new code; this name exists for compatibility with older Perl idioms.

Differences from upstream

Fully compatible with upstream `Socket`.

See also

- `pack_sockaddr_in`, `unpack_sockaddr_in` - the explicit forms.
- `sockaddr_in6` - IPv6 counterpart with the same dual behaviour.

sockaddr_in6

Dual-mode IPv6 shortcut: with two or more arguments it behaves like `pack_sockaddr_in6`; with a single argument it behaves like `unpack_sockaddr_in6`.

Synopsis

```
my $sin6 = sockaddr_in6($port, $addr);
my $sin6 = sockaddr_in6($port, $addr,
    ↪ $scope, $flow);
my ($port, $addr, $scope, $flow) = sockaddr_in6($sockaddr);
```

What you get back

The same values as the underlying `pack_sockaddr_in6` or `unpack_sockaddr_in6`, depending on argument count.

Examples

```
my $sin6 = sockaddr_in6(443, inet_pton(AF_INET6, "::1"));
```

```
my ($port, $addr) = sockaddr_in6(getpeername($sock));
```

Edge cases

- The dispatch is on argument count - a single-element list still takes the unpack branch.
- Prefer the explicit pack/unpack forms in new code.

Differences from upstream

Fully compatible with upstream `Socket`.

See also

- `pack_sockaddr_in6`, `unpack_sockaddr_in6` - the explicit forms.
- `sockaddr_in` - IPv4 counterpart.

getaddrinfo

Resolve a host name and/or service name into a list of ready-to-use address records, honouring IPv4/IPv6 and protocol preferences carried through a hints hash.

Synopsis

```
my ($err, @res) = getaddrinfo($host, $service);  
my ($err, @res) = getaddrinfo($host, $service, { family => AF_INET6,  
→ });
```

What you get back

A list whose first element is an error string - empty on success, otherwise a human-readable message from `gai_strerror`. Every subsequent element is a hash reference describing one resolved endpoint, with keys:

- `family` - integer address family (`AF_INET`, `AF_INET6`, ...).
- `socktype` - integer socket type (`SOCK_STREAM`, `SOCK_DGRAM`, ...).
- `protocol` - integer protocol number.

- `addr` - packed sockaddr ready for bind or connect.
- `canonname` - canonical hostname (only populated when the hints hash requests it with `flags => AI_CANONNAME`).

The hints hash accepts `family`, `socktype`, `protocol`, and `flags`, each an integer; unknown keys are ignored.

Examples

```
my ($err, @res) = getaddrinfo("example.com", "https",
    { socktype => SOCK_STREAM });
die "resolve: $err\n" if $err;
for my $r (@res) {
    socket(my $s, $r->{family}, $r->{socktype}, $r->{protocol}) or
    ↪next;
    connect($s, $r->{addr}) and last;
    close $s;
}
```

```
## Numeric-only, no DNS lookup, no service lookup.
```

```
my ($err, @res) = getaddrinfo("2001:db8::1", "443",
    { flags => AI_NUMERICHOST | AI_NUMERICSERV });
```

```
## Listening side: ask for a wildcard bind address.
```

```
my ($err, @res) = getaddrinfo(undef, "8080",
    { flags => AI_PASSIVE, family => AF_INET6 });
```

Edge cases

- On error the result list is exactly one element - the message
 - so scalar `@res` is 1 and the first hashref is absent.
- Pass `undef` for `$host` to resolve a service with no specific host (typically combined with `AI_PASSIVE`).
- Pass `undef` for `$service` to resolve a host with no port.

Differences from upstream

Fully compatible with upstream `Socket`.

See also

- `getnameinfo` - reverse direction, sockaddr to host and service.
- `inet_pton` - parse a single address string without a full resolver.
- `pack_sockaddr_in`, `pack_sockaddr_in6` - when you already have the address bytes and just need the struct.

getnameinfo

Turn a packed sockaddr into a host name and service name, with optional flags to force numeric output or demand a successful DNS lookup.

Synopsis

```
my ($err, $host, $service) = getnameinfo($sockaddr);
my ($err, $host, $service) = getnameinfo($sockaddr, $flags);
```

What you get back

A three-element list: error string (empty on success), host, and service. On error, host and service come back as undef.

Useful flag bits:

- NI_NUMERICHOST - skip DNS, return a printable address.
- NI_NUMERICSERV - return the port number as a decimal string.
- NI_NAMEREQD - fail instead of falling back to numeric host.
- NI_NOFQDN - local hosts keep their short name.
- NI_DGRAM - look up the service as UDP rather than TCP.

Examples

```
my ($err, $host, $svc) = getnameinfo(getpeername($sock));
die "reverse lookup: $err\n" if $err;
print "peer: $host:$svc\n";
```

```
## Numeric formatting only - no DNS, no /etc/services.

my ($err, $host, $svc) = getnameinfo($sockaddr,
    NI_NUMERICHOST | NI_NUMERICSERV);
```

Edge cases

- Input shorter than 2 bytes returns the error string "Invalid sockaddr" and two undefs.
- Internally the packed sockaddr is capped at 128 bytes; this is enough for every standard family.

Differences from upstream

Fully compatible with upstream Socket.

See also

- `getaddrinfo` - reverse direction, name to `sockaddr`.
- `inet_ntop` - format a raw address without a resolver.
- `unpack_sockaddr_in`, `unpack_sockaddr_in6` - when you need the numeric port and address without any lookup.

5.2.27 Storable

Serialise arbitrary Perl data structures to bytes, reconstruct them later, or deep-clone them in memory.

`Storable` turns a reference - to a scalar, array, hash, blessed object, or any tree built from them - into an opaque byte string that can be written to disk, sent over a socket, or fed back into the module to reproduce the original structure. The round trip preserves shared references, circular references, weak references, tied variables, regexp objects, and blessed classes (with `STORABLE_freeze` / `STORABLE_thaw` hooks honoured).

Two axes of choice cover every use:

- **Where the bytes go.** `freeze` returns them as a string, `store` writes them to a named file, `store_fd` writes them to an open filehandle. `thaw`, `retrieve`, and `fd_retrieve` are the matching readers.
- **Byte order on the wire.** `freeze` / `store` / `store_fd` use the host's native byte order - fastest, but only portable between machines of the same endianness and integer size. The `n`-prefixed variants `nfreeze`, `nstore`, `nstore_fd` use network (big-endian) order and are portable across architectures. The reader side auto-detects which was used.

`dclone` is the in-memory shortcut for "I want a structurally independent copy of this data." It serialises and immediately deserialises, so every element is freshly allocated and every `STORABLE_freeze` / `STORABLE_thaw` hook runs - which is the only safe way to clone objects that wrap non-SV resources (a PDL's engine pointer, a tied variable's backing state, a file handle).

`file_magic` and `read_magic` inspect a frozen payload without thawing it, returning a hash describing byte order, header size, and version metadata. Use them to validate untrusted input or to pick a reader before committing to a thaw.

Functions

Freeze and thaw

`freeze`

Serialise a referenced Perl structure into an opaque byte string using the host's native byte order.

ndefreeze

Serialise a referenced Perl structure to bytes in network (big-endian) byte order for portable transport.

thaw

Reconstruct a Perl structure from a byte string produced by freeze or ndefreeze.

File persistence

store

Serialise a referenced structure and write it to a named file, overwriting any previous contents.

nstore

Serialise a referenced structure in network byte order and write it to a named file.

retrieve

Read a frozen structure from a named file and reconstruct it.

File-descriptor persistence

store_fd

Serialise a referenced structure in native byte order and write it to an open filehandle.

nstore_fd

Serialise a referenced structure in network byte order and write it to an open filehandle.

fd_retrieve

Read one frozen structure from an open filehandle and reconstruct it.

Deep cloning

dclone

Produce a structurally independent deep copy of a Perl structure by round-tripping it through the serialiser.

Magic inspection

file_magic

Inspect a file's Storable header without thawing, returning metadata as a hash reference.

read_magic

Inspect an in-memory frozen buffer's header, returning metadata as a hash reference.

Tuning

stack_depth

Return the current scalar recursion limit used while freezing and thawing.

stack_depth_hash

Return the current hash-specific recursion limit used while freezing and thawing.

freeze

Serialise a referenced Perl structure into an opaque byte string using the host's native byte order.

Synopsis

```
use Storable qw(freeze thaw);
my $bytes = freeze(\%config);
my $copy  = thaw($bytes);
```

What you get back

A single scalar holding the binary image of the structure. The bytes are opaque - treat them as a blob, not text. They can be stored, transmitted, or passed straight back to thaw. Native byte order keeps the serialisation fast but the result is only portable between machines with the same endianness and integer width; for cross-architecture transport use `nfreeze`.

Examples

Freeze and immediately thaw a hash reference:

```
my $bytes = freeze({ host => "db1", port => 5432 });
my $cfg   = thaw($bytes);
print $cfg->{host};           # db1
```

Freeze a nested structure with shared references:

```
my $shared = [1, 2, 3];
my $bytes  = freeze({ a => $shared, b => $shared });
my $h      = thaw($bytes);
$h->{a}[0] = 99;
print $h->{b}[0];           # 99 - sharing preserved
```

Round-trip a blessed object:

```
my $obj    = bless { id => 42 }, "Widget";
my $bytes  = freeze(\$obj);
my $back   = ${ thaw($bytes) }; # same class, same contents
```

Edge cases

- Argument must be a reference. Passing a plain scalar croaks with `Not a reference`.
- Circular references are preserved; the reader rebuilds the same cycle.
- Weak references survive the round trip as weak references.

Differences from upstream

- The on-disk layout differs from upstream's format. Byte strings produced by this freeze can only be thawed by pperl's `Storable`, not by perl5's. The reverse also holds. Pinned by `t/81-xs-native/Storable/010-freeze-thaw.t`.

See also

- `thaw` - the matching reader.
- `ndefreeze` - portable (network-order) variant.
- `store` - write directly to a named file.
- `dclone` - freeze/thaw round-trip as a deep-copy shortcut.

ndefreeze

Serialise a referenced Perl structure to bytes in network (big-endian) byte order for portable transport.

Synopsis

```
use Storable qw(ndefreeze thaw);
my $bytes = ndefreeze(\@rows);
my $back  = thaw($bytes);      # thaw auto-detects byte order
```

What you get back

An opaque byte string prefixed with a marker that identifies the payload as network-order. The reader (`thaw`, `retrieve`, `fd_retrieve`) inspects that marker and byte-swaps as needed, so the same bytes reproduce the structure on any architecture.

Examples

Send serialised data over a socket between hosts of different endianness:

```
my $bytes = nfreeze(\%payload);
$socket->send($bytes);

## on the other side, possibly on a different CPU:

my $payload = thaw($received);
```

Store and retrieve using the portable pair:

```
open my $fh, ">", "data.bin" or die $!;
print $fh nfreeze(\@records);
close $fh;
```

Edge cases

- Argument must be a reference; a plain scalar croaks with `Not a reference`.
- Marginally slower than `freeze` on little-endian hosts because every multi-byte field is byte-swapped on the way out.
- The produced bytes differ from those of `freeze` even when run on a big-endian host, because `nfreeze` writes a leading marker byte.

Differences from upstream

- Byte layout is not wire-compatible with upstream `Storable`'s network format. `nfreeze` output round-trips only through `ppperl`'s `thaw`. Pinned by `t/81-xs-native/Storable/270-nfreeze-byteorder.t`.

See also

- `freeze` - faster, same-architecture variant.
- `thaw` - matching reader; auto-detects byte order.
- `nstore` - write portable bytes directly to a file.
- `nstore_fd` - write portable bytes to a file descriptor.

thaw

Reconstruct a Perl structure from a byte string produced by `freeze` or `nfreeze`.

Synopsis

```
use Storable qw(freeze thaw);
my $data = thaw($bytes);
```

What you get back

A reference to the rebuilt structure - the same kind of reference that was passed to freeze or nfreeze. Shared references, circular references, weak references, tied containers, regexps, and blessed objects are all restored. Blessed objects with a STORABLE_thaw hook have the hook run.

thaw auto-detects whether the input came from freeze (native byte order) or nfreeze (network byte order) - the caller does not need to know which producer was used.

Examples

Basic round trip:

```
my $bytes = freeze({ n => 1, list => [1, 2, 3] });
my $copy  = thaw($bytes);
print $copy->{list}[2];           # 3
```

Receive bytes produced on a different machine:

```
## $buf arrived over the wire from nfreeze on another host

my $msg = thaw($buf);
```

Edge cases

- Empty string or data whose first byte is not a valid tag croaks with a message about corrupt input.
- If the argument is not a defined string, croaks with argument is not a valid Storable frozen string.
- Truncation mid-record produces a corrupt-data error, not a partial structure.

Differences from upstream

- Only accepts payloads produced by pperl's freeze / nfreeze. Upstream Storable bytes are rejected as corrupt. Pinned by t/81-xs-native/Storable/010-freeze-thaw.t.

See also

- freeze - producer for native-order payloads.
- nfreeze - producer for portable payloads.
- retrieve - read a frozen structure back from a file path.
- fd_retrieve - read from an open file descriptor.

dclone

Produce a structurally independent deep copy of a Perl structure by round-tripping it through the serialiser.

Synopsis

```
use Storable qw(dclone);
my $copy = dclone($original);
```

What you get back

A reference of the same kind as the input, pointing to a tree whose every scalar, array, hash, and blessed object is a fresh allocation. Modifying \$copy never touches \$original. Shared references inside the tree remain shared *within the copy*, and circular references are rebuilt as circular references.

Because dclone runs a full freeze/thaw internally, any blessed object with a STORABLE_freeze / STORABLE_thaw hook has those hooks invoked. This is the only reliable way to clone objects that wrap non-SV state - a PDL piddle with an engine pointer, a tied variable, a file handle - because the hooks give each class a chance to duplicate its backing resource instead of aliasing it.

Examples

Independent copy of a nested hash:

```
my $cfg = { servers => [ { host => "a" }, { host => "b" } ] };
my $copy = dclone($cfg);
$copy->{servers}[0]{host} = "z";
print $cfg->{servers}[0]{host}; # a - original untouched
```

Clone a blessed object that owns a resource:

```
my $p = pdl(1, 2, 3);
my $q = dclone($p);
$q->slice("0") .= 99;
print $p->at(0); # 1 - original piddle unchanged
```

Edge cases

- Argument must be a reference. A plain scalar croaks with `Not a reference`.
- A class whose STORABLE_freeze hook dies will propagate that error out of dclone.
- Depth is bounded by `$Storable::recursion_limit` (default 512).

Differences from upstream

- Equivalent semantics; the internal wire format differs (see `freeze`). Pinned by `t/81-xs-native/Storable/020-dclone.t` and `t/81-xs-native/Storable/300-dclone-equiv.t`.

See also

- `freeze + thaw` - the long-form equivalent of `dclone`.
- `ndefreeze` - portable-byte-order producer if you need to ship the clone elsewhere later.
- `store` - persist the structure to disk instead of cloning.

store

Serialise a referenced structure and write it to a named file, overwriting any previous contents.

Synopsis

```
use Storable qw(store retrieve);
store(\%data, "cache.bin") or die "write failed";
my $data = retrieve("cache.bin");
```

What you get back

Returns 1 on success. The file at `$path` is created (or truncated if it already exists) and written with the same native-byte-order payload that `freeze` would produce.

Examples

Persist a cache and reload it on startup:

```
store({ built_at => time(), entries => \@entries }, "cache.bin");

## ... later, possibly a different process:

my $cache = retrieve("cache.bin");
```

Atomic-ish replace via rename:

```
store(\%state, "state.bin.tmp");
rename "state.bin.tmp", "state.bin";
```

Edge cases

- First argument must be a reference; croaks with `Not a reference` otherwise.
- If the file cannot be opened for writing, croaks with `Storable: can't open file for writing`.
- Files written by `store` are only portable between machines of the same endianness and integer size. Use `nstore` for cross-architecture transport.

Differences from upstream

- Payload format differs from upstream `Storable`'s; files are not interchangeable with perl5's `store` output. Pinned by `t/81-xs-native/Storable/030-store-retrieve.t`.

See also

- `retrieve` - matching reader for files written by `store`.
- `nstore` - portable variant (writes network byte order).
- `freeze` - return bytes instead of writing a file.
- `store_fd` - write to an already-open filehandle.

nstore

Serialise a referenced structure in network byte order and write it to a named file.

Synopsis

```
use Storable qw(nstore retrieve);
nstore(\%data, "portable.bin") or die "write failed";
my $data = retrieve("portable.bin");    # works on any host
```

What you get back

Returns 1 on success. The file holds the same portable byte string `freeze` would produce, prefixed with the network-order marker. `retrieve` auto-detects that marker, so the reader side does not need a different function.

Examples

Write a configuration cache that any client can read:

```
nstore(\%config, "/srv/shared/config.bin");
```

Edge cases

- First argument must be a reference; croaks with `Not a reference` otherwise.
- If the file cannot be opened for writing, croaks with `Storable: can't open file for writing`.
- Slightly slower than `store` on little-endian hosts because every multi-byte value is byte-swapped on the way out.

Differences from upstream

- Byte layout is not wire-compatible with upstream `Storable`'s network-order format. Files round-trip only through `ppperl`'s `retrieve`. Pinned by `t/81-xs-native/Storable/250-nstore.t` and `t/81-xs-native/Storable/270-nfreeze-byteorder.t`.

See also

- `retrieve` - matching reader; auto-detects byte order.
- `store` - faster same-architecture variant.
- `nfreeze` - produce bytes in memory without a file.
- `nstore_fd` - write portable bytes to an open filehandle.

retrieve

Read a frozen structure from a named file and reconstruct it.

Synopsis

```
use Storable qw(store retrieve);
my $data = retrieve("cache.bin");
```

What you get back

A reference of the same kind that was passed to `store` or `nstore`. Byte order is auto-detected, so one `retrieve` works for files written by either producer.

Examples

Reload a persisted cache on process start:

```
my $cache = -e "cache.bin" ? retrieve("cache.bin") : {};
```

Consume a file produced by `nstore` on another machine:

```
my $payload = retrieve("incoming.bin");
```

Edge cases

- If the file cannot be opened, croaks with `Storable: can't open file for reading`.
- Empty or corrupt files croak with an explicit “not valid Storable data” message.
- A partial read (network interruption, truncated file) is reported as corrupt, never returned as a half-built structure.

Differences from upstream

- Only accepts files written by pperl’s `store` / `nstore`. Files produced by perl5’s `Storable::store` are rejected as corrupt. Pinned by `t/81-xs-native/Storable/030-store-retrieve.t`.

See also

- `store` - matching writer for native-order payloads.
- `nstore` - matching writer for portable payloads.
- `fd_retrieve` - read from an open filehandle.
- `file_magic` - inspect the file’s header without thawing.

store_fd

Serialise a referenced structure in native byte order and write it to an open filehandle.

Synopsis

```
use Storable qw(store_fd fd_retrieve);
open my $fh, ">", "data.bin" or die $!;
store_fd(\%data, $fh);
close $fh;
```

What you get back

Returns 1 on success. Writes a 4-byte length prefix followed by the serialised payload, so multiple frozen structures can be concatenated on the same stream and separated by matching `fd_retrieve` calls.

Examples

Stream multiple records through one filehandle:

```
open my $fh, ">", "log.bin" or die $!;
store_fd($_, $fh) for @events;
close $fh;
```

(continues on next page)

(continued from previous page)

```
## reader side:

open my $r, "<", "log.bin" or die $!;
while (my $ev = fd_retrieve($r)) { ... }
```

Edge cases

- First argument must be a reference; croaks with `Not a reference` otherwise.
- `$fh` must be an open filehandle with a real underlying file descriptor. A closed or non-file handle writes nothing.
- Bytes are native-order; use `nstore_fd` for portability.

Differences from upstream

- Upstream writes a header then raw payload; pperl prefixes each payload with a 4-byte little-endian length, enabling concatenation on the same stream. Readers on the pperl side (`fd_retrieve`) understand this framing; upstream readers do not. Pinned by `t/81-xs-native/Storable/080-fd.t`.

See also

- `fd_retrieve` - matching reader.
- `nstore_fd` - portable (network-order) variant.
- `store` - write to a path instead of an open handle.
- `freeze` - return bytes to the caller instead of writing.

`nstore_fd`

Serialise a referenced structure in network byte order and write it to an open filehandle.

Synopsis

```
use Storable qw(nstore_fd fd_retrieve);
open my $fh, ">", "portable.bin" or die $!;
nstore_fd(\%data, $fh);
```

What you get back

Returns 1 on success. Emits a length-prefixed payload - same framing as `store_fd` - whose body is a portable (network-order) serialisation. The matching reader is `fd_retrieve`, which detects byte order automatically.

Examples

Ship a record to a remote host over a pipe:

```
open my $out, ">&", \*STDOUT or die $!;  
nstore_fd(\%payload, $out);
```

Edge cases

- First argument must be a reference; croaks with `Not a reference` otherwise.
- If the filehandle has no usable file descriptor, the call succeeds silently without writing. Check the handle first if that matters.

Differences from upstream

- Uses the same 4-byte little-endian length-prefix framing as `store_fd`; upstream `Storable` does not. Pinned by `t/81-xs-native/Storable/080-fd.t`.

See also

- `fd_retrieve` - matching reader.
- `store_fd` - faster same-architecture variant.
- `nstore` - write portable bytes to a named file.
- `ndefreeze` - return portable bytes instead of writing.

fd_retrieve

Read one frozen structure from an open filehandle and reconstruct it.

Synopsis

```
use Storable qw(store_fd fd_retrieve);  
open my $fh, "<", "data.bin" or die $!;  
my $data = fd_retrieve($fh);
```

What you get back

A reference of the same kind written by the matching `store_fd` / `nstore_fd`. Byte order is auto-detected from the payload marker. `fd_retrieve` consumes exactly one framed record, leaving the filehandle positioned at the next one - repeat the call to read subsequent records.

Examples

Drain a multi-record file:

```
open my $fh, "<", "log.bin" or die $!;
while (my $rec = fd_retrieve($fh)) {
    process($rec);
}
```

Read a single incoming message from a socket:

```
my $msg = fd_retrieve($socket);
```

Edge cases

- Returns undef if the filehandle has no readable file descriptor.
- Returns undef on EOF or on a framing error (short read of the length prefix, truncated payload, corrupt header).
- The length prefix is bounded at 256 MiB per record; larger records are rejected as implausible.

Differences from upstream

- Only reads the length-framed format written by pperl's store_fd / nstore_fd. Upstream Storable's fd format is not recognised. Pinned by t/81-xs-native/Storable/080-fd.t.

See also

- store_fd - matching writer (native order).
- nstore_fd - matching writer (portable).
- retrieve - read from a named path instead of a filehandle.
- thaw - feed bytes you already have in memory.

file_magic

Inspect a file's Storable header without thawing, returning metadata as a hash reference.

Synopsis

```
use Storable qw(file_magic);
my $info = file_magic("data.bin");
die "not Storable" unless $info;
```

What you get back

A hash reference describing the file's header, or undef if the file does not exist, cannot be opened, or does not begin with a valid Storable tag. Keys:

- file - the path that was inspected.

- `netorder` - 1 if the payload is network-order (`nstore` / `nstore_fd`), 0 otherwise.
- `major`, `minor` - format version numbers.
- `hdrsize` - number of bytes of framing before the payload proper.

Examples

Decide whether to trust an uploaded file:

```
my $info = file_magic($upload);
die "rejecting non-Storable upload" unless $info;
warn "portable payload" if $info->{netorder};
```

Quickly enumerate a directory of snapshots:

```
for my $f (glob "snapshots/*.bin") {
    my $m = file_magic($f) or next;
    printf "%s: v%d.%d\n", $f, $m->{major}, $m->{minor};
}
```

Edge cases

- Reads only the first 64 bytes; large files are not scanned in full, so this is cheap on any size.
- Returns `undef` rather than croaking on a missing or unreadable file.

Differences from upstream

- Upstream populates additional keys such as `byteorder`, `intsize`, `longsize`, `ptrsize`, `nvsize`, and `version_nv`. pperl's header does not record those fields, so they are omitted rather than filled with placeholder values. `major` / `minor` are fixed at 2 / 11 to match upstream's version baseline that most callers look for. Pinned by `t/81-xs-native/Storable/200-file-magic.t` and `t/81-xs-native/Storable/310-file-magic-accuracy.t`.

See also

- `read_magic` - same inspection for an in-memory buffer.
- `retrieve` - read and thaw the file after validating its magic.
- `nstore` - produces files with `netorder => 1`.

read_magic

Inspect an in-memory frozen buffer's header, returning metadata as a hash reference.

Synopsis

```
use Storable qw(read_magic);
my $info = read_magic($bytes);
die "not Storable" unless $info;
```

What you get back

A hash reference describing the payload's header, or undef if \$bytes does not begin with a valid Storable tag. Keys match those of file_magic, minus file (there is no path to report): netorder, major, minor, hdrsize.

Examples

Validate incoming bytes before committing to a thaw:

```
my $info = read_magic($received);
die "bad input" unless $info;
my $data = thaw($received);
```

Edge cases

- Returns undef for an empty string or any buffer whose first meaningful byte is not a known tag.
- Only the header is inspected; the rest of \$bytes is not scanned.

Differences from upstream

- See file_magic - the same key subset is returned. Pinned by t/81-xs-native/Storable/200-file-magic.t.

See also

- file_magic - same inspection for a path on disk.
- thaw - reconstruct the structure after validating magic.
- freeze - producer whose output this inspects.

stack_depth

Return the current scalar recursion limit used while freezing and thawing.

Synopsis

```
use Storable;
my $d = Storable::stack_depth();
```

What you get back

The integer value of `$Storable::recursion_limit`. This is the maximum nesting depth the serialiser will follow before aborting with a recursion error. The default is 512; assigning to `$Storable::recursion_limit` changes what the next call returns.

Examples

Raise the limit for a particularly deep structure:

```
local $Storable::recursion_limit = 4096;
my $bytes = freeze($deep_tree);
```

Edge cases

- Zero or negative values of `$Storable::recursion_limit` are treated as the default 512 when serialising.

Differences from upstream

Fully compatible with upstream. Pinned by `t/81-xs-native/Storable/240-stack-depth.t`.

See also

- `stack_depth_hash` - the separate limit used for hash serialisation.
- `freeze` / `thaw` - the consumers of this limit.

stack_depth_hash

Return the current hash-specific recursion limit used while freezing and thawing.

Synopsis

```
use Storable;
my $d = Storable::stack_depth_hash();
```

What you get back

The integer value of `$Storable::recursion_limit_hash`. This is a separate ceiling for hash nesting, defaulting to 256. Assigning to `$Storable::recursion_limit_hash` changes what the next call returns.

Edge cases

- Independent of the scalar limit returned by `stack_depth`. Raising one does not raise the other.

Differences from upstream

Fully compatible with upstream. Pinned by `t/81-xs-native/Storable/240-stack-depth.t`.

See also

- `stack_depth` - scalar recursion limit.
- `freeze / thaw` - the consumers of this limit.

5.2.28 Sys

Namespace containing the following modules:

- `Sys::Hostname` — Return the local machine's hostname, cached after the first lookup.

Sys::Hostname

Return the local machine's hostname, cached after the first lookup.

Calls `libc::gethostname` into a stack buffer, returns result as a PV SV. Mirrors `Sys::Hostname XS`.

Functions

Other Functions

ghname

`ghname()` - raw `gethostname` wrapper.

hostname

`hostname()` - cached `hostname` wrapper.

5.2.29 Term

Namespace containing the following modules:

- `Term::ReadLine`

Term::ReadLine

Namespace containing the following modules:

- `Term::ReadLine::Gnu` — Interactive line editing, command history, and configurable key bindings, backed by the GNU Readline library.

Term::ReadLine::Gnu

Interactive line editing, command history, and configurable key bindings, backed by the GNU Readline library.

Install a readline instance, hand it a prompt, and you get the usual bash-style editing: cursor motion, word-wise kill and yank, incremental history search, filename completion, and an inputrc-controlled key map. The module is a thick binding over `libreadline.so` - nearly one hundred entry points expose readline's C surface directly, organised here by the same section names the GNU Readline manual uses.

A typical script calls `new` to obtain the readline object, then `readline` in a loop with `add_history` after each accepted line. Scripts that customise behaviour reach for `rl_parse_and_bind` and `rl_variable_bind` for inputrc-style configuration, or install completion by binding `rl_completion_entry_function` via the `Attribs` tied hash.

Functions

Constructors

`do_perl_setup`

Force the module's deferred Perl-side setup (%`Attribs` tie, `AUTOLOAD`, constants) to run now.

`gnu_new`

Construct a `Term::ReadLine::Gnu` object, wiring `$NAME` into `rl_readline_name` for inputrc dispatch.

`gnu_readline_name`

Return the string `"Term::ReadLine::Gnu"` for the `Term::ReadLine` backend-probing protocol.

Basic readline

`rl_readline`

Display a prompt, read one line with full editing, and return the entered string (or `undef` on EOF).

`rl_initialize`

Initialise (or re-initialise) readline's internal state and re-read the inputrc. Returns `0` on success.

History

add_history

Append \$line to the end of the history list so it shows up in later Up-arrow recalls and searches.

clear_history

Delete every entry from the in-memory history list.

using_history

Reset the history offset pointer back to the end of the list so previous_history starts from the newest entry.

where_history

Return the current offset into the history list (the index a subsequent current_history would read).

history_set_pos

Move the history cursor to \$pos. Returns 1 on success, 0 if the position is out of range.

current_history

Return the line at the current history cursor, or undef if the cursor is past the end.

history_get

Return the history line at absolute \$offset, or undef if no such entry exists.

previous_history

Move the history cursor one step back and return that entry, or undef if already at the oldest line.

next_history

Move the history cursor one step forward and return that entry, or undef if already past the newest line.

remove_history

Delete the history entry at \$which and return its text.

replace_history_entry

Replace the history entry at `$which` with `$line` and return the previous text, or undef if out of range.

stifle_history

Cap the history list at `$max` entries, or lift the cap when `$max` is undef. Returns the effective cap.

unstifle_history

Remove any stifling cap on the history list and return the previous cap (or a negative value if none was set).

history_is_stifled

Return non-zero if the history list is currently stifled (length-capped), zero otherwise.

history_total_bytes

Return the total number of bytes occupied by all history lines in memory.

read_history_range

Load lines from a history file into memory. Reads the whole file by default, or lines `$from..$to` when given.

write_history

Write every entry in the current history list to `$filename`, overwriting the file. Returns 0 on success.

append_history

Append the last `$nelements` entries of the in-memory history to `$filename`, preserving existing contents.

history_truncate_file

Truncate `$filename` to at most `$nlines` of history, keeping the newest lines. Returns 0 on success.

history_expand

Apply csh-style history expansion (`!!`, `!$`, `^old^new^`) to `$line`. Returns (code, expanded).

Synopsis

```
my ($code, $expanded) = $term->history_expand($input);  
  
**$code: 0 no change, 1 expanded, -1 error, 2 print-only**
```

history_search

Search the history for an entry that contains \$string, moving the cursor to the match. Returns the match offset or -1.

history_search_prefix

Same as history_search, but matches only when the history entry *starts* with \$string.

history_search_pos

Search the history starting at absolute \$pos (default: current cursor). Returns match offset or -1.

history_tokenize

Split \$text into tokens the way Bash history expansion does (word-splitting, quote awareness). Returns a list.

Display

rl_redisplay

Repaint the current line from readline's in-memory buffer. Call this after you edit the buffer programmatically.

rl_forced_update_display

Force a full repaint even if readline thinks nothing changed. Useful after background output corrupts the screen.

rl_on_new_line

Tell readline that the cursor is now on a fresh line, so the next redisplay reprints the prompt.

rl_on_new_line_with_prompt

Like rl_on_new_line, but asserts the prompt is already physically on screen - skip the reprint.

rl_reset_line_state

Reset the display state so the next `rl_redisplay` assumes a clean line (no partial output pending).

rl_message

Overlay `$text` as a status message in place of the prompt until `rl_clear_message` is called.

rl_clear_message

Remove the status message installed by `rl_message` and restore the original prompt.

rl_crlf

Emit a newline (CRLF) on the output stream, moving readline's cursor tracking with it.

rl_ding

Ring the terminal bell (or flash, depending on the `bell -style` inputrc setting).

rl_show_char

Display character `$c` using readline's usual control-character rendering (e.g. `^A` for byte 1).

rl_save_prompt

Save the current prompt and line state so it can be restored after a temporary message or sub-prompt.

rl_restore_prompt

Restore the prompt and line state previously stashed by `rl_save_prompt`.

rl_set_prompt

Change the prompt to `$prompt` for the next redisplay. Accepts `\[...\]` escapes to mark invisible regions.

rl_expand_prompt

Parse a prompt string for invisible-character escapes and return its visible length.

Line editing

rl_insert_text

Insert `$text` into the line buffer at point, advancing the cursor past the inserted characters.

rl_delete_text

Delete the characters of the current line in the half-open range `[$start, $end)`. Returns the number removed.

rl_copy_text

Return a copy of the line buffer in the half-open range `[$start, $end)` without modifying the buffer.

rl_kill_text

Delete text in `[$start, $end)` and push it onto the kill ring so it becomes yankable with C-y.

rl_replace_line

Replace the entire line buffer with `$text`. Passing a truthy `$clear_undo` also drops the undo list.

rl_push_macro_input

Queue `$macro` as pending input so readline re-reads it character-by-character on the next key request.

Input

rl_read_key

Block until one key is available on `rl_instream` and return its byte value.

rl_getc

Read one byte from `$stream` (defaulting to `rl_instream`), bypassing readline's macro and timeout logic.

rl_stuff_char

Inject one byte onto readline's pending-input queue, as if the user had just pressed it.

rl_execute_next

Arrange for the next call to `rl_read_key` to return `$c` without doing any terminal I/O.

rl_clear_pending_input

Discard any input queued with `rl_stuff_char` or `rl_execute_next`.

rl_set_keyboard_input_timeout

Set the multi-byte escape-sequence timeout in microseconds. Returns the previous value.

Terminal

rl_prep_terminal

Put the terminal into raw mode for line editing. Pass a truthy `$meta_flag` to enable 8-bit input.

rl_deprep_terminal

Undo `rl_prep_terminal` and restore the terminal to cooked mode.

rl_reset_terminal

Re-query termcap for `$terminal_name` (default `$ENV{TERM}`), refreshing readline's idea of the terminal.

rl_set_screen_size

Override readline's idea of the window size with `$rows` and `$cols`.

rl_get_screen_size

Return the terminal size readline is currently assuming as (`$rows`, `$cols`).

rl_resize_terminal

Re-read the terminal size from the OS - typically called from a SIGWINCH handler.

rl_reset_screen_size

Force readline to re-query TIOCGWINSZ and reset `screenheight`/`screenwidth` from scratch.

Character classification

rl_alphabetic

Return non-zero if byte `$c` is a word-constituent character under readline's locale-aware rules.

Undo

rl_begin_undo_group

Open a new undo group so a sequence of edits collapses into one user-visible undo step.

rl_end_undo_group

Close the undo group opened by `rl_begin_undo_group`.

rl_do_undo

Perform one undo step. Returns `0` when no further undo is available.

rl_free_undo_list

Drop the entire undo history for the current line.

rl_modifying

Record the range [`$start`, `$end`) of the line buffer on the undo stack before you modify it by hand.

Parse and configuration

rl_parse_and_bind

Apply a single line of inputrc syntax ("`\\C-x\\C-r`": re-read-init-file, set editing-mode vi, ...).

rl_read_init_file

Load and apply an inputrc file. Defaults to `$ENV{INPUTRC}` or `~/ .inputrc`.

rl_variable_bind

Set inputrc variable `$name` to the string `$value` (e.g. completion-ignore-case, On).

rl_variable_value

Return the current value of inputrc variable `$variable` as a string, or `undef` if it is unknown.

rl_variable_dumper

Print every inputrc variable and its current value to `rl_outstream`. Pass truthy `$readable` for inputrc syntax.

rl_function_dumper

Print every named readline command and the keys bound to it. Pass truthy `$readable` for inputrc-style output.

rl_macro_dumper

Print every key macro bound in the current keymap. Pass truthy `$readable` for inputrc-style output.

rl_get_termcap

Look up the termcap capability named `$cap` and return its string value, or `undef` if not defined.

Signal handling

rl_cleanup_after_signal

Restore the terminal to cooked mode and reset signal handlers - call this from your own signal handler.

rl_free_line_state

Free the partially-edited line buffer so a signal-handling exit leaves readline in a clean state.

rl_reset_after_signal

Re-prepare the terminal and reinstall readline's signal handlers after a `SIGCONT` or similar resumption.

rl_check_signals

Give readline a chance to run any deferred signal handlers.

rl_echo_signal_char

Echo the tty-configured character for signal `$sig` (e.g. `^C` for `SIGINT`).

rl_pending_signal

Return the number of a signal readline has observed but not yet dispatched, or `0` if none is pending.

rl_set_signals

Install readline's default signal handlers (for SIGINT, SIGTERM, SIGQUIT, SIGWINCH, SIGHUP, ...).

rl_clear_signals

Remove the signal handlers installed by `rl_set_signals`.

State save and restore

rl_save_state

Snapshot readline's internal state and return an opaque handle you can later pass to `rl_restore_state`.

rl_restore_state

Reinstall the state previously captured by `rl_save_state`.

Variable access (tied Attribs)

rl_fetch_str

Return readline's string global at table slot `$id` (used by the tied `%Attribs` hash).

rl_store_str

Write string `$value` into readline's global at table slot `$id` (used by the tied `%Attribs` hash).

rl_store_rl_line_buffer

Replace `rl_line_buffer` with `$value` and clamp `rl_point/rl_end` to the new length.

rl_fetch_int

Return readline's integer global at table slot `$id`, widening the native type to Perl IV.

rl_store_int

Write integer `$value` into readline's global at table slot `$id`, narrowing to the native type.

rl_store_iostream

Set `rl_instream($id == 0)` or `rl_outstream($id == 1)` to the given `FILE*`.

rl_fetch_keymap

Return the current executing-keymap (\$id == 0) or binding-keymap (\$id == 1) as an opaque pointer.

rl_store_function

Install a Perl code reference as the callback backing function-pointer slot \$id.

rl_fetch_function

Return the Perl code reference currently stored in function-pointer slot \$id, or undef.

rl_fetch_last_func

Return a pointer (as an integer) to the last readline command function that ran.

Other Functions

stub

Stand-in for every XS entry when libreadline.so cannot be loaded - croaks with an install hint.

gnu_new

Construct a Term::ReadLine::Gnu object, wiring \$NAME into rl_readline_name for inputrc dispatch.

Synopsis

```
my $term = Term::ReadLine->new('myprog');  
my $term = Term::ReadLine::Gnu->new('myprog', $IN, $OUT);
```

The returned object is a blessed reference to the tied %Attribs hash, so reads and writes of \$term->Attribs->{rl_point} go straight to readline's globals. Calling new also initialises libreadline - no separate rl_initialize is needed afterwards.

rl_readline

Display a prompt, read one line with full editing, and return the entered string (or undef on EOF).

Synopsis

```
while (defined (my $line = $term->readline('prompt> '))) {  
    $term->addhistory($line) if $line =~ /\S/;  
    last if $line eq 'quit';  
}
```

The call blocks until the user presses Return. A bare Ctrl-D on an empty line returns undef. The returned string has its trailing newline already stripped, matching the underlying readline(3) C function. The prompt argument is optional; pass undef for none.

add_history

Append \$line to the end of the history list so it shows up in later Up-arrow recalls and searches.

Synopsis

```
$term->add_history($line) if $line =~ /\S/;
```

Readline does not deduplicate for you. Wrap the call in an if if you want to skip blank or repeated lines. Use write_history to persist the collected lines to disk.

read_history_range

Load lines from a history file into memory. Reads the whole file by default, or lines \$from..\$to when given.

Synopsis

```
$term->ReadHistory;                                # default file, full
↪range
$term->ReadHistory('~/.myhist', 100, -1); # last hundred-odd lines
```

Passing undef for \$filename reads from ~/.history. \$to = -1 means “to the end”. Returns 0 on success, a non-zero errno on failure.

rl_parse_and_bind

Apply a single line of inputrc syntax ("\\C-x\\C-r": re-read-init-file, set editing-mode vi,...).

Synopsis

```
$term->parse_and_bind('set editing-mode vi');
$term->parse_and_bind("\\C-l": clear-screen');
```

This is the programmatic door into the same grammar that ~/.inputrc uses. Returns 0 on a successful parse.

rl_store_function

Install a Perl code reference as the callback backing function-pointer slot \$id.

The current build stores the reference but does not yet invoke Perl-side callbacks - a future phase wires the C trampoline through.

Note

Unimplemented today Callback dispatch is a stub: the stored coderef is never invoked. `rl_completion_entry_function` and other function hooks therefore fall back to readline defaults.

5.2.30 Time

Namespace containing the following modules:

- `Time::HiRes` — High-resolution timers and sleeps: microsecond `gettimeofday`, sub-second sleep, alarm, and POSIX clocks.

`Time::HiRes`

High-resolution timers and sleeps: microsecond `gettimeofday`, sub-second sleep, alarm, and POSIX clocks.

Mirrors `perl5-modules/Time-HiRes-1.9764/HiRes.xs`. All functions operate through the perl5 C API (`p5api`).

Functions

Other Functions

`time`

`HiRes.xs:1393-1406 - time()`

`gettimeofday`

`HiRes.xs:1375-1391 - gettimeofday()`

`usleep`

`HiRes.xs:1139-1171 - usleep($microseconds)`

`nanosleep`

`HiRes.xs:1175-1191 - nanosleep($nanoseconds)`

`sleep`

`HiRes.xs:1207-1247 - sleep($seconds)`

`alarm`

`HiRes.xs:1296-1343 - alarm($seconds, $interval=0)`

ualarm

HiRes.xs:1265-1293 - ualarm(\$usecs, \$interval)

setitimer

HiRes.xs:1414-1445 - setitimer(\$which, \$value, \$interval)

getitimer

HiRes.xs:1447-1465 - getitimer(\$which)

clock_gettime

HiRes.xs:1567-1582 - clock_gettime(\$clock_id)

clock_getres

HiRes.xs:1598-1615 - clock_getres(\$clock_id)

clock

HiRes.xs:1673-1682 - clock()

clock_nanosleep

HiRes.xs:1631-1651 - clock_nanosleep(\$clock_id, \$nsec, \$flags)

tv_interval

tv_interval(\@t0, \@t1?) - pure Perl in HiRes.pm, native here

utime

HiRes.xs:1471-1551 - utime(\$atime, \$mtime, @files)

stat_impl

HiRes.xs:1696-1735 - stat/lstat with nanosecond timestamps. TODO: perl5 creates a fakeop and calls pp_stat to populate PL_statcache. We do direct libc stat. -M/-A/-C after Time::HiRes::stat() won't see cached result.

5.2.31 attributes

The machinery behind `sub foo : lvalue method { ... }` and `my $x : Shared`.

Perl attaches attributes to subs and variables in three ways, and this module is the meeting point for all of them:

- **Built-in attributes** that Perl itself consumes - lvalue, method, const, prototype(...) on subs, shared on variables. These flip flags on the CV (or set a prototype string) and never leave the core.
- **User-defined attributes** - any name Perl doesn't recognise is handed off to `MODIFY_${type}_ATTRIBUTES` in the caller's package. If that handler rejects an attribute (or is absent), the declaration is a fatal error. `FETCH_${type}_ATTRIBUTES` is the read-side counterpart.
- **A query API** - `attributes::get($ref)` returns the combined built-in + user-defined attribute list for a referent; `reftype` is kept here as a backward-compatible alias for `builtin::reftype`.

Attribute syntax is parsed by perl and the resulting list arrives here via `use attributes PACKAGE, \&sub, @attrs` - see the upstream attributes POD for the full compile-time expansion.

Functions

Query

`get`

List every attribute set on a referent - both the built-ins Perl recognises and any package-defined ones returned by a `FETCH_${type}_ATTRIBUTES` handler.

`reftype`

Return the underlying storage type of a reference, ignoring any class it's been blessed into.

Low-level

`modify_attrs`

Apply a list of attributes to a referent and return the ones Perl didn't recognise, so the caller can hand them to a user-defined `MODIFY_${type}_ATTRIBUTES` handler.

`fetch_attrs`

Return the built-in attributes currently set on a referent.

`guess_stash`

Work out which package a reference "belongs to", for the purpose of looking up `FETCH_${type}_ATTRIBUTES` / `MODIFY_${type}_ATTRIBUTES`.

`modify_attrs`

Apply a list of attributes to a referent and return the ones Perl didn't recognise, so the caller can hand them to a user-defined `MODIFY_${type}_ATTRIBUTES` handler.

Synopsis

```
my @unknown = attributes::_modify_attrs(\&foo, 'lvalue', 'Bent');

## 'lvalue' consumed internally; 'Bent' returned for package_
↳dispatch.
```

What you get back

The list of attribute names that weren't built-in. Built-in attributes (lvalue, method, const, prototype(. . .) for subs; shared for variables) are applied to the referent directly and do not appear in the returned list.

Examples

```
## Mark a sub as lvalue, no leftovers:

sub s :lvalue { $x }
my @left = attributes::_modify_attrs(\&s, 'lvalue');    # ()
```

```
## Install a prototype via attribute:

sub p { }
attributes::_modify_attrs(\&p, 'prototype($$)');        # ()
```

```
## Custom attribute flows through:

my @left = attributes::_modify_attrs(\&s, 'MyTag');    # ('MyTag')
```

Edge cases

- Called with zero args or a non-reference first arg: croaks with Usage: attributes::_modify_attrs(@attributes).
- A prototype(attribute without a closing) croaks with Unterminated attribute parameter in attribute list.
- -shared on a non-sub referent croaks with A variable may not be unshared.

Differences from upstream

- lvalue is always consumed as a built-in on CODE referents; the “already-defined subroutine” warning dispatch lives in attributes.pm and is not mirrored inside this XS entry.

See also

- `_fetch_attrs` - read-side counterpart, returns built-in attrs only.
- `get` - public API: `_fetch_attrs` plus `FETCH_${type}_ATTRIBUTES`.
- `_guess_stash` - used by `get` to locate the owning package.

fetch_attrs

Return the built-in attributes currently set on a referent.

Synopsis

```
my @attrs = attributes::_fetch_attrs(\&foo);
```

What you get back

A list of attribute names drawn only from the built-in set. For a CODE referent that list can contain lvalue and method. For any other referent the list is empty. User-defined attributes are not consulted here; use `get` for that.

Examples

```
sub s :lvalue :method { $x }
my @a = attributes::_fetch_attrs(\&s);    # ('lvalue', 'method')
```

```
my $h = {};
my @a = attributes::_fetch_attrs($h);    # ()
```

Edge cases

- Called with the wrong number of args or a non-reference: croaks with `Usage: attributes::_fetch_attrs($reference)`.
- Order is fixed: lvalue before method when both are set.

Differences from upstream

Fully compatible with upstream.

See also

- `get` - public wrapper that also calls `FETCH_${type}_ATTRIBUTES`.
- `_modify_attrs` - write-side counterpart.

get

List every attribute set on a referent - both the built-ins Perl recognises and any package-defined ones returned by a `FETCH_${type}_ATTRIBUTES` handler.

Synopsis

```
use attributes 'get';
my @attrs = get(\&foo);
```

What you get back

A flat list. The built-in names come first (`lvalue`, method for subs), followed by whatever `FETCH_${type}_ATTRIBUTES` returned for the owning package. Returns an empty list when nothing is set.

Examples

```
use attributes 'get';
sub s :lvalue { $x }
my @a = get(\&s);                                # ('lvalue')
```

```
## Package-defined attrs flow through FETCH_CODE_ATTRIBUTES:

package MyPkg;
my %tags;
sub MODIFY_CODE_ATTRIBUTES { my ($p, $c, @a) = @_; $tags{$c} = [@a];
→() }
sub FETCH_CODE_ATTRIBUTES { my ($p, $c) = @_; @{ $tags{$c} || [] }
→}
sub foo :Cached :Traced { }
my @a = attributes::get(\&foo);                  # ('Cached', 'Traced')
```

Edge cases

- Called with the wrong number of args or a non-reference: croaks with `Usage: attributes::get($reference)`.
- The owning package is discovered by the same rules as `_guess_stash`; a referent with no discoverable stash yields built-ins only.
- For a blessed referent, the blessed class is used as the stash, not the compilation package.

Differences from upstream

- Upstream falls back to `scalar caller` when `_guess_stash` returns `undef`; this implementation only consults the stash `_guess_stash` finds, so an anonymous sub compiled in a null-package context yields built-ins only instead of querying the caller's package.

See also

- `_fetch_attrs` - the built-in-only half of this function.
- `_guess_stash` - used to find the package whose `FETCH_${type}_ATTRIBUTES` is called.
- `reftype` - picks the `${type}` used in the handler name.

guess_stash

Work out which package a reference “belongs to”, for the purpose of looking up `FETCH_${type}_ATTRIBUTES` / `MODIFY_${type}_ATTRIBUTES`.

Synopsis

```
my $pkg = attributes::_guess_stash(\&Some::Module::foo);    #
↳ 'Some::Module'
```

What you get back

A package name string, or `undef` if no package can be determined. The rules, in order:

- A blessed referent returns its blessed class.
- A coderef returns the stash of the sub’s glob, or the compilation stash if the glob is missing.
- A typeglob reference returns the glob’s effective stash.
- Plain scalar / array / hash references that aren’t blessed return `undef`.

Examples

```
package Foo;
sub bar { }
my $p = attributes::_guess_stash(\&bar);    # 'Foo'
```

```
my $h = bless {}, 'My::Class';
my $p = attributes::_guess_stash($h);    # 'My::Class'
```

```
my $p = attributes::_guess_stash(\42);    # undef
```

Edge cases

- Called with the wrong number of args or a non-reference: croaks with `Usage: attributes::_guess_stash($reference)`.
- An anonymous sub with no compilation stash returns `undef`.

Differences from upstream

Fully compatible with upstream.

See also

- `get` - the primary consumer of this lookup.
- `reftype` - picks the `${type}` half of the handler name that `get` builds after calling `_guess_stash`.

reftype

Return the underlying storage type of a reference, ignoring any class it's been blessed into.

Synopsis

```
use attributes 'reftype';
my $t = reftype(\@a);      # 'ARRAY'
my $t = reftype(bless {}, 'X'); # 'HASH'
```

What you get back

One of SCALAR, ARRAY, HASH, CODE, GLOB, IO, FORMAT, LVALUE, or REGEXP. Returns undef when the argument is not a reference.

Examples

```
reftype(\1);          # 'SCALAR'
reftype([1, 2, 3]);   # 'ARRAY'
reftype(sub { });     # 'CODE'
reftype(bless [], 'Foo'); # 'ARRAY' - blessing is ignored
reftype(42);          # undef
```

Edge cases

- Not a reference, including undef: returns undef rather than croaking.
- Called with the wrong number of args: croaks with `Usage: attributes::reftype($reference)`.

Differences from upstream

Fully compatible with upstream. In upstream `attributes.pm` this name is a typeglob alias for `builtin::reftype`; this module provides the same behaviour as a direct XS entry.

See also

- `get` - uses `reftype` to build `FETCH_${type}_ATTRIBUTES`.
- `_guess_stash` - complementary helper for the “which package?” half of the same lookup.

5.2.32 mro

Method Resolution Order - how Perl decides which parent class's method a call inherits.

When a method is called on an object, Perl walks a list of classes looking for a matching subroutine. That list - the *linearized MRO* - is derived from `@ISA` and from the algorithm currently in effect for the class. The `mro` module lets you pick the algorithm, inspect the resulting class list, query the reverse `@ISA` relation, and manage the method cache.

Two algorithms are available. `dfs` (depth-first search) is the default and matches Perl's historical behaviour. `c3` produces the *C3 linearization* familiar from Python, Dylan, and Raku, which keeps subclasses ahead of their ancestors even under diamond inheritance. Pick `c3` when multiple inheritance makes the `dfs` order surprising; otherwise leave the default alone.

Loading the module also enables `next::method`, `next::can`, and `maybe::next::method` - the C3-aware cousins of `SUPER::`. They always walk the C3 linearization regardless of which MRO the class itself uses.

Synopsis

```
use mro;                                # enables next::method, next::can,
    ↪ maybe::next::method

use mro 'c3';                            # set C3 linearization for the current
    ↪ package
use mro 'dfs';                            # set depth-first (Perl's default)
    ↪ explicitly

my $order = mro::get_linear_isa('MyClass'); # arrayref of class
    ↪ names
my $algo  = mro::get_mro('MyClass');        # "dfs" or "c3"
mro::set_mro('MyClass', 'c3');             # set per-class at
    ↪ runtime
```

Functions

MRO selection

`set_mro`

Set the method-resolution algorithm for `$class` to either `"dfs"` or `"c3"`.

MRO introspection

`get_linear_isa`

Return the full ordered list of classes Perl would search for a method call on `$class`, starting with `$class` itself.

`get_mro`

Report which method-resolution algorithm is currently in effect for `$class`.

Synopsis

```
my $algo = mro::get_mro($class); # “dfs” or “c3”
```

Returns the string "dfs" or "c3". Classes that have never called `set_mro` or use `mro '...'` report "dfs" (Perl's default). Calling on a non-existent class also reports "dfs"; the call does not force the stash into existence.

`get_isarev`

List every class that inherits from `$class`, directly or transitively, via `@ISA`.

`get_pkg_gen`

Return a counter that increments every time a local method in `$class` changes or its local `@ISA` is modified.

`is_universal`

Report whether `$class` is `UNIVERSAL` itself or one of its ancestors via `@ISA`.

Method cache

`method_changed_in`

Force method-cache invalidation for `$class` and everything that inherits from it.

`invalidate_all_method_caches`

Invalidate the method cache for every package in the interpreter.

`next::dispatch`

`nextcan`

The `mro::_nextcan` primitive that underpins `next::method`, `next::can`, and `maybe::next::method`.

next_can

Return a code reference to the next implementation of the currently executing method, or undef if there is none.

next_method

Tail-call the next implementation of the currently executing method, passing the original arguments.

maybe_next_method

Tail-call the next implementation of the currently executing method if one exists; otherwise return quietly.

get_linear_isa

Return the full ordered list of classes Perl would search for a method call on `$class`, starting with `$class` itself.

Synopsis

```
my $list = mro::get_linear_isa($class); my $list = mro::get_linear_isa($class, $type);
```

The result is an array reference of package names. By default the list is produced using whichever algorithm is active for the class (dfs or c3). Pass an explicit "dfs" or "c3" as `$type` to force one without changing the class's setting.

The list always begins with `$class`, even if `$class` has no stash yet - in that case the result is [`$class`]. UNIVERSAL and its ancestors are not included; they are searched implicitly after the returned list is exhausted.

Inconsistent C3 hierarchies (conflicting parent orderings across a diamond) croak with Inconsistent hierarchy during C3 merge of class '...'. Recursive @ISA chains deeper than 100 levels croak with Recursive inheritance detected in package '...'.

set_mro

Set the method-resolution algorithm for `$class` to either "dfs" or "c3".

Synopsis

```
mro::set_mro($class, 'c3'); mro::set_mro($class, 'dfs');
```

After the call, subsequent method dispatches on `$class` - and subclasses that inherit the setting - walk the chosen linearization. `get_linear_isa($class)` reflects the new choice on the next invocation.

Any value other than "dfs" or "c3" croaks with Invalid mro name: '...'. Usage with any argument count other than two croaks with Usage: `mro::set_mro(classname, type)`. The stash for `$class` is created on demand if it does not yet exist, so selecting an MRO before the package is declared works.

get_isarev

List every class that inherits from `$class`, directly or transitively, via `@ISA`.

Synopsis

```
my $subclasses = mro::get_isarev($class); # arrayref
```

Returns an array reference of package names: each class whose linearized MRO contains `$class`. This is the inverse of `get_linear_isa` - given a parent, find its descendants.

UNIVERSAL is treated specially: its `isarev` list is not populated with every class in the program, even though every class implicitly inherits from it. Check with `is_universal` rather than `get_isarev` when you need that relationship.

Classes with no known subclasses produce an empty array reference.

get_pkg_gen

Return a counter that increments every time a local method in `$class` changes or its local `@ISA` is modified.

Synopsis

```
my $gen = mro::get_pkg_gen($class);
```

Useful to class-introspection code that wants a cheap “has anything I care about changed since last time?” check. Store the value, do your work, compare the next value against the stored one. Equal values mean nothing local changed; unequal values mean *something* did, but you still have to investigate what.

Changes in superclasses do **not** bump this number - only edits inside `$class` itself. A stash instantiated for a normal package starts at 1; `$class` with no stash at all returns 0. Wiping a stash (`undef %Pkg; :`) can reset the counter to 0 or 1 depending on how thorough the wipe was.

is_universal

Report whether `$class` is UNIVERSAL itself or one of its ancestors via `@ISA`.

Synopsis

```
if (mro::is_universal($class)) { ... }
```

Returns a true value exactly when every class in the program can inherit methods from `$class`. That happens in two cases: the class literally *is* UNIVERSAL, or something has pushed it into `@UNIVERSAL : : ISA` (directly or indirectly). Most programs never see the second case, but frameworks that mix a trait into UNIVERSAL rely on it.

A single argument is required; calling with any other arity croaks with Usage: `mro::is_universal(classname)`.

method_changed_in

Force method-cache invalidation for `$class` and everything that inherits from it.

Synopsis

```
mro::method_changed_in($class);
```

Normally Perl notices method installations on its own and you do not need to call this. The one documented case that escapes automatic detection is installing a constant subroutine by aliasing a readonly scalar into the stash - the trick `constant.pm` uses internally. If you build something similar, call `method_changed_in` afterwards so cached dispatches pick up the new sub.

Croaks with `No such class: '...'`! if the stash for `$class` does not exist. A single argument is required; any other arity croaks with `Usage: mro::method_changed_in(classname)`.

invalidate_all_method_caches

Invalidate the method cache for every package in the interpreter.

Synopsis

```
mro::invalidate_all_method_caches();
```

The next method dispatch after the call walks `@ISA` from scratch instead of reusing a cached resolution. Use this when you have globally disturbed the class graph - rebuilt `@ISA` wholesale across many packages, or hot-patched method tables - and cannot name every affected class individually. For a targeted invalidation, prefer `method_changed_in`.

Takes no arguments. Any non-empty call croaks with `Usage: mro::invalidate_all_method_caches()`.

nextcan

The `mro::_nextcan` primitive that underpins `next::method`, `next::can`, and `maybe::next::method`.

Synopsis

```
my $cv = mro::_nextcan($self, 0); # returns CV ref or undef
my $cv = mro::_nextcan($self, 1); # croaks if no method found
```

Called with the invocant and a boolean `$throw_if_not_found`. Looks at the currently executing method, walks the C3 linearization of `$self`'s class past that method's defining package, and returns the next implementation of the same method name as a CV reference.

When nothing matches, returns `undef` if the flag is `0` or croaks with `No next::method found` if the flag is `1`. Callers from outside a method context croak with `next::method/next::can/maybe::next::method` must be used in method context.

You rarely call this directly. Reach for `next::method`, `next::can`, or `maybe::next::method` instead - they package the common usage patterns around this primitive.

next_can

Return a code reference to the next implementation of the currently executing method, or `undef` if there is none.

Synopsis

```
my $cv = $self->next::can; $cv->($self, @args) if $cv;
```

Looks at the method currently on the call stack, walks the C3 linearization of `$self` past its defining package, and returns the first same-named method found further down the list as a CV reference. Unlike `next::method`, finding no such method is not fatal - it returns `undef` so you can decide how to handle the absence.

Must be called inside a method body. Top-level or non-method callers croak with `next::method/next::can/maybe::next::method` must be used in method context.

next_method

Tail-call the next implementation of the currently executing method, passing the original arguments.

Synopsis

```
sub greet { my $self = shift; $self->next::method(@_); # call the parent implementation }
```

Resolves the next method via the C3 linearization of `$self` and jumps to it as if with `goto &sub`. Control does not return to the caller of `next::method`; the resolved method's return value is the one the original caller sees.

Croaks with `No next::method found` when nothing further in the linearization implements the method. Use `maybe::next::method` or `next::can` if the absence of a further implementation is expected.

maybe_next_method

Tail-call the next implementation of the currently executing method if one exists; otherwise return quietly.

Synopsis

```
sub DESTROY { my $self = shift; ... # cleanup $self->maybe::next::method; }
```

Equivalent to `$self->next::method(@_) if $self->next::can` in the common case, but also works as the target of `goto &maybe::next::method` where the conditional Perl form does not. The usual reason to reach for it is a `DESTROY` or hook method where missing parent implementations are normal.

5.2.33 re

The re pragma and regular-expression introspection module.

Two jobs, one module:

- **As a pragma** - use re turns on lexically scoped regex behaviour: compile-time and run-time debug output (use re 'debug', use re 'debugcolor', use re qw(Debug ...)), permission to run code inside patterns derived from interpolated variables (use re 'eval'), taint propagation (use re 'taint'), stricter pattern checking (use re 'strict'), and default pattern flags (use re '/ix').
- **As an introspection API** - when imported, it exposes a small set of utility functions for inspecting compiled patterns (qr// objects) and the named-capture groups of the most recent successful match: is_regexp, regexp_pattern, regname, regnames, regnames_count, regmust, optimization.

Synopsis

```

use re 'debug';           # dump compile + match info
use re qw(Debug PARSE DUMP); # pick specific debug categories
use re 'eval';           # allow (?{...}) from variables
use re '/ix';            # add /i and /x to every regex

use re qw(is_regexp regexp_pattern regnames);
if (is_regexp($qr)) {
    my ($pat, $mods) = regexp_pattern($qr); # list: pattern, _
    ↪modifiers
    my $str          = regexp_pattern($qr); # scalar: "(?^
    ↪flags:pattern)"
}
"foobar" =~ /(?!<first>foo)(?!<rest>bar)/;
my @names = regnames();           # ('first', 'rest')
my $cap   = regname('first');     # 'foo'

```

What you can do with it

- Turn on regex debug output during one lexical scope and turn it back off outside - handy when a single pattern misbehaves in a large program.
- Ask whether an arbitrary value is really a compiled regex (even if it's blessed or overloaded) without having it stringify on you.
- Recover the original pattern text and modifier letters from a qr//, in either list or scalar form.
- Walk the named-capture groups of the last successful match by name, enumerate all names, or just count them.
- Peek at what the regex optimiser found - the longest anchored and floating substrings, the start class, the start-of-string anchors - to understand why a pattern is fast or slow.

Functions

Pragma behaviour

`install`

Return an integer handle to the regex engine and reset the debug-colour flag.

Pattern introspection

`is_regexp`

Return true if the argument is a compiled regular expression.

`regexp_pattern`

Return the pattern text and modifiers of a compiled regular expression.

Named captures

`regname`

Return the contents of a named capture group from the most recent successful match.

`regnames`

Return the names of the named capture groups in the most recent successful match.

`regnames_count`

Return the number of distinct named capture groups in the pattern of the most recent successful match.

Optimiser introspection

`regmust`

Return the longest anchored and longest floating fixed strings the optimiser extracted from a compiled pattern.

`optimization`

Return a hashref of the optimiser's compile-time findings for a compiled pattern.

`install`

Return an integer handle to the regex engine and reset the debug-colour flag.

Not intended for user code. This is the hook `use re` itself calls to wire the debug engine into `%^H{regcomp}`. End users should write `use re 'debug'`, `use re 'eval'`, `use re '/ix'` and friends instead of touching `install` directly.

Synopsis

```
my $handle = re::install(); # pragma plumbing; not for application
↳code
```

What you get back

An integer that, when reinterpreted as a C pointer, resolves to the compile-time regex engine. Any other use of the value produces undefined behaviour.

Differences from upstream

- The returned integer points to the core regex engine, not to the debug engine. The debug engine is not available in this build; use `re 'debug'` therefore cannot produce upstream-style compile-time dumps. Truthiness of the return value - which is all the non-debug subpragmas (`taint`, `eval`) rely on - is preserved.

regmust

Return the longest anchored and longest floating fixed strings the optimiser extracted from a compiled pattern.

A *fixed string* is a substring that must appear in any string that matches. An *anchored* fixed string is one whose offset from the start of the match is known; a *floating* fixed string can appear anywhere in a range of positions. The optimiser uses whichever it finds - preferring the longer, or the floating one if they are the same length - to skip positions that cannot match.

Synopsis

```
use re 'regmust';
my ($anchored, $floating) = regmust(qr/here .* there/x);

## $anchored = 'here', $floating = 'there'
```

What you get back

A two-element list (`$anchored`, `$floating`). Each element is either the fixed-string text as an SV or `&PL_sv_no` (the false-but-defined sentinel) when the optimiser has no fixed string of that kind. Returns `undef` if the argument is not a compiled pattern or its engine is unknown.

Examples

```
my ($a, $f) = regmust(qr/foo\s+bar/); # $a = 'foo', $f = 'bar'
my ($a, $f) = regmust(qr/\d+/);      # no fixed strings - both
↳false
my @got      = regmust("not a regex"); # empty list - ref wasn't a
↳qr//
```

Edge cases

- Argument is not a qr// - returns undef.
- Pattern's engine is not the core engine - returns undef.
- Pattern has only one kind of fixed string - the other slot is &PL_sv_no (false but defined).
- UTF-8 substrings are returned as-is; the optimiser stores the UTF-8 form in a separate slot and this function prefers the byte form when both are present.

Differences from upstream

- The returned fixed strings are the values picked by this build's optimiser; upstream's POD notes that the exact result is optimiser-dependent and may change between Perl versions. The same caveat applies here.

optimization

Return a hashref of the optimiser's compile-time findings for a compiled pattern.

Intended for writing tests that pin optimiser behaviour, not for application logic. The set of keys and the meaning of their values are not a stable API - they change as the optimiser changes, even between minor perl releases.

Synopsis

```
use re 'optimization';
my $info = optimization(qr/^\d+foo/);
print $info->{anchored};           # 'foo'
print $info->{'anchor SBOL'};      # 1
print $info->{minlen};             # minimum match length
```

What you get back

A hashref, or undef if the argument is not a compiled pattern or its engine is unknown. Keys include minlen, minlenret, gofs, anchored, anchored utf8, anchored min offset, anchored max offset, anchored end shift, floating, floating utf8, floating min offset, floating max offset, floating end shift, checking("anchored", "floating", or "none"), the booleans noscan, isall, anchor SBOL, anchor MBOL, anchor GPOS, skip, implicit, and stclass.

Edge cases

- Argument is not a qr// - returns undef.
- Pattern has no extracted substrings - the substring-related keys are undef or 0 and checking is "none".

Differences from upstream

- `stclass` is always undef. Upstream returns a textual rendering of the start class produced by `my_regprop`, which is compiled only with `PERL_EXT_RE_DEBUG` and is not available in this build.

is_regexp

Return true if the argument is a compiled regular expression.

Unlike `ref` or `UNIVERSAL::isa`, this function sees through blessing and overloading - it looks at the value's internal type tag, so a `qr//` blessed into another class, or an object with a stringification overload, is still correctly recognised.

Synopsis

```
use re 'is_regexp';
if (is_regexp($value)) { ... }
```

What you get back

The same true / false values Perl uses everywhere else: `&PL_sv_yes` (the canonical true) or `&PL_sv_no` (the canonical false-but-defined).

Examples

```
is_regexp(qr/foo/);           # true
is_regexp("not a regex");     # false
is_regexp(\"not a regex");    # false - scalar ref, not a qr//
my $blessed = bless qr/foo/, 'My::RE';
is_regexp($blessed);         # true - sees through the bless
```

Edge cases

- Called with zero or more than one argument - croaks with `Usage: re::is_regexp(sv)`.
- Non-reference scalars - always false.
- References that aren't `qr//` (arrayref, hashref, coderef, scalar ref) - always false.

regexp_pattern

Return the pattern text and modifiers of a compiled regular expression.

Context-sensitive: list context gives you the raw pattern and the modifier letters separately, which is usually what you want for inspection and tests. Scalar context gives you the stringified form Perl would produce if you interpolated the `qr//` into a string - the `(?^flags:pattern)` rendering.

Like `is_regexp`, this function looks at the internal regex pointer and is not fooled by blessing or overloading.

Synopsis

```
use re 'regex_pattern';
my ($pat, $mods) = regex_pattern(qr/foo/i); # ('foo', 'i')
my $str          = regex_pattern(qr/foo/i); # ' (?^i:foo) '
```

What you get back

- **List context:** a two-element list (\$pattern, \$modifiers). \$pattern is the original source text, without the (?^... : wrapper or trailing). \$modifiers is a short string of modifier letters - the character-set prefix (l, u, a, aa, or empty for the default) followed by whichever of m s i x xx n p are enabled.
- **Scalar context on a regex:** the full stringified form, e.g. (?^i:foo).
- **Scalar context on a non-regex:** &PL_sv_no - false, but defined. This lets idioms like if (regex_pattern(\$ref) eq ' (?^i:foo) ') run warning-free regardless of what \$ref holds.
- **List context on a non-regex:** the empty list.

Examples

```
my ($p, $m) = regex_pattern(qr/foo/);          # ('foo', '')
my ($p, $m) = regex_pattern(qr/foo/ix);        # ('foo', 'ix')
my ($p, $m) = regex_pattern(qr/(?u)bar/);      # ('(?u)bar', 'u')
my $s       = regex_pattern(qr/foo/i);         # ' (?^i:foo) '
my @e       = regex_pattern("not a regex");    # () - empty list
my $f       = regex_pattern("not a regex");    # false but defined
```

Edge cases

- Called with zero or more than one argument - croaks with Usage: re::regex_pattern(sv).
- UTF-8 patterns keep their UTF-8 flag on the returned pattern.
- The d charset (the default) produces no prefix letter.

regname

Return the contents of a named capture group from the most recent successful match.

With one argument, returns the first defined value captured by the named group. With a truthy second argument, returns an array reference holding every value the group captured - useful when the same name appears in a repeated subpattern or in multiple alternation branches.

Synopsis

```
use re 'regname';
"foobar" =~ /(?!<word>\w+)/;
my $first = regname('word');      # 'foobar'
my $all   = regname('word', 1);    # arrayref of all captures
```

What you get back

- Without \$all (or with a false \$all) - the first defined captured string, or undef if the name didn't match anything or didn't exist in the pattern.
- With a truthy \$all - an array reference containing every captured value for that name, in order.
- If there was no prior successful match - undef.

Examples

```
"2025-04-22" =~ /(?!<y>\d{4})-(?!<m>\d{2})-(?!<d>\d{2})/;
print regname('y');                      # '2025'

"abc123" =~ /(?!<n>\d+)|(?!<n>\w+)/;
print regname('n');                      # first defined capture
print join(',', @{ regname('n', 1) });    # every capture for 'n'

my $missing = regname('nope');            # undef - no such name
```

Edge cases

- Called with zero arguments - croaks with Usage: re::regname(name [, all]).
- No successful match yet in the current scope - returns undef.
- Name was declared but did not participate in the match - returns undef (single form) or an arrayref of undefs (\$all form).

regnames

Return the names of the named capture groups in the most recent successful match.

With no argument (or a false one), you get the names that were actually visited during the match. With a truthy argument, you get every name declared by the pattern, whether or not it participated.

Synopsis

```
use re 'regnames';
"foobar" =~ /(?!<head>foo)(?!<tail>bar)/;
my @used = regnames();                  # names that participated in the match
my @all  = regnames(1);                 # every name declared in the pattern
```

What you get back

A flat list of names in pattern order. If there was no prior successful match in the current scope, or the pattern has no named captures at all, returns undef.

Examples

```
"abc" =~ /( ?<first>a)( ?<second>b)( ?<third>c)/;
my @n = regnames();           # ('first', 'second', 'third')

"abc" =~ /( ?<a>x)|( ?<a>b)/;    # branch reset-style duplicate name
my @n = regnames();           # ('a') - one entry per distinct
                                ↪ name

my @n = regnames(1);           # all declared, even unmatched
```

Edge cases

- No prior successful match - returns undef.
- Pattern has no named captures - returns undef.
- Duplicate names (from branch-reset or multiple alternatives) appear once; use `regname($name, 1)` to see every value.

regnames_count

Return the number of distinct named capture groups in the pattern of the most recent successful match.

The count is always the number of *distinct* names defined in the pattern - it does not depend on which names participated in the match, and it does not necessarily match the length of what `regnames()` returns when that is called without its `$all` argument.

Synopsis

```
use re 'regnames_count';
"abc" =~ /( ?<a>x)|( ?<b>b)/;
my $n = regnames_count();      # 2
```

What you get back

An integer count, or undef if there was no prior successful match in the current scope.

Examples

```
"foobar" =~ /( ?<w>\w+)/;
print regnames_count();        # 1
```

(continues on next page)

(continued from previous page)

```
"abc" =~ /(?!<a>.)(!<b>.)(!<a>.)//;  # duplicate name 'a'
print regnames_count();             # 2 - counts distinct names
```

Edge cases

- No prior successful match - returns undef.
- Pattern has no named captures - returns 0.

5.2.34 version

Turn a version string into an object you can compare, print, and introspect.

Perl has three spellings for a version, and mixing them in one program is normal. `version` gives you a single object type that handles all three so `$a <=> $b` always does the right thing.

- **Decimal** - a plain number like 1.023 or "1.02_03". One integer part, then three-digit chunks after the dot. An underscore marks an alpha release.
- **Dotted-integer** - two or more integers separated by dots, e.g. 1.2.3. No `v` prefix required, but three or more components are treated as dotted-integer regardless.
- **v-string** - an explicit `v` prefix, e.g. `v1.23.45`. This form is recommended for `$VERSION` declarations and is required to pass `is_strict`.

Synopsis

```
use version 0.77;
our $VERSION = version->declare("v1.2.3");
my $v = version->parse("1.0203");
if ($v1 == $v2) { ... }           # numeric equality
@sorted = sort { $a <=> $b } @versions; # ordering
```

`use version` imports `qv` into the caller's package and installs an overloaded `UNIVERSAL::VERSION` so `$module->VERSION` returns a version object rather than a raw string.

Operators

version objects are overloaded so you can use the normal Perl comparison operators directly:

- `==`, `!=` - numeric equality
- `<`, `<=`, `>`, `>=`, `<=>` - numeric ordering
- `eq`, `ne`, `cmp` - same thing, stringwise spelling
- `" "` - stringifies to original form (dotted) or decimal, whichever the object was created with
- `0+` - numifies to a decimal

The overload fallback is on, so any relational operator not listed above works by auto-generation from `<=>`.

Functions

Construction

`new`

Build a version object from any recognisable version string.

`declare`

Build a version object, forcing dotted-integer form regardless of input shape.

Stringification

`stringify`

Render a version object as a string, using the form it was created in.

`numify`

Render a version as a plain decimal number, with three digits per sub-component.

`normal`

Render a version in canonical dotted-integer form with a leading v.

Comparison

`vcmp`

Compare two versions, returning -1, 0, or 1 like <=>.

`bool`

Give a version object its boolean truth value - true unless every component is zero.

`nomethod`

Catch-all handler for overloaded operators with no dedicated method.

Introspection

`is_qv`

Tell whether a version object is in dotted-integer form.

`is_alpha`

Tell whether a version marks an alpha / developer release.

is_lax

Test whether a string is acceptable to parse / new as a version.

is_strict

Test whether a string is a strict version suitable for CPAN indexing.

new

Build a version object from any recognisable version string.

Synopsis

```
my $v = version->new("1.0203");
my $v = version->parse("v1.2.3");
```

parse is an alias for new; the two are interchangeable. The argument may be a decimal ("1.0203"), a dotted-integer ("1.2.3"), or a v-string ("v1.23.45"). Whichever form goes in, \$v understands all the comparison operators and prints back in the form it came from.

What you get back

A blessed version object. It is numeric, stringly, and boolean all at once thanks to overloading: compare it with == or <=>, concatenate it into a string, test it with if (\$v). To pull the normalised numeric form out as a plain Perl number, call numify; for the canonical v1.2.3 string, call normal.

Examples

A decimal version behaves the way old-style \$VERSION did:

```
my $v = version->new("1.0203");
print $v;           # 1.0203
print $v->numify;    # 1.020300
```

A dotted-integer version keeps its components intact:

```
my $v = version->new("1.2.3");
print $v->normal;    # v1.2.3
print $v->numify;    # 1.002003
```

Cross-form comparison works - the whole point of the module:

```
version->new("1.2.3") == version->new("1.002003"); # true
```

An underscore marks an alpha release:

```
my $v = version->new("1.02_03");
print $v->is_alpha;  # 1
```

Edge cases

- Called with no version argument - returns a version equivalent to "0".
- Leading whitespace and a single leading v are both accepted.
- Invocation as `version::new("version", $str)` is accepted; the first argument is treated as the class name.

Differences from upstream

Fully compatible with upstream version 0.9929.

See also

- `declare` - same idea, but always produces a dotted-integer object
- `qv` - alias for `declare`, handy inside `$VERSION` lines
- `numify` - plain-number form of an existing object
- `normal` - canonical `v1.2.3` form of an existing object
- `is_alpha` - test whether the string carried an underscore

declare

Build a version object, forcing dotted-integer form regardless of input shape.

Synopsis

```
our $VERSION = version->declare("v1.2.3");
our $VERSION = qv("v1.2.3");
```

`declare` is the recommended way to set `$VERSION` to a dotted-integer version. `qv` is its short-name alias, imported into the caller's package by use `version`. Both treat the argument as dotted-integer even when the input looks decimal - `declare("1.2")` yields `v1.2.0`, not `1.200`.

What you get back

A blessed version object with the `qv` flag set, so stringification produces `v1.2.3` rather than a decimal number.

Examples

The canonical `$VERSION` idiom:

```
use version;
our $VERSION = version->declare("v1.2.3");
print $VERSION;           # v1.2.3
```

`qv` is the same thing, shorter:

```
use version;  
my $v = qv("1.2");  
print $v->normal;      # v1.2.0
```

A decimal argument is reinterpreted as dotted-integer:

```
my $v = version->declare("1.2");  
print $v->normal;      # v1.2.0 - not 1.200
```

Edge cases

- Called with no argument - produces `v0.0.0`.
- A trailing underscore still sets the alpha flag, but the object remains in dotted-integer form.

Differences from upstream

Fully compatible with upstream version 0.9929.

See also

- `new` - parse a version string in whatever form it happens to be
- `qv` - shorter spelling of `declare`
- `normal` - dotted-integer form of an existing object
- `is_qv` - test whether an object is in dotted-integer form

stringify

Render a version object as a string, using the form it was created in.

Synopsis

```
my $s = $v->stringify;  
my $s = "$v";      # same thing, via overloaded ""
```

Interpolating a version object into a string calls `stringify`. A dotted-integer object prints as `v1.2.3`; a decimal object prints as the original decimal string, preserving trailing zeros and underscores.

What you get back

A plain Perl string. No trailing newline, no quotes.

Examples

Dotted-integer round-trips through normal form:

```
my $v = version->declare("v1.2.3");
print "$v";           # v1.2.3
```

Decimal keeps its exact spelling:

```
my $v = version->new("1.0203");
print "$v";           # 1.0203
```

Differences from upstream

Fully compatible with upstream version 0.9929.

See also

- `numify` - the decimal numeric form, always
- `normal` - the dotted-integer form, always
- `is_qv` - tells you which of the two stringify will pick

numify

Render a version as a plain decimal number, with three digits per sub-component.

Synopsis

```
my $n = $v->numify;
my $n = 0 + $v;           # same thing, via overloaded 0+
```

The first component keeps its natural width; every component after the dot is padded or split into three-digit groups. This is the form old-style \$VERSION comparisons expected.

What you get back

A string that reads as a Perl number: integer part, a single dot, then exactly three digits per trailing component.

Examples

A dotted-integer version collapses into packed three-digit groups:

```
my $v = version->declare("v1.2.3");
print $v->numify;         # 1.002003
```

A decimal version is already in numify shape:

```
my $v = version->new("1.0203");
print $v->numify;      # 1.020300
```

Use it when an older API expects a float:

```
require Some::Module;
die "need 1.2.3+" if Some::Module->VERSION->numify < 1.002003;
```

Differences from upstream

Fully compatible with upstream version 0.9929.

See also

- `normal` - the dotted counterpart
- `stringify` - whichever of the two matches the object's origin

normal

Render a version in canonical dotted-integer form with a leading v.

Synopsis

```
my $s = $v->normal;
```

Always returns the v1.2.3 shape, padded out to at least three components. Use this when you want a stable, comparable string form that ignores how the object was originally built.

What you get back

A Perl string beginning with v, with each integer component separated by a single dot. At least three components are always present - missing ones are zero-filled.

Examples

Decimal input normalises into dotted-integer form:

```
my $v = version->new("1.0203");
print $v->normal;      # v1.20.300
```

Short dotted input gets padded out:

```
my $v = version->declare("v1.2");
print $v->normal;      # v1.2.0
```

Differences from upstream

Fully compatible with upstream version 0.9929.

See also

- `numify` - the decimal counterpart
- `stringify` - whichever form matches the object's origin

`is_qv`

Tell whether a version object is in dotted-integer form.

Synopsis

```
if ($v->is_qv) { ... }
```

Returns a true value for objects built via `declare` or `qv`, or from a string with two or more dots (e.g. `"1.2.3"`). Returns false for decimal versions like `"1.023"`.

What you get back

A boolean: 1 if the `qv` flag is set on the object, the empty string otherwise.

Examples

```
version->declare("v1.2.3")->is_qv;    # 1
version->new("1.023")->is_qv;          # ""
version->new("1.2.3")->is_qv;          # 1 - three components implies_
↪qv
```

Differences from upstream

Fully compatible with upstream version 0.9929.

See also

- `is_alpha` - the other introspection flag
- `declare` - the common way to set the `qv` flag
- `normal` - the stringification `is_qv` objects choose for `""`

`is_alpha`

Tell whether a version marks an alpha / developer release.

Synopsis

```
if ($v->is_alpha) { ... }
```

An alpha version is any version whose source string contained an underscore, e.g. "1.02_03" or "v1.2_3". CPAN uses the underscore to mark a non-indexed developer release.

What you get back

A boolean: 1 if the version's source string contained an underscore, the empty string otherwise.

Examples

```
version->new("1.02_03")->is_alpha;    # 1
version->new("1.0203")->is_alpha;      # ""
```

Differences from upstream

Fully compatible with upstream version 0.9929.

See also

- `is_qv` - the other introspection flag
- `new` - where the underscore is detected

is_lax

Test whether a string is acceptable to parse / new as a version.

Synopsis

```
version::is_lax($str);
version->is_lax($str);
```

Accepts anything the constructor would accept: decimal, dotted-integer, v-string, with or without an underscore for alpha. Use it to validate user input before feeding it to `parse`.

What you get back

A boolean: 1 if the string parses as any valid version, the empty string otherwise.

Examples

```
version::is_lax("1.0203");    # 1
version::is_lax("v1.2.3");    # 1
version::is_lax("1.02_03");   # 1
version::is_lax("");          # ""
version::is_lax("abc");       # ""
```

Differences from upstream

Fully compatible with upstream version 0.9929.

See also

- `is_strict` - the tighter validator, used for CPAN-indexable versions
- `new` - accepts the same set of strings

is_strict

Test whether a string is a strict version suitable for CPAN indexing.

Synopsis

```
version::is_strict($str);
version->is_strict($str);
```

Strict versions rule out alpha underscores and unusual shapes. A decimal version must be a single dot between two digit runs; a dotted-integer version must start with `v` and have at least three components. This is the form PAUSE expects for permanent releases.

What you get back

A boolean: 1 if the string matches the strict grammar, the empty string otherwise.

Examples

```
version::is_strict("1.02");    # 1
version::is_strict("v1.2.3");  # 1
version::is_strict("1.2.3");   # "" - needs the v prefix
version::is_strict("1.02_03"); # "" - underscores disallowed
version::is_strict("v1.2");    # "" - needs 3+ components
```

Differences from upstream

Fully compatible with upstream version 0.9929.

See also

- `is_lax` - the permissive counterpart
- `declare` - the recommended way to produce a strict dotted-integer

vcmp

Compare two versions, returning -1, 0, or 1 like `<=>`.

Synopsis

```
my $ord = $v1 <=> $v2;
my $ord = $v1 cmp $v2;
my $ord = $v1->vcmp($v2);
```

This is the function the overloaded operators dispatch to. Both operands are normalised to dotted-integer parts before comparison, so mixing decimal, dotted-integer, and v-string forms on either side works as expected. `==`, `!=`, `<`, `<=`, `>`, `>=`, `eq`, `ne`, and `cmp` all build on top of `vcmp`.

What you get back

An integer: -1 if the left side sorts first, 0 if equal, 1 if the right side sorts first.

Examples

Ordering different notations produces the intuitive result:

```
version->new("1.2.3") <=> version->new("1.002003");    # 0
version->new("1.2.4") <=> version->new("1.2.3");        # 1
```

The overloaded operators are the common entry point:

```
my @ordered = sort { $a <=> $b } map { version->parse($_) }
               qw(1.2 v1.3.0 1.2.5 1.19);
```

Alpha versions sort below the corresponding release:

```
version->new("1.02_03") < version->new("1.0203");      # true
```

Edge cases

- One side shorter than the other - the missing trailing components are treated as zero, so `v1.2` equals `v1.2.0`.
- A non-version scalar on either side - parsed with `new` first, so `$v == "1.2.3"` is well-defined.

Differences from upstream

Fully compatible with upstream version 0.9929.

See also

- `new` - builds the objects on either side
- `numify` - useful when you need a sortable key for non-version code
- `boolean` - the overloaded truth test, not an ordering

bool

Give a version object its boolean truth value - true unless every component is zero.

Synopsis

```
if ($v) { ... }
```

This backs the overloaded `bool` operator. A version is false only when it has no non-zero component, i.e. it compares equal to `v0.0.0`. Every other version is true, including `v0.0.1`.

What you get back

A boolean: 1 for any non-zero version, the empty string for `v0 / 0.000 / v0.0.0`.

Examples

```
version->new("0")->boolean;      # ""
version->declare("v0.0.0")->boolean; # ""
version->new("0.001")->boolean;    # 1
version->declare("v0.0.1")->boolean; # 1
```

Differences from upstream

Fully compatible with upstream version 0.9929.

See also

- `vcmp` - the three-way comparison behind the ordering operators

nomethod

Catch-all handler for overloaded operators with no dedicated method.

This is the `nomethod` hook and the `((` marker that `overload` installs in the `version` package. It always returns `undef`, which signals Perl's overload machinery to fall back to auto-generation from `<=>` and `cmp` - which is exactly what the module wants. User code normally never calls this directly.

Differences from upstream

Fully compatible with upstream version 0.9929.

See also

- `vcmp` - the method everything ultimately falls back to

ERRATA

This chapter records places where the canonical Perl 5 documentation is wrong, ambiguous, or stale, together with what each PetaPerl runtime actually does. Each entry has the same shape:

1. **Upstream wording** - the relevant text from the official perl5-upstream/pod/source, quoted verbatim.
2. **Correction** - what the documentation should say, established from upstream's own C source and from empirical probing.
3. **ppperl behavior** - what pperl does. Where this differs from upstream perl5, the divergence is flagged and tracked.

The goal of this chapter is operational, not editorial: if a user hits surprising behavior around one of these topics, they should find an answer here and know which runtime they are running.

6.1 Identifier length cap (Identifier too long)

6.1.1 Upstream wording

perl5-upstream/pod/perlvar.pod says, in the **Syntax of Variable Names** section:

Variable names in Perl can have several formats. Usually, they must begin with a letter or underscore, in which case they can be arbitrarily long (up to an internal limit of 251 characters) and may contain letters, digits, underscores, or the special sequence `::` or `'`.

This wording is misleading on two counts.

6.1.2 Correction

The correct statement, as of the **Perl 5.42** stable series, is:

Identifiers may be up to **251 bytes** long. Names exceeding this length cause a parse-time error: Identifier too long.

Two things to take from this:

1. **Bytes, not characters.** The cap is measured in bytes of the source representation, not in Unicode code points. UTF-8 identifiers therefore reach the limit at fewer characters:

Char width	Last accepted	First rejected
1 byte (ASCII)	251 chars	252 chars
2 bytes (e.g. é)	125 chars	126 chars
3 bytes (e.g. Υ)	83 chars	84 chars
4 bytes (e.g. ᚠ - Gothic)	62 chars	63 chars

In every case the rejection occurs when the byte length first exceeds 251. The “251 characters” in the upstream text is correct only for the ASCII case.

2. **Hard error, not “internal limit”.** The phrasing “up to an internal limit” suggests something soft. It is not - perl5’s parser raises a fatal `Identifier too long` exception. The message is the only thing the user sees; there is no truncation, warning, or fallback.

The proximate cause in 5.42 is the `tokenbuf[256]` buffer in the parser state, combined with `e = dest_end - 3` (sigil byte + trailing `\0` + safety margin) accounted for inside `S_parse_ident`. The combination yields 252 bytes total ÷ subtract sigil = 251 bytes of identifier name proper.

Upstream bleed has already raised the cap

Independent of this errata, perl5 development (post-5.42) has already raised the cap. Commit [8785c114b5](#) (“parser.h Allow up to 256 characters in a token”), which first appears in the v5.43.4 development tag, replaces the 256-byte tokenbuf with a 1024-byte buffer. The next stable series (Perl 5.44) is therefore expected to permit identifiers up to roughly 1020 bytes by default. The 251-byte cap is a 5.42-and-earlier characteristic.

6.1.3 pperl behavior

ppperl is the c2rust transpilation of perl5’s C interpreter. Its identifier-length behavior is:

Length	perl5 5.42.2	ppperl
251 bytes	accept	accept
252 bytes	reject	accept (divergence)
1019 bytes	reject	accept (divergence)
1020 bytes	reject	reject
1024 bytes	reject	reject

The divergence is real and visible to user code. **The cause is not in the c2rust transpilation itself.** The c2rust output is a faithful, line-for-line mirror of the C source it is given. The cause is upstream-checkout drift: `perl5-upstream/` is currently pinned to a bleed snapshot (post-5.43.4) rather than to a 5.42.x maintenance tag. Bleed has the raised tokenbuf - the c2rust mirror inherits it.

This is an instance of a broader category of latent divergences: upstream bleed has accumulated 140+ commits to `toke.c` alone since the 5.42.2 tag, and the same arithmetic - *whatever bleed does is what pperl does* - applies to every one of them. Other likely-affected areas (parser-visible only):

- **Enhanced `qr/.../xx` (PPC 0026)** lands in bleed; the whitespace/comment handling inside character classes differs from 5.42.
- **`intuit_more` bug fixes** (multiple commits): edge cases in `$foo[bar]` vs `$foo{bar}` ambiguity resolution behave better in pperl than in 5.42.
- **`S_parse_ident` API rewrite**: the error position reported by identifier-related parse errors moved between 5.42 and bleed.

The remediation is not to patch the c2rust output; it is to pin perl5-upstream/ to v5.42.2 and re-run the transpile pipeline.

Until that is done, pperl is - strictly speaking - “perl5 next” rather than “perl5 5.42 stable”. For most user code this is invisible; for the corner cases above, behavior matches bleed, not 5.42.

6.1.4 Tests

The conformance tests covering this errata live in:

- `t/01-parsing/090-identifier-length-ascii.t`
- `t/01-parsing/091-identifier-length-utf8.t`

They probe the boundary at 251 bytes (ASCII and 2/3/4-byte UTF-8) and at the upstream-blead secondary cutoff near 1024 bytes. Each subtest encodes a perl5 5.42 expectation; pperl currently fails subsets of the suite as documented above. The failing subtests are the operational tracking record for this errata entry.

DISCLAIMER & BUG REPORTS

A.1 PetaPerl is not upstream Perl

PetaPerl is an independent, from-scratch Perl 5 runtime written in Rust. It is **not** a build of the CPAN reference perl interpreter. Many modules you are used to from CPAN - PDL, Scalar::Util, Storable, POSIX, and dozens more - ship with PetaPerl as **native Rust implementations** built into the interpreter. No XS, no C compiler required, no CPAN install step.

This has one consequence that matters to every user:

Important

Any bug, missing feature, or behavioral difference you observe when using a module name that exists on CPAN (PDL::Ufunc, Scalar::Util, IO::File, ...) while running under PetaPerl is **our bug, not upstream's**.

Do **not** file these issues against the upstream CPAN project. Upstream maintainers are volunteers working on a codebase that is not the one you are running.

A.2 Bug reports

PetaPerl is a **research preview under high-velocity change**. A public bug-reporting mechanism is not yet in place.

Note

What we ask you to do today:

- **Do not file issues against upstream CPAN projects** for behaviour you observe in pperl's native re-implementations. Upstream maintainers work on a different codebase and can't fix ours. This is the one message this page exists to deliver.
- **Hold on to your findings.** A minimal reproducing script, the output of `ppperl -V`, and a short description of expected vs. actual behaviour are the three things any future tracker will ask for. Saving them now means less work later.
- **Check whether it's already known.** Every module page carries a *Differences from upstream* section listing deliberate divergences. If your observation matches a bullet there, it's not a bug - it's documented behaviour.

When a public tracker opens, this page will be updated with the submission URL.

A.3 Intent of compatibility

PetaPerl aims at ~100 % behavioral compatibility with **Perl 5.42** and with the specific versions of the CPAN modules we ship (see `ppperl -V`, `PPERL_MODS`). Where we diverge, we diverge deliberately and document it.

Older language forms (e.g. `use v5.18`; features that differed from later versions) are not a compatibility target. PetaPerl follows current upstream Perl, not historical upstream Perl.

A.4 Upstream's contributions

The CPAN modules whose names we carry represent years of work by volunteer maintainers who deserve credit for the interface design and original implementation even when our code shares no lines with theirs. We thank them, and we do not want their inboxes flooded with issues produced by a runtime they did not write. That is the single most important reason this page exists.

A.5 License

See the [License](#) page for the full terms of the PetaMem Research Preview License that governs your use of this Software.

LICENSE**B.1 PetaMem Research Preview License**

Copyright © 2026 PetaMem s.r.o. (IČ: 26512521), Prague, Czech Republic. All rights reserved.

B.1.1 1. Definitions

“Software” means pperl (PetaPerl), a Perl 5-compatible interpreter implemented in Rust, distributed in binary form under this License, including any accompanying documentation. “Licensor” means PetaMem s.r.o. “You” means the individual or entity exercising permissions granted by this License.

B.1.2 2. Grant

Subject to the terms below, Licensor grants You a worldwide, royalty-free, non-exclusive, non-transferable license to use, copy, run, benchmark, evaluate, and redistribute the Software in unmodified binary form, for any purpose, commercial or non-commercial, for so long as the Software’s version number is below 1.0.

B.1.3 3. Attribution and Non-Appropriation

You shall not represent the Software, in whole or in part, as Your own work, nor remove, obscure, or alter any copyright notices, version identifiers, author attributions, or license texts contained in or accompanying the Software. Any redistribution must preserve this License and all original notices intact.

B.1.4 4. Reverse Engineering

Reverse engineering, disassembly, and decompilation of the Software for the purposes of interoperability, security research, academic study, and personal curiosity are expressly permitted. Publication or redistribution of source code, derived source, or modified binaries obtained through such activity is not granted under this License and is prohibited.

B.1.5 5. Research Preview Status

The Software is a research preview. Its interfaces, behavior, file formats, performance characteristics, and licensing terms may change without notice in any subsequent release. This License applies only to the version of the Software with which it was distributed.

B.1.6 6. No Warranty

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, AND ACCURACY. YOU ASSUME ALL RISK ARISING FROM YOUR USE OF THE SOFTWARE.

B.1.7 7. Limitation of Liability

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL LICENSOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, OR EXEMPLARY DAMAGES ARISING OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THIS LICENSE, REGARDLESS OF THE LEGAL THEORY ASSERTED, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

B.1.8 8. Termination

This License and all rights granted hereunder terminate automatically upon Your breach of any term. Upon the public release of version 1.0 or later of the Software, this License ceases to apply to subsequent versions, which will be governed by their own license terms.

B.1.9 9. Reservation of Rights

All rights not expressly granted are reserved by Licensor. No rights in source code, trademarks, patents, or trade secrets are conveyed by this License except as explicitly stated. “PetaMem”, “PetaPerl”, and “ppperl” are used here as identifiers of the Software and its origin; no trademark license is granted.

B.1.10 10. Governing Law and Venue

This License shall be governed by and construed in accordance with the laws of the Czech Republic, without regard to conflict of law provisions. The exclusive venue for any dispute arising hereunder shall be the competent courts of Prague, Czech Republic.

GLOSSARY

Terms, acronyms, and project-specific vocabulary.

ppperl

The PetaPerl runtime - a Rust reimplementation of Perl 5, binary name `ppperl`. Reads the same Perl source code the stock `perl` interpreter does. A behavioural mirror of `perl5`: semantics, edge cases, magic. Where a test disagrees with upstream `perl5`, `ppperl` is the one that's wrong. Adds a JIT and auto-parallelization on top.

SV

Scalar Value. A Perl scalar as represented in `ppperl`'s runtime - the enum lives in `src/runtime/sv.rs` and mirrors `perl5`'s SV layout where the semantics depend on it.

JIT

Just-In-Time compilation. `ppperl` compiles hot loops to machine code via Cranelift; triggered automatically on eligible `for/while` bodies.

named unary operator

A Perl built-in that takes **exactly one argument** at a specific precedence level - tighter than comparison operators, looser than the arithmetic and shift operators. Unlike a **list operator** (which greedily slurps the rest of the expression), a named unary takes its one argument and stops there. Examples: `defined`, `exists`, `ref`, `scalar`, `length`, `uc/lc`, `chr/ord`, `int`, `abs`, `sqrt`, `sin/cos`, `log`, `exp`, `-e/-r/-f/...` file tests. See `perlop` precedence row 10.

list operator

A Perl built-in that takes a **list of arguments** and, at the right-hand side of the operator, greedily consumes the rest of the expression. Examples: `print`, `sort`, `push`, `unshift`, `join`, `split`. Contrast with **named unary operator**.

This is an initial seed. Terms referenced by `{term}` roles elsewhere in the docs should be added here when they appear.

Non-alphabetical

- \$?
 - decoding, single: wait(2),
single: signal; child process,
single: exit status, [426](#)
- \$!
 - volatility, single: errno, [430](#)
- __CLASS__
 - Perl built-in, [559](#)
- __FILE__
 - Perl built-in, [562](#)
- __LINE__
 - Perl built-in, [564](#)
- __PACKAGE__
 - Perl built-in, [567](#)
- __SUB__
 - Perl built-in, [571](#)
- _v
 - Config function, [1454](#)
- A
- abs
 - Perl built-in, [574](#)
 - POSIX function, [1703](#)
- abs_path
 - Cwd function, [1457](#)
- accept
 - Perl built-in, [576](#)
- access
 - POSIX function, [1712](#)
- add
 - Digest::MD5 function, [1478](#)
 - Digest::SHA function, [1501](#)
- add_bits
 - Digest::MD5 function, [1477](#)
 - Digest::SHA function, [1502](#)
- add_history
 - Term::ReadLine::Gnu function, [1814](#)
- add_ui_gmp
 - Math::GMP function, [1648](#)
- addfile
 - Digest::MD5 function, [1480](#)
 - Digest::SHA function, [1506](#)
- addfilebin
 - Digest::SHA function, [1503](#)
- addfileuniv
 - Digest::SHA function, [1504](#)
- advise
 - File::Map function, [1550](#)
- alarm
 - Perl built-in, [579](#)
- algorithm
 - Digest::SHA function, [1510](#)
- all
 - List::Util function, [1624](#)
 - Perl built-in, [582](#)
- all_keys
 - Hash::Util function, [1595](#)
- any
 - List::Util function, [1623](#)
 - Perl built-in, [585](#)
- append mode, single: PIPE_BUF,
single: atomic append, [421](#)
- asctime
 - POSIX function, [1717](#)
- at
 - PDL::Core function, [1666](#)
- atan2
 - Perl built-in, [587](#)
- attributes
 - Perl module, [1816](#)
- attributes::fetch_attrs
 - Perl function, [1819](#)
- attributes::get
 - Perl function, [1819](#)
- attributes::guess_stash
 - Perl function, [1821](#)
- attributes::modify_attrs
 - Perl function, [1817](#)
- attributes::reftype

- Perl function, [1822](#)
- autodie, single: exceptions,
 - pair: autodie
 - lexical scope, [430](#)
- B
- B
 - Perl module, [1427](#)
- b64digest
 - Digest::MD5 function, [1483](#)
 - Digest::SHA function, [1509](#)
- B::cchar
 - Perl function, [1438](#)
- B::cstring
 - Perl function, [1437](#)
- B::cv_gv
 - Perl function, [1439](#)
- B::cv_name_hek
 - Perl function, [1439](#)
- B::gv_name
 - Perl function, [1440](#)
- B::gv_safename
 - Perl function, [1440](#)
- B::gv_sv
 - Perl function, [1440](#)
- B::hash
 - Perl function, [1438](#)
- B::hv_name
 - Perl function, [1440](#)
- B::main_root
 - Perl function, [1437](#)
- B::object_2svref
 - Perl function, [1439](#)
- B::op_ppaddr
 - Perl function, [1441](#)
- B::opnumber
 - Perl function, [1436](#)
- B::pv_pv
 - Perl function, [1439](#)
- B::sv_refcnt
 - Perl function, [1438](#)
- B::sv_svtype
 - Perl function, [1438](#)
- B::svref_2object
 - Perl function, [1436](#)
- B::walkoptree
 - Perl function, [1437](#)
- B::walkoptree_debug
 - Perl function, [1437](#)
- bdiv
 - Math::GMP function, [1649](#)
- bfac
 - Math::GMP function, [1651](#)
- bgcd
 - Math::GMP function, [1650](#)
- bidirectional pipe, single:
 - IPC::Open2, single:
 - IPC::Open3, single:
 - co-process, [426](#)
- binary files, single: read
 - buffered, single: PerlIO layers; :raw, [424](#)
- bincompat_options
 - Config function, [1452](#)
- bind
 - Perl built-in, [589](#)
- binmode
 - Perl built-in, [592](#)
 - strip layers, single: binmode; push layer, [424](#)
- binmode, single: PerlIO layers
 - at open, single: PerlIO layers; via binmode, [422](#)
- blcm
 - Math::GMP function, [1651](#)
- bless
 - Perl built-in, [595](#)
- blshift
 - Math::GMP function, [1651](#)
- bmodinv
 - Math::GMP function, [1649](#)
- bmulf
 - Math::GMP function, [1649](#)
- bnok
 - Math::GMP function, [1652](#)
- bool
 - version function, [1848](#)
- bool_overload
 - PDL::Ops function, [1673](#)
- break
 - Perl built-in, [599](#)
- broot
 - Math::GMP function, [1652](#)
- brootrem
 - Math::GMP function, [1653](#)
- bsd_glob
 - File::Glob function, [1538](#)
- bsqrt
 - Math::GMP function, [1653](#)
- bucket_array
 - Hash::Util function, [1591](#)
- bucket_info

Hash::Util function, 1591
bucket_ratio
Hash::Util function, 1591
bucket_stats
Hash::Util function, 1592
bucket_stats_formatted
Hash::Util function, 1593
C
caller
Perl built-in, 602
catch
Perl built-in, 606
cchar
B function, 1438
chdir
Cwd function, 1460
Perl built-in, 609
chmod
Perl built-in, 612
chomp
Perl built-in, 615
chop
Perl built-in, 618
chown
Perl built-in, 620
chr
Perl built-in, 623
chroot
Perl built-in, 625
class
Perl built-in, 628
cleanup
File::Temp function, 1569
Clone
Perl module, 1441
clone
Clone function, 1442
Digest::MD5 function, 1474
Digest::SHA function, 1498
Clone::clone
Perl function, 1442
close
error checking, single: \$?, 430
error checking, single: flush, 421
Perl built-in, 633
closed_over
PadWalker function, 1730
closedir
Perl built-in, 636
cmpstat
File::Temp function, 1572
compile_date
Config function, 1451
Config
Perl module, 1443
Config::_v
Perl function, 1454
Config::bincompat_options
Perl function, 1452
Config::compile_date
Perl function, 1451
Config::config_re
Perl function, 1451
Config::config_sh
Perl function, 1446
Config::config_vars
Perl function, 1447
Config::exists
Perl function, 1448
Config::fetch
Perl function, 1448
Config::firstkey
Perl function, 1450
Config::header_files
Perl function, 1453
Config::local_patches
Perl function, 1452
Config::myconfig
Perl function, 1446
Config::nextkey
Perl function, 1450
Config::non_bincompat_options
Perl function, 1453
Config::store
Perl function, 1449
config_re
Config function, 1451
config_sh
Config function, 1446
config_vars
Config function, 1447
connect
Perl built-in, 639
constant
POSIX function, 1720
context
Digest::MD5 function, 1484
continue
Perl built-in, 642
cos

- Perl built-in, [645](#)
- crypt
 - Perl built-in, [648](#)
- csh_glob
 - File::Glob function, [1540](#)
- cstring
 - B function, [1437](#)
- ctime
 - POSIX function, [1718](#)
- cv_gv
 - B function, [1439](#)
- cv_name_hek
 - B function, [1439](#)
- Cwd
 - Perl module, [1455](#)
- cwd
 - Cwd function, [1456](#)
- Cwd::abs_path
 - Perl function, [1457](#)
- Cwd::chdir
 - Perl function, [1460](#)
- Cwd::cwd
 - Perl function, [1456](#)
- Cwd::fast_abs_path
 - Perl function, [1459](#)
- D
- Data::Dumper
 - Perl module, [1462](#)
- Data::Dumper::dumpxs
 - Perl function, [1463](#)
- dbmclose
 - Perl built-in, [651](#)
- dbmopen
 - Perl built-in, [653](#)
- dclone
 - Storable function, [1790](#)
- DeadCode
 - Devel::Peek function, [1469](#)
- declare
 - version function, [1840](#)
- decode_base64
 - MIME::Base64 function, [1631](#)
- decode_base64url
 - MIME::Base64 function, [1634](#)
- decode_qp
 - MIME::QuotedPrint function, [1638](#)
- decoded_base64_length
 - MIME::Base64 function, [1636](#)
- defer
 - Perl built-in, [656](#)
- defined
 - Perl built-in, [660](#)
- delete
 - Perl built-in, [664](#)
- destroy
 - Digest::MD5 function, [1475](#)
 - Digest::SHA function, [1499](#)
 - File::Temp function, [1562](#)
- Devel::Peek
 - Perl module, [1465](#)
- Devel::Peek::DeadCode
 - Perl function, [1469](#)
- Devel::Peek::Dump
 - Perl function, [1468](#)
- Devel::Peek::DumpArray
 - Perl function, [1468](#)
- Devel::Peek::DumpProg
 - Perl function, [1469](#)
- Devel::Peek::mstat
 - Perl function, [1469](#)
- Devel::Peek::SvREFCNT
 - Perl function, [1468](#)
- die
 - Perl built-in, [667](#)
- die, single: or die idiom, single: operator precedence or vs ||, single: \$@, [429](#)
- digest
 - Digest::MD5 function, [1481](#)
 - Digest::SHA function, [1507](#)
- Digest::MD5
 - Perl module, [1469](#)
- Digest::MD5::add
 - Perl function, [1478](#)
- Digest::MD5::add_bits
 - Perl function, [1477](#)
- Digest::MD5::addfile
 - Perl function, [1480](#)
- Digest::MD5::b64digest
 - Perl function, [1483](#)
- Digest::MD5::clone
 - Perl function, [1474](#)
- Digest::MD5::context
 - Perl function, [1484](#)
- Digest::MD5::destroy
 - Perl function, [1475](#)
- Digest::MD5::digest
 - Perl function, [1481](#)
- Digest::MD5::hexdigest
 - Perl function, [1482](#)
- Digest::MD5::md5_b64

- Perl function, [1488](#)
- Digest::MD5::md5_bin
 - Perl function, [1486](#)
- Digest::MD5::md5_hex
 - Perl function, [1487](#)
- Digest::MD5::new
 - Perl function, [1472](#)
- Digest::MD5::reset
 - Perl function, [1476](#)
- Digest::SHA
 - Perl module, [1490](#)
- Digest::SHA::add
 - Perl function, [1501](#)
- Digest::SHA::add_bits
 - Perl function, [1502](#)
- Digest::SHA::addfile
 - Perl function, [1506](#)
- Digest::SHA::addfilebin
 - Perl function, [1503](#)
- Digest::SHA::addfileuniv
 - Perl function, [1504](#)
- Digest::SHA::algorithm
 - Perl function, [1510](#)
- Digest::SHA::b64digest
 - Perl function, [1509](#)
- Digest::SHA::clone
 - Perl function, [1498](#)
- Digest::SHA::destroy
 - Perl function, [1499](#)
- Digest::SHA::digest
 - Perl function, [1507](#)
- Digest::SHA::dump
 - Perl function, [1517](#)
- Digest::SHA::getstate
 - Perl function, [1512](#)
- Digest::SHA::getstate_text
 - Perl function, [1514](#)
- Digest::SHA::hashsize
 - Perl function, [1511](#)
- Digest::SHA::hexdigest
 - Perl function, [1508](#)
- Digest::SHA::hmac_functional
 - Perl function, [1494](#)
- Digest::SHA::load
 - Perl function, [1518](#)
- Digest::SHA::new
 - Perl function, [1496](#)
- Digest::SHA::newSHA
 - Perl function, [1495](#)
- Digest::SHA::putstate
 - Perl function, [1513](#)
- Digest::SHA::putstate_text
 - Perl function, [1516](#)
- Digest::SHA::reset
 - Perl function, [1500](#)
- Digest::SHA::sha_functional
 - Perl function, [1492](#)
- dir_destroy
 - File::Temp function, [1575](#)
- dir_dirname
 - File::Temp function, [1573](#)
- dir_stringify
 - File::Temp function, [1573](#)
- dir_unlink_on_destroy
 - File::Temp function, [1574](#)
- do
 - Perl built-in, [672](#)
- Dump
 - Devel::Peek function, [1468](#)
- dump
 - Digest::SHA function, [1517](#)
 - Perl built-in, [677](#)
- DumpArray
 - Devel::Peek function, [1468](#)
- DumpProg
 - Devel::Peek function, [1469](#)
- dumpxs
 - Data::Dumper function, [1463](#)
- dup2
 - POSIX function, [1713](#)
- E
- each
 - Perl built-in, [680](#)
- Encode
 - Perl module, [1519](#)
- Encode::native_decode
 - Perl function, [1524](#)
- Encode::native_decode_utf8
 - Perl function, [1527](#)
- Encode::native_encode
 - Perl function, [1523](#)
- Encode::native_encode_utf8
 - Perl function, [1526](#)
- Encode::native_encodings
 - Perl function, [1530](#)
- Encode::native_find_encoding
 - Perl function, [1528](#)
- Encode::native_from_to
 - Perl function, [1530](#)
- Encode::native_is_utf8
 - Perl function, [1529](#)

Encode::native_obj_decode
 Perl function, [1531](#)
Encode::native_obj_encode
 Perl function, [1531](#)
Encode::native_utf8_on
 Perl function, [1529](#)
encode_base64
 MIME::Base64 function, [1630](#)
encode_base64url
 MIME::Base64 function, [1633](#)
encode_qp
 MIME::QuotedPrint function, [1637](#)
encoded_base64_length
 MIME::Base64 function, [1635](#)
endgrent
 Perl built-in, [683](#)
endhostent
 Perl built-in, [686](#)
endnetent
 Perl built-in, [688](#)
endprotoent
 Perl built-in, [690](#)
endpwent
 Perl built-in, [692](#)
endservent
 Perl built-in, [694](#)
eof
 Perl built-in, [696](#)
Errno
 Perl module, [1531](#)
error handling
 checklist, [431](#)
error handling, single: I/O
 errors, single: \$!, [429](#)
eval
 exception catching, single:
 die; trailing newline, single: exception objects, [431](#)
 Perl built-in, [699](#)
evalbytes
 Perl built-in, [705](#)
exec
 Perl built-in, [709](#)
exists
 Config function, [1448](#)
 Perl built-in, [712](#)
exit
 Perl built-in, [716](#)
exp
 Perl built-in, [719](#)

F
fast_abs_path
 Cwd function, [1459](#)
fc
 Perl built-in, [721](#)
Fcntl
 Perl module, [1532](#)
fcntl
 Perl built-in, [724](#)
Fcntl::s_ifmt
 Perl function, [1535](#)
Fcntl::s_imode
 Perl function, [1535](#)
Fcntl::s_isblk
 Perl function, [1536](#)
Fcntl::s_ischr
 Perl function, [1536](#)
Fcntl::s_isdir
 Perl function, [1536](#)
Fcntl::s_isfifo
 Perl function, [1536](#)
Fcntl::s_islnk
 Perl function, [1537](#)
Fcntl::s_isreg
 Perl function, [1535](#)
Fcntl::s_issock
 Perl function, [1537](#)
fd_retrieve
 Storable function, [1797](#)
fesetround
 POSIX function, [1707](#)
fetch
 Config function, [1448](#)
fetch_attrs
 attributes function, [1819](#)
ffi_call
 Peta::FFI function, [1762](#)
ffi_dlclose
 Peta::FFI function, [1763](#)
ffi_dlopen
 Peta::FFI function, [1761](#)
ffi_scan
 Peta::FFI function, [1764](#)
fibonacci
 Math::GMP function, [1651](#)
 PDL::Primitive function, [1676](#)
field
 Perl built-in, [727](#)
File::Glob
 Perl module, [1537](#)
File::Glob::bsd_glob

Perl function, [1538](#)
File::Glob::csh_glob
Perl function, [1540](#)
File::Glob::glob_error
Perl function, [1541](#)
File::Map
Perl module, [1542](#)
File::Map::advise
Perl function, [1550](#)
File::Map::map_anonymous
Perl function, [1546](#)
File::Map::map_file
Perl function, [1544](#)
File::Map::map_handle
Perl function, [1545](#)
File::Map::protect
Perl function, [1551](#)
File::Map::remap
Perl function, [1549](#)
File::Map::sync
Perl function, [1548](#)
File::Map::sys_map
Perl function, [1547](#)
File::Map::unmap
Perl function, [1549](#)
File::Temp
Perl module, [1552](#)
File::Temp::cleanup
Perl function, [1569](#)
File::Temp::cmpstat
Perl function, [1572](#)
File::Temp::destroy
Perl function, [1562](#)
File::Temp::dir_destroy
Perl function, [1575](#)
File::Temp::dir_dirname
Perl function, [1573](#)
File::Temp::dir_stringify
Perl function, [1573](#)
File::Temp::dir_unlink_on_destroy
Perl function, [1574](#)
File::Temp::filename
Perl function, [1560](#)
File::Temp::mkdtemp
Perl function, [1568](#)
File::Temp::mkstemp
Perl function, [1566](#)
File::Temp::mkstemp
Perl function, [1567](#)
File::Temp::mktemp
Perl function, [1565](#)
File::Temp::new
Perl function, [1558](#)
File::Temp::newdir
Perl function, [1559](#)
File::Temp::numify
Perl function, [1561](#)
File::Temp::safe_level
Perl function, [1571](#)
File::Temp::stringify
Perl function, [1560](#)
File::Temp::tempdir
Perl function, [1557](#)
File::Temp::tempfile
Perl function, [1555](#)
File::Temp::tempnam
Perl function, [1565](#)
File::Temp::tmpfile
Perl function, [1564](#)
File::Temp::tmpnam
Perl function, [1563](#)
File::Temp::top_system_uid
Perl function, [1572](#)
File::Temp::unlink0
Perl function, [1569](#)
File::Temp::unlink1
Perl function, [1570](#)
File::Temp::unlink_on_destroy
Perl function, [1561](#)
file_magic
Storable function, [1798](#)
filename
File::Temp function, [1560](#)
fileno
Perl built-in, [733](#)
Filter::Util::Call
Perl module, [1576](#)
Filter::Util::Call::filter_add
Perl function, [1582](#)
Filter::Util::Call::filter_del
Perl function, [1580](#)
Filter::Util::Call::filter_read
Perl function, [1578](#)
Filter::Util::Call::filter_read_exact
Perl function, [1579](#)
Filter::Util::Call::real_import
Perl function, [1581](#)
filter_add
Filter::Util::Call function, [1582](#)
filter_del

Filter::Util::Call function,
1580
filter_read
Filter::Util::Call function,
1578
filter_read_exact
Filter::Util::Call function,
1579
finally
Perl built-in, 736
first
List::Util function, 1622
firstkey
Config function, 1450
flock
Perl built-in, 741
flock, single: sysopen, single:
opendir, single: readdir,
single: O_EXCL, single:
O_NONBLOCK, 420
flowing
PDL::Core function, 1668
fma
POSIX function, 1705
fork
Perl built-in, 744
format
Perl built-in, 748
formline
Perl built-in, 752
fpathconf
POSIX function, 1720
fpclassify
POSIX function, 1707
freeze
Storable function, 1787
frexp
POSIX function, 1704
G
get
attributes function, 1819
get_isarev
mro function, 1825
get_linear_isa
mro function, 1825
get_pkg_gen
mro function, 1826
get_str_gmp
Math::GMP function, 1647
getaddrinfo
Socket function, 1782
getc
Perl built-in, 756
getgrent
Perl built-in, 758
getgrgid
Perl built-in, 761
getgrnam
Perl built-in, 764
gethostbyaddr
Perl built-in, 767
gethostbyname
Perl built-in, 770
gethostent
Perl built-in, 773
getlines
IO function, 1599
getlines_common
IO function, 1599
getlogin
Perl built-in, 776
getnameinfo
Socket function, 1783
getnetbyaddr
Perl built-in, 778
getnetbyname
Perl built-in, 780
getnetent
Perl built-in, 783
getpeername
Perl built-in, 785
getpgrp
Perl built-in, 788
getppid
Perl built-in, 791
getpriority
Perl built-in, 793
getprotobyname
Perl built-in, 795
getprotobynumber
Perl built-in, 798
getprotoent
Perl built-in, 800
getpwent
Perl built-in, 803
getpwnam
Perl built-in, 806
getpwuid
Perl built-in, 809
getservbyname
Perl built-in, 813

getservbyport
 Perl built-in, 815
 getservent
 Perl built-in, 819
 getsockname
 Perl built-in, 821
 getsockopt
 Perl built-in, 824
 getstate
 Digest::SHA function, 1512
 getstate_text
 Digest::SHA function, 1514
 glob
 Perl built-in, 826
 glob_error
 File::Glob function, 1541
 gmp_copy
 Math::GMP function, 1652
 gmp_lib_version
 Math::GMP function, 1646
 gmp_tstbit
 Math::GMP function, 1652
 gmtime
 Perl built-in, 830
 gnu_new
 Term::ReadLine::Gnu function, 1813
 goto
 Perl built-in, 833
 grep
 Perl built-in, 837
 guess_stash
 attributes function, 1821
 gv_name
 B function, 1440
 gv_safename
 B function, 1440
 gv_sv
 B function, 1440
 H
 hash
 B function, 1438
 Hash::Util
 Perl module, 1584
 Hash::Util::all_keys
 Perl function, 1595
 Hash::Util::bucket_array
 Perl function, 1591
 Hash::Util::bucket_info
 Perl function, 1591
 Hash::Util::bucket_ratio
 Perl function, 1591
 Hash::Util::bucket_stats
 Perl function, 1592
 Hash::Util::bucket_stats_formatted
 Perl function, 1593
 Hash::Util::hash_seed
 Perl function, 1590
 Hash::Util::hash_traversal_mask
 Perl function, 1592
 Hash::Util::hash_value
 Perl function, 1590
 Hash::Util::hidden_keys
 Perl function, 1595
 Hash::Util::hv_store
 Perl function, 1596
 Hash::Util::lock_hash
 Perl function, 1594
 Hash::Util::lock_hash_recurse
 Perl function, 1595
 Hash::Util::lock_keys
 Perl function, 1593
 Hash::Util::lock_ref_keys
 Perl function, 1594
 Hash::Util::lock_ref_keys_plus
 Perl function, 1594
 Hash::Util::lock_value
 Perl function, 1594
 Hash::Util::unlock_keys
 Perl function, 1593
 hash_seed
 Hash::Util function, 1590
 hash_traversal_mask
 Hash::Util function, 1592
 hash_value
 Hash::Util function, 1590
 hashsize
 Digest::SHA function, 1511
 head
 List::Util function, 1606
 header_files
 Config function, 1453
 hex
 Perl built-in, 840
 hexdigest
 Digest::MD5 function, 1482
 Digest::SHA function, 1508
 hidden_keys
 Hash::Util function, 1595
 hmac_functional

- Digest::SHA function, [1494](#)
- howbig
 - PDL::Core function, [1668](#)
- hv_name
 - B function, [1440](#)
- hv_store
 - Hash::Util function, [1596](#)
- I
- I18N::Langinfo
 - Perl module, [1596](#)
- import
 - Perl built-in, [842](#)
- indadd
 - PDL::Primitive function, [1676](#)
- index
 - Perl built-in, [846](#)
- inet_aton
 - Socket function, [1771](#)
- inet_ntoa
 - Socket function, [1772](#)
- inet_ntop
 - Socket function, [1773](#)
- inet_pton
 - Socket function, [1772](#)
- in-memory filehandle
 - layers, single: PerlIO layers; in-memory, [428](#)
 - limitations, single: seek; on in-memory handle, single: fork, [429](#)
 - reading, single: test fixtures, [428](#)
- in-memory filehandle, single:
 - open
 - scalar reference, pair: file-handle; scalar, [427](#)
- inner
 - PDL::Primitive function, [1674](#)
- install
 - re function, [1830](#)
- int
 - Perl built-in, [848](#)
- intify
 - Math::GMP function, [1647](#)
- invalidate_all_method_caches
 - mro function, [1827](#)
- IO
 - Perl module, [1596](#)
- IO::getlines
 - Perl function, [1599](#)
- IO::getlines_common
 - Perl function, [1599](#)
- ioctl
 - Perl built-in, [851](#)
- is_alpha
 - version function, [1844](#)
- is_lax
 - version function, [1845](#)
- is_perfect_power
 - Math::GMP function, [1653](#)
- is_qv
 - version function, [1844](#)
- is_regexp
 - re function, [1833](#)
- is_strict
 - version function, [1846](#)
- is_universal
 - mro function, [1826](#)
- isa
 - Perl built-in, [855](#)
- J
- JIT, [1859](#)
- jn
 - POSIX function, [1706](#)
- join
 - Perl built-in, [858](#)
- K
- keys
 - Perl built-in, [860](#)
- kill
 - Perl built-in, [863](#)
- L
- last
 - Perl built-in, [866](#)
- lc
 - Perl built-in, [870](#)
- lcfirst
 - Perl built-in, [873](#)
- legendre
 - Math::GMP function, [1650](#)
- length
 - Perl built-in, [876](#)
- link
 - Perl built-in, [879](#)
- list operator, [1859](#)
- List::Util
 - Perl module, [1599](#)
- List::Util::all
 - Perl function, [1624](#)

List::Util::any
Perl function, 1623

List::Util::first
Perl function, 1622

List::Util::head
Perl function, 1606

List::Util::max
Perl function, 1605

List::Util::maxstr
Perl function, 1612

List::Util::mesh
Perl function, 1618

List::Util::mesh_shortest
Perl function, 1619

List::Util::min
Perl function, 1604

List::Util::minstr
Perl function, 1611

List::Util::none
Perl function, 1625

List::Util::notall
Perl function, 1625

List::Util::pairfirst
Perl function, 1617

List::Util::pairgrep
Perl function, 1616

List::Util::pairkeys
Perl function, 1614

List::Util::pairmap
Perl function, 1616

List::Util::pairs
Perl function, 1613

List::Util::pairvalues
Perl function, 1615

List::Util::product
Perl function, 1606

List::Util::reduce
Perl function, 1621

List::Util::reductions
Perl function, 1622

List::Util::sample
Perl function, 1609

List::Util::shuffle
Perl function, 1608

List::Util::sub_util_set_prototype
Perl function, 1628

List::Util::sub_util_set_subname
Perl function, 1627

List::Util::sub_util_subname
Perl function, 1626

List::Util::sum
Perl function, 1603

List::Util::sum0
Perl function, 1604

List::Util::tail
Perl function, 1607

List::Util::uniq
Perl function, 1609

List::Util::uniqint
Perl function, 1611

List::Util::uniquum
Perl function, 1610

List::Util::unpairs
Perl function, 1614

List::Util::zip
Perl function, 1619

List::Util::zip_shortest
Perl function, 1620

listen
Perl built-in, 881

load
Digest::SHA function, 1518

local
Perl built-in, 883

local_patches
Config function, 1452

localeconv
POSIX function, 1719

localtime
Perl built-in, 888

lock
Perl built-in, 891

lock_hash
Hash::Util function, 1594

lock_hash_recurse
Hash::Util function, 1595

lock_keys
Hash::Util function, 1593

lock_ref_keys
Hash::Util function, 1594

lock_ref_keys_plus
Hash::Util function, 1594

lock_value
Hash::Util function, 1594

log
Perl built-in, 894

lseek
POSIX function, 1714

lstat
Perl built-in, 897

M`m//``Perl built-in, 900``main_root``B function, 1437``map``Perl built-in, 904``map_anonymous``File::Map function, 1546``map_file``File::Map function, 1544``map_handle``File::Map function, 1545``Math::GMP``Perl module, 1639``Math::GMP::add_ui_gmp``Perl function, 1648``Math::GMP::bdiv``Perl function, 1649``Math::GMP::bfac``Perl function, 1651``Math::GMP::bgcd``Perl function, 1650``Math::GMP::blcm``Perl function, 1651``Math::GMP::blshift``Perl function, 1651``Math::GMP::bmodinv``Perl function, 1649``Math::GMP::bmulf``Perl function, 1649``Math::GMP::bnok``Perl function, 1652``Math::GMP::broot``Perl function, 1652``Math::GMP::brootrem``Perl function, 1653``Math::GMP::bsqrt``Perl function, 1653``Math::GMP::fibonacci``Perl function, 1651``Math::GMP::get_str_gmp``Perl function, 1647``Math::GMP::gmp_copy``Perl function, 1652``Math::GMP::gmp_lib_version``Perl function, 1646``Math::GMP::gmp_tstbit``Perl function, 1652``Math::GMP::intify``Perl function, 1647``Math::GMP::is_perfect_power`
`Perl function, 1653``Math::GMP::legendre`
`Perl function, 1650``Math::GMP::mmod_gmp`
`Perl function, 1648``Math::GMP::new`
`Perl function, 1646``Math::GMP::op_numify`
`Perl function, 1649``Math::GMP::powm_gmp`
`Perl function, 1648``Math::GMP::probab_prime`
`Perl function, 1650``Math::GMP::sizeinbase_gmp`
`Perl function, 1647``max``List::Util function, 1605``maxstr``List::Util function, 1612``maybe_next_method``mro function, 1828``md5_b64``Digest::MD5 function, 1488``md5_bin``Digest::MD5 function, 1486``md5_hex``Digest::MD5 function, 1487``mesh``List::Util function, 1618``mesh_shortest``List::Util function, 1619``method``Perl built-in, 907``method_changed_in``mro function, 1826``MIME::Base64``Perl module, 1629``MIME::Base64::decode_base64`
`Perl function, 1631``MIME::Base64::decode_base64url`
`Perl function, 1634``MIME::Base64::de-`
`coded_base64_length`
`Perl function, 1636``MIME::Base64::encode_base64`
`Perl function, 1630``MIME::Base64::encode_base64url`
`Perl function, 1633``MIME::Base64::en-`
`coded_base64_length`

- Perl function, 1635
- MIME::QuotedPrint
 - Perl module, 1637
- MIME::QuotedPrint::decode_qp
 - Perl function, 1638
- MIME::QuotedPrint::encode_qp
 - Perl function, 1637
- min
 - List::Util function, 1604
- minstr
 - List::Util function, 1611
- mkdir
 - Perl built-in, 911
- mkdtemp
 - File::Temp function, 1568
- mkstemp
 - File::Temp function, 1566
- mkstemps
 - File::Temp function, 1567
- mktemp
 - File::Temp function, 1565
- mktime
 - POSIX function, 1716
- mmod_gmp
 - Math::GMP function, 1648
- modf
 - POSIX function, 1704
- modify_attrs
 - attributes function, 1817
- mro
 - Perl module, 1823
- mro::get_isarev
 - Perl function, 1825
- mro::get_linear_isa
 - Perl function, 1825
- mro::get_pkg_gen
 - Perl function, 1826
- mro::invalidate_all_method_caches
 - Perl function, 1827
- mro::is_universal
 - Perl function, 1826
- mro::maybe_next_method
 - Perl function, 1828
- mro::method_changed_in
 - Perl function, 1826
- mro::next_can
 - Perl function, 1828
- mro::next_method
 - Perl function, 1828
- mro::nextcan
 - Perl function, 1827
- mro::set_mro
 - Perl function, 1825
- msgctl
 - Perl built-in, 914
- msgget
 - Perl built-in, 916
- msgrcv
 - Perl built-in, 919
- msgsnd
 - Perl built-in, 922
- mstat
 - Devel::Peek function, 1469
- my
 - Perl built-in, 925
- myconfig
 - Config function, 1446
- N
- named unary operator, 1859
- nan
 - POSIX function, 1705
- native_decode
 - Encode function, 1524
- native_decode_utf8
 - Encode function, 1527
- native_encode
 - Encode function, 1523
- native_encode_utf8
 - Encode function, 1526
- native_encodings
 - Encode function, 1530
- native_find_encoding
 - Encode function, 1528
- native_from_to
 - Encode function, 1530
- native_is_utf8
 - Encode function, 1529
- native_obj_decode
 - Encode function, 1531
- native_obj_encode
 - Encode function, 1531
- native_utf8_on
 - Encode function, 1529
- new
 - Digest::MD5 function, 1472
 - Digest::SHA function, 1496
 - File::Temp function, 1558
 - Math::GMP function, 1646
 - version function, 1839
- newdir
 - File::Temp function, 1559

newSHA
 Digest::SHA function, [1495](#)
next
 Perl built-in, [930](#)
next_can
 mro function, [1828](#)
next_method
 mro function, [1828](#)
nextcan
 mro function, [1827](#)
nextkey
 Config function, [1450](#)
nfreeze
 Storable function, [1788](#)
nice
 POSIX function, [1718](#)
no
 Perl built-in, [933](#)
nomethod
 version function, [1848](#)
non_bincompat_options
 Config function, [1453](#)
none
 List::Util function, [1625](#)
normal
 version function, [1843](#)
notall
 List::Util function, [1625](#)
nstore
 Storable function, [1793](#)
nstore_fd
 Storable function, [1796](#)
numify
 File::Temp function, [1561](#)
 version function, [1842](#)

O
object_2svref
 B function, [1439](#)
oct
 Perl built-in, [936](#)
op_numify
 Math::GMP function, [1649](#)
op_ppaddr
 B function, [1441](#)
open
 3-arg form, single: lexical
 filehandle, [419](#)
 3-arg form, single: open;
 2-arg form, pair: open;
 security, [420](#)
 modes, single: read mode, sin-
 gle: write mode, single:
 append mode, [420](#)
 Perl built-in, [939](#)
 POSIX function, [1713](#)
 open pragma, single: use open,
 single: PerlIO layers
 :std, [423](#)
 open, single: filehandle, single:
 I/O
 opening files, [419](#)
opendir
 Perl built-in, [942](#)
opnumber
 B function, [1436](#)
optimization
 re function, [1832](#)
ord
 Perl built-in, [945](#)
our
 Perl built-in, [948](#)

P
pack
 Perl built-in, [953](#)
pack_sockaddr_in
 Socket function, [1774](#)
pack_sockaddr_in6
 Socket function, [1776](#)
pack_sockaddr_un
 Socket function, [1778](#)
package
 Perl built-in, [960](#)
PadWalker
 Perl module, [1725](#)
PadWalker::closed_over
 Perl function, [1730](#)
PadWalker::peek_my
 Perl function, [1726](#)
PadWalker::peek_our
 Perl function, [1727](#)
PadWalker::peek_sub
 Perl function, [1728](#)
PadWalker::set_closed_over
 Perl function, [1731](#)
PadWalker::var_name
 Perl function, [1732](#)
pairfirst
 List::Util function, [1617](#)
pairgrep
 List::Util function, [1616](#)

- pairkeys
 - List::Util function, [1614](#)
- pairmap
 - List::Util function, [1616](#)
- pairs
 - List::Util function, [1613](#)
- pairvalues
 - List::Util function, [1615](#)
- PDL
 - Perl module, [1654](#)
- PDL::Bad
 - Perl module, [1655](#)
- PDL::Basic
 - Perl module, [1656](#)
- PDL::Basic::sequence
 - Perl function, [1658](#)
- PDL::Constants
 - Perl module, [1659](#)
- PDL::Core
 - Perl module, [1659](#)
- PDL::Core::at
 - Perl function, [1666](#)
- PDL::Core::flowing
 - Perl function, [1668](#)
- PDL::Core::howbig
 - Perl function, [1668](#)
- PDL::Core::set_sv_to_null_pdl
 - Perl function, [1668](#)
- PDL::FFT
 - Perl module, [1668](#)
- PDL::Lite
 - Perl module, [1669](#)
- PDL::LiteF
 - Perl module, [1669](#)
- PDL::Math
 - Perl module, [1669](#)
- PDL::MatrixOps
 - Perl module, [1670](#)
- PDL::Ops
 - Perl module, [1672](#)
- PDL::Ops::bool_overload
 - Perl function, [1673](#)
- PDL::Primitive
 - Perl module, [1673](#)
- PDL::Primitive::fibonacci
 - Perl function, [1676](#)
- PDL::Primitive::indadd
 - Perl function, [1676](#)
- PDL::Primitive::inner
 - Perl function, [1674](#)
- PDL::Primitive::uniqind
 - Perl function, [1676](#)
- PDL::Slices
 - Perl module, [1676](#)
- PDL::Slices::slice
 - Perl function, [1679](#)
- PDL::Types
 - Perl module, [1681](#)
- PDL::Ufunc
 - Perl module, [1683](#)
- PDL::Ufunc::sum
 - Perl function, [1685](#)
- peek_my
 - PadWalker function, [1726](#)
- peek_our
 - PadWalker function, [1727](#)
- peek_sub
 - PadWalker function, [1728](#)
- PerlIO layers
 - :encoding, single: PerlIO layers; :utf8, single: PerlIO layers; :raw, single: PerlIO layers; :crlf, single: Encode module, single: new-line translation, [423](#)
- PerlIO layers, single: encoding, single: layered I/O, pair: encoding
 - filehandle, [422](#)
- PerlIO::encoding
 - Perl module, [1733](#)
- PerlIO::mmap
 - Perl module, [1734](#)
- PerlIO::via
 - Perl module, [1734](#)
- Peta::FFI
 - Perl module, [1735](#)
- Peta::FFI::Cairo
 - Perl module, [1737](#)
- Peta::FFI::Cairo::Context
 - Perl module, [1739](#)
- Peta::FFI::Cairo::Font
 - Perl module, [1743](#)
- Peta::FFI::Cairo::Matrix
 - Perl module, [1747](#)
- Peta::FFI::Cairo::Path
 - Perl module, [1749](#)
- Peta::FFI::Cairo::Pattern
 - Perl module, [1752](#)
- Peta::FFI::Cairo::Region
 - Perl module, [1754](#)
- Peta::FFI::Cairo::Surface

- Perl module, [1756](#)
- Peta::FFI::ffi_call
 - Perl function, [1762](#)
- Peta::FFI::ffi_dlclose
 - Perl function, [1763](#)
- Peta::FFI::ffi_dlopen
 - Perl function, [1761](#)
- Peta::FFI::ffi_scan
 - Perl function, [1764](#)
- Peta::FFI::Libc
 - Perl module, [1761](#)
- Peta::FFI::UUID
 - Perl module, [1761](#)
- PetaPerl, [1858](#)
- pipe
 - Perl built-in, [965](#)
 - POSIX function, [1714](#)
- pipe open
 - list form, single: pipe open; string form, single: shell injection, single: /bin/sh, [425](#)
 - read from command, single: -|, single: fork, single: exec, single: glob, [425](#)
 - write to command, single: |-, single: close; pipe, [425](#)
- pipe open, single: open
 - to process, pair: pipe; child process, single: \$?, [424](#)
- pop
 - Perl built-in, [969](#)
- pos
 - Perl built-in, [971](#)
- POSIX
 - Perl module, [1686](#)
- POSIX::abs
 - Perl function, [1703](#)
- POSIX::access
 - Perl function, [1712](#)
- POSIX::asctime
 - Perl function, [1717](#)
- POSIX::constant
 - Perl function, [1720](#)
- POSIX::ctime
 - Perl function, [1718](#)
- POSIX::dup2
 - Perl function, [1713](#)
- POSIX::fesetround
 - Perl function, [1707](#)
- POSIX::fma
 - Perl function, [1705](#)
- POSIX::fpathconf
 - Perl function, [1720](#)
- POSIX::fpclassify
 - Perl function, [1707](#)
- POSIX::frexp
 - Perl function, [1704](#)
- POSIX::jn
 - Perl function, [1706](#)
- POSIX::localeconv
 - Perl function, [1719](#)
- POSIX::lseek
 - Perl function, [1714](#)
- POSIX::mktime
 - Perl function, [1716](#)
- POSIX::modf
 - Perl function, [1704](#)
- POSIX::nan
 - Perl function, [1705](#)
- POSIX::nice
 - Perl function, [1718](#)
- POSIX::open
 - Perl function, [1713](#)
- POSIX::pipe
 - Perl function, [1714](#)
- POSIX::read
 - Perl function, [1714](#)
- POSIX::remquo
 - Perl function, [1706](#)
- POSIX::setlocale
 - Perl function, [1719](#)
- POSIX::setpgid
 - Perl function, [1711](#)
- POSIX::sigaction
 - Perl function, [1724](#)
- POSIX::sigprocmask
 - Perl function, [1724](#)
- POSIX::sigset_new
 - Perl function, [1723](#)
- POSIX::sprintf
 - Perl function, [1710](#)
- POSIX::strerror
 - Perl function, [1710](#)
- POSIX::strftime
 - Perl function, [1715](#)
- POSIX::strtod
 - Perl function, [1708](#)
- POSIX::strtol
 - Perl function, [1709](#)
- POSIX::strtoul
 - Perl function, [1709](#)

POSIX::sysconf
 Perl function, [1712](#)
 POSIX::tcflow
 Perl function, [1721](#)
 POSIX::tcflush
 Perl function, [1721](#)
 POSIX::termios_getattr
 Perl function, [1722](#)
 POSIX::termios_getcc
 Perl function, [1723](#)
 POSIX::termios_new
 Perl function, [1722](#)
 POSIX::termios_setattr
 Perl function, [1722](#)
 POSIX::times
 Perl function, [1717](#)
 POSIX::tzname
 Perl function, [1716](#)
 POSIX::uname
 Perl function, [1711](#)
 POSIX::wexitstatus
 Perl function, [1708](#)
 POSIX::write
 Perl function, [1715](#)
 POSIX::yn
 Perl function, [1706](#)
 powm_gmp
 Math::GMP function, [1648](#)
 pperl, [1858](#), [1859](#)
 print
 Perl built-in, [973](#)
 to buffer, single: printf,
 single: in-memory filehan-
 dle; writing, [427](#)
 to filehandle, single: trun-
 cate; on open, single:
 buffered I/O, [421](#)
 printf
 Perl built-in, [976](#)
 probab_prime
 Math::GMP function, [1650](#)
 product
 List::Util function, [1606](#)
 protect
 File::Map function, [1551](#)
 prototype
 Perl built-in, [979](#)
 push
 Perl built-in, [982](#)
 putstate
 Digest::SHA function, [1513](#)
 putstate_text
 Digest::SHA function, [1516](#)
 pv_pv
 B function, [1439](#)
Q
 q//
 Perl built-in, [985](#)
 qq//
 Perl built-in, [988](#)
 qr//
 Perl built-in, [993](#)
 quotemeta
 Perl built-in, [997](#)
 qw//
 Perl built-in, [999](#)
 qx//
 Perl built-in, [1002](#)
R
 rand
 Perl built-in, [1005](#)
 re
 Perl module, [1828](#)
 re::install
 Perl function, [1830](#)
 re::is_regexp
 Perl function, [1833](#)
 re::optimization
 Perl function, [1832](#)
 re::regexp_pattern
 Perl function, [1833](#)
 re::regmust
 Perl function, [1831](#)
 re::regname
 Perl function, [1834](#)
 re::regnames
 Perl function, [1835](#)
 re::regnames_count
 Perl function, [1836](#)
 read
 Perl built-in, [1007](#)
 POSIX function, [1714](#)
 read_history_range
 Term::ReadLine::Gnu function,
 [1814](#)
 read_magic
 Storable function, [1799](#)
 readdir
 Perl built-in, [1012](#)
 readline
 Perl built-in, [1014](#)

readline, single: diamond operator, single: chomp, single: close, single: \$/, 420

readlink
Perl built-in, 1018

readpipe
Perl built-in, 1021

read-write mode, single: sysopen, single: sysread, single: syswrite, single: atomic rename, 422

real_import
Filter::Util::Call function, 1581

recv
Perl built-in, 1023

redo
Perl built-in, 1026

reduce
List::Util function, 1621

reductions
List::Util function, 1622

ref
Perl built-in, 1029

reftype
attributes function, 1822

regex_pattern
re function, 1833

regmust
re function, 1831

regname
re function, 1834

regnames
re function, 1835

regnames_count
re function, 1836

remap
File::Map function, 1549

remquo
POSIX function, 1706

rename
Perl built-in, 1033

require
Perl built-in, 1035

reset
Digest::MD5 function, 1476
Digest::SHA function, 1500
Perl built-in, 1041

retrieve
Storable function, 1794

return
Perl built-in, 1044

reverse
Perl built-in, 1048

rewinddir
Perl built-in, 1051

rindex
Perl built-in, 1053

rl_parse_and_bind
Term::ReadLine::Gnu function, 1814

rl_readline
Term::ReadLine::Gnu function, 1813

rl_store_function
Term::ReadLine::Gnu function, 1814

rmdir
Perl built-in, 1056

S

s///
Perl built-in, 1058

s_ifmt
Fcntl function, 1535

s_imode
Fcntl function, 1535

s_isblk
Fcntl function, 1536

s_ischr
Fcntl function, 1536

s_isdir
Fcntl function, 1536

s_isfifo
Fcntl function, 1536

s_islnk
Fcntl function, 1537

s_isreg
Fcntl function, 1535

s_issock
Fcntl function, 1537

safe_level
File::Temp function, 1571

sample
List::Util function, 1609

say
Perl built-in, 1062

scalar
Perl built-in, 1065

scalar reference
as filehandle target, 427

Scalar::Util

- Perl module, [1768](#)
- SDBM_File
 - Perl module, [1764](#)
- SDBM_File::store
 - Perl function, [1767](#)
- SDBM_File::tiehash
 - Perl function, [1767](#)
- seek
 - Perl built-in, [1068](#)
- seekdir
 - Perl built-in, [1071](#)
- select
 - Perl built-in, [1073](#)
- semctl
 - Perl built-in, [1076](#)
- semget
 - Perl built-in, [1079](#)
- semop
 - Perl built-in, [1082](#)
- send
 - Perl built-in, [1085](#)
- sequence
 - PDL::Basic function, [1658](#)
- set_closed_over
 - PadWalker function, [1731](#)
- set_mro
 - mro function, [1825](#)
- set_sv_to_null_pdl
 - PDL::Core function, [1668](#)
- setgrent
 - Perl built-in, [1087](#)
- sethostent
 - Perl built-in, [1090](#)
- setlocale
 - POSIX function, [1719](#)
- setnetent
 - Perl built-in, [1092](#)
- setpgid
 - POSIX function, [1711](#)
- setpgrp
 - Perl built-in, [1094](#)
- setpriority
 - Perl built-in, [1097](#)
- setprotoent
 - Perl built-in, [1099](#)
- setpwent
 - Perl built-in, [1101](#)
- setservent
 - Perl built-in, [1103](#)
- setsockopt
 - Perl built-in, [1106](#)
- sha_functional
 - Digest::SHA function, [1492](#)
- shift
 - Perl built-in, [1109](#)
- shmctl
 - Perl built-in, [1111](#)
- shmget
 - Perl built-in, [1113](#)
- shmread
 - Perl built-in, [1116](#)
- shmwrite
 - Perl built-in, [1119](#)
- shuffle
 - List::Util function, [1608](#)
- shutdown
 - Perl built-in, [1121](#)
- sigaction
 - POSIX function, [1724](#)
- sigprocmask
 - POSIX function, [1724](#)
- sigset_new
 - POSIX function, [1723](#)
- sin
 - Perl built-in, [1124](#)
- sizeinbase_gmp
 - Math::GMP function, [1647](#)
- sleep
 - Perl built-in, [1127](#)
- slice
 - PDL::Slices function, [1679](#)
- slurp mode, single: \$/
 - slurp, single: local, [421](#)
- sockaddr_family
 - Socket function, [1779](#)
- sockaddr_in
 - Socket function, [1780](#)
- sockaddr_in6
 - Socket function, [1781](#)
- Socket
 - Perl module, [1768](#)
- socket
 - Perl built-in, [1130](#)
- Socket::getaddrinfo
 - Perl function, [1782](#)
- Socket::getnameinfo
 - Perl function, [1783](#)
- Socket::inet_aton
 - Perl function, [1771](#)
- Socket::inet_ntoa
 - Perl function, [1772](#)
- Socket::inet_ntop

Perl function, [1773](#)
Socket::inet_pton
Perl function, [1772](#)
Socket::pack_sockaddr_in
Perl function, [1774](#)
Socket::pack_sockaddr_in6
Perl function, [1776](#)
Socket::pack_sockaddr_un
Perl function, [1778](#)
Socket::sockaddr_family
Perl function, [1779](#)
Socket::sockaddr_in
Perl function, [1780](#)
Socket::sockaddr_in6
Perl function, [1781](#)
Socket::unpack_sockaddr_in
Perl function, [1775](#)
Socket::unpack_sockaddr_in6
Perl function, [1777](#)
Socket::unpack_sockaddr_un
Perl function, [1779](#)
socketpair
Perl built-in, [1133](#)
sort
Perl built-in, [1136](#)
splice
Perl built-in, [1139](#)
split
Perl built-in, [1142](#)
sprintf
Perl built-in, [1147](#)
POSIX function, [1710](#)
sqrt
Perl built-in, [1152](#)
srand
Perl built-in, [1155](#)
stack_depth
Storable function, [1800](#)
stack_depth_hash
Storable function, [1801](#)
stat
Perl built-in, [1158](#)
state
Perl built-in, [1161](#)
STDIN, single: STDOUT, single:
STDERR, single: ARGV, sin-
gle: pre-opened handles,
[419](#)
Storable
Perl module, [1785](#)
Storable::dclone
Perl function, [1790](#)
Storable::fd_retrieve
Perl function, [1797](#)
Storable::file_magic
Perl function, [1798](#)
Storable::freeze
Perl function, [1787](#)
Storable::nfreeze
Perl function, [1788](#)
Storable::nstore
Perl function, [1793](#)
Storable::nstore_fd
Perl function, [1796](#)
Storable::read_magic
Perl function, [1799](#)
Storable::retrieve
Perl function, [1794](#)
Storable::stack_depth
Perl function, [1800](#)
Storable::stack_depth_hash
Perl function, [1801](#)
Storable::store
Perl function, [1792](#)
Storable::store_fd
Perl function, [1795](#)
Storable::thaw
Perl function, [1789](#)
store
Config function, [1449](#)
SDBM_File function, [1767](#)
Storable function, [1792](#)
store_fd
Storable function, [1795](#)
strerror
POSIX function, [1710](#)
strftime
POSIX function, [1715](#)
stringify
File::Temp function, [1560](#)
version function, [1841](#)
strtod
POSIX function, [1708](#)
strtol
POSIX function, [1709](#)
strtoul
POSIX function, [1709](#)
study
Perl built-in, [1165](#)
sub
Perl built-in, [1167](#)
sub_util_set_prototype

- List::Util function, 1628
- sub_util_set_subname
 - List::Util function, 1627
- sub_util_subname
 - List::Util function, 1626
- substr
 - Perl built-in, 1173
- sum
 - List::Util function, 1603
 - PDL::Ufunc function, 1685
- sum0
 - List::Util function, 1604
- SV, 1859
- sv_refcnt
 - B function, 1438
- sv_svtype
 - B function, 1438
- svref_2object
 - B function, 1436
- SvREFCNT
 - Devel::Peek function, 1468
- symlink
 - Perl built-in, 1177
- sync
 - File::Map function, 1548
- Sys::Hostname
 - Perl module, 1802
- sys_map
 - File::Map function, 1547
- syscall
 - Perl built-in, 1180
- sysconf
 - POSIX function, 1712
- sysopen
 - Perl built-in, 1183
- sysread
 - Perl built-in, 1186
- sysseek
 - Perl built-in, 1190
- system
 - Perl built-in, 1194
- syswrite
 - Perl built-in, 1197
- T
- tail
 - List::Util function, 1607
- tcflow
 - POSIX function, 1721
- tcflush
 - POSIX function, 1721
- tell
 - Perl built-in, 1200
- telldir
 - Perl built-in, 1202
- tempdir
 - File::Temp function, 1557
- tempfile
 - File::Temp function, 1555
- tempnam
 - File::Temp function, 1565
- Term::ReadLine::Gnu
 - Perl module, 1802
- Term::ReadLine::Gnu::add_history
 - Perl function, 1814
- Term::ReadLine::Gnu::gnu_new
 - Perl function, 1813
- Term::ReadLine::Gnu::read_his-
tory_range
 - Perl function, 1814
- Term::ReadLine::Gnu::rl_parse_and_bind
 - Perl function, 1814
- Term::ReadLine::Gnu::rl_readline
 - Perl function, 1813
- Term::ReadLine::Gnu::rl_store_func-
tion
 - Perl function, 1814
- termios_getattr
 - POSIX function, 1722
- termios_getcc
 - POSIX function, 1723
- termios_new
 - POSIX function, 1722
- termios_setattr
 - POSIX function, 1722
- thaw
 - Storable function, 1789
- tie
 - Perl built-in, 1205
- tied
 - Perl built-in, 1209
- tiehash
 - SDBM_File function, 1767
- time
 - Perl built-in, 1211
- Time::HiRes
 - Perl module, 1815
- times
 - Perl built-in, 1213
 - POSIX function, 1717
- tmpfile
 - File::Temp function, 1564

tmpnam
 File::Temp function, 1563
top_system_uid
 File::Temp function, 1572
tr///
 Perl built-in, 1216
truncate
 Perl built-in, 1220
try
 Perl built-in, 1223
tzname
 POSIX function, 1716

U

uc
 Perl built-in, 1228
ucfirst
 Perl built-in, 1231
umask
 Perl built-in, 1234
uname
 POSIX function, 1711
undef
 Perl built-in, 1237
uniq
 List::Util function, 1609
uniqind
 PDL::Primitive function, 1676
uniqint
 List::Util function, 1611
uniqnum
 List::Util function, 1610
unlink
 Perl built-in, 1241
unlink0
 File::Temp function, 1569
unlink1
 File::Temp function, 1570
unlink_on_destroy
 File::Temp function, 1561
unlock_keys
 Hash::Util function, 1593
unmap
 File::Map function, 1549
unpack
 Perl built-in, 1244
unpack_sockaddr_in
 Socket function, 1775
unpack_sockaddr_in6
 Socket function, 1777
unpack_sockaddr_un
 Socket function, 1779
unpairs
 List::Util function, 1614
unshift
 Perl built-in, 1250
untie
 Perl built-in, 1253
use
 Perl built-in, 1255
UTF-8, single: mojibake, single:
 characters vs bytes, 422
utime
 Perl built-in, 1260

V

values
 Perl built-in, 1263
var_name
 PadWalker function, 1732
vcmp
 version function, 1847
vec
 Perl built-in, 1266
version
 Perl module, 1837
version::bool
 Perl function, 1848
version::declare
 Perl function, 1840
version::is_alpha
 Perl function, 1844
version::is_lax
 Perl function, 1845
version::is_qv
 Perl function, 1844
version::is_strict
 Perl function, 1846
version::new
 Perl function, 1839
version::nomethod
 Perl function, 1848
version::normal
 Perl function, 1843
version::numify
 Perl function, 1842
version::stringify
 Perl function, 1841
version::vcmp
 Perl function, 1847

W

wait

- Perl built-in, [1269](#)
- waitpid
 - Perl built-in, [1271](#)
- walkoptree
 - B function, [1437](#)
- walkoptree_debug
 - B function, [1437](#)
- wantarray
 - Perl built-in, [1275](#)
- warn
 - Perl built-in, [1278](#)
- wexitstatus
 - POSIX function, [1708](#)
- write
 - Perl built-in, [1283](#)
 - POSIX function, [1715](#)

Y

- y///
 - Perl built-in, [1287](#)
- yn
 - POSIX function, [1706](#)

Z

- zip
 - List::Util function, [1619](#)
- zip_shortest
 - List::Util function, [1620](#)