PetaPerl documentation

***Experimental preview — draft documentation* 0.6.6**

PetaMem, s.r.o.

26 Ιουλίου 2026

Περιεχόμενα

Ένα περιβάλλον εκτέλεσης Perl 5 νέας γενιάς γραμμένο σε Rust, με JIT και αυτόματη παραλληλοποίηση. Η τεκμηρίωση αυτή καλύπτει την εγκατάσταση, τη χρήση, την αναφορά των ενσωματωμένων αρθρωμάτων και τα εσωτερικά του διερμηνευτή.

Αυτή είναι η τεκμηρίωση του **pperl 0.6.6**, παραχθείσα στις 2026-07-26. (μεταφρασμένα ≈ λέξεις, %)

Ξεκινώντας Πρώτα βήματα: εγκαταστήστε το pperl, εκτελέστε το πρώτο σας σενάριο, δείτε τι διαφέρει από το perl του συστήματος.

Ξεκινώντας Οδηγοί χρήσης Υλικό προσανατολισμένο σε εργασίες. Οι **οδηγοί** είναι εκτενείς και πολυκέφαλοι (regex, αντικειμενοστρέφεια, αποσφαλμάτωση, ταυτόχρονη εκτέλεση, αρχιτεκτονική του pperl). Τα **tutorials** ολοκληρώνονται σε μία συνεδρία (pack, ρεύματα I/O, αναφορές, Unicode).

Οδηγοί χρήσης Αναφορά Αναφορά ανά άρθρωμα για κάθε ενσωματωμένο άρθρωμα· αναφορά της γλώσσας Perl (μεταβλητές, συναρτήσεις, τελεστές, regex).

Αναφορά Ευρετήριο & γλωσσάρι Αλφαβητικό ευρετήριο κάθε τεκμηριωμένου αρθρώματος, συνάρτησης και όρου· σύντομο γλωσσάρι του λεξιλογίου του έργου.

genindex

ΚΕΦΑΛΑΙΟ 1

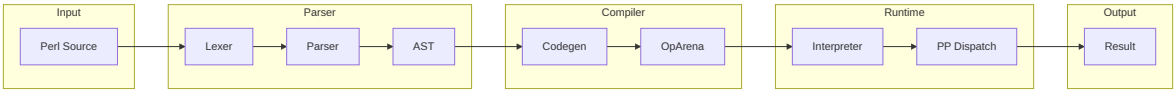
Εισαγωγή

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

1.1 Στόχοι

- **Αυτόματη παραλληλοποίηση** - Αυτόματη παραλληλοποίηση των `map`, `grep`, βρόχων `for`, `while` μέσω work-stealing του Rayon
- **Μεταγλώττιση JIT** - Παραγωγή εγγενούς κώδικα μέσω του Cranelift για θερμούς βρόχους (έως 76x ταχύτερη από το perl5)
- **Βιωσιμότητα Pure Perl** - Αρκετά γρήγορο ώστε το XS να γίνεται προαιρετικό

1.2 Επισκόπηση αρχιτεκτονικής



1.3 Γρήγορη εκκίνηση

```
# Run a script
pperl script.pl

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

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

1.4 Συμβατότητα

pperl aims for high compatibility with Perl 5.44+.

1.5 Υποστήριξη πλατφορμών

- **Μόνο Linux** (οποιαδήποτε αρχιτεκτονική)
- Όχι Windows, macOS ή άλλες πλατφόρμες

ΚΕΦΑΛΑΙΟ 2

Ξεκινώντας

2.1 Εγκατάσταση

Κατεβάστε εκτελέσιμα και βρείτε τεκμηρίωση στο perl.petamem.com.

2.2 Πρώτο σενάριο

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

2.3 Επιλογές γραμμής εντολών

Επιλογή	Περιγραφή
`-e 'code'`	Execute one-liner
`-E 'code'`	Like `-e`, with all optional features enabled (`say`, ...)
`-I dir`	Prepend `dir` to the module search path `@INC`
`-c`	Έλεγχος μόνο της σύνταξης
`-w`	Ενεργοποίηση προειδοποιήσεων
`-v`	Εμφάνιση έκδοσης
`-V`	Show configuration, `%ENV` and `@INC`
`--no-jit`	Απενεργοποίηση μεταγλώττισης JIT
`--no-parallel`	Απενεργοποίηση αυτόματης παραλληλοποίησης

Δείτε τις *επιλογές CLI* για την πλήρη λίστα επιλογών.

2.4 Σημειώσεις

- Defaults match perl: modern features (`say`, `state` and friends) are enabled by `use v5.42;` in scripts or `-E` for one-liners
- Τα αρθρώματα XS δεν υποστηρίζονται· παρέχονται εγγενείς υλοποιήσεις σε Rust για τα κοινόχρηστα αρθρώματα
- Module search paths work as in perl: `-I` and the `PERL5LIB` environment variable are honored; `PPERL5LIB` additionally provides pperl-only paths (see *CLI options*)

ΚΕΦΑΛΑΙΟ 3

Επιλογές CLI

Το pperl δέχεται ένα μικρό σύνολο σημαιών γραμμής εντολών. Οι περισσότερες είναι συμβατές με το `perl`· λίγες είναι ειδικές για το pperl (JIT, παραλληλισμός, επιλογή runtime, cache bytecode). Η αυθεντική λίστα είναι ό,τι τυπώνει το `pperl --help`· οι παρακάτω πίνακες δείχνουν αυτές τις σημαίες, εξαγμένες από τον πηγαίο κώδικα του διερμηνευτή και αποτυπωμένες εδώ σε κάθε χτίσιμο της τεκμηρίωσης.

Σημείωση

Η πραγματική έξοδος του `pperl --help` στο τερματικό είναι πάντα στα Αγγλικά - το εκτελέσιμο δεν τοπικοποιεί τα μηνύματα βοήθειας. Η μεταφρασμένη τεκμηρίωση παρουσιάζει τις ίδιες σημαίες με περιγραφές στη γλώσσα της σελίδας, ώστε οι αναγνώστες να μελετήσουν τι κάνει η καθεμία· τα ίδια τα ονόματα των σημαιών (`-T`, `--threads=N`, …) μένουν πάντα αυτούσια.

PetaPerl - Περιβάλλον εκτέλεσης Perl 5 νέας γενιάς

3.1 Χρήση

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

3.2 Επιλογές συμβατές με το Perl

Σημαία	Περιγραφή
`-e 'code'`	Αριθμητικά μονογραμμικά και μονογραμμικά ημερομηνιών
`-E 'code'`	Like -e, but enable all optional features (say, ...)
`-Mmodule`	use module before executing (-MO=Deparse, -Mstrict)
`-mmodule`	use module () before executing (no imports)
`-Idirectory`	Προσθέτει τον κατάλογο στην αρχή του @INC
`-c`	Έλεγχος μόνο της σύνταξης, χωρίς εκτέλεση
`-w`	Ενεργοποίηση προειδοποιήσεων
`-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]`	Αυτόματο chomp της εισόδου, ορισμός τερματιστή εγγραφής εξόδου
`-0[octal]`	Set input record separator (-0 = NUL, -00 = paragraph)
`-i[ext]`	Edit files in place (optional backup extension)
`-T`	Taint mode (forced)
`-t`	Ενεργοποίηση προειδοποιήσεων
`-v`	Εμφάνιση πληροφοριών έκδοσης
`-V`	Show configuration summary, %ENV and @INC
`-V:key`	Show the value of a Config key, e.g. -V:archname
`--interactive`	Launch interactive REPL (requires “shell” feature)
`-h, -?, --help`	Εμφάνιση αυτού του μηνύματος βοήθειας

3.3 Επιλογές ειδικές για το pperl

Σημαία	Περιγραφή
`--stats, -s`	Εμφάνιση στατιστικών απόδοσης (χρόνος, μνήμη, ops)

3.4 Επιλογές runtime

Σημαία	Περιγραφή
`--no-jit`	Απενεργοποίηση μεταγλώττισης JIT (ενεργή από προεπιλογή)
`--no-parallel`	Απενεργοποίηση αυτόματης παραλληλοποίησης (ενεργή από προεπιλογή)
`--threads=SPEC`	Parallel thread count. SPEC is one of:
`phys`	all physical cores (DEFAULT)
`log`	all logical cores (hyperthreads)
`phys-N`	physical minus N (min 1)
`log-N`	logical minus N (min 1)
`P%`	P percent of logical cores
`N`	an explicit count
`--parallel-threshol`	Ελάχιστος αριθμός επαναλήψεων για παραλληλοποίηση (προεπιλογή: 100)
`--jit-stats`	Print JIT summary to stderr at exit (see also PPERL_JIT_LOG)

3.5 Daemon options

Σημαία	Περιγραφή
`--daemon`	Run the fast-start daemon in the foreground
`--daemon=start`	Start the daemon in the background
`--daemon=stop`	Stop the running daemon for this binary
`--daemon=status`	Show daemon status and cached script templates
`--daemon-idle=SECS`	Daemon exits after SECS without a request (default: 900)
`--daemon-max-scripts=`	Cache at most N script templates (default: 100)
`--daemon-reset`	Drop all cached script templates
`--daemon-reset=FILE`	Drop the cached template for FILE only
`--via-daemon`	Route this run through the daemon, same as PPERL_DAEMON=1
`--via-daemon=script`	Reuse a per-script template, same as PPERL_DAEMON=2
`--build-id`	Print the binary identity that keys the daemon socket

3.6 Παραδείγματα

```
pperl script.pl
```

Εκτέλεση ενός σεναρίου Perl

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

Εκτέλεση μονόγραμμου (τα say, state κ.λπ. είναι πάντα διαθέσιμα)

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

Έλεγχος σύνταξης σεναρίου χωρίς εκτέλεση

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

Πέρασμα ορισμάτων στο σενάριο (@ARGV)

Για περισσότερες πληροφορίες, δείτε: https://gl.petatech.eu/petatech/peta-perl

3.7 Module search paths: `-I`, `PERL5LIB`, `@INC`

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

Δύο σκόπιμες διαφορές από την perl:

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

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

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

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

The rest of `@INC` is built by probing the host's standard library trees (Arch, Debian and Fedora layouts) rather than baking one distribution's paths into the binary; `pperl -V` prints the real result. Details and rationale: *differences from upstream* and the `@INC` reference in *program input*.

3.8 Μεταβλητές περιβάλλοντος

Το pperl ορίζει διάφορες μεταβλητές περιβάλλοντος εντός της διεργασίας που εκτελεί, ώστε τα σενάρια να μπορούν να εξετάσουν τον τρέχοντα διερμηνευτή:

Μεταβλητή	Σημασία
`PPERL`	Συμβολοσειρά έκδοσης (ίδια με του `pperl -v`)
`PPERL_FEATUI`	Λίστα διαχωρισμένων με κόμμα ενσωματωμένων χαρακτηριστικών Cargo
`PPERL_MODS`	Ζεύγη `Όνομα/έκδοση` διαχωρισμένα με κόμμα για κάθε εγγενές άρθρωμα
`PPERL_SHELL`	Παραλλαγή ενσωμάτωσης shell (κενή εάν το χαρακτηριστικό `shell` είναι απενεργοποιημένο)
`PPERL_STATI`	`yes` εάν το εκτελέσιμο είναι στατικά συνδεδεμένο με musl, αλλιώς `no`

Εκτελέστε `pperl -V` για να δείτε όλες τις τιμές για το τρέχον εκτελέσιμο.

ΚΕΦΑΛΑΙΟ 4

Οδηγοί χρήσης

Σε αυτή την ενότητα υπάρχουν δύο ειδών υλικά προσανατολισμένα σε εργασίες. Η διάκριση έχει σημασία, διότι σας λέει πόσο χρόνο πρέπει να αφιερώσετε και τι θα έχετε αποκομίσει στο τέλος.

Οι **οδηγοί** είναι εκτενείς και πολυκέφαλοι. Διαβάζονται όπως ένα βιβλίο - από την αρχή έως το τέλος, ή κεφάλαιο προς κεφάλαιο όπου το καθένα βασίζεται στο προηγούμενο. Ένας οδηγός καλύπτει ένα ολόκληρο αντικείμενο σε βάθος. Ο οδηγός για τα regex, για παράδειγμα, σας οδηγεί από το `m//` μέχρι τις ιδιότητες Unicode και την απόδοση του backtracking· ο οδηγός αποσφαλμάτωσης καλύπτει τα πάντα, από τη χρήση του `-d` μέχρι την αποκωδικοποίηση stack traces παραγωγής.

Τα **tutorials** είναι στενά και ολοκληρώνονται σε μία συνεδρία. Ένα tutorial, ένα θέμα, μία συγκεκριμένη ερώτηση που απαντιέται. Φτάνετε σε ένα tutorial με μια εργασία στο μυαλό σας και φεύγετε αφού την έχετε ολοκληρώσει. Χωρίς ευρύτερο τόξο, χωρίς απαιτούμενη προηγούμενη ανάγνωση.

Οδηγοί ***Αρχιτεκτονική του PPerl*** - πώς είναι κατασκευασμένο το pperl: το διπλό runtime, το JIT, το FFI, τα εγγενή αρθρώματα και οι συγκεκριμένες διαφορές από το upstream Perl 5.

Ταυτόχρονη εκτέλεση - εργασία παράλληλα: αυτόματη παραλληλοποίηση, η πραγματικότητα των ithreads και πώς να επιλέξετε ανάμεσά τους.

Κανονικές εκφράσεις - τα regex της Perl από τις βασικές αρχές μέχρι την βελτιστοποίηση απόδοσης.

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.

Αντικειμενοστραφής προγραμματισμός - πρώτα τα σύγχρονα `class`/`field`/`method`, το κλασικό `bless` για συντηρητές, διαδρομές μετάβασης.

Αποσφαλμάτωση - εντοπισμός και διόρθωση σφαλμάτων: ο debugger `-d`, επιθεώρηση ζωντανής κατάστασης, εργαλεία διαδρομής παραγωγής.

Μονόγραμμα - το `pperl` στο shell: `-e`, σιωπηροί βρόχοι, σταδιακές συνταγές, παραμετρικά alias, παγίδες με τα εισαγωγικά.

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*** - διάταξη δυαδικών δεδομένων: οδηγίες, endianness, πρωτόκολλα δικτύου, μορφές αρχείων.

Εργασία με ρεύματα I/O - η πλήρης ιστορία του `open`: λειτουργίες, στρώματα κωδικοποίησης, pipes, handles στη μνήμη, χειρισμός σφαλμάτων.

Αναφορές Perl - βαθμωτά, πίνακες, hashes, κώδικας, ανώνυμες και αδύναμες αναφορές, εμφωλευμένες δομές.

Unicode στην Perl - κείμενο έναντι bytes, κωδικοποίηση πηγής, στρώματα PerlIO, regex με Unicode, συνήθεις παγίδες.

Λογική Boole για προγραμματιστές Perl - αληθινότητα, οι δεκαέξι δυαδικοί τελεστές, De Morgan, πληρότητα NAND/NOR, εφαρμογές σε bitwise και regex.

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

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

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

1. **Exactness.** Compiled code produces byte-identical output to the interpreter, always. Not «close enough»: the test harness compares pperl's output byte-for-byte against perl5 on every test, with the JIT and parallelizer on.

2. **Decline, never guess.** Anything the compiler cannot prove it can reproduce exactly, it declines. A declined loop runs on the interpreter at normal speed. There is no mode in which compiled code is «probably right».

What compiles

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

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

The exactness machinery

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

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

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

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

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

Παράλληλη εκτέλεση και locale

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

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

Observing the JIT

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

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

Εφαρμογές

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

Numbers

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

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

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

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	Σκοπός
0	Peta::FFI	Raw FFI: dlopen + call with type signatures
1	Peta::FFI::Libc, ::UUID, ::GMP	Pre-baked bindings for specific libraries
2	Peta::FFI::scan()	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...
print "sqrt(2) = $result\n";
dlclose($lib);
```

Type Signature Format

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

Κωδικός	C Type	Perl Mapping
v	κενό	(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)

Παραδείγματα:

- `"(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("12345678901234567890123456789");
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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	Σημειώσεις
Scalar::Uti	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	Σημειώσεις
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	Σημειώσεις
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	Σημειώσεις
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	Σημειώσεις
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	Σημειώσεις
POSIX	floor, ceil, fmod, strtod, strftime, …	Subset
Math::GMP	Arbitrary precision integers	Via Peta::FFI::GMP

Build / Config

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

Ενδοσκόπηση

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

Απόδοση

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):

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

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

Intentional Differences

Υπό συνθήκη φόρτωση

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

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

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

Module loading and `@INC`

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

- **Native modules have absolute priority.** A module built into the binary is served by `require` directly; a same-name `.pm` on disk can never shadow it, regardless of `-I` or `PERL5LIB`. Everything not built in resolves through `@INC` in normal first-hit-wins order.
- **No architecture directories.** Neither the host's `archlib` trees nor `x86_64-linux`-style children of `-I`/`PERL5LIB` entries are searched. They exist to hold compiled `.so` objects and their loaders; skipping them is what lets library trees built by a real perl (`local::lib`, Carton) work unmodified.
- **Host library trees are probed, not baked.** `@INC` is constructed by checking the standard Arch, Debian and Fedora pure-perl tree locations for existence, and versioned children of those trees and of `-I`/`PERL5LIB` entries are ordered by nearness to pperl's own version: exact match first, then the same minor release descending, then newer versions, then the plain directory, then older versions. A host perl of a different version is used gracefully as a last resort rather than rejected. `pperl -V` always shows the actual probed result.

Additionally, pperl-private search locations exist for parallel installations: the `PPERL5LIB` environment variable (honored before `PERL5LIB`, same rules) and the overlay trees under `/usr/local/share/pperl` and `/usr/share/pperl`, probed

before every host tree. They allow shipping or overriding a `.pm` for pperl without touching the shared perl trees; the host perl never reads them, and nothing in them can outrank a built-in native module.

No interpreter threads

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

Linux only

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

Ταυτότητα διεργασίας

`$]` reports `5.044099` and `$^V` reports `v5.44.99`: a 5.44-class interpreter, above any released 5.44.x patch level. `use v5.44` and version checks against 5.44 or below behave exactly as on a real 5.44. `$Config{version}` reports `5.44.0`.

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

Επιλογές γραμμής εντολών

Δεν υποστηρίζονται:

Σημαία	Περιγραφή
`-M` / `-m`	Load module from command line
`-x`	Extract script from message
`-d`	Ο διαδραστικός αποσφαλματωτής

`-I`, `-n`/`-p`, `-l`, `-a`/`-F`, `-0`, `-i`, `-T`/`-t`, `-c`, `-w`, `-e`/`-E` and `-V`/`-V:key` work as in perl. pperl-specific flags (JIT, parallelism) are listed in *CLI options*.

Γνωστά κενά

Current, specific, and expected to shrink:

- **`charnames` emits redefinition warnings.** Loading `charnames` (or `\N{...}` names) prints `Subroutine re::is_regexp redefined` warnings before working correctly: the built-in `re` module pre-registers helpers that `_charnames.pm` also defines. Output is otherwise exact.
- **`B::Deparse` on subs with signatures.** The optree for signature parameters uses a newer internal representation than the deparser expects; deparsing such subs produces an «unexpected OP» note instead of the signature. Ordinary subs deparse exactly.

- **Embedding pperl in C hosts** (mod_perl-class, programs linking libperl) is out of scope until a concrete user appears.

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

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

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

One gate is stricter than the side-effect analysis and deserves its own sentence: **bit-exactness**. Integer reductions always qualify. Floating-point reductions qualify only when reassociating the sum provably cannot change the result (for example when

every contribution is integral and within the exact range of a double). An inexact float reduction runs compiled but sequential - pperl never trades determinism for cores.

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

CLI Control

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

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

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

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

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

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

Απόδοση

Mandelbrot Set (1000x1000, mid-2026)

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

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

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.

pperl status - read this first

pperl **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. pperl 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 pperl, 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 pperl equivalent.

How this chapter is organised

ithreads basics

This chapter describes the perl5 ithreads model as it exists in upstream Perl 5.44. It is the vocabulary you need to read existing threaded Perl code. pperl does not run this code - see the landing page for the reason and *Alternatives* for the pperl-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(\&thrsub, 'one');
my $thr2 = threads->create(\&thrsub, 'two');

sub thrsub {
    my ($label) = @_;
    sleep 1;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    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.

pperl 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.

pperl'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*.

Δείτε επίσης

- `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 5.44, 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.

pperl 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 processed by a data-parallel loop, or a compiled `map`/`grep` - 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.

Διαφορές από το upstream

pperl 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.

pperl 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.

Δείτε επίσης

- `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

pperl 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.»

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

Decision table

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

Auto-parallelisation for compute loops

pperl'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 += $i * $i % 7;
}
print $sum, "\n";
```

Under pperl 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.

map and grep at native speed

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

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

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

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.

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: $!";
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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.

Δείτε επίσης

- *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.

Διασταυρούμενοι σύνδεσμοι αναφοράς

- `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 Κανονικές εκφράσεις

Μια κανονική έκφραση (regexp, regex) είναι ένα μοτίβο που αποφασίζει αν μια συμβολοσειρά έχει ένα δεδομένο σχήμα, ή που εξάγει τμήματα από μια συμβολοσειρά που το έχει. Η Perl αντιμετωπίζει τις regex ως υπογλώσσα πρώτης τάξης: εμφανίζονται οπουδήποτε γίνεται αντιστοίχιση (`m//`), αντικατάσταση (`s///`), παράθεση μοτίβου (`qr//`), ή διαχωρισμός με βάση διαχωριστή (`split`).

Αυτός ο οδηγός είναι η οριστική αναφορά της pperl για τις regex. Κάθε κεφάλαιο καλύπτει ένα θέμα σε τρία επίπεδα: μια επισκόπηση που σας προσανατολίζει, μια ακριβή αναφορά που ορίζει τι κάνει η κατασκευή, και τις λεπτομέρειες - οριακές περιπτώσεις, παθολογίες απόδοσης, και τις διαφορές μεταξύ μηχανών που εκπλήσσουν έναν αναγνώστη εκπαιδευμένο στην Perl. Το ίδιο κεφάλαιο εξυπηρετεί τον αναγνώστη που χρειάζεται μια απάντηση μιας παραγράφου και τον αναγνώστη που θέλει να ξέρει ακριβώς πότε ένα μοτίβο θα συμπεριφερθεί παράξενα.

4.3.1 Σε ποιους απευθύνεται

Σε αναγνώστες που γνωρίζουν Perl αρκετά καλά ώστε να χρησιμοποιούν βαθμωτά, πίνακες και κατακερματισμούς, αλλά αντιμετωπίζουν τις regex ως κάτι που αντιγράφουν από αλλού ελπίζοντας να δουλέψει. Μετά την ανάγνωση, θα διαβάζετε ένα άγνωστο μοτίβο και θα ξέρετε με τι θα ταιριάξει - και πότε θα αρνηθεί, και γιατί.

4.3.2 Πώς είναι οργανωμένος ο οδηγός

Τα κεφάλαια προορίζονται να διαβαστούν με σειρά σε μια πρώτη ανάγνωση, αλλά καθένα στέκεται από μόνο του ως μεταγενέστερη αναφορά.

Βασικά

Η μικρότερη χρήσιμη regex είναι μια απλή συμβολοσειρά. Το `"Hello World" =~ /World/` ρωτά: περιέχει η συμβολοσειρά στα αριστερά το μοτίβο στα δεξιά; Το περιέχει, άρα η έκφραση είναι αληθής.

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

Τα `//` περικλείουν το μοτίβο. Ο τελεστής `=~` συνδέει το μοτίβο με τη συμβολοσειρά που θέλετε να ελέγξετε. Χωρίς τελεστή σύνδεσης, η Perl εφαρμόζει το μοτίβο στο `$_`.

Ο τελεστής αντιστοίχισης

Η εκτεταμένη μορφή είναι `m//`:

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

Το `m` σάς επιτρέπει να επιλέξετε οποιοδήποτε διαχωριστικό. Αυτό έχει σημασία όταν το ίδιο το μοτίβο περιέχει το προεπιλεγμένο διαχωριστικό `/` - συγκρίνετε

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

Τα ζευγαρωτά διαχωριστικά (`{}`, `()`, `[]`, `<>`) εμφωλεύονται, κάτι που είναι χρήσιμο όταν το μοτίβο περιέχει τον χαρακτήρα του διαχωριστικού.

Χωρίς `m`, η αρχική κάθετος είναι υποχρεωτική: μόνο `/pat/`. Με `m`, το αρχικό `m` είναι υποχρεωτικό: `m{pat}`, όχι `{pat}`.

Μια μικρή αλλά χρήσιμη γωνία: τα `m''` (απλά εισαγωγικά ως διαχωριστικά) κάνουν το μοτίβο *τύπου απλών εισαγωγικών* - χωρίς παρεμβολή μεταβλητών, χωρίς διαφυγές διπλών εισαγωγικών. Χρήσιμο όταν το μοτίβο πρόκειται να περιέχει κυριολεκτικό `$` ή `@` και δεν θέλετε να τα διαφεύγετε.

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

Άλλοι τελεστές regex

Το `m//` είναι ένα από τα τέσσερα:

Τελεστής	Σκοπός
`m//`	αντιστοίχιση - ταιριάζει το μοτίβο με τη συμβολοσειρά;
`s///`	αντικατάσταση - αντικαθιστά την αντιστοιχία με κάτι άλλο
`qr//`	μεταγλωττίζει ένα αντικείμενο μοτίβου για επαναχρησιμοποίηση
`tr///` (επίσης `y///`)	μετάφραση χαρακτήρα προς χαρακτήρα

Το `tr///` *δεν* είναι τελεστής regex παρότι βρίσκεται στην ίδια γειτονιά - εκτελεί μετάφραση κλάσεων χαρακτήρων, όχι αντιστοίχιση μοτίβων. Αναφέρεται για πληρότητα· δείτε `tr` για τη σημασιολογία του.

Τα `m`, `s` και `qr` μοιράζονται όλα τη σύνταξη regex που καλύπτεται σε αυτόν τον οδηγό. Οι διαφορές βρίσκονται σε αυτό που κάνουν *με* μια επιτυχημένη αντιστοιχία.

Επαναχρησιμοποίηση μοτίβου με qr//

Το `qr//` μεταγλωττίζει ένα μοτίβο μία φορά και παράγει ένα επαναχρησιμοποιήσιμο αντικείμενο. Χρησιμοποιήστε το όταν το ίδιο μοτίβο αντιστοιχίζεται επανειλημμένα, ιδίως μέσα σε βρόχους:

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

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

Το μεταγλωττισμένο αντικείμενο `qr//` εισάγεται σε άλλα μοτίβα μέσω παρεμβολής. Επίσης σας επιτρέπει να χτίζετε μοτίβα από ονομασμένα τμήματα, κάτι που γίνεται απαραίτητο για κάθε regex πάνω από περίπου δέκα γραμμές:

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

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

Το ίδιο όφελος της μίας μεταγλώττισης, εκφρασμένο στο σωστό επίπεδο αφαίρεσης.

Σύνδεση: =~ και !~

Το `=~` ρωτά «ταιριάζει;». Το `!~` ρωτά «αποτυγχάνει να ταιριάξει;».

```
$s = "Hello World";

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

Το `!~` δεν είναι ξεχωριστή κατασκευή regex - είναι η αρνημένη σύνδεση. Ισοδυναμεί με `not ($s =~ /pat/)`.

Αντιστοίχιση με το $_

Αν παραλείψετε τη σύνδεση, η αντιστοίχιση γίνεται με το `$_`:

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

Αυτό είναι ιδιωματικό σε βρόχους `while (<>)`, μέσα σε `grep` και `map`, και μέσα σε βρόχους `for` που ορίζουν το `$_`.

Διάκριση πεζών-κεφαλαίων και η προεπιλεγμένη άγκυρα

Οι αντιστοιχίες γίνονται με διάκριση πεζών-κεφαλαίων και χωρίς αγκύρωση:

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

Για αντιστοίχιση χωρίς διάκριση πεζών-κεφαλαίων, προσθέστε `/i`. Για να περιορίσετε την αντιστοιχία στην αρχή ή το τέλος της συμβολοσειράς, χρησιμοποιήστε άγκυρες. Και τα δύο καλύπτονται στα δικά τους κεφάλαια.

Όταν ένα μοτίβο μπορεί να ταιριάξει σε περισσότερες από μία θέσεις, η Perl δοκιμάζει από τα αριστερά και παίρνει την πρώτη που λειτουργεί:

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

Ο κανόνας «η αντιστοιχία πιο αριστερά κερδίζει» είναι θεμελιώδης και υπερισχύει κάθε άλλης προτίμησης αντιστοίχισης: μια αντιστοιχία σε προγενέστερη θέση είναι πάντα καλύτερη από μια αντιστοιχία σε μεταγενέστερη θέση, ανεξάρτητα από τις εσωτερικές επιλογές κάθε αντιστοίχισης. Αυτό αποκαλείται μερικές φορές *ιδιότητα bump-along*: η μηχανή προσπαθεί να ταιριάξει στη θέση 0· αν αποτύχει, μετακινείται στη θέση 1· αν αποτύχει, στη θέση 2· κ.ο.κ., επιστρέφοντας την πρώτη επιτυχία.

Μετα-χαρακτήρες

Οι περισσότεροι χαρακτήρες σε ένα μοτίβο ταιριάζουν με τον εαυτό τους. Οι παρακάτω όχι:

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

Δύο ακόμα είναι ειδικοί μόνο σε συγκεκριμένα περιβάλλοντα:

- Το `-` είναι μετα-χαρακτήρας μόνο μέσα σε κλάση χαρακτήρων, όπου σχηματίζει εύρος (`[a-z]`). Εκτός `[…]` είναι κυριολεκτικό.
- Το `#` είναι μετα-χαρακτήρας μόνο υπό τη σημαία `/x`, όπου εισάγει σχόλιο μέχρι το τέλος της γραμμής. Χωρίς `/x` είναι κυριολεκτικό.

Κάθε μετα-χαρακτήρας έχει μια ειδική σημασία που καλύπτεται αργότερα. Για να ταιριάξετε ένα κυριολεκτικό αντίγραφο, βάλτε μια ανάστροφη κάθετο μπροστά:

```
"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.
```

Η ίδια η ανάστροφη κάθετος είναι μετα-χαρακτήρας, οπότε μια κυριολεκτική ανάστροφη κάθετος σε ένα μοτίβο χρειάζεται `\\`:

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

Ένας μετα-χαρακτήρας που δεν έχει τίποτα ειδικό να κάνει στο περιβάλλον του επιστρέφει στο να ταιριάζει με τον εαυτό του. Το } κλείνει μόνο έναν ποσοδείκτη {…}· εκτός αυτού του περιβάλλοντος είναι κυριολεκτικό }. Αυτό είναι βολικό αλλά εύκολο να παρερμηνευτεί· δείτε *Αυστηρή λειτουργία* παρακάτω.

Αυστηρή λειτουργία: `use re 'strict'`

Το `use re 'strict'` μετατρέπει την προηγουμένως ανεκτή χαλαρότητα στις regex σε σφάλματα κατά τη μεταγλώττιση. Χρησιμοποιήστε το όταν θέλετε ο μεταγλωττιστής regex να επισημαίνει μοτίβα που *πιθανώς* σημαίνουν κάτι διαφορετικό από αυτό που γράψατε:

```
use re 'strict';

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

Η αυστηρή λειτουργία είναι ανά λεξική εμβέλεια, οπότε μπορεί να ενεργοποιηθεί για αρθρώματα με πολλές regex χωρίς να επηρεάζει άλλον κώδικα. Δεν είναι προεπιλεγμένη γιατί θα έσπαζε λειτουργικά παλαιότερα μοτίβα· ο νέος κώδικας θα έπρεπε να την εξετάζει.

Ακολουθίες διαφυγής

Οι μη εκτυπώσιμοι χαρακτήρες χρησιμοποιούν τις ίδιες ακολουθίες διαφυγής όπως στις συμβολοσειρές διπλών εισαγωγικών:

Ακολουθία	Ταιριάζει με
`\t`	στηλοθέτη (tab)
`\n`	νέα γραμμή
`\r`	επαναφορά κεφαλής (carriage return)
`\f`	αλλαγή σελίδας (form feed)
`\a`	ειδοποίηση (κουδούνι)
`\e`	escape (`\x1B`)
`\0`	byte NUL
`\xHH`	byte με δεκαεξαδική τιμή HH
`\x{…}`	σημείο κώδικα Unicode με δεκαεξαδική τιμή
`\o{…}`	οκταδικό σημείο κώδικα
`\cX`	control-X
`\N{…}`	χαρακτήρας Unicode με όνομα

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

Η πλήρης ιστορία Unicode βρίσκεται στο κεφάλαιο *unicode*.

Μεταβλητές μέσα στα μοτίβα

Ένα μοτίβο (από προεπιλογή) παρεμβάλλεται όπως μια συμβολοσειρά διπλών εισαγωγικών, οπότε οι μεταβλητές αντικαθίστανται πριν την αντιστοίχιση:

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

Για να ταιριάξετε ένα κυριολεκτικό $ ή @, διαφεύγετέ το:

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

Αν μια συμβολοσειρά που παρέχεται από τον χρήστη πρόκειται να παρεμβληθεί σε ένα μοτίβο και θέλετε οι μετα-χαρακτήρες της να αντιμετωπιστούν κυριολεκτικά, χρησιμοποιήστε `quotemeta` - ή το εντός μοτίβου ισοδύναμό του `\Q…\E`:

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

Χωρίς `\Q…\E` το + θα διαβαζόταν ως ποσοδείκτης.

Πώς διαβάζεται μια regex

Η Perl διαβάζει ένα μοτίβο regex σε τέσσερις φάσεις. Η γνώση των φάσεων εξηγεί μερικά ερωτήματα του τύπου «γιατί δουλεύει αυτό;»:

1. **Φάση Α**: ο αναλυτής αναγνωρίζει το διαχωριστικό και εντοπίζει το τέλος του μοτίβου. Τα σχόλια `(?#…)` αφαιρούνται.
2. **Φάση Β**: το μοτίβο αναλύεται ως *συμβολοσειρά τύπου διπλών εισαγωγικών* - οι μεταβλητές παρεμβάλλονται, οι ακολουθίες διαφυγής επεξεργάζονται, το `\Q…\E` μεταφράζεται σε διαφυγή τύπου `quotemeta`.
3. **Φάση Γ**: υπό `/x` ή `/xx`, οι λευκοί χαρακτήρες χωρίς διαφυγή και τα σχόλια μετά το # αφαιρούνται.
4. **Φάση Δ**: ο μεταγλωττιστής regex διαβάζει το αποτέλεσμα και το μετατρέπει στην εσωτερική μορφή της μηχανής.

Η σειρά έχει σημασία επειδή οι φάσεις Β και Δ λειτουργούν σε διαφορετικές αναπαραστάσεις. Το `\Q$dir\E` επεξεργάζεται στη Φάση Β, πριν το δει ο μεταγλωττιστής regex - μέχρι τη Φάση Δ, τα περιεχόμενα της μεταβλητής έχουν ήδη διαφύγει, και ο μεταγλωττιστής regex βλέπει ένα κυριολεκτικό μοτίβο. Αντιθέτως, το `\U…\E` ερμηνεύεται από τη Φάση Β ως οδηγία επεξεργασίας συμβολοσειράς (μετατροπή των περιεχομένων σε κεφαλαία), κάτι που σχεδόν σίγουρα *δεν* είναι αυτό που θέλετε μέσα σε μια regex. Η σύμβαση είναι: `\Q` και `\E` για σκοπούς regex· `\U`, `\L`, `\u`, `\l` μόνο όταν ξέρετε τι κάνετε.

Το `(?#comment)` αφαιρείται προτού η Φάση Β δει καν το μοτίβο. Ένα κυριολεκτικό # μέσα στο `(?#…)` είναι μια χαρά. Ένα κυριολεκτικό `)` όχι - το σχόλιο τελειώνει στην πρώτη `)`.

Η νωρίτερη αντιστοιχία κερδίζει - το bump-along

Ο Friedl το διατυπώνει με ακρίβεια:

> Η αντιστοιχία που ξεκινά νωρίτερα κερδίζει.

Η θέση 0 δοκιμάζεται πρώτη, μετά η 1, μετά η 2, κ.ο.κ. Η μηχανή *ποτέ* δεν προτιμά μια μεταγενέστερη, μακρύτερη ή «πιο αισθητικά καλή» αντιστοιχία έναντι μιας προγενέστερης. Αυτό είναι τόσο θεμελιώδες ώστε διαμορφώνει κάθε άλλον κανόνα σε αυτόν τον οδηγό:

- «Οι άπληστοι ποσοδείκτες αρπάζουν όσο το δυνατόν περισσότερα» - *στην τρέχουσα θέση εκκίνησης*. Η αρπαγή συμβαίνει αφού το bump-along έχει επιλέξει τη θέση.
- «Η πιο αριστερή εναλλακτική κερδίζει» - *στην τρέχουσα θέση εκκίνησης*. Η εναλλακτική που επιλέγεται επηρεάζει τι συλλαμβάνεται αλλά όχι το πού ξεκινά η αντιστοιχία.
- Άγκυρες όπως η ^ περιορίζουν *ποιες θέσεις είναι νόμιμες*, όχι τι προτιμάται ανάμεσα στις νόμιμες.

Πείτε αυτό που εννοείτε

Το επαναλαμβανόμενο σημείο του Friedl: η ασάφεια στη regex προκαλεί τόσο προβλήματα ορθότητας όσο και προβλήματα απόδοσης. Το παράδειγμα στο οποίο επιμένει:

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

Διαβασμένο ως αγγλικά: «προαιρετικό πρόσημο, προαιρετικά ψηφία, προαιρετική υποδιαστολή, προαιρετικά ψηφία». Διαβασμένο ως κώδικας: κάθε τμήμα είναι προαιρετικό, οπότε το μοτίβο *ταιριάζει με την κενή συμβολοσειρά* - στην αρχή οποιασδήποτε εισόδου, προτού καν κοιτάξει η μηχανή. Εφαρμόστε το στο «nothing here» και ταιριάζει στη θέση 0, συλλαμβάνοντας την κενή συμβολοσειρά, επιστρέφοντας επιτυχία.

Η διόρθωση είναι να απαιτήσετε τουλάχιστον ένα ψηφίο σε τουλάχιστον μία πλευρά της υποδιαστολής:

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

Δύο εναλλακτικές, κάθε μία απαιτεί τουλάχιστον ένα ψηφίο. Το μοτίβο τώρα ταιριάζει με αυτό που σκόπευε ο συγγραφέας του.

Το μάθημα γενικεύεται. Ένα μοτίβο που *μπορεί* να ταιριάξει με την κενή συμβολοσειρά συνήθως *θα* ταιριάξει με την κενή συμβολοσειρά κάπου. Αν η αντιστοιχία σας πρέπει να σημαίνει κάτι, γράψτε απαιτήσεις που την αναγκάζουν να σημαίνει κάτι.

Αντικατάσταση με μια ματιά

Η αντικατάσταση κειμένου χρησιμοποιεί τον τελεστή `s///`, ο οποίος δέχεται ένα μοτίβο και μια συμβολοσειρά αντικατάστασης:

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

Η αντικατάσταση καλύπτεται σε βάθος στο δικό της κεφάλαιο· αναφέρεται εδώ ώστε να μπορείτε να τη συνδυάσετε με τα παραπάνω δεδομένα. Σχεδόν ό,τι ισχύει για τα μοτίβα `m//` ισχύει και μέσα στα μοτίβα `s///`.

Πού να πάτε στη συνέχεια

Οι κυριολεκτικές αντιστοιχίες σάς πάνε εκπληκτικά μακριά, αλλά κάθε πραγματική regex χρησιμοποιεί κλάσεις χαρακτήρων, άγκυρες, ή ποσοδείκτες. Οι κλάσεις χαρακτήρων ακολουθούν - επιτρέπουν σε μία θέση στο μοτίβο να δέχεται οποιονδήποτε από αρκετούς χαρακτήρες.

Αν διαβάζετε τον οδηγό για μια συγκεκριμένη ερώτηση, τα κεφάλαια είναι ανεξάρτητα και ο *δείκτης* τα παραθέτει όλα. Μοτίβα που διαβάζονται μια χαρά αλλά τρέχουν για ώρες καλύπτονται στο *performance* - όταν αμφιβάλλετε, η απάντηση συνήθως βρίσκεται εκεί.

Δείτε επίσης

- `m` - πλήρης αναφορά για τον τελεστή αντιστοίχισης.
- `s` - πλήρης αναφορά για την αντικατάσταση.
- `qr` - μεταγλώττιση αντικειμένου μοτίβου.
- `quotemeta` - διαφυγή συμβολοσειράς για ασφαλή παρεμβολή σε μοτίβο.
- Το κεφάλαιο *character classes* - τι ακολουθεί μετά την κυριολεκτική αντιστοίχιση.
- Το κεφάλαιο *performance* - τι μπορεί να πάει στραβά.

Κλάσεις χαρακτήρων

Μια κλάση χαρακτήρων ταιριάζει με ακριβώς έναν χαρακτήρα, επιλεγμένο από ένα σύνολο που ορίζετε. Ενώ ένα κυριολεκτικό `a` ταιριάζει μόνο με `a`, η κλάση `[abc]` ταιριάζει με οποιονδήποτε από τους `a`, `b`, ή `c`.

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

Η κλάση εξακολουθεί να καταναλώνει έναν χαρακτήρα της συμβολοσειράς. Το `[bcr]at` ποτέ δεν ταιριάζει με `at` (δεν υπάρχει γράμμα) και ποτέ δεν ταιριάζει με `brat` (δύο γράμματα ενώ η κλάση περιμένει ένα).

Εύρη

Μέσα στα `[…]`, μια παύλα ανάμεσα σε δύο χαρακτήρες δηλώνει ένα συνεχές εύρος στο υποκείμενο σύνολο χαρακτήρων:

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

Τα εύρη μπορούν να συνδυαστούν με μεμονωμένους χαρακτήρες:

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

Μια παύλα που βρίσκεται πρώτη ή τελευταία μέσα στην κλάση είναι κυριολεκτική:

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

Άρνηση

Ένα ^ ως πρώτος χαρακτήρας μέσα στα [...] αντιστρέφει την κλάση:

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

Ένα ^ οπουδήποτε αλλού είναι κυριολεκτικό:

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

Μια αρνημένη κλάση εξακολουθεί να ταιριάζει με *έναν* χαρακτήρα - το [^a] δεν ταιριάζει με την κενή συμβολοσειρά· απαιτεί έναν χαρακτήρα διαφορετικό από a.

Ειδικοί χαρακτήρες μέσα σε κλάση

Μέσα στα [...] το ειδικό σύνολο συρρικνώνεται σε -] \ ^ $ (και στο διαχωριστικό του μοτίβου). Οι υπόλοιποι - ., *, +, ?, (,), {, }, | - είναι κυριολεκτικοί μέσα σε κλάση:

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

Για να ταιριάξετε] μέσα στην κλάση, είτε διαφεύγετέ το είτε τοποθετήστε το πρώτο (μετά από τυχόν αρχικό ^):

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

Τα $ και \ είναι ελαφρώς δύσχρηστα επειδή αλληλεπιδρούν με την παρεμβολή και τη διαφυγή:

```
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
```

Το \b μέσα σε κλάση χαρακτήρων σημαίνει *backspace* (\x08), όχι «όριο λέξης». Εκτός κλάσης είναι η διεκδίκηση ορίου λέξης. Αυτή είναι η συχνότερη παγίδα διπλής σημασίας στη σύνταξη regex - δείτε το κεφάλαιο *anchors and assertions* για τη μορφή ορίου.

Συντομογραφικές κλάσεις

Αρκετές συνηθισμένες κλάσεις έχουν συντομογραφικά ονόματα που χρησιμοποιούνται τόσο μέσα όσο και έξω από τα `[...]`:

Συντομογραφία	Ταιριάζει με
`\d`	ένα ψηφίο
`\D`	ένα μη ψηφίο
`\w`	έναν χαρακτήρα λέξης (αλφαριθμητικό ή _)
`\W`	έναν χαρακτήρα μη λέξης
`\s`	λευκό χαρακτήρα (διάστημα, στηλοθέτη, `\r`, `\n`, `\f`, και άλλους)
`\S`	μη λευκό χαρακτήρα
`\h`	οριζόντιο λευκό χαρακτήρα (διάστημα, στηλοθέτη, Unicode)
`\H`	μη οριζόντιο λευκό χαρακτήρα
`\v`	κάθετο λευκό χαρακτήρα (`\n`, `\r`, `\f`, ...)
`\V`	μη κάθετο λευκό χαρακτήρα
`\R`	αλλαγή γραμμής: `\r\n`, `\n`, `\v`, `\f`, `\x{85}`, ...
`\N`	οποιονδήποτε χαρακτήρα εκτός από `\n` (ανεξαρτήτως `/s`)

Υπό Unicode (η προεπιλογή), τα `\d`, `\w`, `\s` ταιριάζουν με περισσότερα από μόνο ASCII. Το `\d` ταιριάζει με οποιοδήποτε ψηφίο Unicode (ψηφία Ντεβανάγκαρι, αραβο-ινδικά ψηφία, και πολλά άλλα), το `\w` ταιριάζει με οποιοδήποτε γράμμα σε οποιαδήποτε γραφή συν σημάδια και συνδετική στίξη, και το `\s` προσθέτει χαρακτήρες διαστήματος Unicode όπως το αδιάσπαστο διάστημα.

Για να τα περιορίσετε σε ASCII, προσθέστε τον τροποποιητή `/a` ή χρησιμοποιήστε ρητά εύρη όπως `[0-9]` και `[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
```

Το `\R` είναι η συντομογραφία αλλαγής γραμμής - ταιριάζει με οποιαδήποτε από τις αναγνωρισμένες ακολουθίες αλλαγής γραμμής ως μία μονάδα. Χρήσιμο για την ανάλυση κειμένου που μπορεί να έχει εναλλακτικά CRLF, LF, ή σπανιότερους τερματιστές γραμμής. Σε αντίθεση με μια κλάση, το `\R` μπορεί να ταιριάζει με *δύο* χαρακτήρες (η περίπτωση CRLF) και έτσι δεν μπορεί να εμφανιστεί μέσα στα `[...]`.

Το `\N` (κεφαλαίο) σημαίνει «οποιονδήποτε χαρακτήρα εκτός από `\n`», και *δεν* επηρεάζεται από τον τροποποιητή `/s`. Αυτή είναι η διπλή σημασία για την οποία πρέπει να προσέξετε: το `\N{NAME}` (με αγκύλη) είναι ονομαστική διαφυγή χαρακτήρα Unicode (δείτε το κεφάλαιο *unicode*)· το γυμνό `\N` είναι η κλάση μη νέας γραμμής.

Η τελεία

Η `.` ταιριάζει με οποιονδήποτε μεμονωμένο χαρακτήρα εκτός από τη νέα γραμμή. Υπό τον τροποποιητή `/s` (καλύπτεται στο κεφάλαιο *modifiers*), η `.` ταιριάζει επίσης με νέα γραμμή:

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

Όταν θέλετε «οποιονδήποτε χαρακτήρα συμπεριλαμβανομένης της νέας γραμμής» χωρίς `/s`, ο κλασικός ιδιωματισμός είναι το `[\s\S]` (ή `[\d\D]`):

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

Το τέχνασμα είναι ότι οποιοσδήποτε χαρακτήρας είναι είτε λευκός χαρακτήρας είτε μη λευκός χαρακτήρας· η κλάση καλύπτει και τα δύο.

Σύνθεση κλάσεων

Μπορείτε να αναμείξετε συντομογραφίες, εύρη, και μεμονωμένους χαρακτήρες μέσα σε μία κλάση:

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

Ο νόμος του De Morgan έχει σημασία: το `[^\d\w]` *δεν* είναι το `[\D\W]`. Το πρώτο απαιτεί ο χαρακτήρας να είναι *και* μη ψηφίο *και* μη χαρακτήρας λέξης. Όμως κάθε ψηφίο είναι χαρακτήρας λέξης, οπότε το `[^\d\w]` απλοποιείται σε `[^\w]`, δηλαδή `\W`. Να είστε προσεκτικοί όταν συνδυάζετε αρνημένες συντομογραφίες.

Κλάσεις POSIX

Οι κλάσεις χαρακτήρων POSIX χρησιμοποιούν τη μορφή `[:name:]` και δουλεύουν μόνο μέσα σε `[…]`:

POSIX	Ισοδύναμο
`[:alpha:]`	αλφαβητικός
`[:alnum:]`	αλφαριθμητικός
`[:digit:]`	ψηφίο (όπως `\d`)
`[:word:]`	χαρακτήρας λέξης (επέκταση Perl)
`[:space:]`	λευκός χαρακτήρας (όπως `\s`)
`[:upper:]`	κεφαλαία
`[:lower:]`	πεζά
`[:xdigit:]`	δεκαεξαδικό ψηφίο
`[:ascii:]`	0x00–0x7F
`[:cntrl:]`	χαρακτήρας ελέγχου
`[:graph:]`	εκτυπώσιμος, όχι διάστημα
`[:print:]`	εκτυπώσιμος, συμπεριλαμβανομένου του διαστήματος
`[:punct:]`	στίξη
`[:blank:]`	διάστημα ή στηλοθέτη

Αρνηθείτε μια κλάση POSIX με `^` *μέσα* στις άνω-κάτω τελείες:

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

Οι κλάσεις POSIX ακολουθούν τους ίδιους κανόνες Unicode έναντι ASCII με τις συντομογραφίες: χωρίς `/a`, η `[:alpha:]` είναι το αλφαβητικό σύνολο Unicode.

Το POSIX ορίζει επίσης δύο σχετικές κατασκευές που σπάνια υλοποιούνται:

- **Στοιχεία ταξινόμησης** `[.span-ll.]` - ταιριάζουν με ένα στοιχείο ταξινόμησης πολλών χαρακτήρων ως μία μονάδα (π.χ. το ισπανικό `ll` ιστορικά).
- **Κλάσεις ισοδυναμίας** `[[=n=]]` - ταιριάζουν με οποιονδήποτε χαρακτήρα που είναι ισοδύναμος υπό τους κανόνες ταξινόμησης του locale (π.χ. τονισμένες παραλλαγές του n).

Η Perl αναγνωρίζει τη σύνταξη αλλά αντιμετωπίζει και τις δύο μορφές ως τους κυριολεκτικούς χαρακτήρες. Στην πράξη κανένα φορητό σενάριο δεν βασίζεται σε αυτές· τεκμηριώνονται για πληρότητα.

Ιδιότητες Unicode

Το Unicode ορίζει χιλιάδες ιδιότητες. Η σημειογραφία είναι `\p{Name}` για «έχει αυτή την ιδιότητα» και `\P{Name}` για «δεν έχει αυτή την ιδιότητα»:

```
/\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
```

Σύντομα μονόγραμμα ψευδώνυμα υπάρχουν για συνηθισμένες ιδιότητες και παραλείπουν τα άγκιστρα: το `\pL` είναι γράμμα, το `\pN` αριθμός, το `\pP` στίξη. Το `\p{L}` είναι το ίδιο με το `\pL`.

Το κεφάλαιο *unicode* καλύπτει τις ιδιότητες λεπτομερώς, συμπεριλαμβανομένης της σύνθετης μορφής `\p{Name=Value}`, του συμπλέγματος γραφημάτων `\X`, και των τροποποιητών συνόλου χαρακτήρων `/a`, `/u`, `/l`, `/d`.

Εκτεταμένες κλάσεις σε αγκύλες - `(?[ ])`

Η τυπική σύνταξη `[…]` χειρίζεται καλά τις ενώσεις («οποιοσδήποτε από αυτούς τους χαρακτήρες») αλλά δεν έχει πράξεις συνόλων στις κλάσεις. Η εκτεταμένη μορφή `(?[ … ])` έχει:

Τελεστής	Σημασία
`+` ή `|`	ένωση (οι ίδιοι χαρακτήρες που έχει οποιοσδήποτε από τους τελεσταίους)
`&`	τομή (και στους δύο)
`-`	διαφορά (στον αριστερό, όχι στον δεξιό)
`^`	συμμετρική διαφορά (στον έναν αλλά όχι και στους δύο)
`!`	συμπλήρωμα (όλα εκτός από)

Οι λευκοί χαρακτήρες μέσα σε `(?[...])` αγνοούνται, οπότε οι τελεστές διαβάζονται ως αριθμητικοί.

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

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

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

Η κατασκευή είναι πιο χρήσιμη όταν συνδυάζονται ιδιότητες Unicode που επικαλύπτονται. Χωρίς αυτήν, οι ίδιες εκφράσεις θα απαιτούσαν εκτεταμένες διεκδικήσεις περιβάλλοντος ή λογική εκτός μοτίβου.

Το `(?[...])` είναι το ίδιο μια κλάση χαρακτήρων - καταναλώνει έναν χαρακτήρα και μπορεί να ποσοδεικτηθεί:

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

Επιφύλαξη: το `(?[...])` έχει τον δικό του μικρό αναλυτή μέσα στον αναλυτή regex. Μέσα του, αναγνωρίζονται μόνο συγκεκριμένοι τελεστές και τελεσταίοι. Τα λάθη παράγουν συγκεκριμένα σφάλματα κατά τη μεταγλώττιση, κάτι που με τη σειρά του σημαίνει ότι η *αυστηρή λειτουργία* (`use re 'strict'`) πιάνει περισσότερες κακές εκφράσεις κλάσεων όταν χρησιμοποιείτε την εκτεταμένη μορφή.

Η αρνημένη κλάση κερδίζει το `.*?`

Ένα σύνηθες μοτίβο για νεοφερμένους: χρησιμοποιήστε `.*?` (μη άπληστη `.`) για να ταιριάξετε τα πάντα μέχρι ένα διαχωριστικό, όπως `<.*?>`. Το μοτίβο δουλεύει στις εισόδους που το δοκιμάσατε· σε αντιπαραθετική είσοδο όχι.

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

Στη δεύτερη περίπτωση η μηχανή ταίριαξε πρώτα το `<a>`, μετά χρειαζόταν `foo` αλλά βρήκε ένα διάστημα. Υπό αυτή την πίεση οπισθοχώρησης το μη άπληστο `.+?` αναγκάστηκε να επεκταθεί, καταναλώνοντας ευχαρίστως το `>` του `<a>` και το διάστημα μέχρι να ευθυγραμμιστεί το `foo`. Η αρνημένη κλάση χαρακτήρων δεν μπορεί να υποχωρήσει με αυτόν τον τρόπο:

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

Δύο λόγοι να προτιμάτε το `[^>]+` έναντι του `.+?` οποτεδήποτε το διαχωριστικό είναι μεμονωμένος χαρακτήρας:

1. **Ορθότητα**: η αρνημένη κλάση είναι σκληρό φράγμα· η μη άπληστη μορφή είναι προτίμηση.
2. **Απόδοση**: μια αρνημένη κλάση συμμετέχει στη βελτιστοποίηση απλής επανάληψης· το `.+?` όχι (η μηχανή πρέπει να βγαίνει από τον εσωτερικό βρόχο σε κάθε επανάληψη για να ελέγξει τι ακολουθεί). Σε μακριές εισόδους αυτό έχει σημασία.

Επεξεργασμένο παράδειγμα: μια regex διεύθυνσης IP

Η κανονική άσκηση «ειδικότητα έναντι πολυπλοκότητας». Πέντε επαναλήψεις, κάθε μία διορθώνει μια κατηγορία ασάφειας:

1. Αφελής. «Τέσσερις ομάδες ψηφίων χωρισμένες με τελεία.»

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

Ταιριάζει ευχαρίστως με `and then.....?` - κάθε ομάδα είναι προαιρετική, το μοτίβο ικανοποιείται από τέσσερις τελείες και τίποτα άλλο.

2. Απαιτήστε ψηφία. Αγκυρώστε το μοτίβο.

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

Ταιριάζει με `1234.5678.9101112.131415`. Κάθε ομάδα έχει ψηφία αλλά χωρίς άνω όριο πλήθους.

3. Περιορίστε το πλήθος ψηφίων, κακώς.

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

Ταιριάζει με `192.168.001.001` αλλά απορρίπτει το `1.2.3.4` - τα αρχικά μηδενικά δεν γράφονται πάντα.

4. Επιτρέψτε 1 έως 3 ψηφία.

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

Τώρα ταιριάζει με `1.2.3.4` και απορρίπτει το `1234.5.6.7`. Αλλά ταιριάζει επίσης με `999.999.999.999` - πέρα από το εύρος 0–255.

5. Σωστό ως προς το εύρος.

```
/^(?: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;
```

Κάθε ομάδα είναι μία από: `25[0-5]` (250–255), `2[0-4]\d` (200–249), ή `[01]?\d\d?` (0–199 σε διάφορες μορφές). Το μοτίβο τώρα ταιριάζει με *ακριβώς* τις συμβολοσειρές που ονομάζουν μια συντακτικά έγκυρη διεύθυνση IPv4.

Το μάθημα δεν είναι «αποστηθίστε αυτή τη regex». Είναι η *εξέλιξη*: κάθε επανάληψη σφίγγει ένα συγκεκριμένο είδος ασάφειας. Η σωστή regex για ένα πρόβλημα είναι αυτή που δέχεται ακριβώς τις σωστές εισόδους και απορρίπτει όλες τις υπόλοιπες, και φτάνετε εκεί μόνο ρωτώντας τι πραγματικά επέτρεπε η *προηγούμενη* regex.

Μεταξύ μηχανών: συντομογραφικές κλάσεις

Οι συντομογραφίες `\d`, `\w`, `\s` δεν είναι φορητές. Το κεφάλαιο *cross-engine* έχει τον πλήρη πίνακα· οι σχετικές γραμμές αποσπασμένες:

Συντομογ	Perl 5.44 (default)	PCRE	Emacs	POSIX BRE / ERE	RE2 / Go (προεπιλογή)
`\d`	ASCII (ή Unicode υπό `/u`)	ASCI	OXI (χρησιμοποιήστε `[0-9]` ή `[[:digit:]]`)	OXI	ASCII· Unicode υπό `(?u)`
`\w`	ASCII ή Unicode	ASCI	ναι (καθοδηγείται από τον πίνακα σύνταξης)	OXI	ASCII· Unicode υπό `(?u)`
`\s`	ASCII ή Unicode	ASCI	ναι	OXI	ASCII· Unicode υπό `(?u)`
όριο λέξης `\b`	ναι	ναι	ναι (`\<`/`\>` παραδοσιακά)	OXI	ναι
`\h`, `\v`	ναι	ναι	OXI	OXI	OXI

Δύο πράγματα να αφομοιώσετε:

1. Το POSIX BRE και το ERE στερούνται εντελώς των `\d`, `\w`, `\s`. Τα φορητά σενάρια κελύφους χρησιμοποιούν `[0-9]`, `[[:alnum:]_]`, `[[:space:]]`.
2. Ο Emacs έχει `\w` και `\s` αλλά όχι `\d`. Η σύνταξη `\s` του Emacs *ακολουθείται από* έναν χαρακτήρα κλάσης σύνταξης (`\s-` για λευκό χαρακτήρα, `\sw` για λέξη) - μοναδικό στον Emacs.

Οι κλάσεις σε αγκύλες POSIX (`[[:digit:]]`, `[[:alpha:]]`, ...) είναι η καθολικά φορητή γραφή: κάθε μηχανή στη σύγκριση τις αναγνωρίζει.

Μια χρήσιμη συνήθεια

Οι ονομαστικές και οι συντομογραφικές κλάσεις είναι σχεδόν πάντα σαφέστερες από τα ρητά εύρη. Το `\d{4}-\d{2}-\d{2}` διαβάζεται· το `[0-9]{4}-[0-9]{2}-[0-9]{2}` χρειάζεται μια στιγμή. Χρησιμοποιείτε τα εύρη μόνο όταν έχετε συγκεκριμένο λόγο - συνήθως απόδοση σε καυτό βρόχο, ή σκόπιμο περιορισμό σε ASCII.

Δείτε επίσης

- Το κεφάλαιο *unicode* - `\p{...}`, `\P{...}`, `\X`, οι τροποποιητές συνόλου χαρακτήρων `/a`, `/u`, `/l`, `/d`.
- Το κεφάλαιο *anchors and assertions* - `\b` εκτός κλάσης.
- Το κεφάλαιο *cross-engine* - πλήρης πίνακας υποστήριξης συντομογραφιών μεταξύ μηχανών.
- `m` - ο τελεστής αντιστοίχισης.
- `qr` - μεταγλώττιση μοτίβου για επαναχρησιμοποίηση.

Άγκυρες και διεκδικήσεις

Οι άγκυρες και οι διεκδικήσεις ταιριάζουν με *θέσεις* αντί για χαρακτήρες. Ελέγχουν πού μέσα στη συμβολοσειρά βρίσκεστε, όχι τι υπάρχει εκεί. Μια απλή κλάση χαρακτήρων καταναλώνει έναν χαρακτήρα· μια άγκυρα δεν καταναλώνει κανέναν.

Γι" αυτόν τον λόγο ονομάζονται *διεκδικήσεις μηδενικού πλάτους* - έχουν πλάτος μηδέν, αλλά διεκδικούν ότι πρέπει να ισχύει μια ιδιότητα.

Αρχή και τέλος συμβολοσειράς

Το ^ ταιριάζει στην αρχή της συμβολοσειράς. Το $ ταιριάζει στο τέλος, ή ακριβώς πριν από μια τελική νέα γραμμή.

```
"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
```

Χρησιμοποιημένα μαζί, τα ^…$ αναγκάζουν το μοτίβο να καλύψει ολόκληρη τη συμβολοσειρά:

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

Για μια κυριολεκτική συμβολοσειρά, το `$s eq "bert"` είναι ταχύτερο και σαφέστερο. Το ^…$ δικαιολογεί τη χρήση του μόνο όταν το μεσαίο τμήμα χρησιμοποιεί πραγματικά χαρακτηριστικά regex.

Απόλυτες έναντι σχετικών ως προς γραμμή αγκύρων

Υπό τον τροποποιητή /m, τα ^ και $ αγκυρώνουν σε κάθε όριο γραμμής μέσα στη συμβολοσειρά, όχι μόνο στα εξωτερικά άκρα:

```
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
```

Για τα απόλυτα άκρα *ανεξαρτήτως του* /m, χρησιμοποιήστε τις ειδικές άγκυρες:

- \A - απόλυτη αρχή συμβολοσειράς.
- \z - απόλυτο τέλος συμβολοσειράς.
- \Z - τέλος συμβολοσειράς, ή ακριβώς πριν από μια τελική νέα γραμμή. Παρόμοιο με το $ αλλά ανεπηρέαστο από το /m.

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

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
$x =~ /girl$/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
```

Πρακτικός κανόνας: \A και \z όταν εννοείτε ολόκληρη τη συμβολοσειρά· ^ και $ όταν εννοείτε γραμμές (συνήθως με /m). Τα \A και \z είναι επίσης ταχύτερα: η μηχανή ξέρει ότι περιορίζουν τη θέση, και παραλείπει τις επανειλημμένες προσπάθειες bump-along που επιτρέπει το ^.

Οι τέσσερις λειτουργίες νέας γραμμής μπορούν να συνοψιστούν σε έναν πίνακα. Επιλέξτε εκείνη που ταιριάζει με αυτό που πρέπει να σημαίνουν τα ^, $, και . για την είσοδό σας:

Λειτουργί	Το ^ ταιριάζει σε	Το $ ταιριάζει σε	Ταιριάζει το . με \n;
προεπιλογ	αρχή συμβολοσειράς μόνο	τέλος συμβολοσειράς (και πριν την τελική \n)	όχι
/m	αρχή κάθε γραμμής	τέλος κάθε γραμμής	όχι
/s	αρχή συμβολοσειράς μόνο	τέλος συμβολοσειράς (και πριν την τελική \n)	ναι
/sm	αρχή κάθε γραμμής	τέλος κάθε γραμμής	ναι

Τα \A, \Z, \z είναι ανεπηρέαστα από οποιονδήποτε από τους δύο τροποποιητές.

Όρια λέξεων

Το \b ταιριάζει με μια θέση ανάμεσα σε έναν χαρακτήρα λέξης (\w) και έναν χαρακτήρα μη λέξης (\W), ή ανάμεσα σε οποιονδήποτε από τους δύο και την άκρη της συμβολοσειράς. Δεν καταναλώνει κανέναν χαρακτήρα.

```
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
```

Το \B είναι η άρνηση - μια θέση που *δεν* είναι όριο λέξης.

Το \b εκτός […] είναι όριο λέξης. Το \b *μέσα* στα […] είναι ο χαρακτήρας *backspace*, \x08. Αυτή η διπλή σημασία είναι η συχνότερη παγίδα στη σύνταξη regex. Πάντοτε διαβάζετε το περιβάλλον του \b προτού διαβάσετε τι κάνει.

Το \b και στοιχεία που ξεκινούν με \W - η παγίδα του $3.75

Το \b απαιτεί μια μετάβαση ανάμεσα σε χαρακτήρες λέξης και μη λέξης. Αν και οι δύο πλευρές της θέσης είναι χαρακτήρες μη λέξης, δεν υπάρχει όριο. Αυτό πιάνει όσους χτίζουν μοτίβα μέσω παρεμβολής:

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

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

Γιατί; Αφού επεξεργαστεί το \Q$item\E, το μοτίβο είναι \b\$3\.75\b. Η μηχανή αναζητά μια θέση αμέσως *πριν* το $ όπου η μία πλευρά της θέσης είναι χαρακτήρας λέξης. Ο χαρακτήρας πριν το $ στην είσοδο είναι ένα διάστημα (μη λέξη), και το ίδιο το $ είναι επίσης μη λέξη. Δεν υπάρχει μετάβαση λέξης/μη λέξης· το \b αποτυγχάνει.

Για να αγκυρώσετε την αρχή του $item ανεξαρτήτως του πρώτου του χαρακτήρα:

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

Χρησιμοποιήστε άγκυρες άκρων συμβολοσειράς ή λευκού χαρακτήρα όταν το παρεμβαλλόμενο περιεχόμενο μπορεί να ξεκινά με χαρακτήρα μη λέξης. Το \b είναι για περιβάλλοντα όπου *ξέρετε* ότι το περιβάλλον κείμενο μοιάζει με λέξη.

Παραλλαγές ορίου λέξης Unicode

Το απλό \b ακολουθεί τον ορισμό \w/\W, ο οποίος υπό Unicode καλύπτει γράμματα, ψηφία, σημάδια, και συνδετική στίξη. Το πραγματικό κείμενο - φυσική γλώσσα με αποστρόφους, παύλες, αριθμούς με κόμματα - απαιτεί περισσότερη απόχρωση. Η Perl παρέχει τέσσερις σημασιολογικές παραλλαγές ορίου:

Όριο	Σημασία
\b{wb}	Όριο λέξης Unicode - χειρίζεται don't, state-of-the-art
\b{sb}	όριο πρότασης
\b{lb}	αλλαγή γραμμής (κατάλληλο για αναδίπλωση γραμμών)
\b{gcb}	όριο συμπλέγματος γραφημάτων (ίδιο αποτέλεσμα με το \X)

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

Για επεξεργασία φυσικής γλώσσας, τα \b{wb} και \b{sb} είναι σχεδόν πάντα αυτό που θέλετε. Το απλό \b είναι για περιβάλλοντα αναγνωριστικών ASCII.

\G - η θέση της τελευταίας αντιστοιχίας

Το \G αγκυρώνει στη θέση όπου τελείωσε η προηγούμενη επιτυχημένη αντιστοιχία /g στην ίδια συμβολοσειρά. Αυτό είναι που κάνει αξιόπιστη την τμηματοποίηση με /g σε βρόχους while.

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

Χωρίς \G, το ίδιο μοτίβο θα ξανασάρωνε από όπου ταίριαξε πρώτα, παραλείποντας χαρακτήρες που δεν χωρούσαν - χάνοντας σιωπηλά δεδομένα. Ο τροποποιητής /gc διατηρεί τη θέση σε αποτυχία· καλύπτεται στο κεφάλαιο *modifiers*.

Γιατί έχει σημασία το \G: η επιλογή συγχρονισμού ή παράλειψης

Το επεξεργασμένο παράδειγμα του Friedl. Βρείτε κάθε πεντάψηφιο ZIP code που ξεκινά με 44 σε μια κολλημένη συμβολοσειρά:

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

Χωρίς \G, το bump-along της μηχανής βρίσκει ευχαρίστως τα 44272 και 44106 σωστά *καθώς επίσης και* το 44272 από διαφορετική θέση εκκίνησης οποτεδήποτε απορρίπτεται μια προηγούμενη αντιστοιχία. Η έξοδος είναι «η σωστή απάντηση συν επιπλέον λάθος».

Με \G σε κάθε επανάληψη, η μηχανή δεν μπορεί να κάνει bump-along σε αποτυχία - πρέπει να συνεχίσει *ακριβώς* εκεί που τελείωσε η τελευταία αντιστοιχία. Μια αποτυχία τερματίζει τον βρόχο:

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

Το \G ουσιαστικά απενεργοποιεί το bump-along, κάτι που απαιτεί η συγχρονισμένη τμηματοποίηση. Ο γενικός κανόνας: **αν η ορθότητα του βρόχου σας εξαρτάται από το ότι κάθε αντιστοιχία είναι γειτονική με την τελευταία, χρειάζεστε το \G**.

Το \G διαβάζει την τιμή που επιστρέφει η pos($string). Η ρητή τοποθέτηση του pos επανεκκινεί την άγκυρα. Στην πρώτη επανάληψη ενός βρόχου /g (ή σε οποιοδήποτε μοτίβο δεν έχει ακόμη αντιστοιχιστεί με αυτή τη συμβολοσειρά), το \G ισοδυναμεί με \A.

Πρόσθια διεκδίκηση

Η πρόσθια διεκδίκηση ελέγχει τι ακολουθεί χωρίς να το καταναλώνει. Η θετική πρόσθια διεκδίκηση είναι (?=…):

```
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
```

Το `(?!…)` είναι αρνητική πρόσθια διεκδίκηση: αντιστοιχεί μόνο αν το εσωτερικό μοτίβο *δεν* ισχύει. Μηδενικού πλάτους, όπως ακριβώς τα ^ και $.

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

Επεξεργασμένο παράδειγμα πρόσθιας διεκδίκησης: ψηφία που δεν ακολουθούνται από τελεία

Μια ερώτηση που ακούγεται φυσική αλλά είναι πιο δύσκολη από όσο φαίνεται: «ταίριασε ακολουθίες ψηφίων που δεν ακολουθούνται από τελεία.»

Η πρώτη απόπειρα: `\d+(?!\.)`. Εφαρμόστε τη στο `OH 44272`:

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

Αυτό φαίνεται σωστό, αλλά για λάθος λόγο. Το άπληστο `\d+` άρπαξε πρώτα ολόκληρη την ακολουθία ψηφίων 44272· ο επόμενος χαρακτήρας είναι το τέλος της συμβολοσειράς (όχι `.`), οπότε το `(?!\.)` πέτυχε αμέσως - η πρόσθια διεκδίκηση τυχαία ίσχυσε χωρίς να χρειαστεί ποτέ η μηχανή να υποχωρήσει. Η *διαισθητική* ανάγνωση «η ακολουθία ψηφίων τελειώνει με κάτι άλλο πέρα από τελεία» ικανοποιείται για την περίπτωση τελικών ψηφίων, αλλά το επόμενο παράδειγμα δείχνει τι συμβαίνει όταν δεν ικανοποιείται.

Τώρα εφαρμόστε τη στο `4423.45`:

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

Η πρώτη απόπειρα της μηχανής: το `\d+` ταιριάζει με 4423. Το `(?!\.)` ελέγχει τον επόμενο χαρακτήρα: είναι `.`, η αρνητική πρόσθια διεκδίκηση αποτυγχάνει. Η μηχανή οπισθοχωρεί: το `\d+` ταιριάζει με 442, ο επόμενος χαρακτήρας είναι 3, όχι `.`, επιτυχία. Το μοτίβο ταιριάζει με 442 - *όχι* με ολόκληρη την ακολουθία ψηφίων πριν την τελεία. Ο άπληστος ποσοδείκτης ήταν διατεθειμένος να υποχωρήσει για να πετύχει η πρόσθια διεκδίκηση.

Η διόρθωση: απαιτήστε από την πρόσθια διεκδίκηση να απορρίπτει επίσης άλλα ψηφία, ώστε η ακολουθία να μην μπορεί να υποχωρήσει σε «ψηφίο ακολουθούμενο από ψηφίο».

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

Το `\d+(?![\d.])` διαβάζεται ως «ταίριασε ακολουθία ψηφίων που δεν ακολουθείται ούτε από άλλο ψηφίο ούτε από τελεία». Ο άπληστος ποσοδείκτης ακόμη επεκτείνεται μεγιστοτικά, αλλά η πρόσθια διεκδίκηση τώρα αρνείται να πετύχει στη μέση μιας ακολουθίας ψηφίων, οπότε οι μόνες θέσεις που ταιριάζουν είναι τα πραγματικά τέλη ακολουθιών ψηφίων.

Το μάθημα γενικεύεται: **μια πρόσθια διεκδίκηση που αποκλείει μόνο έναν χαρακτήρα μπορεί να ηττηθεί από έναν άπληστο ποσοδείκτη που υποχωρεί**. Πάντα συμπεριλαμβάνετε τους χαρακτήρες του *ατόμου* με το οποίο ταιριάζει ο ποσοδείκτης στην άρνηση της πρόσθιας διεκδίκησης.

Μια σχετική περίπτωση: αρχική αρνητική πρόσθια διεκδίκηση και το bump-along.

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

Η πρόσθια διεκδίκηση `(?!cat)` απορρίπτει τη θέση 0 (όπου θα ταίριαζε το `cat`). Η μηχανή κάνει bump-along: η θέση 1 ξεκινά με `attle`, όπου το `(?!cat)` πετυχαίνει, και το `\w+` ταιριάζει με `attle`.

Για να επιβάλετε «η πρώτη λέξη που δεν αρχίζει με `cat`», αγκυρώστε την πρόσθια διεκδίκηση σε όριο λέξης:

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

Τώρα η πρόσθια διεκδίκηση εφαρμόζεται σε κάθε όριο λέξης, που στο `cattle` είναι μόνο η θέση 0.

Οπίσθια διεκδίκηση

Η οπίσθια διεκδίκηση ελέγχει τι προηγήθηκε. Η θετική οπίσθια διεκδίκηση είναι `(?<=…)`:

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

Η αρνητική οπίσθια διεκδίκηση είναι `(?<!…)`:

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

Η σύγχρονη Perl υποστηρίζει οπίσθια διεκδίκηση μεταβλητού μήκους από 1 έως 255 χαρακτήρες. Μοτίβα όπως `(?<=cat|kitten)` είναι μια χαρά. Μοτίβα όπου η ίδια η οπίσθια διεκδίκηση περιέχει συλλαμβάνουσες ομάδες παράγουν πειραματική προειδοποίηση (τα περιεχόμενα των συλλήψεων δεν είναι πλήρως καθορισμένα όταν η οπίσθια διεκδίκηση έχει μεταβλητό μήκος).

Επειδή η οπίσθια διεκδίκηση πρέπει να τελειώσει πριν την τρέχουσα θέση, το μέγιστο μήκος της περιορίζεται σε 255 *χαρακτήρες* υπό προεπιλεγμένη σημασιολογία. Υπό `/i`, μερικοί χαρακτήρες πτυσσονται σε ακολουθίες πολλών χαρακτήρων (το ß σε `ss`), κάτι που μετράει το *αναπτυγμένο* μήκος προς το όριο των 255 - άρα μια οπίσθια διεκδίκηση 127 χαρακτήρων που περιέχει ß είναι μια χαρά, αλλά 128 τέτοιοι χαρακτήρες όχι.

Μια πρακτική παράκαμψη για μακρύτερη οπίσθια διεκδίκηση: χρησιμοποιήστε το `\K` (επόμενη ενότητα), το οποίο δεν έχει όριο μήκους.

`\K` - διατήρηση αριστερού

Το `\K` είναι η εναλλακτική της οπίσθιας διεκδίκησης. Ό,τι ταίριαξε *πριν* το `\K` εξαιρείται από το `$&`, αλλά απαιτείται να έχει ταιριάξει. Πρακτικά: «ταίριαξε αυτό το πρόθεμα, αλλά μην το συμπεριλάβεις στην έξοδο.»

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

Ισοδύναμο με το `(?<=the )cat`, αλλά:

- Χωρίς όριο μήκους. Το `\K` λειτουργεί μετά από οποιοδήποτε πρόθεμα.
- Συχνά ουσιωδώς ταχύτερο από το `(?<=…)`, επειδή η μηχανή δεν χρειάζεται να κοιτάξει πίσω - αντ” αυτού ξεχνάει την αρχή της αντιστοιχίας.
- Ιδιαιτέρως χρήσιμο σε αντικατάσταση: το `s/foo\Kbar/QUUX/` είναι καθαρότερο από το `s/(foo)bar/$1QUUX/`.

Το \K ξεχνάει μόνο τα $& και @-/@+[0]· οι συλλαμβάνουσες ομάδες πριν το \K εξακολουθούν να ορίζονται ως συλληφθείσες. Η κατασκευή μπορεί να εμφανιστεί μέσα σε άλλες διεκδικήσεις περιβάλλοντος, αν και η συμπεριφορά εκεί περιγράφεται ως «προς το παρόν μη καλώς ορισμένη» - η συντηρητική χρήση είναι στο ανώτατο επίπεδο μιας αντιστοιχίας ή αντικατάστασης.

Ψευδώνυμα μακράς μορφής για διεκδικήσεις περιβάλλοντος

Κάθε κατασκευή διεκδίκησης περιβάλλοντος έχει μια γραφή τύπου ρήματος. Τα ψευδώνυμα μακράς μορφής διαβάζονται πιο καθαρά σε μοτίβα που ήδη χρησιμοποιούν (*VERB:…) για έλεγχο οπισθοχώρησης:

Τυπική μορφή	Μορφή ρήματος (σύντομη)	Μορφή ρήματος (μακρά)
(?=…)	(*pla:…)	(*positive_lookahead:…)
(?!…)	(*nla:…)	(*negative_lookahead:…)
(?<=…)	(*plb:…)	(*positive_lookbehind:…)
(?<!…)	(*nlb:…)	(*negative_lookbehind:…)

Οι μορφές ρήματος είναι ακριβώς ισοδύναμες με τις τυπικές μορφές. Γίνονται δεκτές· δεν είναι ιδιωματικές σε χειρόγραφη Perl. Θα τις δείτε σε αυτόματα παραγόμενα μοτίβα και σε μεταφερόμενα από PCRE2.

Συνδυασμός διεκδικήσεων περιβάλλοντος με split

Οι άγκυρες και οι διεκδικήσεις περιβάλλοντος επιτρέπουν στο split να χωρίζει μια συμβολοσειρά σε αόρατες θέσεις αντί για ορατούς χαρακτήρες:

```
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")
```

Η δεύτερη και η τρίτη εναλλακτική ταιριάζουν *ανάμεσα* σε χαρακτήρες - χωρίς κατανάλωση. Είναι νόμιμοι διαχωριστές split επειδή το split χωρίζει ευχαρίστως σε αντιστοιχία μηδενικού πλάτους.

Δύο διεκδικήσεις μηδενικού πλάτους σε παράθεση συντίθενται με AND

Ένα μικρό αλλά διαφωτιστικό γεγονός: όταν δύο διεκδικήσεις περιβάλλοντος εμφανίζονται δίπλα-δίπλα σε ένα μοτίβο, πρέπει και οι δύο να ισχύουν στην ίδια θέση. Αυτό δουλεύει όπως αναμένεται:

```
$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'.
```

Και τα (?=\d) και (?!123) εφαρμόζονται στην ίδια θέση· η μηχανή αντιμετωπίζει τη σύζευξή τους ως την απαίτηση. Χωρίς το (?=\d), η αρνητική πρόσθια

διεκδίκηση `(?!123)` μπορεί να ικανοποιηθεί από *οποιαδήποτε* συνέχεια εκτός 123 - συμπεριλαμβανομένου του τέλους συμβολοσειράς ή ενός μη ψηφίου, κανένα από τα οποία δεν είναι αυτό που εννοούσε ο συγγραφέας.

Αυτό γενικεύει τη διατύπωση του Friedl: η παράθεση σε μια regex πάντα σημαίνει AND, εκτός όταν γράφεται με `|`. Το `/ab/` σημαίνει «a AND (μετά) b», όπως ακριβώς το `/^$/` σημαίνει «αρχή AND τέλος» - με τη διαφορά ότι το AND είναι σε διαφορετικές θέσεις για το ab και σε μία θέση για τις διεκδικήσεις περιβάλλοντος.

Μεταξύ μηχανών: άγκυρες και διεκδικήσεις περιβάλλοντος

Το κεφάλαιο *cross-engine* έχει τον πλήρη πίνακα· οι σχετικές γραμμές αποσπασμένες.

Άγκυρες και χειρισμός νέας γραμμής

Θέμα	Perl 5.44	PCRE	Emacs	POSIX BRE / ERE	RE2 / Go
^ αρχή συμβολοσειράς	ναι	ναι	αρχή ενταμιευτή	ναι	ναι
^ μετά από \n σε λειτουργία πολλαπλών γραμμών	/m	(?m)	πάντα (γραμμοπροσανατ	μη προσδιορισ	(?m)
$ τέλος συμβολοσειράς	ναι	ναι	τέλος ενταμιευτή	ναι	ναι
$ πριν την τελική \n	ναι	ναι	όχι	ποικίλλει	ναι
\A / \z	ναι	ναι	\` / \'	όχι	ναι
\Z	ναι	ναι	όχι	όχι	όχι (χρησιμοποιήστε \z)
\G	ναι	ναι	όχι	όχι	όχι

Διεκδικήσεις περιβάλλοντος και ατομικές κατασκευές

Χαρακτηριστικό	Perl 5.44	PCRE2	Emacs	POSIX	RE2 / Go
Πρόσθια διεκδίκηση (?=…) / (?!…)	ναι	ναι	ΟΧΙ	ΟΧΙ	ΟΧΙ
Οπίσθια διεκδίκηση σταθερού πλάτους	ναι	ναι	ΟΧΙ	ΟΧΙ	ΟΧΙ
Οπίσθια διεκδίκηση μεταβλητού μήκους	ναι (πειραμ.)	ναι	ΟΧΙ	ΟΧΙ	ΟΧΙ
\K	ναι	ναι	ΟΧΙ	ΟΧΙ	ΟΧΙ
Ατομική ομάδα (?>…)	ναι	ναι	ΟΧΙ	ΟΧΙ	ΟΧΙ

Στην ουσία, μόνο η PCRE2 (και άλλες μηχανές προερχόμενες από την Perl εκτός αυτού του πίνακα - Java, Python, .NET) υποστηρίζει διεκδικήσεις περιβάλλοντος. Τα εργαλεία POSIX και η RE2 / Go απλώς δεν τις διαθέτουν. Μοτίβα που βασίζονται σε διεκδικήσεις περιβάλλοντος δεν μεταφέρονται μέσω αυτού του χάσματος.

Σύνοψη

Άγκυρα	Ταιριάζει με
`^`	αρχή συμβολοσειράς, ή αρχή γραμμής υπό `/m`
`$`	τέλος συμβολοσειράς (πριν την τελική `\n`), ή γραμμής υπό `/m`
`\A`	απόλυτη αρχή συμβολοσειράς
`\z`	απόλυτο τέλος συμβολοσειράς
`\Z`	τέλος συμβολοσειράς ή πριν την τελική `\n`, αγνοεί το `/m`
`\b`	όριο λέξης
`\B`	μη όριο λέξης
`\b{wb}`	όριο λέξης Unicode
`\b{sb}`	όριο πρότασης
`\b{lb}`	αλλαγή γραμμής
`\b{gcb}`	όριο συμπλέγματος γραφημάτων
`\G`	τέλος προηγούμενης αντιστοιχίας `/g` (ή `\A` αν δεν υπάρχει)
`(?=…)`	θετική πρόσθια διεκδίκηση
`(?!…)`	αρνητική πρόσθια διεκδίκηση
`(?<=…)`	θετική οπίσθια διεκδίκηση
`(?<!…)`	αρνητική οπίσθια διεκδίκηση
`\K`	διατήρηση αριστερού (ξέχνα το πρόθεμα από το `$&`)

Δείτε επίσης

- Το κεφάλαιο *modifiers* - `/m`, `/s`, `/g`, `/c`.
- Το κεφάλαιο *unicode* - `\b{wb}`, `\b{sb}`, `\b{lb}`, `\b{gcb}` και τι σημαίνουν.
- Το κεφάλαιο *performance* - ατομικές ομάδες `(?>…)`, ειδικά ρήματα ελέγχου οπισθοχώρησης.
- Το κεφάλαιο *cross-engine* - πλήρης πίνακας συμβατότητας αγκύρων και διεκδικήσεων περιβάλλοντος.
- `split` - διαχωρισμός σε θέσεις μηδενικού πλάτους.
- `pos` - ανάγνωση ή ορισμός της θέσης όπου αγκυρώνει το `\G`.

Ποσοδείκτες

Ένας ποσοδείκτης λέει «αυτό το προηγούμενο πράγμα, N φορές». Συνδέεται με έναν μεμονωμένο χαρακτήρα, μια κλάση χαρακτήρων, ή μια ομάδα, και μετατρέπει μια αντιστοιχία ενός χαρακτήρα σε αντιστοιχία πολλών.

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

Οι βασικοί ποσοδείκτες

Ποσοδείκτης	Σημασία
a?	0 ή 1 φορά
a*	0 ή περισσότερες φορές
a+	1 ή περισσότερες φορές
a{n}	ακριβώς n φορές
a{n,m}	τουλάχιστον n, το πολύ m
a{n,}	τουλάχιστον n
a{,m}	το πολύ m

Τα *, + και ? είναι ακριβή συνώνυμα των {0,}, {1,} και {0,1}. Οι μορφές με άγκιστρα χρησιμοποιούνται κυρίως όταν τα n και m είναι συγκεκριμένοι μικροί αριθμοί· οι γυμνές μορφές διαβάζονται πιο φυσικά για «οποιοδήποτε πλήθος από αυτά».

Τα άνω όρια n και m δεν μπορούν να υπερβούν μια ενσωματωμένη στη μηχανή σταθερά (συνήθως 65534 - το πραγματικό όριο εμφανίζεται στο σφάλμα αν το υπερβείτε). Για «οποιονδήποτε λογικό αριθμό» αυτό είναι υπεραρκετό· μοτίβα που χτυπούν στο όριο είναι συνήθως αυτόματα παραγόμενα, όχι χειρόγραφα.

Οι λευκοί χαρακτήρες μέσα στα {…} είναι ανεκτοί αλλά όχι υποχρεωτικοί. Τα a{2, 4} και a{2,4} είναι ταυτόσημα.

```
/[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
```

Ο ποσοδείκτης εφαρμόζεται στο άτομο αμέσως πριν από αυτόν:

- ab? - a και μετά προαιρετικό b.
- (ab)? - προαιρετικό ab.
- [ab]? - προαιρετικό a ή b.

Πρώτα ομάδα ή κλάση, μετά ποσοδείκτης.

Άπληστοι από προεπιλογή

Οι βασικοί ποσοδείκτες είναι *άπληστοι*: αρπάζουν όσο τους επιτρέπει η συμβολοσειρά και υποχωρούν μόνο αν το υπόλοιπο μοτίβο δεν μπορεί να ταιριάξει αλλιώς. Σκεφτείτε:

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

Εδώ το .* στη μέση δουλεύει όπως θα περιμένατε - σταματά στο cat επειδή το cat είναι το μόνο σημείο όπου το υπόλοιπο μοτίβο μπορεί να ταιριάξει. Τώρα συγκρίνετε:

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

Υπάρχουν *δύο* σημεία όπου εμφανίζεται το `at` - στο τέλος του `cat` και στο τέλος του `hat`. Το πρώτο `.*` αρπάζει όσο το δυνατόν περισσότερα αφήνοντας ακόμη χώρο για το `at` κάπου, οπότε αρπάζει μέχρι το *τελευταίο* `at`.

Ο άπληστος υποχωρεί μόνο όσο πρέπει

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

Το άπληστο `.*` ταίριαξε πρώτα ολόκληρη τη συμβολοσειρά. Για να ταιριάξει το `[0-9]+`, έπρεπε να επιστρέψει αρκετούς χαρακτήρες ώστε να αποκαλύψει τουλάχιστον ένα ψηφίο. Επέστρεψε ακριβώς έναν, αφήνοντας το 4 για σύλληψη. Η διαίσθηση ότι το `.*` «σταματά στα ψηφία» είναι λανθασμένη - αρπάζει τα πάντα, και μετά η μηχανή ρωτά πόσο λίγα μπορεί να επιστρέψει.

Απληστία έναντι αντιστοίχισης

Η μηχανή δεν ταιριάζει πάντα με ό,τι υποδεικνύει η άπληστη ερμηνεία:

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

Η πρώτη απόπειρα αποτυγχάνει: το `\d\d` μπορεί να ταιριάξει με 27, αλλά ο επόμενος χαρακτήρας είναι `.`, οπότε καμία διακλάδωση του `[1-9]?` δεν αφήνει το `\d+` να προχωρήσει. Ο κανόνας «κερδίζει η προγενέστερη θέση» μετακινεί την αρχή μέχρι το `\d\d` να προσγειωθεί στο 62. Εκεί, το `[1-9]?` δοκίμασε άπληστα το 5, αλλά τότε το `\d+` δεν είχε εναπομένον ψηφίο - οπότε επέστρεψε το 5 και πήγε σε κενό, αφήνοντας το 5 για το `\d+`. Οι προαιρετικοί ποσοδείκτες προτιμούν τη *μακρύτερη* εναλλακτική (παίρνοντας τον χαρακτήρα) αλλά θα υποχωρήσουν αν χρειαστεί· ο κανόνας ταξινόμησης «κερδίζει η προγενέστερη θέση, μετά η μακρύτερη σε εκείνη τη θέση» περιορίζει μόνο *ολόκληρη την αντιστοιχία*, όχι τις επιμέρους συλλήψεις της.

Αρχές της αντιστοίχισης

Με άπληστους ποσοδείκτες, η μηχανή ακολουθεί τέσσερις αρχές με σειρά:

1. **Κερδίζει η προγενέστερη θέση.** Ολόκληρο το μοτίβο δοκιμάζεται ξεκινώντας από τη θέση 0, μετά τη 1, μετά τη 2, μέχρι να βρεθεί αντιστοιχία. Η προγενέστερη θέση εκκίνησης πάντα κερδίζει.
2. **Κερδίζει η πιο αριστερή εναλλακτική** (σε `a|b|c`).
3. **Οι άπληστοι ποσοδείκτες αρπάζουν όσο το δυνατόν περισσότερα** ενώ εξακολουθούν να επιτρέπουν στο υπόλοιπο μοτίβο να ταιριάξει.

4. **Ο πιο αριστερός άπληστος ποσοδείκτης παίρνει προτεραιότητα.** Αν δύο `.*` συναγωνίζονται για τους ίδιους χαρακτήρες, το πρώτο τους παίρνει.

Παραδείγματα:

```
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}.
```

Μη άπληστοι ποσοδείκτες

Η προσθήκη ? σε οποιονδήποτε ποσοδείκτη τον γυρίζει από *άρπαξε όσο το δυνατόν περισσότερα* σε *άρπαξε όσο το δυνατόν λιγότερα*.

Μη άπληστος	Σημασία
`a??`	0 ή 1, προτιμά 0
`a*?`	0 ή περισσότερες, προτιμά λιγότερες
`a+?`	1 ή περισσότερες, προτιμά λιγότερες
`a{n,m}?`	n έως m, προτιμά n

Η συνολική αντιστοιχία πρέπει ακόμη να επιτύχει, οπότε η μηχανή θα *επεκτείνει* τον μη άπληστο ποσοδείκτη ένα βήμα τη φορά μέχρι να επιτύχει.

```
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.
```

Ο μη άπληστος είναι συνήθως αυτό που θέλετε όταν σαρώνετε ανάμεσα σε διαχωριστικά:

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

Με άπληστο `.*` η αντιστοιχία θα κατάπινε το πρώτο `</b>` και θα σταματούσε στο τελικό `</i>`, καταστρέφοντας τη σύλληψη.

Όταν ο μη άπληστος χάνει από αρνημένη κλάση

Ένας σύνηθης ιδιωματισμός: `<.*?>` για αντιστοίχιση μιας ετικέτας. Ένα σύνηθες σφάλμα: το ίδιο μοτίβο, εφαρμοσμένο σε χαλασμένη είσοδο, επεκτείνει τον μη άπληστο ποσοδείκτη πέρα από περιεχόμενο που δεν θα έπρεπε. Η καθαρότερη μορφή χρησιμοποιεί αρνημένη κλάση χαρακτήρων:

```
"<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
```

Όταν απαιτείται μια λεκτική μονάδα μετά το `>`, η μη άπληστη μορφή *επεκτείνεται* υπό την πίεση οπισθοχώρησης: η μηχανή δοκίμασε πρώτα το `<a>`, δεν βρήκε `foo`, και ανάγκασε το `.+?` να μεγαλώσει πέρα από το `>` και παραπέρα. Η αρνημένη κλάση δεν μπορεί να υποχωρήσει με αυτόν τον τρόπο - η `[^>]` είναι σκληρό φράγμα, οπότε η μηχανή μετακινεί τη θέση εκκίνησης στο `<b>` αντί να καταπιεί ενδιάμεσα `>`.

Μια αρνημένη κλάση είναι επίσης ταχύτερη: συμμετέχει στο γρήγορο μονοπάτι απλής επανάληψης της μηχανής, το οποίο η μη άπληστη μορφή απενεργοποιεί.

Χρησιμοποιείτε `.+?` μόνο όταν καμία αρνημένη κλάση δεν μπορεί να εκφράσει την ίδια απαίτηση. Χρησιμοποιείτε `[^X]+` όποτε το διαχωριστικό είναι μεμονωμένος χαρακτήρας.

Κτητικοί ποσοδείκτες

Η προσθήκη `+` μετά από έναν ποσοδείκτη (όχι `?`) τον κάνει *κτητικό*: άπληστο, αλλά αρνείται να επιστρέψει οτιδήποτε σε οπισθοχώρηση. Μόλις ταιριάξει, αυτοί οι χαρακτήρες βγαίνουν εκτός παιχνιδιού για το υπόλοιπο μοτίβο.

Κτητικός	Σημασία
`a?+`	0 ή 1, χωρίς υποχώρηση
`a*+`	0 ή περισσότερες, χωρίς υποχώρηση
`a++`	1 ή περισσότερες, χωρίς υποχώρηση
`a{n,m}+`	n έως m, χωρίς υποχώρηση

Το όφελος είναι ταχύτητα - σε μοτίβα που *πρέπει* να αποτύχουν, οι κτητικοί ποσοδείκτες αποτυγχάνουν *άμεσα* αντί να οπισθοχωρούν εξαντλητικά.

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

Το κλασικό `'aaaa' =~ /a++a/`:

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

Το a++ καταβροχθίζει και τα τέσσερα a. Το τελικό a τότε δεν έχει με τι να ταιριάξει. Επειδή το a++ αρνείται να υποχωρήσει, η μηχανή αποτυγχάνει άμεσα. (Το ίδιο μοτίβο με άπληστο a+a ταιριάζει με `'aaaa'` μετά από οπισθοχώρηση.)

Ένας σύνηθης ιδιωματισμός για αντιστοίχιση συμβολοσειρών σε εισαγωγικά χωρίς καταστροφή οπισθοχώρησης:

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

Κάθε επανάληψη της εσωτερικής ομάδας είτε καταβροχθίζει μια απεριόριστη ακολουθία μη-εισαγωγικών, μη-ανάστροφων χαρακτήρων (κτητικά), είτε μία διαφυγή. Καμία εναλλακτική δεν είναι διατεθειμένη να υποχωρήσει, οπότε η αποτυχία είναι γρήγορη. Η πλήρης ανάλυση αυτού του μοτίβου βρίσκεται στο κεφάλαιο *performance*.

Ο κτητικός ισοδυναμεί με συντακτική ζάχαρη ατομικής ομάδας

Οι κτητικοί ποσοδείκτες είναι ακριβής συντακτική ζάχαρη για ατομική ομάδα γύρω από το ποσοδεικτημένο άτομο. Τα ακόλουθα είναι ισοδύναμα:

Κτητική μορφή	Μορφή ατομικής ομάδας
`PAT*+`	`(?>PAT*)`
`PAT++`	`(?>PAT+)`
`PAT?+`	`(?>PAT?)`
`PAT{n,m}+`	`(?>PAT{n,m})`

Η κτητική σημειογραφία είναι συντομότερη· η σημειογραφία ατομικής ομάδας γενικεύεται περισσότερο (μπορείτε να τυλίξετε αυθαίρετες εναλλαγές, όχι μόνο ποσοδεικτημένα άτομα). Το κεφάλαιο *groups and captures* καλύπτει τις ατομικές ομάδες λεπτομερώς.

Μη επιτρεπόμενοι συνδυασμοί κτητικού και μη άπληστου

Ο κτητικός και ο μη άπληστος είναι αντιφατικοί· η Perl απορρίπτει τους συνδυασμούς. Υπάρχει ισοδύναμη νόμιμη μορφή για κάθε έναν:

Μη επιτρεπτό	Νόμιμο ισοδύναμο
`X??+`	`X{0}`
`X+?+`	`X{1}`
`X{n,m}?+`	`X{n}`

Αν διαπιστώσετε ότι θέλετε «μη άπληστο *και* κτητικό», αυτό που στην πραγματικότητα θέλετε είναι σταθερό πλήθος.

Επιλέγοντας τον σωστό ποσοδείκτη

- **Άπληστος** - η προεπιλογή, και σωστός τις περισσότερες φορές.
- **Μη άπληστος** - όταν σαρώνετε ανάμεσα σε δείκτη αρχής και τέλους που μπορούν και οι δύο να εμφανίζονται νόμιμα πολλαπλές φορές.
- **Κτητικός** - όταν η αντιστοιχία είτε δουλεύει με ακριβώς έναν τρόπο είτε αποτυγχάνει, και η ταχύτητα στην αποτυχία έχει σημασία.

Όταν αμφιβάλλετε, γράψτε τον ως άπληστο και προσθέστε έναν έλεγχο που αποτυγχάνει αν τον έγραψατε λάθος.

Ποσοδεικτοποίηση μηδενικού πλάτους

Ένας ποσοδείκτης σε διεκδίκηση μηδενικού πλάτους είναι σχεδόν πάντα λάθος. Το `\b*` είναι συντακτικά νόμιμο αλλά ταιριάζει με την κενή συμβολοσειρά σε κάθε όριο λέξης, κάτι που δεν είναι χρήσιμο. Η μηχανή απορρίπτει κάποιους από αυτούς ρητά με την προειδοποίηση «matches null string many times» υπό `use warnings`· για τους υπόλοιπους, τα αποτελέσματα συνήθως δεν είναι αυτά που εννοούσε ο συγγραφέας.

Η ίδια επιφύλαξη ισχύει για ποσοδεικτημένες ομάδες των οποίων το εσωτερικό περιεχόμενο μπορεί να ταιριάξει με τίποτα. Αν το `(\w*)*` φτάσει σε επανάληψη όπου το `\w*` ταίριαξε με μηδέν χαρακτήρες, το εξωτερικό `*` θα έκανε αιώνιο βρόχο. Η Perl σπάει αυτόν τον βρόχο αυτόματα - δείτε το κεφάλαιο *performance* σχετικά με τον τερματισμό αντιστοιχίας μηδενικού μήκους.

Επεξεργασμένα παραδείγματα από την ενότητα οπισθοχώρησης του `perlre`

Οκτώ παραλλαγές του «βρες τα ψηφία στο τέλος μιας συμβολοσειράς». Κάθε μία αποτυγχάνει ή πετυχαίνει για διαφορετικό λόγο· η ανάγνωσή τους ως σύνολο είναι ένας γρήγορος τρόπος να αφομοιώσετε τη συμπεριφορά άπληστων και μη άπληστων.

```
$_ = "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"
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
/(.*)\b(\d+)$/      => "I have 2 numbers: "      "53147"
/(.*\D)(\d+)$/      => "I have 2 numbers: "      "53147"
```

Διαβάζοντας καθεμία:

- `(.*)(\d*)` - το άπληστο `.*` τρώει τα πάντα· το `\d*` ικανοποιείται από την κενή συμβολοσειρά. Ταιριάζει αλλά καμία σύλληψη δεν είναι αυτό που ήθελε ο συγγραφέας.
- `(.*)(\d+)` - το `.*` υποχωρεί όσο χρειάζεται ώστε να αποκαλύψει τουλάχιστον ένα ψηφίο, αλλά μόνο το *τελευταίο* ψηφίο της ακολουθίας.
- `(.*?)(\d*)` - και τα δύο μη άπληστα: προτιμούν την κενή αντιστοιχία· και οι δύο συλλήψεις είναι κενές.
- `(.*?)(\d+)` - το πρώτο μη άπληστο, το δεύτερο άπληστο. Το `.*?` επεκτείνεται μέχρι το `\d+` να αρπάξει 2. Σταματά στην πρώτη ακολουθία ψηφίων.
- `(.*)(\d+)$` - αγκυρωμένο: το `\d+` πρέπει να είναι στο τέλος. Το άπληστο `.*` υποχωρεί ψηφία μέχρι το τελευταίο να είναι στο τέλος της συμβολοσειράς· το αποτέλεσμα είναι η ίδια παραδοξότητα «μόνο το τελευταίο ψηφίο».
- `(.*?)(\d+)$` - μη άπληστο και αγκυρωμένο: το `.*?` επεκτείνεται μέχρι το `\d+$` να αρπάξει *ολόκληρη* την τελική ακολουθία ψηφίων. Σωστό.
- `(.*)\b(\d+)$` - αγκυρωμένο, με `\b` για να επιβληθεί το όριο πριν την ακολουθία ψηφίων. Ισοδύναμο.
- `(.*\D)(\d+)$` - ρητό μη ψηφίο πριν την ακολουθία ψηφίων. Ισοδύναμο.

Τα μαθήματα:

1. Ο ποσοδείκτης `(.*` έναντι `.*?)` ελέγχει την *απληστία*, αλλά η αγκύρωση (`$`, `\b`, `\D`) είναι αυτό που ελέγχει *ποια ακολουθία ταιριάζει*.
2. Ο άπληστος σπανίως είναι «λάθος» από μόνος του· το λάθος είναι μια άγκυρα που λείπει.

Σημειώσεις μεταξύ μηχανών

Η σύνταξη ποσοδεικτών είναι μία από τις πιο αποκλίνουσες επιφάνειες ανάμεσα στις μηχανές regex. Ο πλήρης πίνακας βρίσκεται στο κεφάλαιο *cross-engine*· η περίληψη:

- Το **POSIX BRE** αντιμετωπίζει τα γυμνά `+` και `?` ως *κυριολεκτικούς χαρακτήρες*. Χρησιμοποιήστε `\+` και `\?` για τις μορφές μετα-χαρακτήρα.
- Το **POSIX ERE** έχει `+`, `?`, `{m,n}` ως μετα-χαρακτήρες (όπως η Perl) αλλά χωρίς κτητικούς ποσοδείκτες και χωρίς ατομικές ομάδες.
- Η **RE2 / Go** έχει `+`, `?`, `*`, `*?`, αλλά καθόλου κτητικούς ποσοδείκτες, ατομικές ομάδες, ή οπισθαναφορές. Η εγγύηση γραμμικού χρόνου αντικαθιστά την ανάγκη τους.

Δείτε επίσης

- Το κεφάλαιο *performance* - καταστροφική οπισθοχώρηση, ξετύλιγμα του βρόχου, και πού οι κτητικοί ποσοδείκτες δικαιολογούν τη χρήση τους.
- Το κεφάλαιο *groups and captures* - ατομικές ομάδες `(?>…)`, η γενική μορφή του «χωρίς οπισθοχώρηση μέσα σε αυτή την κατασκευή».
- Το κεφάλαιο *character classes* - `[^X]+` ως ταχύτερη, ασφαλέστερη εναλλακτική του `.*?`.
- Το κεφάλαιο *cross-engine* - διαθεσιμότητα ποσοδεικτών μεταξύ οικογενειών.
- `m`, `s`, `qr` - αναφορές τελεστών.

Ομάδες και συλλήψεις

Η ομαδοποίηση κάνει δύο πράγματα, τα οποία εύκολα συγχέονται: μετατρέπει μια ακολουθία στοιχείων μοτίβου σε μία μονάδα για ποσοδεικτοποίηση και εναλλαγή, και αποθηκεύει ό,τι ταίριαξε αυτή η μονάδα για μεταγενέστερη χρήση.

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

Πέρα από την απλή συλλαμβάνουσα μορφή `(…)`, η Perl παρέχει έναν μικρό ζωολογικό κήπο κατασκευών σε παρενθέσεις: μη συλλαμβάνουσες ομάδες για καθαρή ομαδοποίηση, ονομαστικές συλλήψεις για αυτο-τεκμηριωμένα μοτίβα, ατομικές ομάδες για έλεγχο της οπισθοχώρησης, αναδρομικά υπομοτίβα για εμφωλευμένες δομές, υπό συνθήκη μοτίβα για διακλάδωση με βάση προηγούμενες συλλήψεις, και την κατασκευή επαναφοράς διακλάδωσης για παράλληλες εναλλακτικές. Αυτό το κεφάλαιο τις καλύπτει όλες.

Συλλαμβάνουσες ομάδες: $1, $2, …

Κάθε ζεύγος μη διαφεμένων παρενθέσεων σε ένα μοτίβο ανοίγει μια συλλαμβάνουσα ομάδα. Μετά από μια επιτυχημένη αντιστοίχιση, το ταιριασμένο κείμενο της n-οστής ομάδας βρίσκεται στο `$n`:

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

Σε περιβάλλον λίστας, μια αντιστοίχιση επιστρέφει απευθείας τη λίστα των συλληφθεισών συμβολοσειρών:

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

Αν το μοτίβο αποτύχει, η λίστα είναι κενή - ένας χρήσιμος ιδιωματισμός για «ανάλυσε ή παραιτήσου»:

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

Δεν υπάρχει άνω όριο στο πλήθος των ομάδων σύλληψης. Οι ομάδες αριθμούνται με την πιο αριστερή ανοιχτή παρένθεση ως ομάδα 1, την επόμενη ως ομάδα 2, κ.ο.κ.:

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

Το $1 συλλαμβάνει την εξωτερική ομάδα, το $2 την πρώτη εσωτερική, το $3 την επόμενη, το $4 την πιο εσωτερική.

Συνέλαβε την κενή συμβολοσειρά έναντι δεν ταίριαξε καθόλου

Μια ομάδα σύλληψης που δεν συμμετείχε στην αντιστοίχιση έχει το $n *απροσδιόριστο*. Μια ομάδα που συμμετείχε και ταίριαξε με κενή συμβολοσειρά έχει το $n ορισμένο και ίσο με "". Η διάκριση έχει σημασία:

```
"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 = ""
```

Στις δύο πρώτες περιπτώσεις, ο ποσοδείκτης είναι *εκτός* της ομάδας, οπότε η ίδια η ομάδα ποτέ δεν έκλεισε (η μηχανή ταίριαξε μηδέν επαναλήψεις - η ομάδα δεν εισήλθε καθόλου). Η οπισθαναφορά \g1 επομένως δεν έχει τιμή για σύγκριση και αποτυγχάνει. Στις δύο δεύτερες περιπτώσεις, ο ποσοδείκτης είναι *μέσα* στην ομάδα, οπότε η ομάδα έτρεξε ακριβώς μία φορά και συνέλαβε μια κενή συμβολοσειρά· το \g1 ταιριάζει με την ίδια κενή συμβολοσειρά σε εκείνη τη θέση.

Το μάθημα: όταν μια ομάδα είναι προαιρετική, βάλτε τον ποσοδείκτη μέσα αν θέλετε τη σημασιολογία «η κενή αντιστοιχία μετράει ως αντιστοιχία».

Ελέγχετε πάντα τις συλλήψεις με `defined`, όχι για αλήθεια - μια κενή σύλληψη είναι αληθής-αλλά-κενή υπό τη σημασιολογία συμβολοσειρών:

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

Οι αποτυχημένες αντιστοιχίσεις δεν επαναφέρουν τις μεταβλητές σύλληψης

Αν μια αντιστοίχιση αποτύχει, τα $1, $2, ... διατηρούν τις προηγούμενες τιμές τους από την τελευταία *επιτυχημένη* αντιστοίχιση στην ίδια εμβέλεια. Αυτό είναι χαρακτηριστικό: σας επιτρέπει να γράψετε μια σειρά από πιο εξειδικευμένα μοτίβα και να αναφερθείτε στις συλλήψεις της καλύτερης αντιστοιχίας μετά.

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

Είναι επίσης μια συχνή πηγή σύγχυσης. Ελέγχετε πάντα την τιμή επιστροφής της αντιστοίχισης πριν διαβάσετε τις μεταβλητές σύλληψης.

Μη συλλαμβάνουσες ομάδες: `(?:…)`

Αν χρειάζεστε την ομαδοποίηση μόνο για ποσοδεικτοποίηση ή εναλλαγή, και δεν θέλετε τη σύλληψη, χρησιμοποιήστε `(?:…)`:

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

Οι μη συλλαμβάνουσες ομάδες είναι μικρή νίκη σε ταχύτητα και μεγαλύτερη νίκη σε σαφήνεια. Σηματοδοτούν «αυτή η ομαδοποίηση είναι για σύνταξη, όχι για δεδομένα». Επίσης αποτρέπουν την επαναρίθμηση των συλλαμβανουσών ομάδων που σας ενδιαφέρουν:

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

Χωρίς τα περιτυλίγματα `(?:…)`, τα `$2`, `$3`, `$4` θα ορίζονταν όλα και το επιθυμητό `$2` (ο εκθέτης) θα μετατοπιζόταν στο `$5`.

Η μορφή με εμβέλεια `(?flags:…)` (π.χ. `(?i:cat)`, `(?xms:…)`) προσαρτά τροποποιητές μόνο στο εσωτερικό μοτίβο και είναι επίσης μη συλλαμβάνουσα - δείτε το κεφάλαιο *modifiers*.

Το `split` επωφελείται επίσης από το `(?:…)`. Το `split /(?:\s+)/` διαχωρίζει σε ακολουθίες λευκών χαρακτήρων χωρίς να εισάγει τους διαχωριστές στην έξοδο· το `split /(\s+)/` τους αφήνει σε εναλλασσόμενες θέσεις.

Ονομαστικές συλλήψεις

Το `(?<name>…)` ή το `(?'name'…)` ονοματίζει μια ομάδα. Η αντιστοιχία της είναι προσβάσιμη μέσω του κατακερματισμού `%+`:

```
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
}
```

Οι ονομαστικές ομάδες επίσης γεμίζουν τα `$1`, `$2`, … με τη συνηθισμένη σειρά από αριστερά προς τα δεξιά, οπότε κώδικας που χρησιμοποιεί και τις δύο συμβάσεις λειτουργεί. Μέσα στο μοτίβο, αναφερθείτε σε μια ονομαστική ομάδα με `\k<name>` (ή με οποιαδήποτε από τις μορφές αγκίστρων/εισαγωγικών παρακάτω):

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

Τα ονόματα ακολουθούν τους κανόνες αναγνωριστικών της Perl (`[_A-Za-z][_A-Za-z0-9]*`)· δεν μπορούν να ξεκινούν με ψηφίο και δεν μπορούν να περιέχουν παύλες.

Αν δύο διακριτές ομάδες μοιράζονται ένα όνομα, το `$+{name}` αναφέρεται στην *πιο αριστερή ορισμένη* ομάδα στην αντιστοίχιση. Αυτό σπάνια το θέλετε *εκτός* μιας επαναφοράς διακλάδωσης (`(?|…)`), όπου τα κοινά ονόματα είναι ιδιωματικά.

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

Για προγραμματιστές που μεταφέρουν από το άρθρωμα `re` της Python ή το PCRE, η Perl δέχεται τις γραφές τύπου Python:

Μορφή Python	Ισοδύναμο Perl
`(?P<NAME>...)`	`(?<NAME>...)`
`(?P=NAME)`	`\k<NAME>`
`(?P>NAME)`	`(?&NAME)`

Οι μορφές Python δουλεύουν αλλά δεν είναι ιδιωματική Perl· προτιμήστε τις φυσικές μορφές σε νέο κώδικα.

Οπισθαναφορές

Μια οπισθαναφορά σε ένα μοτίβο απαιτεί μια μεταγενέστερη θέση να ταιριάξει με το *ίδιο κείμενο* που συνέλαβε μια προηγούμενη ομάδα - όχι με το ίδιο μοτίβο, αλλά με τους ίδιους πραγματικούς χαρακτήρες.

Μορφή	Αναφέρεται σε
`\1 ... \9`	πρώτη έως ένατη συλλαμβάνουσα ομάδα (παλαιά μορφή)
`\g1, \g2`	αριθμημένη σύλληψη· ισοδύναμο με `\1, \2`
`\g{1}`	μορφή με άγκιστρα· υποχρεωτική όταν διαφορετικά θα ακολουθούσαν ψηφία
`\g-1`	πιο πρόσφατα ανοιχθείσα συλλαμβάνουσα ομάδα (σχετική)
`\g{-2}`	δεύτερη πιο πρόσφατα ανοιχθείσα ομάδα
`\k<name>`	ονομαστική σύλληψη
`\k'name'`	ίδιο, μορφή με απλά εισαγωγικά
`\k{name}`	ίδιο, μορφή με άγκιστρα (επιτρέπει περιβάλλοντα διαστήματα)
`\g{name}`	ονομαστική, εναλλακτική γραφή

Παραδείγματα:

```
# 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{…}` έναντι `\1` για αποσαφήνιση

Χρησιμοποιείτε `\g{…}` (ή `\k<…>`) όταν ψηφία ή χαρακτήρες που μοιάζουν με οκταδικούς ακολουθούν την αναφορά, για να αποφύγετε την ασάφεια:

```
/(\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
```

Ο κανόνας της Perl για τη γυμνή μορφή `\N`: τα `\1` έως `\9` σημαίνουν πάντα οπισθαναφορές. Τα `\10`, `\11`, … σημαίνουν οπισθαναφορά *μόνο αν* έχουν ανοίξει τόσες ομάδες σύλληψης νωρίτερα στο μοτίβο· αλλιώς είναι κυριολεκτικοί οκταδικοί χαρακτήρες. Γι” αυτό το `\g{...}` είναι η ασφαλέστερη μορφή όταν το μοτίβο χτίζεται με συνένωση.

Αν χρησιμοποιείτε τη μορφή με άγκιστρα, επιτρέπονται προαιρετικά περιβάλλοντα διαστήματα - τα `\g{ -1 }` και `\k{ name }` είναι έγκυρα.

Σχετικές οπισθαναφορές

Τα `\g-1`, `\g{-1}` αναφέρονται στην *αμέσως προηγούμενη* συλλαμβάνουσα ομάδα· το `\g{-2}` αναφέρεται στην προηγούμενη αυτής· κ.ο.κ. Η απόσταση μετράται με *ανοιγμένες* παρενθέσεις, συμπεριλαμβανομένων των μη κλειστών. Αυτό έχει σημασία όταν ένα τμήμα μοτίβου παρεμβάλλεται μέσα σε ένα άλλο:

```
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
```

Χωρίς σχετικές οπισθαναφορές, αυτό θα απαιτούσε να ξέρετε πόσες ομάδες προηγούνται του `$pair` σε κάθε σημείο παρεμβολής.

Οι ονομαστικές και οι σχετικές αναφορές κάνουν τα μακριά μοτίβα ανθεκτικά στην αντιγραφή-επικόλληση.

Ατομικές ομάδες: `(?>…)`

Το `(?>…)` είναι μια μη συλλαμβάνουσα ομάδα της οποίας τα περιεχόμενα, αφού ταιριάξουν, *δεσμεύονται*. Η μηχανή δεν μπορεί να οπισθοχωρήσει μέσα στην ομάδα σε αποτυχία - μόνο να την προσπεράσει ως σύνολο.

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

Η πλήρης ανάλυση βρίσκεται στο κεφάλαιο *performance*. Η κατασκευή τεκμηριώνεται εδώ επειδή η σύνταξή της είναι μια ομάδα σε παρενθέσεις· δομικά ανήκει στο κεφάλαιο των συλλήψεων δίπλα στο `(?:…)`.

Οι κτητικοί ποσοδείκτες (`*+`, `++`, `?+`, `{n,m}+`) είναι ακριβής συντακτική ζάχαρη για `(?>…)` γύρω από το ποσοδεικτημένο άτομο. Τα ακόλουθα είναι ισοδύναμα:

Κτητικός	Μορφή ατομικής ομάδας
`PAT*+`	`(?>PAT*)`
`PAT++`	`(?>PAT+)`
`PAT?+`	`(?>PAT?)`
`PAT{n,m}+`	`(?>PAT{n,m})`

Η γραφή μακράς μορφής `(*atomic:…)` γίνεται επίσης δεκτή.

Αναδρομικά υπομοτίβα

Οι regex της Perl μπορούν να αναφέρονται πίσω σε μια ομάδα σύλληψης σαν να ήταν κλήση υπορουτίνας. Η κατασκευή ξανατρέχει το συλληφθέν μοτίβο στην τρέχουσα θέση. Αυτό κάνει αληθινά αναδρομικές δομές αντιστοιχίσιμες - ισοζυγισμένες παρενθέσεις, S-εκφράσεις, εμφωλευμένες αγκύλες - χωρίς να βγαίνετε από τη DSL της regex.

Μορφή	Καλεί αναδρομικά
`(?R)`	ολόκληρο το μοτίβο
`(?0)`	ίδιο με `(?R)`
`(?1)`	ομάδα 1
`(?2)`	ομάδα 2 (κ.ο.κ.)
`(?-1)`	πιο πρόσφατα ανοιχθείσα ομάδα (σχετική)
`(?+1)`	επόμενη ομάδα προς άνοιγμα (σχετική προς τα εμπρός)
`(?&NAME)`	ονομαστική ομάδα
`(?P>NAME)`	ίδιο με `(?&NAME)` (συμβατό με Python)

Σημείωση: η σχετική αναδρομή μετράει μη κλειστές ομάδες, σε αντίθεση με τις σχετικές οπισθαναφορές. Το `(?-1)` σημαίνει πάντα την πιο πρόσφατα ανοιχθείσα ομάδα είτε έχει κλείσει είτε όχι.

Ένας αντιστοιχιστής ισοζυγισμένων παρενθέσεων:

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

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

Το μπλοκ `(?(DEFINE)...)` δηλώνει ένα ονομαστικό υπομοτίβο χωρίς το ίδιο να ταιριάζει με τίποτα· το `(?&paren)` μετά από αυτό καλεί εκείνο το υπομοτίβο, και το αναδρομικό `(?&paren)` μέσα στο σώμα είναι αυτό που δίνει το απεριόριστο βάθος. Το `(?R)` δεν θα δούλευε εδώ - καλεί αναδρομικά *ολόκληρο* το περικλείον μοτίβο, συμπεριλαμβανομένων των αγκύρων ^ και $ που προστίθενται στο σημείο κλήσης, κάτι που αναγκάζει κάθε επίπεδο αναδρομής να απαιτήσει αρχή και τέλος συμβολοσειράς.

Ένα πιο λεπτομερές αναδρομικό μοτίβο: ταίριαξε μια συνάρτηση `foo(...)` όπου το όρισμα μπορεί το ίδιο να περιέχει ισοζυγισμένες παρενθέσεις.

```
my $re = qr/(             # group 1: full function call
              foo
              (           # 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)
```

Κατάσταση συλλήψεων μέσα σε αναδρομή

Όταν μια ομάδα καλεί αναδρομικά τον εαυτό της, οι συλλήψεις που ορίζονται μέσα στην αναδρομή *δεν* είναι ορατές στον καλούντα αφού επιστρέψει η αναδρομή. Η αναδρομική κλήση έχει τη δική της κατάσταση συλλήψεων, η οποία απορρίπτεται κατά την επιστροφή. Γι" αυτό τα περισσότερα αναδρομικά μοτίβα τυλίγουν μια δευτερεύουσα ομάδα σύλληψης γύρω από την αναδρομική κλήση όταν χρειάζεται το ταιριασμένο κείμενο:

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

Εδώ το `$+{NAME}` είναι η εξωτερική σύλληψη· το `$+{NAME_PAT}` είναι απροσδιόριστο επειδή ζούσε μόνο μέσα στην αναδρομή.

Το ανώτατο όριο βάθους αναδρομής είναι μεταγλωττισμένο στη μηχανή. Μοτίβα που καλούν αναδρομικά χωρίς να καταναλώνουν είσοδο αποτυγχάνουν άμεσα αντί να

τρέχουν επ” άπειρον - η μηχανή ανιχνεύει τον κύκλο.

Μπλοκ (DEFINE)

Το μπλοκ `(?(DEFINE)…)` περιέχει ονομαστικά υπομοτίβα που ποτέ δεν τρέχουν από μόνα τους - καλούνται μόνο μέσω `(?&NAME)`. Έτσι γράφεται μια regex που μοιάζει με μια μικρή γραμματική:

```
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;
```

Το μπλοκ DEFINE στο τέλος κρατά τα ονομαστικά υπομοτίβα σε ένα μέρος· η κορυφή του μοτίβου διαβάζεται ως καθαρή προδιαγραφή.

Δύο προειδοποιήσεις:

- Τα υπομοτίβα σε ένα μπλοκ DEFINE *προσμετρώνται* στην απόλυτη και σχετική αρίθμηση των ομάδων σύλληψης στο περιβάλλον μοτίβο. Ονομάστε τα πάντα στο DEFINE ώστε να μην χρειάζεται να μετράτε.
- Ο βελτιστοποιητής είναι λιγότερο αποτελεσματικός σε μοτίβα τύπου DEFINE από ό,τι στην ισοδύναμη ενσωματωμένη μορφή. Τρέχουν, αλλά όχι πάντοτε σε μέγιστη ταχύτητα.

Υπό συνθήκη μοτίβα: (?(cond)yes|no)

Ένα υπό συνθήκη μοτίβο επιλέγει ανάμεσα σε δύο υπομοτίβα βάσει ενός ελέγχου σε χρόνο εκτέλεσης:

- `(?(N)YES|NO)` - ταίριαξε YES αν η ομάδα N ταίριαξε κάτι, αλλιώς NO.
- `(?(<NAME>)YES|NO)` - ίδιο, αλλά με όνομα.
- `(?(?=LOOK)YES|NO)` - ταίριαξε YES αν η πρόσθια διεκδίκηση επιτύχει.
- `(?(?{CODE})YES|NO)` - ταίριαξε YES αν το μπλοκ κώδικα επιστρέψει αληθές.
- `(?(R)YES|NO)` - ταίριαξε YES αν βρίσκεστε αυτή τη στιγμή μέσα σε αναδρομή.
- `(?(R1)YES|NO)` - ταίριαξε YES αν γίνεται αναδρομή μέσω της ομάδας 1.
- `(?(R&NAME)YES|NO)` - ταίριαξε YES αν γίνεται αναδρομή μέσω ονομαστικής ομάδας.

Η διακλάδωση NO είναι προαιρετική· η απουσία NO αντιμετωπίζεται ως «ταίριαξε πάντα».

Επεξεργασμένο παράδειγμα: προαιρετικά παρενθεσιακό κείμενο. Αν η είσοδος ανοίγει με `(`, απαίτησε ένα τελικό `)`. Διαφορετικά μην επιτρέπεις παρενθέσεις:

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

Χωρίς υπό συνθήκη μοτίβα, αυτό θα χρειαζόταν μια εναλλαγή που καλύπτει και τα δύο σχήματα· η υπό συνθήκη εκδοχή φανερώνει την πρόθεση.

Επεξεργασμένο παράδειγμα: ισοζυγισμένη γραμματική με ονομαστική αναδρομή. Χρήσιμο μέσα σε μπλοκ DEFINE για επικύρωση ευαίσθητη ως προς τα συμφραζόμενα:

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

Τα υπό συνθήκη είναι το τμήμα της regex που μοιάζει περισσότερο με πραγματική γλώσσα προγραμματισμού· καταφύγετε σε αυτά όταν μια εναλλαγή αμοιβαία αποκλειόμενων σχημάτων θα ήταν δυσκίνητη.

Επαναφορά διακλάδωσης: `(?|…)`

Μέσα στο `(?|…)`, κάθε εναλλακτική διακλάδωση ξεκινά την αρίθμηση των συλλήψεών της από την ίδια θέση. Μετά την ομάδα, η αρίθμηση συνεχίζει μία θέση μετά το μέγιστο από όλες τις διακλαδώσεις. Η πλήρης ανάλυση βρίσκεται στο κεφάλαιο *alternation*· οι συνέπειες ως προς την αρίθμηση συλλήψεων ανήκουν εδώ.

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

Αφού ταιριάξει αυτό το μοτίβο:

- Το `$1` είναι πάντα το αρχικό `a`.
- Το `$2` είναι `y`, `(p q r)`, ή `t` ανάλογα με τη διακλάδωση.
- Το `$3` είναι undef από τη διακλάδωση 1, `q` από τη διακλάδωση 2, `v` από τη διακλάδωση 3.
- Το `$4` είναι το τελικό `z`.

Αν χρησιμοποιείτε ονομαστικές συλλήψεις μέσα σε `(?|…)`, χρησιμοποιείτε *τα ίδια ονόματα με την ίδια σειρά* σε κάθε διακλάδωση:

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

Η ανάμειξη ονομάτων μεταξύ διακλαδώσεων λειτουργεί (η Perl αναλύει στο πιο αριστερό ορισμένο όνομα) αλλά παράγει εκπληκτικά αποτελέσματα: κάθε όνομα αναφέρεται στην ίδια θέση, οπότε τα `$+{a}` και `$+{b}` μπορεί να έχουν την ίδια τιμή σε διαφορετικές διακλαδώσεις.

Πίνακες θέσεων: @- και @+

Μετά από μια επιτυχημένη αντιστοίχιση, οι @- και @+ περιέχουν τις μετατοπίσεις αρχής και τέλους ολόκληρης της αντιστοιχίας και κάθε ομάδας σύλληψης:

- `$-[0]`, `$+[0]` - μετατοπίσεις ολόκληρης της αντιστοιχίας.
- `$-[n]`, `$+[n]` - μετατοπίσεις της n-οστής σύλληψης, ή undef αν η ομάδα δεν συμμετείχε.

```
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)
```

Οι μετατοπίσεις είναι συχνά ευκολότερες από τις υποσυμβολοσειρές όταν χρειάζεται να τροποποιήσετε την αρχική συμβολοσειρά στη θέση αντιστοίχισης.

@{^CAPTURE} - οι συλλήψεις ως πίνακας

Η Perl εκθέτει όλες τις αριθμημένες συλλήψεις ως έναν ενιαίο πίνακα @{^CAPTURE}, ευρετηριασμένο από το 0 (όπου ο δείκτης 0 είναι το $1, ο δείκτης 1 είναι το $2, ...):

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

Αυτό είναι βολικό όταν το πλήθος των συλλήψεων είναι μεταβλητό ή άγνωστο - κώδικας που παίρνει συλλήψεις από μοτίβο παρεχόμενο από τον χρήστη δεν χρειάζεται πια να μετράει (για να ξέρει πόσα $1/$2/... να διαβάσει.

Η ευρετηρίαση απαιτεί την οριοθετημένη μορφή με άγκιστρα:

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

Τα %{^CAPTURE_ALL} και %{^CAPTURE_NAMES} υπάρχουν για ενδοσκόπηση ονομαστικών συλλήψεων - δείτε `perlvar`.

Προ-αντιστοιχία, αντιστοιχία, μετα-αντιστοιχία

Η Perl ορίζει τρία ειδικά βαθμωτά μετά από κάθε αντιστοίχιση που εκθέτουν το περιβάλλον κείμενο:

- $` - οτιδήποτε πριν την αντιστοιχία (η *προ-αντιστοιχία*).
- $& - η ίδια η αντιστοιχία.
- $' - οτιδήποτε μετά την αντιστοιχία (η *μετα-αντιστοιχία*).

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

Οι ονομαστικές παραλλαγές `${^PREMATCH}`, `${^MATCH}`, `${^POSTMATCH}` είναι ισοδύναμες. Και οι δύο μορφές είναι μηδενικού κόστους· χρησιμοποιήστε όποια διαβάζεται καλύτερα.

$+ και $^N

Το `$+` περιέχει την αντιστοιχία της ομάδας σύλληψης με τον υψηλότερο αριθμό που πέτυχε.

Το `$^N` περιέχει την αντιστοιχία της πιο πρόσφατα κλειστής ομάδας σύλληψης - της πιο δεξιάς `)` που ολοκληρώθηκε. Αυτό είναι ακριβώς αυτό που θέλετε μέσα σε ένα μπλοκ κώδικα `(?{...})` για να προσπελάσετε την πιο πρόσφατη σύλληψη χωρίς να μετράτε παρενθέσεις:

```
$_ = "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
```

Δείτε το κεφάλαιο *performance* για την πλήρη ιστορία ενσωματωμένου κώδικα.

Μεγιστοποίηση υπο-εκφράσεων POSIX - τι κάνουν άλλες μηχανές

Μερικές μηχανές (οποιοδήποτε συμμορφούμενο POSIX NFA, συν υλοποιήσεις μηχανών σχεδιασμένες να μιμούνται το POSIX, συν τα περισσότερα υβρίδια DFA) ακολουθούν διαφορετικό κανόνα για το ποια αντιστοιχία κερδίζει όταν είναι δυνατές πολλές: *πιο μακρά-πιο αριστερή*, με τον περιορισμό ότι *κάθε υπο-έκφραση συλλαμβάνει τη μακρύτερη υποσυμβολοσειρά που μπορεί*.

Για το μοτίβο `(to|top)(o|polo)?(gical|o?logical)` εφαρμοσμένο στο `topological`:

- Η Perl (και η PCRE2, και κάθε παραδοσιακό NFA) δοκιμάζει τις εναλλακτικές από αριστερά προς τα δεξιά. Το `(to|top)` ταιριάζει με `to`· η μηχανή προχωράει. Το `(o|polo)` ταιριάζει με `o`· προχωράει. Το `(gical|o?logical)` ταιριάζει με `logical`. Τέλος.
- Η σημασιολογία POSIX απαιτεί η *συνολική* αντιστοιχία να είναι η μακρύτερη, και ανάμεσα σε αντιστοιχίες ίσου μήκους, η σύλληψη κάθε επιμέρους ομάδας να είναι η μακρύτερη. Το αποτέλεσμα είναι `top` `polo` `gical`, όπου κάθε σύλληψη είναι στο πλατύτερό της.

Οι σύγχρονες μηχανές ουσιαστικά ποτέ δεν χρησιμοποιούν σημασιολογία POSIX-NFA, αλλά μερικά εργαλεία βασισμένα σε DFA (`grep`, `awk`) την προσεγγίζουν. Το κεφάλαιο *cross-engine* έχει τον πλήρη πίνακα.

Μεταξύ μηχανών: συλλήψεις και οπισθαναφορές

Αυτό είναι το χαρακτηριστικό όπου οι μηχανές αποκλίνουν περισσότερο. Ο πλήρης πίνακας βρίσκεται στο κεφάλαιο *cross-engine*· οι σχετικές γραμμές αποσπασμένες:

Χαρακτηριστικό	Perl 5.44	PCRE	Emac	POSIX BRE	POSIX ERE	RE2/Go
Αριθμημένες συλλήψεις	ναι	ναι	ναι	ναι	αυστηρά όχι, GNU ναι	ναι (χωρίς οπισθαναφορές)
Οπισθαναφορές σε μοτίβο (\1–)	ναι	ναι	ναι	ναι (\1–\9)	αυστηρά όχι	ΟΧΙ
Ονομαστικές συλλήψεις (?<name>...)	ναι	ναι	ΟΧΙ	ΟΧΙ	ΟΧΙ	ναι
Επαναφορά διακλάδωσης (?|…)	ναι	ναι	ΟΧΙ	ΟΧΙ	ΟΧΙ	ΟΧΙ
πίνακας @{^CAPTURE}	ναι	ΟΧΙ	ΟΧΙ	ΟΧΙ	ΟΧΙ	ΟΧΙ (διαφορετικό API)
Αναδρομικά υπομοτίβα (?R) κτλ.	ναι	ναι	ΟΧΙ	ΟΧΙ	ΟΧΙ	ΟΧΙ
Υπό συνθήκη μοτίβα (?(c)y|n)	ναι	ναι	ΟΧΙ	ΟΧΙ	ΟΧΙ	ΟΧΙ

Το πιο σημαντικό συμπέρασμα: **η regex της RE2 / Go δεν υποστηρίζει καθόλου οπισθαναφορές**. Αυτό δεν είναι παράβλεψη· είναι το τίμημα που πληρώνει η RE2 για εγγυημένη αντιστοίχιση γραμμικού χρόνου. Μοτίβα που χρειάζονται οπισθαναφορές δεν είναι κανονικές γλώσσες και δεν μπορούν να αντιστοιχιστούν από ένα DFA σε γραμμικό χρόνο.

Αν μεταφέρετε regex της Perl σε υπηρεσία Go, ελέγξτε τις για οπισθαναφορές προτού υποθέσετε ότι η μετατροπή είναι μηχανική. Μοτίβα όπως `/(?<a>\w+) and \k<a>/` απλώς δεν μπορούν να εκφραστούν στην RE2.

Δείτε επίσης

- Το κεφάλαιο *alternation* - επαναφορά διακλάδωσης και οι συνέπειες αρίθμησης εναλλακτικών του (?|…).
- Το κεφάλαιο *performance* - ατομικές ομάδες, αναδρομικά μοτίβα σε επεξεργασμένα παραδείγματα, ενσωματωμένος κώδικας.
- Το κεφάλαιο *modifiers* - /n για καταστολή όλων των συλλήψεων.
- `perlvar` - %+, %-, $&, $^N, @{^CAPTURE} και συναφή.

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 5.44, 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.

Επαναφορά διακλάδωσης: `(?|…)`

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.

Σύνοψη

- `|` 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.

Δείτε επίσης

- 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.

Τροποποιητές

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

Τροποποιητής	Αποτέλεσμα
`/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
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
$x =~ /first$/m;    # matches
$x =~ /third$/m;    # matches
```

`\A`, `\z`, `\Z` remain absolute string anchors even under `/m`

- δείτε το κεφάλαιο *anchors and assertions*.

/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` - μη καταστροφική αντικατάσταση

`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/^ | $//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.

Τροποποιητής	Συμπεριφορά
`/u`	full Unicode semantics. Default under `use v5.12+`.
`/a`	restrict `\d`, `\w`, `\s`, `[[:…:]]` to ASCII.
`/aa`	`/a` plus `/i` does not cross ASCII/non-ASCII.
`/l`	follow the current `use locale` POSIX locale.
`/d`	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.
- Το `(?#comment)` αφαιρείται προτού η Φάση B δει καν το μοτίβο. Ένα κυριολεκτικό `#` μέσα στο `(?#...)` είναι μια χαρά. Ένα κυριολεκτικό `)` όχι - το σχόλιο τελειώνει στην πρώτη `)`.

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
```

Δείτε επίσης

- 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` - αναφορές τελεστών.

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})/$+{d}\/$+{m}\/
 ↪$+{y}/;
# 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)/$1\U$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` - μη καταστροφική αντικατάσταση

`/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
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
        =~ 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 nth character in the first list becomes the nth 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.

Δείτε επίσης

- `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. Regexps 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:

Μορφή	Παράδειγμα	Σημειώσεις
`\xHH`	`\x41` (A)	Two hex digits, 0–255
`\x{…}`	`\x{263a}` (☺)	Any code point
`\o{…}`	`\o{101}` (A)	Octal
`\N{name}`	`\N{GREEK SMALL LETTER SIGMA}`	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	Ταιριάζει με
`\p{L}`	any letter
`\p{Ll}`	lowercase letter
`\p{Lu}`	uppercase letter
`\p{Lt}`	titlecase letter
`\p{N}`	any numeric character
`\p{Nd}`	decimal digit
`\p{P}`	στίξη
`\p{M}`	mark (combining character)
`\p{S}`	symbol
`\p{Z}`	separator (including spaces)
`\p{C}`	control, format, unassigned, private-use
`\p{ASCII}`	code point 0–127
`\p{White_Space}`	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
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
/\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 $s = "A\x{030A}";      # A + combining ring
length($s);               # 2  - code-point count
$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
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:

Όριο	Σημασία
`\b{wb}`	Όριο λέξης Unicode - χειρίζεται don't, state-of-the-art
`\b{sb}`	όριο πρότασης
`\b{lb}`	αλλαγή γραμμής (κατάλληλο για αναδίπλωση γραμμών)
`\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
```

Για επεξεργασία φυσικής γλώσσας, τα \b{wb} και \b{sb} είναι σχεδόν πάντα αυτό που θέλετε. Το απλό \b είναι για περιβάλλοντα αναγνωριστικών ASCII.

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 а (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 а 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:

Μορφή	Σημασία
`(*script_run:PAT)` / `(*sr:PAT)`	match PAT, then check it is a script run
`(*atomic_script_run:PAT)` / `(*asr:PAT)`	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

Χαρακτηριστικό	Perl 5.44	PCRE2	Emacs (limited)	POSIX	RE2 / Go
`\p{General_Categor`	ναι	ναι	partial	όχι	ναι
`\p{Script}`	ναι	ναι	partial	όχι	ναι
`\p{Property=Value`	ναι	ναι	partial	όχι	ναι
`\X` grapheme cluster	ναι	ναι	όχι	όχι	yes (`(?-X:...)` is something else)
`\b{wb}`, `\b{sb}`	ναι	όχι	όχι	όχι	όχι
Script Runs `(*sr:…)`	ναι	όχι	όχι	όχι	όχι
Default to Unicode `\d/\w`	under `/u`	όχι	όχι	όχι	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 `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.

Δείτε επίσης

- 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.

Απόδοση

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 lookaround 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:

```
"aaaaaaaaaaaaaaaaaaaa" =~ /(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.

Κτητικοί ποσοδείκτες

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:

Κτητική μορφή	Μορφή ατομικής ομάδας
`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).

A consequence of «possessive equals atomic»: some combinations make no sense and Perl rejects them.

Illegal form	Equivalent legal form
`X??+`	`X{0}`
`X+?+`	`X{1}`
`X{n,m}?+`	`X{n}`

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 regexp 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/[^"'\/]+/;

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
            )*+
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
            \)                  # 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 `"aaabcdeeeee" =~ /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 lookaround, `(?=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` ⇒ `[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 5.44 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.

Μέτρηση

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.

Δείτε επίσης

- 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-regexp`	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 `regexp`	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 5.44 patterns and supports nearly the same feature set: backreferences, 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 5.44:

- `(?{...})` 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

Χαρακτηριστικό	Perl 5.44	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	ναι	ναι	όχι	όχι	όχι	ναι
Possessive *+, ++, ?+	ναι	ναι	όχι	όχι	όχι	όχι
Non-greedy *?, +?	ναι	ναι	ναι	όχι	όχι	ναι
Atomic group (?>...)	ναι	ναι	όχι	όχι	όχι	όχι

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.

Συντομο	Perl 5.44 (default)	Perl 5.44 + /u	PCRE2 (default	Emacs	POSIX BRE	POSIX ERE	RE2 / Go (προεπιλογή)
\d	ASCII (ή Unicode υπό /u)	Unicod	ASCII	ΟΧΙ	ΟΧΙ	ΟΧΙ	ASCII· Unicode υπό (?u)
\w	ASCII ή Unicode	Unicod	ASCII	ναι (καθοδηγείται από τον πίνακα σύνταξης)	ΟΧΙ	ΟΧΙ	ASCII· Unicode υπό (?u)
\s	ASCII ή Unicode	Unicod	ASCII	ναι	ΟΧΙ	ΟΧΙ	ASCII· Unicode υπό (?u)
όριο λέξης \b	ναι	ναι	ναι	ναι (\</\> παραδοσιακά)	ΟΧΙ	ΟΧΙ	ναι
\h horizont WS	ναι	ναι	ναι	ΟΧΙ	ΟΧΙ	ΟΧΙ	ΟΧΙ
\v vertical WS	ναι	ναι	ναι	ΟΧΙ	ΟΧΙ	ΟΧΙ	ΟΧΙ

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.

Άγκυρες και χειρισμός νέας γραμμής

Χαρακτηριστικό	Perl 5.44	PCRE	Emacs	POSIX BRE / ERE	RE2 / Go
^ αρχή συμβολοσειράς	ναι	ναι	αρχή ενταμιευτή	ναι	ναι
^ μετά από \n σε λειτουργία πολλαπλών γραμμών	/m	(?m)	πάντα (γραμμοπροσανατ	μη προσδιορισ	(?m)
$ τέλος συμβολοσειράς	ναι	ναι	τέλος ενταμιευτή	ναι	ναι
$ πριν την τελική \n	ναι	ναι	όχι	ποικίλλει	ναι
. matches \n	/s	(?s)	όχι	όχι	(?s)
\A / \z	ναι	ναι	\` / \'	όχι	yes (\A, \z)
\Z (before optional final \n)	ναι	ναι	όχι	όχι	όχι (χρησιμοποιήστε \z)
\G (last-match position)	ναι	ναι	όχι	όχι	όχι

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.

Διεκδικήσεις περιβάλλοντος και ατομικές κατασκευές

Χαρακτηριστικό	Perl 5.44	PCRE2	Emacs	POSIX	RE2 / Go
Πρόσθια διεκδίκηση (?=…) / (?!…)	ναι	ναι	OXI	OXI	OXI
Fixed-width lookbehind (?<=…)	ναι	ναι	OXI	OXI	OXI
Οπίσθια διεκδίκηση μεταβλητού μήκους	ναι (πειραμ.)	ναι	OXI	OXI	OXI
\K keep-left	ναι	ναι	OXI	OXI	OXI
Ατομική ομάδα (?>…)	ναι	ναι	OXI	OXI	OXI
Κτητικοί ποσοδείκτες	ναι	ναι	OXI	OXI	OXI

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

Captures and backreferences

Χαρακτηριστικό	Perl 5.44	PCRE2	Emac	POSIX BRE	POSIX ERE	RE2 / Go
Αριθμημένες συλλήψεις	ναι	ναι	ναι	ναι	αυστηρά όχι, GNU ναι	ναι
Οπισθαναφορές σε μοτίβο (\1–)	yes (1–99)	ναι	ναι	ναι (\1–\9)	αυστηρά όχι, GNU ναι	OXI
Named captures (?<name>…)	ναι	ναι	OXI	OXI	OXI	yes ((?P<name>…))
Επαναφορά διακλάδωσης (?|…)	ναι	ναι	OXI	OXI	OXI	OXI
Recursive subpatterns (?R)	ναι	ναι	OXI	OXI	OXI	OXI
Υπό συνθήκη μοτίβα (?(c)y|n)	ναι	ναι	OXI	OXI	OXI	OXI
(?{…}) embedded code	ναι	callou	OXI	OXI	OXI	OXI
πίνακας @{^CAPTURE}	ναι	OXI	OXI	OXI	OXI	OXI (διαφορετικό 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 5.44	Emacs	POSIX BRE / ERE (typical impl)	RE2 / Go
Algorithm	Traditional NFA + JIT (PCRE2)	Traditiona NFA	DFA (GNU) / NFA (others)	Hybrid NFA → lazy DFA
Worst-case time	Exponential	Exponenti	Linear (DFA)	Linear (guaranteed)
Backtracking model	ναι	ναι	DFA: no	όχι
Catastrophic backtracking risk	ναι	ναι	no (DFA)	όχι
Backreferences supported	ναι	ναι	ποικίλλει	OXI

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 lookaround 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.

Δείτε επίσης

- The *character classes* chapter - the shorthand-by-engine row appears in its L3 sidebar.
- The *anchors and assertions* chapter - the lookaround 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.
- **Βασικά** - `m//`, `s///`, οι τελεστές σύνδεσης =~ και !~, τι θεωρείται μετα-χαρακτήρας, πώς να διαφεύγετε χαρακτήρες, το μοντέλο ανάλυσης τεσσάρων φάσεων.
- **Κλάσεις χαρακτήρων** - κλάσεις σε αγκύλες `[…]`, αρνημένες κλάσεις, συντομογραφίες `\d \w \s`, κλάσεις POSIX, η εκτεταμένη μορφή σε αγκύλες `(?[…])`, το επεξεργασμένο παράδειγμα διεύθυνσης IP.
- **Άγκυρες και διεκδικήσεις** - `^`, `$`, `\b`, `\A`, `\z`, `\G`, `\K`, πρόσθια διεκδίκηση, οπίσθια διεκδίκηση, τα όρια Unicode `\b{…}`, η παράθεση ως AND.
- **Ποσοδείκτες** - `*`, `+`, `?`, `{n,m}`, άπληστοι έναντι μη άπληστων έναντι κτητικών, οι αρχές της αντιστοίχισης.
- **Ομάδες και συλλήψεις** - `(...)`, `(?:...)`, ονομαστικές συλλήψεις, οπισθαναφορές, ατομικές ομάδες, αναδρομικά υπομοτίβα, υπό συνθήκη μοτίβα, επαναφορά διακλάδωσης.

- **Εναλλαγή** - `|`, προτεραιότητα, παραγοντοποίηση κοινού προθέματος, κενές εναλλακτικές, επαναφορά διακλάδωσης.
- **Τροποποιητές** - `/i`, `/m`, `/s`, `/x`, `/xx`, `/g`, `/c`, `/r`, `/e`, `/n`, `/p`, `/o`, `/a`, `/aa`, `/u`, `/l`, `/d`, ενσωματωμένες μορφές `(?i…)`, μορφές με εμβέλεια `(?i:…)`, το μοντέλο ανάλυσης τεσσάρων φάσεων, `use re 'strict'`.
- **Αντικατάσταση** - `s///` σε βάθος: η συμβολοσειρά αντικατάστασης, `/e`, `/ee`, `/r`, `\K` στην αντικατάσταση, τερματισμός αντιστοιχίας μηδενικού μήκους.
- **Unicode** - `\p{…}`, `\P{…}`, `\X`, γραφές, οι τροποποιητές συνόλου χαρακτήρων, πτύχωση πεζών-κεφαλαίων, δρομείς γραφής, κωδικοποίηση έναντι χαρακτήρα.
- **Απόδοση** - οπισθοχώρηση, καταστροφικά μοτίβα, ξετύλιγμα του βρόχου, ατομικές ομάδες, κτητικοί ποσοδείκτες, αναδρομικά μοτίβα, ενσωματωμένος κώδικας, ειδικά ρήματα ελέγχου οπισθοχώρησης, εσωτερικές βελτιστοποιήσεις που εφαρμόζει η μηχανή για λογαριασμό σας.
- **Μεταξύ μηχανών** - σύγκριση με PCRE2, Emacs, POSIX BRE, POSIX ERE, και RE2 / Go.

4.3.3 Μια πρώτη ολοκληρωμένη διαδρομή

Το συντομότερο χρήσιμο πρόγραμμα regex - εύρεση τρίγραμμων λέξεων που επαναλαμβάνονται διαδοχικά, χωρισμένες με ένα διάστημα:

```
my $text = "I said the the other day";
if ($text =~ /\b(\w{3})\s\1\b/) {
    print "Repeated: $1\n";        # Repeated: the
}
```

- Το `\b` είναι όριο λέξης - το μοτίβο ενεργοποιείται μόνο στις άκρες λέξεων.
- Το `(\w{3})` συλλαμβάνει ακριβώς τρεις χαρακτήρες λέξης στο `$1`.
- Το `\s` είναι ένας λευκός χαρακτήρας ανάμεσα στα δύο αντίγραφα.
- Το `\1` είναι η οπισθαναφορά: ταίριαξε ξανά τους ίδιους τρεις χαρακτήρες.

Κάθε πιο σύνθετο μοτίβο στον οδηγό χτίζεται πάνω σε αυτή την ιδέα - αγκυρώστε όπου χρειάζεται, περιγράψτε αυτό που θέλετε, συλλάβετε αυτό που θέλετε να ανακτήσετε.

4.3.4 Συμβάσεις στα παραδείγματα

- Η έξοδος των παραδειγμάτων εμφανίζεται ως ενσωματωμένο σχόλιο `# …` δίπλα στην έκφραση που την παράγει.
- Τα παραδείγματα προϋποθέτουν τους προεπιλεγμένους τροποποιητές, εκτός αν εμφανίζουν `/i`, `/x`, κτλ.
- Όπου ένα μοτίβο εμφανίζεται μόνο του (χωρίς `=~`), θεωρήστε το ως τμήμα που το περιβάλλον κείμενο θα συνδυάσει με μια συμβολοσειρά.
- Όταν χρειάζεται ένας χαρακτήρας Unicode, τα παραδείγματα χρησιμοποιούν `\x{263a}` ή `\N{GREEK SMALL LETTER SIGMA}` ώστε το αποδιδόμενο κείμενο να ταιριάζει με την πηγή.

4.3.5 Σχετικές σελίδες αναφοράς

Ο οδηγός είναι διδακτική αναφορά· αυτές είναι οι αναφορές τελεστών στις οποίες παραπέμπει ο οδηγός παντού.

- `m` - ο τελεστής αντιστοίχισης.
- `s` - αντικατάσταση.
- `qr` - μεταγλώττιση μοτίβου για επαναχρησιμοποίηση.
- `split` - διαχωρισμός συμβολοσειράς με βάση μια regex.
- `pos` - ανάγνωση ή ορισμός της θέσης αντιστοίχισης `/g`.
- `quotemeta` - διαφυγή μετα-χαρακτήρων σε συμβολοσειρά.
- `tr` - μετάφραση χαρακτήρα προς χαρακτήρα. Συχνά συγχέεται με την αντικατάσταση regex αλλά είναι άσχετη ως προς τη σημασιολογία.

4.4 Απόδοση

Πώς να κάνετε την Perl να τρέχει γρήγορα στο pperl και - εξίσου σημαντικό - πώς να διαπιστώσετε αν το κάνει ήδη. Αυτός ο οδηγός είναι το πρακτικό στρώμα: πού οι κλασικές συμβουλές απόδοσης της Perl εξακολουθούν να ισχύουν, πού το JIT και ο αυτόματος παραλληλοποιητής του pperl τις έχουν αθόρυβα καταστήσει άσχετες, και πού το pperl ανταμείβει κώδικα διαμορφωμένο με τρόπους που η upstream Perl δεν ένοιαζε ποτέ.

Σκόπιμα δεν είναι μια ξενάγηση στον μηχανισμό. Το JIT, ο παράλληλος χρονοπρογραμματιστής, η αποστολή των native αρθρωμάτων και το FFI έχουν το καθένα το δικό του κεφάλαιο στον *οδηγό αρχιτεκτονικής*· αυτός ο οδηγός σας λέει τι να *γράψετε* ώστε να ενεργοποιηθούν, και παραπέμπει σε εκείνα τα κεφάλαια για το *γιατί*.

4.4.1 Σε ποιους απευθύνεται

Ένας ενεργός προγραμματιστής Perl με ένα πρόγραμμα που είναι πολύ αργό, ή ένα που θέλει να το διατηρήσει γρήγορο. Ξέρετε πώς να γράφετε τον κώδικα· θέλετε να μάθετε ποιες συνήθειες αποδίδουν εδώ, ποιες είναι πλέον περιττό βάρος, και πώς να μετρήσετε τη διαφορά χωρίς να μαντεύετε.

Δεν χρειάζεται να κατανοείτε το Cranelift, το Rayon ή τον εσωτερικό βρόχο του διερμηνευτή. Όπου μια συμπεριφορά εξαρτάται από αυτά, ο οδηγός δηλώνει τη συμπεριφορά και παραπέμπει στο κεφάλαιο που την εξηγεί.

4.4.2 Ο ένας κανόνας που επιβιώνει από τα πάντα

Μετρήστε πρώτα. Κάθε άλλος κανόνας σε αυτόν τον οδηγό εξαρτάται από μια μέτρηση που δεν έχετε ακόμη κάνει. Η διαίσθηση για την απόδοση της Perl, ακονισμένη στον upstream διερμηνευτή, προβλέπει λανθασμένα στο pperl και προς τις δύο κατευθύνσεις: κώδικας που περιμένετε να είναι αργός μεταγλωττίζεται ορισμένες φορές σε εγγενή κώδικα μηχανής και τρέχει γρηγορότερα από τη C· κώδικας που περιμένετε να σώσει το JIT επιστρέφει ορισμένες φορές στον διερμηνευτή επειδή μια λειτουργία συμβολοσειρών τρύπωσε μέσα σε έναν αριθμητικό βρόχο.

Ο προφίλερ δεν λέει ψέματα, ενώ η διαίσθησή σας, μεταφερμένη από έναν άλλο runtime, λέει. Ξεκινήστε από το *Μέτρηση*.

4.4.3 Πώς είναι οργανωμένος ο οδηγός

Μέτρηση

Δεν μπορείτε να ρυθμίσετε ό,τι δεν έχετε μετρήσει, και στο pperl τα συνήθη εργαλεία μέτρησης χωρίζονται σε τρεις ομάδες: εκείνα που λειτουργούν αμετάβλητα, εκείνα που λειτουργούν επειδή είναι απλή Perl που τρέχει σε έναν γρηγορότερο διερμηνευτή, και εκείνο που όλοι καταφεύγουν σε αυτό και δεν λειτουργεί καθόλου. Αυτό το κεφάλαιο τα ξεδιαλύνει και δείχνει πώς να διαβάζετε όσα σας λένε εκείνα που λειτουργούν.

Το εργαλείο που δεν λειτουργεί: Devel::NYTProf

Το `Devel::NYTProf` είναι ο τυπικός προφίλερ γραμμής και υπορουτίνας στην upstream Perl. Είναι άρθρωμα XS - αγκιστρώνεται στον διερμηνευτή μέσω μιας επέκτασης C. Το pperl δεν διαθέτει στρώμα XS, οπότε το `Devel::NYTProf` δεν φορτώνεται. Ούτε φορτώνεται οποιοσδήποτε άλλος προφίλερ βασισμένος σε XS (`Devel::DProf`, τα μονοπάτια XS του `Devel::Profiler`).

Αυτό δεν είναι ένα κενό που πρέπει να παρακαμφθεί με μια υλοποίηση του μορφότυπου εξόδου του NYTProf αμιγώς σε Perl. Το pperl έχει τον δικό του προφίλερ, ενσωματωμένο στο runtime, που βλέπει τι πραγματικά εκτελεί ο διερμηνευτής - πλήθη ops και τον χρόνο που δαπανάται στο καθένα. Χρησιμοποιήστε τον.

Where the time goes: the JIT log

pperl does not currently ship a line-or-sub profiler of its own (the one it once had belonged to a retired execution engine and never saw the current runtime). What it does ship is precise visibility into the optimization layer, which for pperl-specific tuning is usually the question you are actually asking: is my hot loop compiled, and if not, why not?

- `--jit-stats` prints counters at exit: candidates found, loops compiled, entries run compiled, guard failures, deopts.
- `PPERL_JIT_LOG=/path/file` appends one line per compiler event. The `compile skip reason=...` lines name the exact operation that kept a loop interpreted.

For symbol-level CPU attribution, use the platform profiler on a release build - `perf record ./target/release/pperl script.pl` then `perf report`. pperl is a native binary; perf sees straight through it, including JIT-compiled frames.

Χρονομέτρηση ολόκληρου του προγράμματος: ο εκτελεστής benchmark

Όταν το ερώτημα είναι «πόσο γρήγορο είναι το pperl σε αυτό, έναντι της συστημικής perl, με και χωρίς JIT και παραλληλισμό», ο εκτελεστής benchmark του έργου το απαντά απευθείας. Το `bench/run-perlbench` τρέχει πάντα το `/usr/bin/perl` ως γραμμή αναφοράς (κανονικοποιημένη στο 100) και συγκρίνει το pperl σε έως τέσσερις διαμορφώσεις.

```
./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
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
./bench/run-perlbench -t arith       # only benchmarks matching
↪"arith"
```

Κάθε εκτελεστής μοιράζεται έναν κοινό αριθμό επαναλήψεων, που προκύπτει από τον ρυθμό κενού βρόχου του πιο αργού εκτελεστή, ώστε οι αναφερόμενοι λόγοι να συγκρίνουν όμοια με όμοια. Οι στήλες +J / +P είναι ακριβώς η σύγκριση που θέλετε όταν αποφασίζετε αν ένα κομμάτι κώδικα ωφελείται από το JIT ή τον παραλληλοποιητή: αν το +J δεν είναι γρηγορότερο από την απλή εκτέλεση, ο βρόχος δεν μεταγλωττίζεται, και το *Γράφοντας γρήγορο pperl* εξηγεί γιατί και τι να αλλάξετε.

Εργαλεία αμιγούς Perl που λειτουργούν: Benchmark και Time::HiRes

Τα κλασικά εργαλεία μέτρησης εντός προγράμματος είναι αμιγής Perl. Τρέχουν στο pperl μέσω του κανονικού μονοπατιού αρθρωμάτων - το `Benchmark` είναι προσβάσιμο στο τυπικό μονοπάτι ενσωμάτωσης και εκτελείται ως Perl στον διερμηνευτή του pperl· το `Time::HiRes` παρέχεται natively. Και τα δύο συμπεριφέρονται ακριβώς όπως περιμένει ένας προγραμματιστής Perl.

Το `Benchmark` για A/B σύγκριση δύο τρόπων γραφής της ίδιας ρουτίνας:

```
use Benchmark qw(cmpthese);

my @data = (1 .. 100_000);

cmpthese(-3, {              # run each for ~3 CPU-seconds
    grep_count => sub { my $n = grep { $_ % 2 == 0 } @data },
    loop_count => sub { my $n = 0; $_ % 2 or $n++ for @data },
});
```

Το `Time::HiRes` για περιστασιακή χρονομέτρηση πραγματικού χρόνου ενός μεμονωμένου μπλοκ:

```
use Time::HiRes qw(time);

my $t0 = time;
do_the_work();
printf "took %.3f s\n", time - $t0;
```

Μια προειδοποίηση που μετράει περισσότερο στο pperl απ” ό,τι στο upstream: το `Benchmark` οδηγεί τον κώδικά σας καλώντας μια αναφορά κώδικα ξανά και ξανά, και αυτή η αποστολή ανά κλήση είναι το κόστος του ίδιου του Benchmark, όχι του κώδικά σας. Ένας βρόχος *μέσα* στο callback εξακολουθεί να μεταγλωττίζεται με JIT (οι βρόχοι μεταγλωττίζονται όπου κι αν ζουν, σε εμβέλεια αρχείου ή σε σώμα υπορουτίνας), αλλά για μικρά σώματα το κόστος κλήσης του πλαισίου κυριαρχεί και η σύγκριση υποτιμά την πραγματική ταχύτητα. Όταν αυτό που χρονομετράτε είναι ένας βρόχος που θα έπρεπε να περάσει από JIT, προτιμήστε το `bench/run-perlbench` ή ένα διάστημα `Time::HiRes` γύρω από τον βρόχο επιτόπου (δείτε *Μεταγλώττιση JIT*).

Απενεργοποίηση των βελτιστοποιητών για την απομόνωση μιας αιτίας

Δύο σημαίες σάς επιτρέπουν να αποδώσετε μια επιτάχυνση - ή μια επιβράδυνση - στο σωστό στρώμα:

```
pperl --no-jit script.pl          # interpreter only, no native␣
↪compilation
pperl --no-parallel script.pl     # single-threaded, no Rayon␣
↪dispatch
```

Τρέξτε ένα πρόγραμμα με τρεις τρόπους - προεπιλογή, `--no-jit`, `--no-parallel` - και οι διαφορές σάς λένε ποιος βελτιστοποιητής κουβαλά την εργασία. Αν το `--no-jit` μετά βίας αλλάζει τον χρόνο, το JIT δεν εμπλεκόταν σε αυτόν τον κώδικα, και η μορφή του βρόχου είναι αυτό που πρέπει να εξετάσετε. Αν το `--no-parallel` μετά βίας τον αλλάζει, ο παραλληλοποιητής απέρριψε τον βρόχο - συνήθως λόγω μιας παρενέργειας που δεν μπόρεσε να αποκλείσει (δείτε *Παράλληλη εκτέλεση*).

Μια πειθαρχία μέτρησης

1. Μετρήστε τον χρόνο του προγράμματος στην προεπιλογή έναντι `--no-jit` έναντι `--no-parallel`. Οι διαφορές σάς λένε ποιο στρώμα κουβαλά τη δουλειά.
2. Αν μια θερμή ρουτίνα είναι ένας βρόχος που περιμένατε να μεταγλωττιστεί, επιβεβαιώστε με `run-perlbench +J` ή χρονομετρώντας τον με `--no-jit` έναντι της προεπιλογής. Ένας βρόχος που δεν επιταχύνεται υπό το JIT είναι κακοσχηματισμένος, όχι αργός.
3. Μόνο τώρα αλλάξτε κώδικα. Ξαναμετρήστε με τον ίδιο τρόπο. Κρατήστε την αλλαγή αν ο αριθμός κινήθηκε και το πρόγραμμα εξακολουθεί να παράγει την ίδια έξοδο.

Οι ισχυρισμοί περί απόδοσης είναι φθηνοί· η αναφορά του προφίλερ και ο λόγος του benchmark δεν είναι. Εμπιστευτείτε τη μέτρηση.

Δείτε επίσης

- *Γράφοντας γρήγορο pperl* - αφού βρείτε τον θερμό βρόχο, πώς να τον διαμορφώσετε ώστε να τον αναλάβουν το JIT και ο παραλληλοποιητής.
- *Ιδιωματισμοί* - ποιες κλασικές βελτιστοποιήσεις αξίζει ακόμη να εφαρμόσετε προτού καταφύγετε στον προφίλερ.
- *Μεταγλώττιση JIT* - τι μεταγλωττίζει το JIT και γιατί οι χρονομετρήσεις ερμηνευμένων ops εξαφανίζονται για τους μεταγλωττισμένους βρόχους.
- *Παράλληλη εκτέλεση* - τι ασκούν η στήλη `+P` και η σημαία `--no-parallel`.
- *Απόδοση regex* - αν ο προφίλερ δείχνει σε ένα regexp, η αιτία και η θεραπεία βρίσκονται εκεί, όχι εδώ.

Ιδιωματισμοί

Η Perl κουβαλά δεκαετίες λαϊκής σοφίας περί απόδοσης - συνήθειες που απέδιδαν στον upstream διερμηνευτή και κληροδοτήθηκαν ως κανόνες. Μέρος αυτών των συμβουλών είναι διαχρονικό. Μέρος τους είναι πλέον ενεργά άσκοπο στο pperl, επειδή το JIT ή

ο αναδιπλωτής χρόνου μεταγλώττισης κάνει ήδη τη δουλειά. Και λίγες νέες συνήθειες έχουν σημασία εδώ που κανένας upstream προγραμματιστής Perl δεν χρειάστηκε ποτέ.

Αυτό το κεφάλαιο διαλέγει τους κλασικούς ιδιωματισμούς σε τρεις κάδους:

- **Εξακολουθεί να ισχύει** - η συμβουλή κρατά· το κόστος που αποφεύγει είναι υπαρκτό και στο pperl.
- **Κατέστη άνευ αντικειμένου** - ο μεταγλωττιστής ή το JIT του pperl εξαλείφει ήδη το κόστος, οπότε η περιστροφή δεν αποφέρει τίποτε και μόνο βλάπτει την αναγνωσιμότητα.
- **Νέο** - σχετικό *λόγω* του τρόπου που μεταγλωττίζει το pperl, χωρίς upstream προηγούμενο.

Μια γενική επιφύλαξη διατρέχει ολόκληρο τον κάδο «κατέστη άνευ αντικειμένου»: το JIT αναλαμβάνει έναν βρόχο μόνο όταν ο βρόχος είναι *διαμορφωμένος* γι" αυτό. Ένας βρόχος βεβαρημένος με κλήσεις υπορουτινών, αντιστοιχίες regexp ή χτίσιμο συμβολοσειρών δεν μεταγλωττίζεται και τρέχει στον διερμηνευτή - όπου η κλασική συμβουλή εφαρμόζεται ξανά πλήρως. Έτσι το «το JIT το χειρίζεται» ισχύει *για τους βρόχους που χειρίζεται το JIT*. Το *Γράφοντας γρήγορο pperl* είναι το συνοδευτικό κεφάλαιο για το πώς να εξασφαλίσετε ότι ο βρόχος σας είναι ένας από αυτούς.

Εξακολουθεί να ισχύει

Αυτά επιβιώνουν αμετάβλητα από τη μετάβαση στο pperl. Το κόστος που αποφεύγουν πληρώνεται στον διερμηνευτή, και ο περισσότερος θερμός κώδικας αγγίζει τον διερμηνευτή κάπου.

- **Βγάλτε την αμετάβλητη εργασία έξω από τους βρόχους.** Ο υπολογισμός της ίδιας τιμής σε κάθε επανάληψη είναι σπαταλημένη εργασία είτε ο βρόχος είναι ερμηνευμένος είτε μεταγλωττισμένος. Το JIT δεν βγάζει έξω αυθαίρετες εκφράσεις για λογαριασμό σας· βγάλτε το αμετάβλητο έξω μόνοι σας.
- **Αποφύγετε περιττά αντίγραφα μεγάλων δομών.** Το πέρασμα ενός μεγάλου πίνακα κατά τιμή, ή το τεμαχισμό του απλώς για μέτρηση, αντιγράφει. Περάστε μια αναφορά, χρησιμοποιήστε μια ψευδωνυμοποιημένη μεταβλητή βρόχου, ή εργαστείτε επιτόπου. Το μοντέλο τιμών του pperl έχει τα ίδια κόστη αντιγραφής που είχε πάντα η Perl.
- **Προμεταγλωττίστε τα regexp που χρησιμοποιούνται σε έναν βρόχο με `qr`.** Η μηχανή regex και η cache μεταγλώττισής της λειτουργούν όπως στο upstream· ένα μοτίβο που χτίζεται εκ νέου σε κάθε επανάληψη πληρώνει για επαναμεταγλώττιση. Αυτό είναι έδαφος regex - δείτε *απόδοση regex*.
- **Επιλέξτε τη σωστή δομή δεδομένων.** Μια αναζήτηση σε hash νικά μια γραμμική σάρωση στα ίδια μεγέθη όπως πάντα. Η αλγοριθμική πολυπλοκότητα δεν επηρεάζεται από κανένα runtime· ένας βρόχος $O(n^2)$ είναι αργός ακόμη και πλήρως JIT'd.
- **Διαβάστε αρχεία σε κομμάτια σωστού μεγέθους.** Το γραμμή προς γραμμή έναντι του slurp-and-split είναι ζήτημα I/O, και το I/O δεν μεταγλωττίζεται με JIT. Οι συμβιβασμοί του upstream μεταφέρονται απευθείας.
- **Προτιμήστε τις ενσωματωμένες λειτουργίες λίστας από τους χειρόγραφους βρόχους όπου η ενσωματωμένη είναι native.** Οι `sum`, `min`, `max`, `first` από το `List::Util` τρέχουν με ταχύτητα ενσωματωμένης συνάρτησης χωρίς στρώμα XS

(δείτε *native αρθρώματα*). Ένας χειρόγραφος βρόχος συσσωρευτή *ενδέχεται* να φτάσει με JIT στην ίδια ταχύτητα, αλλά η native ενσωματωμένη σάς οδηγεί εκεί χωρίς να εξαρτάται από τη μορφή του βρόχου.

Κατέστη άνευ αντικειμένου

Αυτά ήταν πραγματικές νίκες στην upstream Perl. Στο pperl ο μεταγλωττιστής ή το JIT αφαιρεί ήδη το κόστος, οπότε η εφαρμογή του ιδιωματισμού με το χέρι δεν αποφέρει τίποτε και συνήθως κοστίζει σε σαφήνεια.

Χειροκίνητη ενσωμάτωση θερμής αριθμητικής

Η παλιά συνήθεια του ξετυλίγματος ενός σφιχτού αριθμητικού βρόχου, ή της χειροκίνητης ενσωμάτωσης ενός μικρού αριθμητικού βοηθού για να αποφευχθεί η επιβάρυνση κλήσης, στοχεύει ένα κόστος που το JIT αφαιρεί. Ένας αριθμητικός βρόχος `for` ή `while` πάνω σε ακέραιους ή κινητής υποδιαστολής συσσωρευτές μεταγλωττίζεται σε εγγενή κώδικα μηχανής, με τις μεταβλητές του βρόχου να κρατιούνται σε καταχωρητές CPU - δείτε *Μεταγλώττιση JIT*. Το χειροκίνητο ξετύλιγμα ενός τέτοιου βρόχου τον κάνει μακρύτερο και δυσκολότερο στην ανάγνωση χωρίς να νικά τον μεταγλωττιστή.

```
# 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;
}
```

Η περιστροφή αποδίδει μόνο αν ο βρόχος *δεν* περνά από JIT - και αν δεν περνά, η λύση είναι να τον κάνετε να περάσει από JIT (αφαιρέστε την κλήση ή τη λειτουργία συμβολοσειράς που τον απέκλεισε), όχι να τον ξετυλίξετε με το χέρι.

Φύλακες καταγραφής με σταθερά DEBUG

Ο ιδιωματισμός της προστασίας ακριβής καταγραφής πίσω από μια ψευδή σταθερά χρόνου μεταγλώττισης -

```
use constant DEBUG => 0;
# ...
warn "state: @stuff\n" if DEBUG;
```

- δεν κοστίζει απολύτως τίποτε στο pperl, που είναι όλο το νόημα του ιδιωματισμού, και το pperl το προσφέρει πλήρως. Όταν η `DEBUG` είναι ψευδής σταθερά, ο μεταγλωττιστής αναδιπλώνει τον φύλακα `if DEBUG` κατά τη μεταγλώττιση: η απενεργοποιημένη πρόταση καταρρέει σε no-op, χωρίς έλεγχο συνθήκης και χωρίς διακλάδωση να απομένει στο μεταγλωττισμένο πρόγραμμα. Η παρεμβολή `"@stuff"` δεν χτίζεται ποτέ. Μπορείτε να αφήνετε φυλασσόμενη καταγραφή σε θερμά μονοπάτια με ήσυχη συνείδηση· μια ψευδής σταθερά `DEBUG` την σβήνει.

Αυτός είναι ο ιδιωματισμός που λειτουργεί *όπως σχεδιάστηκε* - συνεχίστε να τον γράφετε. Άνευ αντικειμένου είναι κάθε περαιτέρω εξυπνάδα από πάνω: δεν χρειάζεται

να σχολιάσετε ως ανενεργή την καταγραφή, να τη μετακινήσετε πίσω από μια sub, ή να την αφαιρέσετε για παραγωγή. Η σταθερά το κάνει.

Αναδίπλωση σταθερών εκφράσεων μόνοι σας

Ο προϋπολογισμός ενός σταθερού αριθμητικού αποτελέσματος, ή μιας σταθερής κλήσης σε μια καθαρή ενσωματωμένη, και η αποθήκευσή του σε μια μεταβλητή για να «αποφευχθεί ο επαναϋπολογισμός του» δεν αποφέρει τίποτε - ο μεταγλωττιστής αναδιπλώνει τις σταθερές εκφράσεις και τις καθαρές ενσωματωμένες με σταθερά ορίσματα (`sin`, `cos`, `sqrt`, `chr`, `ord`, και τα παρόμοια) κατά τη μεταγλώττιση, εφόσον η κλήση βρίσκεται σε έγκυρο πεδίο ορισμού. Τα `my $two_pi = 2 * 3.14159265358979;` και `my $root = sqrt(2);` υπολογίζονται μία φορά, κατά τη μεταγλώττιση, είτε «βοηθήσετε» βγάζοντάς τα έξω είτε όχι.

Η μία λεπτή απόχρωση: η αναδίπλωση παραλείπεται για ορίσματα εκτός του πεδίου ορισμού μιας συνάρτησης (`sqrt` αρνητικού, `log` του μηδενός), οπότε αυτά διαφεύγουν στον χρόνο εκτέλεσης και κράζουν εκεί ακριβώς όπως πρέπει. Ποτέ δεν χρειάζεται να αναδιπλώσετε μια σταθερά με το χέρι για να κερδίσετε ταχύτητα.

Αφαίρεση νεκρών βρόχων με σταθερή συνθήκη

Ένα `while (0) { ... }` ή ένα μπλοκ που φυλάσσεται από μια σταθερά-ψευδή συνθήκη εξαλείφεται κατά τη μεταγλώττιση - το σώμα αφαιρείται, δεν παρακάμπτεται απλώς κατά τον χρόνο εκτέλεσης. (Ένα `for (; 0; ...)` σε στυλ C χειρίζεται ελαφρώς διαφορετικά: το σώμα δεν εισέρχεται ποτέ, αν και η op του σκελετού του βρόχου ενδέχεται να παραμείνει.) Κώδικας που μεταγλωττίζεται εκτός υπό συνθήκη πίσω από μια ψευδή σταθερά δεν φέρει βάρος κατά τον χρόνο εκτέλεσης· δεν χρειάζεται να τον διαγράψετε φυσικά για να γίνει το πρόγραμμα γρηγορότερο.

Νέο

Αυτά δεν έχουν upstream αντίστοιχο. Έχουν σημασία λόγω του τρόπου που το pperl μεταγλωττίζει και παραλληλοποιεί, και η εσφαλμένη εφαρμογή τους εκχωρεί αθόρυβα τις μεγαλύτερες επιταχύνσεις που προσφέρει το pperl.

Κρατήστε τους θερμούς αριθμητικούς βρόχους αριθμητικούς

Η μοναδική πολυτιμότερη νέα συνήθεια. Το JIT μεταγλωττίζει έναν αριθμητικό βρόχο και, όταν η παραλληλοποίηση είναι ενεργοποιημένη (η προεπιλογή) και ο βρόχος είναι μια καθαρή αναγωγή πάνω σε ένα γνωστό εύρος, τον αποστέλλει σε όλα τα νήματα - η διαφορά μεταξύ μιας επιτάχυνσης 76× και 431× στο benchmark Mandelbrot (δείτε *παράλληλη εκτέλεση*).

Μια λειτουργία συμβολοσειράς μέσα σε αυτόν τον βρόχο εκχωρεί το παράλληλο μισό. Όταν το JIT δει μια μεταβλητή συμβολοσειράς στον βρόχο - έναν στόχο συνένωσης `.=`, έναν συσσωρευτή συμβολοσειρών - υποβαθμίζει τον βρόχο σε *σειριακό* μεταγλωττισμένο βρόχο: παραμένει εγγενής κώδικας, αλλά δεν απλώνεται πλέον σε όλα τα νήματα.

```
# Numeric reduction - JIT-compiled and eligible for parallel
↪dispatch.
my $total = 0;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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
}
```

Αν χρειάζεστε και την αριθμητική αναγωγή και τη συσσώρευση συμβολοσειρών, χωρίστε τες σε δύο βρόχους: ο αριθμητικός παραλληλοποιείται, ο συμβολοσειρών τρέχει ως σειριακός μεταγλωττισμένος βρόχος. Η ανάμειξή τους σάς κοστίζει την παράλληλη επιτάχυνση στην αριθμητική εργασία χωρίς κανένα κέρδος στην εργασία με συμβολοσειρές.

Διαμορφώστε τις αναγωγές ως συσσώρευση-χωρίς-μηδενισμό

Ο παραλληλοποιητής αναγνωρίζει μια μεταβλητή αναγωγής βλέποντάς την να συσσωρεύεται και να μη μηδενίζεται ποτέ μέσα στο σώμα του βρόχου (η ανάλυση αφαιρεί τα μοτίβα μηδενισμού από τα μοτίβα συσσώρευσης· μια μεταβλητή που είναι και τα δύο δεν αντιμετωπίζεται ως αναγωγή). Το να γράφετε τον συσσωρευτή σας στην απλή μορφή `$sum += ...`, δηλωμένο *πριν* τον βρόχο και ποτέ μη επαναρχικοποιημένο μέσα του, είναι αυτό που επιτρέπει στον βρόχο να διεκδικηθεί για παράλληλη εκτέλεση.

```
my $sum = 0;                    # declared outside - a reduction
for my $x (@data) {
    $sum += weight($x);         # accumulate, never reset →␣
↪parallelizable
}
```

Η επαναρχικοποίηση του συσσωρευτή μέσα στον βρόχο, ή η παρεμβολή ενός μηδενισμού, ματαιώνει την αναγνώριση και ο βρόχος τρέχει σειριακά. Αυτό καλύπτεται πλήρως στην ενότητα *ανίχνευση αναγωγής*.

Αφήστε τον βρόχο να περάσει από JIT - μην τον κρύβετε σε μια sub

Το JIT μεταγλωττίζει *βρόχους*, όχι σώματα υπορουτινών. Ένας θερμός αριθμητικός βρόχος γραμμένος σε εμβέλεια αρχείου ή μέσα σε μια μεγαλύτερη ρουτίνα μεταγλωττίζεται μια χαρά· η ίδια αριθμητική αναδιαρθρωμένη ώστε κάθε επανάληψη να είναι μια κλήση υπορουτίνας όχι - το πλαίσιο κλήσης είναι έδαφος του διερμηνευτή, και η επιβάρυνση ανά κλήση είναι ακριβώς αυτό που το JIT δεν μπορεί να αφαιρέσει. Όταν ένας βρόχος είναι θερμός και αριθμητικός, κρατήστε το σώμα του ενσωματωμένο αντί να μεταφέρετε κάθε επανάληψη σε μια sub για τάξη. Αναδιαρθρώστε για σαφήνεια αλλού· κρατήστε τον εσωτερικό βρόχο επίπεδο.

Δείτε επίσης

- *Γράφοντας γρήγορο pperl* - το θετικό πρόγραμμα: πώς να διαμορφώσετε έναν βρόχο ώστε να τον αναλάβουν το JIT και ο παραλληλοποιητής.
- *Μέτρηση* - επιβεβαιώστε ότι ένας ιδιωματισμός πράγματι κίνησε τον αριθμό πριν και μετά την εφαρμογή του.
- *Μεταγλώττιση JIT* - τι μεταγλωττίζεται, γιατί η χειροκίνητη ενσωμάτωση αριθμητικής είναι περιττή.
- *Παράλληλη εκτέλεση* - η ανάλυση αναγωγής και η πύλη παρενεργειών πίσω από τους «νέους» ιδιωματισμούς.
- *Ταξινόμηση* - οι ειδικοί για την ταξινόμηση ιδιωματισμοί (Schwartzian, Guttman-Rosler) αποκτούν το δικό τους κεφάλαιο.

Ταξινόμηση

Η `sort` είναι το σημείο όπου κρύβεται ένα εκπληκτικό ποσοστό του χρόνου ενός προγράμματος, επειδή το κόστος δεν είναι η ταξινόμηση - είναι ο συγκριτής, που καλείται O(n log n) φορές. Ένας συγκριτής που κάνει πραγματική εργασία ανά κλήση πολλαπλασιάζει αυτή την εργασία με τον λογάριθμο του μεγέθους της λίστας. Αυτό το κεφάλαιο αφορά την αναγνώριση του πότε ο συγκριτής είναι το σημείο συμφόρησης και τους δύο μετασχηματισμούς που το διορθώνουν.

Η σελίδα αναφοράς για την `sort` ορίζει τον τελεστή, το συμβόλαιο του συγκριτή και τους μηχανισμούς των `$a`/`$b`. Αυτό το κεφάλαιο υποθέτει ότι τα γνωρίζετε και εστιάζει στην ταχύτητα.

Ο συγκριτής είναι το κόστος

Η ίδια η `sort` είναι μια σταθερή ταξινόμηση με συγχώνευση - γρήγορη, και όχι κάτι που μπορείτε να βελτιώσετε από την Perl. Αυτό που ελέγχετε είναι το πόση εργασία συμβαίνει *μέσα* σε κάθε σύγκριση. Δύο γεγονότα καθορίζουν τον προϋπολογισμό:

- Ο συγκριτής τρέχει O(n log n) φορές. Για μια λίστα 100.000 στοιχείων αυτό είναι αρκετά πάνω από ένα εκατομμύριο κλήσεις.
- Στο pperl, το μπλοκ του συγκριτή είναι ένα callback, και τα callbacks τρέχουν στον διερμηνευτή. Το JIT μεταγλωττίζει βρόχους, όχι σώματα συγκριτών `sort` (δείτε *Μεταγλώττιση JIT*), οπότε δεν μπορείτε να βασιστείτε στο JIT για να σώσει έναν ακριβό συγκριτή με τον τρόπο που σώζει έναν αριθμητικό βρόχο. Το κόστος του συγκριτή πληρώνεται πλήρως, σε κάθε κλήση.

Αυτό το δεύτερο γεγονός είναι ο ειδικός για το pperl λόγος που οι παρακάτω μετασχηματισμοί έχουν εδώ τόση σημασία όση και στην upstream Perl - αναμφισβήτητα περισσότερη, επειδή οι επιταχύνσεις του JIT αλλού κάνουν μια αμετασχημάτιστη ταξινόμηση να ξεχωρίζει πιο έντονα σε ένα προφίλ.

Η διάγνωση είναι πάντα η ίδια: αν η *Μέτρηση* δείχνει χρόνο συγκεντρωμένο σε έναν συγκριτή, το κλειδί επαναϋπολογίζεται σε κάθε σύγκριση. Υπολογίστε το αντ” αυτού μία φορά.

Ο μετασχηματισμός Schwartzian

Όταν το κλειδί ταξινόμησης είναι ακριβό να εξαχθεί από κάθε στοιχείο - μια εξαγωγή υποσυμβολοσειράς, μια αναζήτηση πεδίου, ένα μήκος, ένας υπολογισμός - η αφελής ταξινόμηση το επαναϋπολογίζει σε κάθε σύγκριση:

```
# Slow: the key (-M $_) is computed twice per comparison, O(n log n)␣
↪times.
my @sorted = sort { -M $a <=> -M $b } @files;
```

Ο μετασχηματισμός Schwartzian υπολογίζει κάθε κλειδί ακριβώς μία φορά, ταξινομεί βάσει του προϋπολογισμένου κλειδιού, μετά το απορρίπτει - μια `map`, μια `sort`, μια `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;
```

Διαβάστε το από κάτω προς τα πάνω: η κατώτερη `map` χτίζει ένα ζεύγος `[key, element]` για κάθε στοιχείο (n υπολογισμοί κλειδιού), η `sort` συγκρίνει μόνο το φθηνό προϋπολογισμένο `$_->[0]` (κανένας υπολογισμός κλειδιού στον συγκριτή), και η ανώτερη `map` πετά το κλειδί. Το ακριβό κλειδί υπολογίζεται n φορές αντί για O(n log n) φορές.

Χρησιμοποιήστε τον όποτε το κλειδί κοστίζει περισσότερο για να εξαχθεί απ" ό,τι μια αριθμητική ή συμβολοσειρική σύγκριση. Για μια λίστα 100.000 στοιχείων ταξινομημένη βάσει ενός κλειδιού που χρειάζεται ένα μικροδευτερόλεπτο για να υπολογιστεί, η αφελής μορφή δαπανά δευτερόλεπτα μόνο στον υπολογισμό κλειδιού· ο μετασχηματισμός δαπανά ένα δέκατο του δευτερολέπτου.

Τα κομμάτια είναι η `map` και η `sort` - η σελίδα αναφοράς της `sort` δείχνει τον ίδιο μετασχηματισμό ως λυμένο παράδειγμα.

Ο μετασχηματισμός Guttman-Rosler

Ο μετασχηματισμός Schwartzian ταξινομεί arrayrefs, που κοστίζει μια κατανομή arrayref και μια αποαναφορά ανά σύγκριση. Όταν το κλειδί μπορεί να κωδικοποιηθεί ως ένα πρόθεμα *συμβολοσειράς* σταθερού πλάτους, ο μετασχηματισμός Guttman-Rosler (GRT) πάει παραπέρα: πακετάρει το κλειδί και το στοιχείο σε μία μόνο συμβολοσειρά, ταξινομεί αυτές τις συμβολοσειρές με την προεπιλεγμένη συμβολοσειρική `sort` (καθόλου μπλοκ συγκριτή), μετά αφαιρεί το πρόθεμα.

```
my @sorted =
    map  { substr($_, 8) }                              # 3. strip the␣
↪8-char key prefix
    sort                                                # 2. plain␣
↪string sort, no comparator
    map  { pack('N', $_->{priority}) . $_->{name} } # 1. fixed-width␣
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
↪key + payload
   @records;
```

Το όφελος έχει δύο μέρη. Πρώτον, το κλειδί προϋπολογίζεται μία φορά, όπως στον Schwartzian. Δεύτερον - και αυτό είναι το διακριτό πλεονέκτημα του GRT - η `sort` *δεν έχει μπλοκ συγκριτή*, οπότε δεν υπάρχει καθόλου callback ανά σύγκριση. Η προεπιλεγμένη συμβολοσειρική `sort` συγκρίνει τις πακεταρισμένες συμβολοσειρές απευθείας, πράγμα που αποφεύγει εντελώς το κόστος του callback. Η αφαίρεση του callback του συγκριτή έχει μεγαλύτερη σημασία στο pperl ακριβώς επειδή εκείνο το callback θα έτρεχε στον διερμηνευτή.

Το κόστος του GRT είναι ότι το κλειδί πρέπει να είναι κωδικοποιήσιμο ώστε μια συμβολοσειρική σύγκριση κατά byte να παράγει τη σειρά που θέλετε:

- Οι ακέραιοι πρέπει να πακετάρονται big-endian (`pack 'N'` / `pack 'Q>'`) ώστε η σειρά των byte να ταιριάζει με την αριθμητική σειρά, και να μετατοπίζονται ώστε να είναι μη αρνητικοί (η συμβολοσειρική σύγκριση δεν έχει bit προσήμου).
- Οι αριθμοί κινητής υποδιαστολής και οι προσημασμένοι αριθμοί χρειάζονται κωδικοποίηση με μεροληψία (bias) για να ταξινομηθούν σωστά ως byte - αρκετά λεπτεπίλεπτο ώστε ο Schwartzian να είναι συνήθως η καλύτερη επιλογή εκτός αν η ταξινόμηση είναι πραγματικά θερμή.
- Κάθε κλειδί πρέπει να έχει το *ίδιο πλάτος*, αλλιώς το payload ενός κοντού κλειδιού διαρρέει μέσα στη σύγκριση.

Καταφύγετε στον GRT όταν η ταξινόμηση είναι θερμή, το κλειδί είναι μη προσημασμένος ακέραιος ή συμβολοσειρά, και ο Schwartzian εξακολουθεί να εμφανίζεται στο προφίλ. Για οτιδήποτε άλλο, ο Schwartzian είναι απλούστερος και σχεδόν πάντα αρκετά γρήγορος.

Φθηνότεροι συγκριτές χωρίς μετασχηματισμό

Δεν χρειάζεται κάθε αργή ταξινόμηση έναν μετασχηματισμό. Ορισμένες φορές ο συγκριτής είναι απλώς γραμμένος πιο ακριβά απ” όσο χρειάζεται.

- **Συγκρίνετε αριθμούς με `<=>`, συμβολοσειρές με `cmp`.** Η χρήση της `cmp` σε αριθμούς μετατρέπει και τις δύο πλευρές σε συμβολοσειρά σε κάθε σύγκριση· η χρήση της `<=>` σε συμβολοσειρές είναι παγίδα μετατροπής σε αριθμό. Ταιριάξτε τον τελεστή με τα δεδομένα.
- **Διατάξτε τους αποκλεισμούς ισοβαθμίας από τον φθηνότερο πρώτα.** Ένας πολυκλειδικός συγκριτής με `||` σταματά στο πρώτο μη μηδενικό αποτέλεσμα. Βάλτε το διακριτικό, φθηνό κλειδί πρώτο ώστε οι περισσότερες συγκρίσεις να μη φτάνουν ποτέ στο ακριβό:

```
sort { $a->{rank} <=> $b->{rank}          # cheap, decides most␣
↪pairs
    || $a->{name} cmp $b->{name} }        # only when ranks tie
       @rows;
```

- **Ταξινομήστε μία φορά, αντιστρέψτε ξεχωριστά.** Για να ταξινομήσετε φθίνουσα, η αύξουσα ταξινόμηση και η κλήση της `reverse` είναι συχνά σαφέστερη και όχι πιο

αργή από την αντιστροφή του συγκριτή, και κρατά τον συγκριτή στη φθηνότερη μορφή του.

Σειρά αποφάσεων

1. Προφιλάρετε. Αν η *Μέτρηση* δεν τοποθετεί τον χρόνο στην ταξινόμηση, αφήστε την ήσυχη.
2. Αν ο συγκριτής επαναϋπολογίζει ένα κλειδί, εφαρμόστε τον μετασχηματισμό Schwartzian. Αυτό διορθώνει τη μεγάλη πλειονότητα των αργών ταξινομήσεων.
3. Αν η ταξινόμηση είναι ακόμη θερμή και το κλειδί είναι μη προσημασμένος ακέραιος ή συμβολοσειρά, εξετάστε τον μετασχηματισμό Guttman-Rosler για να εγκαταλείψετε εντελώς το callback του συγκριτή.
4. Ξαναμετρήστε μετά από κάθε αλλαγή.

Δείτε επίσης

- `sort` - η αναφορά του τελεστή: συμβόλαιο συγκριτή, `$a`/`$b`, σταθερότητα.
- `map` - το μετασχηματιστικό μισό τόσο της μορφής Schwartzian όσο και της Guttman-Rosler.
- `reverse` - φθίνουσες ταξινομήσεις χωρίς αντιστροφή του συγκριτή.
- *Μέτρηση* - επιβεβαιώστε ότι ο συγκριτής είναι το κόστος προτού τον μετασχηματίσετε.
- *Μεταγλώττιση JIT* - γιατί ένα μπλοκ συγκριτή `sort` δεν μεταγλωττίζεται με JIT, οπότε το κόστος του πληρώνεται πλήρως.

Γράφοντας γρήγορο pperl

Τα άλλα κεφάλαια αυτού του οδηγού αφορούν την αφαίρεση κόστους. Αυτό αφορά την κατάκτηση των επιταχύνσεων που προσφέρει το pperl και που καμία άλλη Perl δεν έχει: το JIT που μεταγλωττίζει τους θερμούς βρόχους σε εγγενή κώδικα, και τον παραλληλοποιητή που απλώνει έναν επιλέξιμο βρόχο σε όλους τους πυρήνες της CPU. Και τα δύο είναι αυτόματα - δεν τα καλείτε, δεν τα σχολιάζετε, ούτε τα διαμορφώνετε. Αλλά ενεργοποιούνται μόνο σε κώδικα *διαμορφωμένο* με τον σωστό τρόπο, και μια μόνο λάθος τοποθετημένη λειτουργία μπορεί να απενεργοποιήσει αθόρυβα οποιοδήποτε από τα δύο.

Αυτό το κεφάλαιο είναι το θετικό πρόγραμμα: γράψτε τον βρόχο έτσι, και ο μεταγλωττιστής τον αναλαμβάνει. Τα κεφάλαια *JIT* και *παράλληλη εκτέλεση* εξηγούν τον μηχανισμό· αυτό σας λέει τι να πληκτρολογήσετε.

Η μορφή που θέλει το JIT

Το JIT στοχεύει **βρόχους** - `for` και `while` - των οποίων τα σώματα είναι αριθμητική και σύγκριση πάνω σε αριθμητικές μεταβλητές. Όταν αναγνωρίσει αυτή τη μορφή, μεταγλωττίζει το σώμα σε εγγενή κώδικα μηχανής και κρατά τις μεταβλητές του βρόχου σε καταχωρητές CPU αντί στη στοίβα του διερμηνευτή. Από εκεί προέρχονται οι επιταχύνσεις τάξης μεγέθους.

Ένας βρόχος που το JIT μεταγλωττίζει καθαρά:

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

Η ακέραια και κινητής υποδιαστολής αριθμητική (+, -, *, /), οι συγκρίσεις (<, >, <=, >=, ==) και η ροή ελέγχου (`if`, `last`) μέσα στον βρόχο μεταγλωττίζονται όλα. Οι εμφωλευμένοι βρόχοι μεταγλωττίζονται επίσης, πολλά επίπεδα βαθιά, με κάθε επίπεδο συνδεδεμένο με το επόμενο. Η φυσική, ιδιωματική μορφή είναι η γρήγορη μορφή - δεν ξετυλίγετε, δεν βγάζετε έξω σε προσωρινές μεταβλητές, ούτε την παραμορφώνετε αλλιώς.

Τι σπρώχνει έναν βρόχο πίσω στον διερμηνευτή

The JIT declines a loop whose body does work it cannot prove it can reproduce exactly. 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 coverage is broad (arithmetic, strings, array and hash elements, push/pop, `map`/`grep`, numeric `sort`, `join`/`split`, regex matches and constant substitutions, calls to small pure subs, and the common `Scalar::Util`/`List::Util` operations - `blessed`, `reftype`, `weaken`, `sum`, `min`, `max` and friends), so the practical disqualifiers are fewer than you might expect:

- **Calls to subs with side effects.** A call compiles when the sub's body is a `my (...) = @_;` prelude plus one pure numeric expression - including recursion. A sub that touches globals, does I/O, uses `return`, or takes references to `@_` declines the loop. Keep hot helpers small and pure and the call costs almost nothing; factoring work OUT of a sub is no longer the advice.
- **I/O.** Ένα `print`, μια ανάγνωση αρχείου, μια κλήση συστήματος - η εργασία που περιορίζεται από το I/O δεν ωφελείται από την εγγενή μεταγλώττιση και δεν μεταγλωττίζεται.
- **Regex beyond the compiled forms.** Scalar matches and `s/pat/constant/` (including `/g`) compile; `s///e`, scalar `m//g`, and patterns with embedded code do not. See *regex performance* for tuning the engine's own work.
- **Magic in any form.** Tied variables, overloaded objects, `local`, taint mode.

Κανένα από αυτά δεν είναι σφάλμα· ο βρόχος απλώς τρέχει με τον ερμηνευμένο τρόπο. Αλλά αν ένας βρόχος που περιμένατε να είναι γρήγορος δεν είναι, τρέξτε τον με `--no-jit` έναντι της προεπιλογής (*Μέτρηση*): αν οι χρόνοι ταιριάζουν, το JIT τον απέρριπτε ήδη. Ορίστε `PPERL_JIT_LOG=/tmp/jit.log` και διαβάστε τις γραμμές `compile skip reason=...` για να δείτε ακριβώς ποια πράξη ευθύνεται (*Μεταγλώττιση JIT*).

Κρατήστε τις συμβολοσειρές έξω από τους αριθμητικούς βρόχους

Το JIT *όντως* χειρίζεται μία λειτουργία συμβολοσειράς - την `.` = συνένωση-ανάθεση, και το άδειασμα μιας συμβολοσειράς με `$x = ""` - μέσω κλήσεων πίσω στο runtime. Έτσι ένας βρόχος χτισίματος συμβολοσειρών εξακολουθεί να μεταγλωττίζεται. Αλλά αυτή είναι η μία περίπτωση όπου η μορφή του βρόχου αλλάζει *ποιοι βελτιστοποιητές* εφαρμόζονται, και είναι ο πολυτιμότερος κανόνας αυτού του κεφαλαίου.

Όταν το JIT δει μια μεταβλητή συμβολοσειράς σε έναν βρόχο, ο βρόχος μεταγλωττίζεται ως **σειριακός** εγγενής βρόχος - αλλά η παράλληλη αποστολή κλείνει. Ο παραλληλοποιητής χρειάζεται το σώμα του βρόχου ελεύθερο από την κοινόχρηστη μεταβλητή κατάσταση που απαιτεί το χτίσιμο συμβολοσειρών, οπότε τη στιγμή που εμφανίζεται μια μεταβλητή συμβολοσειράς, ο βρόχος δεν απλώνεται πλέον σε όλα τα νήματα. Εξακολουθεί να τρέχει ως μεταγλωττισμένος εγγενής κώδικας· απλώς τρέχει σε έναν πυρήνα.

Η πρακτική συνέπεια: μια αριθμητική αναγωγή που *θα* παραλληλοποιούνταν χάνει αυτόν τον παραλληλισμό αν χτίσετε μια συμβολοσειρά στον ίδιο βρόχο.

```
# Parallelizable: pure numeric reduction.
my $total = 0;
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 ";
}
```

Αν χρειάζεστε και τα δύο, χωρίστε την εργασία σε δύο βρόχους. Ο αριθμητικός βρόχος παραλληλοποιείται· ο βρόχος συμβολοσειρών τρέχει ως σειριακός μεταγλωττισμένος βρόχος. Η ανάμειξή τους κοστίζει την παράλληλη επιτάχυνση στο αριθμητικό μισό και δεν κερδίζει τίποτε στο μισό των συμβολοσειρών.

Η μορφή που θέλει ο παραλληλοποιητής

Όταν η παραλληλοποίηση είναι ενεργοποιημένη - είναι ενεργή εξ ορισμού· η `--no-parallel` την απενεργοποιεί - ένας μεταγλωττισμένος με JIT βρόχος αποστέλλεται σε όλα τα νήματα αν η ανάλυση μπορεί να αποδείξει ότι είναι ασφαλής. Δύο συνθήκες έχουν τη μεγαλύτερη βαρύτητα:

Μια αναγωγή διαμορφωμένη ως συσσώρευση-χωρίς-μηδενισμό. Ο παραλληλοποιητής αναγνωρίζει μια μεταβλητή αναγωγής βλέποντάς την να συσσωρεύεται και να μη μηδενίζεται ποτέ μέσα στον βρόχο. Δηλώστε τον συσσωρευτή πριν τον βρόχο, χρησιμοποιήστε την απλή μορφή +=, και ποτέ μην τον επαναρχικοποιείτε μέσα στο σώμα:

```
my $sum = 0;                    # declared outside the loop
for my $x (@data) {
    $sum += score($x);          # accumulate only → recognised as a␣
↪reduction
}
```

Η επαναρχικοποίηση της `$sum` μέσα στον βρόχο, ή η παρεμβολή ενός μηδενισμού, ματαιώνει την αναγνώριση (η ανάλυση αφαιρεί τα μοτίβα μηδενισμού από τα μοτίβα

συσσώρευσης) και ο βρόχος τρέχει σειριακά. Πλήρης λεπτομέρεια στην *ανίχνευση αναγωγής*.

Ένα σώμα βρόχου ελεύθερο από παρατηρήσιμες παρενέργειες. Ο αναλυτής είναι σκόπιμα συντηρητικός. Οποιοδήποτε από αυτά αποκλείει έναν βρόχο από την παράλληλη αποστολή:

- I/O (`print`, open, αναγνώσεις αρχείων)
- Εγγραφές σε μεταβλητές πακέτου/καθολικές
- Κλήσεις υπορουτινών που δεν έχουν αποδειχθεί καθαρές
- Λειτουργίες regex με παρενέργειες (`s///`)

Μια χαμένη παραλληλοποίηση είναι ασφαλής - ο βρόχος απλώς τρέχει σειριακά. Μια *εσφαλμένη* θα ήταν σφάλμα, οπότε η ανάλυση σφάλλει προς την απόρριψη. Αυτή η συντηρητικότητα είναι ο λόγος που η διατήρηση του σώματος του βρόχου μικρού, τοπικού και ελεύθερου από καθολικές μεταβλητές είναι αυτό που το οδηγεί στην παραλληλοποίηση.

map and grep compile too

Οι `map` και `grep` με καθαρά callbacks πάνω σε μια αρκετά μεγάλη συλλογή μπορούν να τρέξουν παράλληλα, με τη σειρά εξόδου διατηρημένη:

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

The block must be a side-effect-free expression over `$_`, loop variables, and constants; mutating `$_` (which writes through to the source array) or producing a boolean as the map VALUE declines. Like all array work, compiled map/grep runs sequentially - array state gates parallel dispatch off - but the native-loop speedup over the interpreted op machinery is itself several-fold.

Όταν η εργασία δεν είναι βρόχος

Δεν είναι κάθε πρόγραμμα ένας αριθμητικός βρόχος, και το JIT και ο παραλληλοποιητής δεν έχουν τίποτε να προσφέρουν σε κώδικα που περιορίζεται από κλήσεις, από το I/O, ή που έχει μορφή συμβολοσειράς. Για αυτόν τον κώδικα ισχύει πλήρως το υπόλοιπο αυτού του οδηγού:

- Επιλέξτε native ενσωματωμένες συναρτήσεις αντί για χειρόγραφους βρόχους όπου υπάρχουν - το `List::Util` τρέχει με ταχύτητα ενσωματωμένης συνάρτησης.
- Εφαρμόστε τους κλασικούς ιδιωματισμούς που εξακολουθούν να ισχύουν από τους *Ιδιωματισμούς*: βγάλτε έξω τα αμετάβλητα, αποφύγετε αντίγραφα, προμεταγλωττίστε regexp.
- Μετασχηματίστε τις ακριβές ταξινομήσεις (*Ταξινόμηση*).
- Καλέστε τη C μέσω του *FFI* μόνο όταν η συχνότητα κλήσης είναι χαμηλή - το κόστος μαρσαλαρίσματος ανά κλήση το καθιστά λάθος εργαλείο μέσα σε έναν θερμό εσωτερικό βρόχο.

Μια λίστα ελέγχου για έναν θερμό βρόχο

1. Είναι το σώμα εκφράσιμη δουλειά - αριθμητική, συμβολοσειρές, πρόσβαση σε στοιχεία, μεταγλωττισμένες μορφές regex, καθαρές κλήσεις υπορουτινών; Αν ναι, το JIT το μεταγλωττίζει. Αν όχι, το `PPERL_JIT_LOG` ονομάζει την πράξη που το αποκλείει - αντιμετωπίστε το αυτό πρώτα.
2. Υπάρχει λειτουργία συμβολοσειράς στον βρόχο; Αν ναι, και ο βρόχος είναι κατά τα άλλα μια αριθμητική αναγωγή, χωρίστε έξω την εργασία με συμβολοσειρές ώστε ο αριθμητικός βρόχος να μπορεί να παραλληλοποιηθεί.
3. Είναι ο συσσωρευτής δηλωμένος έξω από τον βρόχο και μόνο συσσωρεύεται, ποτέ δεν μηδενίζεται; Αυτό είναι που τον κάνει αναγνωρισμένη αναγωγή.
4. Είναι το σώμα ελεύθερο από I/O, καθολικές εγγραφές και μη καθαρές κλήσεις; Αυτό είναι που επιτρέπει στον βρόχο να παραλληλοποιηθεί.
5. Μετρήστε την προεπιλογή έναντι `--no-jit` έναντι `--no-parallel` (*Μέτρηση*) για να επιβεβαιώσετε ποιος βελτιστοποιητής εμπλέκεται.

Δείτε επίσης

- *Μεταγλώττιση JIT* - η σωλήνωση μεταγλώττισης, το μοντέλο τύπου μεταβλητών, η αποθήκευση σε cache.
- *Παράλληλη εκτέλεση* - η ανάλυση αναγωγής, η πύλη παρενεργειών, οι έλεγχοι νημάτων.
- *Μέτρηση* - πώς να επιβεβαιώσετε ότι το JIT και ο παραλληλοποιητής πράγματι εμπλέκονται στον κώδικά σας.
- *Ιδιωματισμοί* - ποιες κλασικές βελτιστοποιήσεις έχουν ακόμη σημασία και ποιες χειρίζεται ήδη ο μεταγλωττιστής.
- *Ταξινόμηση* - το ένα κοινό θερμό μονοπάτι που το JIT δεν φτάνει.

Γρήγορη εκκίνηση: ο δαίμονας

Κάθε κλήση του `pperl` πληρώνει κανονικά το πλήρες κόστος της εκκίνησης: δημιουργία διεργασίας, αρχικοποίηση του διερμηνέα και - το ακριβό μέρος για τα πραγματικά προγράμματα - μεταγλώττιση του σεναρίου σας και κάθε αρθρώματος που εισάγει με `use`. Ο δαίμονας επιτρέπει σε επαναλαμβανόμενες κλήσεις του ίδιου σεναρίου να παρακάμπτουν αυτό το κόστος: μια μόνιμη διεργασία κρατά μια πλήρως μεταγλωττισμένη εικόνα του σεναρίου, και κάθε νέα εκτέλεση είναι ένα fork αυτής της εικόνας που περνά κατευθείαν στην εκτέλεση.

Το χαρακτηριστικό είναι αυστηρά **opt-in**, ανά κλήση. Τίποτα δεν αλλάζει για ένα απλό `pperl script.pl`, και όποτε ο δαίμονας δεν μπορεί να εξυπηρετήσει ένα αίτημα ακριβώς, το pperl καταφεύγει σιωπηλά σε μια κανονική ψυχρή εκτέλεση. Η ορθότητα δεν εξαρτάται ποτέ από τον δαίμονα· μόνο η ταχύτητα εξαρτάται.

Γρήγορο ξεκίνημα

```
pperl --daemon=start                    # start the daemon (background)

PPERL_DAEMON=2 pperl report.pl          # first run: compiles, caches
PPERL_DAEMON=2 pperl report.pl          # every further run: fork + run

pperl --daemon=status                   # what is cached, what is live
pperl --daemon=stop                     # shut it down
```

Η πρώτη opt-in εκτέλεση ενός σεναρίου χτίζει το *πρότυπό* του: ο δαίμονας μεταγλωττίζει το σενάριο (συμπεριλαμβανομένων όλων των αρθρωμάτων που εισάγονται με `use` και των μπλοκ `BEGIN`) και κρατά το αποτέλεσμα μόνιμο. Κάθε μεταγενέστερη εκτέλεση του ίδιου σεναρίου κάνει fork το πρότυπο και απλώς εκτελεί.

Πότε αποδίδει

Η εξοικονόμηση είναι ο χρόνος μεταγλώττισης του σεναρίου σας: η ανάλυση, συν η φόρτωση αρθρωμάτων, συν η εργασία στο `BEGIN`. Για ένα σενάριο που φορτώνει μια χούφτα αρθρώματα, αυτό είναι συνήθως το κυρίαρχο μέρος της εκκίνησης, και μια εκτέλεση από πρότυπο μπορεί να είναι πολλαπλάσια ταχύτερη από άκρη σε άκρη. Για ένα ασήμαντο σενάριο που δεν φορτώνει τίποτα, η εξοικονόμηση ισούται περίπου με το ίδιο το overhead του δαίμονα και το αποτέλεσμα είναι ισοπαλία - ποτέ ουσιαστική απώλεια.

Όπως παντού σε αυτόν τον οδηγό: μετρήστε το δικό σας σενάριο, στο δικό σας μηχάνημα, πριν βγάλετε συμπεράσματα - δείτε *Μέτρηση*.

Τι σημαίνει η εκτέλεση από πρότυπο

Μια εκτέλεση από πρότυπο εκτελεί μόνο τη *φάση εκτέλεσης* του σεναρίου σας. Η μεταγλώττιση έγινε ήδη, μία φορά, όταν χτίστηκε το πρότυπο. Το pperl εγγυάται ότι τα ακόλουθα είναι φρέσκα και ορθά για κάθε εκτέλεση:

- `@ARGV`, `%ENV`, ο τρέχων κατάλογος, umask
- η τυπική είσοδος, έξοδος και σφάλμα (οι ανακατευθύνσεις και οι σωληνώσεις σας)
- το id της διεργασίας `$$`
- οι κωδικοί εξόδου και ο θάνατος από σήμα (ένα σκοτωμένο σενάριο σκοτώνει το `pperl` που το κάλεσε με τον ίδιο τρόπο)
- ο χειριστής `__DATA__` - κάθε εκτέλεση τον διαβάζει από την αρχή, ανεξάρτητα από τις άλλες εκτελέσεις

Αυτό που **δεν** ξαναγίνεται είναι οτιδήποτε έκανε ο κώδικάς σας κατά τη μεταγλώττιση. Τα μπλοκ `BEGIN` και ο κώδικας αρθρωμάτων που εκτελέστηκε κατά τον χρόνο `use` έτρεξαν μία φορά όταν χτίστηκε το πρότυπο· οι παρενέργειές τους είναι μέρος της εικόνας που κληρονομεί κάθε εκτέλεση. Συνέπειες που αναλαμβάνετε όταν εντάσσετε ένα σενάριο:

- Το `$^T` (ο χρόνος έναρξης του σεναρίου) και η κατάσταση του τυχαίου σπόρου χρονολογούνται από τη δημιουργία του προτύπου. Αν αυτό έχει σημασία για το σενάριό σας, ξανασφραγίστε τα στην αρχή της φάσης εκτέλεσης:

```
INIT { $^T = time; srand(); }
```

- Οι τιμές που διαβάστηκαν από το %ENV *κατά τη μεταγλώττιση* (μέσα σε BEGIN ή κατά τον χρόνο use) ήταν οι τιμές της εκτέλεσης που έχτισε το πρότυπο. Διαβάστε τη διαμόρφωση κατά τον χρόνο εκτέλεσης, ή σε ένα μπλοκ INIT, αν πρέπει να παρακολουθεί το περιβάλλον που κάλεσε.
- Οι χειριστές αρχείων, οι υποδοχές ή οι συνδέσεις βάσης δεδομένων που ανοίγονται κατά τη μεταγλώττιση μοιράζονται με κάθε εκτέλεση. Ανοίγετε τις συνδέσεις κατά τον χρόνο εκτέλεσης - ο ίδιος κανόνας που ισχύει από παλιά για τους διακομιστές εφαρμογών με προφόρτωση.

Αν ένα σενάριο δεν μπορεί να ικανοποιήσει αυτό το συμβόλαιο, απλώς μην το εντάξετε· το PPERL_DAEMON=2 ισχύει ανά κλήση, όχι καθολικά.

Η επεξεργασία του σεναρίου και η κρυφή μνήμη

Ο δαίμονας αντιλαμβάνεται όταν αλλάζει το ίδιο το αρχείο του σεναρίου και ξαναχτίζει το πρότυπο αυτόματα στην επόμενη εκτέλεση - η επεξεργασία του σεναρίου σας δεν σας σερβίρει ποτέ μια ξεπερασμένη εικόνα.

Οι αλλαγές σε *αρθρώματα* που φορτώνει το σενάριο δεν ανιχνεύονται αυτόματα. Αφού αλλάξετε ένα άρθρωμα, απορρίψτε το επηρεαζόμενο πρότυπο με το χέρι:

```
pperl --daemon-reset=report.pl   # forget this script's template
pperl --daemon-reset             # forget all templates
```

Τα πρότυπα είναι μια φραγμένη κρυφή μνήμη: το πολύ --daemon-max-scripts από αυτά (προεπιλογή 100), που εκδιώκονται με βάση τη μεγαλύτερη αχρησία, και το καθένα λήγει μετά το χρονικό όριο αδράνειας του δαίμονα (--daemon-idle, προεπιλογή 900 δευτερόλεπτα).

Πότε το pperl καταφεύγει σε ψυχρή εκτέλεση

Ο δαίμονας αρνείται - και ο πελάτης καταφεύγει σε μια κανονική εκτέλεση, σιωπηλά και ορθά - όποτε δεν θα μπορούσε να αναπαραγάγει την κλήση ακριβώς:

- δεν τρέχει κανένας δαίμονας για αυτό το εκτελέσιμο pperl
- οι μεταβλητές περιβάλλοντος που διαμορφώνουν την εκκίνηση του διερμηνέα διαφέρουν από αυτές του δαίμονα (PERL_HASH_SEED, PERL_PERTURB_KEYS, PERL_RAND_SEED, PERL_INTERNAL_RAND_SEED, PERL_UNICODE, LANG, οποιαδήποτε LC_*, οποιαδήποτε MALLOC_*)
- οι σημαίες της γραμμής εντολών διαφέρουν από αυτές με τις οποίες χτίστηκε το πρότυπο (ένα πρότυπο αποθηκεύει μία διαμόρφωση)
- το πρότυπο χτίζεται ακόμη από μια ταυτόχρονη πρώτη εκτέλεση
- η κλήση είναι ένα μονόγραμμο -e (τα πρότυπα αποθηκεύουν σε κρυφή μνήμη αρχεία σεναρίων)
- είναι σε ισχύ το --interactive ή το --stats

Ασφάλεια και ταυτότητα

Ο δαίμονας εξυπηρετεί ακριβώς έναν χρήστη: η υποδοχή του βρίσκεται σε έναν κατάλογο με δικαιώματα μόνο για τον ιδιοκτήτη και το user id του συνομιλητή κάθε σύνδεσης επαληθεύεται. Εξυπηρετεί επίσης ακριβώς ένα *build* του pperl: η υποδοχή είναι κλειδωμένη στην ταυτότητα του εκτελέσιμου (το `pperl --build-id` την εκτυπώνει), οπότε μετά από μια αναβάθμιση του pperl ο παλιός δαίμονας απλώς δεν επικοινωνείται ποτέ ξανά και τερματίζει στο χρονικό όριο αδράνειάς του.

Περιορισμοί

- Ο έλεγχος εργασιών είναι κατά προσέγγιση: το σενάριο που τρέχει είναι παιδί του δαίμονα, όχι του κελύφους σας. Τα σήματα που στέλνετε στο `pperl` (συμπεριλαμβανομένου του `Ctrl-C`) προωθούνται στο σενάριο, αλλά το `Ctrl-Z` αναστέλλει μόνο το `pperl` στο προσκήνιο, όχι τη διεργασία του σεναρίου.
- Ένα πρότυπο ανά διαδρομή σεναρίου, που κρατά μία διαμόρφωση σημαιών. Η κλήση του ίδιου σεναρίου με διαφορετικές σημαίες καταφεύγει σε μια ψυχρή εκτέλεση αντί να ξαναχτίσει το πρότυπο.
- Το `PPERL_DAEMON=1` (ή το `--via-daemon`) υπάρχει ως μειωμένη λειτουργία που κάνει fork έναν προαρχικοποιημένο διερμηνέα αλλά εξακολουθεί να μεταγλωττίζει το σενάριο σε κάθε εκτέλεση. Δεν εξοικονομεί σχεδόν τίποτα και είναι κυρίως χρήσιμο για τη δοκιμή του ίδιου του δαίμονα· χρησιμοποιήστε το `PPERL_DAEMON=2` για το πραγματικό όφελος.
- **Μέτρηση** - τα εργαλεία που πράγματι λειτουργούν υπό το pperl. Οι upstream προφίλερ XS (`Devel::NYTProf`) δεν φορτώνονται· το αρχείο καταγραφής JIT, ο προφίλερ της πλατφόρμας και ο εκτελεστής `bench/run-perlbench` λειτουργούν. Πώς να διαβάζετε την έξοδό τους και πώς να κάνετε A/B σύγκριση δύο εκδόσεων μιας ρουτίνας.
- **Ιδιωματισμοί** - μια διαλογή των κλασικών ιδιωματισμών απόδοσης της Perl. Ο καθένας φέρει ετικέτα: εξακολουθεί να ισχύει, κατέστη άνευ αντικειμένου από το JIT ή τον παραλληλοποιητή, ή απέκτησε νέα σημασία λόγω του τρόπου που μεταγλωττίζει το pperl. Οι ιδιωματισμοί που ήταν λαϊκή σοφία στην upstream Perl δεν είναι όλοι λαϊκή σοφία εδώ.
- **Ταξινόμηση** - η `sort` σε βάθος: ο μετασχηματισμός Schwartzian, ο μετασχηματισμός Guttman-Rosler, το κόστος της υπορουτίνας `sort`, και πότε ο συγκριτής είναι το σημείο συμφόρησης.
- **Γράφοντας γρήγορο pperl** - το θετικό πρόγραμμα: πώς να δομήσετε έναν θερμό βρόχο ώστε να τον μεταγλωττίσει το JIT, πώς να διαμορφώσετε μια αναγωγή ώστε να τη διεκδικήσει ο παραλληλοποιητής, και το ένα λάθος - η ανάμειξη εργασίας με συμβολοσειρές μέσα σε έναν αριθμητικό βρόχο - που απενεργοποιεί αθόρυβα και τα δύο.
- **Fast start-up: the daemon** - opt-in start-up acceleration for repeatedly invoked scripts: a resident daemon keeps each script's compiled image and every run forks it, skipping parsing and module loading. What it guarantees per run, the preload-safety contract your script signs, and when pperl silently falls back to a normal cold start.

4.4.4 Τι δεν καλύπτει αυτός ο οδηγός

Οι μηχανισμοί βρίσκονται αλλού και αυτός ο οδηγός δεν τους επαναλαμβάνει:

- *Μεταγλώττιση JIT* - τι μεταγλωττίζεται, το μοντέλο cache, η ανάλυση τύπου μεταβλητών.
- *Παράλληλη εκτέλεση* - η ανάλυση αναγωγής, η πύλη παρενεργειών, οι έλεγχοι νημάτων του CLI.
- *Απόδοση regex* - οπισθοχώρηση, καταστροφικά μοτίβα, ο βελτιστοποιητής της μηχανής. Η ταχύτητα των regex είναι ξεχωριστός κλάδος· τίποτε σε αυτόν τον οδηγό δεν τον επαναλαμβάνει.
- *Native αρθρώματα* - γιατί το `List::Util` και τα συναφή τρέχουν με ταχύτητα ενσωματωμένης συνάρτησης χωρίς στρώμα XS.
- *Auto-FFI* - κλήση της C χωρίς XS, και το κόστος μαρσαλαρίσματος ανά κλήση που το καθιστά λάθος εργαλείο για έναν θερμό εσωτερικό βρόχο.

4.4.5 Μια σημείωση για τους αριθμούς

Οι συγκεκριμένες επιταχύνσεις σε αυτόν τον οδηγό αναπαράγονται από τα κεφάλαια αρχιτεκτονικής και ταυτοχρονισμού, τα οποία φέρουν τα μετρημένα μεγέθη και το benchmark που τα παρήγαγε. Όπου αυτός ο οδηγός δηλώνει μια επιτάχυνση χωρίς αριθμό, η ποιοτική συμπεριφορά είναι τεκμηριωμένη, αλλά το ακριβές μέγεθος εξαρτάται από το μηχάνημά σας, τα δεδομένα σας και την κατασκευή σας - τρέξτε το `bench/run-perlbench` και εμπιστευτείτε τη δική σας μέτρηση έναντι οποιουδήποτε τυπωμένου λόγου.

4.5 Αντικειμενοστραφής προγραμματισμός

Η Perl διαθέτει δύο συστήματα αντικειμένων που συνυπάρχουν: την **σύγχρονη σύνταξη `class`/`field`/`method`** που εισήχθη στην 5.38 (κωδική ονομασία *Corinna*) και το **κλασικό μοντέλο `bless`/`@ISA`** που προηγείται κατά τρεις δεκαετίες. Η pperl υποστηρίζει και τα δύο, και τα δύο εμφανίζονται σε πραγματικό κώδικα που θα διαβάσετε και θα συντηρήσετε.

Η σύντομη εκδοχή:

- Για **νέο κώδικα**, χρησιμοποιήστε `class`. Η σύνταξη είναι πρώτης τάξης, ο κατασκευαστής δημιουργείται για εσάς, τα πεδία είναι γνήσια ιδιωτικά, και η απλή κληρονομικότητα έχει καθαρή δηλωτική μορφή.
- Για **παλαιό κώδικα**, κατανοήστε τα `bless`, `@ISA` και τη σύμβαση `$self = shift`. Θα τα συναντήσετε σε κάθε βάση κώδικα πριν την 5.38, σε κάθε κύρια διανομή του CPAN και σε μεγάλο μέρος της ίδιας της Perl πυρήνα.
- Τα **Moose, Moo, Class::Accessor, Object::Pad** είναι τα οικοσυστήματα του CPAN που κάλυψαν τις τραχιές γωνίες του κλασικού μοντέλου όσο το χαρακτηριστικό του πυρήνα ήταν ακόμη υπό σχεδίαση. Αποτελούν ιστορικό κεφάλαιο· ο νέος κώδικας στρέφεται στο `class` του πυρήνα.

Αυτό το δέντρο οδηγιών καλύπτει και τα δύο συστήματα από άκρη σε άκρη. Διαβάστε τα κεφάλαια με τη σειρά σε μια πρώτη ανάγνωση ή μεταβείτε σε αυτό που ταιριάζει στον κώδικα που έχετε μπροστά σας.

4.5.1 Ποιο κεφάλαιο χρειάζεστε

- *Σύγχρονες κλάσεις* - `class`, `field`, `method`, `ADJUST`, `:param`, `:reader`. Ο συνιστώμενος τρόπος συγγραφής αντικειμενοστραφούς κώδικα στην pperl.
- *Κλασική αντικειμενοστρέφεια* - `bless`, `@ISA`, `sub new`, accessors με το χέρι, `SUPER::`. Το μοντέλο που θα συντηρείτε σε υπάρχοντα κώδικα.
- *Κληρονομικότητα και επίλυση μεθόδων* - πώς λειτουργεί η αναζήτηση μεθόδου, `isa`, `can`, `DOES`, MRO και οι παγίδες της πολλαπλής κληρονομικότητας.
- *Ρόλοι και ανάθεση* - πρότυπα σύνθεσης για συμπεριφορά κοινή σε άσχετες κλάσεις.
- *Μετάβαση από κλασική σε σύγχρονη* - μια μηχανική συνταγή για τη μετατροπή μιας κλάσης βασισμένης σε `bless` στο χαρακτηριστικό `class`, και τα σημεία όπου η μετάφραση δεν είναι μηχανική.

4.5.2 Πότε να χρησιμοποιείτε αντικειμενοστρέφεια

Η αντικειμενοστρέφεια είναι εργαλείο, όχι ύφος. Στραφείτε σε αυτήν όταν:

- Το πρόβλημα έχει μικρό λεξιλόγιο ουσιαστικών (χρήστης, συνεδρία, αίτημα, εγγραφή) και μεγαλύτερο λεξιλόγιο ρημάτων που ενεργούν σε καθένα.
- Ένα κομμάτι δεδομένων φέρει αναλλοίωτες ιδιότητες που πρέπει να επιβάλλονται σε ένα σημείο του κώδικα, όχι σε κάθε σημείο κλήσης.
- Αναμένετε ότι η υλοποίηση πίσω από ένα σύνολο λειτουργιών θα αλλάξει, ενώ οι ίδιες οι λειτουργίες θα παραμείνουν σταθερές.
- Μοντελοποιείτε μια κατηγορία με παραλλαγές και θέλετε ο πολυμορφισμός να επιλέγει τη σωστή συμπεριφορά.

Μη στρέφεστε στην αντικειμενοστρέφεια όταν:

- Το πρόγραμμα είναι μικρό, γραμμικό και χωρίς επαναχρησιμοποιήσιμα ουσιαστικά.
- Ένα απλό hash από hashes ή μια closure αρκούν.
- Ο μόνος λόγος για να γράψετε μια κλάση είναι «μου φάνηκε πιο επαγγελματικό». Ένα σενάριο δώδεκα γραμμών δεν χρειάζεται κλάση.

4.5.3 Σημείωση για την εμβέλεια

Αυτό το δέντρο είναι εγχειρίδιο εκμάθησης, όχι αναφορά. Οι εξαντλητικοί πίνακες σύνταξης βρίσκονται στις σελίδες αναφοράς ανά δεσμευμένη λέξη:

- `class`
- `field`
- `method`
- `bless`
- `ref`
- `isa`

Συνδεθείτε σε αυτές όποτε χρειάζεστε την πλήρη επιφάνεια υπογραφής· αυτό το δέντρο σάς δίνει την επισκόπηση του προγραμματιστή που εργάζεται.

Σύγχρονες κλάσεις

Το χαρακτηριστικό `class` είναι ο συνιστώμενος τρόπος συγγραφής αντικειμενοστραφούς κώδικα στην pperl. Αντικαθιστά τον χειροποίητο κατασκευαστή, τη χειροκίνητη δημιουργία accessor, τη λογιστική των hash-slot και τον χορό του `use parent` με μία ενιαία δηλωτική σύνταξη. Εσείς περιγράφετε *τι* είναι η κλάση· το runtime οργανώνει *πώς* υλοποιείται.

Αυτό το κεφάλαιο διασχίζει το χαρακτηριστικό όπως θα το συναντούσε ένας προγραμματιστής εν δράσει: πρώτα μια ελάχιστη κλάση, μετά πεδία και attributes, μετά μεθόδους, μετά hooks κατασκευής και τέλος κληρονομικότητα. Κάθε ενότητα είναι σύντομη σκόπιμα· όταν θέλετε την πλήρη επιφάνεια μιας δεσμευμένης λέξης, ακολουθήστε τη διασταυρωμένη σύνδεση προς τη σελίδα αναφοράς της.

Ενεργοποίηση του χαρακτηριστικού

The feature is flagged experimental in Perl 5.44 and pperl mirrors that status. Every file that uses it opens with:

```
use v5.38;
use feature 'class';
no warnings 'experimental::class';
```

Η γραμμή `use v5.38` από μόνη της ενεργοποιεί ήδη το χαρακτηριστικό `class`, αλλά η ρητή δήλωση κάνει την εξάρτηση ορατή σε όποιον διαβάζει το αρχείο.

Μια ελάχιστη κλάση

```
use v5.38;
use feature 'class';
no warnings 'experimental::class';

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
```

Τρία πράγματα που πρέπει να προσέξετε:

- Δεν υπάρχει `sub new`. Το runtime δημιουργεί τον κατασκευαστή από τις δηλώσεις `field`. Ένα `sub new` γραμμένο από τον χρήστη μέσα σε σώμα `class` είναι σφάλμα μεταγλώττισης σκόπιμα - ο δημιουργημένος γνωρίζει πώς να συνδέσει τα `:param`, να εκτελέσει αρχικοποιητές και να ενεργοποιήσει τα μπλοκ `ADJUST` με τη σωστή σειρά.

- Τα πεδία αναφέρονται με το *δηλωμένο όνομά* τους μέσα στις μεθόδους, όχι μέσω του `$self->{x}`. Η αποθήκευση του στιγμιότυπου είναι γνήσια ιδιωτική· κανένας κώδικας έξω από το σώμα της κλάσης δεν μπορεί να την πειράξει.
- Η δεσμευμένη λέξη `method` φροντίζει για το `$self`. Δεν γράφετε ποτέ `my $self = shift`.

Πεδία

Ένα `field` είναι αποθήκευση ανά στιγμιότυπο. Κάθε αντικείμενο της κλάσης παίρνει τη δική του θυρίδα· κάθε μέθοδος και κάθε μπλοκ `ADJUST` βλέπει το πεδίο σαν να ήταν λεξιλογική μεταβλητή.

```
class Counter {
    field $value = 0;
    field $step  = 1;

    method tick  { $value += $step }
    method value { $value }
}
```

Τα πεδία δέχονται βαθμωτά, πίνακες και hashes:

```
class Registry {
    field $name;
    field @items;
    field %seen;
}
```

Attributes σε πεδία

Τα attributes των πεδίων αφαιρούν τη δηλωτική δουλειά από τα χέρια σας.

- **`:param`** - συνδέει αυτό το πεδίο με ένα όρισμα του κατασκευαστή. Αν ο καλών το παραλείψει, το πεδίο είτε μένει χωρίς τιμή είτε παίρνει τη δηλωμένη προεπιλογή.
- **`:param(alt)`** - συνδέει με διαφορετικό όνομα στον κατασκευαστή.
- **`:reader`** - δημιουργεί έναν accessor μόνο για ανάγνωση με το όνομα του πεδίου χωρίς το sigil.
- **`:reader(custom)`** - δημιουργεί έναν reader με διαφορετικό όνομα.
- **`:writer`** - δημιουργεί έναν 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
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
$u->set_email('ada@analytical.org');
say $u->role;                                   # guest
```

Προεπιλεγμένες τιμές

Οι προεπιλογές χρησιμοποιούν έναν από τρεις τελεστές, καθένας με διαφορετικό κανόνα:

- **=** - εφαρμόζεται όταν ο καλών *παράλειψε* εντελώς την παράμετρο.
- **//=** - εφαρμόζεται επίσης όταν ο καλών έδωσε undef.
- **||=** - εφαρμόζεται επίσης όταν ο καλών έδωσε οποιαδήποτε ψευδή τιμή.

```
class Window {
    field $title :param         = 'untitled';
    field $width :param         //= 640;
    field $shown :param         ||= 1;
}
```

Επιλέξτε = εκτός αν έχετε συγκεκριμένο λόγο να επιτρέψετε τις ισχυρότερες μορφές. Η σιωπηρή προαγωγή του undef ή του 0 σε προεπιλογή είναι μαγνήτης σφαλμάτων.

Τα πεδία δεν είναι λεξιλογικές μεταβλητές με τη συνήθη έννοια

Τα πεδία μοιάζουν με λεξιλογικές μεταβλητές μέσα στο σώμα μιας μεθόδου, αλλά έχουν αυστηρότερους κανόνες:

- Δεν μπορείτε να κάνετε our ένα πεδίο, δεν μπορείτε να κάνετε local ένα πεδίο και δεν μπορείτε να πάρετε αναφορά σε ένα πεδίο και να την εξάγετε.
- Τα πεδία είναι ιδιωτικά στην κλάση. Οι υποκλάσεις **δεν** κληρονομούν τα πεδία ενός γονέα· αν μια υποκλάση χρειάζεται την κατάσταση του γονέα, τη ζητάει μέσω μεθόδου.
- Ο αρχικοποιητής ενός πεδίου εκτελείται πριν συνδεθεί το $self. Μέσα σε field $x = EXPR, μπορείτε να χρησιμοποιήσετε προγενέστερα πεδία με το όνομά τους και μπορείτε να χρησιμοποιήσετε το __CLASS__, αλλά δεν μπορείτε να χρησιμοποιήσετε το $self. Οτιδήποτε χρειάζεται το πλήρως κατασκευασμένο στιγμιότυπο ανήκει σε ένα μπλοκ ADJUST.

Μέθοδοι

Μια method είναι μια υπορουτίνα που συνδέει αυτόματα το $self και διαβάζει τα πεδία της περικλείουσας κλάσης με το όνομά τους.

```
class File {
    field $path :param :reader;

    method slurp {
        open my $fh, '<', $path or die "open $path: $!";
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
        local $/;
        return <$fh>;
    }

    method rename ($new) {
        rename $path, $new or die "rename: $!";
        $path = $new;
    }
}
```

Οι μέθοδοι χρησιμοποιούν αυτόματα signatures - το `use feature 'signatures'` υπονοείται μέσα σε σώμα κλάσης. Το `$self` είναι το σιωπηρό πρώτο όρισμα της μεθόδου και **δεν** εμφανίζεται στο signature.

Ιδιωτικές μέθοδοι

Μια μέθοδος που δηλώνεται με `my` είναι λεξιλογική στο σώμα της κλάσης και καλείται μέσω του τελεστή `->&`:

```
class Safe {
    field $secret :param;

    my method check ($n) { $n == $secret }

    method unlock ($n) {
        return $self->&check($n) ? 'open' : 'denied';
    }
}
```

Η `my method` είναι ό,τι πιο κοντινό έχει η pperl σε αυστηρά ιδιωτική μέθοδο· κανείς εξωτερικός καλών δεν μπορεί να την προσεγγίσει επειδή το όνομα δεν υπάρχει καθόλου στο πακέτο.

Hooks κατασκευής: ADJUST

Ένα μπλοκ `ADJUST` εκτελείται μετά την αρχικοποίηση των πεδίων, με το `$self` ήδη συνδεδεμένο. Είναι το κατάλληλο σημείο για:

- Επικύρωση μετά την κατασκευή.
- Παράγωγα πεδία που εξαρτώνται από άλλα πεδία.
- Άνοιγμα ενός handle ή σύνδεση σε έναν πόρο του οποίου η διάρκεια ζωής ταυτίζεται με αυτή του αντικειμένου.

```
class TempFile {
    field $prefix :param = 'tmp';
    field $path;
    field $fh;

    ADJUST {
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
        $path = "/tmp/$prefix-$$-" . int(rand 1_000_000);
        open $fh, '>', $path or die "open $path: $!";
    }

    method write ($data) { print {$fh} $data }
    method path { $path }
}
```

Πολλαπλά μπλοκ ADJUST σε μία κλάση εκτελούνται με τη σειρά δήλωσης. Σε αλυσίδα κληρονομικότητας, τα μπλοκ ADJUST του γονέα εκτελούνται πριν εκείνα του παιδιού.

Κληρονομικότητα

Μια κλάση κληρονομεί από το πολύ έναν γονέα χρησιμοποιώντας το attribute `:isa`:

```
class Shape {
    field $colour :param :reader = 'black';
    method area { 0 }
}

class Circle :isa(Shape) {
    field $radius :param :reader;
    method area { 3.14159265 * $radius ** 2 }
}
```

Κληρονομούνται μόνο οι **μέθοδοι**. Τα πεδία είναι αποθήκευση ανά κλάση. Μια υποκλάση που χρειάζεται την κατάσταση του γονέα τη ζητάει μέσω ενός reader:

```
class Labelled :isa(Shape) {
    field $label :param :reader;
    method describe { "$label (" . $self->colour . ')' }
}
```

Η πολλαπλή κληρονομικότητα δεν υποστηρίζεται. Το `:isa(A, B)` είναι συντακτικό σφάλμα, και μια κλάση δεν μπορεί να φέρει περισσότερα από ένα attribute `:isa`. Το *κεφάλαιο για την κληρονομικότητα* εξηγεί γιατί και καλύπτει την εναλλακτική που βασίζεται σε ρόλους.

Κλήση προς τα πάνω στην αλυσίδα

Το `SUPER::` συνεχίζει να λειτουργεί, ακριβώς όπως σε κλάσεις βασισμένες σε `bless`:

```
class Dog :isa(Animal) {
    method introduce {
        $self->SUPER::introduce;
        say "and I can fetch";
    }
}
```

`__CLASS__`

Μέσα σε αρχικοποιητή πεδίου ή σε σώμα μεθόδου, το `__CLASS__` είναι η κλάση που κατασκευάζεται εκείνη τη στιγμή. Σε υποκλάση, το `__CLASS__` αναφέρεται στην υποκλάση ακόμη και μέσα σε κληρονομημένη μέθοδο ή σε κληρονομημένο αρχικοποιητή πεδίου. Χρησιμοποιήστε το για hooks τύπου factory:

```
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
```

Διαλειτουργικότητα με την κλασική αντικειμενοστρέφεια

Μια κλάση δηλωμένη με `class` είναι ένα κανονικό πακέτο. Κώδικας που χρησιμοποιεί `ref`, τον τελεστή `isa`, `UNIVERSAL::isa` και `UNIVERSAL::can` συνεχίζει να λειτουργεί:

```
my $c = Circle->new(radius => 2);
say ref $c;                                 # Circle
say $c isa Shape ? 'yes' : 'no';            # yes
say $c->can('area') ? 'yes' : 'no';         # yes
```

Μια κλασική κλάση μπορεί να γίνει υποκλάση μιας σύγχρονης κλάσης και αντίστροφα, αρκεί η κλασική πλευρά να περνά μέσα από μεθόδους (ποτέ μέσω `$self->{fieldname}`) - δείτε το *κεφάλαιο μετάβασης* για την πλήρη ιστορία διαλειτουργικότητας.

Οριακές περιπτώσεις που αξίζει να γνωρίζετε

- **Μη γράφετε `sub new`.** Ο κατασκευαστής δημιουργείται αυτόματα. Ένα `new` γραμμένο από τον χρήστη μέσα σε σώμα `class` απορρίπτεται κατά τη μεταγλώττιση.
- **Αντιμετωπίστε το `@ISA` ως μόνο για ανάγνωση** για κλάσεις δηλωμένες με `class`. Η τροποποίηση κατά τον χρόνο εκτέλεσης είναι απροσδιόριστη συμπεριφορά.
- **Το `$self` είναι μόνο για ανάγνωση** μέσα σε μέθοδο. Η ανάθεση στο `$self` είναι σφάλμα χρόνου εκτέλεσης.
- **Το package δεν μπορεί να ξανανοίξει μια κλάση.** Μόλις ένας χώρος ονομάτων δηλωθεί με `class`, μια μεταγενέστερη δήλωση `package` για το ίδιο όνομα είναι σφάλμα μεταγλώττισης, και αντίστροφα.
- **Εμβέλεια μορφής εντολής.** Το `class Foo;` καταναλώνει το υπόλοιπο του περικλείοντος μπλοκ, συνήθως το υπόλοιπο του αρχείου. Μια επόμενη `class` ή

`package` τερματίζει το σώμα· δεν υπάρχει τρόπος να κλείσετε πρόωρα μια κλάση σε μορφή εντολής.

Πού να πάτε στη συνέχεια

- *Κληρονομικότητα και επίλυση μεθόδων* - η πλήρης ιστορία του πώς λειτουργεί η αναζήτηση μεθόδου σε αλυσίδες `:isa` και σε μεικτές κλασικές/σύγχρονες ιεραρχίες.
- *Ρόλοι και ανάθεση* - η συνιστώμενη απάντηση στο «θέλω να μοιραστώ συμπεριφορά μεταξύ άσχετων κλάσεων».
- *Μετάβαση από κλασική σε σύγχρονη* - αν κοιτάτε μια κλάση βασισμένη σε `bless` και αναρωτιέστε πώς θα έμοιαζε το ισοδύναμο μπλοκ `class`.

Κλασική αντικειμενοστρέφεια

Πριν φτάσει το χαρακτηριστικό `class` στην 5.38, η αντικειμενοστρέφεια της Perl ήταν χτισμένη πάνω σε τρεις πρωτογενείς οντότητες: ένα πακέτο, μια αναφορά και τη συνάρτηση `bless`. Αυτές οι τρεις οντότητες υπάρχουν ακόμη - η σύγχρονη δεσμευμένη λέξη `class` κάθεται πάνω στην ίδια μηχανική - και κάθε βάση κώδικα πριν την 5.38, καθώς και το μεγαλύτερο μέρος του CPAN, τις χρησιμοποιεί ακόμη απευθείας.

Αυτό το κεφάλαιο είναι για την ανάγνωση και τη συντήρηση τέτοιου κώδικα. Αν γράφετε νέα κλάση, ξεκινήστε από το *κεφάλαιο της σύγχρονης κλάσης*.

Οι τρεις πρωτογενείς οντότητες

1. **Ένα πακέτο** ορίζει έναν χώρο ονομάτων. Οποιοδήποτε πακέτο μπορεί να είναι κλάση· η διαφορά είναι αμιγώς στον τρόπο χρήσης του.
2. **Μια αναφορά** κρατάει τα δεδομένα του στιγμιότυπου. Η σύμβαση είναι μια αναφορά hash, αλλά οποιοσδήποτε τύπος αναφοράς λειτουργεί.
3. **Το `bless`** ετικετοποιεί τον αναφερόμενο με ένα όνομα κλάσης, έτσι ώστε η αποστολή μεθόδων μέσω της αναφοράς να αναζητά subs σε αυτό το πακέτο.

Αυτό είναι όλο το μοντέλο. Όλα τα υπόλοιπα - κατασκευαστές, accessors, κληρονομικότητα, ρόλοι - είναι σύμβαση χτισμένη από πάνω.

Μια ελάχιστη κλάση

```
package File;

sub new {
    my ($class, %args) = @_;
    my $self = {
        path    => $args{path},
        content => $args{content},
    };
    return bless $self, $class;
}
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
sub path    { $_[0]{path} }
sub content { $_[0]{content} }

sub print_info {
    my $self = shift;
    print "This file is at $self->{path}\n";
}

1;
```

Βασικές συμβάσεις, καθεμιά κρίσιμη:

- **Το `sub new`** είναι ο κατασκευαστής. Είναι μια συνηθισμένη μέθοδος· η γλώσσα δεν διαθέτει δεσμευμένη λέξη κατασκευαστή. Από σύμβαση ονομάζεται `new` και επιστρέφει την blessed αναφορά.
- **Το `my ($class, %args) = @_`** αποπακετάρει πρώτα τον επικαλούμενο και μετά τα ονομασμένα ορίσματα. Μια κλήση μεθόδου κλάσης `File->new(path => ...)` περνάει το όνομα της κλάσης ως πρώτο όρισμα.
- **Το `bless $self, $class`** προσαρτά την ετικέτα της κλάσης. Η χρήση του `$class` (του πραγματικού επικαλούμενου) αντί ενός σταθερά κωδικοποιημένου `'File'` είναι αυτό που καθιστά τον κατασκευαστή *ασφαλή για κληρονομικότητα*: μια υποκλάση που κληρονομεί τη `new` θα κάνει bless στο όνομα της υποκλάσης.
- **Το `return` συνήθως παραλείπεται** μετά το `bless` επειδή το `bless` επιστρέφει το πρώτο του όρισμα. Το `bless $self, $class` ως τελευταία έκφραση στη sub είναι ιδιωματικό. Εδώ το γράφουμε ρητά για ευκολία ανάγνωσης.

Accessor μέθοδοι με το χέρι

Το μοντέλο δεν σας δίνει accessors· τους γράφετε εσείς. Η συμπυκνωμένη μορφή εκμεταλλεύεται το γεγονός ότι το `$_[0]` είναι ο επικαλούμενος:

```
sub path    { $_[0]{path} }
sub content { $_[0]{content} }
```

Η μορφή με ονομασμένη παράμετρο είναι σαφέστερη και είναι αυτή που πρέπει να γράφετε όταν ο accessor κάνει οτιδήποτε μη τετριμμένο:

```
sub path {
    my $self = shift;
    return $self->{path};
}
```

Ένας accessor ανάγνωσης/εγγραφής:

```
sub name {
    my $self = shift;
    $self->{name} = shift if @_;
    return $self->{name};
}
```

Αυτό είναι το επαναλαμβανόμενο boilerplate που το *σύγχρονο χαρακτηριστικό class* αντικαθιστά με `:reader` και `:writer`.

Γιατί αναφορές hash

Οι αναφορές hash κυριαρχούν επειδή σας επιτρέπουν να προσθέτετε και να μετονομάζετε πεδία χωρίς να αλλάζετε τη διάταξη, και επειδή τα ονομασμένα κλειδιά επιβιώνουν ενός round-trip μέσω `Data::Dumper` ή `JSON::encode`. Οι αναφορές πινάκων είναι ταχύτερες και πιο συμπαγείς αλλά εύθραυστες: η αναρίθμηση μιας θυρίδας σπάει κάθε σημείο πρόσβασης.

Τα στιγμιότυπα με υπόβαθρο hash έχουν ένα σοβαρό μειονέκτημα: καμία ενθυλάκωση. Οποιοσδήποτε καλών μπορεί να φτάσει στα εσωτερικά του αντικειμένου με `$obj->{path}`. Το κλασικό μοντέλο δεν έχει κανένα εργαλείο να τους σταματήσει - η αυτοπειθαρχία και η ανασκόπηση κώδικα είναι οι μόνες άμυνες. Αυτός είναι ο μεγαλύτερος μεμονωμένος λόγος για να προτιμήσετε το *σύγχρονο χαρακτηριστικό class* για νέο κώδικα.

Κληρονομικότητα με @ISA

Η κληρονομικότητα είναι μια λίστα σε επίπεδο πακέτου με όνομα `@ISA`. Η αποστολή μεθόδων τη διασχίζει όταν το τρέχον πακέτο δεν ορίζει τη μέθοδο.

```
package File::MP3;
our @ISA = ('File');

sub print_info {
    my $self = shift;
    $self->SUPER::print_info;
    print "Title: $self->{title}\n";
}

1;
```

Το `SUPER::` σημαίνει «αναζήτησε αυτή τη μέθοδο ξεκινώντας ένα βήμα ψηλότερα στην αλυσίδα ISA από το πακέτο όπου *μεταγλωττίστηκε* η τρέχουσα μέθοδος»

- όχι από την κλάση του αντικειμένου. Αυτή η λεπτή διάκριση μετράει στη ρομβική κληρονομικότητα· το *κεφάλαιο για την κληρονομικότητα* την καλύπτει.

Δήλωση του γονέα - τέσσερις τρόποι

Υπάρχουν τέσσερις γραφές της ίδιας ιδέας. Προτιμήστε την τρίτη για νέο κώδικα που εξακολουθείτε να συντηρείτε με κλασικό ύφος.

```
# 1. Raw @ISA assignment.
package Child;
our @ISA = ('Parent');

# 2. use base.
package Child;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
use base 'Parent';         # deprecated in modern Perl

# 3. use parent.
package Child;
use parent 'Parent';       # current recommendation for classical

# 4. @ISA with a require.
package Child;
require Parent;
our @ISA = ('Parent');
```

Το `use parent` χειρίζεται τη φόρτωση του γονικού αρθρώματος και την προσθήκη στο `@ISA` σε μία γραμμή. Το `use base` είναι μια παλαιότερη παραλλαγή που εισάγει επιπλέον πεδία τύπου `Exporter`· αποφύγετέ το.

Ο επικαλούμενος `$self`

Κάθε μέθοδος είναι μια υπορουτίνα της οποίας το πρώτο όρισμα είναι είτε το όνομα της κλάσης (για μεθόδους κλάσης όπως η `new`) είτε το αντικείμενο (για μεθόδους στιγμιότυπου). Η σύμβαση είναι να γίνεται shift σε μεταβλητή με όνομα `$self`:

```
sub method {
    my $self = shift;
    ...
}
```

Ή να γίνεται αποπακετάρισμα μαζί με άλλα ορίσματα:

```
sub method {
    my ($self, $arg1, $arg2) = @_;
    ...
}
```

Δεν υπάρχει επιβολή κατά τον χρόνο εκτέλεσης. Αν ξεχάσετε να κάνετε shift το `$self`, το πρώτο όρισμα που δίνει ο χρήστης γίνεται σιωπηρά `$self` και επικρατεί χάος. Οι στατικοί αναλυτές το πιάνουν· η γλώσσα όχι.

Καταστροφείς

Μια μέθοδος `DESTROY`, αν ορίζεται, εκτελείται όταν ο μετρητής αναφορών του αντικειμένου πέσει στο μηδέν. Χρησιμοποιήστε την για να απελευθερώσετε εξωτερικούς πόρους:

```
sub DESTROY {
    my $self = shift;
    close $self->{fh} if defined $self->{fh};
}
```

Παγίδες που δαγκώνουν κάθε έργο τουλάχιστον μία φορά:

- Η DESTROY εκτελείται σε απρόβλεπτες χρονικές στιγμές. Μη βασίζεστε σε αυτήν για καθαρισμό κρίσιμο για την ορθότητα - μια ρητή `$obj->close` νικά την σιωπηρή καταστροφή.
- Η DESTROY μπορεί να δει ένα μερικώς κατασκευασμένο αντικείμενο αν η new πέθανε στη μέση. Οι αμυντικοί έλεγχοι `defined` είναι φτηνή ασφάλεια.
- Κατά την καθολική καταστροφή (χρόνος END), οι εξαρτήσεις μπορεί να έχουν ήδη χαθεί. Μια DESTROY που διασχίζει αναφορές μπορεί να αποτύχει με «Attempt to free unreferenced scalar» ή παρόμοιο. Τυλίξτε τον επικίνδυνο καθαρισμό σε `eval { ... }`.

AUTOLOAD

Αν μια κλήση μεθόδου δεν βρει αντίστοιχη sub στην κλάση ή στους προγόνους της, η Perl ψάχνει για μια sub AUTOLOAD. Το όνομα της καλούμενης μεθόδου εμφανίζεται στο our $AUTOLOAD:

```
our $AUTOLOAD;
sub AUTOLOAD {
    my $self = shift;
    my $name = $AUTOLOAD;
    $name =~ s/.*::/;
    return if $name eq 'DESTROY';       # critical
    ...
}
```

Ο φύλακας DESTROY είναι υποχρεωτικός: χωρίς αυτόν, η καταστροφή του αντικειμένου ενεργοποιεί την AUTOLOAD με `$name = 'DESTROY'` και είτε συμπεριφέρεστε λάθος είτε καταρρέετε.

Η AUTOLOAD είναι ισχυρή και εύκολο να καταχραστεί. Οι περιπτώσεις όπου αξίζει τη θέση της:

- Οκνηρή δημιουργία accessor - συνθέστε έναν accessor στην πρώτη κλήση και μετά εγκαταστήστε τον ως πραγματική sub μέσω ανάθεσης typeglob, ώστε η επόμενη κλήση να είναι άμεση.
- Προώθηση - αναθέστε άγνωστες μεθόδους σε ένα εμπεριεχόμενο αντικείμενο (αλλά δείτε *Ρόλοι και ανάθεση* για μια καθαρότερη προσέγγιση).

Περιπτώσεις όπου βλάπτει περισσότερο από όσο βοηθάει: να προσποιείται ότι είναι ένα πλήρες MOP, να πιάνει τυπογραφικά ως σιωπηρά no-op, να υλοποιεί «κλάσεις» των οποίων όλη η επιφάνεια είναι η AUTOLOAD.

Ένα επεξεργασμένο παράδειγμα

Μια κλασική κλάση `Counter` με accessor ανάγνωσης/εγγραφής και κληρονομικότητα:

```
package Counter;

sub new {
    my ($class, %args) = @_;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    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;
    $self->{value} += $self->{step};
}

1;
```

Μια υποκλάση που διπλασιάζει το βήμα:

```
package Counter::Double;
use parent 'Counter';

sub new {
    my ($class, %args) = @_;
    $args{step} //= 2;
    return $class->SUPER::new(%args);
}

1;
```

Σε σύγκριση με την έκδοση *σύγχρονης κλάσης*, η κλασική μορφή έχει περίπου τρεις φορές τις γραμμές, και καθεμιά τους είναι ένα σημείο όπου μπορείτε να κάνετε λάθος.

Πότε η κλασική αντικειμενοστρέφεια είναι η σωστή επιλογή

Παρ” όλα αυτά, υπάρχουν θεμιτοί λόγοι να μείνετε με κλάσεις βασισμένες σε `bless`:

- **Συντηρείτε υπάρχοντα κώδικα.** Η επανεγγραφή μιας κλασικής κλάσης που λειτουργεί σε μορφή `class` είναι έργο μετάβασης, όχι δωρεάν γεύμα - δείτε το *κεφάλαιο μετάβασης*.
- **Πρέπει να υποστηρίξετε παλαιότερη Perl.** Αν ο κώδικάς σας πρέπει να τρέχει σε Perl πριν την 5.38, το χαρακτηριστικό `class` δεν είναι διαθέσιμο.
- **Χρειάζεστε αντικείμενα με υπόβαθρο πίνακα ή κώδικα.** Το `class` δημιουργεί μόνο στιγμιότυπα με υπόβαθρο hash. Αν χτίζετε μια κρίσιμη για την απόδοση δομή με στιγμιότυπα αναφοράς πίνακα, το κλασικό `bless` παραμένει η απάντηση.

- **Κάνετε κάτι που το χαρακτηριστικό `class` σκόπιμα απαγορεύει** - ξανανοίγετε το πακέτο από έξω, πολλαπλή κληρονομικότητα ή βαθύ μεταπρογραμματισμό μέσω χειρισμού του `@ISA`.

Μια ιστορική σημείωση για τα συστήματα αντικειμενοστρέφειας του CPAN

Επειδή το κλασικό μοντέλο αφήνει τόσα πολλά στη σύμβαση, το CPAN ανέπτυξε ένα οικοσύστημα αρθρωμάτων που προσθέτουν δηλωτική σύνταξη από πάνω:

- **Moose** - πλήρες σύστημα αντικειμενοστρέφειας με μινιατούρα τύπων, ρόλους, method modifiers, πλήρες API ενδοσκόπησης και ένα μεγάλο οικοσύστημα επεκτάσεων. Βαρύ κατά τη φόρτωση· η κυρίαρχη επιλογή για μεγάλες εφαρμογές που χτίστηκαν μεταξύ 2007 και περίπου 2020.
- **Moo** - υποσύνολο του API του Moose χωρίς το στρώμα ενδοσκόπησης. Ελαφρύτερο, pure-Perl, αρκετά συμβατό σε επίπεδο API ώστε μια κλάση Moo και μια κλάση Moose να μοιράζονται ένα δέντρο κληρονομικότητας.
- **Class::Accessor** - δημιουργεί απλούς accessors και τη `new`. Μινιμαλιστικό, pure-Perl, χωρίς ρόλους.
- **Class::Tiny** - το μικρότερο από τα αρθρώματα δημιουργίας accessor. Όλοι οι accessors είναι ανάγνωσης/εγγραφής, χωρίς ελέγχους τύπου.
- **Object::Pad** - το πειραματικό πρωτότυπο που έγινε το χαρακτηριστικό `class` του πυρήνα. Παραμένει εγκαταστάσιμο από το CPAN για κώδικα που χρειάζεται το χαρακτηριστικό σε παλαιότερες Perl.
- **Role::Tiny** - σύνθεση ρόλων για έργα που δεν χρησιμοποιούν Moose αλλά θέλουν παρ” όλα αυτά κοινή χρήση τύπου `does`.

Αυτά τα συστήματα αποτελούν **ιστορικό πλαίσιο** στην pperl. Ο νέος κώδικας στην pperl θα πρέπει να χρησιμοποιεί το χαρακτηριστικό `class` του πυρήνα, που καλύπτει το έδαφος που πρωτοπόρησαν τα Moose/Moo χωρίς το κόστος φόρτωσης και εξαρτήσεων. Όταν κληρονομείτε μια βάση κώδικα που χρησιμοποιεί κάποιο από αυτά, μάθετε όσα μόλις χρειάζονται από την επιφάνειά του ώστε να παραμείνετε παραγωγικοί και προγραμματίστε τη μετάβαση με το *κεφάλαιο μετάβασης*.

Κληρονομικότητα και επίλυση μεθόδων

Η αναζήτηση μεθόδων στην Perl είναι απλή στην περίπτωση απλής κληρονομικότητας και γνήσια λεπτή μόλις μπουν στο παιχνίδι ρόμβοι ή μεικτά συστήματα κλάσεων. Αυτό το κεφάλαιο καλύπτει την πλήρη ιστορία: τους αλγορίθμους MRO, το `SUPER::`, τον τελεστή `isa`, το `can`, το `DOES` και την αλληλεπίδραση μεταξύ κλασικών και σύγχρονων κλάσεων σε μία ιεραρχία.

Ο αλγόριθμος αναζήτησης

Όταν γράφετε `$obj->foo(@args)`, η Perl κάνει τα εξής:

1. Προσδιορίζει την κλάση. Για ένα αντικείμενο, είναι η κλάση στην οποία έγινε bless ο αναφερόμενος· για κλήση μεθόδου κλάσης, είναι ο ίδιος ο επικαλούμενος.
2. Διασχίζει τη **σειρά επίλυσης μεθόδων** (MRO) της κλάσης κατά σειρά. Για κάθε πακέτο στη σειρά, αναζητά μια sub με το όνομα `foo`.

3. Η πρώτη αντιστοιχία κερδίζει. Αν δεν βρεθεί αντιστοιχία σε καμία κλάση του MRO, ξαναδιασχίζει αναζητώντας AUTOLOAD.
4. Αν δεν βρεθεί ούτε AUTOLOAD, εκπέμπει `Can't locate object method "foo" via package "Class"`.

Το MRO είναι αυτό στο οποίο γραμμικοποιείται η αλυσίδα `@ISA` της κλάσης. Ο κανόνας γραμμικοποίησης εξαρτάται από το ποιο MRO είναι σε ισχύ.

Τα δύο MRO: `dfs` και `c3`

Η Perl παρέχει δύο αλγορίθμους MRO:

- **`dfs`** - αναζήτηση κατά βάθος του `@ISA`. Η προεπιλογή. Διασχίζει κάθε γονέα και τους γονείς κάθε γονέα, από αριστερά προς τα δεξιά, βυθιζόμενη πλήρως σε κάθε υποδέντρο πριν προχωρήσει.
- **`c3`** - η γραμμικοποίηση C3 που χρησιμοποιούν η Python, η Raku και πολλές άλλες. Εγγυάται μονοτονικότητα: μια υποκλάση δεν εμφανίζεται ποτέ αργότερα από κάποιον πρόγονό της στη σειρά αναζήτησης.

Στην απλή κληρονομικότητα οι δύο είναι ταυτόσημες: η λίστα ανεβαίνει ευθεία στην αλυσίδα.

Στην πολλαπλή κληρονομικότητα αποκλίνουν. Σκεφτείτε τον ρόμβο:

```
    A
   / \
  B   C
   \ /
    D
```

- **`dfs`**: `D, B, A, C, A` (συμπτυσσόμενη σε `D, B, A, C`). Μια μέθοδος στο C που υπερκαλύπτει το A **επικαλύπτεται** από το A επειδή το A επιτυγχάνεται πρώτα μέσω του B.
- **`c3`**: `D, B, C, A`. Η υπερκάλυψη του C είναι ορατή.

Το C3 είναι σχεδόν πάντα αυτό που θέλετε υπό πολλαπλή κληρονομικότητα. Επιλέξτε το ανά κλάση:

```
use mro 'c3';
package Diamond::D;
our @ISA = ('Diamond::B', 'Diamond::C');
```

Για κλάσεις δηλωμένες με `class`, η πολλαπλή κληρονομικότητα δεν υποστηρίζεται καθόλου, οπότε η επιλογή MRO είναι αμιγώς μια γραμμική διάσχιση και η προεπιλογή `dfs` είναι μια χαρά.

`SUPER::`

Το `$self->SUPER::foo(@args)` καλεί τη `foo` σε μια από τις γονικές κλάσεις του *πακέτου όπου μεταγλωττίστηκε η τρέχουσα μέθοδος*. Αυτή η τελευταία φράση είναι αυτή που όλοι λανθάνουν.

```
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;
}
```

Η Puppy::speak καλεί την SUPER::speak. Η Perl αναζητά από το **Puppy** (το πακέτο μεταγλώττισης), όχι από το ref($self). Η αναζήτηση ξεκινά στο @ISA του Puppy, οπότε βρίσκει την Dog::speak και επιστρέφει 'woof'.

Αυτό έχει σημασία όταν οι μέθοδοι κληρονομούνται και το παιδί καλεί από μέσα το SUPER::. Επειδή το SUPER:: αγκυρώνεται στο πακέτο μεταγλώττισης, μια μέθοδος κληρονομημένη από το Dog και καλούμενη σε αντικείμενο Puppy θα εξακολουθήσει να καλεί το Animal:: μέσω του δικού της SUPER::, όχι το Dog. Συνήθως αυτή είναι η σωστή συμπεριφορά· όταν δεν είναι, στραφείτε στη ρητή πλοήγηση της αλυσίδας με next::method, που καλύπτεται παρακάτω.

next::method - το SUPER με επίγνωση του MRO

Η mro::next::method (εκτεθειμένη ως $self->next::method(@args)) διασχίζει το ζωντανό MRO από την *πραγματική* θέση κλήσης της τρέχουσας μεθόδου, όχι από το πακέτο μεταγλώττισης:

```
use mro 'c3';

sub speak {
    my $self = shift;
    return $self->next::method;
}
```

Υπό C3 με ρόμβο, η next::method είναι αυτό που θέλετε· το SUPER:: μπορεί να παρακάμψει έναν αδελφικό κλάδο επειδή συμβουλεύεται μόνο τον αριστερότερο πρόγονο.

Ο τελεστής isa

Ο τελεστής isa ελέγχει την ιδιότητα μέλους κλάσης χωρίς την παγίδα της κλήσης UNIVERSAL::isa ως συνάρτησης:

```
if ($obj isa 'File::MP3') { ... }
if ($obj isa File::MP3)   { ... }     # bareword form
```

Τόσο οι κλάσεις βασισμένες σε bless όσο και εκείνες που δηλώνονται με class

απαντούν σωστά στο `isa`, και οι σχέσεις υποκλάσεων ρέουν μέσω των `:isa` και `@ISA` πανομοιότυπα.

Τρεις κανόνες που αξίζει να θυμάστε:

- Το `isa` επιστρέφει ψευδές (όχι σφάλμα) όταν του δοθεί μη αντικείμενο. Το `42 isa Foo` είναι ψευδές. Το `undef isa Foo` είναι ψευδές.
- Το `isa` σέβεται το ζωντανό MRO τη στιγμή της κλήσης. Αν κώδικας προσθέσει στοιχεία στο `@ISA` κατά τον χρόνο εκτέλεσης, το `isa` τα βλέπει. (Αυτός είναι άλλος ένας λόγος που η pperl απορρίπτει τον χειρισμό του `@ISA` για κλάσεις δηλωμένες με `class` - οι εγγυήσεις θα γίνονταν θολές.)
- Προτιμήστε τον τελεστή `isa` έναντι του `UNIVERSAL::isa($obj, 'Class')`. Ο τελεστής βραχυκυκλώνει σε μη αντικείμενα αντί να πεθαίνει· η μορφή κλήσης συνάρτησης ηττάται επίσης από overloaded αντικείμενα.

can

Το `$obj->can('method')` επιστρέφει μια αναφορά κώδικα αν το αντικείμενο έχει μέθοδο με αυτό το όνομα, διαφορετικά `undef`:

```
if (my $code = $obj->can('save')) {
    $code->($obj, @args);
}
```

Χρήσεις:

- Ανίχνευση χαρακτηριστικού πριν την κλήση: αποφεύγει το σφάλμα `Can't locate object method`.
- Αποστολή σε μέθοδο με βάση το όνομα από συμβολοσειρά αποθηκευμένη σε μεταβλητή - πιο ευανάγνωστο από τη μορφή `$obj->$name` όταν η μέθοδος ενδέχεται να μην υπάρχει.

Μη-χρήσεις:

- **Μη** χρησιμοποιείτε το `can` για να αποφασίσετε αν ένα αντικείμενο υλοποιεί έναν εννοιολογικό ρόλο. Μια κλάση μπορεί να ορίζει μια μέθοδο `save` που κάνει κάτι εντελώς άσχετο με την μονιμοποίηση. Για ιδιότητα μέλους ρόλου, δείτε *ρόλοι και ανάθεση*.

DOES

Το `$obj->DOES('RoleOrClass')` είναι το αντίστοιχο με επίγνωση ρόλων του `isa`. Από προεπιλογή το `DOES` αναθέτει στο `isa`, οπότε για κλάσεις χωρίς ρητή μηχανική ρόλων τα δύο συμπεριφέρονται το ίδιο. Τα συστήματα ρόλων (κλασικά Moose, Moo και το μελλοντικό χαρακτηριστικό ρόλων του πυρήνα) υπερκαλύπτουν το `DOES` ώστε να επιστρέφει αληθές και για ρόλους που καταναλώνει η κλάση.

```
if ($obj->DOES('Printable')) { ... }
```

Στην pperl, το `DOES` είναι η σωστή μέθοδος ελέγχου όταν θέλετε να ρωτήσετε «υλοποιεί αυτό το αντικείμενο το συμβόλαιο;» αντί για «κληρονομεί αυτό το αντικείμενο από αυτή την κλάση;».

Μεικτές κλασικές / σύγχρονες ιεραρχίες

Οι κλάσεις δηλωμένες με `class` και εκείνες βασισμένες σε `bless` διαλειτουργούν και προς τις δύο κατευθύνσεις, υπό μια χούφτα κανόνες.

Μια σύγχρονη κλάση που κληρονομεί από κλασική κλάση

```
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 }
}
```

Αυτό λειτουργεί. Ο δημιουργημένος κατασκευαστής της σύγχρονης κλάσης αναθέτει στη new του γονέα, και οι μέθοδοι που κληρονομούνται από τον κλασικό γονέα εμφανίζονται στο σύγχρονο παιδί. Η κατάσταση των hash-slot του γονέα ζει *δίπλα* στην αποθήκευση πεδίων του παιδιού.

Επιφύλαξη: το σύγχρονο παιδί δεν μπορεί να φτάσει στο hash του γονέα. Το `$self->{colour}` δεν είναι θεμιτό μέσα σε μέθοδο της `Circle`· πρέπει να καλέσετε `$self->colour`.

Μια κλασική κλάση που κληρονομεί από σύγχρονη κλάση

```
use feature 'class';
class Animal {
    field $name :param :reader;
    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' }
```

Και αυτό λειτουργεί - ο δημιουργημένος κατασκευαστής του σύγχρονου γονέα κληρονομείται, και το `SUPER::new` επιλύεται σε αυτόν. Το κλασικό παιδί δεν πρέπει να επιχειρήσει να τροποποιήσει απευθείας τα πεδία του γονέα· τα πεδία είναι ιδιωτικά στην κλάση που τα δήλωσε.

Τι αποτυγχάνει σε μεικτές ιεραρχίες

- **Πρόσβαση `$self->{fieldname}` πέρα από το σύνορο.** Τα σύγχρονα πεδία δεν είναι hash-slot, και οι κλασικές hash-slot δεν είναι ορατές ως πεδία. Διαπερνάτε πάντα το σύνορο μέσω μεθόδου.
- **Πολλαπλή κληρονομικότητα σε σύγχρονη κλάση.** Το attribute `:isa` παίρνει μόνο έναν στόχο. Αν χρειάζεστε να αναμίξετε συμπεριφορά από πολλαπλά σημεία, δείτε *ρόλοι και ανάθεση*.
- **Χειρισμός του `@ISA` κατά τον χρόνο εκτέλεσης σε σύγχρονη κλάση.** Απροσδιόριστη συμπεριφορά. Αντιμετωπίστε το `@ISA` ως μόνο για ανάγνωση για κάθε κλάση δηλωμένη με `class`.

Γιατί η πολλαπλή κληρονομικότητα συνήθως βλάπτει

Κάθε γλώσσα με πολλαπλή κληρονομικότητα τελικά ανακαλύπτει τις ίδιες παθολογίες. Η αναζήτηση μεθόδων γίνεται εξαρτημένη από τη σειρά· τα προβλήματα του ρόμβου σας αναγκάζουν να σκεφτείτε γραμμικοποίηση· η περίπτωση «κληρονόμησα δύο φορές από την ίδια ρίζα» μπερδεύει την αρχικοποίηση. Οι ρόλοι - σύνθεση συμπεριφοράς χωρίς σχέση κληρονομικότητας - λύνουν το υποκείμενο πρόβλημα κοινής χρήσης χωρίς τις παθολογίες.

Στην pperl το χαρακτηριστικό `class` απαγορεύει σκόπιμα την πολλαπλή κληρονομικότητα. Κλασικός κώδικας που τη χρησιμοποιεί θα έπρεπε να μεταβαίνει προς ένα μοντέλο ρόλων· το *κεφάλαιο για τους ρόλους* είναι η επόμενη στάση.

Περαιτέρω ανάγνωση

- `class` - πλήρης σύνταξη των δηλώσεων σύγχρονης κλάσης, συμπεριλαμβανομένου του `:isa`.
- `bless` - η πρωτογενής οντότητα κάτω από την κλασική κληρονομικότητα.
- `ref` - αναφέρει το όνομα της κλάσης μιας blessed αναφοράς.
- `isa` - ο προτιμώμενος τελεστής ιδιότητας μέλους κλάσης.
- *Ρόλοι και ανάθεση* - η εναλλακτική που βασίζεται στη σύνθεση έναντι της πολλαπλής κληρονομικότητας.
- *Μετάβαση* - μετακίνηση μιας υπάρχουσας ιεραρχίας από κλασική σε σύγχρονη.

Ρόλοι και ανάθεση

Η κληρονομικότητα απαντά στο «τι είναι αυτό το αντικείμενο;». Οι ρόλοι και η ανάθεση απαντούν σε άλλο ερώτημα: «τι μπορεί να κάνει αυτό το αντικείμενο;». Όταν δύο άσχετες κλάσεις χρειάζεται να μοιραστούν συμπεριφορά, η εξαναγκασμένη υπαγωγή τους σε κοινό γονέα είναι το λάθος εργαλείο. Η σύνθεση - να δίνετε σε κάθε κλάση τη συμπεριφορά που χρειάζεται χωρίς να προσποιείστε ότι σχετίζονται - είναι το σωστό.

Αυτό το κεφάλαιο καλύπτει τα τρία πρότυπα σύνθεσης που χρησιμοποιεί στην πράξη ο κώδικας της pperl:

1. **Κοινή χρήση τύπου mixin** μέσω ενός μικροσκοπικού βοηθητικού πακέτου.

2. **Σύνθεση ρόλων** μέσω `Role::Tiny` ή ενός συστήματος της οικογένειας Moose σε παλαιό κώδικα.
3. **Ανάθεση** - προώθηση μεθόδων σε ένα εμπεριεχόμενο αντικείμενο.

Το χαρακτηριστικό `class` του πυρήνα δεν διαθέτει ακόμη πρωτογενή οντότητα ρόλου. Πρόκειται για σκόπιμη απόφαση σταδιοποίησης από το upstream: το χαρακτηριστικό `class` έφτασε πρώτο, οι ρόλοι είναι ξεχωριστή πρόταση. Μέχρι να φτάσει το χαρακτηριστικό ρόλων του πυρήνα, τα πρότυπα αυτού του κεφαλαίου είναι οι λειτουργικές απαντήσεις.

Το πρόβλημα

Φανταστείτε μια κλάση `Radio` και μια κλάση `Computer`. Και οι δύο έχουν διακόπτη on/off. Δεν σχετίζονται - ένα ραδιόφωνο δεν είναι υπολογιστής, και κανένα από τα δύο δεν εξειδικεύει μια υποθετική `Machine`. Μια γονική κλάση `HasOnOffSwitch` θα ήταν τεχνητή ομαδοποίηση που μοναδικός σκοπός της είναι η επαναχρησιμοποίηση δύο μεθόδων.

Τρεις λύσεις, κατά αύξουσα σειρά φιλοδοξίας:

- Αντιγράψτε τις μεθόδους `turn_on` / `turn_off` σε κάθε κλάση. Ανεκτό για δύο κλάσεις, οδυνηρό για δέκα.
- Εξάγετέ τις σε ένα βοηθητικό πακέτο, χρησιμοποιήστε έναν από τους μηχανισμούς κοινής χρήσης παρακάτω.
- Χρησιμοποιήστε ένα σύστημα ρόλων με τυπική σημασιολογία σύνθεσης.

Πρότυπο 1: χειροκίνητο mixin

Ο μικρότερος δυνατός μηχανισμός «μοιράσου αυτές τις μεθόδους» είναι ένα απλό πακέτο που εγκαθιστά υπορουτίνες στην κλάση-καταναλωτή κατά τον χρόνο `use`:

```
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;
```

Σε χρήση:

```
package Radio;
use HasOnOffSwitch;

sub new { bless { is_on => 0 }, shift }
```

Αυτό λειτουργεί. Είναι επίσης εύθραυστο: φτάνει απευθείας στο hash της κλάσης-καταναλωτή (`$_[0]->{is_on}`), που είναι ακριβώς η παραβίαση ενθυλάκωσης

για την οποία φημίζεται η κλασική αντικειμενοστρέφεια με υπόβαθρο hash. Σπάει αν ο καταναλωτής είναι κλάση δηλωμένη με `class`, επειδή τα πεδία δεν είναι hash-slot.

Χρησιμοποιήστε αυτό το πρότυπο μόνο για πολύ μικρή κοινόχρηστη συμπεριφορά σε κλασική βάση κώδικα. Για οτιδήποτε μεγαλύτερο, χρησιμοποιήστε ένα πραγματικό σύστημα ρόλων.

Πρότυπο 2: `Role::Tiny`

Το `Role::Tiny` είναι το ελαφρύ σύστημα ρόλων που λειτουργεί με απλές κλασικές κλάσεις, με `Moo` και με `Moose` (στην πλευρά του καταναλωτή). Είναι ό,τι πιο κοντινό σε φορητή πρωτογενή οντότητα ρόλου στο οικοσύστημα του CPAN.

```
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;
```

Ο καταναλωτής υιοθετεί τον ρόλο με `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} }
```

Τι σας δίνει το `Role::Tiny` σε σχέση με το χειροκίνητο mixin:

- **Απαιτήσεις.** Το `requires 'is_on'` αποτυγχάνει κατά τον χρόνο σύνθεσης αν ο καταναλωτής δεν παρέχει `is_on`. Το χειροκίνητο mixin αποτυγχάνει σιωπηρά στην πρώτη κλήση μεθόδου.
- **Ανίχνευση συγκρούσεων.** Η κατανάλωση δύο ρόλων που ορίζουν αμφότεροι την ίδια μέθοδο είναι σφάλμα κατά τον χρόνο σύνθεσης, όχι σιωπηρή κυριαρχία του τελευταίου ορισμού.
- **Method modifiers** - `before`, `after`, `around` - αν τους θέλετε.

Το `Role::Tiny` **δεν** είναι το ίδιο πράγμα με το σύστημα ρόλων του Moose. Η εμβέλειά του είναι σκόπιμα στενή. Αν η βάση κώδικα χρησιμοποιεί ήδη Moose, χρησιμοποιήστε `Moose::Role`· αν χρησιμοποιεί Moo, χρησιμοποιήστε `Moo::Role`.

Πρότυπο 3: ρόλοι οικογένειας Moose

Οι παλαιές βάσεις κώδικα χτισμένες σε Moose ή Moo χρησιμοποιούν `Moose::Role` και `Moo::Role` αντίστοιχα. Η επιφάνεια μοιάζει με το σύστημα ρόλων που θα αποκτήσει τελικά το χαρακτηριστικό `class` του πυρήνα:

```
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;
```

Στην pperl, το Moose εξακολουθεί να λειτουργεί. Ο νέος κώδικας δεν θα έπρεπε να στρέφεται σε αυτό - το κόστος στον χρόνο εκτέλεσης και το βάρος των εξαρτήσεων είναι αμφότερα μεγάλα, και το χαρακτηριστικό `class` του πυρήνα καλύπτει πλέον τη δηλωτική πλευρά για την οποία χτίστηκε το Moose. Όταν φτάσει το χαρακτηριστικό ρόλων του πυρήνα, η μετάβαση από `Moose::Role` στον ρόλο του πυρήνα θα είναι το φυσικό επόμενο βήμα.

Πρότυπο 4: ανάθεση

Η ανάθεση λέει «δεν το κάνω εγώ ο ίδιος, αλλά κρατάω κάτι που το κάνει». Αντί να κληρονομείτε συμπεριφορά, περιέχετε ένα αντικείμενο που την παρέχει, και προωθείτε μεθόδους σε αυτό.

Η χειροκίνητη μορφή:

```
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 }
}
```

Κάθε προωθούμενη μέθοδος είναι τρεις γραμμές μηχανικής σωλήνωσης. Για δύο ή τρεις μεθόδους αυτό είναι εντάξει· για ευρεία επιφάνεια, στραφείτε στο `Class::Method::Delegate` ή σε accessor `handles` του Moose:

```
package Player;
use Moose;

has radio => (
    is      => 'ro',
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    handles => ['turn_on', 'turn_off', 'is_on'],
);
```

Το handles δημιουργεί τους προωθητές κατά τον χρόνο κατασκευής της κλάσης.

Πότε να αναθέτετε έναντι να κληρονομείτε

- **Αναθέστε** όταν το εμπεριεχόμενο αντικείμενο είναι *συνεργάτης* - κάτι που ο κάτοχος *έχει* αντί να *είναι*. Ένας Player έχει ένα Radio· ένας Player δεν είναι είδος Radio.
- **Κληρονομήστε** όταν το παιδί γνήσια *είναι* ένας εξειδικευμένος γονέας. Ένας Circle είναι είδος Shape· ένα File::MP3 είναι είδος File.

Αν η απάντηση στο «is-a ή has-a;» δεν είναι άμεση, η σχέση είναι πιθανότατα has-a. Η προεπιλογή της σύνθεσης είναι σχεδόν πάντα η λιγότερο λανθασμένη επιλογή - μπορείτε να αναδιαμορφώσετε τη σύνθεση σε κληρονομικότητα με λιγότερο πόνο από το αντίστροφο.

Ανάθεση με οκνηρή κατασκευή

Ένα συχνό πρότυπο: ο συνεργάτης κοστίζει να χτιστεί, οπότε χτίστε τον στην πρώτη πρόσβαση. Με το σύγχρονο χαρακτηριστικό class:

```
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";
    }
}
```

Το fh είναι reader που χειρίζεται και την εφάπαξ κατασκευή. Οι καλούντες βλέπουν μια απλή μέθοδο· η οκνηρή cache είναι ιδιωτική.

Τι να μην κάνετε

- **Πολλαπλή κληρονομικότητα για να μοιραστείτε δύο μεθόδους.** Πληρώνετε το πλήρες κόστος του MRO για μια ασήμαντη νίκη, και ο επόμενος που θα κληρονομήσει από την κλάση σας κληρονομεί επίσης τον πόνο του MRO. Χρησιμοποιήστε έναν ρόλο ή ένα mixin.

- **Η AUTOLOAD ως αναθέτης.** Βολική στο πεντάλεπτο demo, αδύνατο να ενδοσκοπηθεί, και τρώει τυπογραφικά. Γράψτε ρητούς προωθητές ή χρησιμοποιήστε `handles`.
- **Ρόλοι που φέρουν κατάσταση.** Ένας ρόλος που ορίζει τόσο μεθόδους όσο και attributes είναι πραγματικά μια μικρή κλάση. Ο καθαρότερος ρόλος ορίζει συμπεριφορά που υποστηρίζει η κατάσταση της κλάσης-καταναλωτή· ο ρόλος απαιτεί τους accessors που χρειάζεται και εμπιστεύεται τον καταναλωτή ότι θα τους παράσχει. Το παράδειγμα χειροκίνητου mixin στο πρότυπο 1 το έκανε λάθος φτάνοντας απευθείας στο `$_[0]->{is_on}`.
- **Βαθιές αλυσίδες ανάθεσης.** Αν το `$player->radio->volume->level` εμφανίζεται επανειλημμένα, προσθέστε έναν accessor `volume_level` στον `Player`. Οι αλυσίδες μεθόδων τύπου εκτροχιασμένου τρένου είναι μυρωδιά κακής σύνθεσης.

Περαιτέρω ανάγνωση

- *Σύγχρονες κλάσεις* - το βασικό στρώμα πάνω στο οποίο κάθεται η σύνθεση.
- *Κληρονομικότητα και επίλυση μεθόδων* - όταν η απάντηση πραγματικά *είναι* η κληρονομικότητα.
- *Μετάβαση* - πώς ο παλαιός κώδικας βασισμένος σε ρόλους μεταφράζεται στο μοντέλο `class` του πυρήνα (spoiler: στο μεγαλύτερο μέρος του αμετάβλητος· οι ρόλοι μεταφέρονται τελευταίοι).
- `class` - αναφορά για τη σύγχρονη σύνταξη κλάσεων.
- `isa` - έλεγχος ιδιότητας μέλους κλάσης· σε ζεύγος με το `DOES` για ιδιότητα μέλους ρόλου.

Μετάβαση από κλασική σε σύγχρονη

Αν συντηρείτε μια βάση κώδικα βασισμένη σε `bless` και αναρωτιέστε πώς θα έμοιαζε το ισοδύναμο μπλοκ `class`, αυτό το κεφάλαιο είναι η συνταγή. Η μετάφραση είναι ως επί το πλείστον μηχανική, αλλά τα σημεία όπου δεν είναι μηχανική είναι εκεί όπου αναδύονται πραγματικές αποφάσεις σχεδίασης - και όπου η μετάβαση αποπληρώνεται μόνη της.

Πριν ξεκινήσετε, διαβάστε το *κεφάλαιο της σύγχρονης κλάσης* για τη σύνταξη *προς την οποία* μεταφράζετε, και διατρέξτε το *κεφάλαιο της κλασικής αντικειμενοστρέφειας* για όποιες κατασκευές χρησιμοποιεί ο υπάρχων κώδικας και ίσως δεν θυμάστε με λεπτομέρεια.

Πότε να μεταβείτε

Η μετάβαση δεν είναι δωρεάν. Κάντε την όταν αποκομίζετε πραγματική αξία:

- Η κλάση βρίσκεται υπό **ενεργή ανάπτυξη**. Πρόκειται να προσθέσετε πεδία, να προσθέσετε μεθόδους ή να αλλάξετε το συμβόλαιο του κατασκευαστή. Το να γίνεται αυτή η δουλειά πάνω στο χαρακτηριστικό `class` είναι λιγότερο επιρρεπές σε σφάλματα από το να γίνεται πάνω σε χειροποίητες hash-slot.
- Η κλάση έχει **ασαφή ενθυλάκωση**. Οι καλούντες φτάνουν απευθείας στο `$obj->{fieldname}`, και θέλετε να το σταματήσετε χωρίς να ξαναγράψετε κάθε

σημείο κλήσης ταυτόχρονα. Το χαρακτηριστικό `class` κάνει την παραβίαση σφάλμα μεταγλώττισης τη στιγμή που μεταβαίνετε.

- Η κλάση κάθεται πάνω σε **Moose ή Moo**, και πληρώνετε το κόστος φόρτωσης χωρίς να χρησιμοποιείτε τα χαρακτηριστικά με βαριές εξαρτήσεις. Το σύγχρονο `class` αποτελεί άμεση αντικατάσταση για τις περισσότερες κλάσεις Moose/Moo που χρησιμοποιούν μόνο `has` και `with`.

Μη μεταφέρετε μια κλάση που λειτουργεί και είναι παγωμένη. Μια σταθερή κλασική κλάση που δεν πρόκειται να δει νέο κώδικα για άλλα πέντε χρόνια είναι ακριβώς το ίδιο καλή με το σύγχρονο αντίστοιχό της, και η ίδια η μετάβαση θα εισαγάγει σφάλματα που η σταθερή κλάση δεν έχει.

Η μηχανική μετάφραση

Πάρτε αυτή την κλασική κλάση:

```
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;
```

Η σύγχρονη μορφή:

```
use v5.38;
use feature 'class';
no warnings 'experimental::class';

class Counter {
    field $value :param :reader :writer = 0;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    field $step  :param :reader          = 1;

    ADJUST {
        die "step must be positive" if $step <= 0;
    }

    method tick { $value += $step }
}

1;
```

Η αντιστοίχιση, κομμάτι κομμάτι:

Κλασική κατασκευή	Σύγχρονη κατασκευή
`package Counter;`	`class Counter { ... }`
`sub new { ... bless ... }`	*(διαγραμμένο - δημιουργείται αυτόματα)*
Προεπιλογή ορίσματος κατασκευαστή με `//`	`= DEFAULT` ή `:param ... = DEFAULT`
`$args{foo}` αποθηκευμένο στο `$self->{foo}`	`field $foo :param`
Accessor μόνο για ανάγνωση	`:reader`
Accessor ανάγνωσης/εγγραφής	`:reader :writer`
Επικύρωση στο τέλος της new	Μπλοκ ADJUST
`$self->{foo}` μέσα σε μέθοδο	`$foo` (το όνομα του πεδίου)
`my $self = shift`	*(διαγραμμένο - σιωπηρό)*
`use parent 'Base';` / `our @ISA = ('Base')`	`class Sub :isa(Base) { ... }`
`$self->SUPER::foo(@args)`	αμετάβλητο - εξακολουθεί να λειτουργεί

Τα μη μηχανικά μέρη

Όλα τα παραπάνω είναι μετάφραση από μνήμης. Τα σημεία όπου χρειάζεται να σκεφτείτε είναι όλα παραλλαγές του «ο κλασικός κώδικας έκανε κάτι που το χαρακτηριστικό `class` δεν κάνει με τον ίδιο τρόπο».

Πολλαπλή κληρονομικότητα

Το `class` δεν υποστηρίζει `:isa(A, B)`. Αν η κλασική κλάση έχει `@ISA = ('Parent1', 'Parent2')`, έχετε τρεις επιλογές:

1. **Συμπτύξτε σε απλή κληρονομικότητα.** Συνήθως ο δεύτερος γονέας είναι ένα mixin ή ρόλος μεταμφιεσμένος. Μεταφέρετέ τον σε έναν *ρόλο* και καταναλώστε τον με `with`.
2. **Συμπτύξτε σε σύνθεση.** Αν ο δεύτερος γονέας είναι συνεργάτης παρά πρόγονος, μεταβείτε σε ανάθεση αντί για κληρονομικότητα.

3. **Μη μεταφέρετε την κλάση.** Αν η πολλαπλή κληρονομικότητα είναι γνήσια κρίσιμη και καμία από τις παραπάνω δεν ταιριάζει, η κλάση δεν ταιριάζει στο σύγχρονο χαρακτηριστικό. Αφήστε τη κλασική μέχρι να φτάσει το χαρακτηριστικό ρόλων του πυρήνα.

Στην πράξη η επιλογή 1 ή 2 ισχύει για περίπου το 95% των περιπτώσεων πολλαπλής κληρονομικότητας σε πραγματικό κώδικα.

Άμεση πρόσβαση hash από τους καλούντες

Αν εξωτερικοί καλούντες κάνουν `$counter->{value}` αντί για `$counter->value`, η μετάβαση της κλάσης θα τους σπάσει. Το σφάλμα του μεταγλωττιστή είναι καθαρό και τοπικό - το μήνυμα σφάλματος κατονομάζει το σημείο - αλλά η μετάβαση δεν είναι πλέον απευθείας αντικαταστάτης.

Στρατηγική άμβλυνσης:

1. **Έλεγχος πρώτα.** Κάντε grep για πρότυπα `$obj_name->{` σε όλη τη βάση κώδικα. Ο αριθμός είναι συνήθως μικρός.
2. **Προσθέστε έναν accessor** στην κλασική κλάση που να καλύπτει κάθε πεδίο στο οποίο γίνεται πρόσβαση.
3. **Αντικαταστήστε** τα σημεία κλήσης `$obj->{fieldname}` με `$obj->fieldname()` σε ένα αφιερωμένο commit.
4. **Μετά μεταφέρετε** την κλάση στη σύγχρονη μορφή.

Ο διαχωρισμός της αναδιαμόρφωσης σε δύο φάσεις κρατάει κάθε commit μικρό και bisectable.

AUTOLOAD

Η `AUTOLOAD` σε κλασική κλάση δεν έχει σύγχρονο ισοδύναμο. Το χαρακτηριστικό `class` δεν σας επιτρέπει να πιάνετε κλήσεις απουσιαζουσών μεθόδων.

Αν η `AUTOLOAD` κάνει ένα από τα κοινά πρότυπα, μεταφράστε το:

- **Οκνηρή σύνθεση accessor** - δηλώστε τους accessors ρητά ως πεδία με `:reader` ή `:writer`. Η δημιουργία κώδικα του χαρακτηριστικού `class` αντικαθιστά ό,τι έκανε η `AUTOLOAD`.
- **Ανάθεση σε εμπεριεχόμενο αντικείμενο** - αντικαταστήστε με ρητές μεθόδους προώθησης (δείτε *ρόλοι και ανάθεση*) ή με έναν accessor `handles` τύπου Moose αν η βάση κώδικα μπορεί να εξαρτηθεί από το Moose.
- **Ένα γενικό proxy** - κρατήστε την κλάση κλασική. Το χαρακτηριστικό `class` δεν είναι το σωστό εργαλείο.

DESTROY

Το `class` δεν εισάγει μια μορφή καταστροφέα πρώτης τάξης. Αν η κλασική κλάση έχει `DESTROY`, γράψτε την ως απλή μέθοδο μέσα στο μπλοκ `class` - καλείται μέσω του ίδιου hook μετρητή αναφορών:

```
class Logger {
    field $fh;

    ADJUST { ... }

    sub DESTROY {
        my $self = shift;
        close $self->{fh} if $self->{fh};
    }
}
```

Προσέξτε τον σπυρί: μέσα στη DESTROY δεν μπορείτε να ονομάσετε το πεδίο με το δηλωμένο όνομά του, επειδή τα σώματα sub δεν βλέπουν τις συνδέσεις πεδίων. Είτε χρησιμοποιήστε μια κανονική μέθοδο στην οποία αναθέτει η DESTROY, είτε αποδεχτείτε την ελαφρά ασυμμετρία.

Ένα καθαρότερο πρότυπο όταν η διάρκεια ζωής είναι υπό τον έλεγχό σας είναι να προσθέσετε μια ρητή μέθοδο close και να την καλείτε ντετερμινιστικά, αφήνοντας τη DESTROY ως ζώνη-και-τιράντες εφεδρεία.

Στιγμιότυπα που δεν είναι hash

Αν η κλασική κλάση κάνει bless \@self, $class ή bless \$self, $class - στιγμιότυπο με υπόβαθρο πίνακα ή βαθμωτού - το σύγχρονο χαρακτηριστικό δεν έχει ισοδύναμο. Η διάταξη στιγμιότυπου τύπου hash είναι ψημένη μέσα. Κρατήστε αυτές τις κλάσεις κλασικές.

Tied hashes / overloading

Οι ενσωματώσεις use overload και tie λειτουργούν αμετάβλητες σε μπλοκ class. Τίποτα ιδιαίτερο να κάνετε.

Μετάβαση κλάσεων Moose / Moo

Η μετάφραση από Moose σε class είναι καθαρότερη από εκείνη από κλασικό κώδικα επειδή το Moose είναι ήδη δηλωτικό. Ο αδρός χάρτης:

Moose / Moo	Σύγχρονο class
use Moose; / use Moo;	use feature 'class'; συν μπλοκ
has foo => (is => 'ro', ...)	field $foo :param :reader
has foo => (is => 'rw', ...)	field $foo :param :reader :writer
has foo => (default => sub { ... })	field $foo = EXPR (ή ADJUST)
extends 'Parent'	class Sub :isa(Parent) { ... }
BUILD { ... }	ADJUST { ... }
with 'Role::Name'	*(αναμένετε τους ρόλους του πυρήνα)*
has foo => (isa => 'Int', ...)	*(κανένα άμεσο ισοδύναμο)*
method_modifiers (before/after/around)	*(κανένα άμεσο ισοδύναμο)*

Τα στοιχεία που λείπουν - περιορισμοί τύπων, method modifiers, ρόλοι - είναι αυτά που κρατούν μερικές βάσεις κώδικα Moose στο Moose προς το παρόν. Αν η χρήση του Moose περιορίζεται σε `has`, `extends` και `BUILD`, η μετάβαση είναι άμεση. Αν εξαρτάται έντονα από το σύστημα τύπων του Moose ή από τους method modifiers, η μετάβαση είναι έργο, όχι commit.

Δοκιμή της μετάβασης

Μια μετάβαση κλάσης που δεν αλλάζει εξωτερική συμπεριφορά θα έπρεπε να περάσει την υπάρχουσα σουίτα δοκιμών αμετάβλητη. Βήματα:

1. Βεβαιωθείτε ότι η κλάση έχει δοκιμές που καλύπτουν το δημόσιο API της. Αν δεν έχει, γράψτε τες **πριν** τη μετάβαση - οι δοκιμές είναι το δίχτυ ασφαλείας της μετάβασης.
2. Μεταφέρετε την κλάση.
3. Τρέξτε τις δοκιμές. Ένα καθαρό πέρασμα είναι το σήμα ολοκλήρωσης.
4. Κάντε grep στη βάση κώδικα για προσβάσεις `$obj->{...}` στα στιγμιότυπα της μεταφερμένης κλάσης. Διορθώστε όσα βρείτε.
5. Αφαιρέστε όποιο πλέον νεκρό `use parent`, `use base`, ρητό `sub new` ή boilerplate accessor χρειαζόταν η παλιά μορφή.

Αν το βήμα 3 αποτύχει, οι αποτυχίες συνήθως εμπίπτουν σε μια σύντομη λίστα:

- **Άμεση πρόσβαση hash** σε μέθοδο που η παλιά μορφή ανεχόταν και η νέα μορφή απορρίπτει. Αντικαταστήστε με αναφορές πεδίων.
- **Ένα όρισμα κατασκευαστή που γινόταν σιωπηρά αποδεκτό.** Ο δημιουργημένος κατασκευαστής είναι αυστηρός με τα ονόματα `:param`. Είτε δηλώστε την παράμετρο είτε αλλάξτε το σημείο κλήσης.
- **Ένας αρχικοποιητής πεδίου που αναφερόταν στο `$self`.** Οι αρχικοποιητές πεδίων εκτελούνται πριν συνδεθεί το `$self`. Μετακινήστε την εξαρτώμενη λογική σε ένα μπλοκ `ADJUST`.

Τι κερδίζετε από αυτό

Η μετάβαση από κλασική σε σύγχρονη είναι ασυνήθιστη στο ότι το αποτέλεσμα είναι ταυτόχρονα συντομότερο και πιο σωστό:

- **Συντομότερο.** Το παράδειγμα `Counter` παραπάνω πέφτει από ~20 γραμμές σε ~10, και οι γραμμές που αφαιρέθηκαν είναι όλες boilerplate.
- **Ιδιωτικά πεδία.** Ο εξωτερικός κώδικας δεν μπορεί πια να φτάσει μέσα. Η ενθυλάκωση που υπερασπιζόσασταν στην ανασκόπηση κώδικα τώρα την υπερασπίζεται ο μεταγλωττιστής.
- **Κανένα ξεχασμένο `bless`.** Ο δημιουργημένος κατασκευαστής είναι πάντα ασφαλής ως προς την κληρονομικότητα· δεν μπορείτε να στείλετε έναν κατασκευαστή που κατά λάθος κάνει bless σε λάθος κλάση.
- **Κανένα ξεχασμένο `$self = shift`.** Η σιωπηρή σύνδεση δεν μπορεί να λείψει.

- **Μια ονομασμένη σημαία χαρακτηριστικού.** Το `no warnings 'experimental::class'` τεκμηριώνει στην κορυφή του αρχείου από τι εξαρτάται η κλάση.

Περαιτέρω ανάγνωση

- *Σύγχρονες κλάσεις* - αναφορά για τη σύνταξη προς την οποία μεταβαίνετε.
- *Κλασική αντικειμενοστρέφεια* - αναφορά για τη σύνταξη από την οποία μεταβαίνετε.
- *Κληρονομικότητα και επίλυση μεθόδων* - το κεφάλαιο για τις μεικτές ιεραρχίες, χρήσιμο όσο η βάση κώδικα είναι μισομεταφερμένη.
- *Ρόλοι και ανάθεση* - όπου καταλήγει η πολλαπλή κληρονομικότητα στον μεταφερμένο κώδικα.
- `class`, `field`, `method` - οι σελίδες αναφοράς για τις τρεις δεσμευμένες λέξεις στην καρδιά της μετάβασης.
- `bless`, `ref`, `isa` - οι πρωτογενείς οντότητες που στηρίζουν τόσο την παλιά όσο και τη νέα μορφή και που εξακολουθούν να λειτουργούν σε μεικτές ιεραρχίες.

4.6 Αποσφαλμάτωση προγραμμάτων Perl

Η Perl διαθέτει μια ασυνήθιστα πλήρη επιφάνεια αποσφαλμάτωσης: έναν ενσωματωμένο αποσφαλματωτή που δουλεύει δήλωση προς δήλωση, έναν ιχνηλάτη της μηχανής regex, ένα τεκμηριωμένο API για τη συγγραφή εναλλακτικών αποσφαλματωτών, και ένα οικοσύστημα στο CPAN από προφίλερ, εντοπιστές διαρροών, εργαλεία post-mortem, και ενσωματώσεις IDE. Η εκ νέου υλοποίηση της pperl διατηρεί τα σημεία hook του διερμηνέα, οπότε όλη η στοίβα δουλεύει: `perl -d`, `Devel::NYTProf`, `Devel::Cover`, `Perl::LanguageServer`, και η υπόλοιπη οικογένεια `-d:Module`.

4.6.1 Ποιο κεφάλαιο θέλω;

Σύμπτωμα	Ξεκινήστε εδώ
Το πρόγραμμα συμπεριφέρεται παράξενα και θέλω να το επιθεωρήσω ζωντανά	*interactive-debugger*
Το πρόγραμμα καταρρέει με ένα κρυπτικό μήνυμα	*exceptions*
Θέλω να σκορπίσω print και dump	*print-and-die*
Θέλω να μετατρέψω περισσότερα λάθη σε σφάλματα μεταγλώττισης	*preflight*
Πρέπει να σταματήσω σε μια συγκεκριμένη γραμμή ή συνθήκη	*breakpoints*
Πρέπει να δω τι περιέχει πραγματικά μια δομή δεδομένων	*inspecting-state*
Θέλω να βλέπω κάθε γραμμή / sub καθώς εκτελείται	*tracing*
Θέλω breakpoints στο VS Code / Neovim	*ide-dap*

4.6.2 Τα τρία επίπεδα αυτού του οδηγού

- **Προσανατολισμός** - αυτή η σελίδα.
- **Προγραμματιστής στη δουλειά** - τα κεφάλαια που αναφέρονται παραπάνω. Κάθε ένα είναι αυτοτελές· η ανάγνωση ενός μόνον αρκεί για να λύσει το πρόβλημα που κατονομάζει.

4.6.3 Γνωστά κενά

Επιφάνειες αποσφαλμάτωσης όπου το ίδιο το οικοσύστημα είναι ισχνό, ή όπου η υλοποίηση της pperl δεν έχει φτάσει σε ισοτιμία με το upstream. Κατονομάζονται εδώ ώστε οι αναγνώστες να μη σπαταλούν χρόνο σε αυτές.

- **Μορφή εξόδου του ιχνηλάτη regex.** Η μηχανή regex της pperl είναι μια ντόπια εκ νέου υλοποίηση· το `use re 'debug'` παράγει ίχνος αλλά τα ονόματα κόμβων και οι μορφές πεδίων δεν είναι byte-πανομοιότυπες με αυτές της perl5. Καταναλωτές που αναλύουν το ίχνος (λίγοι υπάρχουν) θα χρειαστούν προσαρμογή.
- **Αρθρώματα ενδοσκόπησης `B::*`.** Τα `B::Deparse`, `B::Concise`, και συγγενικά εκθέτουν τον γράφο op της perl5. Η αναπαράσταση op της pperl διαφέρει· ντόπια ισοδύναμα είναι σε εξέλιξη.
- **Αρθρώματα ενδοσκόπησης που εξαρτώνται από XS.** Η οικογένεια `Devel::Peek`, `Devel::FindRef`, `Devel::Gladiator`, `Devel::Refcount`, `Devel::Size`, `Devel::MAT::Dumper`, `Devel::Leak` - όλα φτάνουν στο SV μέσω XS (μακροεντολές επιπέδου C επί του δείκτη SV). Το p5 runtime της pperl αντικατοπτρίζει πιστά τη διάταξη SV της perl5, αλλά η φόρτωση XS δεν υποστηρίζεται· συνεπώς αυτά τα αρθρώματα χρειάζονται ντόπιες (σε Rust) εκ νέου υλοποιήσεις που εκθέτουν το ίδιο API επιπέδου Perl. Η λειτουργικότητα τύπου `Devel::Peek` σε καθαρή Perl παρέχεται ήδη ντόπια από το ενσωματωμένο `Devel::Peek`· τα υπόλοιπα είναι σε εξέλιξη.
- **Το `Enbugger` σε Perl ⩾ 5.38.** Τελευταία έκδοση το 2012· δουλεύει ανέκδοτα σε σύγχρονες perl αλλά δεν έχει ενεργό συντηρητή. Η διαδρομή `gdb-inject-perl` είναι πιο αξιόπιστη.
- **Προέλευση async.** Δεν υπάρχει ισοδύναμο στην Perl με το `AsyncLocalStorage` του Node: ένα `Future` που αποτυγχάνει σε μια απομακρυσμένη επανάκληση δεν φέρει αυτόματη εγγραφή του ποιος το προγραμμάτισε.
- **Breadcrumbs Sentry.** Το `Sentry::Raven` είναι λειτουργικό αλλά μόνο σε φάση συντήρησης· δεν διαθέτει υποστήριξη breadcrumb πρώτης τάξης. Το OpenTelemetry Perl SDK είναι ο μελλοντικός δρόμος για νέες εγκαταστάσεις.
- **Σύνδεση του `Perl::LanguageServer` σε εκτελούμενη διεργασία.** Η εκκίνηση υποστηρίζεται· η εκ των υστέρων σύνδεση όχι.

4.6.4 Δείτε επίσης

- `perlfunc/die`, `perlfunc/warn`, `perlfunc/eval`, `perlfunc/caller` - τα γλωσσικά πρωτογενή πάνω στα οποία οικοδομούνται ο αποσφαλματωτής και τα εργαλεία ίχνους στοίβας.
- *PPerl Architecture guide* - τα δύο runtimes και γιατί μόνο το p5 φέρει τον πλήρη αποσφαλματωτή.

Preflight: σύλληψη σφαλμάτων πριν τον χρόνο εκτέλεσης

Αυτό το κεφάλαιο καλύπτει τις pragmas, τους ελέγχους κατά τη μεταγλώττιση, και τις ρυθμίσεις υγιεινής χρόνου εκτέλεσης που εμποδίζουν τα περισσότερα σφάλματα να φτάσουν καν στον αποσφαλματωτή. Ένα καθαρό preflight είναι ο συντομότερος δρόμος για σύντομη συνεδρία αποσφαλμάτωσης.

Οι αναγνώστες που ήδη γράφουν `use v5.36` ή νεότερη τρέχουν εξ ορισμού υπό strict και warnings και μπορούν να μεταβούν απευθείας στις ενότητες διαγνωστικών και autoflush. Το υπόλοιπο του κεφαλαίου υποθέτει ότι ενδέχεται να συντηρείτε παλαιότερο κώδικα όπου ούτε το ένα ούτε το άλλο είναι σιωπηρά ενεργό.

Strict και warnings

Από την Perl 5.36, το `use v5.36;` (και οποιαδήποτε δήλωση μεγαλύτερης έκδοσης) ενεργοποιεί σιωπηρά τα `use strict;` και `use warnings;`. Τα σύγχρονα σενάρια δεν γράφουν αυτές τις γραμμές - η δήλωση έκδοσης καλύπτει και τα δύο:

```
use v5.40;
```

Το `use strict` αρνείται τρεις κλάσεις πρόχειρου κώδικα κατά τη μεταγλώττιση: αδήλωτες μεταβλητές (`strict vars`), συμβολικές αναφορές (`strict refs`), και barewords που χρησιμοποιούνται ως συμβολοσειρές (`strict subs`). Το `use warnings` ενεργοποιεί διαγνωστικά χρόνου εκτέλεσης για αναγνώσεις από μη αρχικοποιημένες τιμές, αριθμητικές μετατροπές με απώλειες, χρήση filehandles μετά το κλείσιμό τους, και αρκετές δεκάδες άλλα αιχμηρά άκρα.

Παλαιωμένος κώδικας που δεν χρησιμοποιεί κανένα από τα δύο επωφελείται και από τα δύο: προσθέστε τα, διορθώστε τα σύμβολα για τα οποία παραπονούνται, κάντε commit.

Στενές εξαιρέσεις παραμένουν χρήσιμες:

```
no strict 'refs';                   # genuine symbolic deref
no warnings 'uninitialized';        # known-safe defaulting
no warnings 'experimental::try';    # on 5.36, 5.38 - see below
```

Ποτέ μη απενεργοποιείτε γενικά: το `no warnings;` χωρίς λίστα κατηγοριών είναι σχεδόν πάντα λάθος.

Προαγωγή προειδοποιήσεων σε μοιραίες για μια περιοχή:

```
local $SIG{__WARN__} = sub { die $_[0] };
```

Χρήσιμο στην κορυφή του προγράμματος κατά την ανάπτυξη· η πρώτη προειδοποίηση γίνεται κατάρρευση που παράγει ίχνος στοίβας.

use diagnostics

Επεκτείνει κάθε προειδοποίηση ή μοιραίο μήνυμα στην πλήρη παράγραφο από το `perldiag`:

```
use diagnostics;
```

Η έξοδος είναι φλύαρη - πολύ φλύαρη για παραγωγή - αλλά κατά την αποσφαλμάτωση εξηγεί τι σημαίνει στην πραγματικότητα ένα λιτό μήνυμα όπως «Odd number of elements in hash assignment» και πώς να το διορθώσετε.

Εκ των υστέρων εκδοχή: σωληνώστε το stderr ενός προγράμματος μέσω της εντολής `splain` για να σχολιάσετε προειδοποιήσεις εκ των υστέρων.

Έλεγχοι κατά τη μεταγλώττιση

Το `perl -c script.pl` αναλύει το σενάριο και τρέχει κάθε `BEGIN` / `use`, μετά σταματάει πριν την πρώτη δήλωση χρόνου εκτέλεσης. Ένα καθαρό `-c` σημαίνει ότι το αρχείο και οι εξαρτήσεις του μεταγλωττίζονται· δεν σημαίνει ότι το πρόγραμμα τρέχει σωστά.

```
perl -c script.pl
script.pl syntax OK
```

Συνδυάστε με `-w` για να ενεργοποιήσετε καθολικά τις warnings για τη φάση μεταγλώττισης και κάθε κώδικα κατά τη μεταγλώττιση:

```
perl -cw script.pl
```

Το `perl -MO=Deparse script.pl` εκτυπώνει αυτό που πραγματικά ανέλυσε η perl - λάθος αναλύσεις προτεραιότητας, κλάδους που πτυχώθηκαν σε σταθερές, `for` που ξαναγράφτηκε ως `while`. Χρησιμοποιήστε το για να επαληθεύσετε ότι η perl είδε αυτό που εννοούσατε:

```
perl -MO=Deparse script.pl
```

Το `perl -MO=Deparse,-p` επιβάλλει πλήρη παρενθεσιοποίηση και κάνει την προτεραιότητα ρητή.

Το `perl -MO=Concise script.pl` εκτυπώνει συμπαγές dump του δέντρου op· κυρίως για αναγνώστες που κάνουν εργασία βαθιάς βελτιστοποίησης.

autodie: κάντε την αποτυχία να εκτοξεύει

Οι ενσωματωμένες που σηματοδοτούν αποτυχία μέσω ψευδών τιμών επιστροφής (`open`, `close`, `chdir`, `unlink`) είναι εύκολο να ξεχάσετε να ελέγξετε. Το `autodie` τις αντικαθιστά με εκδοχές που εκτοξεύουν:

```
use autodie qw(open close chdir);

open my $fh, '<', $path;      # now dies on failure; no `or die` needed
```

Το `autodie` είναι λεξιλογικό: επηρεάζονται μόνο οι κλήσεις στην εμβέλειά του. Στενέψτε την εξαίρεση για συγκεκριμένο σημείο κλήσης:

```
{
    no autodie 'open';
    open my $fh, '<', $path or warn "cannot read $path: $!";
}
```

Προτιμήστε το `autodie` έναντι του αποσυρμένου αρθρώματος `Fatal`, το οποίο δεν κάλυπτε συναρτήσεις που επιστρέφουν μετρητές.

Έξοδος χωρίς ενταμιευτή

Όταν η διαγνωστική έξοδος εξαφανίζεται πριν από κατάρρευση, ή φτάνει παράξενα διαπλεγμένη με μηνύματα `warn`, η αιτία είναι σχεδόν πάντα stdout με ενταμίευση μπλοκ. Ορίστε το `$|` στην κορυφή του σεναρίου:

```
$| = 1;
```

Κάθε `print` τώρα εκκενώνει άμεσα. Το κόστος είναι μια μικρή απώλεια απόδοσης I/O· το κέρδος είναι ότι αυτό που βλέπετε στο τερματικό αντιστοιχεί σε αυτό που πραγματικά έκανε το πρόγραμμα, με τη σωστή σειρά.

Για συγκεκριμένο filehandle χωρίς αλλαγή του επιλεγμένου:

```
use IO::Handle;
$log_fh->autoflush(1);
```

Συνηθισμένα πρώιμα διαγνωστικά και τι σημαίνουν

Διαγνωστικό	Σημασία
`Global symbol "$x" requires explicit package name`	Το strict είναι ενεργό και το `$x` δεν δηλώθηκε ποτέ· προσθέστε `my`.
`Odd number of elements in hash assignment`	Λίστα που ανατέθηκε σε κατακερματισμό έχει περιττό πλήθος· ένα `qw(...)` ή συμβολοσειρά με ενσωματωμένο κενό παρήγαγε δύο τιμές εκεί που αναμενόταν μία.
`Use of uninitialized value in ...`	Ένα `undef` έφτασε σε τελεστή. Συνήθως τυπογραφικό σε όνομα μεταβλητής ή return που λείπει.
`Name "main::x" used only once: possible typo`	Μια μεταβλητή πακέτου αναφέρθηκε ακριβώς μία φορά. Σχεδόν πάντα τυπογραφικό ή ένα `my` που λείπει.
`Can't locate Foo.pm in @INC`	Ένα `use` ή `require` δεν μπόρεσε να βρει το άρθρωμα. Ελέγξτε την ορθογραφία, το `@INC`, το `PERL5LIB`.
`Reference found where even-sized list expected`	`my %h = { ... }` - οι αγκύλες χτίζουν hashref· χρησιμοποιήστε παρενθέσεις.

Μάθετε περισσότερα

- *print-and-die* - όταν το preflight δεν αρκεί και χρειάζεστε δομημένα διαγνωστικά χρόνου εκτέλεσης.
- *exceptions* - μετατροπή διαδρομών αποτυχίας σε τυποποιημένες εξαιρέσεις στις οποίες ο καλών μπορεί να ταιριάξει.

Αποσφαλμάτωση με print, η warn, και η οικογένεια Carp

Αυτό το κεφάλαιο καλύπτει μη διαδραστικά διαγνωστικά: εισαγωγή print, dump μεταβλητών, ίχνη στοίβας από μέσα στο πρόγραμμα. Αυτές οι τεχνικές παραμένουν ο γρηγορότερος δρόμος για να καταλάβετε «τι βλέπει πραγματικά ο κώδικάς μου εδώ;» για σφάλματα που αναπαράγονται εύκολα.

Οι αναγνώστες που γνωρίζουν ήδη τις `print` και `warn` θα βρουν εδώ τις δομημένες παραλλαγές που αξίζουν τη χρήση τους: την οικογένεια Carp που αναφέρει από τον καλούντα, το `Data::Dumper` και τους σύγχρονους αντικαταστάτες του, και τα μονόγραμμα για ίχνος στοίβας παντού.

warn έναντι print - επιλέξτε τη σωστή ροή

Η `warn` γράφει στο STDERR· η `print` γράφει εξ ορισμού στο STDOUT. Χρησιμοποιήστε την `warn` για διαγνωστικά: διατηρεί την έξοδο του προγράμματος καθαρή, επιβιώνει της ανακατεύθυνσης του STDOUT, και είναι η ροή που οι loggers ήδη συλλαμβάνουν.

```
warn "*** foo=[$foo] bar=[@bar]\n";
```

Τυλίξτε τις τιμές σε κυριολεκτικές αγκύλες ώστε τα τελικά κενά και τα newline να είναι ορατά στην έξοδο. Το `"foo=[$foo]"` αποκαλύπτει `foo=[ ]` εκεί που το `"foo=$foo"` δείχνει απλώς `foo= .`

Χωρίς τελικό `\n`, η `warn` (και η `die`) προσθέτουν `at FILE line N.` από την οπτική του καλούντος. Με `\n`, δεν το κάνουν. Πρακτικός κανόνας: προσθέστε `\n` για μηνύματα προς τον χρήστη· παραλείψτε το για διαγνωστικά προγραμματιστή όπου θέλετε τη θέση.

Φτηνές ετικέτες θέσης από τον μεταγλωττιστή:

```
warn "reached $0 " . __FILE__ . ':' . __LINE__ . "\n";
warn "in @{[__SUB__->name]}\n";          # 5.16+
```

Η οικογένεια Carp - αναφορά από τον καλούντα

Όταν μια συνάρτηση βιβλιοθήκης αποτυγχάνει, το να δείχνει στον δικό της κώδικα σχεδόν ποτέ δεν είναι χρήσιμο. Η οικογένεια `Carp` αναφέρει από το πλαίσιο του καλούντος:

Συνάρτηση	Αναφέρει από	Μοιραίο;	Ίχνος στοίβας;
`carp`	caller	όχι	όχι
`croak`	caller	ναι	όχι
`cluck`	caller	όχι	πλήρες
`confess`	caller	ναι	πλήρες

```
use Carp;

sub open_config {
    my $path = shift // croak "open_config: missing path";
    open my $fh, '<', $path or croak "cannot read $path: $!";
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    return $fh;
}
```

Κώδικας βιβλιοθήκης πρέπει να χρησιμοποιεί croak/carp, όχι die/warn - το μήνυμα σφάλματος δείχνει στη γραμμή του καλούντος, εκεί που πρέπει να πάει η διόρθωση.

Ίχνη στοίβας χωρίς να αγγίξετε τον κώδικα

Το Carp::Always εγκαθιστά τους $SIG{__DIE__} και $SIG{__WARN__} ώστε να δρομολογούν μέσω των Carp::confess / Carp::cluck. Τρέξτε ένα προβληματικό σενάριο με:

```
perl -MCarp::Always script.pl
```

Κάθε die και warn παράγει πλήρη στοίβα. Χωρίς αλλαγές στον κώδικα. Αυτό είναι το πρώτο πράγμα στο οποίο καταφεύγει κανείς όταν μια κατάρρευση δίνει ένα άχρηστο μονόγραμμο μήνυμα.

Το Devel::Confess είναι απευθείας ισοδύναμος αντικαταστάτης με λιγότερες οριακές περιπτώσεις γύρω από overloaded αντικείμενα:

```
perl -d:Confess script.pl
```

Και τα δύο αρθρώματα προσθέτουν κόστος σε κάθε warn και die· χρησιμοποιήστε τα κατά τη διερεύνηση, όχι σε σταθερή παραγωγή.

Για μόνιμη εγκατάσταση μέσα στον κώδικα:

```
use Carp;
$SIG{__DIE__}  = sub { return if $^S; Carp::confess(@_) };
$SIG{__WARN__} = sub { Carp::cluck(@_) };
```

Το return if $^S; - «επιστροφή αν είμαστε μέσα σε eval». Χωρίς αυτόν τον φύλακα κάθε παγιδευμένη εξαίρεση πληρώνει το κόστος του ίχνους. Δείτε *exceptions* για το πλήρες μοτίβο $SIG{__DIE__}.

Dump δομών δεδομένων

Περάστε **αναφορά** σε κάθε dumper· το dump γυμνού κατακερματισμού ή πίνακα εκτυπώνει την επίπεδη μορφή λίστας και χάνει τη δομή.

Data::Dumper - καθολικό, στον πυρήνα, με δυνατότητα πλήρους κύκλου μέσω eval

```
use Data::Dumper;

$Data::Dumper::Sortkeys = 1;
$Data::Dumper::Indent   = 1;
$Data::Dumper::Terse    = 1;
$Data::Dumper::Deepcopy = 1;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
warn Dumper(\%config);
```

- Το `Sortkeys` καθιστά την έξοδο σταθερή σε diff.
- Το `Indent=1` είναι η ευανάγνωστη ρύθμιση (0 = επίπεδο, 2 = προεπιλογή, 3 = με δείκτες πίνακα).
- Το `Terse=1` καταστέλλει τα προθέματα `$VAR1 =` όταν δεν χρειάζεστε δυνατότητα πλήρους κύκλου.
- Το `Deepcopy=1` απενεργοποιεί την παρακολούθηση «έχω ξαναδεί αυτή την αναφορά» `$VAR1->{...}` - εκτυπώνει τα δεδομένα, όχι μια ανακατασκευή γράφου.

Το `Data::Dumper::Concise` είναι απευθείας αντικαταστάτης με λογικές προεπιλογές προεφαρμοσμένες· το πρότυπο της κοινότητας DBIx::Class / Moose.

`Data::Printer` - έγχρωμο, σύγχρονο, ευανάγνωστο

```
use DDP;
p %config;                    # not a reference - DDP handles both
```

Το `Data::Printer` εκτυπώνει εμφανισιακά καλύτερη έξοδο από το `Data::Dumper`, χρωματίζει ανά τύπο, χειρίζεται ειδικά τα αντικείμενα (κλάση, γνωρίσματα, overloads), και δεν επιχειρεί πλήρη κύκλο. Προτιμήστε το όταν ο στόχος είναι η ανάγνωση του dump, όχι η εκ νέου εισαγωγή του.

`Data::Dump` - συμπαγές μονόγραμμο

```
use Data::Dump qw(dd);
dd \%config;                  # prints to STDERR in compact form
```

Το `Data::Dump` του Gisle Aas. Χρήσιμο όταν η προεπιλεγμένη έξοδος του `Data::Dumper` είναι πολύ φλύαρη για μία γραμμή warn.

Πότε να καταφεύγει κανείς σε ποιο

- `Data::Dumper` όταν ενδέχεται να τροφοδοτήσετε την έξοδο πίσω σε `eval` ή να συγκρίνετε δύο dump.
- `Data::Printer` (DDP) όταν την έξοδο τη διαβάζει άνθρωπος.
- `Data::Dump` όταν θέλετε σύνοψη μιας γραμμής.
- `Data::Dumper::Concise` σε έργα που ήδη το χρησιμοποιούν.

Print εντοπισμού σφαλμάτων υπό συνθήκη

Μια σταθερά κατά τη μεταγλώττιση βελτιστοποιείται πλήρως όταν είναι ψευδής:

```
use constant DEBUG => 0;

warn "state=$state\n" if DEBUG;
```

Με `DEBUG => 0`, ο έλεγχος `if DEBUG` πτυχώνεται κατά τη μεταγλώττιση και η `warn` δεν εμφανίζεται ποτέ στο δέντρο op - μηδενικό κόστος χρόνου εκτέλεσης.

Συνδέστε στο περιβάλλον ώστε η σημαία να αλλάζει χωρίς επεξεργασία:

```
use constant DEBUG => $ENV{APP_DEBUG} // 0;
```

Πολλαπλά κανάλια μέσω μάσκας bits:

```
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";
dbg REGEX, "matched: $&\n";         # (unless you care about perf - see below)
```

Εναλλακτική με φίλτρο πηγαίου: `Smart::Comments`

```
use Smart::Comments;

my @stuff = compute();
### @stuff                  # prints: @stuff: [...]
### Checking key: $key      # print, plus asserts $key is truthy
### Processing |=| ... for @stuff
```

Γραμμές που ξεκινούν με `###` ξαναγράφονται ως print (και, με ειδική σύνταξη, μπάρες προόδου και διεκδικήσεις). Το άρθρωμα είναι φίλτρο πηγαίου - ξαναγράφει το κείμενο του προγράμματος πριν τη μεταγλώττιση - οπότε οι αναφερόμενοι αριθμοί γραμμών μπορούν να μετατοπιστούν κατά ένα. Όταν δεν εισάγεται, τα σχόλια `###` μεταγλωττίζονται σε τίποτα. Η μετατόπιση είναι το τίμημα του να μη γεμίζει ο κώδικας με ρητές γραμμές `warn`.

Taint και $& - δευτερεύοντα κόστη που πρέπει να γνωρίζετε

Η αναφορά στα `$&`, `$\``, ή `$'` **οπουδήποτε** στο πρόγραμμα αναγκάζει την perl να διατηρεί αυτές τις καθολικές μεταβλητές περιβάλλοντος αντιστοιχίας για **κάθε** αντιστοιχία regex σε όλο το πρόγραμμα, συμπεριλαμβανομένων αντιστοιχιών μέσα σε φορτωμένα αρθρώματα. Ιστορικά αυτό επέβαλλε αισθητή επιβράδυνση· στις σύγχρονες perl το κόστος είναι μικρότερο αλλά μη μηδενικό, και ο κανόνας ισχύει ακόμη: μην τις αγγίζετε αν μπορείτε να το αποφύγετε.

Χρησιμοποιήστε τα `@-` / `@+` (πίνακες θέσεων αντιστοιχίας), ονομαστικές συλλήψεις, ή τον τροποποιητή `/p` με τα `${^MATCH}`, `${^PREMATCH}`, `${^POSTMATCH}` αντί αυτών. Οι παραλλαγές `/p` επιβάλλουν το κόστος μόνο για αντιστοιχίες που το επιλέγουν.

Καταγραφή με χρονική σφραγίδα

Μια warn φτιαγμένη με το χέρι με localtime είναι επαρκής για εφάπαξ διερευνήσεις:

```
use Time::HiRes qw(gettimeofday);
sub t { my ($s, $us) = gettimeofday; sprintf "%d.%06d", $s, $us }
warn sprintf "[%s] state=%s\n", t(), $state;
```

Για οτιδήποτε πιο δομημένο, χρησιμοποιήστε ένα framework καταγραφής αντί να φτιάχνετε δικό σας:

- Log::Any - ασφαλές για βιβλιοθήκες· εκπέμπει logs χωρίς να επιλέγει backend. Συνδυάστε με προσαρμογέα (Log::Any::Adapter::Log4perl, ::Stdout, ::Syslog) στην εφαρμογή.
- Log::Log4perl - κατηγορία / επίπεδο / appender / διάταξη· καθοδηγείται από αρχείο ρυθμίσεων.
- Log::Dispatch - δρομολογητής χαμηλότερου επιπέδου· συχνά χρησιμοποιείται κάτω από το Log::Any.

Πρότυπα επίπεδα, από το χαμηλότερο στο υψηλότερο: TRACE, DEBUG, INFO, WARN, ERROR, FATAL. Φυλάξτε τα TRACE / DEBUG πίσω από μια σημαία περιβάλλοντος στην παραγωγή.

Μάθετε περισσότερα

- *exceptions* - die, $@, $SIG{__DIE__}, τυποποιημένες κλάσεις εξαιρέσεων, το σύγχρονο try/catch.
- *interactive-debugger* - όταν η εισαγωγή print κοστίζει περισσότερο από την εκκίνηση του perl -d.
- *tracing* - ίχνος γραμμή-προς-γραμμή ολόκληρου του προγράμματος χωρίς επεξεργασία του πηγαίου.

Εξαιρέσεις, die, και τυποποιημένα αντικείμενα σφαλμάτων

Αυτό το κεφάλαιο καλύπτει το πώς εγείρετε, παγιδεύετε, και ταξινομείτε αποτυχίες χρόνου εκτέλεσης: η die και η eval, το ενσωματωμένο try/catch, η υγιεινή των ειδικών μεταβλητών ($@, $!, $?), το $SIG{__DIE__}, και ο σχεδιασμός κλάσεων εξαιρέσεων.

Οι αναγνώστες που γνωρίζουν τα die και eval { } θα βρουν εδώ τους σύγχρονους αντικαταστάτες για αποστολή με βάση συμβολοσειρές πάνω σε μηνύματα εξαιρέσεων και τους κινδύνους του $SIG{__DIE__} που το τυπικό μοτίβο eval σιωπηρά κρύβει.

Η die και η eval

Η die εκτοξεύει μια εξαίρεση· το πλησιέστερο περικλείον eval { } την παγιδεύει και αφήνει το μήνυμα στο $@:

```
eval {
    risky();
};
if (my $err = $@) {
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    warn "caught: $err";
}
```

Χωρίς τελικό \n, το όρισμα της die παίρνει at FILE line N. προσαρτημένο από το πλαίσιο του καλούντος. Με \n, δεν προσαρτάται θέση - χρησιμοποιήστε το για μηνύματα που θα παρουσιαστούν στους χρήστες, παραλείψτε το για μηνύματα που θέλετε να ιχνηλατήσετε.

Μη χρησιμοποιείτε ποτέ τη μορφή συμβολοσειράς eval "..." για χειρισμό εξαιρέσεων. Μεταγλωττίζει το όρισμά της και φέρει κάθε κίνδυνο κώδικα που παρέχει ο χρήστης.

Το ενσωματωμένο try/catch

Σε Perl 5.34 και μεταγενέστερες, υπάρχει διαθέσιμο ένα ειδικό try/catch πίσω από σημαία χαρακτηριστικού. Από την 5.40 δεν είναι πειραματικό:

```
use v5.40;
use feature 'try';

try {
    risky();
}
catch ($e) {
    warn "caught: $e";
}
```

finally remains experimental as of 5.44. Expect:

```
use feature 'try';
no warnings 'experimental::try';
```

σε perl μεταξύ 5.34 και 5.40.

Για κώδικα που πρέπει να τρέχει σε 5.34 ή 5.36 και επίσης σε νεότερες perl, το Feature::Compat::Try διαλέγει το χαρακτηριστικό του πυρήνα όπου είναι διαθέσιμο και πέφτει εφεδρικά στο Syntax::Keyword::Try:

```
use Feature::Compat::Try;

try { risky(); }
catch ($e) { warn "caught: $e" }
```

Γιατί το ενσωματωμένο try έναντι του Try::Tiny

Το Try::Tiny εξακολουθεί να δουλεύει και παραμένει σύνηθες. Προτιμήστε την ενσωματωμένη μορφή για νέο κώδικα: αποφεύγει το πλαίσιο κλήσης sub που εισάγει το Try::Tiny (το οποίο εμφανίζεται σε ίχνη στοίβας), έχει μηδενικό κόστος εκκίνησης, και συντίθεται καθαρά με τον τελεστή isa.

Τα παρακάτω έχουν αποσυρθεί - μην τα υιοθετείτε:

- TryCatch - εγκαταλελειμμένο.

- `Error.pm` - έχει αντικατασταθεί.
- `Exception::Class::TryCatch` - έχει αντικατασταθεί από το `Throwable`.

Αλλοίωση του $@ - ο κλασικός κίνδυνος της eval

Το `$@` τίθεται από την `eval`· είναι επίσης παρατηρήσιμο καθολικά. Οποιοσδήποτε κώδικας τρέχει μεταξύ της επιστροφής της `eval` και του ελέγχου σας στο `$@` - μια μέθοδος `DESTROY`, ένας χειριστής tied μεταβλητής, μια άσχετη εμφωλευμένη `eval` - μπορεί να το επανεγγράψει. Πάντα αντιγράψτε πρώτα:

```
eval { risky(); };
my $err = $@;
return unless $err;
```

Το ενσωματωμένο `try/catch` και το `Syntax::Keyword::Try` αποφεύγουν αυτή την παγίδα δίνοντάς σας το σφάλμα ως παράμετρο του `catch`· η σκέτη `eval` όχι. Προτιμήστε το πρώτο για οποιονδήποτε κώδικα παίρνει στα σοβαρά τις εξαιρέσεις.

Ειδικές μεταβλητές γειτονικές στις εξαιρέσεις

Μεταβλη	Τίθεται από	Διαβάστε άμεσα διότι...
`$@`	`eval { }` / `die`	οποιαδήποτε ενδιάμεση κλήση μπορεί να το επανεγγράψει.
`$!`	Τελευταία αποτυχημένη κλήση συστήματος	οποιαδήποτε επόμενη κλήση συστήματος το επανεγγράφει.
`$?`	παιδί `system` / backticks	το επόμενο παιδί το επανεγγράφει.
`$^E`	Εκτεταμένο σφάλμα OS	σε μη-Unix· σε Linux συνήθως αντικατοπτρίζει το `$!`.

Αποκωδικοποίηση του `$?` μετά από `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; ... }
```

Ταξινόμηση εξαιρέσεων: αποστολή με βάση τον τύπο

Για μη τετριμμένα προγράμματα, η διάκριση «δεν βρέθηκε χρήστης» από «η βάση δεδομένων είναι εκτός» από «σφάλμα προγραμματισμού» μέσω ανάλυσης συμβολοσειρών σφάλματος είναι εύθραυστη. Εκτοξεύστε τυποποιημένα αντικείμενα και ταιριάξτε σε αυτά:

```
use v5.40;
use feature 'try';
use Scalar::Util qw(blessed);
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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");
    }
}
```

Ο τελεστής `isa` (σταθερός σε 5.36+) κάνει το σωστό για overloaded αντικείμενα και είναι μηδενικού κόστους όταν ο αριστερός τελεστέος δεν είναι αναφορά. Συνδυάστε τον με έναν τυποποιημένο περιτυλιγτή στο πιο εξωτερικό όριο ώστε όλα προς τα κάτω να είναι `MyApp::Err`.

Σχεδιασμός κλάσεων εξαιρέσεων

Η προεπιλογή του 2026: ρόλος `Throwable` από το CPAN, προαιρετικά μέσω του `Throwable::SugarFactory` για δηλωτικές ιεραρχίες.

Ελάχιστη βιώσιμη κλάση:

```
package MyApp::Err::NotFound;
use Moo;
with 'Throwable';

has resource => ( is => 'ro', required => 1 );
has id       => ( is => 'ro', required => 1 );

sub message {
    my $self = shift;
    sprintf 'not found: %s/%s', $self->resource, $self->id;
}

1;
```

Χρήση:

```
MyApp::Err::NotFound->throw(resource => 'user', id => $uid);
```

Η `throw` παρέχεται από τον ρόλο· κάνει die με `$self` (το blessed αντικείμενο). Η `->new` χωρίς `throw` κατασκευάζει χωρίς να εγείρει.

Δηλωτική μορφή μέσω `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;
```

Αυτό παράγει: την κλάση, έναν βοηθό κατασκευαστή-και-εκτόξευσης `not_found(id => ...)`, ένα κατηγόρημα `is_not_found($e)`, και `->to_hash` σε κάθε στιγμιότυπο για σειριοποίηση JSON.

Τι πρέπει να έχει μια κλάση

1. **Όνομα κλάσης** - ώστε το `$e isa MyApp::Err::X` να αποστέλλει.
2. **Δομημένα πεδία** - `id`, `resource`, `code`.
3. **Ένα ανθρώπινο `message`** - χωρίς αυτό, η `warn $e` εκτυπώνει `MyApp::Err=HASH(0x...)`.
4. **Σειριοποίηση** - `->to_hash` για καταγραφή / aggregators / JSON API.
5. **Προέλευση** - αρχείο + γραμμή που συλλαμβάνονται στο `throw`. Το `Throwable` τα συλλαμβάνει αυτόματα.
6. **Ίχνος στοίβας** αν αξίζει να διερευνηθεί το σφάλμα αργότερα: το `with 'StackTrace::Auto'` προσθέτει την `->stack_trace`.

Πότε να μη χρησιμοποιείτε κλάση

Η `die` με σκέτη συμβολοσειρά παραμένει σωστή για σύντομα σενάρια, εφάπαξ εργασίες, και κάθε περίπτωση όπου κανένας καλών δεν θα αποστείλει με βάση το σφάλμα:

```
die "permission denied: $path\n";
```

Το τελικό `\n` καταστέλλει την προσάρτηση αρχείου-και-γραμμής· το μήνυμα απευθύνεται στον χρήστη.

Η `die` με αναφορά κατακερματισμού - `die { type => 'not_found', id => $uid }` - είναι μεταβατική μορφή. Τη στιγμή που ένας καλών γράφει `if (ref $e eq 'HASH' && $e->{type} eq 'not_found')` τρεις φορές, αναδιαρθρώστε σε κλάση.

Αποσυρμένα frameworks εξαιρέσεων - μην τα υιοθετείτε:

- `Error.pm` - χρησιμοποιήστε `Throwable` (βασισμένο σε ρόλους) ή `Exception::Class`.
- `Class::Throwable` - λειτουργικό· το `Throwable` είναι η σύγχρονη επιλογή.

- `Exception::Class::TryCatch` - έχει αντικατασταθεί από το ενσωματωμένο `try`.

`$SIG{__DIE__}` - χειριστείτε με προσοχή

Το `$SIG{__DIE__}` πυροδοτείται σε κάθε `die`, **συμπεριλαμβανομένων** των `die` μέσα σε `eval`. Μια αφελής εγκατάσταση μετατρέπει κάθε παγιδευμένη εξαίρεση σε θόρυβο:

```
# WRONG: fires on every eval that catches an error
$SIG{__DIE__} = sub { Carp::confess(@_) };
```

Η ειδική μεταβλητή `$^S` σας λέει αν βρίσκεστε μέσα σε `eval`: αληθές σημαίνει «ναι, κάποιος θα την παγιδεύσει αυτή»· ψευδές σημαίνει «αυτή φτάνει στην κορυφή». Φυλαχτείτε με βάση αυτή:

```
$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
};
```

Για ενσωμάτωση Sentry / aggregator, τυλίξτε την κλήση του reporter σε δικό της `eval` - ένας reporter σφαλμάτων που αποτυγχάνει δεν πρέπει να επισκιάσει την αρχική εξαίρεση.

Πριν εγκαταστήσετε πάνω από έναν υπάρχοντα χειριστή, αλυσιδώστε:

```
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__}` - κάντε τις προειδοποιήσεις μοιραίες, ή δρομολογήστε τις

Συμμετρικό προς το `$SIG{__DIE__}`, πυροδοτείται σε κάθε `warn`:

```
$SIG{__WARN__} = sub { die $_[0] };      # promote warnings to
↪exceptions
```

Χρήσιμο κατά την ανάπτυξη ως παγίδα συναγερμού· σπάνια επιθυμητό στην παραγωγή. Πιο συνηθισμένη χρήση στην παραγωγή: δρομολόγηση προειδοποιήσεων μέσω logger.

Μάθετε περισσότερα

- *print-and-die* - η οικογένεια Carp και διακόπτες για ίχνος στοίβας παντού.
- `perlfunc/die`, `perlfunc/eval`, `perlfunc/warn`, `perlvar` - σελίδες αναφοράς για τα γλωσσικά πρωτογενή.

Ο διαδραστικός αποσφαλματωτής

Αυτό το κεφάλαιο καλύπτει το `perl -d`: εκκίνηση προγράμματος υπό τον αποσφαλματωτή, την προτροπή, τις απαραίτητες εντολές κίνησης και επιθεώρησης, και το λεξιλόγιο πλοήγησης στη στοίβα. Μετά από αυτό το κεφάλαιο μπορείτε να διερευνήσετε διαδραστικά οποιοδήποτε εκτελούμενο πρόγραμμα.

Οι αναγνώστες ξεκινούν γνωρίζοντας ότι τα `print` / `warn` δουλεύουν αλλά δεν αρκούν: πρέπει να σταματήσετε, να κοιτάξετε γύρω σας, να ρίξετε ματιά σε μεταβλητές, να διασχίσετε τη στοίβα κλήσεων, και μετά να συνεχίσετε. Αυτό προσφέρει αυτό το κεφάλαιο.

Εκκίνηση

Για οποιοδήποτε σενάριο `app.pl`, προσθέστε `-d` μπροστά:

```
perl -d app.pl --verbose input.txt
```

Τα ορίσματα μετά το όνομα του σεναρίου περνούν στο σενάριο κανονικά.

Συνηθισμένες μορφές εκκίνησης:

Μορφή	Σκοπός
`perl -d script.pl args`	Αποσφαλμάτωση του δοθέντος σεναρίου.
`perl -d -e 'CODE'`	Αποσφαλμάτωση έκφρασης μονογράμμου.
`perl -de 0`	Διαδραστικό sandbox REPL.
`perl -d:Module script.pl`	Φόρτωση του `Devel::Module` ως αποσφαλματωτή.
`perl -d:NYTProf script`	Προφιλάρισμα μέσω του πρόσθετου NYTProf.
`perl -S -d script.pl`	Αναζήτηση του `$PATH` για το σενάριο.

Το `-d` αναγκάζει τον μεταγλωττιστή να εκπέμπει σημεία hook ανά δήλωση· ο διερμηνέας προφορτώνει το `perl5db.pl` πριν την πρώτη γραμμή χρόνου εκτέλεσης.

Μη σωληνώνετε ποτέ είσοδο στο `perl -d` και μη χρησιμοποιείτε ποτέ το `/dev/stdin` - ο αποσφαλματωτής χρειάζεται TTY για να διαβάσει εντολές. Εκκινήστε από πραγματικό τερματικό.

Η προτροπή

```
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.

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
main::(app.pl:4):     my $key = 'welcome';
  DB<1>
```

Η γραμμή `main::(app.pl:4):` δείχνει την επόμενη γραμμή που **πρόκειται να εκτελεστεί** - τίποτα στο πρόγραμμα δεν έχει τρέξει ακόμη. Ο κώδικας κατά τη μεταγλώττιση (`BEGIN`, `use`) έχει ήδη τρέξει· η πρώτη προτροπή έρχεται πριν από την πρώτη δήλωση χρόνου εκτέλεσης.

Το `DB<1>` - ο αριθμός είναι μετρητής εντολών που αυξάνει σε κάθε εντολή. Το `!N` ανακαλεί την εντολή με αριθμό N. Το `DB<<N>>` με διπλές αγκύλες σημαίνει ότι βρίσκεστε μέσα σε εμφωλευμένο αποσφαλματωτή (μια έκφραση στην οποία μπήκατε με βηματική εκτέλεση κάλεσε άλλη έκφραση στην οποία μπορεί να σπάσει η εκτέλεση).

Ο αποσφαλματωτής δείχνει πάντα τη γραμμή που πρόκειται να εκτελεστεί, ποτέ τη γραμμή που μόλις εκτελέστηκε.

Έξοδος

```
  DB<1> q
```

Όχι `quit`, όχι `exit`. Η εγγραφή `exit` τρέχει την `exit` της Perl μέσα στην `eval` του αποσφαλματωτή, που δεν είναι το ίδιο. Το `^D` στην προτροπή δουλεύει στα περισσότερα τερματικά.

Βοήθεια

```
  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}
```

Πού βρίσκομαι;

Το `v` δείχνει ένα παράθυρο γύρω από την τρέχουσα γραμμή. Το `l` παραθέτει ένα καθορισμένο εύρος:

```
  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      );
```

- Το `==>` σημαίνει την επόμενη γραμμή προς εκτέλεση.

- Το : σημαίνει εκτελέσιμες γραμμές - μπορείτε να ορίσετε breakpoint εκεί.
- Γραμμές χωρίς : δεν δέχονται διακοπή (σχόλια, κενά, συνέχειες πολυγραμμικής δήλωσης).

Πλοήγηση:

Εντολή	Αποτέλεσμα
`v`	Προβολή παραθύρου γύρω από την τρέχουσα γραμμή.
`v LINE`	Προβολή παραθύρου γύρω από τη γραμμή LINE.
`l`	Παράθεση του επόμενου παραθύρου· η επανάληψη προχωράει.
`l LINE`	Παράθεση μόνο της γραμμής LINE.
`l MIN-MAX`	Παράθεση εύρους.
`l SUB`	Παράθεση του πρώτου παραθύρου της SUB.
`-`	Προηγούμενο παράθυρο.
`.`	Επαναφορά της προβολής στην τρέχουσα γραμμή.
`f FILE`	Εναλλαγή της προβολής στο FILE (regex έναντι του %INC).
`/PAT/`	Αναζήτηση προς τα εμπρός στον πηγαίο.
`?PAT?`	Αναζήτηση προς τα πίσω.

Κίνηση: s, n, Enter, c, r

Οι τέσσερις απαραίτητες εντολές κίνησης:

Εντολή	Αποτέλεσμα
`s`	Step - εκτέλεση της επόμενης δήλωσης· είσοδος μέσα σε υπορουτίνες.
`n`	Next - εκτέλεση της επόμενης δήλωσης· πέρασμα πάνω από τις υπορουτίνες.
`<Enter>`	Επανάληψη του τελευταίου `s` ή `n`.
`c`	Συνέχεια ως το επόμενο breakpoint ή το τέλος.
`c LINE`	Συνέχεια ως ένα breakpoint μίας χρήσης στη γραμμή LINE, μετά διακοπή.
`c SUB`	Συνέχεια ως την πρώτη γραμμή της SUB.
`r`	Επιστροφή από την τρέχουσα sub· διακοπή στον καλούντα.

```
 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
```

Μια πολυγραμμική δήλωση εκτελείται ατομικά με ένα βήμα - ένα `s` εκτελεί ολόκληρη πολυγραμμική ανάθεση κατακερματισμού.

Σε γραμμή που δεν καλεί υπορουτίνες, τα `s` και `n` είναι αδιάκριτα μεταξύ τους.

Εκτύπωση τιμών: p και x

```
  DB<1> p $key
welcome

  DB<2> p join ',', sort keys %data
this,tom,welcome,zip
```

Το `p EXPR` αποτιμά την EXPR στο τρέχον πακέτο του προγράμματος και εκτυπώνει το αποτέλεσμα σε βαθμωτό περιβάλλον ακολουθούμενο από newline. Το p είναι επίπεδο - για αναφορά εκτυπώνει `HASH(0x...)`, όχι περιεχόμενα.

Για δομές, χρησιμοποιήστε το x:

```
  DB<3> x \%data
0  HASH(0x5651f9a82358)
   'this' => 'that'
   'tom' => 'and'
   'welcome' => 'jerry'
   'zip' => 'Hello World'
```

Πάντα περάστε **αναφορά** (`x \%h`, `x \@a`) - το `x %h` κάνει dump την επίπεδη λίστα κλειδιού/τιμής, που κρύβει τη δομή.

Δείτε *inspecting-state* για το πλήρες λεξιλόγιο επιθεώρησης (`V`, `X`, `y`, `m`, `M`, `i`, χειριστήρια βάθους).

Η στοίβα κλήσεων: T, s, r

Το `T` εκτυπώνει backtrace - την αλυσίδα κλήσεων sub που οδήγησαν στην τρέχουσα γραμμή:

```
  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
```

- `$` - κλήση σε βαθμωτό περιβάλλον.
- `@` - κλήση σε περιβάλλον λίστας.
- `.` - κενό περιβάλλον.
- Τα ορίσματα εμφανίζονται με αποκοπή τύπου `Data::Dumper`· το ακριβές όριο μήκους είναι η επιλογή `maxTraceLen`.

Πλοηγηθείτε στη στοίβα με `s` (μέσα σε κλήση) και `r` (έξω από το τρέχον πλαίσιο). Δεν υπάρχει εντολή «frame up / frame down»: επανα-εκτελέστε βήματα αν φτάσετε πολύ βαθιά, ή χρησιμοποιήστε το b για να ορίσετε breakpoint στον καλούντα και c για να το φτάσετε.

Για να επιθεωρήσετε τις λεξιλογικές ενός εξωτερικού πλαισίου, χρησιμοποιήστε το `y LEVEL` από το *inspecting-state* - διαβάζει μέσω του `PadWalker` και δείχνει τις μεταβλητές

my της περικλείουσας εμβέλειας.

Αποτίμηση Perl στην προτροπή

Ό,τι δεν αναγνωρίζει ο αποσφαλματωτής ως εντολή υφίσταται `eval` ως κώδικας Perl στο πακέτο DB στην τρέχουσα λεξιλογική εμβέλεια:

```
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 $_
```

Ένα φρέσκο `my $x` στην προτροπή εξαφανίζεται όταν η εντολή επιστρέψει - η προτροπή έχει τη δική της λεξιλογική εμβέλεια. Χρησιμοποιήστε bareword, `our`, ή αλυσιδώστε σε μία γραμμή:

```
DB<1> our $tmp = compute();         # survives
DB<2> $tmp = compute(); p $tmp      # one-line chain
```

Επιβάλλετε ερμηνεία Perl όταν μια έκφραση μοιάζει με εντολή αποσφαλματωτή:

```
DB<1> ; h EXPR                      # ; prefix forces eval
DB<2> + h EXPR                      # + prefix works too
```

Επανεκκίνηση και rerun

Το `R` επανεκκινεί το πρόγραμμα μέσω `exec()`. Διατηρούνται ιστορικό, breakpoints, ενέργειες, επιλογές, και η γραμμή εντολών:

```
DB<1> R
```

Το `rerun` επανεκτελεί χωρίς να φύγει από τον αποσφαλματωτή, διατηρώντας κατάσταση:

Μορφή	Αποτέλεσμα
`rerun`	Επανεκκίνηση, χωρίς αναπαραγωγή προηγούμενων εντολών.
`rerun -N`	Επανεκκίνηση και αναπαραγωγή ως N εντολές πίσω.
`rerun N`	Επανεκκίνηση από την εγγραφή N του ιστορικού.

Επανορισμός sub στη μέση της συνεδρίας

Επικολλήστε ολόκληρο το `sub NAME { ... }` στην προτροπή. Ή επεξεργαστείτε αρχείο αντικατάστασης και φορτώστε το:

```
DB<1> do 'fixed.pl'
```

Οι μεταγενέστερες κλήσεις χρησιμοποιούν τον νέο ορισμό. Τα υπάρχοντα πλαίσια στοίβας συνεχίζουν με το παλιό σώμα.

Εντολές ως τώρα

Εντολή	Αποτέλεσμα
`h`, `h CMD`, `h h`	Βοήθεια: σύνοψη / μία εντολή / πλήρης.
`|CMD`	Σωλήνωση της εξόδου μέσω του `$ENV{PAGER}`.
`q`	Έξοδος.
`v`, `v LINE`	Προβολή πηγαίου γύρω από την τρέχουσα γραμμή / LINE.
`l`, `l LINE`, `l SUB`, `l MIN-MAX`	Παράθεση.
`.`	Επαναφορά στην τρέχουσα γραμμή.
`-`	Προηγούμενο παράθυρο.
`f FILE`	Εναλλαγή προβολής στο FILE.
`/PAT/`, `?PAT?`	Αναζήτηση προς τα εμπρός / προς τα πίσω.
`s`	Step μέσα.
`n`	Step πάνω από.
`<Enter>`	Επανάληψη τελευταίου `s` / `n`.
`r`	Επιστροφή από την τρέχουσα sub.
`c`, `c LINE`, `c SUB`	Συνέχεια / breakpoint μίας χρήσης.
`p EXPR`	Print (βαθμωτό περιβάλλον).
`x EXPR`	Dump (δομημένο, περάστε αναφορά).
`T`	Backtrace στοίβας.
`R`	Επανεκκίνηση μέσω `exec()`.
`rerun [N|-N]`	Επανεκκίνηση με αναπαραγωγή ιστορικού.

Μάθετε περισσότερα

- *breakpoints* - `b`, υπό συνθήκη, ενέργειες, watches, πίνακας χρονισμού.
- *inspecting-state* - `V`, `X`, `y`, `m`, `M`, `i`, χειριστήρια βάθους.
- *tracing* - `t`, `AutoTrace`, έξοδος πληροφορίας γραμμής σε αρχείο.

Breakpoints, ενέργειες, και watches

Αυτό το κεφάλαιο καλύπτει τη διακοπή του αποσφαλματωτή σε συγκεκριμένο σημείο - μια γραμμή, μια υπορουτίνα, μια συνθήκη, μια αλλαγή τιμής - και τη μηχανική των hook προτροπής που σας επιτρέπει να τρέχετε αυτόματα κώδικα σε κάθε διακοπή.

Οι αναγνώστες που φτάνουν εδώ ξέρουν να εκκινούν το `perl -d` και να κάνουν βηματική εκτέλεση γραμμή προς γραμμή. Η επόμενη δεξιότητα είναι να μεταβαίνετε απευθείας στο ενδιαφέρον σημείο και να σταματάτε μόνο όταν το πρόβλημα πρόκειται να εκδηλωθεί.

Breakpoints σε γραμμές και σε sub

Εντολή	Αποτέλεσμα
`b`	Breakpoint στην τρέχουσα γραμμή.
`b LINE`	Breakpoint στη γραμμή LINE στο τρέχον αρχείο.
`b LINE COND`	Υπό συνθήκη· διακοπή όταν η COND είναι αληθής.
`b FILE:LINE`	Μεταξύ αρχείων· το FILE ταιριάζει με εγγραφή του `%INC`.
`b FILE:LINE COND`	Υπό συνθήκη μεταξύ αρχείων.
`b SUB`	Διακοπή στην πρώτη γραμμή της SUB (η SUB πρέπει να είναι μεταγλωττισμένη).
`b SUB COND`	Breakpoint sub υπό συνθήκη.
`b $coderef`	Διακοπή στη sub στην οποία αναφέρεται το `$coderef` (δεν υποστηρίζεται COND).
`b compile SUB`	Διακοπή αφού τελειώσει η μεταγλώττιση της SUB, πριν τρέξει το σώμα της.
`b postpone SUB`	Διακοπή την πρώτη φορά που καλείται η SUB, ακόμη και αν δεν έχει φορτωθεί.
`b load FILE`	Διακοπή όταν το FILE γίνεται `require` / do.
`B LINE`	Διαγραφή breakpoint στη γραμμή LINE.
`B *`	Διαγραφή όλων των breakpoints.
`disable [FILE:]LINE`	Απενεργοποίηση χωρίς αφαίρεση.
`enable [FILE:]LINE`	Επανενεργοποίηση.
`L`	Λίστα από breakpoints, ενέργειες, και watches.
`L b`	Λίστα μόνο από breakpoints.

Οι συνθήκες είναι **σκέτες εκφράσεις**, όχι `if (…)`:

```
  DB<1> b 237 $count > 10
  DB<2> b 42 /unauthorized/i
  DB<3> b validate_payload ref($obj) eq 'HASH' && keys %$obj
```

Τα breakpoints τοποθετούνται μόνο σε γραμμές που περιέχουν εκτελέσιμο κώδικα. Μια κενή γραμμή, ένα καθαρό σχόλιο, ή η συνέχεια προηγούμενης δήλωσης απορρίπτεται:

```
  DB<1> b 3
Line 3 not breakable.
```

Μια γραμμή φέρει το πολύ ένα breakpoint και μία ενέργεια· η προσθήκη δεύτερου επανεγγράφει το πρώτο.

Πίνακας χρονισμού

Πού στον κύκλο ζωής του προγράμματος πυροδοτείται ένα breakpoint έχει σημασία - ειδικά για κώδικα σε μπλοκ `BEGIN` ή αρθρώματα που γίνονται `require` τεμπέλικα.

Στόχος	Εντολή
Διακοπή στην πρώτη δήλωση χρόνου εκτέλεσης	Προεπιλογή - ο αποσφαλματωτής ήδη σταματάει εκεί.
Διακοπή μέσα σε μπλοκ `BEGIN { }`	Εισάγετε `$DB::single = 1;` μέσα στο μπλοκ.
Διακοπή πριν τρέξει το σώμα μιας sub (μετά τη μεταγλώττισή της)	`b compile SUB`
Διακοπή στην πρώτη κλήση μιας sub που δεν έχει ακόμη φορτωθεί	`b postpone SUB`
Διακοπή όταν ένα αρχείο γίνεται `require` ή do	`b load /full/path/from/%INC`
Διακοπή σε συγκεκριμένη γραμμή	`b LINE`
Διακοπή σε γραμμή μόνο όταν ισχύει μια συνθήκη	`b LINE COND`

Το `$DB::single = 1;` μέσα σε μπλοκ `BEGIN` είναι το κόλπο για αποσφαλμάτωση κατά τη μεταγλώττιση: η εγγραφή `b` στην προτροπή λειτουργεί μόνο αφού τρέχει ο αποσφαλματωτής, αλλά ο κώδικας `BEGIN` τρέχει πριν από την πρώτη προτροπή. Ρίξτε την ανάθεση μέσα στο μπλοκ, εκκινήστε υπό `-d`, και ο αποσφαλματωτής σταματάει στην επόμενη γραμμή στην οποία μπορεί να σπάσει η εκτέλεση μέσα στο μπλοκ.

Η πρόσβαση στο `$DB::single` όταν δεν τρέχετε υπό `-d` αυτο-δημιουργεί το `DB::` αβλαβώς - το μοτίβο είναι ασφαλές να γίνει commit.

Παράθεση και διαχείριση 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 *
```

Τα `disable` / `enable` εναλλάσσουν ένα breakpoint χωρίς να ξεχνούν τη συνθήκη του - χρήσιμο όταν θέλετε να παρακάμψετε προσωρινά ένα «καυτό» breakpoint και να το επανοπλίσετε αργότερα.

Ενέργειες: αυτόματη εκτέλεση κώδικα σε μια γραμμή

Μια **ενέργεια** τρέχει κώδικα Perl πριν εκτελεστεί η καθορισμένη γραμμή, χωρίς να σταματάει. Χρησιμοποιήστε την για παθητική παρατήρηση:

```
  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'
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
  DB<3> L a
app.pl:
 42: ...
       action: print "entering: user_id=$user_id\n"
  DB<4> A 42              # delete action at line 42
  DB<5> A *               # delete all actions
```

Διάταξη σε γραμμή στην οποία μπορεί να σπάσει η εκτέλεση:

1. Ελέγχεται η συνθήκη του breakpoint.
2. Εκτυπώνεται η έξοδος ίχνους (αν η ιχνηλάτηση είναι ενεργή).
3. Τρέχει η ενέργεια.
4. Εμφανίζεται η προτροπή (αν χτυπήθηκε breakpoint ή κάνετε βηματική εκτέλεση).
5. Εκτελείται η ίδια η γραμμή.

Ενέργειες και breakpoints συνυπάρχουν στην ίδια γραμμή - το πολύ ένα από κάθε είδος.

Watches: διακοπή σε αλλαγή τιμής

Ένα watch αναφέρει την παλιά και τη νέα τιμή όταν μια παρακολουθούμενη έκφραση αλλάζει, και σας οδηγεί στην προτροπή:

```
  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
```

Τα watches αποτιμώνται σε κάθε γραμμή στην οποία μπορεί να σπάσει η εκτέλεση. Οι εκφράσεις πρέπει να αναφέρονται σε καθολικές ή προσβάσιμα ψευδώνυμα· οι λεξιλογικές μεταβλητές my δεν παρακολουθούνται απευθείας. Όταν μια λεξιλογική έχει σημασία, χρησιμοποιήστε το `Tie::Watch` στον κώδικα για να εγκαταστήσετε watcher στη στιγμή του tie, ή αναθέστε μέσω ψευδωνύμου εμβέλειας πακέτου.

Τα watches προσθέτουν κόστος ανά δήλωση ανάλογο του πλήθους των watches και του κόστους κάθε έκφρασης. Κρατήστε τα στενά και αφαιρέστε τα όταν τελειώσετε.

Hooks προτροπής: εκτέλεση κώδικα σε κάθε διακοπή

Τα < / > εγκαθιστούν κώδικα Perl που τρέχει αυτόματα πριν / μετά από κάθε προτροπή. Τα { / {{ κάνουν το ίδιο για εντολές αποσφαλματωτή:

Εντολή	Αποτέλεσμα
`< CMD`	Κώδικας Perl πριν από κάθε προτροπή (= μετά τη γραμμή που μόλις εκτελέστηκε).
`<< CMD`	Προσάρτηση στη λίστα προ-προτροπής.
`< ?`	Λίστα κώδικα προ-προτροπής.
`< *`	Διαγραφή όλου του κώδικα προ-προτροπής.
`> CMD` / `>> CMD`	Κώδικας Perl μετά από κάθε προτροπή (= πριν από την επόμενη γραμμή).
`{ DBCMD`	Εντολή αποσφαλματωτή (όχι Perl) προ-προτροπής.
`{{ DBCMD`	Προσάρτηση στη λίστα εντολών αποσφαλματωτή προ-προτροπής.

Τυπική χρήση: χρονική σφραγίδα σε κάθε βήμα:

```
  DB<1> < print "stop: ", scalar localtime, "\n"
  DB<2> > print "start: ", scalar localtime, "\n"
```

Ή πάντα να εμφανίζεται η τρέχουσα γραμμή και μια παρακολουθούμενη μεταβλητή:

```
  DB<1> { v
  DB<2> { p $important
```

Πολλαπλών γραμμών: διαφύγετε το newline με `\`:

```
  DB<1> < for my $f (@important_files) {    \
cont:     print "  $f: ", (-s $f) // 'n/a', "\n"; \
cont: }
```

Ψευδώνυμα

Μόνιμες συντομεύσεις εντολών μέσω =:

```
  DB<1> = len s/^len(.*)/p length($1)/
  DB<2> = quit s/^quit(\s*)/exit/
  DB<3> =                      # list all aliases
```

Τα ψευδώνυμα ζουν στο `%DB::alias` και εφαρμόζονται στη στιγμή της αποστολής. Ορίστε τα μία φορά στο `.perldb` για επιμονή πέρα από τη συνεδρία.

Breakpoints μίας χρήσης

Το `c LINE` ορίζει breakpoint στη γραμμή LINE, συνεχίζει, και καθαρίζει το breakpoint όταν χτυπήσει:

```
  DB<1> c 42
```

Ο απλούστερος τρόπος για να πείτε «απλώς τρέξε ως τη γραμμή 42 και σταμάτα εκεί». Χωρίς χειροκίνητη ζευγαροποίηση `b` / `B`.

Διακοπή στην εισαγωγή / στην πρώτη κλήση τεμπέλικα

Όταν μια sub ζει σε άρθρωμα που μπορεί να μην έχει ακόμη φορτωθεί, το `b SUB` αποτυγχάνει:

```
  DB<1> b MyApp::Worker::dispatch
Subroutine not yet compiled.
```

Το `b postpone` περιμένει να τελειώσει η μεταγλώττιση της sub:

```
  DB<1> b postpone MyApp::Worker::dispatch
  DB<2> c
```

Για αρχεία που φορτώνονται με `require` / `do`, το `b load` διακόπτει στην πρώτη φόρτωση:

```
  DB<1> b load /usr/share/perl5/MyApp/Worker.pm
```

Το όνομα αρχείου πρέπει να ταιριάζει με την πλήρη διαδρομή στο `%INC`, όχι μόνο με το όνομα του αρθρώματος.

Μάθετε περισσότερα

- *inspecting-state* - τι να κοιτάξετε όταν πυροδοτηθεί ένα breakpoint.
- *tracing* - όταν θέλετε να παρακολουθήσετε τι συμβαίνει *χωρίς* διακοπή.

Επιθεώρηση κατάστασης

Αυτό το κεφάλαιο καλύπτει το λεξιλόγιο επιθεώρησης δεδομένων του αποσφαλματωτή: `x`, `V`, `X`, `y`, `m`, `M`, `i`, και τις επιλογές που ελέγχουν πόσο βαθιά γίνεται dump συγκεντρωτικών τιμών. Μετά από αυτό το κεφάλαιο μπορείτε να ρωτήσετε οποιοδήποτε εκτελούμενο πρόγραμμα «πώς μοιάζει στην πραγματικότητα αυτή η μεταβλητή, αυτό το πακέτο, αυτό το αντικείμενο αυτή τη στιγμή;»

Οι αναγνώστες φτάνουν εδώ γνωρίζοντας ότι το `p EXPR` εκτυπώνει ένα βαθμωτό. Οι υπόλοιπες εντολές επιθεώρησης κερδίζουν τη χρήση τους τη στιγμή που η απάντηση στο «τι είναι αυτό;» απαιτεί πάνω από μία τιμή.

x: dump δομών

Το `x EXPR` εκτυπώνει όμορφα σε περιβάλλον λίστας, αναδρομικά μέσα σε αναφορές:

```
  DB<1> x \%config
0  HASH(0x5651f9a82358)
   'debug' => 1
   'dsn' => 'dbi:SQLite::memory:'
   'timeout' => 30
   'users' => ARRAY(0x5651f9b28a10)
      0  'alice'
      1  'bob'
      2  'carol'
```

Πάντα περάστε **αναφορά** για συγκεντρωτικά. Το x %h κάνει dump την επίπεδη λίστα κλειδιού/τιμής, που κρύβει τη δομή. Το x \%h κάνει αυτό που θέλετε.

```
  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'
```

Για βαθιές ή κυκλικές δομές, περιορίστε την αναδρομή:

```
  DB<4> x 2 \%tree      # recurse 2 levels deep
  DB<5> x 1 $obj        # one level
```

Αν το πρώτο διακριτικό της έκφρασης είναι αριθμητικό, βάλτε σε παρενθέσεις:

```
  DB<6> x (3 + $offset)     # without parens, 3 is mis-parsed as␣
↪depth
```

V και X: μεταβλητές πακέτου

Το V [PKG [VARS]] κάνει dump τις καθολικές ενός πακέτου:

```
  DB<1> V main
$counter = 42
@ARGV = (
    0  '--verbose'
    1  'input.txt'
)
%ENV = (
    'PATH' => '/usr/bin:/bin'
    ...
)
```

Με δεύτερο όρισμα, φιλτράρει μεταβλητές:

```
  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␣
↪/^_/
```

Το X είναι το V για το τρέχον πακέτο:

```
  DB<5> X                 # dump current package's globals
  DB<6> X $foo @bar       # filtered
```

Τα V / X **δεν** βλέπουν λεξιλογικές (μεταβλητές my). Για αυτές, χρησιμοποιήστε το y.

y: λεξιλογικές στην περικλείουσα εμβέλεια

Το `y LEVEL [VARS]` διαβάζει λεξιλογικές μεταβλητές my μέσω του 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
```

Το y απαιτεί PadWalker έκδοση 0.08 ή νεότερη. Λειτουργεί μόνο μέσα σε sub (το εξωτερικό σώμα του σεναρίου δεν έχει pad για να διασχιστεί).

m: μέθοδοι αντικειμένου ή κλάσης

Το `m EXPR` παραθέτει μεθόδους που μπορούν να κληθούν σε ένα αντικείμενο ή κλάση:

```
  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
```

Ομαδοποιεί τις μεθόδους ανά το πακέτο που τις παρέχει - χρήσιμο για να εντοπίσετε από ποιον ρόλο ή υπερκλάση προέρχεται μια μέθοδος.

M και i: αρθρώματα και κληρονομικότητα

Το M παραθέτει τα φορτωμένα αρθρώματα με τις εκδόσεις τους:

```
  DB<1> M
  'DBI' (version 1.643)
  'JSON::PP' (version 4.16)
  'MyApp::Model' (version undef)
  ...
```

Το `i CLASS` ή το `i $obj` εκτυπώνει το δέντρο κληρονομικότητας:

```
  DB<2> i $user
MyApp::User 0.12
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
   MyApp::Model 0.42
      Moo::Object ?
```

Χρήσιμο για το «από πού προέρχεται στην πραγματικότητα αυτή η μέθοδος;» όταν η επίλυση μεθόδου διασχίζει πολλούς ρόλους.

S: ονόματα υπορουτινών

Το `S [[!]PAT]` παραθέτει ονόματα sub που ταιριάζουν (ή δεν ταιριάζουν) με ένα μοτίβο:

```
  DB<1> S ^MyApp::Worker::
MyApp::Worker::cancel
MyApp::Worker::dispatch
MyApp::Worker::retry

  DB<2> S !^(main|DB|UNIVERSAL)::      # everything except these
```

Το S σκέτο τα παραθέτει όλα - μακρύ.

p έναντι x: ποιο να χρησιμοποιείτε πότε

Θέλετε	Χρησιμοποιήστε
Την τιμή ενός μόνο βαθμωτού, σκέτη	`p $x`
Μια συνενωμένη / μορφοποιημένη έκφραση	`p join ',', sort keys %h`
Κατακερματισμό ή πίνακα	`x \%h, x \@a`
Τα πεδία ενός αντικειμένου	`x $obj`
Μόνο το ανώτατο επίπεδο	`x 1 $deep_tree`
Μεταβλητές πακέτου του Pkg	`V Pkg`
Λεξιλογικές της τρέχουσας sub	`y 0`
Λίστα μεθόδων του $obj	`m $obj`
Αλυσίδα @ISA του $obj	`i $obj`

Ειδική περίπτωση: το `p` είναι επίπεδο print. Για αναφορά, το `p $ref` εκτυπώνει `HASH(0x…)`. Χρησιμοποιήστε `x $ref` για τα περιεχόμενα.

Επιλογές βάθους και συμπαγότητας

Η έξοδος dump διαμορφώνεται από επιλογές που ορίζονται μέσω της εντολής o. Οι σχετικές ρυθμίσεις για επιθεώρηση:

Επιλογή	Αποτέλεσμα
`dumpDepth`	Βάθος αναδρομής (αρνητικό = απεριόριστο). Προεπιλογή 4.
`arrayDepth`	Μόνο τα πρώτα N στοιχεία πίνακα· κενό = όλα.
`hashDepth`	Το ίδιο, για κατακερματισμούς.
`compactDump`	Σύντομοι πίνακες σε μία γραμμή.
`veryCompact`	Ακόμη πιο σφιχτό.
`globPrint`	Dump περιεχομένων glob.
`DumpReused`	Πλήρης ανάπτυξη επαναλαμβανόμενων αναφορών (προσοχή σε κύκλους).
`quote`	`"` ή `'` ή `auto` (προεπιλογή).
`undefPrint`	Πώς αποδίδεται το `undef`.
`bareStringify`	Εκτύπωση της overload-μετατραμμένης σε συμβολοσειρά μορφής.

Ορισμός επιλογής μέσα σε συνεδρία:

```
  DB<1> o dumpDepth=6
  DB<2> o compactDump=1
  DB<3> o arrayDepth=20
  DB<4> o compactDump?            # query current value
```

Ορισμός επιλογών στην εκκίνηση μέσω του `PERLDB_OPTS` ή στο `.perldb`:

```
PERLDB_OPTS="dumpDepth=6 compactDump=1" perl -d script.pl
```

Επιθεώρηση εσωτερικών SV

Το `Devel::Peek` εκθέτει την εσωτερική αποθήκευση του βαθμωτού - σημαίες τύπου, refcount, διάταξη PV/NV/IV, αλυσίδα magic - όταν η ερώτηση είναι «τι είδους βαθμωτό είναι πραγματικά αυτό;»:

```
  DB<1> use Devel::Peek; Dump($sv)
```

Κυρίως για τη διερεύνηση tied μεταβλητών, overloaded αντικειμένων, ή ανωμαλιών απόδοσης όπου η απάντηση εξαρτάται από το αν ένα βαθμωτό αποθηκεύεται αυτή τη στιγμή ως ακέραιος, συμβολοσειρά, ή και τα δύο.

Σημείωση: το `Devel::Peek` στην pperl είναι στρώμα συμβατότητας - αναφέρει μια ειδική για την pperl διάταξη Sv αντί της διάταξης SV της perl5. Πεδία όπως `REFCNT`, `FLAGS`, `PV`, `IV`, `NV` έχουν ισοδύναμα· η ακριβής έξοδος διαφέρει. Δείτε τον *pperl-architecture guide* για λεπτομέρειες διάταξης.

Αποτίμηση στην προτροπή

Ό,τι δεν αναγνωρίζεται ως εντολή αποτιμάται ως Perl. Χρησιμοποιήστε το για να εμβαθύνετε πέρα από μία μόνο εντολή:

```
  DB<1> p keys %{ $obj->{prefs} }
theme,locale,tz

  DB<2> x [ grep { !defined $_->{parent} } @nodes ]
  DB<3> x { map { $_->id => $_->name } @users }
```

Για εφάπαξ διεκδικήσεις χωρίς διακοπή:

```
  DB<1> die "unexpected" if ref $obj ne 'MyApp::User'
```

I/O προγράμματος έναντι I/O αποσφαλματωτή

Η είσοδος και έξοδος του ίδιου του αποσφαλματωτή περνούν μέσω των `$DB::IN` και `$DB::OUT`, που ανοίγουν απευθείας στο `/dev/tty`. Το πρόγραμμα μπορεί να ανακατευθύνει ελεύθερα STDIN / STDOUT / STDERR· η προτροπή του αποσφαλματωτή εμφανίζεται κανονικά. Αντίστροφα, η έξοδος του `p EXPR` δεν μολύνει το δεσμευμένο stdout του προγράμματος.

Μάθετε περισσότερα

- *breakpoints* - σταματήστε εκεί που μπορείτε να επιθεωρήσετε· αυτό το κεφάλαιο αφορά το τι να κοιτάξετε αφού σταματήσετε.
- *tracing* - παρακολουθήστε τιμές να αλλάζουν με τον χρόνο χωρίς να σταματάτε σε κάθε βήμα.

Ιχνηλάτηση: παρακολούθηση της ροής εκτέλεσης

Αυτό το κεφάλαιο καλύπτει ίχνη που καταγράφουν τι έκανε ένα πρόγραμμα χωρίς να το σταματούν: η εντολή `t` του αποσφαλματωτή και οι επιλογές ιχνηλάτησης, η οικογένεια αρθρωμάτων `Devel::Trace`, και ο ιχνηλάτης της μηχανής regex. Χρησιμοποιήστε ιχνηλάτηση όταν το σφάλμα είναι «ποια διαδρομή κώδικα έτρεξε στην πραγματικότητα;» αντί για «τι τιμή έχει το X στη γραμμή N;».

Οι αναγνώστες που έρχονται γνωρίζουν πώς να ορίζουν breakpoints και να επιθεωρούν κατάσταση. Ένα ίχνος είναι το συμπλήρωμα - «ακολούθα κάθε βήμα και πες μου γι” αυτό» αντί για «σταμάτα σε ένα ενδιαφέρον σημείο».

t: ιχνηλάτηση μέσα στον αποσφαλματωτή

Η `t` εναλλάσσει την ιχνηλάτηση ανά δήλωση. Κάθε γραμμή στην οποία μπορεί να σπάσει η εκτέλεση εκτυπώνει έναν δείκτη θέσης πριν εκτελεστεί:

```
  DB<1> t
Trace = on
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
  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;
...
```

Περιορισμός του βάθους:

```
  DB<3> t 2           # toggle with maximum sub-depth 2
  DB<4> t 0 my_work() # trace one expression only
```

Η επιλογή `frame` ελέγχει τι εκτυπώνει το ίχνος σε κλήσεις και επιστροφές sub:

Bit	Σημαία	Αποτέλεσμα
1	enter	Εκτύπωση κατά την είσοδο σε sub.
2	exit	Εκτύπωση κατά την έξοδο από sub.
4	ctx+args	Συμπερίληψη περιβάλλοντος κλήσης και ορισμάτων.
8	tied/ref	Σήμανση κλήσεων sub μέσω tied μεταβλητής και αναφοράς.
16	return	Εκτύπωση της τιμής επιστροφής.

Ορίστε το `frame` μέσω της `o`:

```
  DB<5> o frame=15       # enter + exit + ctx/args + tied/ref
```

Συνηθισμένοι συνδυασμοί:

- `frame=2`: μόνο ίχνη εισόδου/εξόδου, μία γραμμή ανά κλήση - η όψη δέντρου κλήσεων.
- `frame=6`: προσθήκη περιβάλλοντος καλούντος και dump ορισμάτων - πλήρες ίχνος.
- `frame=30`: σαν `frame=6` συν τις τιμές επιστροφής.

Η έξοδος ίχνους πηγαίνει στη ροή εξόδου του αποσφαλματωτή. Ανακατευθύνετέ την σε αρχείο ώστε να μην πνίγει τη διαδραστική συνεδρία:

```
  DB<6> o LineInfo=trace.out
```

Διαβάστε το αρχείο σε άλλο τερματικό:

```
tail -f trace.out
```

Μη διαδραστική ιχνηλάτηση: NonStop + AutoTrace

Για «καταγραφή πλήρους ίχνους εκτέλεσης χωρίς αλληλεπίδραση», συνδυάστε `NonStop=1` (χωρίς προτροπές) και `AutoTrace=1` (ιχνηλάτηση ενεργή από την αρχή):

```
PERLDB_OPTS="NonStop=1 AutoTrace=1 frame=2 LineInfo=trace.out" \
  perl -d script.pl
```

Το πρόγραμμα τρέχει χωρίς διακοπές· το `trace.out` συλλαμβάνει κάθε γραμμή στην οποία μπορεί να σπάσει η εκτέλεση και κάθε είσοδο σε sub. Ιδανικό για σύλληψη αποτυχίας σε λειτουργία CI, διακομιστές χωρίς οθόνη, ή κάθε περιβάλλον όπου προτροπή TTY δεν είναι διαθέσιμη.

Προσθέστε `inhibit_exit=0` για να αφήσετε το πρόγραμμα να φτάσει στο τέλος χωρίς την προτροπή «Debugged program terminated».

`Devel::Trace`: κάθε γραμμή, χωρίς τον αποσφαλματωτή

Το `Devel::Trace` ιχνηλατεί κάθε γραμμή κάθε αρχείου στο STDERR. Καμία αλληλεπίδραση γραμμής εντολών. Κανένα breakpoint. Απλό και θορυβώδες:

```
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";
```

Τρέξτε το μία φορά, κάντε grep στην έξοδο για τον γρίφο. Το μέγεθος της εξόδου είναι ανάλογο της διάρκειας εκτέλεσης του προγράμματος - χρησιμοποιήστε σε ελάχιστο αναπαραγωγέα, όχι σε ολόκληρη τη σουίτα δοκιμών.

Παραλλαγές:

- `Devel::DumpTrace` - το ίδιο συν τιμές μεταβλητών ανά γραμμή.
- `Devel::TraceUse` - «ποια αρθρώματα φορτώθηκαν, με ποια σειρά, από ποιον»:

  ```
  perl -d:TraceUse script.pl
  ```

- `Devel::TraceINC` - στενότερο: μόνο η αλυσίδα `require` με χρονομέτρηση.
- `Devel::Trace::Subs` - γεννήτρια ροής κλήσεων και ίχνους στοίβας για «ποιες sub έτρεξαν στην πραγματικότητα».

Ιχνηλάτηση συγκεκριμένης περιοχής

Η μεταβλητή `$DB::trace` εναλλάσσει την ιχνηλάτηση από κώδικα Perl:

```
$DB::trace = 1;
interesting_region();
$DB::trace = 0;
```

Η ανάθεση στο `$DB::trace` όταν δεν τρέχετε υπό `-d` αυτο-δημιουργεί τη μεταβλητή αβλαβώς - ο κώδικας είναι ασφαλές να γίνει commit και τρέχει ως no-op εκτός του αποσφαλματωτή.

Ίδιο μοτίβο για βηματική εκτέλεση μιας περιοχής:

```
$DB::single = 1;        # break here when next statement runs under
↪-d
```

Ιχνηλάτηση regex: `use re 'debug'`

Η μηχανή regex έχει δικό της ιχνηλάτη. Ενεργοποιήστε τον λεξιλογικά μέσα σε περιοχή, ή στη γραμμή εντολών:

```
{
    use re 'debug';
    $str =~ /(\w+)\s+(\w+)/;
}
```

```
perl -Mre=debug -e '"hello world" =~ /(\w+)\s+(\w+)/'
```

Η έξοδος της φάσης μεταγλώττισης δείχνει τι έκανε ο βελτιστοποιητής:

- `Compiling REx 'PATTERN'` - το αρχικό μοτίβο.
- `size N` - μέγεθος εσωτερικού πίνακα κόμβων.
- `anchored 'SUB' at K / floating 'SUB' at K..LIMIT` - ποια υποσυμβολοσειρά θα ψάξει η μηχανή ως προφίλτρο.
- `stclass TYPE` - η κλάση χαρακτήρα εκκίνησης.
- `minlen N` - καμία αντιστοιχία δεν είναι δυνατή σε είσοδο μικρότερη από N.
- Λίστα κόμβων - το μεταγλωττισμένο πρόγραμμα.

Η έξοδος χρόνου εκτέλεσης δείχνει κάθε βήμα:

```
  POS <PRE-MATCH> <POST-MATCH>   |  NODE_NUM: OPNAME
```

Η εσοχή αντικατοπτρίζει το βάθος της οπισθοχώρησης· τα `failed...` και `failed, try continuation...` σημαίνουν οπισθοχωρήσεις· τα `Match successful!` / `Match failed.` κλείνουν το ίχνος.

Αν δεν εμφανιστεί καθόλου έξοδος χρόνου εκτέλεσης, ο βελτιστοποιητής αποφάσισε την αντιστοιχία χωρίς είσοδο στη μηχανή - είτε το `minlen` έκανε προφιλτράρισμα είτε το μοτίβο πτυχώθηκε σε έλεγχο σταθεράς.

Η μηχανή regex της pperl είναι μια ντόπια εκ νέου υλοποίηση, όχι μεταφορά αυτής της perl5. Το ίχνος είναι λειτουργικά ισοδύναμο αλλά τα ακριβή ονόματα κόμβων και οι μορφές πεδίων δεν είναι byte-πανομοιότυπες με αυτές του upstream. Η προγραμματιστική ανάλυση του ίχνους (λίγα εργαλεία το κάνουν) θα χρειαστεί προσαρμογή.

Μερικοί διακόπτες στενεύουν την έξοδο:

```
use re 'debugcolor';                    # ANSI colour
use re debug => 'EXECUTE';              # runtime only, no compile
↪output
```

`Devel::TraceUse` για σφάλματα φόρτωσης αρθρωμάτων

Ένα πρόγραμμα που τρέχει αργά στην εκκίνηση, ή ένα «γιατί φορτώνεται καν αυτό το άρθρωμα;», είναι σχεδόν πάντα ζήτημα γράφου αρθρωμάτων. Το `Devel::TraceUse` εκτυπώνει το πλήρες δέντρο `use`/`require`:

```
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]
   ...
```

Η εσοχή δείχνει την αλυσίδα εξαρτήσεων. Αντιπαραβάλετε με τον αριθμό γραμμής στο γονικό αρχείο για να δείτε πού συμβαίνει κάθε φόρτωση.

Συνδυασμός ιχνών με τον αποσφαλματωτή

Ιχνηλάτηση σε αρχείο, βηματική εκτέλεση διαδραστικά:

```
PERLDB_OPTS="frame=6 LineInfo=trace.out" perl -d app.pl
```

Η προτροπή του αποσφαλματωτή εμφανίζεται κανονικά· το `trace.out` συσσωρεύει το δέντρο κλήσεων καθώς κάνετε βηματική εκτέλεση και συνεχίζετε. Χρήσιμο όταν μια αλυσίδα κλήσεων είναι πολύ βαθιά για να ακολουθηθεί βηματικά αλλά εξακολουθείτε να θέλετε να σταματάτε σε συγκεκριμένα σημεία για επιθεώρηση.

Μάθετε περισσότερα

- *breakpoints* - διακοπή σε συγκεκριμένο σημείο αντί καταγραφής κάθε βήματος.
- *inspecting-state* - οι εντολές που απαντούν στο «τι είναι αυτή η τιμή;».

Ενσωμάτωση με IDE και DAP

Αυτό το κεφάλαιο καλύπτει την αποσφαλμάτωση με οδηγό τον editor: `Perl::LanguageServer` στο VS Code και τα αντίστοιχα σε Neovim / Emacs / IntelliJ, απομακρυσμένη αποσφαλμάτωση μέσω SSH, και αποσφαλμάτωση σε λειτουργία container μέσω `kubectl` ή `docker`.

Όσοι αναγνώστες καταφεύγουν σε IDE αντί σε προτροπή τερματικού θα βρουν εδώ την ελάχιστη βιώσιμη εγκατάσταση, τα κλειδιά ρυθμίσεων που έχουν σημασία, και τα γνωστά σημεία ευθραυστότητας (Coro, αντιστοίχιση διαδρομών, attach έναντι launch).

Το τοπίο

Δύο πρωτόκολλα επί της γραμμής, και τα δύο με backends σε Perl:

Πρωτόκολλο	Backend Perl	Κατάσταση
DAP	`Perl::LanguageServer`	Ενεργό.
DBGp	`Devel::Debug::DBGp`	Στάσιμο από το 2018.

Για νέες εγκαταστάσεις, χρησιμοποιήστε DAP μέσω του `Perl::LanguageServer`. Το DBGp επιβιώνει μόνο εκεί όπου ένας υπάρχων πελάτης (vim vdebug, pugdebug) είναι

ήδη σε χρήση.

Το `Perl::LanguageServer` συγκεντρώνει έναν LSP (νοημοσύνη κώδικα, διαγνωστικά, μετάβαση σε ορισμό) και έναν προσαρμογέα DAP (αποσφαλματωτή) σε μία διεργασία. Το μισό LSP λειτουργεί σε οποιοδήποτε αρχείο Perl στον editor· το μισό DAP τρέχει μόνο κατά τη διάρκεια συνεδρίας αποσφαλμάτωσης.

Ελάχιστη εγκατάσταση VS Code

Πλευρά CPAN - εγκαταστήστε στην Perl υπό την οποία θα τρέξει το debuggee:

```
cpanm Perl::LanguageServer
```

Πλευρά editor - εγκαταστήστε την επέκταση `richterger.perl`:

```
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
    }
  ]
}
```

Ανοίξτε ένα `.pl` ή `.pm`, ορίστε breakpoints στο περιθώριο, πατήστε F5. Η Debug Console δέχεται οποιαδήποτε έκφραση Perl, που αποτιμάται στο τρέχον πλαίσιο στοίβας.

`.vscode/settings.json` για το `@INC` και την επιλογή διερμηνέα:

```
{
  "perl.perlInc": [
    "${workspaceFolder}/lib",
    "${workspaceFolder}/local/lib/perl5"
  ],
  "perl.perlCmd": "/usr/bin/perl"
}
```

Δυνατότητες

Το `Perl::LanguageServer` υποστηρίζει τα χαρακτηριστικά DAP που έχουν σημασία:

- Launch, pause, step in / over / out, return.
- Breakpoints υπό συνθήκη και χωρίς συνθήκη, προστιθέμενα ανά πάσα στιγμή.
- Επιθεώρηση μεταβλητών σε όλα τα πλαίσια στοίβας.
- **Set variable** - αλλαγή τιμής στη μέση της συνεδρίας από τον editor.
- Εκφράσεις watch.
- Αποτίμηση εντός περιβάλλοντος στη Debug Console.
- Hot-reload αρθρωμάτων μέσω `reloadModules: true`.

Δεν υποστηρίζονται:

- **Σύνδεση σε εκτελούμενη διεργασία.** Το `request: "attach"` δεν είναι υλοποιημένο· μόνο το `"launch"` (ο LS εκκινεί ο ίδιος το debuggee).

Απομακρυσμένη αποσφαλμάτωση μέσω SSH

Επεξεργαστείτε τοπικά· αποσφαλματώστε σε απομακρυσμένο υπολογιστή. `settings.json`:

```
{
  "perl.sshCmd": "ssh",
  "perl.sshAddr": "deploy@host.example",
  "perl.sshUser": "deploy",
  "perl.sshWorkspaceRoot": "/srv/app",
  "perl.pathMap": [
    ["/srv/app", "${workspaceFolder}"]
  ]
}
```

Το `Perl::LanguageServer` τρέχει στον απομακρυσμένο υπολογιστή· ο editor μένει τοπικός. Το `pathMap` μεταφράζει τα ονόματα αρχείων που αναφέρει το debuggee (`/srv/app/lib/X.pm`) σε διαδρομές του τοπικού editor (`${workspaceFolder}/lib/X.pm`).

Breakpoints που σιωπηρά αποτυγχάνουν να συνδεθούν είναι σχεδόν πάντα αναντιστοιχία `pathMap`.

Αποσφαλμάτωση σε λειτουργία container

Το `launch.json` δέχεται κλειδιά container που επικαλούνται τα `docker`, `podman`, `docker-compose`, ή `kubectl`:

Κλειδί	Τιμές
`containerCmd`	`"docker"`, `"docker-compose"`, `"podman"`, `"kubectl"`
`containerMode`	`"run"` (φρέσκο container) ή `"exec"` (υπάρχον)
`containerName`	Image (για `run`) ή όνομα container (για `exec`)
`containerArgs`	Επιπλέον ορίσματα που περνούν στο runtime του container.
`pathMap`	`[["/in/container", "/on/host"], ...]`

Σύνδεση σε ήδη εκτελούμενο container Docker:

```
{
  "type": "perl",
  "request": "launch",
  "name": "Debug in container",
  "program": "/app/bin/worker.pl",
  "containerCmd": "docker",
  "containerMode": "exec",
  "containerName": "my-running-worker",
  "pathMap": [["/app", "${workspaceFolder}"]]
}
```

Pod Kubernetes:

```
{
  "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}"]]
}
```

Περιορισμοί:

- **Το `Perl::LanguageServer` πρέπει να είναι εγκατεστημένο μέσα στο image.** Ενσωματώστε το στο Dockerfile ή χτίστε παραλλαγή dev-image.
- Το `pathMap` είναι η μεγαλύτερη αιτία αποτυχίας. Breakpoints που δεν συνδέονται ποτέ σημαίνουν ότι το debuggee αναφέρει διαδρομή που δεν μεταφράζεται.

Άλλοι editors

- **Neovim** - χρησιμοποιήστε `nvim-dap` με το `Perl::LanguageServer` ως προσαρμογέα. Η προδιαγραφή του προσαρμογέα είναι ένα σύντομο απόσπασμα Lua που δείχνει στο `perl -MPerl::LanguageServer -e '...'`.
- **Emacs** - `dap-mode` με τον ίδιο προσαρμογέα.
- **IntelliJ** - `Devel::Camelcadedb` + το πρόσθετο Perl της JetBrains. Όχι DAP· ιδιόκτητο πρωτόκολλο. Βιώσιμο αλλά εκτός του οικοσυστήματος

`Perl::LanguageServer`.

- **Sublime Text** - υπάρχουν πελάτες DAP μέσω προσθέτων· χρησιμοποιήστε το `Perl::LanguageServer` ως προσαρμογέα.

Εκκινήσεις ειδικής λειτουργίας

Κλειδιά `launch.json` για μη προεπιλεγμένη επίκληση:

Κλειδί	Αποτέλεσμα
`useTaintForDebu`	Εισάγει `-T` στην επίκληση perl του debuggee.
`sudoUser`	Επανεκτελεί το debuggee ως αυτός ο χρήστης (χρειάζεται sudo χωρίς κωδικό).
`reloadModules`	Ενεργοποιεί το hot-reload αρθρωμάτων κατά τη διάρκεια της συνεδρίας.
`stopOnEntry`	Διακοπή στην πρώτη δήλωση.

Παγίδες

- **Σφάλματα σε χρόνο BEGIN** εμφανίζονται ως γεγονότα «output» του DAP, όχι ως γεγονότα διακοπής σε breakpoint. Χρησιμοποιήστε `stopOnEntry: true` για να πετύχετε καν διακοπή αν το σενάριο πεθάνει στο `BEGIN`.
- **Το `$SIG{__DIE__}` της εφαρμογής** μπορεί να καταπιεί εξαιρέσεις πριν τρέξει ο χειριστής του αποσφαλματωτή. Τυλίξτε τον χειριστή της εφαρμογής σε φύλακα `if ($^S)` (δείτε *exceptions*).
- **Οι εκφράσεις watch επανα-αποτιμώνται σε κάθε βήμα.** Αποφύγετε ακριβές εκφράσεις ή εκφράσεις με παρενέργειες.
- **Όχι source maps.** Η Perl δεν έχει ισοδύναμη έννοια με το source map της JavaScript. Φίλτρα πηγαίου (`Smart::Comments`) μπορούν να μετατοπίσουν τους αναφερόμενους αριθμούς γραμμών κατά ένα· breakpoints σε φιλτραρισμένα αρχεία μπορεί να προσγειωθούν σε λάθος γραμμή.
- **Βηματική εκτέλεση `Future::AsyncAwait`.** Λειτουργεί σε απλές περιπτώσεις· η βηματική εκτέλεση σε σύνθετες αλυσίδες `await` είναι αναξιόπιστη. Πέστε σε `perl -d` για βαθιά διερεύνηση async, ή χρησιμοποιήστε καταγραφή.
- **Σε Windows native** - ο χειρισμός stdin του `Perl::LanguageServer` είναι σπασμένος σε Windows. Το WSL δουλεύει.
- **Coro.** Το `Perl::LanguageServer` βασίζεται εσωτερικά στο Coro· το πιο συχνά αναφερόμενο σημείο ευθραυστότητας είναι η κακή αλληλεπίδραση του Coro με ασυνήθιστες ρυθμίσεις build. Αν ο διακομιστής αρνείται να ξεκινήσει, ελέγξτε πρώτα το log εγκατάστασης του Coro.

Μάθετε περισσότερα

- *interactive-debugger* - απευθείας `perl -d`, όταν η στοίβα IDE είναι υπερβολή ή μη διαθέσιμη.

4.7 Ο οριστικός οδηγός μονογραμμικών Perl

Ένα μονόγραμμο Perl είναι ένα πλήρες πρόγραμμα που δίνεται ως όρισμα στο `pperl -e` (ή `-E`) και εκτελείται απευθείας από το κέλυφος. Καταλαμβάνει την ίδια θέση με τα `awk`, `sed` και τις εκ των ενόντων σωληνώσεις κελύφους, με πλουσιότερη μηχανή regex, πλήρη τυπική βιβλιοθήκη και τη δυνατότητα να εξελιχθεί σε πραγματικό σενάριο τη στιγμή που μία γραμμή πάψει να αρκεί.

Ο οδηγός αυτός απευθύνεται σε χρήστη του κελύφους Linux που ήδη συνθέτει σωληνώσεις `grep`/`sed`/`awk`/`xargs` και έχει γράψει ένα-δυο μικρά σενάρια Perl. Διδάσκει την επιφάνεια της γραμμής εντολών από άκρη σε άκρη: τις σημαίες, έναν προοδευτικό κατάλογο συνταγών, μετασχηματισμούς οδηγούμενους από regex, αριθμητική δουλειά, πώς να τυλίξετε συνταγές σε επαναχρησιμοποιήσιμα ψευδώνυμα κελύφους, και τις παγίδες που στοιχίζουν χρόνο.

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 5.44.

4.7.1 Ποιο κεφάλαιο θέλω;

Θέλετε να ...	Ξεκινήστε εδώ
Μάθετε τι κάνει κάθε σημαία (`-n`, `-p`, `-a`, ...)	*switches*
Περιηγηθείτε σε συνταγές από τετριμμένες αντικαταστάσεις έως αναγωγές δεδομένων	*progression*
Ταιριάξτε, εξαγάγετε ή ξαναγράψτε κείμενο με regexps	*regex-recipes*
Άθροισμα, μέσος όρος, παραγωγή πρώτων, μετατροπή IP, αριθμητική ημερομηνιών	*numeric*
Μετατρέψτε ένα μονόγραμμο σε επαναχρησιμοποιήσιμη συνάρτηση ή ψευδώνυμο κελύφους	*aliases*
Αποσφαλματώστε εισαγωγικά, κωδικοποίηση, επιτόπιες επεξεργασίες και εκπλήξεις με τον διαχωριστή εγγραφών	*gotchas*

4.7.2 Τα τρία επίπεδα του οδηγού

- **Προσανατολισμός** - η σελίδα αυτή.
- **Εργαζόμενος προγραμματιστής** - τα έξι κεφάλαια παραπάνω. Καθένα στέκει αυτόνομο· περιδιαβείτε όσα σχετίζονται με το τρέχον πρόβλημά σας.
- **Βαθιά αναφορά** - ο συμπυκνωμένος πίνακας σημαιών/μεταβλητών στο τέλος του *switches*.

4.7.3 Δύο παραδείγματα πριν ξεκινήσετε

Ένας χρήστης sed/awk που διαβάζει αυτόν τον οδηγό θέλει να δει τι του προσφέρει ένα μονόγραμμο. Δύο επιδείξεις.

Άθροιση των αριθμών στην τελευταία στήλη ενός αρχείου χωρισμένου με λευκούς χαρακτήρες

```
pperl -lane '$s += $F[-1]; END { print $s }' data.txt
```

Το `-l` κάνει chomp τους χαρακτήρες αλλαγής γραμμής εισόδου και προσθέτει έναν στην έξοδο. Το `-a` διασπά κάθε γραμμή στο `@F`. Το `-n` τυλίγει το πρόγραμμα σε έναν σιωπηρό βρόχο ανάγνωσης. Το `$F[-1]` είναι η τελευταία στήλη. Το `END { … }` εκτελείται μία φορά αφού εξαντληθεί η είσοδος. Χωρίς προσωρινό αρχείο, χωρίς `awk 'BEGIN{…}END{…}'`, χωρίς συμβατικό πλαίσιο.

Εξαγωγή κάθε μοναδικής διεύθυνσης IPv4 από αρχείο καταγραφής, με τις πιο πρόσφατες πρώτες

```
pperl -lne 'print for /\b(?:\d{1,3}\.){3}\d{1,3}\b/g' access.log \
  | awk '!seen[$0]++'
```

Το μονόγραμμο αναλαμβάνει το δύσκολο μέρος (την εύρεση όλων των αντιστοιχιών ανά γραμμή, όχι μόνο της πρώτης), το awk κρατά την πρώτη εμφάνιση καθεμίας. Γραμμένα ως δύο σύντομα εργαλεία, καθένα που κάνει μία δουλειά - αυτός είναι ο τρόπος εργασίας αυτού του οδηγού.

4.7.4 Διαβάστε με αυτή τη σειρά στην πρώτη ανάγνωση

1. *switches* - το λεξιλόγιο.
2. *progression* - οι συνταγές, ταξινομημένες κατά δυσκολία.
3. *regex-recipes*, *numeric* - εφαρμοσμένα κομμάτια.
4. *aliases* - μεταφορά όσων χρησιμοποιείτε καθημερινά σε συναρτήσεις κελύφους.
5. *gotchas* - οι παγίδες, μόλις τις συναντήσετε.

Σε μεταγενέστερες αναγνώσεις, πηγαίνετε απευθείας στο κεφάλαιο που ονοματίζει το πρόβλημα που έχετε στα χέρια σας.

4.7.5 Βιβλιογραφία

Ο οδηγός αυτός δεν προέρχεται από ένα μόνο βιβλίο. Ερευνήθηκε όπως θα ερευνούσε το θέμα ένας τεχνικός συγγραφέας - perldoc, τεκμηρίωση αρθρωμάτων CPAN, ομιλίες συνεδρίων, αναρτήσεις ιστολογίων, αρχεία λιστών αλληλογραφίας και τα βιβλία που παρατίθενται παρακάτω - και στη συνέχεια αποστάχθηκε σε αρκετά περάσματα: αφαιρώντας αναφορές σε εργαλεία και εκδόσεις που κανείς πια δεν τρέχει, συμπτύσσοντας την ίδια συμβουλή που επαναλαμβάνεται από διαφορετικούς συγγραφείς σε μία διατύπωση, και συγχωνεύοντας πολλαπλές μερικές πραγματεύσεις ενός θέματος σε μία πλήρη πραγμάτευση. Το αποτέλεσμα καλύπτει περισσότερο έδαφος από οποιαδήποτε μεμονωμένη πηγή που συμβουλευτήκαμε, επειδή καμία μεμονωμένη πηγή δεν καλύπτει όλο το πεδίο. Είναι επίσης πυκνότερα διασυνδεδεμένος από οποιαδήποτε από αυτές: όταν το κεφάλαιο των regex αγγίζει κάτι που ο οδηγός regex εξηγεί ήδη σε βάθος, αυτό απέχει ένα κλικ - όχι ένα κεφάλαιο που πρέπει να βρείτε μόνοι σας σε άλλο τόμο.

Τα δύο βιβλία παρακάτω είναι οι πλησιέστεροι δημοσιευμένοι σύντροφοι σε έντυπη μορφή. Αγοράστε τα αν το ιδίωμα των μονογραμμικών είναι καθημερινό εργαλείο για εσάς και θέλετε μια έντυπη αναφορά στο ράφι.

- Peteris Krumins, *Perl One-Liners: 130 Programs That Get Things Done*, No Starch Press, 2014.
- Sundeep Agarwal, *Perl One-Liners Guide* (learnbyexample), 2023.

Οι σημαίες

Κάθε μονόγραμμο είναι ένας συνδυασμός ενός προγράμματος (που δίνεται στο `-e` ή στο `-E`) και μιας χούφτας σημαιών που τυλίγουν γύρω του σιωπηρή συμπεριφορά. Το κεφάλαιο αυτό καλύπτει τις σημαίες που θα χρειαστείτε. Καθεμία κερδίζει τη θέση της στα κεφάλαια προοδευτικότητας, regex και αριθμητικής που ακολουθούν.

Οι αναγνώστες που έρχονται από `awk` ή `sed` ξέρουν ήδη πώς μοιάζει η επανάληψη ανά γραμμή, η διάσπαση πεδίων και η επιτόπια επεξεργασία· το ερώτημα είναι ποια σημαία του pperl ενεργοποιεί την καθεμία.

`-e` - εκτέλεση προγράμματος που δίνεται στη γραμμή εντολών

```
pperl -e 'print "hello\n"'
```

Το `-e` δέχεται ως όρισμα τον πηγαίο κώδικα Perl. Πολλαπλά κομμάτια `-e` συνενώνονται· συμπεριφέρονται ως ένα πρόγραμμα με ερωτηματικά ανάμεσά τους.

```
pperl -e 'my $x = 41;' -e 'print $x + 1, "\n"'    # 42
```

Χρησιμοποιήστε μονά εισαγωγικά γύρω από το πρόγραμμα ώστε το κέλυφος να αφήσει αδιάφορα τα `$_`, `$1`, `\n` και τους ανάστροφους χαρακτήρες. Δείτε το *gotchas* για το τι πάει στραβά με τα διπλά εισαγωγικά.

`-E` - `-e` συν το σύνολο σύγχρονων χαρακτηριστικών

Το `-E` είναι το `-e` με ενεργοποιημένα όλα τα προαιρετικά χαρακτηριστικά: `say`, `state`, `fc`, τον τελεστή defined-or `//`, σημασιολογία συμβολοσειρών unicode υπό `use feature 'unicode_strings'`, και άλλα.

```
pperl -E 'say "hello"'                    # instead of print "hello\
 ↪n"
pperl -E 'say "pi = " . (atan2(1,1)*4)'
```

Καταφύγετε στο `-E` όταν γράφετε νέα μονογραμμικά. Καταφύγετε στο `-e` μόνο όταν έχει σημασία η συμβατότητα αντιγραφής-επικόλλησης με πολύ παλιά σενάρια.

`-n` - τύλιγμα σιωπηρού βρόχου ανάγνωσης γύρω από το πρόγραμμα

Το `-n` μετατρέπει το πρόγραμμα σε σώμα βρόχου πάνω σε κάθε γραμμή κάθε αρχείου εισόδου (ή στο STDIN αν δεν δοθεί κανένα). Ο βρόχος είναι ακριβώς:

```
while (<>) {
    # your program
}
```

Το $_ κρατάει κάθε γραμμή εισόδου, συμπεριλαμβανομένου του τελικού χαρακτήρα αλλαγής γραμμής. Δεν τυπώνεται τίποτα αυτόματα - εσείς αποφασίζετε τι φτάνει στο STDOUT.

```
# Print lines containing "error"
pperl -ne 'print if /error/' app.log
```

Ισοδύναμη εκτεταμένη μορφή, που δείχνεται μία φορά ώστε να είναι σαφής η μηχανική επέκταση:

```
while (<>) {
    print if /error/;
}
```

Κάθε επόμενη συνταγή -n στον οδηγό βασίζεται σε αυτή την επέκταση - υποθέστε την χωρίς να την επανατυπώνετε.

Το BEGIN { … } εκτελείται πριν τον βρόχο, το END { … } εκτελείται μετά. Και τα δύο είναι κανονικά μπλοκ Perl και συνθέτονται καθαρά με το -n.

```
# Count lines matching a pattern
pperl -ne '$n++ if /WARN/; END { print "$n\n" }' app.log
```

-p - σαν το -n, αλλά τυπώνει το $_ μετά από κάθε επανάληψη

```
pperl -pe 's/foo/bar/g' input.txt
```

Μηχανική επέκταση:

```
while (<>) {
    # your program
} continue {
    print or die "-p destination: $!";
}
```

Καταφύγετε στο -p όταν η έξοδος είναι η (πιθανώς τροποποιημένη) είσοδος. Καταφύγετε στο -n όταν η έξοδος είναι μια σύνοψη, ένα φιλτραρισμένο υποσύνολο ή τίποτα.

-l - χειρισμός τέλους γραμμής

Το -l κάνει δύο πράγματα ταυτόχρονα:

1. Στην είσοδο, chomp κάθε γραμμή καθώς διαβάζεται υπό -n / -p. Το $_ δεν τελειώνει πλέον σε "\n".
2. Στην έξοδο, ορίζει το $\ (τον διαχωριστή εγγραφών εξόδου) ώστε κάθε print να προσαρτά "\n".

```
# Print just the first field of each tab-separated line
pperl -F'\t' -lane 'print $F[0]' table.tsv
```

Χωρίς το `-l`, είτε θα γράφατε `print "$F[0]\n"` (ο χαρακτήρας αλλαγής γραμμής μεταφέρεται μέσα στο πρόγραμμα) είτε θα παράγατε κολλημένη έξοδο.

Το `-l` δέχεται προαιρετικό οκταδικό όρισμα που ορίζει το `$\` σε διαφορετικό χαρακτήρα: το `-l012` διατηρεί τον χαρακτήρα αλλαγής γραμμής (την προεπιλογή), το `-l0` ορίζει το `$\` σε NUL, και ούτω καθεξής. Η συνηθισμένη χρήση είναι η μορφή χωρίς όρισμα.

`-a` - αυτόματη διάσπαση σε @F

Σε συνδυασμό με `-n` ή `-p`, το `-a` διασπά το `$_` σε λευκούς χαρακτήρες και τοποθετεί τα πεδία στο `@F`. Η προεπιλεγμένη διάσπαση είναι `split(' ', $_)` - περικόπτει τους λευκούς χαρακτήρες στην αρχή και αντιμετωπίζει κάθε διαδοχή λευκών χαρακτήρων ως έναν διαχωριστή.

```
# Third column of every line
pperl -lane 'print $F[2]' data.txt

# Reverse column order
pperl -lane 'print "@{[reverse @F]}"' data.txt
```

Το `@F` έχει δείκτες από το μηδέν. Το `$F[-1]` είναι το τελευταίο πεδίο. Το `scalar @F` είναι το πλήθος πεδίων.

`-F` - αλλαγή του μοτίβου διάσπασης

`-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
pperl -F: -lane 'print $F[1]' /etc/passwd

# Split on any run of commas or semicolons
pperl -F'[,;]+' -lane 'print scalar @F' mixed.csv
```

Το `-F` δεν υπαινίσσεται τίποτα για το `-a`· χρειάζεστε ούτως ή άλλως το `-a`. Το συνηθισμένο ιδίωμα `-F: -lane` σημαίνει διάσπαση με άνω κάτω τελεία, chomp εφαρμοσμένο, με το `@F` διαθέσιμο.

Η παρουσίαση στηλοθέτη απαιτεί κυριολεκτικό στηλοθέτη μέσα στο κέλυφος ή τη μορφή regex:

```
pperl -F'\t' -lane 'print $F[0]' data.tsv       # regex form, always␣
↪safe
pperl -F'        ' -lane 'print $F[0]' data.tsv     # literal tab -␣
↪fragile
```

`-i` - επιτόπια επεξεργασία αρχείων

Το `-i` ξαναγράφει το αρχείο από το οποίο διαβάζει το `pperl`, αντικαθιστώντας το περιεχόμενό του με ό,τι τυπώνει το `-p`. Όλη η έξοδος του αρχείου πηγαίνει σε ένα προσωρινό αρχείο· όταν ο βρόχος τελειώσει, αυτό το προσωρινό αντικαθιστά το πρωτότυπο.

```
# Replace CRLF with LF in every .txt under cwd (no backup)
pperl -i -pe 's/\r\n/\n/g' *.txt

# Same, keeping a .bak copy beside each original
pperl -i.bak -pe 's/\r\n/\n/g' *.txt
```

Πάντα να χρησιμοποιείτε `-i.bak` (ή άλλη κατάληξη) μέχρι να επαληθεύσετε την επεξεργασία πάνω σε αντιπροσωπευτικό αρχείο. Ένα σφάλμα στο πρόγραμμα με σκέτο `-i` καταστρέφει την είσοδο. Δείτε το *gotchas*.

Το `-i` δεν έχει αποτέλεσμα χωρίς `-p` (ή ρητές εκτυπώσεις υπό `-n`): αν δεν τυπωθεί τίποτα, το αρχείο αντικατάστασης είναι κενό.

`-M` και `-m` - φόρτωση αρθρωμάτων πριν εκτελεστεί το πρόγραμμα

Το `-Mmodule` ισοδυναμεί με `use module;` στην κορυφή του προγράμματος. Το `-Mmodule=sym1,sym2` είναι `use module qw(sym1 sym2);`.

```
pperl -MList::Util=sum -lane 'print sum @F' data.txt
pperl -MPOSIX=strftime -le 'print strftime("%F", localtime)'
pperl -MData::Dumper -e 'print Dumper({a=>1, b=>[2,3]})'
```

Το πεζό `-m` είναι `no strict …` αντί για `use …`· σπάνια χρειάζεται στη γραμμή εντολών.

Πιο συνηθισμένα αρθρώματα σε μονογραμμικά: `List::Util` (sum/min/max/uniq/any), `POSIX` (strftime, mktime, fmod), `Data::Dumper` (επιθεώρηση δομών), `JSON::PP` (ανάλυση/εκπομπή JSON), `Text::CSV` (CSV με εισαγωγικά), `MIME::Base64`, `Digest::MD5`, `Digest::SHA`, `Socket`.

`-0` - ορισμός του διαχωριστή εγγραφών εισόδου

Η Perl κανονικά διαβάζει μία γραμμή τη φορά. Το `-0NNN` ορίζει το `$/` στον χαρακτήρα με τον δοσμένο οκταδικό κωδικό, αλλάζοντας τι σημαίνει «μία εγγραφή» για τα `<>`, `<STDIN>` και τον βρόχο `-n` / `-p`.

Συνηθισμένες μορφές:

Μορφή	Σε τι γίνεται το `$/`	Αποτέλεσμα
`-0`	`"\0"` (NUL)	Διάβασμα εγγραφών οριοθετημένων με NUL (ταιριάζει με `find -print0`).
`-0777`	`undef`	Slurp ολόκληρου του αρχείου ως μία εγγραφή.
`-00`	`""`	Λειτουργία παραγράφου: οι κενές γραμμές διαχωρίζουν εγγραφές.
`-0NNN`	`chr(0NNN)`	Οποιοδήποτε μεμονωμένο οκταδικό 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
pperl -0777 -ne 'print "yes\n" if /BEGIN.*?END/s' text.txt

# Print every paragraph containing "TODO"
pperl -00 -ne 'print if /TODO/' notes.txt
```

Η λειτουργία παραγράφου (`-00`) αλληλεπιδρά με το `-l`: χωρίς `-l`, ο κενός διαχωριστής παραμένει προσαρτημένος σε κάθε εγγραφή· με `-l`, εφαρμόζεται chomp.

`-C` - Unicode σε stdio ή/και @ARGV

Το `-C` ενεργοποιεί τον χειρισμό Unicode σε καθορισμένα ρεύματα. Το όρισμα είναι είτε αριθμός (μάσκα bits) είτε συμβολοσειρά γραμμάτων.

```
pperl -CSD -ne 'print' input.txt      # stdin, stdout, stderr all
↪UTF-8
pperl -CA  -e  'print $ARGV[0]' äöü   # @ARGV treated as UTF-8
pperl -CSA -nE 'say length'           # S + A
```

Κωδικοί γραμμάτων: `I` = stdin, `O` = stdout, `E` = stderr, `S` = `I+O+E`, `A` = `@ARGV`, `D` = ορίζει προεπιλεγμένα στρώματα I/O (συμπεριλαμβάνει ανοίγματα αρχείων).

Για πλήρως UTF-8 συνεδρία, `-CSD`. Για εφάπαξ επεξεργασία ονομασμένων αρχείων UTF-8, `-CSAD`. Δείτε το *gotchas* για τα συμπτώματα που σας λένε ότι λείπει το `-C`.

Πίνακας αναφοράς

Κάθε σημαία, η μηχανική επέκτασή της και το κεφάλαιο όπου εμφανίζεται πρώτη φορά σε συνταγή.

Σημαία	Επεκτείνεται σε	Πρώτη χρήση στο
`-e CODE`	Εκτέλεση του CODE ως προγράμματος	*progression*
`-E CODE`	`-e` συν όλα τα τρέχοντα χαρακτηριστικά ενεργοποιημένα	*progression*
`-n`	`while (<>) { CODE }`	*progression*
`-p`	`while (<>) { CODE } continue { print }`	*progression*
`-l`	chomp εισόδου υπό `-n`/`-p`· ορίζει `$\ = "\n"` για την έξοδο	*progression*
`-a`	Αυτόματη διάσπαση του `$_` σε `@F`	*progression*
`-F PAT`	Αλλαγή του μοτίβου διάσπασης του `-a` στο regex PAT	*progression*
`-i[EXT]`	Επιτόπια επεξεργασία· με EXT, διατηρεί αντίγραφο ασφαλείας `original + EXT`	*progression*
`-M MOD`	`use MOD;` πριν το πρόγραμμα	*numeric*
`-M MOD=X`	`use MOD qw(X);`	*numeric*
`-0`	`$/ = "\0"` - εγγραφές οριοθετημένες με NUL	*progression*
`-00`	`$/ = ""` - λειτουργία παραγράφου	*progression*
`-0777`	`$/ = undef` - slurp ολόκληρου αρχείου	*progression*
`-C`	Ενεργοποίηση Unicode στο stdio, στο `@ARGV` ή/και στα προεπιλεγμένα στρώματα	*gotchas*

Ειδικές μεταβλητές που στήνουν οι σημαίες

Σύντομος κατάλογος· οι πλήρεις ορισμοί ζουν στο `perlvar`.

Μεταβλη	Ρόλος υπό `-n`/`-p`
`$_`	Η τρέχουσα γραμμή εισόδου (chomped αν `-l`, ωμή αλλιώς).
`$.`	Τρέχων αριθμός γραμμής εισόδου (δεν μηδενίζεται μεταξύ αρχείων από προεπιλογή).
`$/`	Διαχωριστής εγγραφών εισόδου. Η οικογένεια `-0` τον αλλάζει.
`$\`	Διαχωριστής εγγραφών εξόδου που προσαρτά το `print`. Το `-l` τον ορίζει σε `"\n"`.
`$,`	Διαχωριστής πεδίων εξόδου που εισάγεται ανάμεσα στα ορίσματα του `print`.
`$"`	Διαχωριστής ανάμεσα στα στοιχεία πίνακα όταν παρεμβάλλονται σε `"@array"`.
`@F`	Πεδία αυτόματης διάσπασης υπό `-a`.
`@ARGV`	Εναπομείναντα ορίσματα γραμμής εντολών· τα ονόματα αρχείων καταναλώνονται καθώς επεξεργάζονται.
`$ARGV`	Όνομα του αρχείου που διαβάζεται τη στιγμή αυτή (το `"-"` σημαίνει STDIN).

Μάθετε περισσότερα

- Το *progression* εφαρμόζει κάθε σημαία σε συνταγές αυξανόμενης πολυπλοκότητας.
- Το `perlvar` απαριθμεί κάθε ειδική μεταβλητή που ορίζει το pperl, όχι μόνο εκείνες που αγγίζουν οι σημαίες.
- Το `perlop` καλύπτει το `qr//`, τον τελεστή flip-flop και τον τελεστή «διαμάντι» που χρησιμοποιούν οι βρόχοι `-n` / `-p`.

Προοδευτικές συνταγές

Το κεφάλαιο αυτό προχωρά από τετριμμένες αντικαταστάσεις προς αναγωγές, εξαγωγή και δομημένη έξοδο. Κάθε ενότητα επαναχρησιμοποιεί τις σημαίες που εισήχθησαν στην προηγούμενη, ώστε ένας αναγνώστης που διαβάζει συνεχόμενα να χτίζει το λεξιλόγιο των ιδιωμάτων χωρίς ποτέ να συναντά νέα σημαία στη μέση μιας συνταγής.

Οι αναγνώστες που έρχονται από awk/sed θα πρέπει να αναγνωρίσουν τις πρώτες ενότητες· το ενδιαφέρον υλικό ξεκινά γύρω από τις *Αναγωγές σε γραμμές* και τα *Φίλτρα παραθύρου περιβάλλοντος*.

Κάθε συνταγή αντιγράφεται και επικολλάται απευθείας. Όταν έχει σημασία ένα δείγμα εισόδου, παρουσιάζεται ενσωματωμένα ώστε η αναμενόμενη έξοδος να είναι σαφής.

Η αφετηρία: `-e` και `-E`

Το συντομότερο μονόγραμμο. Χωρίς βρόχο I/O, χωρίς διάσπαση πεδίων - απλώς ένα πρόγραμμα που εκτελείται μία φορά.

```
pperl -e 'print 2 ** 32, "\n"'                    # 4294967296
pperl -E 'say 2 ** 32'                            # same, with -E/
↪say
pperl -E 'say for 1 .. 5'                         # 1\n2\n3\n4\n5
```

Το `,` στο `print` διαχωρίζει ορίσματα· ο διαχωριστής πεδίων εξόδου `$,` (προεπιλεγμένα κενός) μπαίνει ανάμεσά τους.

Φιλτράρισμα: `-n` + `print if`

Το `-n` τυλίγει το πρόγραμμα σε έναν βρόχο ανάγνωσης πάνω σε κάθε γραμμή κάθε αρχείου εισόδου (ή στο STDIN). Δεν τυπώνεται τίποτα εκτός αν το πει το πρόγραμμα.

```
# Sample input
$ printf 'gate\napple\nwhat\nkite\n' > words.txt

pperl -ne 'print if /at/' words.txt               # gate, what
pperl -ne 'print unless /e/' words.txt            # what
pperl -ne 'print if /^[aeiou]/' words.txt         # apple
```

Η αντιστοίχιση regex γίνεται κατά του `$_` από προεπιλογή· το `/at/` είναι συντομογραφία του `$_ =~ m/at/`.

Πολλαπλές συνθήκες

```
# Lines containing both "ERROR" and "timeout"
pperl -ne 'print if /ERROR/ && /timeout/' app.log

# Lines containing neither
pperl -ne 'print if !/ERROR/ && !/timeout/' app.log

# Events A, B, C appearing in order on one line
pperl -ne 'print if /A.*B.*C/' events.log
```

Με αριθμό γραμμής

```
pperl -ne 'print if $. == 1'                    # head -n 1
pperl -ne 'print if $. <= 10'                   # head -n 10
pperl -ne 'print if 17 .. 30'                   # lines 17-30␣
↪(range op)
pperl -ne 'print if /START/ .. /END/'           # regex range,␣
↪inclusive
pperl -ne 'print unless $. % 2'                 # even lines
```

Ο τελεστής εύρους `..` σε βαθμωτό περιβάλλον είναι flip-flop: συγκρίνεται με το `$.` όταν και τα δύο άκρα είναι ακέραιοι, με το `$_` όταν είναι μοτίβα. Δείτε το `perlop`.

Με μήκος

Χρησιμοποιήστε `-l` ώστε το μετρούμενο μήκος να αποκλείει τον τελικό χαρακτήρα αλλαγής γραμμής.

```
pperl -lne 'print if length >= 80' src.c        # long lines
pperl -lne 'print if length() < 5' words.txt    # short lines
```

Σκέτο `length` χωρίς όρισμα εφαρμόζεται στο `$_`.

Αντικατάσταση: `-p` + `s///`

Το `-p` είναι το `-n` συν ένα σιωπηρό `print` μετά από κάθε επανάληψη. Χρησιμοποιήστε το όταν η έξοδος είναι η (πιθανώς τροποποιημένη) είσοδος.

```
# Replace every "foo" with "bar"
pperl -pe 's/foo/bar/g' input.txt

# First occurrence per line only
pperl -pe 's/foo/bar/' input.txt

# Conditional: replace only on lines that also match BAZ
pperl -pe 's/foo/bar/g if /BAZ/' input.txt
```

Μετασχηματισμοί πεζών-κεφαλαίων

```
pperl -nle 'print uc'                                  # to uppercase
pperl -nle 'print lc'                                  # to lowercase
pperl -nle 'print ucfirst lc'                          # Sentence case
pperl -ple 's/(\w+)/\u$1/g'                            # Title Case Per␣
↪Word
```

Τα `uc`, `lc` και `ucfirst` χωρίς όρισμα λειτουργούν επί του `$_`.

Κανονικοποίηση λευκών χαρακτήρων

```
pperl -ple 's/^\s+//'                                  # trim left
pperl -ple 's/\s+$//'                                  # trim right
pperl -ple 's/^\s+|\s+$//g'                            # trim both
pperl -ple 's/\s+/ /g'                                 # collapse runs
pperl -ne  'print if /\S/'                             # drop blank␣
↪lines
```

Μετατροπή τέλους γραμμής

```
pperl -pe 's/\r\n/\n/g' win.txt                        # DOS  → UNIX
pperl -pe 's/(?<!\r)\n/\r\n/g' unix.txt                # UNIX → DOS
```

Η οπίσθια διεκδίκηση `(?<!\r)` αποφεύγει την παραγωγή `\r\r\n` όταν η είσοδος είναι ήδη DOS-κωδικοποιημένη. Δείτε το *regex-recipes* για διεκδικήσεις γύρω σε βάθος.

Δουλειά προσανατολισμένη στα πεδία: `-a` και `-F`

Το `-a` διασπά αυτόματα κάθε γραμμή στο `@F`. Το `-F` αλλάζει τον οριοθέτη.

```
$ cat table.txt
brown bread mat hair 42
blue cake mug shirt -7
yellow banana window shoes 3.14

# Second field
pperl -lane 'print $F[1]' table.txt
# bread, cake, banana

# Last field
pperl -lane 'print $F[-1]' table.txt
# 42, -7, 3.14

# Count fields per line
pperl -lane 'print scalar @F' table.txt
# 5, 5, 5

# Lines whose last field is negative
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
pperl -lane 'print if $F[-1] < 0' table.txt
# blue cake mug shirt -7
```

Προσαρμοσμένος οριοθέτης

```
# /etc/passwd: print username (field 1) and shell (last field)
pperl -F: -lane 'print "$F[0] $F[-1]"' /etc/passwd

# Tab-separated
pperl -F'\t' -lane 'print $F[2]' metrics.tsv

# Comma or semicolon, one or more
pperl -F'[,;]+' -lane 'print scalar @F' mixed.csv
```

Το γνήσιο CSV (πεδία σε εισαγωγικά με ενσωματωμένα κόμματα) δεν είναι δουλειά για το `-F`. Χρησιμοποιήστε `-MText::CSV` - δείτε το *numeric* για το μοτίβο.

Επανένωση πεδίων

Η παρεμβολή του `@F` μέσα σε διπλά εισαγωγικά χρησιμοποιεί το `$"` (προεπιλογή: ένα διάστημα) ως διαχωριστή.

```
# Reverse field order, space-separated
pperl -lane 'print "@{[reverse @F]}"' table.txt

# Reverse field order, colon-separated
pperl -F: -lane 'local $" = ":"; print "@{[reverse @F]}"' /etc/
↪passwd
```

Το `@{[ EXPR ]}` είναι το ιδίωμα αναφορά-πίνακα-μετά-αποαναφορά που επιτρέπει αυθαίρετες εκφράσεις να παρεμβληθούν σε συμβολοσειρά διπλών εισαγωγικών.

Αναγωγές σε γραμμές

Το πέρασμα από φίλτρα ανά γραμμή σε υπολογισμό κατά μήκος γραμμών.

Μετρήσεις

```
# wc -l
pperl -lne 'END { print $. }' big.txt

# grep -c pattern
pperl -lne '$n++ if /pattern/; END { print $n + 0 }' file

# Count blank lines
pperl -lne '$n++ if /^$/; END { print $n + 0 }' file
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
# Total fields across the file
pperl -lane '$t += @F; END { print $t }' table.txt
```

Το ιδίωμα `$n + 0` τυπώνει `0` αντί για κενό όταν ο μετρητής δεν αυξήθηκε ποτέ.

Αθροίσματα, ελάχιστα, μέγιστα

Χρησιμοποιήστε το `List::Util` μέσω `-M`.

```
# Per line
pperl -MList::Util=sum -lane 'print sum @F' data.txt
pperl -MList::Util=min -lane 'print min @F' data.txt
pperl -MList::Util=max -lane 'print max @F' data.txt

# Across all lines
pperl -MList::Util=sum -lane '$s += sum @F; END { print $s }' data.
↪txt
pperl -MList::Util=max -alne '$m = max($m // (), @F); END { print $m␣
↪}' data.txt
```

Το `$m // ()` επιστρέφει κενή λίστα όταν το `$m` είναι απροσδιόριστο, ώστε η πρώτη επανάληψη να μην περάσει `undef` στο `max`. Δείτε το *numeric* για το πλήρες κεφάλαιο αριθμητικής.

Μακρύτερη και βραχύτερη γραμμή

```
pperl -ne '$l = $_ if length > length $l; END { print $l }' file
pperl -ne '$s = $_ if $. == 1 || length < length $s; END { print $s␣
↪}' file
```

Μοναδικές γραμμές (μόνο πρώτη εμφάνιση)

```
pperl -ne 'print unless $seen{$_}++' file

# Or the List::Util one-liner (slurps):
pperl -MList::Util=uniq -e 'print uniq <>' file
```

Η πρώτη μορφή ροηγείται - η μνήμη κλιμακώνεται με τις διακριτές γραμμές, όχι με τις συνολικές. Η δεύτερη ρουφάει τα πάντα στη μνήμη, μετά εκπέμπει. Προτιμήστε την πρώτη για μεγάλες εισόδους.

Μόνο διπλότυπα

```
# Print each duplicate line once, when first seen for the second␣
↪time
pperl -ne 'print if ++$seen{$_} == 2' file
```

Φίλτρα παραθύρου περιβάλλοντος

Εκτύπωση της γραμμής *γύρω* από μια αντιστοιχία, όχι μόνο της ίδιας της αντιστοιχίας.

```
# Line immediately before each match (grep -B 1, with caveats)
pperl -ne '/pattern/ && $prev && print $prev; $prev = $_' file

# Line immediately after each match (grep -A 1)
pperl -ne 'print if $p; $p = /pattern/' file

# tail -n 10
pperl -ne 'push @a, $_; shift @a if @a > 10; END { print @a }' file
```

Για πλούσια συμπεριφορά `-A` / `-B` / `-C`, το `grep` είναι το σωστό εργαλείο. Τα μονογραμμικά ταιριάζουν όταν ο κανόνας παραθύρου είναι ασυνήθιστος (π.χ. «τύπωσε τη γραμμή που ταιριάζει με X, καθώς και κάθε γραμμή μέχρι την επόμενη κενή γραμμή»).

```
# Print from /SECTION/ to the next blank line
pperl -ne 'print if /SECTION/ .. /^$/' file
```

Λειτουργία εγγραφών: `-0`, `-00`, `-0777`

Απομακρυνθείτε από την ανάγνωση βασισμένη σε γραμμές όταν η φυσική εγγραφή δεν είναι μια γραμμή.

Slurp με `-0777`

```
# Single-shot regex across the whole file
pperl -0777 -ne 'print "match\n" if /BEGIN.*?END/s' text.txt

# Replace only the first occurrence across the file
pperl -0777 -i.bak -pe 's/TODO/DONE/' notes.txt
```

Το `-0777` ορίζει `$/ = undef`. Όλο το αρχείο έρχεται ως ένα `$_`. Ο τροποποιητής `/s` κάνει το `.` να ταιριάζει με χαρακτήρες αλλαγής γραμμής. Δείτε το *regex-recipes* για την πλήρη πολυγραμμική εικόνα.

Λειτουργία παραγράφου με `-00`

Το `-00` ορίζει `$/ = ""`. Κάθε ανάγνωση επιστρέφει τα πάντα μέχρι τον επόμενο οριοθέτη κενής γραμμής.

```
# Paragraphs containing "TODO"
pperl -00 -ne 'print if /TODO/' notes.md

# Reverse paragraph order
pperl -00 -e 'print reverse <>' notes.md
```

Εγγραφές οριοθετημένες με NUL μέσω `-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/'
```

Επιτόπια επεξεργασία με `-i`

Το `-i` αντικαθιστά κάθε αρχείο εισόδου με ό,τι τυπώνει το `-p`. Το `-i.bak` διατηρεί αντίγραφο ασφαλείας.

```
# Lowercase every occurrence of HOSTNAME in every .conf (keep␣
↪backup)
pperl -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)
pperl -i -ne 'print unless /DEPRECATED/' file.txt
```

Τα αρχεία αντιγράφων ασφαλείας συσσωρεύονται. Καθαρίστε τα με `rm *.bak` αφού επαληθεύσετε. Δείτε το *gotchas* για τις κλασικές παγίδες επιτόπιας επεξεργασίας.

Δομημένη έξοδος

Τύπου CSV (ελαφρύ - τα κόμματα δεν εισάγονται σε εισαγωγικά)

```
# Rebuild a TSV as CSV (naive, assumes no commas in fields)
pperl -F'\t' -lane 'print join ",", @F' data.tsv

# Add a column of line numbers
pperl -lane 'print join "\t", $., @F' data.tsv
```

JSON (τυπική βιβλιοθήκη)

```
# Emit one JSON object per line (no commas in values assumed)
pperl -MJSON::PP -F: -lane '
    print JSON::PP->new->encode({
        user => $F[0], uid => 0+$F[2], shell => $F[-1]
    })
' /etc/passwd
```

Τα πολυγραμμικά μονογραμμικά σαν αυτό είναι εκεί που ένα *alias* αρχίζει να αποδίδει.

Αριθμημένη είσοδος

```
pperl -pe '$_ = "$. $_"'                          # 1 gate, 2 apple,␣
↪...
pperl -pe 's/^/sprintf "%5d  ", $./e'             # padded width
```

Ο τροποποιητής `/e` στο `s///` αντιμετωπίζει την αντικατάσταση ως κώδικα Perl. Δείτε το *regex-recipes*.

Μάθετε περισσότερα

- *regex-recipes* - τα μονογραμμικά που εξαρτώνται από regex: συλλήψεις, διεκδικήσεις γύρω, ονομαστικές ομάδες, `/e`, `/r`.
- *numeric* - αριθμητική, στατιστική, τυχαία, αριθμητική ημερομηνιών.
- *aliases* - μετατρέψτε τις συνταγές που χρησιμοποιείτε εβδομαδιαία σε συναρτήσεις κελύφους.
- *gotchas* - εισαγωγικά, κωδικοποίηση, αντίγραφα ασφαλείας `-i`, εκπλήξεις με τον διαχωριστή εγγραφών.
- `perlop` - ο τελεστής flip-flop, ο τελεστής «διαμάντι», `s///`, `tr///`, μορφές εισαγωγικών.

Μονογραμμικά οδηγούμενα από regex

Το προηγούμενο κεφάλαιο κράτησε τα regexps στο επίπεδο που γνωρίζουν ήδη οι χρήστες `sed` ή `awk`: κυριολεκτικά μοτίβα μέσα σε `m//` και `s///`. Το παρόν κεφάλαιο καλύπτει μονογραμμικά των οποίων η μορφή εξαρτάται από κάποιο χαρακτηριστικό regex πέρα από την απλή αντιστοίχιση - εξαγωγές, συλλήψεις, διεκδικήσεις γύρω, κώδικα-στην-αντικατάσταση και μη καταστροφικές επανεγγραφές.

Οι αναγνώστες που γνωρίζουν ήδη τις κατασκευές (`(?:...)`, `(?<name>...)`, `(?=...)`, τους τροποποιητές `/e` και `/r`) μπορούν να περιδιαβούν το κείμενο για το πλαίσιο γραμμής εντολών. Όσοι ξέρουν τι κάνει το `/g` αλλά δεν έχουν γράψει ποτέ `/e` θα ωφεληθούν περισσότερο διαβάζοντας συνεχόμενα.

Για την ίδια τη γλώσσα regex, δείτε τον *οδηγό κανονικών εκφράσεων*.

Εξαγωγή όλων των αντιστοιχιών

Το `/g` σε περιβάλλον λίστας επιστρέφει κάθε αντιστοιχία, όχι μόνο την πρώτη.

```
# Every integer in the input
pperl -ne 'print "$_\n" for /-?\d+/g' file.txt

# Every IPv4-shaped substring
pperl -lne 'print for /\b(?:\d{1,3}\.){3}\d{1,3}\b/g' access.log

# Every HTTP header value (naive: no folded headers)
pperl -nE 'say $1 if /^[\w-]+:\s*(.+)$/' headers.txt
```

Το `say` (ενεργοποιείται από `-E`) προσαρτά χαρακτήρα αλλαγής γραμμής, εξοικονομώντας ένα `"\n"` και κρατώντας το πρόγραμμα σύντομο.

Μη άπληστοι ποσοδείκτες

`*?`, `+?`, `??`, `{n,m}?` - ταιριάζουν με όσο λιγότερους χαρακτήρες γίνεται.

```
# Everything between the first < and the next > (tag-by-tag)
pperl -lne 'print for /<(.+?)>/g' markup.html

# Every quoted string (double quotes, no escape handling)
pperl -lne 'print for /"([^"]*)"/g' config.txt
```

Η αρνημένη κλάση χαρακτήρων `[^"]*` είναι συνήθως ταχύτερη και ασφαλέστερη μη άπληστη επιλογή από το `.*?` όταν γνωρίζετε τι δεν πρέπει να εμφανιστεί μέσα.

Συλλήψεις στην αντικατάσταση

Η δεξιά πλευρά του `s///` μπορεί να αναφέρεται σε συλλήψεις της αριστερής πλευράς.

```
# Swap every "word1 word2" pair
pperl -pe 's/(\w+)\s+(\w+)/$2 $1/g' file.txt

# Quote every unquoted key in key=value pairs
pperl -pe 's/(\w+)=/$1="/g; s/=([^"\n]+)/"$1"/g' config.txt

# Increase every trailing number by one
pperl -pe 's/(\d+)$/$1 + 1/e' file.txt
```

`/e` - η αντικατάσταση είναι κώδικας Perl

Ο τροποποιητής `/e` αντιμετωπίζει την αντικατάσταση ως κώδικα· η τιμή επιστροφής του είναι αυτό που τοποθετείται.

```
# Number every word in the file (global counter)
pperl -pe 's/(\w+)/++$i . ".$1"/ge' file.txt

# Number words per line (counter reset each line)
pperl -pe '$i = 0; s/(\w+)/++$i . ".$1"/ge' file.txt

# Wrap every integer in base 16
pperl -pe 's/\d+/sprintf "0x%x", $&/ge' file.txt

# Convert Celsius-labelled numbers to Fahrenheit
pperl -pe 's/(\d+)C\b/sprintf "%.1fF", $1 * 9 / 5 + 32/ge' weather.
↪txt
```

Το `/ge` μαζί: αποτίμηση της δεξιάς πλευράς ως κώδικα, για κάθε αντιστοιχία. Το `$&` είναι ολόκληρο το κείμενο που ταίριαξε· τα `$1`, `$2`, … είναι συλλήψεις.

Το `/ee` αποτιμά δύο φορές - το πρώτο e παράγει συμβολοσειρά, το δεύτερο αποτιμά αυτή

τη συμβολοσειρά. Χρήσιμο όταν η ίδια η συμβολοσειρά αντικατάστασης είναι δυναμικός κώδικας που διαβάζεται από την είσοδο· χρειάζεται σπάνια, συνιστά κίνδυνο συχνά.

`/r` - μη καταστροφική αντικατάσταση

Το `/r` επιστρέφει την τροποποιημένη συμβολοσειρά αντί να ξαναγράψει το `$_`. Το πρωτότυπο παραμένει ανέπαφο.

```
# Print filename and its .bak variant side by side
ls *.txt | pperl -pe '$_ = $_ . ($_ =~ s/$/.bak/r)'

# Compute a transformed version without touching $_
pperl -ne 'my $upper = $_ =~ tr/a-z/A-Z/r; print $_, $upper' file.
↪txt
```

Καταφύγετε στο `/r` όταν μία γραμμή πρέπει να τυπωθεί σε δύο μορφές, ή όταν ο μετασχηματισμός είναι υπό συνθήκη και θέλετε την ανέπαφη γραμμή πίσω στον κλάδο αποτυχίας.

Ονομαστικές συλλήψεις

Το `(?<name>...)` συλλαμβάνει στο `$+{name}` και στο `%+`.

```
# Parse combined-log-format access lines
pperl -nE '
    if (/^(?<ip>\S+).*?"(?<method>\S+) (?<path>\S+)/) {
        say "$+{ip} $+{method} $+{path}";
    }
' access.log
```

Οι ονομαστικές συλλήψεις αξίζουν τον κόπο μόλις το μοτίβο έχει πάνω από δύο ομάδες, ή όταν η ίδια ομάδα χρειάζεται αναφορά τόσο στο μοτίβο όσο και στον κώδικα αντικατάστασης.

Πρόσθια και οπίσθια διεκδίκηση

Διεκδικήσεις μηδενικού πλάτους: ταιριάζουν με μια θέση όπου το περιβάλλον κείμενο ικανοποιεί μια συνθήκη, χωρίς να καταναλώνουν την ίδια τη συνθήκη.

```
# Digits preceded by $ (e.g. "$42" → 42, ignore "42")
pperl -lne 'print for /(?<=\$)\d+/g' prices.txt

# Words NOT followed by "!"
pperl -lne 'print for /\b\w+\b(?!!)/g' shouts.txt

# Insert CRLF only where LF is not already preceded by CR
pperl -pe 's/(?<!\r)\n/\r\n/g' unix.txt
```

Συνηθισμένα ζεύγη:

Διεκδίκηση γύρω	Σημασία
`(?=X)`	Οι επόμενοι χαρακτήρες ταιριάζουν με X
`(?!X)`	Οι επόμενοι χαρακτήρες δεν ταιριάζουν με X
`(?<=X)`	Οι προηγούμενοι χαρακτήρες ταιριάζουν με X
`(?<!X)`	Οι προηγούμενοι χαρακτήρες δεν ταιριάζουν με X

Η οπίσθια διεκδίκηση στο pperl υποστηρίζει μοτίβα μεταβλητού μήκους· δείτε *κανονικές εκφράσεις: άγκυρες και διεκδικήσεις*.

Πολυγραμμικές αντιστοιχίσεις σε ρουφηγμένη είσοδο

Το `-0777` ρουφάει ολόκληρο το αρχείο στο `$_`. Οι τροποποιητές regex που έχουν σημασία σε αυτή τη λειτουργία:

- `/s` - το `.` ταιριάζει με `\n`.
- `/m` - τα `^` και `$` ταιριάζουν σε κάθε αρχή/τέλος γραμμής, όχι μόνο στην αρχή/τέλος ολόκληρης της συμβολοσειράς.

```
# Every BEGIN...END block (non-greedy), each on one output line
pperl -0777 -lne '
    while (/BEGIN(.*?)END/sg) {
        (my $body = $1) =~ s/\n/ /g;
        print $body;
    }
' text.txt

# Paragraphs where every line starts with ">"
pperl -00 -ne 'print if /\A(?:>.*\n?)+\z/m' mail.txt
```

Τα `\A` και `\z` αγκυρώνουν την αρχή και το τέλος ολόκληρης της συμβολοσειράς, ανεπηρέαστα από το `/m`. Χρησιμοποιήστε τα όταν το `/m` είναι ενεργό αλλά εννοείτε πραγματικά ολόκληρη τη συμβολοσειρά.

Μεταγραμματισμός με `tr`

Το `tr` (γραμμένο και ως `y`) είναι αντικατάσταση χαρακτήρα-προς-χαρακτήρα. Δεν είναι regex, αλλά κοντινός συγγενής - και το σωστό εργαλείο για δουλειές με σύνολα χαρακτήρων.

```
pperl -pe 'tr/A-Za-z/N-ZA-Mn-za-m/'                # ROT13
pperl -pe 'tr/A-Za-z/a-zA-Z/'                      # swap case
pperl -pe 'tr/a-z//d'                              # delete lowercase␣
↪letters
pperl -pe 'tr/ \t/ /s'                             # squeeze␣
↪whitespace runs to one space

# Count commas on each line (tr returns the count)
pperl -ne 'print tr/,//, "\n"' data.csv
```

Η σημαία d διαγράφει χαρακτήρες του αριστερού συνόλου που δεν έχουν αντίστοιχο στα δεξιά. Η σημαία s συμπιέζει διαδοχές. Η σημαία c παίρνει το συμπλήρωμα του αριστερού συνόλου (ό,τι ΔΕΝ απαριθμείται).

Ανάλυση οριοθετημένων υποσυμβολοσειρών

CSV με κόμματα σε εισαγωγικά

Μια πρόχειρη διάσπαση που σέβεται τα πεδία σε διπλά εισαγωγικά:

```
pperl -ne '
    my @F = /"([^"]*)"|([^,]+)/g;
    @F = grep defined, @F;
    print join("|", @F), "\n";
' quoted.csv
```

Σωστή και αρκετά γρήγορη για εφάπαξ χρήση. Για οτιδήποτε πρέπει να χειριστεί ενσωματωμένα εισαγωγικά (`"he said ""hi"""`), χρησιμοποιήστε `Text::CSV`· δείτε *numeric*.

Συμβολοσειρές ερωτημάτων URL

```
# Each key=value pair on its own line
pperl -lne 'print for /([^?&=]+)=([^&]*)/g' urls.txt
```

Πρακτικά μονογραμμικά εξαγωγής

Λέξεις κατά μήκος

```
# Words of exactly 5 characters
pperl -lne 'print for /\b\w{5}\b/g' file.txt
```

Αριθμοί με μονάδες

```
pperl -lne 'print for /\b\d+(?:\.\d+)?\s*(?:ms|s|kb|mb|gb)\b/gi'␣
↪timings.txt
```

HTTP User-Agent από αρχείο καταγραφής αιτημάτων

```
pperl -nE 'say $1 if /^User-Agent: (.+)$/' request.log
```

Ισοζυγισμένα άγκιστρα (όσα υποστηρίζει η μηχανή regex του pperl)

Η μηχανή regex του pperl υποστηρίζει αναδρομή μέσω `(?R)` και `(?1)`, άρα η ισοζύγιση αγκίστρων ενός επιπέδου είναι ένα μονόγραμμο:

```
pperl -0777 -nE 'say $& while /\{(?:[^{}]|(?R))*\}/g' code.txt
```

Για βαθιά εμφωλευμένες δομές, καταφύγετε σε αναλυτή. Το regex είναι το σωστό εργαλείο μέχρι το επίπεδο πολυπλοκότητας όπου πάψει να είναι.

Μάθετε περισσότερα

- *οδηγός κανονικών εκφράσεων* - η γλώσσα που υπόκειται κάθε συνταγής εδώ, συμπεριλαμβανομένων τροποποιητών, απόδοσης και Unicode.
- `m` - ο τελεστής αντιστοίχισης.
- `s` - αντικατάσταση, το πλήρες σύνολο τροποποιητών.
- `tr` - αναφορά μεταγραμματισμού.
- *progression* - απλούστερες συνταγές `s///` χωρίς συλλήψεις ή κώδικα.
- `perlop` - η μορφή `qr//` για προμεταγλώττιση μοτίβων που αναφέρονται με όνομα μέσα σε μονόγραμμο.

Αριθμητικά μονογραμμικά και μονογραμμικά ημερομηνιών

Το pperl ως αριθμομηχανή, εργαλείο στατιστικών και μηχανή αριθμητικής ημερομηνιών. Το κεφάλαιο αυτό συγκεντρώνει τις συνταγές που μετατρέπουν αριθμητικές στήλες, χρονικές σφραγίδες και διευθύνσεις IP σε χρήσιμη έξοδο χωρίς αρχείο σεναρίου.

Η υποτιθέμενη αφετηρία είναι αναγνώστης που έχει διαβάσει το *switches* και το *progression*. Τα ιδιώματα `-lane` / `-MList::Util=...` που χρησιμοποιούνται παρακάτω εισήχθησαν εκεί.

Όπου το παρόν κεφάλαιο καταφεύγει στην απόδοση: οι μεγάλες αριθμητικές αναγωγές είναι μία από τις περιπτώσεις όπου το JIT του pperl αξίζει το κόστος του. Μια συσσώρευση `for (1 .. 100_000_000) { $s += $_ }` εκτελείται ως μεταγλωττισμένος κώδικας μηχανής αφού ζεσταθεί ο διερμηνέας. Δεν δηλώνετε συμμετοχή - ενεργοποιείται αυτόματα σε αριθμητικούς βρόχους με έντονη αριθμητική.

Μονογραμμικά απλής αριθμητικής

```
pperl -E 'say 2 ** 64'                                  # 1.
↪84467440737096e+19
pperl -E 'say 2 ** 64 - 1'                              # rounds (float)
pperl -MMath::BigInt -E 'say Math::BigInt->new(2)->bpow(64)'
                                                        #␣
↪18446744073709551616

pperl -E 'say 7 / 3'                                    # 2.33333333333333
pperl -E 'say int(7 / 3)'                               # 2
pperl -E 'say 7 % 3'                                    # 1
pperl -E 'say sqrt(2)'                                  # 1.4142135623731
pperl -E 'say atan2(1, 1) * 4'                          # 3.14159265358979␣
↪(pi)
```

Το `sprintf` ελέγχει το πλάτος και την ακρίβεια της εξόδου:

```
pperl -E 'printf "%.4f\n", 22/7'                    # 3.1429
pperl -E 'printf "%10.2f\n", 42.5'                  #      42.50
pperl -E 'printf "%08d\n", 42'                      # 00000042
pperl -E 'printf "%x %o %b\n", 255, 255, 255'       # ff 377 11111111
```

Αθροίσματα, ελάχιστα, μέγιστα, μέσοι όροι

Το `List::Util` διανέμεται με την Perl 5 - δεν απαιτείται εγκατάσταση.

Ανά γραμμή

```
pperl -MList::Util=sum -lane 'print sum @F'                 data.txt
pperl -MList::Util=min -lane 'print min @F'                 data.txt
pperl -MList::Util=max -lane 'print max @F'                 data.txt

# Per-line mean
pperl -MList::Util=sum -lane 'print sum(@F) / @F'           data.txt
```

Σε όλες τις γραμμές

```
# Total sum
pperl -MList::Util=sum -lane '$s += sum @F; END { print $s }' data.
↪txt

# Running min / max using // (defined-or)
pperl -MList::Util=min -alne '$m = min($m // (), @F); END { print $m
↪}' data.txt
pperl -MList::Util=max -alne '$m = max($m // (), @F); END { print $m
↪}' data.txt

# Mean of one column (third), ignoring blank lines
pperl -lane 'next unless /\S/; $s += $F[2]; $n++;
             END { printf "%.4f\n", $s / $n }' data.txt
```

Το `$m // ()` επιστρέφει κενή λίστα όταν το `$m` είναι απροσδιόριστο, ώστε η πρώτη επανάληψη να μη μολύνει τα min/max με undef. Δείτε το `perlop` για το `//`.

Άθροιση μίας στήλης

```
# Sum of column 3 (zero-indexed: $F[2])
pperl -lane '$s += $F[2]; END { print $s }' data.txt

# Negative or zero entries only
pperl -lane '$s += $F[2] if $F[2] <= 0; END { print $s }' data.txt
```

Τυπική απόκλιση

```
pperl -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
```

Για οτιδήποτε πέρα από μέση τιμή+τυπική απόκλιση, καταφύγετε στο `Statistics::Descriptive`:

```
pperl -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
```

Τυχαίοι αριθμοί και δειγματοληψία

```
# 10 random integers in [5, 15)
pperl -E 'say join ",", map { int(rand(10)) + 5 } 1 .. 10'

# 8-character random lowercase password
pperl -E 'say join "", map { ("a".."z")[rand 26] } 1 .. 8'

# 8-character random alphanumeric
pperl -E 'say join "", map { ("a".."z", 0..9)[rand 36] } 1 .. 8'

# Random UUID (using Data::UUID)
pperl -MData::UUID -E 'say Data::UUID->new->create_str'

# Shuffle the fields of each line
pperl -MList::Util=shuffle -alne 'print join " ", shuffle @F' data.
↪txt

# Pick one random line from a file (reservoir sample, size 1)
pperl -ne 'rand($.) < 1 && ($pick = $_); END { print $pick }' file.
↪txt
```

Το κόλπο του ταμιευτήρα είναι η σωστή απάντηση όταν το αρχείο είναι πολύ μεγάλο για slurp ή ο αριθμός γραμμών του είναι άγνωστος - κάθε γραμμή έχει ίση πιθανότητα να είναι αυτή που τυπώνεται, με ακριβώς ένα πέρασμα.

Παραγοντικά, ΜΚΔ, ΕΚΠ, πρώτοι αριθμοί

```
# Factorial
pperl -E '$f = 1; $f *= $_ for 1 .. 20; say $f'
pperl -MMath::BigInt -E 'say Math::BigInt->new(100)->bfac'

# GCD / LCM (built into Math::BigInt under bgcd / blcm)
pperl -MMath::BigInt=bgcd -anle 'print bgcd(@F)' nums.txt
pperl -MMath::BigInt=blcm -anle 'print blcm(@F)' nums.txt

# Euclid's algorithm for two numbers
pperl -le '$n=20; $m=35; ($m,$n) = ($n, $m % $n) while $n; print $m'
```

Έλεγχος πρώτου αριθμού:

```
# Regex-based primality test (Abigail). Input: one integer per line.
pperl -lne '(1 x $_) !~ /^1?$|^(11+?)\1+$/ && print "$_ prime"'␣
↪nums.txt

# Reliable for large numbers: use Math::Prime::Util
pperl -MMath::Prime::Util=is_prime -lne '
    print "$_ prime" if is_prime($_)
' nums.txt
```

Το κόλπο με regex είναι περιέργεια· το `Math::Prime::Util` είναι η σοβαρή απάντηση.

IP ↔ ακέραιος

```
# Dotted-quad → 32-bit integer
pperl -MSocket -le 'print unpack "N", inet_aton "127.0.0.1"'
# 2130706433

# Integer → dotted-quad
pperl -MSocket -le 'print inet_ntoa pack "N", 2130706433'
# 127.0.0.1
```

Το `unpack "N"` διαβάζει τέσσερα bytes ως big-endian μη προσημασμένο ακέραιο 32 bits. Το αντίστοιχο `pack "N"` γράφει έναν.

Ημερομηνία και ώρα

Σήμερα, τώρα, epoch

```
pperl -E 'say time'                                    # epoch seconds
pperl -E 'say scalar localtime'                        # Mon Apr 22␣
↪14:31:07 2026
pperl -E 'say scalar gmtime'                           # same, UTC
pperl -MPOSIX=strftime -E 'say strftime "%F %T", localtime'
                                                       # 2026-04-22␣
↪14:31:07
```

Το `strftime` είναι ο καθαρός τρόπος μορφοποίησης· το `localtime` σε περιβάλλον λίστας επιστρέφει `(sec, min, hour, mday, mon, year, wday, yday, isdst)`.

Αριθμητική ημερομηνιών

Το `POSIX::mktime` κανονικοποιεί πεδία εκτός εύρους (ο μήνας 13 γίνεται μήνας 1 του επόμενου έτους, η ημέρα 32 γίνεται ημέρα 1 του επόμενου μήνα κ.λπ.), οπότε η αριθμητική ημερομηνιών σημαίνει «αφαιρώ N από ένα πεδίο, mktime, strftime».

```
# Yesterday's date
pperl -MPOSIX=strftime -MPOSIX=mktime -E '
    my @t = localtime;
    $t[3] -= 1;
    say strftime "%F", localtime mktime @t
'

# 30 days ago
pperl -MPOSIX=strftime -MPOSIX=mktime -E '
    my @t = localtime;
    $t[3] -= 30;
    say strftime "%F", localtime mktime @t
'

# First Monday of next month
pperl -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
'
```

Για οτιδήποτε πιο σύνθετο (ζώνες ώρας, αριθμητική εργάσιμων ημερών, ISO εβδομάδες), καταφύγετε στο `DateTime`:

```
pperl -MDateTime -E '
    my $d = DateTime->now->subtract(days => 30);
    say $d->strftime("%F %T %Z")
'
```

Χρονικές σφραγίδες σε αρχεία καταγραφής

```
# Sum the durations (last field, milliseconds) of lines matching
 ↪"slow"
pperl -lane '$s += $F[-1] if /slow/; END { print $s, " ms total" }'␣
 ↪app.log

# Count events per hour (assumes ISO timestamp first field)
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
pperl -lane '
    (my $hour = $F[0]) =~ s/:\d{2}:\d{2}.*//;
    $h{$hour}++;
    END { print "$_ $h{$_}" for sort keys %h }
' events.log
```

CSV και TSV

TSV with no embedded tabs: `-F'\t'` is fine. Real CSV needs `Text::CSV`:

```
# Sum the third column of a properly quoted CSV
pperl -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
pperl -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
```

Η επιλογή `binary => 1` κρατά τα bytes ανέπαφα· η επιλογή `eol => "\n"` κάνει το `print` να προσαρτά χαρακτήρες αλλαγής γραμμής.

JSON

Το `JSON::PP` διανέμεται με την Perl· το `Cpanel::JSON::XS` είναι ταχύτερο, εφόσον είναι εγκατεστημένο.

```
# Extract the "name" field from each line of NDJSON
pperl -MJSON::PP -lne '
    my $o = JSON::PP->new->decode($_);
    print $o->{name};
' events.ndjson

# Convert each line of TSV to JSON
pperl -MJSON::PP -F'\t' -lane '
    print JSON::PP->new->encode({ id => $F[0], name => $F[1] })
' data.tsv

# Pretty-print a single JSON document
pperl -MJSON::PP -0777 -ne '
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    print JSON::PP->new->pretty->encode(JSON::PP->new->decode($_))
' small.json
```

Για μεγάλο JSON, οι ροηγούμενοι αναλυτές (`JSON::Streaming::Reader`) υπερτερούν του slurping· αρχεία ενός εγγράφου κάτω από μερικές εκατοντάδες megabytes εξυπηρετούνται καλά με τα παραπάνω.

Base64, κωδικοποίηση URL, hex

```
# Base64 encode/decode (of a whole file)
pperl -MMIME::Base64 -0777 -ne 'print encode_base64($_)' file
pperl -MMIME::Base64 -0777 -ne 'print decode_base64($_)' file.b64

# URL encode/decode
pperl -MURI::Escape -lne 'print uri_escape($_)'   < urls.txt
pperl -MURI::Escape -lne 'print uri_unescape($_)' < encoded.txt

# Hex ↔ integer
pperl -E 'printf "%x\n", 255'                     # ff
pperl -E 'say hex "ff"'                           # 255

# Hex dump of a file (like xxd -p)
pperl -0777 -ne 'print unpack "H*", $_' file.bin

# Reverse: hex dump → bytes
pperl -0777 -ne 'chomp; print pack "H*", $_' file.hex > file.bin
```

Μάθετε περισσότερα

- *aliases* - τυλίξτε τις συνταγές που χρησιμοποιείτε εβδομαδιαία σε συναρτήσεις κελύφους με θεσικά ορίσματα.
- *progression* - τα ιδιώματα αναγωγής στα οποία βασίζονται αυτές οι αριθμητικές συνταγές.
- `sprintf`, `printf` - μορφοποιημένη έξοδος.
- `time`, `localtime`, `gmtime` - οι ενσωματωμένες συναρτήσεις που υπόκεινται στις συνταγές ημερομηνίας.

Από μονόγραμμο σε ψευδώνυμο κελύφους

Ένα μονόγραμμο είναι εκ σχεδιασμού αναλώσιμο. Τη στιγμή που πληκτρολογείτε το ίδιο δύο φορές, κάτι έχει κερδίσει θέση στη ρύθμιση του κελύφους σας. Το κεφάλαιο αυτό καλύπτει τη μηχανική του πώς ανυψώνεται μια συνταγή pperl σε συνάρτηση κελύφους bash ή zsh, πώς να την παραμετροποιήσετε, και πότε να σταματήσετε και να γράψετε ένα πραγματικό σενάριο αντί γι” αυτό.

Υποτιθέμενη αφετηρία: έχετε κέλυφος bash ή zsh, ένα `~/.bashrc` ή `~/.zshrc` (ή έναν κατάλογο `rc.d` / `functions` τον οποίο φορτώνει το κέλυφός σας), και την υπομονή να κλείσετε και να ξανανοίξετε ένα τερματικό μετά την επεξεργασία του.

Ψευδώνυμο έναντι συνάρτησης

Το bash και το zsh έχουν δύο μηχανισμούς. Δεν είναι εναλλάξιμοι.

- Το `alias` είναι κειμενική αντικατάσταση. Εκτελείται στην αρχή μιας γραμμής εντολών, αντικαθιστώντας το όνομα του ψευδώνυμου με την τιμή του πριν το κέλυφος ξανα-αναλύσει. Δεν δέχεται ορίσματα, δεν παρουσιάζει ιδιαιτερότητες σωλήνωσης πέρα από αυτό που σας δίνει η αντικατάσταση.
- Μια συνάρτηση κελύφους είναι ένα ονοματισμένο κομμάτι κώδικα κελύφους. Δέχεται θεσικά ορίσματα ($1, $2, $@), διαβάζει το stdin όπως κάθε άλλη εντολή και μπορεί να σωληνωθεί προς και από άλλες εντολές.

Εμπειρικός κανόνας. Αν η συνταγή δεν δέχεται ορίσματα, αρκεί το ψευδώνυμο. Αν χρειάζεται οποιοδήποτε όρισμα (αριθμό στήλης, regex, όνομα αρχείου), χρησιμοποιήστε συνάρτηση.

Ψευδώνυμο - η συνταγή χωρίς ορίσματα

```
# In ~/.bashrc
alias sum-stdin='pperl -MList::Util=sum -lane '"'"'print sum @F'"'"'
alias count-lines='pperl -lne '"'"'END { print $. }'"'"'
```

Τα εισαγωγικά είναι άσχημα γιατί μια συμβολοσειρά μονών εισαγωγικών στο bash δεν μπορεί να περιέχει μονό εισαγωγικό· το ιδίωμα `'"'"'` κλείνει και ξανανοίγει. Αυτός είναι ο κύριος λόγος που προτιμώνται οι συναρτήσεις: σας επιτρέπουν να εισαγάγετε τον κώδικα Perl μία φορά, καθαρά.

Συνάρτηση - η συνηθισμένη περίπτωση

```
# In ~/.bashrc
sum-stdin() { pperl -MList::Util=sum -lane 'print sum @F'; }
count-lines() { pperl -lne 'END { print $. }'; }
```

Η χρήση είναι η ίδια:

```
$ printf '1 2 3\n4 5 6\n' | sum-stdin
6
15
```

Παραμετροποιημένες συναρτήσεις

Το κέλυφος παρεμβάλλει τα $1, $@, "$1" στο σώμα της συνάρτησης πριν την εκτελέσει. Μέσα στο πρόγραμμα Perl, το $1 σε επίπεδο κελύφους εμφανίζεται ως κυριολεκτική τιμή - το ίδιο το $1 της Perl (η σύλληψη regex) μένει ανέπαφο από την παρεμβολή του κελύφους μόνο αν χρησιμοποιήσετε μονά εισαγωγικά.

Το μείγμα επιλύεται με προσοχή. Δύο ιδιώματα καλύπτουν σχεδόν κάθε περίπτωση.

Ιδίωμα 1: στήλη / τιμή με παρεμβολή κελύφους

Χρησιμοποιήστε διπλά εισαγωγικά γύρω από το πρόγραμμα Perl ώστε το κέλυφος να επεκτείνει το $1, και έπειτα διαφεύγετε από το ίδιο το $ της Perl με \$.

```
# 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
```

Κάθε \$ είναι κυριολεκτικό σύμβολο δολαρίου για την Perl· κάθε $1 (χωρίς διαφυγή) είναι το πρώτο θεσικό όρισμα που επεκτείνει το κέλυφος.

Ιδίωμα 2: πρόγραμμα σε μονά εισαγωγικά, δεδομένα μέσω περιβάλλοντος

Η Perl σε μονά εισαγωγικά παραμένει ευανάγνωστη και οι τιμές φτάνουν στο πρόγραμμα μέσω του %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
```

Έτσι το $1 (σύλληψη Perl) παραμένει χρησιμοποιήσιμο και αποφεύγεται ο καταρράκτης ανάστροφων καθέτων.

Ιδίωμα 3: ανάμειξη και των δύο

```
# Replace column N's commas with semicolons, in place: csv-fix N␣
↪FILE...
csv-fix() {
    local col=$1
    shift
    col="$col" pperl -i.bak -F'\t' -lane '
        my $c = $ENV{col};
        $F[$c] =~ tr/,/;/;
        print join("\t", @F);
    ' "$@"
}
```

Ο αριθμός στήλης φτάνει στο `$ENV{col}` - χωρίς χορό διαφυγών. Η λίστα αρχείων διαδίδεται μέσω του `"$@"` ώστε τα κενά στα ονόματα αρχείων να επιβιώνουν.

Πρακτικά ψευδώνυμα που αξίζει να κρατηθούν

Σχολιασμένη λίστα συνταγών που κερδίζουν τη θέση τους σε ρύθμιση κελύφους για όποιον επεξεργάζεται κείμενο στη γραμμή εντολών καθημερινά.

`sumcol N` - άθροισμα στήλης N (χωρισμένης με λευκούς χαρακτήρες)

```
sumcol() {
    pperl -MList::Util=sum -lane "\$s += \$F[$1]; END { print \$s }"
}
```

`sumcolf N FS` - άθροισμα στήλης N με προσαρμοσμένο διαχωριστή πεδίων

```
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 / μέσος όρος / πλήθος για τη στήλη 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` - μοναδικότητα σε ροή, διατηρώντας την πρώτη εμφάνιση

```
uniq-stream() { pperl -ne 'print unless $seen{$_}++'; }
```

Σε αντίθεση με το `sort -u`, η σειρά διατηρείται. Σε αντίθεση με το `uniq`, η είσοδος δεν χρειάζεται να είναι ταξινομημένη.

`extract PAT` - τυπώνει κάθε αντιστοιχία regex, μία ανά γραμμή

```
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` - τυπώνει τα πάντα μεταξύ regex A και regex B, συμπεριλαμβανομένων

```
between() {
    a="$1" b="$2" pperl -ne 'print if /$ENV{a}/ .. /$ENV{b}/'
}

$ between 'BEGIN cert' 'END cert' < bundle.pem
```

`json-pretty` - όμορφη εκτύπωση ενός εγγράφου JSON

```
json-pretty() {
    pperl -MJSON::PP -0777 -ne '
        print JSON::PP->new->pretty->canonical->encode(
            JSON::PP->new->decode($_)
        )
    '
}
```

`passwd-json` - το /etc/passwd ως 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` - μέτρηση συχνοτήτων λέξεων, ταξινομημένη

```
words-by-freq() {
    pperl -ne '
        $w{lc $_}++ for /\b[[:alpha:]]+\b/g;
        END { print "$w{$_} $_\n" for sort { $w{$b} <=> $w{$a} }␣
↪keys %w }
    '
}
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
$ words-by-freq < essay.txt | head -20
```

`strip-comments LANG` - αφαίρεση σχολίων για μερικές κοινές γλώσσες

```
strip-comments() {
    case "$1" in
        sh|py|rb|pl) pperl -pe 's/(?<!\\)#.*//' ;;
        c|cpp|rs|java|js) pperl -0777 -pe 's{//[^\n]*|/\*.*?\*/}{}gs
↪' ;;
        sql|lua) pperl -pe 's/--[^\n]*//' ;;
        *) echo "unknown language: $1" >&2; return 1 ;;
    esac
}
```

Πότε να σταματήσετε και να γράψετε σενάριο

Η μετάβαση μονόγραμμο → ψευδώνυμο τελειώνει όταν:

- Το πρόγραμμα χρειάζεται κλάδο `if` / `else` μεγαλύτερο από έναν μεμονωμένο υπό συνθήκη τελεστή `? :`.
- Υπάρχουν πάνω από δύο-τρία αρθρώματα `-M`.
- Βρίσκετε τον εαυτό σας να αποσφαλματώνει μέσω δοκιμής και λάθους στη γραμμή εντολών.
- Το πρόγραμμα είναι το μόνο ψευδώνυμο στη ρύθμισή σας με σχόλιο από πάνω που εξηγεί τι κάνει.
- Ένας συνάδελφος πρόκειται να το τρέξει.

Μεταφερθείτε σε αρχείο σεναρίου. Βάλτε τις σημαίες στη γραμμή shebang και αφήστε το πρόγραμμα να διαβάζεται φυσικά:

```
#!/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];
}
print "$total\n";
```

Δώστε του ένα λογικό όνομα, `chmod +x`, και προχωρήστε. Το μονόγραμμο έκανε τη δουλειά του: ήταν η συντομότερη διαδρομή από το «έχω μια ιδέα» στο «ξέρω ότι αυτό δουλεύει»· το σενάριο είναι η διαδρομή από το «ξέρω ότι αυτό δουλεύει» στο «ο συνάδελφός μου μπορεί να το διαβάσει».

Μάθετε περισσότερα

- *progression* - οι συνταγές που τυλίγουν αυτά τα ψευδώνυμα.
- *gotchas* - εισαγωγικά μέσα σε συναρτήσεις κελύφους και ψευδώνυμα.
- *numeric* - περισσότεροι υποψήφιοι για τη λίστα ψευδωνύμων, ειδικά αυτοί της αριθμητικής ημερομηνιών.

Παγίδες και εκπλήξεις

Οι τρόποι με τους οποίους τα μονογραμμικά του pperl κάνουν σιωπηρά κάτι διαφορετικό από αυτό που εννοούσε ο χρήστης του κελύφους. Κάθε ενότητα ονοματίζει μια παγίδα, δείχνει το σύμπτωμα της και δίνει τη διόρθωση. Διαβάστε το συνεχόμενα μία φορά για να εμπεδώσετε τα μοτίβα· επιστρέψτε όταν μια συνταγή συμπεριφέρεται περίεργα.

Η κόλαση των εισαγωγικών

Η μεγαλύτερη μεμονωμένη πηγή χαμένου χρόνου σε δουλειά με μονογραμμικά.

Τα μονά εισαγωγικά είναι ο φίλος σας

Στα bash και zsh, το κείμενο μέσα σε μονά εισαγωγικά δεν παρεμβάλλεται. Καμία επέκταση `$VAR`, καμία εκτέλεση με ανάστροφους χαρακτήρες, καμία αντικατάσταση ιστορικού `!` (το zsh δεν το κάνει· το bash το κάνει, και δαγκώνει). Χρησιμοποιείτε μονά εισαγωγικά για προγράμματα Perl εκτός αν έχετε λόγο να μην το κάνετε.

```
# Correct
pperl -ne 'print if /$var/' file              # Perl sees literal
 ↪$var

# Trap
pperl -ne "print if /$var/" file              # shell expands $var␣
 ↪before pperl sees it
```

Η έκπληξη ! σε διαδραστικό bash

```
$ pperl -E 'say "hello!"'
bash: !": event not found
```

Η επέκταση ιστορικού του bash πυροδοτείται σε `!` χωρίς εισαγωγικά (και σε μερικές μορφές με διαφυγή). Λύσεις:

- Το `set +H` στο `~/.bashrc` απενεργοποιεί την επέκταση ιστορικού.
- Διαφύγετε το `!` όταν γράφετε την εντολή: `"hello\!"`.
- Διατηρήστε τη συμβολοσειρά μονών εισαγωγικών του κελύφους γύρω από το πρόγραμμα - το `!` είναι αδρανές μέσα σε `'...'` εφόσον η επέκταση ιστορικού είναι απενεργοποιημένη.

Ένα μονό εισαγωγικό μέσα σε συμβολοσειρά μονών εισαγωγικών

Το bash δεν έχει διαφυγή γι" αυτό. Η λύση είναι ο χορός κλεισίματος-ανοίγματος τεσσάρων χαρακτήρων: το `'\''` διαβάζεται ως κλείσιμο-εισαγωγικού, κυριολεκτικό-εισαγωγικό, άνοιγμα-εισαγωγικού.

```
# Want Perl to see:  print "it's here"
pperl -e 'print "it'\''s here\n"'
```

Για οποιοδήποτε πρόγραμμα με περισσότερες από μία ενσωματωμένες αποστρόφους, εγκαταλείψτε το `'...'` και χρησιμοποιήστε `q/.../` μέσα σε διπλά εισαγωγικά:

```
pperl -e "print q/it's here/, qq/\n/"
```

ή μεταφέρετε το πρόγραμμα σε αρχείο.

Ο καταρράκτης διαφυγών $ υπό διπλά εισαγωγικά

Αν εισάγετε το πρόγραμμα Perl σε διπλά εισαγωγικά (επειδή θέλετε το `$1` να παρεμβληθεί από το κέλυφος), οι ίδιες οι μεταβλητές της Perl χρειάζονται `\$`:

```
col=3
pperl -lane "\$s += \$F[$col]; END { print \$s }"      # works
```

Αυτό είναι ευανάγνωστο μία φορά. Μετά το τρίτο τέτοιο μονόγραμμο στη σειρά, μεταφερθείτε σε μεταβλητές περιβάλλοντος:

```
col=3
col=$col pperl -lane '$s += $F[$ENV{col}]; END { print $s }'
```

Δείτε *aliases#parametrised-functions*.

Locale και κωδικοποίηση

Το προεπιλεγμένο I/O του pperl είναι bytes. Αν η είσοδός σας είναι UTF-8, η Perl εξακολουθεί να βλέπει συμβολοσειρά bytes μέχρι να ζητήσετε διαφορετικά.

Το σύμπτωμα

```
$ echo 'héllo' | pperl -lne 'print length'
6                                          # "h" + 2 bytes for é
 ↪+ 3 more = 6
$ echo 'héllo' | pperl -CSD -lne 'print length'
5                                          # correct
```

Η διόρθωση είναι το `-C` (δείτε *switches*):

- `-CSD` - STDIN/STDOUT/STDERR και ανοίγματα αρχείων όλα UTF-8.
- `-CS` - μόνο stdio.
- `-CA` - αντιμετωπίζει το `@ARGV` ως UTF-8.

Αντιστοίχιση regex σε UTF-8

Χωρίς `-C`, το `\w` ταιριάζει μόνο με χαρακτήρες λέξης ASCII. Με `-C`, το `\w` ταιριάζει με γράμματα/ψηφία Unicode όπως ορίζονται από τους πίνακες ιδιοτήτων Unicode της Perl.

```
# Count "word" characters in a multilingual file
pperl -CSD -lne '$c += () = /\w/g; END { print $c }' file.txt
```

Mojibake στην έξοδο

Αν το pperl διαβάζει UTF-8 καθαρά αλλά το τερματικό δείχνει αλλοιωμένους χαρακτήρες, το πρόβλημα είναι παρακάτω - πιθανώς το `LC_ALL` ή η ίδια η κωδικοποίηση του τερματικού. Τα `locale` και `tput` το διαγιγνώσκουν· αυτό δεν είναι πρόβλημα του pperl.

Επιτόπια επεξεργασία με `-i`

Πάντα `.bak` μέχρι την επαλήθευση

```
# Destructive if the pattern is wrong
pperl -i -pe 's/OLD/NEW/g' *.conf

# Safe
pperl -i.bak -pe 's/OLD/NEW/g' *.conf
# ...verify...
rm *.conf.bak
```

Ένα λάθος μοτίβο με σκέτο `-i` αντικαθιστά το περιεχόμενο του αρχείου με κάτι άχρηστο. Δεν υπάρχει ανάκτηση πλην του ελέγχου εκδόσεων. Χρησιμοποιήστε `-i.bak`, επιβεβαιώστε, διαγράψτε.

Η παγίδα του κενού αρχείου

Το `-i` γράφει ό,τι εκπέμπει το `-p` (ή το `-n` συν ρητές εκτυπώσεις). Αν το πρόγραμμα δεν τυπώσει τίποτα, το αρχείο γίνεται κενό.

```
# Wrong: -i with -n and no print - empties the file
pperl -i -ne '/KEEP/ && print' file.txt      # prints only lines␣
↪matching KEEP

# Same thing, probably what you meant: keep the matching lines
pperl -i -ne 'print if /KEEP/' file.txt
```

Και τα δύο είναι σωστά. Το θέμα: υπό `-n`, σας ανήκει κάθε byte της εξόδου.

Επιτόπια επεξεργασία σε πολλά αρχεία με προειδοποιήσεις ιδιοκτησίας

Το `-i` δημιουργεί ένα νέο αρχείο και το μετονομάζει. Στο Linux, αυτό σημαίνει ότι το νέο αρχείο κληρονομεί την ιδιοκτησία και τη μάσκα umask του καλούντος χρήστη. Ρυθμίσεις ιδιοκτησίας root που επεξεργάζονται από μη-root χρήστες θα αποτύχουν· ρυθμίσεις ιδιοκτησίας root που επεξεργάζονται από root θα διατηρήσουν το mode αλλά μπορεί

να χάσουν το πλαίσιο SELinux. Γνωρίστε το μοντέλο απειλών σας πριν στρέψετε το `-i` στο `/etc`.

Ο τελεστής «διαμάντι» <> και το μαγικό άνοιγμα

Το <> (το «διαμάντι») είναι από όπου διαβάζουν τα `-n` / `-p`. Διαβάζει από κάθε όρισμα στο `@ARGV` ως όνομα αρχείου, καταφεύγοντας στο STDIN αν το `@ARGV` είναι κενό.

Το «-» είναι το STDIN

Αν ένα όρισμα είναι `"-"`, το <> διαβάζει από το STDIN εκείνη τη στιγμή. Χρήσιμο για διαπλοκή:

```
echo prefix | pperl -ne 'print' - trailer.txt
# prints: prefix, then contents of trailer.txt
```

Μετα-χαρακτήρες στα ονόματα αρχείων (και γιατί έχουν λιγότερη σημασία από ό,τι είχαν)

Ιστορικά, το σκέτο <> αντιμετώπιζε το `">foo"` ως «άνοιξε το foo για εγγραφή» - ένα άνοιγμα δύο ορισμάτων που διάβαζε τη λειτουργία από το όνομα αρχείου. Στη σύγχρονη Perl, το <> χρησιμοποιεί εσωτερικά open τριών ορισμάτων, οπότε ονόματα αρχείων που αρχίζουν με <, >, | ή περιέχουν σωληνώσεις αντιμετωπίζονται ως απλά ονόματα αρχείων. Καμία επιφάνεια επίθεσης στα ονόματα αρχείων εισόδου με το σύγχρονο pperl.

Το όνομα αρχείου βρίσκεται στο $ARGV

```
pperl -lne 'print "$ARGV: $_" if /\bERROR\b/' *.log
```

Το `$ARGV` ενημερώνεται καθώς ανοίγει κάθε αρχείο. Το `"-"` είναι η ειδική τιμή για το STDIN.

Το $. συνεχίζει να μετρά μεταξύ αρχείων

Το `$.` δεν μηδενίζεται όταν το <> περνά στο επόμενο αρχείο. Αν θέλετε αριθμούς γραμμής ανά αρχείο:

```
pperl -lne 'close ARGV if eof; print "$ARGV:$.: $_" if /TODO/' *.pl
```

Το `close ARGV if eof` μηδενίζει το `$.` όταν τελειώνει το τρέχον αρχείο.

Παγίδες διαχωριστή εγγραφών

Το -l αφαιρεί και έπειτα αποκαθιστά - μία φορά

Το `-l` εφαρμόζει chomp στην είσοδο και ορίζει το `$\` στον διαχωριστή που έφαγε το chomp για την έξοδο. Εντάξει υπό `-n` / `-p`. Αλλά το `-l` εφαρμοσμένο δύο φορές (ας πούμε, προσθέσατε `-l` μετά από `-0`) μπερδεύει τον κόσμο:

```
# Read NUL-delimited input, output newline-terminated
pperl -0 -lpe ''                              # correct: -l sets $\
 ↪= "\n"

# But if you write -l0 Perl parses that as -l with argument 0:
pperl -l0 -pe ''                              # $\ becomes NUL -
 ↪probably NOT what you wanted
```

Βάλτε το `-l` μετά το `-0` (χωρίς όρισμα στο `-l`) όταν θέλετε το τυπικό ζεύγος.

Η λειτουργία παραγράφου διατηρεί τα τελικά κενά

Το `-00` (λειτουργία παραγράφου) ορίζει `$/ = ""`. Η Perl διαβάζει μέχρι και συμπεριλαμβανομένου του διαχωριστή κενής γραμμής. Χωρίς `-l`, ο διαχωριστής μένει προσαρτημένος σε κάθε παράγραφο· τυπώνοντάς τον πάλι διατηρείται η κενή γραμμή. Με `-l`, ο διαχωριστής υφίσταται chomp και το `$\ = "\n"` προσαρτά έναν χαρακτήρα αλλαγής γραμμής στην έξοδο - οι παράγραφοι συμπτύσσονται.

```
$ 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
```

Ξέρετε ποιο από τα δύο θέλετε.

Το `-0777` και το `$.`

Υπό `-0777`, κάθε αρχείο είναι μία εγγραφή. Το `$.` αυξάνεται ανά εγγραφή, οπότε το `$.` είναι ο αριθμός των αρχείων που έχουν επεξεργαστεί, όχι ο αριθμός των γραμμών.

```
pperl -0777 -e 'while (<>) { print "file $.: length=", length, "\n"
 ↪}' *.txt
# file 1: length=1024
# file 2: length=2048
```

Η ερμηνεία του ορίσματος του `-F`

Το `-F` δέχεται regex. Εκδόσεις της Perl πριν την 5.10 απαιτούσαν να γράψετε τους οριοθέτες· το σύγχρονο pperl δέχεται και τις δύο μορφές:

```
pperl -F:      -lane '...'                    # bare: Perl quotes it
pperl -F/:/    -lane '...'                    # explicit delimiters
pperl -F'\t'   -lane '...'                    # tab - regex form
```

Το συνηθισμένο λάθος είναι ένας κυριολεκτικός στηλοθέτης σε κέλυφος όπου το τερματικό τον κατάπιε:

```
pperl -F'        ' -lane '...'                    # fragile: tab may␣
↪be a space in your paste
```

Write `-F'\t'` unless you are pasting from known-good source.

Προτεραιότητα τελεστών γύρω από το print

Το `print` έχει χαμηλότερη προτεραιότητα από τους περισσότερους τελεστές. Αυτό σκοντάφτει όσους έχουν λάθος αντανακλαστικό από awk για την Perl:

```
# Wrong: prints into a filehandle named "..."
pperl -ne 'print "out.txt" $_'

# Right: parentheses or commas
pperl -ne 'print $_, "\n"'
pperl -ne 'print("foo\n")'

# Wrong: the binary || applies to print's argument list
pperl -ne 'print if $_ || "never"'            # works but obscure

# One particular case worth memorising:
pperl -ne 'print (1+1)*2'                     # prints 2, not 4
# ^ print(1+1) is the function call; *2 is applied to its return␣
↪value
```

Όταν αμφιβάλλετε, βάλτε παρενθέσεις γύρω από τα ορίσματα του `print` ή τερματίστε το πρόγραμμα με ρητό ερωτηματικό.

Το tr δεν είναι regex

Τα `tr` / `y` δέχονται σύνολα χαρακτήρων, όχι μοτίβα. Το `tr/\d/X/` ΔΕΝ ταιριάζει με ψηφία - αντικαθιστά κυριολεκτικούς χαρακτήρες ανάστροφης-καθέτου-d.

```
pperl -pe 'tr/0-9/X/'                        # replace digits with␣
↪X (character range)
pperl -pe 's/\d/X/g'                         # replace digits with␣
↪X (regex)
```

Χρησιμοποιήστε `tr` για δουλειές κλάσης χαρακτήρων (μετατροπή πεζών-κεφαλαίων, διαγραφή, εύρη) και `s///` όταν χρειάζεστε οποιαδήποτε μηχανική regex.

Εμβέλεια BEGIN και END

Τα μπλοκ `BEGIN { ... }` και `END { ... }` είναι μέρος του προγράμματος, όχι του σώματος του βρόχου. Μεταβλητές που δηλώνονται με `my` μέσα στο `BEGIN` δεν είναι ορατές στο σώμα του βρόχου. Χρησιμοποιήστε `our` ή καθολικές μεταβλητές για κατάσταση που πρέπει να επιμείνει από το `BEGIN` στον βρόχο:

```
# Wrong: $greeting is not in scope inside the loop
pperl -ne 'BEGIN { my $greeting = "Hi"; } print "$greeting $_"'

# Right: declare without my, or use our
pperl -ne 'BEGIN { $greeting = "Hi"; } print "$greeting $_"'
```

Στην πράξη, τα περισσότερα μπλοκ BEGIN κάνουν αρχικοποίηση με παρενέργειες ($\ = "\n", ορισμό μορφής) αντί για συνδέσεις μεταβλητών.

Το die και η σύμβαση \n

Το die χωρίς τελικό \n προσαρτά " at -e line N.\n" στο μήνυμά του. Με τελικό \n, τυπώνει το μήνυμα ως έχει. Επιλέξτε σκόπιμα:

```
# Developer diagnostic - want the location
pperl -E 'die "unreachable" if $x > 100'

# User-facing message - don't leak program structure
pperl -E 'die "bad input\n" unless /\d+/'
```

Ο ίδιος κανόνας ισχύει για το warn.

Λείπει χαρακτήρας αλλαγής γραμμής στο τέλος του αρχείου

Αν η τελευταία γραμμή ενός αρχείου εισόδου δεν έχει τελικό χαρακτήρα αλλαγής γραμμής, το -l εξακολουθεί να εφαρμόζει chomp σε ό,τι υπάρχει εκεί (τίποτα), και το $_ εξακολουθεί να στερείται \n. Το -p το ξανατυπώνει χωρίς τελικό χαρακτήρα αλλαγής γραμμής, και η επόμενη γραμμή εξόδου του κελύφους τρέχει πάνω της. Σπάνια πρόβλημα ορθότητας· συχνά πρόβλημα εμφάνισης.

```
pperl -pe '' no-final-newline.txt              # output lacks␣
↪trailing \n
```

Μάθετε περισσότερα

- *switches* - οι μηχανικές επεκτάσεις κάθε σημαίας που αναφέρεται εδώ.
- perlvar - ο πλήρης ορισμός κάθε ειδικής μεταβλητής από την οποία εξαρτώνται οι παραπάνω παγίδες ($/, $\, $., $ARGV, @ARGV).
- perlop - ο τελεστής «διαμάντι», το tr, και οι μορφές εισαγωγικών που αναφέρονται παραπάνω.

4.8 Δέσιμο μεταβλητών με tie

Μια μεταβλητή με **tie** μοιάζει με συνηθισμένο βαθμωτό, πίνακα, hash ή filehandle, αλλά κάθε ανάγνωση, εγγραφή και επανάληψη πάνω της δρομολογείται εκ νέου μέσα από μεθόδους που γράφετε εσείς. Η μεταβλητή είναι το δημόσιο πρόσωπο· μια κλάση δικού σας σχεδιασμού είναι ο μηχανισμός από πίσω. Η ανάθεση σε ένα βαθμωτό με tie καλεί το δικό σας STORE· η ανάγνωση ενός κλειδιού hash με tie καλεί το δικό σας FETCH· η

εκτέλεση του `keys` πάνω σε ένα hash με tie οδηγεί τα δικά σας `FIRSTKEY` και `NEXTKEY`. Ο καλών γράφει απλή Perl και δεν βλέπει ποτέ την κλάση.

Αυτή η έμμεση δρομολόγηση είναι που κάνει το tie ισχυρό. Ένα hash μπορεί να είναι μια πρόσοψη πάνω από μια βάση δεδομένων στον δίσκο. Ένα βαθμωτό μπορεί να επαναϋπολογίζει τον εαυτό του σε κάθε ανάγνωση. Ένα filehandle μπορεί να συγκεντρώνει σε μια συμβολοσειρά οτιδήποτε εκτυπώνεται σε αυτό. Ο κώδικας που χρησιμοποιεί τη μεταβλητή δεν αλλάζει· αλλάζει μόνο η κλάση από πίσω.

Αυτός ο οδηγός αφορά τη **συγγραφή της κλάσης** - το μέρος που το `perltie` αποκαλεί υλοποίηση. Η πλευρά των ενσωματωμένων (`tie`, `tied`, `untie`) και τα πλήρη μενού μεθόδων ανά τύπο μεταβλητής τεκμηριώνονται ήδη στη σελίδα αναφοράς του `tie`. Αυτός ο οδηγός παραπέμπει εκεί για τις υπογραφές και επικεντρώνεται σε ό,τι δεν διδάσκουν εκείνες οι σελίδες: πώς να φτιάξετε μια λειτουργική κλάση tie, τι πρέπει να επιστρέφει κάθε μέθοδος, και ακριβώς πότε την καλεί ο χρόνος εκτέλεσης.

4.8.1 Ποιο κεφάλαιο χρειάζεστε

Κάθε τύπος μεταβλητής έχει τον δικό του κατασκευαστή και το δικό του σύνολο μεθόδων. Διαβάστε το κεφάλαιο που ταιριάζει με το sigil που έχετε μπροστά σας.

- *Βαθμωτά με tie* - `TIESCALAR`, `FETCH`, `STORE`, `DESTROY`. Η μικρότερη διεπαφή tie· ξεκινήστε από εδώ για να μάθετε τη μορφή όλων των υπολοίπων.
- *Hash με tie* - `TIEHASH` συν το πρωτόκολλο επανάληψης (`FIRSTKEY` / `NEXTKEY`), `EXISTS`, `DELETE`, `CLEAR` και `SCALAR`. Ο πλουσιότερος από τους τύπους δεδομένων.
- *Πίνακες με tie* - `TIEARRAY`, το πρωτόκολλο μεγέθους (`FETCHSIZE` / `STORESIZE` / `EXTEND`) και οι μεταλλάκτες (`PUSH`, `POP`, `SHIFT`, `UNSHIFT`, `SPLICE`).
- *Filehandles με tie* - `TIEHANDLE` και τα hooks I/O (`PRINT`, `PRINTF`, `WRITE`, `READLINE`, `READ`, `GETC` και άλλα συναφή).

4.8.2 Πώς είναι δομημένο κάθε κεφάλαιο

Κάθε κεφάλαιο ακολουθεί την ίδια μορφή:

- **Το συμβόλαιο μέσα από παράδειγμα.** Αντί να επαναδιατυπώνει το μενού μεθόδων (η σελίδα αναφοράς του `tie` παραθέτει ήδη κάθε υπογραφή), κάθε κεφάλαιο φτιάχνει μία πλήρη, εκτελέσιμη κλάση και εξηγεί κάθε μέθοδο καθώς εμφανίζεται.
- **Χρονισμός κλήσεων.** Το μη προφανές μέρος του tie είναι *πότε* ο χρόνος εκτέλεσης επικαλείται κάθε hook - ποιος τελεστής πυροδοτεί ποια μέθοδο, σε ποιο περιβάλλον, με ποια ορίσματα. Κάθε κεφάλαιο το αναλύει αυτό.
- **Ελάχιστο βιώσιμο έναντι πλήρους.** Οι λιγότερες μέθοδοι που μπορείτε να ορίσετε και να έχετε ακόμα ένα χρηστικό tie, και έπειτα το υπόλοιπο μενού και τι σας προσφέρει η καθεμία.
- **Ο εύκολος τρόπος.** Κάθε τύπος δεδομένων συνοδεύεται από μια βασική κλάση `Tie::*` που υλοποιεί τις κουραστικές μεθόδους με όρους λίγων βασικών. Κληρονομήστε από αυτήν και υπερκαλύψτε μόνο τα ενδιαφέροντα hooks.

4.8.3 Τι κοστίζουν τα ties

Κάθε λειτουργία πάνω σε μια μεταβλητή με tie είναι μια κλήση μεθόδου. Μια αναζήτηση σε hash με tie είναι μια αποστολή FETCH, εκεί που μια απλή αναζήτηση hash είναι μία και μόνη προσπέλαση μνήμης - αισθητά πιο αργή σε έναν σφιχτό βρόχο. Το tie είναι το σωστό εργαλείο όταν η έμμεση δρομολόγηση αξίζει τον κόπο (μια μεταβλητή που στηρίζεται σε αρχείο, μια υπολογιζόμενη τιμή, μια πρόσοψη καταγραφής), όχι όταν θέλετε απλώς ένα πιο φανταχτερό hash. Για μαζική εργασία πάνω σε έναν περιέκτη με tie, διαβάστε τον σε έναν απλό πίνακα ή hash μία φορά, επεξεργαστείτε, και γράψτε τα πίσω.

4.8.4 Διαφορές από το upstream

The tie mechanism is Perl 5.44's own, so tie classes behave exactly as they do on upstream Perl.

Βαθμωτά με tie

Ένα βαθμωτό με tie είναι η μικρότερη διεπαφή tie και το καλύτερο μέρος για να μάθετε τη μορφή όλων των υπολοίπων. Ορίζετε έναν κατασκευαστή και δύο μεθόδους προσπέλασης· από εκεί και πέρα, κάθε ανάγνωση της μεταβλητής καλεί τη μία και κάθε εγγραφή καλεί την άλλη.

Η κλάση χρειάζεται τέσσερις μεθόδους, οι δύο από τις οποίες είναι προαιρετικές:

- `TIESCALAR` - ο κατασκευαστής, που καλείται μία φορά από το `tie`.
- `FETCH` - καλείται σε κάθε ανάγνωση της μεταβλητής.
- `STORE` - καλείται σε κάθε εγγραφή στη μεταβλητή.
- `DESTROY` - καλείται όταν η μεταβλητή με tie παύει να υπάρχει (προαιρετική).

Η πλήρης λίστα υπογραφών βρίσκεται στη σελίδα αναφοράς του `tie`. Αυτό το κεφάλαιο δείχνει τι κάνει κάθε μέθοδος φτιάχνοντας μία.

Μια πλήρης κλάση

Ένας μετρητής που περικόπτει τις εγγραφές ώστε η τιμή να μην πέφτει ποτέ κάτω από το μηδέν:

```
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 { }
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
}

tie my $hits, 'Counter', 10;
say $hits;              # 10
$hits = 25;
say $hits;              # 25
$hits = -7;
say $hits;              # 0
```

Το αντικείμενο πίσω από το βαθμωτό μπορεί να είναι οτιδήποτε blessed - εδώ μια αναφορά σε έναν απλό αριθμό. Η `LIST` μετά το όνομα της κλάσης στην κλήση `tie` (`10`) περνά κατευθείαν στο `TIESCALAR`.

Το συμβόλαιο των μεθόδων

`TIESCALAR classname, LIST`
: Ο κατασκευαστής. Λαμβάνει το όνομα της κλάσης και όποια επιπλέον ορίσματα πέρασαν στο `tie`. Πρέπει να επιστρέψει μια blessed αναφορά· αυτή η αναφορά γίνεται το αντικείμενο με tie και η τιμή επιστροφής της κλήσης `tie`. Μια συνηθισμένη συντόμευση είναι μια blessed αναφορά βαθμωτού που κρατά την τιμή, όπως παραπάνω, αλλά μια blessed αναφορά hash είναι εξίσου έγκυρη όταν η κλάση χρειάζεται περισσότερη κατάσταση.

`FETCH this`
: Καλείται κάθε φορά που η μεταβλητή διαβάζεται. Δέχεται μόνο το αντικείμενο. Ό,τι επιστρέφει είναι αυτό που βλέπει ο αναγνώστης. Το `FETCH` είναι το σημείο όπου ένα υπολογιζόμενο βαθμωτό αξίζει τον κόπο: μπορεί να επαναϋπολογίσει, να αναζητήσει κάτι, ή να καταγράψει την προσπέλαση και να επιστρέψει μια αποθηκευμένη τιμή.

`STORE this, value`
: Καλείται κάθε φορά που γίνεται ανάθεση στη μεταβλητή. Λαμβάνει το αντικείμενο και τη μοναδική νέα τιμή. Η τιμή επιστροφής της αγνοείται - η τιμή της έκφρασης ανάθεσης προέρχεται από ένα επακόλουθο `FETCH`, όχι από το `STORE`. Εδώ ζει η επικύρωση, η περικοπή ή η εγγραφή προς έναν υποκείμενο αποθηκευτικό χώρο.

`DESTROY this`
: Καλείται όταν η μεταβλητή με tie καταστρέφεται (έξοδος από εμβέλεια, ή πτώση της τελευταίας αναφοράς). Ορίστε την μόνο όταν η κλάση κρατά έναν πόρο που χρειάζεται απελευθέρωση - ένα ανοιχτό handle, ένα κλείδωμα. Ένας μετρητής δεν έχει τίποτα να καθαρίσει, οπότε το `DESTROY` του είναι κενό ή παραλείπεται εντελώς.

Χρονισμός κλήσεων

- Η ανάγνωση σε οποιοδήποτε περιβάλλον καλεί το `FETCH`. Τα `say $hits`, `$x = $hits + 1` και `"$hits"` πυροδοτούν το καθένα ένα `FETCH`.
- Η ανάθεση καλεί το `STORE` με την τιμή του δεξιού μέλους. Το `$hits = 25` καλεί `STORE($obj, 25)`.
- Η σύνθετη ανάθεση πρώτα διαβάζει και μετά γράφει: το `$hits += 5` καλεί `FETCH`, προσθέτει 5, και μετά `STORE`. Το ίδιο κάνει και το `$hits++`.

- Η τιμή μιας έκφρασης ανάθεσης προέρχεται από το FETCH, όχι από την τιμή επιστροφής του STORE. Μετά το `my $r = ($hits = -7)`, το `$r` είναι `0` επειδή η περικοπή έτρεξε στο STORE και η τιμή της έκφρασης ανακτήθηκε εκ νέου.

Ελάχιστη βιώσιμη κλάση

Το μικρότερο χρήσιμο βαθμωτό με tie ορίζει ακριβώς τα TIESCALAR, FETCH και STORE. Παραλείψτε το DESTROY εκτός αν δεσμεύετε κάτι. Μια κλάση με μόνο TIESCALAR και FETCH παράγει ένα βαθμωτό μόνο για ανάγνωση: το tie γίνεται κανονικά, αλλά η πρώτη ανάθεση πεθαίνει με `Can't locate object method "STORE"`.

Ο εύκολος τρόπος: κληρονομήστε από το `Tie::StdScalar`

Το `Tie::StdScalar` (που συνοδεύει το `Tie::Scalar`) υλοποιεί τα TIESCALAR, FETCH, STORE και DESTROY πάνω σε μια blessed αναφορά βαθμωτού. Κληρονομήστε από αυτό και υπερκαλύψτε μόνο το hook που θέλετε να αλλάξετε:

```
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
```

Το STORE μετατρέπει σε κεφαλαία κατά την είσοδο· το FETCH και ο κατασκευαστής προέρχονται από τη βασική κλάση αμετάβλητα.

Το ίδιο το `Tie::Scalar` είναι η *αφηρημένη* βάση - ορίζει FETCH και STORE που κράζουν, οπότε η απευθείας υποκλασοποίησή του σημαίνει ότι παρέχετε το δικό σας TIESCALAR. Καταφύγετε στο `Tie::StdScalar` όταν θέλετε έναν λειτουργικό κατασκευαστή δωρεάν· καταφύγετε στο `Tie::Scalar` μόνο όταν σκοπεύετε ούτως ή άλλως να γράψετε κάθε μέθοδο και θέλετε η διεπαφή να τεκμηριώνεται μέσω κληρονομικότητας.

Δείτε επίσης

- `tie` - η ενσωματωμένη, με το πλήρες μενού μεθόδων ανά τύπο
- `tied` - ανακτήστε το αντικείμενο που στηρίζει τη μεταβλητή για να καλέσετε πάνω του μεθόδους ειδικές της κλάσης
- `untie` - διαλύστε τη δέσμευση
- *Hash με tie* - το επόμενο βήμα: ένας κατασκευαστής συν ένα πρωτόκολλο επανάληψης
- `Tie::Scalar` - το άρθρωμα της βασικής κλάσης· το `Tie::StdScalar` είναι η συγκεκριμένη εκδοχή που συνήθως θέλετε

Hash με tie

Ένα hash με tie είναι ο πλουσιότερος από τους τύπους tie, επειδή ένα hash κάνει περισσότερα από το να διαβάζει και να γράφει μεμονωμένα στοιχεία - επαναλαμβάνει, αναφέρει το μέγεθός του, ελέγχει την ύπαρξη κλειδιών, και καθαρίζει. Καθένα από αυτά είναι ένα ξεχωριστό hook. Τα hash ήταν ο πρώτος τύπος δεδομένων της Perl που μπορούσε να δεθεί με tie (η αρχική περίπτωση χρήσης ήταν το δέσιμο ενός %hash σε ένα αρχείο DBM στον δίσκο), και η διεπαφή ακόμη αντανακλά αυτή την κληρονομιά.

Το πλήρες μενού μεθόδων βρίσκεται στη σελίδα αναφοράς του `tie`. Αυτό το κεφάλαιο φτιάχνει μία κλάση και εξηγεί κάθε hook, με προσοχή στο μέρος που μπερδεύει τον κόσμο: το πρωτόκολλο επανάληψης.

Μια πλήρης κλάση

Ένα hash του οποίου τα κλειδιά είναι χωρίς διάκριση πεζών-κεφαλαίων - τα `$h{Name}` και `$h{NAME}` αναφέρονται στην ίδια θυρίδα:

```
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 }
}

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
```

Το αντικείμενο είναι μια blessed αναφορά hash που χρησιμοποιείται ως ο πραγματικός

αποθηκευτικός χώρος· τα hooks μετατρέπουν το κλειδί σε πεζά πριν το αγγίξουν. Παρατηρήστε ότι η επανάληψη επιστρέφει τα *αποθηκευμένα* (πεζά) κλειδιά, που είναι ακριβώς αυτό που πρέπει να κάνει ένα hash χωρίς διάκριση πεζών-κεφαλαίων.

Το συμβόλαιο των μεθόδων

`TIEHASH classname, LIST`
: Ο κατασκευαστής. Επιστρέφει μια blessed αναφορά - συνήθως αλλά όχι κατ" ανάγκη μια αναφορά hash - που γίνεται το αντικείμενο με tie.

`FETCH this, key` / **`STORE this, key, value`**
: Ανάγνωση και εγγραφή ενός μόνο στοιχείου, το ίδιο ζεύγος με τα βαθμωτά αλλά με ένα όρισμα κλειδιού. Η τιμή επιστροφής του `STORE` αγνοείται.

`EXISTS this, key` / **`DELETE this, key`**
: Στηρίζουν τις ενσωματωμένες `exists` και `delete`. Η τιμή επιστροφής του `DELETE` γίνεται η τιμή επιστροφής του `delete`· για να ταιριάζει με ένα απλό hash, επιστρέψτε ό,τι θα είχε επιστρέψει το `FETCH` για εκείνο το κλειδί πριν την αφαίρεσή του.

`CLEAR this`
: Πυροδοτείται όταν ολόκληρο το hash αδειάζει, τυπικά με την ανάθεση της κενής λίστας (`%h = ()`). Αφαιρέστε τα πάντα.

`FIRSTKEY this` / **`NEXTKEY this, lastkey`**
: Το πρωτόκολλο επανάληψης - δείτε παρακάτω. Μαζί οδηγούν τα `keys`, `values` και `each`.

`SCALAR this`
: Καλείται όταν το hash αποτιμάται σε βαθμωτό ή λογικό περιβάλλον (`scalar %h`, `if (%h)`, και από την 5.28 το `keys %h` σε λογικό περιβάλλον). Επιστρέψτε μια τιμή που είναι αληθής όταν το hash δεν είναι κενό και ψευδής όταν είναι κενό.

`DESTROY this` και **`UNTIE this`**
: Προαιρετικά hooks καθαρισμού, ταυτόσημα σε ρόλο με την περίπτωση του βαθμωτού. Το `UNTIE` καλύπτεται στην παγίδα του untie παρακάτω.

Το πρωτόκολλο επανάληψης

Αυτό είναι το μέρος της διεπαφής hash που δεν έχει ανάλογο στη διεπαφή του βαθμωτού, και το μέρος που πιο συχνά πάει στραβά.

Όταν ξεκινά μια επανάληψη `keys`, `values` ή `each`, ο χρόνος εκτέλεσης καλεί το `FIRSTKEY` για να πάρει το πρώτο κλειδί, και έπειτα το `NEXTKEY` επανειλημμένα, περνώντας κάθε φορά το κλειδί που επιστράφηκε προηγουμένως, μέχρι μια μέθοδος να επιστρέψει την κενή λίστα ή `undef`. Και τα δύο καλούνται πάντα σε βαθμωτό περιβάλλον και πρέπει να επιστρέφουν απλώς ένα κλειδί· ο χρόνος εκτέλεσης καλεί ο ίδιος το `FETCH` για να πάρει κάθε τιμή.

Δύο κανόνες κρατούν την επανάληψη σωστή:

- **Το `FIRSTKEY` πρέπει να επαναφέρει κάθε εσωτερικό επαναλήπτη.** Όταν ο υποκείμενος αποθηκευτικός χώρος είναι ένα πραγματικό hash, αποτιμήστε πρώτα το `keys %$self` σε κενό ή βαθμωτό περιβάλλον - αυτό επαναφέρει τον δρομέα `each` της Perl ανά hash - και έπειτα καλέστε το `each`. Το αναλώσιμο `my $reset =`

`keys %$self` στο παράδειγμα κάνει ακριβώς αυτό. Παραλείψτε το και ένα φρέσκο `keys %h` μπορεί να συνεχίσει από όπου σταμάτησε η τελευταία επανάληψη.

- **Σηματοδοτήστε το τέλος με την κενή λίστα.** Όταν ο υποκείμενος αποθηκευτικός χώρος δεν είναι πραγματικό hash και δεν έχει δικό του each, επιστρέψτε την κενή λίστα (ή undef) μόλις παραδώσετε το τελευταίο κλειδί.

Γιατί αξίζει να ορίσετε το SCALAR

Χωρίς μια μέθοδο SCALAR, ο χρόνος εκτέλεσης μαντεύει την αλήθεια του hash σε λογικό περιβάλλον, και η εικασία μπορεί να είναι λανθασμένη - ιδίως, μπορεί να αναφέρει ένα hash ως μη κενό αμέσως μετά το άδειασμά του με επανειλημμένα DELETE. Αν το `if (%tied_hash)` έχει σημασία για τους καλούντές σας, ορίστε το SCALAR και επιστρέψτε την πραγματική απάντηση, όπως κάνει το παράδειγμα με το `scalar %$self`.

Χρονισμός κλήσεων

- Αναγνώσεις `$h{k}` → FETCH· `$h{k} = $v` → STORE.
- `exists $h{k}` → EXISTS· `delete $h{k}` → DELETE.
- `%h = ()` → CLEAR· `%h = (a => 1)` → CLEAR και μετά STORE.
- `keys %h` / `values %h` / `each %h` → FIRSTKEY μία φορά, και μετά NEXTKEY ανά βήμα. Τα keys και values σε περιβάλλον λίστας καλούν επίσης FETCH για κάθε κλειδί.
- `scalar %h`, `if (%h)` → SCALAR (αν είναι ορισμένο).
- Μια φέτα hash `@h{qw(a b)}` καλεί το FETCH μία φορά ανά κλειδί· δεν υπάρχει hook για φέτες.

Ελάχιστη βιώσιμη κλάση

Ένα hash με tie ανάγνωσης/εγγραφής χρειάζεται τουλάχιστον τα TIEHASH, FETCH και STORE. Αν οι καλούντες πρόκειται ποτέ να το επαναλάβουν, προσθέστε τα FIRSTKEY και NEXTKEY - χωρίς αυτά, τα keys και each βλέπουν ένα κενό hash. Προσθέστε τα EXISTS και DELETE αν οι καλούντες χρησιμοποιούν εκείνες τις ενσωματωμένες· χωρίς αυτά, τα exists/delete πεθαίνουν στο σημείο χρήσης. Τα SCALAR και CLEAR ολοκληρώνουν την ορθότητα για λογικούς ελέγχους και μαζικό άδειασμα.

Ο εύκολος τρόπος: κληρονομήστε από το `Tie::StdHash`

Το `Tie::StdHash` (που συνοδεύει το `Tie::Hash`) υλοποιεί ολόκληρο το μενού πάνω σε μια blessed αναφορά hash. Κληρονομήστε από αυτό και υπερκαλύψτε μόνο τα hooks που πρέπει να συμπεριφέρονται διαφορετικά:

```
package LoudHash {
    use Tie::Hash;
    our @ISA = ('Tie::StdHash');
    sub STORE { my ($self, $k, $v) = @_; $self->{$k} = uc $v }
}
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
tie my %loud, 'LoudHash';
$loud{greeting} = 'hi';
say $loud{greeting};                # HI
```

Μόνο το STORE αλλάζει· τα FETCH, EXISTS, το πρωτόκολλο επανάληψης και ο κατασκευαστής προέρχονται όλα από τη βασική κλάση. Το `Tie::Hash` (ο γονέας) είναι η αφηρημένη εκδοχή που παρέχει ένα προεπιλεγμένο TIEHASH και new αλλά αφήνει σε εσάς να γράψετε τις μεθόδους προσπέλασης· το `Tie::StdHash` είναι η συγκεκριμένη που συνήθως θέλετε.

Δείτε επίσης

- `tie` - η ενσωματωμένη και το πλήρες μενού μεθόδων hash
- `tied` - προσεγγίστε το αντικείμενο που στηρίζει για μεθόδους ειδικές της κλάσης
- `each` - ο επαναλήπτης που υλοποιούν τα FIRSTKEY / NEXTKEY· η δική του σελίδα τεκμηριώνει το συμβόλαιο τέλους επανάληψης
- `exists` και `delete` - οι ενσωματωμένες πίσω από τα EXISTS και DELETE
- *Βαθμωτά με tie* - η απλούστερη διεπαφή πάνω στην οποία βασίζεται αυτή
- `Tie::Hash` - το άρθρωμα της βασικής κλάσης· το `Tie::StdHash` είναι η συγκεκριμένη εκδοχή

Πίνακες με tie

Ένας πίνακας με tie έχει δύο στρώματα διεπαφής. Το κατώτερο στρώμα - FETCH, STORE, FETCHSIZE, STORESIZE, CLEAR - είναι υποχρεωτικό και χειρίζεται την προσπέλαση στοιχείων και την αλλαγή μεγέθους. Το ανώτερο στρώμα - PUSH, POP, SHIFT, UNSHIFT, SPLICE, DELETE, EXISTS - είναι προαιρετικό και επιτρέπει στον πίνακα να ανταποκρίνεται στις αντίστοιχες ενσωματωμένες· χωρίς ένα συγκεκριμένο hook, η αντίστοιχη ενσωματωμένη είτε καταφεύγει στο κατώτερο στρώμα είτε πεθαίνει.

Το πλήρες μενού βρίσκεται στη σελίδα αναφοράς του `tie`. Αυτό το κεφάλαιο φτιάχνει μια κλάση, και έπειτα εξηγεί το πρωτόκολλο μεγέθους και τους προαιρετικούς μεταλλάκτες, με τον χρονισμό κλήσεών τους επαληθευμένο έναντι του εκτελούμενου διερμηνευτή.

Μια πλήρης κλάση

Ένας αριθμητικός πίνακας του οποίου οι κενές θυρίδες διαβάζονται ως `0` αντί για undef, και του οποίου το STORESIZE γεμίζει τις νέες θυρίδες με `0`:

```
use v5.36;

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␣
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
↪}
    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;                  # STORESIZE grows the array, padding with␣
↪0
say "@a";                 # 11 0 33 0 0
push @a, 44;
say pop @a;               # 44
```

Το αντικείμενο είναι μια blessed αναφορά hash που περιτυλίγει έναν πραγματικό πίνακα στη θυρίδα `data` της - ένα συνηθισμένο πρότυπο, επειδή η κλάση συνήθως θέλει χώρο για κατάσταση πέρα από τον ίδιο τον πίνακα.

Το υποχρεωτικό στρώμα

`TIEARRAY classname, LIST`
: Ο κατασκευαστής· επιστρέφει μια blessed αναφορά που γίνεται το αντικείμενο με tie.

`FETCH this, index` / `STORE this, index, value`
: Ανάγνωση και εγγραφή ενός μόνο στοιχείου. Ένας αρνητικός δείκτης μεταφράζεται σε θετικό - ο χρόνος εκτέλεσης καλεί πρώτα το `FETCHSIZE` και το προσθέτει στον δείκτη - εκτός αν η κλάση θέσει το `$NEGATIVE_INDICES` σε αληθή τιμή, οπότε ο ακατέργαστος αρνητικός δείκτης φτάνει στα `FETCH` και `STORE`.

`FETCHSIZE this`
: Επιστρέφει τον αριθμό των στοιχείων. Αυτό στηρίζει το `scalar @array` και το ανώτατο όριο για το `$#array`. Είναι η μεμονωμένα πιο συχνά καλούμενη μέθοδος σε έναν πίνακα με tie, καθώς ο χρόνος εκτέλεσης χρειάζεται το μέγεθος για να

κανονικοποιήσει τους αρνητικούς δείκτες και να οριοθετήσει την επανάληψη.

STORESIZE this, count
: Ορίζει το μήκος του πίνακα. Η μεγέθυνση πρέπει να γεμίζει τις νέες θυρίδες με την έννοια του «κενού» της κλάσης (εδώ, `0`)· η συρρίκνωση πρέπει να αφαιρεί το πλεόνασμα. Αυτό στηρίζει το `$#array = N` και την αλλαγή μεγέθους που οδηγείται από ανάθεση.

CLEAR this
: Αδειάζει τον πίνακα - πυροδοτείται από το `@array = ()`.

Το πρωτόκολλο μεγέθους στην πράξη

Τα FETCHSIZE και STORESIZE είναι το σημείο όπου οι πίνακες με tie διαφέρουν περισσότερο από τα hash με tie. Τρία πράγματα να έχετε ξεκάθαρα:

- **Το `$#a` διαβάζει μέσω του FETCHSIZE και γράφει μέσω του STORESIZE.** Η ρύθμιση `$#a = 4` καλεί `STORESIZE($obj, 5)`.
- **Οι θυρίδες που παραλείπονται είναι πρόβλημα της κλάσης.** Η ανάθεση στο `$a[2]` όταν ο πίνακας έχει μήκος 1 καλεί απευθείας `STORE($obj, 2, ...)` - ο χρόνος εκτέλεσης *δεν* καλεί το STORESIZE για να γεμίσει το κενό. Αν μια ανάγνωση της θυρίδας 1 πρέπει να επιστρέφει `0` αντί για undef, το FETCH πρέπει να το πει (το `// 0` στο παράδειγμα).
- **Το EXTEND είναι συμβουλευτικό.** Όταν ο χρόνος εκτέλεσης αναμένει ότι ένας πίνακας θα μεγαλώσει
  - για παράδειγμα ακριβώς πριν από μια μεγάλη ανάθεση - μπορεί να καλέσει `EXTEND this, count` ως υπόδειξη για προδέσμευση. Είναι πάντα ασφαλές να αφήσετε το EXTEND αόριστο ή κενό· είναι καθαρά μια ευκαιρία βελτιστοποίησης, και το να το κάνετε να συμπεριφέρεται σαν STORESIZE είναι σφάλμα, επειδή ο χρόνος εκτέλεσης καλεί το EXTEND σε στιγμές που δεν θέλει στην πραγματικότητα να αλλάξει το ορατό μήκος.

Οι προαιρετικοί μεταλλάκτες

Καθένα από τα `push`, `pop`, `shift`, `unshift`, `splice`, `delete` και `exists` δρομολογεί στο ομώνυμο hook με κεφαλαία όταν η κλάση το ορίζει. Ο χρονισμός κλήσεών τους, επιβεβαιωμένος έναντι του διερμηνευτή:

- `push @a, 1, 2` → μία κλήση PUSH με ολόκληρη τη λίστα.
- `pop @a` → POP· `shift @a` → SHIFT· `unshift @a, 0` → UNSHIFT.
- `splice @a, 1, 1, 'x', 'y'` → μία κλήση SPLICE που μεταφέρει το offset, το μήκος και τη λίστα αντικατάστασης· η τιμή επιστροφής της γίνεται η τιμή του `splice`.
- `exists $a[0]` → EXISTS· `delete $a[0]` → DELETE.

Όταν ένα hook μεταλλάκτη **απουσιάζει**, η συμπεριφορά εξαρτάται από τον τελεστή. Το `Tie::Array` (παρακάτω) παρέχει εφεδρικές υλοποιήσεις για τα συνηθισμένα πέντε με όρους του υποχρεωτικού στρώματος· χωρίς εκείνη τη βασική κλάση, η κλήση ενός μη υλοποιημένου μεταλλάκτη πεθαίνει στο σημείο χρήσης.

Χρονισμός κλήσεων

- `$a[$i]` → FETCH· `$a[$i] = $v` → STORE.
- `scalar @a`, κανονικοποίηση αρνητικού δείκτη → FETCHSIZE.
- `$#a = $n`, αλλαγή μεγέθους `@a = (...)` → STORESIZE (και πρώτα CLEAR για ανάθεση ολόκληρου του πίνακα).
- push/pop/shift/unshift/splice/delete/exists → το αντίστοιχο hook με κεφαλαία όταν είναι ορισμένο.
- Μια φέτα πίνακα `@a[1, 3]` καλεί το FETCH μία φορά ανά δείκτη· δεν υπάρχει hook για φέτες.

Ελάχιστη βιώσιμη κλάση

Ο μικρότερος πίνακας με tie ορίζει τα TIEARRAY, FETCH, STORE, FETCHSIZE και STORESIZE. Με αυτά τα πέντε, η προσπέλαση στοιχείων, το `scalar @a`, το `$#a` και η επανάληψη λειτουργούν όλα. Προσθέστε το CLEAR ώστε το `@a = ()` να συμπεριφέρεται σωστά. Προσθέστε τους μεταλλάκτες μόνο για τις ενσωματωμένες που πραγματικά χρησιμοποιούν οι καλούντες - ή κληρονομήστε τους.

Ο εύκολος τρόπος: κληρονομήστε από το `Tie::Array`

Το `Tie::Array` υλοποιεί τα PUSH, POP, SHIFT, UNSHIFT και SPLICE με όρους των υποχρεωτικών πέντε, οπότε μια υποκλάση που παρέχει τα TIEARRAY, FETCH, STORE, FETCHSIZE και STORESIZE αποκτά όλους τους μεταλλάκτες λίστας δωρεάν:

```
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
```

Το `push @log, 'x', 'y'` λειτουργεί χωρίς μέθοδο PUSH επειδή η κληρονομημένη αναπτύσσεται σε δύο κλήσεις STORE - γι” αυτό το παράδειγμα προειδοποιεί δύο φορές. Το `Tie::Array` δεν ορίζει το ίδιο το TIEARRAY, οπότε η υποκλάση πρέπει να παρέχει τον κατασκευαστή· τα προεπιλεγμένα του DELETE και EXISTS απλώς κράζουν, οπότε υπερκαλύψτε τα αν οι καλούντές σας χρειάζονται τα `delete` και `exists`.

Δείτε επίσης

- `tie` - η ενσωματωμένη και το πλήρες μενού μεθόδων πίνακα
- `tied` - προσεγγίστε το αντικείμενο που στηρίζει
- `splice` - ο πιο περίπλοκος μεταλλάκτης που αντιπροσωπεύει το hook `SPLICE`
- `push` και `pop` - οι μεταλλάκτες που το `Tie::Array` παράγει από το υποχρεωτικό στρώμα
- *Hash με tie* - η άλλη πλούσια διεπαφή περιέκτη
- `Tie::Array` - το άρθρωμα της βασικής κλάσης που παρέχει τους μεταλλάκτες λίστας

Filehandles με tie

Ένα filehandle με tie δρομολογεί εκ νέου το I/O όπως ένα βαθμωτό με tie δρομολογεί εκ νέου τις αναγνώσεις και τις εγγραφές. Εκτυπώστε σε αυτό και τρέχει το δικό σας `PRINT`· διαβάστε μια γραμμή και τρέχει το δικό σας `READLINE`. Οι κλασικές χρήσεις είναι η ανακατεύθυνση της εξόδου κάπου ασυνήθιστα (σε μια συμβολοσειρά, μέσα από δίκτυο, μέσα από ένα φίλτρο) και η ενσωμάτωση της Perl σε ένα πρόγραμμα-ξενιστή που χρειάζεται ειδικό χειρισμό των `STDOUT` και `STDERR`.

Η διεπαφή είναι μεγαλύτερη από τις άλλες αλλά κατά κύριο λόγο προαιρετική: υλοποιείτε μόνο τις λειτουργίες που θα λάβει στην πραγματικότητα το handle. Το πλήρες μενού βρίσκεται στη σελίδα αναφοράς του `tie`· αυτό το κεφάλαιο φτιάχνει μια κλάση και εξηγεί τα hooks και τον χρονισμό τους.

Μια πλήρης κλάση

Ένα handle μόνο για εγγραφή που συγκεντρώνει σε μια συμβολοσειρά οτιδήποτε εκτυπώνεται σε αυτό:

```
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;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
# hello world
# 3 items
```

Το αντικείμενο είναι μια blessed αναφορά βαθμωτού που κρατά το συσσωρευμένο κείμενο. Η επιπλέον μέθοδος `buffer` προσεγγίζεται μέσω του `tied`, όχι μέσω κάποιας λειτουργίας I/O - αυτός είναι ο τυπικός τρόπος με τον οποίο μια κλάση tie εκθέτει λειτουργίες χωρίς σύνταξη filehandle.

Δέσιμο με tie του glob, όχι του βαθμωτού

Το πρώτο όρισμα στο `tie` για ένα handle πρέπει να είναι ένα glob - ξεκινά με αστερίσκο:

- Για να δέσετε με tie ένα κατονομασμένο handle: `tie *OUT, 'Collector'`.
- Για να δέσετε με tie ένα handle που κρατείται σε ένα βαθμωτό `$fh`: `tie *$fh, 'Collector'`. Το να γράψετε `tie $fh, ...` δένει το *βαθμωτό* `$fh`, όχι το handle μέσα του - ένα συνηθισμένο και μπερδευτικό λάθος.

Το συμβόλαιο των μεθόδων

`TIEHANDLE classname, LIST`
: Ο κατασκευαστής· επιστρέφει μια blessed αναφορά οποιουδήποτε είδους. Μπορεί να κρατά όποια εσωτερική κατάσταση χρειάζεται το handle.

`PRINT this, LIST`
: Στηρίζει τα `print` και `say`. Λαμβάνει την εκτυπωμένη λίστα. Τιμήστε το `$,` (τον διαχωριστή πεδίων εξόδου) ανάμεσα στα στοιχεία και το `$\` (τον τερματιστή εγγραφής εξόδου) στο τέλος αν θέλετε πλήρη πιστότητα - το `print` δεν τα εισάγει για εσάς.

`PRINTF this, format, LIST`
: Στηρίζει το `printf`. Λαμβάνει τη συμβολοσειρά μορφής και τα ορίσματα· περάστε τα εσείς οι ίδιοι μέσα από το `sprintf`.

`WRITE this, scalar, length, offset`
: Στηρίζει το `syswrite` - την ακατέργαστη, προσανατολισμένη σε ενταμιευτή εγγραφή, διακριτή από το `PRINT`.

`READLINE this`
: Στηρίζει τα `<$fh>` και `readline`. Σε βαθμωτό περιβάλλον επιστρέψτε την επόμενη γραμμή ή `undef` στο τέλος της εισόδου· σε περιβάλλον λίστας επιστρέψτε όλες τις υπόλοιπες γραμμές ή μια κενή λίστα. Οι επιστρεφόμενες συμβολοσειρές πρέπει να περιλαμβάνουν τον διαχωριστή εγγραφής εισόδου `$/`, ώστε να ταιριάζουν με ό,τι σας δίνει ένα πραγματικό handle.

`READ this, scalar, length, offset` / `GETC this`
: Στηρίζουν τα `read` / `sysread` και `getc` αντίστοιχα.

`OPEN`, `CLOSE`, `EOF`, `BINMODE`, `FILENO`, `SEEK`, `TELL`
: Στηρίζουν τις αντίστοιχες ενσωματωμένες όταν χρησιμοποιούνται πάνω στο handle. Υλοποιήστε εκείνες που έχουν νόημα για τον υποκείμενο αποθηκευτικό χώρο σας και αφήστε τις υπόλοιπες απ” έξω.

DESTROY this / UNTIE this
: Προαιρετικός καθαρισμός, ταυτόσημος σε ρόλο με τους άλλους τύπους tie. Μια κλάση handle συχνά κλείνει εδώ τον υποκείμενο πόρο της.

Χρονισμός κλήσεων

- `print FH @list` και `say FH @list` → `PRINT`. Το `say` επιπλέον εντοπίζει τοπικά το `$\` σε `"\n"` γύρω από την κλήση, οπότε ένα `PRINT` που τιμά το `$\` χειρίζεται το `say` χωρίς ειδική περίπτωση.
- `printf FH $fmt, @args` → `PRINTF`.
- `syswrite FH, $buf, $len, $off` → `WRITE`.
- `<FH>` / `readline FH` → `READLINE`, στο περιβάλλον που το περικλείει.
- `read`/`sysread` → `READ`· `getc FH` → `GETC`.
- `close FH`, `eof FH`, `binmode FH`, `seek`, `tell`, `fileno`, `open FH, ...` → το αντίστοιχο hook με κεφαλαία όταν είναι ορισμένο.

Μια σημείωση για το `STDERR`: όταν το `STDERR` είναι δεμένο με tie, το `PRINT` του είναι αυτό που εκδίδει τις προειδοποιήσεις και τα μηνύματα σφαλμάτων. Ο χρόνος εκτέλεσης απενεργοποιεί το tie για τη διάρκεια εκείνης της κλήσης, οπότε μπορείτε να καλέσετε `warn` μέσα από το `PRINT` χωρίς να πυροδοτήσετε άπειρη αναδρομή.

Ελάχιστη βιώσιμη κλάση

Δεν υπάρχει καμία μεμονωμένη υποχρεωτική μέθοδος πέρα από το `TIEHANDLE` - υλοποιείτε όποιες λειτουργίες θα λάβει το handle. Ένα handle μόνο για έξοδο χρειάζεται το `TIEHANDLE` συν το `PRINT` (και συνήθως το `PRINTF`). Ένα handle μόνο για είσοδο χρειάζεται το `TIEHANDLE` συν το `READLINE` (και ίσως το `GETC` / `READ`). Οι λειτουργίες των οποίων το hook λείπει πεθαίνουν στο σημείο χρήσης, οπότε ορίστε ακριβώς το σύνολο που χρησιμοποιούν οι καλούντές σας.

Ο εύκολος τρόπος: κληρονομήστε από το `Tie::StdHandle`

Το `Tie::StdHandle` (που συνοδεύει το `Tie::Handle`) δένει ένα handle πάνω σε ένα πραγματικό υποκείμενο: το `OPEN` του ανοίγει ένα πραγματικό αρχείο και τα hooks I/O προωθούν σε αυτό. Κληρονομήστε από αυτό και υπερκαλύψτε μόνο τις λειτουργίες που θέλετε να υποκλέψετε:

```
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
    }
}
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
my $data = "a\nb\n";
tie *DUP, 'Doubler';
tied(*DUP)->OPEN('<', \$data);
print while defined($_ = <DUP>);
# a
# a
# b
# b
```

Το READLINE καλεί προς τα πάνω τη βασική κλάση για την πραγματική επόμενη γραμμή, και έπειτα τη διπλασιάζει· κάθε άλλη λειτουργία ρέει μέσα από το Tie::StdHandle αμετάβλητη. Το Tie::Handle (ο γονέας) είναι η αφηρημένη βάση που τεκμηριώνει τη διεπαφή και παρέχει ένα προεπιλεγμένο PRINT/PRINTF με όρους του WRITE· το Tie::StdHandle είναι η συγκεκριμένη, έτοιμη προς χρήση.

Δείτε επίσης

- tie - η ενσωματωμένη και το πλήρες μενού μεθόδων handle
- tied - ο μόνος τρόπος να προσεγγίσετε τις μη-I/O μεθόδους μιας κλάσης handle, όπως το buffer παραπάνω
- print και printf - οι ενσωματωμένες πίσω από τα PRINT και PRINTF
- readline - η σελίδα του τεκμηριώνει το συμβόλαιο επιστροφής βαθμωτού/λίστας που πρέπει να τιμά το READLINE
- *Βαθμωτά με tie* - ο απλούστερος τύπος tie, ένα καλό προθέρμασμα για τη διεπαφή handle
- Tie::Handle - το άρθρωμα της βασικής κλάσης· το Tie::StdHandle είναι η συγκεκριμένη εκδοχή

4.9 Ερχόμενοι από μια άλλη γλώσσα

Ξέρετε ήδη να προγραμματίζετε. Φτάνετε σε ένα hash, σε έναν βρόχο, σε μια συνάρτηση, σε ένα regex χωρίς να το σκέφτεστε - σε κάποια άλλη γλώσσα. Αυτοί οι οδηγοί διδάσκουν την Perl αντιστοιχίζοντας αυτό που ήδη κάνετε με το πώς το κάνει η Perl, μία κατασκευή τη φορά. Το λεξικό σας είναι ένα hash της Perl· εδώ είναι η σύνταξη, και εδώ είναι το ένα σημείο όπου η αναλογία στάζει.

Κάθε οδηγός προϋποθέτει ότι είστε **ευχερείς στη γλώσσα προέλευσης και νέοι στην Perl**. Δεν σας ξαναδιδάσκει τη γλώσσα προέλευσης. Διδάσκει την πλευρά της Perl σε κάθε ζευγάρωμα: τη σύνταξη, το ιδίωμα, και - το πιο χρήσιμο απ” όλα - την παγίδα που κρύβει η επιφανειακή ομοιότητα.

Κάθε οδηγός σε αυτή τη σειρά καλύπτει τα **ίδια θέματα με την ίδια σειρά**, ώστε η αντιστοίχιση να είναι συγκρίσιμη μεταξύ γλωσσών και η σειρά να επεκτείνεται καθαρά καθώς προστίθενται νέες γλώσσες. Η σειρά κινείται από αυτά που μοιράζεται κάθε γλώσσα (μεταβλητές, συμβολοσειρές, αριθμοί), μέσα από τα διακριτικά σχήματα της Perl (sigils, ο διαχωρισμός array/hash, και το **περιβάλλον** - η μία ιδέα που δεν έχει καμία άλλη γλώσσα), προς τη δομή (υπορουτίνες, αναφορές, αρθρώματα, αντικείμενα), και

τέλος μέσα από την επιφάνεια εργασίας (regex, I/O, χειρισμός σφαλμάτων, ιδιώματα), καταλήγοντας σε μια ανά γλώσσα συλλογή των παγίδων που πραγματικά δαγκώνουν.

4.9.1 Ποιον οδηγό θέλετε

- *Από Python σε Perl* - sigils εκεί που η Python δεν έχει κανένα, ο διαχωρισμός array/hash εκεί που η Python έχει `list` και `dict`, comprehensions ξαναγραμμένα ως `map`/`grep`, και η έννοια του περιβάλλοντος για την οποία η Python δεν έχει λέξη.
- *Από PHP σε Perl* - έχετε ήδη το `$`, αλλά στην Perl αλλάζει ανάλογα με την προσπέλαση, ο ένας τύπος `array` της PHP χωρίζεται σε `@array` και `%hash`, και τα `.` έναντι `+` (και `==` έναντι `eq`) παύουν να είναι εναλλάξιμα.
- *Από Ruby σε Perl* - τα blocks γίνονται μπλοκ και closures της Perl, οι κλήσεις-μεθόδων-σε-όλα γίνονται απλές συναρτήσεις, και το «μόνο τα `nil` και `false` είναι ψευδή» της Ruby αντιστρέφεται στην αλήθεια της Perl.
- *Από Lua σε Perl* - ο ένας τύπος table χωρίζεται σε `@array` και `%hash`, η ευρετηρίαση γίνεται 0-based, τα μοτίβα της Lua γίνονται πραγματικά regex, και οι πολλαπλές τιμές επιστροφής είναι η γέφυρά σας προς το περιβάλλον της Perl.
- *Sed to Perl* - the same `s///` you already know, sed's `y///` becomes `tr///`, address ranges become the `/a/ .. /b/` flip-flop, and the hold space becomes an ordinary Perl variable.
- *Awk to Perl* - Perl grew out of awk, so almost everything maps directly: fields become `@F` and `split`, `NR` becomes `$.`, 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 Η μία έννοια που πρέπει να διαβάσετε πρώτη

Αν διαβάσετε μόνο μία ενότητα από όποιον οδηγό ταιριάζει στο υπόβαθρό σας, διαβάστε το **Περιβάλλον**. Κάθε άλλη γλώσσα σε αυτή τη σειρά συμπτύσσει μια συλλογή σε ένα πλήθος, σε ένα πρώτο στοιχείο, ή σε μια λογική τιμή με ρητή σύνταξη (`len(x)`, `x[0]`, `bool(x)`). Η Perl το κάνει μέσω του *περιβάλλοντος*: η ίδια έκφραση δίνει μια λίστα σε ένα σημείο και ένα μοναδικό βαθμωτό σε άλλο, ανάλογα με το πού τη γράψατε. Είναι η πηγή των περισσότερων συγχύσεων του τύπου «γιατί ο πίνακάς μου έγινε ο αριθμός 3;», και μόλις το συλλάβετε η υπόλοιπη Perl μπαίνει στη θέση της.

4.9.3 Μια σημείωση για το runtime

These guides target `pperl`, a faithful Rust reimplementation of Perl 5.44. 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.

Από Python σε Perl

Ξέρετε Python. Γράφετε list comprehensions στον ύπνο σας, φτάνετε σε ένα `dict` χωρίς να το σκέφτεστε, και το `for x in xs:` είναι μυϊκή μνήμη. Αυτός ο οδηγός αντιστοιχίζει καθεμία από αυτές τις συνήθειες στην Perl, ώστε να γίνετε παραγωγικοί στο `pperl` μέσα σε ένα απόγευμα αντί να ξαναμάθετε προγραμματισμό από το μηδέν.

Δύο σοκ φτάνουν μέσα στα πρώτα πέντε λεπτά, οπότε γνωρίστε τα τώρα:

- **Η Perl χρησιμοποιεί άγκιστρα και ερωτηματικά, όχι εσοχές.** Δεν υπάρχει σημαντικός λευκός χαρακτήρας. Ένα μπλοκ είναι `{ ... }`· μια εντολή τελειώνει με `;`. Κάντε εσοχές για τη δική σας ψυχική υγεία, αλλά ο αναλυτής τις αγνοεί.
- **Οι μεταβλητές φορούν sigils.** Ένα βαθμωτό είναι `$x`, ένας πίνακας είναι `@x`, ένα hash είναι `%x`. Το sigil είναι μέρος του πώς διαβάζετε τη μεταβλητή, όχι απλώς του ονόματός της. Αυτή είναι η μεγαλύτερη επιφανειακή διαφορά από την Python, και η πρώτη ενότητα δεν αφορά τίποτε άλλο.

Every example here runs on `pperl`, a Rust reimplementation of Perl 5.44. Each Perl snippet was executed on `pperl`; the `# ...` comments show its real output. Start your scripts with:

```
use v5.44;
```

Αυτή η μία γραμμή ενεργοποιεί το `strict` (καμία τυχαία καθολική μεταβλητή), τα `warnings`, την ενσωματωμένη `say` (ένα `print` με τελικό χαρακτήρα νέας γραμμής, όπως το `print` της Python), τις υπογραφές υπορουτινών, και το σύγχρονο σύνολο χαρακτηριστικών που χρησιμοποιείται σε όλον αυτόν τον οδηγό.

Πώς να το διαβάσετε αυτό

- *Μεταβλητές και sigils* - η μετάβαση $/@/%.
- *Συμβολοσειρές* - παρεμβολή και συνένωση.
- *Αριθμοί* - ένας επιφανειακός τύπος, διαίρεση κινητής υποδιαστολής.
- *Πίνακες και λίστες* - η `list` σας, συν map/grep.
- *Hashes* - το `dict` σας.
- *Περιβάλλον* - η έννοια για την οποία η Python δεν έχει λέξη. Διαβάστε αυτήν.
- *Αλήθεια* - τι μετράει ως ψευδές, με εκπλήξεις.
- *Ροή ελέγχου* - `if`, βρόχοι, οι μεταθεματικές μορφές.
- *Υπορουτίνες* - το `def` γίνεται `sub`.
- *Αναφορές* - εμφωλευμένα δεδομένα και closures.
- *Κανονικές εκφράσεις* - το regex ως σύνταξη, όχι ως βιβλιοθήκη.
- *I/O αρχείων και ροών* - open, αναγνώσεις γραμμών, slurp.
- *Αρθρώματα και πακέτα* - το `import` γίνεται `use`.
- *Αντικείμενα* - η σύγχρονη σύνταξη `class`.
- *Χειρισμός σφαλμάτων* - το `try`/`except` γίνεται `eval` ή `try`/`catch`.

- *Ιδιώματα* - τα καθημερινά μονόγραμμα.
- *Παγίδες ερχόμενοι από την Python* - σαρώστε τες πριν γράψετε.

Μεταβλητές και sigils

Στην Python ένα όνομα απλώς ονοματίζει μια τιμή. Στην Perl το **sigil σας λέει το σχήμα**: `$` για μία τιμή, `@` για μια διατεταγμένη λίστα, `%` για μια αντιστοίχιση κλειδιού/τιμής. Το `my` δηλώνει μια μεταβλητή με λεξιλογική εμβέλεια, το αντίστοιχο της Perl για μια συνηθισμένη τοπική σύνδεση.

```
name = "Ada"
langs = ["Python", "Perl"]
ages = {"Ada": 36, "Linus": 56}
```

```
use v5.44;
my $name = "Ada";
my @langs = ("Python", "Perl");
my %ages  = (Ada => 36, Linus => 56);
say $name;            # Ada
say $langs[0];        # Python
say $ages{Ada};       # 36
```

Η παγίδα: όταν φτάνετε *μέσα* σε ένα σύνολο για ένα στοιχείο, το sigil ακολουθεί την τιμή που παίρνετε πίσω, όχι τον περιέκτη. Ένα μεμονωμένο στοιχείο του `@langs` είναι βαθμωτό, οπότε είναι `$langs[0]`, όχι `@langs[0]`· μία τιμή από το `%ages` είναι `$ages{Ada}`. Η μορφή `@`/`%` είναι όλη η συλλογή· η μορφή `$` είναι ένα κομμάτι της.

Συμβολοσειρές

Τα διπλά εισαγωγικά παρεμβάλλουν μεταβλητές, τα μονά όχι. Δεν υπάρχει f-string επειδή η προεπιλεγμένη συμβολοσειρά *είναι* ένα f-string. Η συνένωση είναι `.`, όχι `+` (το οποίο προορίζεται για αριθμούς).

```
name = "Ada"
greeting = f"Hello, {name}"
shout = "AB" * 3
```

```
use v5.44;
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
```

Η παγίδα: η επανάληψη συμβολοσειράς είναι `x`, όχι `*`, και ο τελεστής ένωσης είναι `.`, όχι `+`. Το μπέρδεμά τους είναι το πιο συχνό τυπογραφικό λάθος της πρώτης μέρας για έναν προγραμματιστή της Python. Τα στοιχεία πίνακα και οι τιμές hash παρεμβάλλονται και μέσα σε διπλά εισαγωγικά: το `"First: $langs[0]"` δουλεύει όπως είναι γραμμένο.

Αριθμοί

Η Perl έχει έναν αριθμητικό τύπο στην επιφάνεια· δεν δηλώνετε `int` έναντι `float`. Η διαίρεση πάντα παράγει έναν αριθμό κινητής υποδιαστολής, ακριβώς όπως η Python 3.

```
print(7 / 2)        # 3.5
print(7 // 2)       # 3
print(2 ** 10)      # 1024
print(10 % 3)       # 1
```

```
use v5.44;
say 7 / 2;          # 3.5
say int(7 / 2);     # 3
say 2 ** 10;        # 1024
say 10 % 3;         # 1
```

Η παγίδα: οι τελεστές σύγκρισης και ισότητας έρχονται σε δύο οικογένειες, και επιλέγετε με βάση τον *τύπο των τελεστέων*, όχι των μεταβλητών. Οι αριθμοί χρησιμοποιούν `==`, `!=`, `<`, `>`, `<=>`· οι συμβολοσειρές χρησιμοποιούν `eq`, `ne`, `lt`, `gt`, `cmp`. Το `"10" == "10.0"` είναι αληθές (αριθμητικά), αλλά το `"10" eq "10.0"` είναι ψευδές (συμβολοσειρά). Η χρήση `==` σε συμβολοσειρές, ή `eq` σε αριθμούς, είναι ένα συχνό και σιωπηλό σφάλμα.

Πίνακες και λίστες

Ένας πίνακας της Perl είναι η `list` σας στην Python. Τα ρήματα έχουν μετονομαστεί αλλά οι λειτουργίες είναι ίδιες, και οι αρνητικοί δείκτες δουλεύουν όπως περιμένετε.

```
nums = [1, 2, 3, 4, 5]
nums.append(6)
last = nums.pop()
print(len(nums))    # 5
print(nums[1:3])    # [2, 3]
```

```
use v5.44;
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)
```

Τα comprehensions γίνονται `map` και `grep`. Το `map` μετασχηματίζει (το `[f(x) for x in xs]` της Python)· το `grep` φιλτράρει (το `[x for x in xs if p(x)]` της Python). Το `$_` είναι το τρέχον στοιχείο, η σιωπηρή μεταβλητή βρόχου της Perl.

```
squares = [x * x for x in nums]
evens   = [x for x in nums if x % 2 == 0]
desc    = sorted(nums, reverse=True)
```

```
use v5.44;
my @nums    = (1, 2, 3, 4, 5);
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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
```

Η παγίδα: το `sort` συγκρίνει ως συμβολοσειρές από προεπιλογή, οπότε το `sort (10, 9, 100)` δίνει `10 100 9`. Για αριθμούς πρέπει να περάσετε έναν συγκριτή: `sort { $a <=> $b } @nums`. Τα `$a` και `$b` είναι τα δύο στοιχεία που συγκρίνονται, και το `<=>` είναι ο αριθμητικός τρίτιμος τελεστής (το `cmp` είναι το αδελφάκι του για συμβολοσειρές).

Hashes

Ένα hash της Perl είναι το `dict` σας στην Python. Δηλώνεται με `%`, και προσπελαύνεται μία τιμή τη φορά με `$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 v5.44;
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}";
}
```

Η παγίδα: ο έλεγχος μέλους είναι `exists $h{key}`, όχι το `in` της Python, και το παχύ κόμμα `=>` είναι ο ιδιωματικός διαχωριστής κλειδιού/τιμής. Το `=>` επίσης βάζει αυτόματα εισαγωγικά στο bareword στα αριστερά του, οπότε το `apple => 3` δεν χρειάζεται εισαγωγικά γύρω από το `apple`. Ένα hash δεν έχει σειρά· επαναλάβετε με `sort keys %h` όταν θέλετε μια σταθερή.

Περιβάλλον

Αυτό είναι το κεφάλαιο που πρέπει να διαβάσετε δύο φορές. Η Python δεν έχει αντίστοιχο, και σχεδόν κάθε ιστορία «η Perl είναι παράξενη» ανάγεται σε αυτό.

Κάθε έκφραση στην Perl τρέχει είτε σε **περιβάλλον λίστας** είτε σε **βαθμωτό περιβάλλον**, που καθορίζεται από το πού εμφανίζεται, όχι από το τι είναι. Ο ίδιος πίνακας συμπεριφέρεται διαφορετικά σε καθένα. Η ανάθεση ενός πίνακα σε ένα *βαθμωτό* δίνει το μήκος του· η ανάθεσή του σε μια *λίστα* (ένα σύνολο μεταβλητών σε παρενθέσεις) αντιγράφει στοιχεία.

```
use v5.44;
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
```

Οι παρενθέσεις στα αριστερά είναι όλη η διαφορά. Το `my $x = @list` ρωτάει «πόσα;» και παίρνει 3. Το `my ($x) = @list` ρωτάει «δώσε μου τα στοιχεία» και παίρνει 10. Ένας προγραμματιστής της Python διαβάζει και τα δύο ως «ανάθεσε τη λίστα», που είναι ακριβώς η παγίδα.

Μια υπορουτίνα μπορεί να ρωτήσει σε ποιο περιβάλλον κλήθηκε, με `wantarray`, και να επιστρέψει διαφορετικά πράγματα ανάλογα:

```
use v5.44;
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
```

Οι ενσωματωμένες το χρησιμοποιούν κι αυτές. Το `scalar @array` επιβάλλει βαθμωτό περιβάλλον για να πάρει ένα πλήθος· μια αντιστοίχιση regex με `/g` επιστρέφει τη λίστα των αντιστοιχιών σε περιβάλλον λίστας και ένα αληθές/ψευδές σε βαθμωτό περιβάλλον. Το κλασικό ιδίωμα για το «μέτρα τις αντιστοιχίες» επιβάλλει μια ανάθεση λίστας σε βαθμωτό περιβάλλον με τη λεγόμενη μορφή *countof*:

```
use v5.44;
my $n = () = "a1b2c3" =~ /\d/g;
say $n;                   # 3
```

Η παγίδα: δεν υπάρχει μία μόνο απάντηση στο «τι κάνει αυτός ο πίνακας» - εξαρτάται από την περιβάλλουσα έκφραση. Όταν κάτι επιστρέφει έναν αριθμό που περιμένατε να είναι λίστα (ή το αντίστροφο), το περιβάλλον είναι ο πρώτος ύποπτος. Φτάστε στο `scalar` για να επιβάλετε ένα πλήθος και σε παρενθέσεις στα αριστερά για να επιβάλετε αντιγραφή στοιχείων.

Αλήθεια τιμών

Οι ψευδείς τιμές της Perl είναι μια σύντομη, συγκεκριμένη λίστα: ο αριθμός `0`, η κενή συμβολοσειρά `""`, η συμβολοσειρά `"0"`, και το `undef` (το None της Perl). **Όλα τα άλλα είναι αληθή** - και αυτό περιλαμβάνει κάποιες συμβολοσειρές που ένας προγραμματιστής της Python υποθέτει ότι είναι ψευδείς.

```
use v5.44;
for my $v (0, 1, "", "0", "00", "0.0", "false") {
    say "'$v' is ", ($v ? "true" : "false");
}
```

Αυτό τυπώνει, ακριβώς:

```
'0' is false
'1' is true
'' is false
'0' is false
'00' is true
'0.0' is true
'false' is true
```

Η παγίδα: τα `"00"` και `"0.0"` είναι **αληθή** στην Perl, παρόλο που μοιάζουν αριθμητικά μηδέν, επειδή μόνο η ακριβής συμβολοσειρά `"0"` είναι ψευδής. Η συμβολοσειρά `"false"` είναι αληθής (είναι μια μη κενή, μη-`"0"` συμβολοσειρά). Και σε αντίθεση με την Python, ένας κενός πίνακας ή κενό hash δεν είναι ο ίδιος μια ψευδής *τιμή* - σε λογικό περιβάλλον ένας πίνακας δίνει το μήκος του, οπότε το `if (@list)` είναι αληθές όταν η λίστα είναι μη κενή, που είναι ο έλεγχος που θέλετε. Για μια επιλογή «χρησιμοποίησε αυτό αν είναι ορισμένο, αλλιώς μια προεπιλογή», χρησιμοποιήστε το defined-or `//`, που πέφτει πίσω μόνο στο undef (όχι στο `0` ή στο `""`):

```
use v5.44;
my %cfg = (host => "localhost");
my $port = $cfg{port} // 8080;   # key absent -> undef -> 8080
say $port;                       # 8080
```

Ροή ελέγχου

Οι δεσμευμένες λέξεις είναι οικείες· η γραφή διαφέρει. Το `elif` είναι `elsif`, και δεν υπάρχει `:` ούτε εσοχή - τα μπλοκ είναι σε άγκιστρα.

```
if n < 0:
    sign = "neg"
elif n == 0:
    sign = "zero"
else:
    sign = "pos"

for x in nums:
    print(x)

while running:
    step()
```

```
use v5.44;
my $sign;
if    ($n < 0)  { $sign = "neg"  }
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
elsif ($n == 0) { $sign = "zero" }
else            { $sign = "pos"  }

for my $x (@nums) { say $x }

while ($running) { step() }
```

Η Perl προσθέτει δύο πράγματα που δεν έχει η Python. Οι **μεταθεματικοί τροποποιητές** επιτρέπουν σε μια απλή εντολή να φέρει τη δική της συνθήκη ή βρόχο, κάτι που διαβάζεται φυσικά για μονόγραμμα:

```
use v5.44;
say $_ for @nums;                # loop, one statement
say "negative" if $n < 0;        # guarded statement
$sum += $_ for @nums;            # accumulate
```

Και τα `unless`/`until` είναι οι αρνημένες μορφές των `if`/`while`, για όταν η θετική διατύπωση διαβάζεται αδέξια: `say "ok" unless $error`.

Η παγίδα: ο τριαδικός τελεστής αλυσιδώνεται όπως της Python, αλλά με `?`/`:`, όχι `if`/`else`: `my $sign = $n < 0 ? "neg" : $n == 0 ? "zero" : "pos"`. Το `?:` σε στυλ C είναι το ιδίωμα· δεν υπάρχει μορφή `x if c else y`.

Υπορουτίνες

Το `def` γίνεται `sub`. Η σύγχρονη Perl έχει υπογραφές που μοιάζουν πολύ με τις λίστες παραμέτρων της Python, συμπεριλαμβανομένων των προεπιλογών.

```
def greet(name, greeting="Hello"):
    return f"{greeting}, {name}!"

def total(*nums):
    return sum(nums)
```

```
use v5.44;
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;
    return $sum;
}
say total(1, 2, 3, 4);      # 10
```

Τα ονοματισμένα ορίσματα αντιστοιχίζονται σε μια παράμετρο hash, αφού τα ζεύγη `key => value` ισοπεδώνονται σε μια λίστα που μια υπογραφή `%opts` συγκεντρώνει:

```
use v5.44;
sub connect_to (%opts) {
    return "host=$opts{host} port=$opts{port}";
}
say connect_to(host => "localhost", port => 8080);
# host=localhost port=8080
```

Η παγίδα: μια sub επιστρέφει την τιμή της τελευταίας της έκφρασης ακόμη και χωρίς `return`, και η επιστροφή είναι **ευαίσθητη στο περιβάλλον** (δείτε *Περιβάλλον*). Αν κάνετε `return @list`, ο καλών παίρνει τη λίστα σε περιβάλλον λίστας αλλά το *πλήθος* σε βαθμωτό περιβάλλον. Να είστε εσκεμμένοι σχετικά με το τι επιστρέφετε όταν μπορεί να χρησιμοποιηθεί με οποιονδήποτε τρόπο.

Αναφορές

Οι πίνακες και τα hashes της Perl δεν εμφωλεύονται απευθείας· τα εμφωλεύετε μέσω **αναφορών**, που είναι το αντίστοιχο της Perl για τις λαβές αντικειμένων της Python (στην Python κάθε περιέκτης είναι ήδη αναφορά, οπότε αυτό το βήμα είναι εκεί αόρατο σε εσάς). Μια αναφορά είναι ένα βαθμωτό που δείχνει σε ένα σύνολο. Πάρτε μία με `\`, ακολουθήστε τη με `->`.

```
data = {"name": "Ada", "langs": ["Perl", "Python"]}
print(data["langs"][1])     # Python
data["langs"].append("Ruby")
```

```
use v5.44;
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
```

Το `{ ... }` φτιάχνει μια ανώνυμη αναφορά hash και το `[ ... ]` μια ανώνυμη αναφορά πίνακα - τα κυριολεκτικά δομικά στοιχεία των εμφωλευμένων δεδομένων, που αντιστοιχούν στις κυριολεξίες `{}` και `[]` της Python. Το βέλος `->` ακολουθεί μια αναφορά κατά ένα βήμα· ανάμεσα σε δύο δείκτες είναι προαιρετικό, οπότε τα `$data->{langs}[1]` και `$data->{langs}->[1]` είναι ίδια.

Οι ανώνυμες subs είναι closures, ακριβώς όπως το `lambda` της Python αλλά χωρίς τον περιορισμό της μίας έκφρασης:

```
use v5.44;
my $double = sub ($n) { $n * 2 };
say $double->(21);          # 42
```

Η παγίδα: οι μεταθεματικές μορφές αποαναφοράς (`->@*`, `->%*`, `->$*`) είναι ο σύγχρονος, ευανάγνωστος τρόπος να αποαναφέρετε ένα ολόκληρο σύνολο, και το `"@{ ... }"` είναι ο τρόπος να παρεμβάλετε έναν αποαναφερμένο πίνακα μέσα σε μια συμβολοσειρά. Οι παλαιότερες μορφές `@$ref` και `${$ref}{key}` εξακολουθούν να δουλεύουν και εμφανίζονται σε υπάρχοντα κώδικα, αλλά προτιμήστε το βέλος και τις μεταθεματικές μορφές σε νέο κώδικα.

Κανονικές εκφράσεις

Στην Python το regex είναι η βιβλιοθήκη `re`. Στην Perl είναι **σύνταξη**: το `=~` συνδέει ένα μοτίβο με μια συμβολοσειρά, το `m//` αντιστοιχίζει, το `s///` αντικαθιστά. Οι ομάδες σύλληψης καταλήγουν στα `$1`, `$2`, και οι ονομαστικές συλλήψεις στο `%+`.

```
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 v5.44;
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
```

Μια αντιστοίχιση `/g` σε περιβάλλον λίστας επιστρέφει *όλες* τις συλλήψεις, που είναι το άμεσο αντίστοιχο του `re.findall`:

```
use v5.44;
my @nums = "x1y2z3" =~ /(\d)/g;     # 1 2 3
say "@nums";
```

Η παγίδα: το `s///` τροποποιεί τον στόχο του *επιτόπια* και επιστρέφει τον αριθμό των αντικαταστάσεων, όχι μια νέα συμβολοσειρά. Γι" αυτό το παράδειγμα αντιγράφει πρώτα με `(my $t = "a,b,c")` και μετά αντικαθιστά στο αντίγραφο - αλλιώς θα επεξεργαζόσασταν μια κυριολεξία συμβολοσειράς. Για να πάρετε μια νέα συμβολοσειρά χωρίς να αγγίξετε την αρχική, χρησιμοποιήστε τη σημαία `/r`: `my $t = $s =~ s/,/;/gr`.

I/O αρχείων και ροών

Το ιδίωμα είναι το open με τρία ορίσματα και ένα λεξιλογικό filehandle, το άμεσο αντίστοιχο του `with open(...) as f:` της Python. Ο τελεστής με γωνιώδεις αγκύλες `<$fh>` διαβάζει μια γραμμή, με τον ίδιο ρόλο όπως η επανάληψη ενός αντικειμένου αρχείου της Python.

```
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:
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
        print("got:", line.rstrip())
```

```
use v5.44;
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;
```

Για να ρουφήξετε (slurp) όλες τις γραμμές μονομιάς, διαβάστε το handle σε περιβάλλον λίστας:

```
use v5.44;
open my $fh, '<', "/tmp/sample.txt" or die $!;
my @lines = <$fh>;          # all lines, including newlines
say scalar @lines;          # 3
```

Η παγίδα: μια γραμμή που διαβάστηκε με `<$fh>` κρατάει τον τελικό χαρακτήρα νέας γραμμής· καλέστε `chomp` για να τον αφαιρέσετε (το αντανακλαστικό `line.rstrip()` της Python). Και το `open` επιστρέφει ψευδές σε αποτυχία αντί να εγείρει εξαίρεση, οπότε το ιδίωμα `or die "...: $!"` είναι υποχρεωτικό - το `$!` είναι το μήνυμα σφάλματος συστήματος, το `errno` της Perl.

Αρθρώματα και πακέτα

Το `import` γίνεται `use`. Ο διαχωριστής χώρου ονομάτων είναι το `::`, όχι το `.`, και ένα άρθρωμα `Foo::Bar` βρίσκεται στο `Foo/Bar.pm` στη διαδρομή ένταξης `@INC` (το `sys.path` της Perl). Η λίστα εισαγωγής ενός αρθρώματος είναι μια επιλογή ονομάτων, όπως ακριβώς το `from module import name`.

```
from math import sqrt
import json
data = json.loads(text)
```

```
use v5.44;
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
```

Η τυπική βιβλιοθήκη είναι μεγάλη και πολλά από αυτά για τα οποία θα κατέφευγες στο PyPI έρχονται στον πυρήνα ή στο CPAN, το αρχείο πακέτων της Perl. Τα `List::Util`, `Scalar::Util`, `POSIX`, `File::Spec`, και `Data::Dumper` είναι τα καθημερινά.

Η παγίδα: το `use` τρέχει κατά τη *μεταγλώττιση*, όχι στο σημείο του αρχείου όπου εμφανίζεται, οπότε η σειρά των γραμμών `use` σε σχέση με τον κώδικά σας δεν έχει σημασία όπως έχει η τοποθέτηση του `import`. Η εισαγωγή είναι επίσης προαιρετική ανά όνομα: το `use List::Util qw(sum0 max)` φέρνει ακριβώς τα `sum0` και `max` στον χώρο ονομάτων σας, τίποτε άλλο.

Αντικείμενα

For new code, Perl 5.44 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).

```python
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)
```

```perl
use v5.44;
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)
```

Ο κατασκευαστής δημιουργείται για εσάς: το `:param` σημειώνει ένα πεδίο που η αυτόματα κατασκευασμένη `new` δέχεται ως ονοματισμένο όρισμα, και το `= 0` είναι η προεπιλογή του. Η κληρονομικότητα είναι `:isa`:

```
use v5.44;
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 5.44, 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*.

Χειρισμός σφαλμάτων

Το `try`/`except` της Python έχει δύο αντίστοιχα. Το κλασικό τυλίγει κώδικα σε `eval { }` και εξετάζει το `$@` (το σφάλμα που πιάστηκε) μετά:

```
try:
    risky(-1)
except Exception as e:
    print("caught:", e)
```

```
use v5.44;
my $result = eval { risky(-1) };
if ($@) {
    print "caught: $@";     # caught: negative!
}
```

Η σύγχρονη μορφή είναι το `try`/`catch`, που διαβάζεται σχεδόν ακριβώς όπως της Python:

```
use v5.44;
use feature 'try';

try {
    risky(-4);
}
catch ($e) {
    print "caught: $e";     # caught: negative!
}
```

όπου η πλευρά που εγείρει είναι το `die`:

```
use v5.44;
sub risky ($n) {
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    die "negative!\n" if $n < 0;
    return sqrt $n;
}
```

Η παγίδα: το `die` με μια συμβολοσειρά που τελειώνει σε `\n` παράγει ακριβώς αυτό το μήνυμα· το `die` με μια συμβολοσειρά που *δεν* τελειώνει σε χαρακτήρα νέας γραμμής παίρνει αυτόματα την προσθήκη `" at SCRIPT line N.\n"`. Τελειώστε τις συμβολοσειρές σφάλματός σας με `\n` όταν θέλετε ένα καθαρό μήνυμα και παραλείψτε το όταν θέλετε το ίχνος αρχείου-και-γραμμής. Σημειώστε επίσης τις δύο μεταβλητές σφάλματος: το `$@` είναι η τελευταία εξαίρεση που πιάστηκε (το `except ... as e` της Python), ενώ το `$!` είναι το τελευταίο σφάλμα *συστήματος* (ένα αποτυχημένο open, η συμβολοσειρά `errno`).

Ιδιωματισμοί

Μερικά μοτίβα που θα βλέπετε καθημερινά και θα θέλετε στα δάχτυλά σας.

Ταξινομήστε μια λίστα με βάση ένα υπολογισμένο κλειδί (ο Schwartzian transform - το `sorted(xs, key=f)` της Python γινόμενο με το χέρι, διακόσμησε μετά ταξινόμησε μετά αφαίρεσε τη διακόσμηση):

```
use v5.44;
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
```

Φτιάξτε ένα hash από μια λίστα, το αντίστοιχο του dict-comprehension (`{w: len(w) for w in words}`):

```
use v5.44;
my @words = qw(apple pear fig);
my %len = map { $_ => length $_ } @words;
say "$_ => $len{$_}" for sort keys %len;
# apple => 5
# fig => 3
# pear => 4
```

Και το `pperl` είναι ένα εργαλείο γραμμής εντολών στην εξειδικευμένη θέση των awk/sed. Το μονόγραμμο με σιωπηρό βρόχο είναι το καθημερινό αντανακλαστικό του προγραμματιστή της Perl:

```
pperl -lane '$s += $F[-1]; END { print $s }' data.txt
```

Η παγίδα: διαβάστε τον *οδηγό Μονόγραμμων* για τους διακόπτες (`-n`, `-p`, `-a`, `-l`, `-e`) και τις δεκάδες συνταγές που είναι χτισμένες πάνω τους· είναι ένα από τα πράγματα με τη μεγαλύτερη μόχλευση που μπορείτε να μάθετε ερχόμενοι από υπόβαθρο κελύφους-και-Python.

Παγίδες ερχόμενοι από την Python

Οι παγίδες, συγκεντρωμένες για μια τελική σάρωση πριν αρχίσετε να γράφετε:

- **Τα sigils αλλάζουν με τη χρήση.** Ένα στοιχείο του `@a` είναι `$a[0]`· μία τιμή του `%h` είναι `$h{k}`. Το sigil του περιέκτη και το sigil του στοιχείου διαφέρουν.
- **Το `.` ενώνει συμβολοσειρές, το `+` προσθέτει αριθμούς.** Ποτέ μην κάνετε `+` δύο συμβολοσειρές για να τις συνενώσετε· αυτό τις εξαναγκάζει σε αριθμούς και προειδοποιεί.
- **Δύο οικογένειες σύγκρισης.** `==`/`<`/`<=>` για αριθμούς, `eq`/`lt`/ `cmp` για συμβολοσειρές. Επιλέξτε με βάση τον τύπο του τελεστέου, όχι με βάση τη μεταβλητή.
- **Το `sort` ταξινομεί ως συμβολοσειρές από προεπιλογή.** Περάστε `{ $a <=> $b }` για αριθμητική σειρά.
- **Περιβάλλον.** Το `my $x = @a` είναι το *μήκος*· το `my ($x) = @a` είναι το *πρώτο στοιχείο*. Οι παρενθέσεις είναι όλη η διαφορά. Ξαναδιαβάστε το *Περιβάλλον* όταν μια τιμή επιστρέφει με λάθος σχήμα.
- **Εκπλήξεις αλήθειας.** Το `"0"` είναι ψευδές, αλλά τα `"00"` και `"0.0"` είναι **αληθή**. Μόνο τα `0`, `""`, `"0"`, και `undef` είναι ψευδή.
- **Το `s///` επεξεργάζεται επιτόπια** και επιστρέφει ένα πλήθος· χρησιμοποιήστε τη σημαία `/r` για να πάρετε μια νέα συμβολοσειρά αντ” αυτού.
- **Το open δεν εγείρει εξαίρεση.** Χρησιμοποιήστε `open ... or die "...: $!"`.
- **Κάντε `chomp` στις αναγνώσεις γραμμών σας.** Το `<$fh>` κρατάει τον τελικό χαρακτήρα νέας γραμμής.
- **Non-ASCII source needs `use utf8;`.** Under `use v5.44;` a literal non-ASCII byte in your source is a compile error without it. Keep string data ASCII unless you specifically add `use utf8;`.
- **Το χαρακτηριστικό `class` τυπώνει πειραματικές ειδοποιήσεις στο stderr** στο `pperl`. Σωστή έξοδος, θορυβώδης ροή σφαλμάτων.

Από PHP σε Perl

Ξέρετε PHP. Φτάνετε στο `$arr['key']` χωρίς να το σκέφτεστε, το `foreach ($items as $item)` είναι μυϊκή μνήμη, και ενώνετε συμβολοσειρές με `.` επειδή φυσικά το κάνετε. Η Perl και η PHP είναι στενά ξαδέλφια: και οι δύο είναι γλώσσες που χρησιμοποιούν sigils, πρακτικές, φτιαγμένες για να βγαίνει η δουλειά, και η PHP δανείστηκε πολλά από την Perl στις πρώτες της μέρες. Αυτός ο οδηγός αντιστοιχίζει τις συνήθειές σας στην PHP με την Perl ώστε να γίνετε παραγωγικοί στο `pperl` μέσα σε ένα απόγευμα.

Δύο σοκ φτάνουν μέσα στα πρώτα πέντε λεπτά, οπότε γνωρίστε τα τώρα:

- **Το sigil `$` δεν είναι όλη η ιστορία.** Στην PHP κάθε μεταβλητή είναι `$x`, πάντα. Στην Perl το sigil αλλάζει ανάλογα με το πώς *προσπελαύνετε* την τιμή: το `@a` είναι όλος ο πίνακας αλλά το `$a[0]` είναι ένα στοιχείο· το `%h` είναι όλο το hash αλλά το `$h{k}` είναι μία τιμή. Το δολάριο που ήδη ξέρετε σημειώνει ένα μεμονωμένο βαθμωτό· τα `@` και `%` είναι νέα και η πρώτη ενότητα δεν αφορά τίποτε άλλο.

- **Ο ένας τύπος `array` της PHP γίνεται δύο τύποι στην Perl.** Ένα `array` της PHP είναι μια διατεταγμένη αντιστοίχιση που κάνει χρέη και λίστας και λεξικού. Η Perl το χωρίζει σε `@array` (με ακέραιους δείκτες) και `%hash` (με κλειδιά συμβολοσειρών), και επιλέγετε εξαρχής ποιο έχετε. Αυτός ο διαχωρισμός καθοδηγεί τις ενότητες Πίνακες και Hashes.

Every example here runs on `pperl`, a Rust reimplementation of Perl 5.44. Each Perl snippet was executed on `pperl`; the `# ...` comments show its real output. Start your scripts with:

```
use v5.44;
```

Αυτή η μία γραμμή ενεργοποιεί το `strict` (καμία τυχαία καθολική μεταβλητή), τα `warnings`, την ενσωματωμένη `say` (ένα `print` με τελικό χαρακτήρα νέας γραμμής, όπως το `echo` συν `PHP_EOL` της PHP), τις υπογραφές υπορουτινών, και το σύγχρονο σύνολο χαρακτηριστικών που χρησιμοποιείται σε όλον αυτόν τον οδηγό.

Πώς να το διαβάσετε αυτό

- *Μεταβλητές και sigils* - η μετάβαση $/@/%.
- *Συμβολοσειρές* - παρεμβολή και συνένωση με `..`
- *Αριθμοί* - == έναντι eq, η κλασική παγίδα της PHP.
- *Πίνακες και λίστες* - το μισό από το `array` σας.
- *Hashes* - το άλλο μισό από το `array` σας.
- *Περιβάλλον* - η έννοια για την οποία η PHP δεν έχει λέξη. Διαβάστε αυτήν.
- *Αλήθεια* - τι μετράει ως ψευδές, με εκπλήξεις.
- *Ροή ελέγχου* - `if`, βρόχοι, οι μεταθεματικές μορφές.
- *Υπορουτίνες* - το `function` γίνεται `sub`.
- *Αναφορές* - εμφωλευμένα δεδομένα και closures.
- *Κανονικές εκφράσεις* - το regex ως σύνταξη, όχι `preg_*`.
- *I/O αρχείων και ροών* - `fopen`, αναγνώσεις γραμμών, slurp.
- *Αρθρώματα και πακέτα* - τα `require`/`use` γίνονται `use`.
- *Αντικείμενα* - η σύγχρονη σύνταξη `class`.
- *Χειρισμός σφαλμάτων* - `try`/`catch`, η ίδια ιδέα.
- *Ιδιώματα* - τα καθημερινά μονόγραμμα.
- *Παγίδες ερχόμενοι από την PHP* - σαρώστε τες πριν γράψετε.

Μεταβλητές και sigils

Χρησιμοποιείτε ήδη το $, οπότε το πρώτο μισό των sigils της Perl είναι δωρεάν. Η νέα ιδέα είναι ότι **το sigil σημειώνει την προσπέλαση, όχι τη μεταβλητή**. Όλος ο πίνακας είναι `@a`· ένα στοιχείο είναι `$a[0]`. Όλο το hash είναι `%h`· μία τιμή είναι `$h{k}`. Η PHP κρατάει το $ σε όλα και αλλάζει τις αγκύλες (`$a[0]`, `$h['k']`)· η Perl αλλάζει το sigil αντ” αυτού.

Το my δηλώνει μια μεταβλητή με λεξιλογική εμβέλεια, το αντίστοιχο μιας συνηθισμένης τοπικής σύνδεσης.

```
$name = "Ada";
$langs = ["PHP", "Perl"];
$ages = ["Ada" => 36, "Rasmus" => 56];
```

```
use v5.44;
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
```

Η παγίδα: όταν φτάνετε *μέσα* σε ένα σύνολο για ένα στοιχείο, το sigil ακολουθεί την τιμή που παίρνετε πίσω, όχι τον περιέκτη. Ένα μεμονωμένο στοιχείο του `@langs` είναι βαθμωτό, οπότε είναι `$langs[0]`, όχι `@langs[0]`· μία τιμή από το `%ages` είναι `$ages{Ada}`. Διαβάστε το αρχικό sigil ως «τι σχήμα παίρνω» - `$` για ένα πράγμα, `@` για μια λίστα, `%` για όλη την αντιστοίχιση - και τις αγκύλες ως «από πού».

Συμβολοσειρές

Η συνένωση είναι `.`, ακριβώς όπως στην PHP. Τα διπλά εισαγωγικά παρεμβάλλουν μεταβλητές, τα μονά όχι - ο ίδιος διαχωρισμός που κάνει η PHP ανάμεσα σε `"..."` και `'...'`. Δεν υπάρχει σύνταξη `sprintf`-μέσα-στη-συμβολοσειρά να μάθετε επειδή η προεπιλεγμένη συμβολοσειρά με διπλά εισαγωγικά ήδη παρεμβάλλει.

```
$name = "Ada";
$greeting = "Hello, $name";
$literal  = 'Hello, $name';
$joined   = "AB" . "CD";
$shout    = str_repeat("AB", 3);
```

```
use v5.44;
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
```

Η παγίδα: η επανάληψη συμβολοσειράς είναι ο τελεστής `x`, όχι συνάρτηση - `"AB" x 3`, όχι `str_repeat`. Η παρεμβολή ενός στοιχείου πίνακα ή hash δεν χρειάζεται διαφυγή με άγκιστρα: τα `"First: $langs[0]"` και `"Name: $h{name}"` δουλεύουν και τα δύο όπως είναι γραμμένα, ενώ η PHP θα σας έβαζε να γράψετε `"{$arr['k']}"`. Για να παρεμβάλετε το αποτέλεσμα μιας *έκφρασης*, τυλίξτε το σε `@{[ ... ]}`: το `"sum: @{[ 2 + 3 ]}"` τυπώνει `sum: 5`.

Αριθμοί

Η Perl έχει έναν αριθμητικό τύπο στην επιφάνεια· δεν δηλώνετε `int` έναντι `float`. Αυτό που πραγματικά θα σας δαγκώσει είναι οι τελεστές, οπότε διαβάστε την παγίδα δύο φορές.

```
echo 7 / 2;           // 3.5
echo intdiv(7, 2);    // 3
echo 2 ** 10;         // 1024
echo 10 % 3;          // 1
```

```
use v5.44;
say 7 / 2;          # 3.5
say int(7 / 2);     # 3
say 2 ** 10;        # 1024
say 10 % 3;         # 1
```

Η παγίδα: η Perl έχει **δύο οικογένειες τελεστών και επιλέγετε με βάση τον τύπο του τελεστέου, όχι με βάση τη μεταβλητή**. Οι αριθμοί χρησιμοποιούν ==, !=, <, >, <=>· οι συμβολοσειρές χρησιμοποιούν eq, ne, `lt`, `gt`, `cmp`. Και το + είναι *πάντα* αριθμητικό - δεν υπάρχει υπερφόρτωση τελεστή όπως το + της PHP κάνει ένωση πινάκων. Συνενώστε με `.`, προσθέστε με +, και ποτέ μην τα διασταυρώσετε:

```
use v5.44;
say "12" . "3";     # 123   (string concat)
say "12" + "3";     # 15    (numeric add)
say "10" == "10.0" ? "same" : "diff";   # same  (numeric compare)
say "10" eq "10.0" ? "same" : "diff";   # diff  (string compare)
```

Η PHP απέσυρε το μπέρδεμα του χαλαρού == στις σύγχρονες εκδόσεις· η Perl δεν το είχε ποτέ, αλλά λύνει το ίδιο πρόβλημα κάνοντάς σας να πείτε ποια σύγκριση εννοείτε. Το `"10" == "10.0"` είναι αληθές (αριθμητικά), το `"10" eq "10.0"` είναι ψευδές (συμβολοσειρά). Η χρήση == σε συμβολοσειρές, ή eq σε αριθμούς, είναι το πιο συχνό σφάλμα κατά τη μετάβαση από PHP σε Perl.

Πίνακες και λίστες

Εδώ είναι που ο ένας τύπος `array` της PHP χωρίζεται στα δύο. Όταν χρησιμοποιούσατε ένα `array` ως αριθμητικά ευρετηριασμένη λίστα - `$items[0]`, `array_push`, `foreach` πάνω σε τιμές - αυτό είναι ένας **πίνακας** της Perl, που δηλώνεται με @. Τα ρήματα έχουν μετονομαστεί αλλά οι λειτουργίες είναι ίδιες, και οι αρνητικοί δείκτες δουλεύουν όπως περιμένετε.

```
$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 v5.44;
my @nums = (1, 2, 3, 4, 5);
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
push @nums, 6;
my $last = pop @nums;       # 6
say scalar @nums;           # 5
say "@nums[1,2]";           # 2 3   (a slice)
```

Τα array_map και array_filter σας γίνονται map και grep. Το map μετασχηματίζει (το array_map της PHP)· το grep φιλτράρει (το array_filter της PHP). Το $_ είναι το τρέχον στοιχείο, η σιωπηρή μεταβλητή βρόχου της Perl, όπως είναι η παράμετρος του callback στην PHP - μόνο που δεν το ονοματίζετε.

```
$squares = array_map(fn($x) => $x * $x, $nums);
$evens   = array_filter($nums, fn($x) => $x % 2 == 0);
rsort($nums);   // descending, in place
```

```
use v5.44;
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
```

Η παγίδα: το sort συγκρίνει ως συμβολοσειρές από προεπιλογή, οπότε το sort (10, 9, 100) δίνει 10 100 9. Για αριθμούς πρέπει να περάσετε έναν συγκριτή: sort { $a <=> $b } @nums. Τα $a και $b είναι τα δύο στοιχεία που συγκρίνονται, και το <=> είναι ο αριθμητικός τρίτιμος τελεστής (το cmp είναι το αδελφάκι του για συμβολοσειρές). Σημειώστε επίσης ότι το grep κρατάει στοιχεία, επιστρέφοντας μια νέα λίστα, ενώ το array_filter της PHP διατηρεί τα κλειδιά - οι πίνακες της Perl δεν έχουν κλειδιά να διατηρήσουν, οπότε το αποτέλεσμα απλώς ξανα-ευρετηριάζεται.

Hashes

Το άλλο μισό του array της PHP - το μέρος που χρησιμοποιούσατε με κλειδιά συμβολοσειρών, $config['host'], array_key_exists, foreach ($h as $k => $v) - είναι ένα **hash** της Perl, που δηλώνεται με % και προσπελαύνεται μία τιμή τη φορά με $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 v5.44;
my %h = (apple => 3, pear => 5, plum => 2);
say $h{apple};                          # 3
say join ",", sort keys %h;             # apple,pear,plum
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
say exists $h{pear} ? "yes" : "no";     # yes
delete $h{plum};
for my $k (sort keys %h) {
    say "$k => $h{$k}";                 # apple => 3 ... pear => 5
}
```

Η παγίδα: ο έλεγχος μέλους είναι `exists $h{key}` (το κλειδί υπάρχει) ή `defined $h{key}` (το κλειδί υπάρχει *και* η τιμή του δεν είναι `undef`) - μαζί καλύπτουν τα `array_key_exists` και `isset` της PHP. Το παχύ κόμμα `=>` είναι ο ιδιωματικός διαχωριστής κλειδιού/τιμής και βάζει αυτόματα εισαγωγικά στο bareword στα αριστερά του, οπότε το `apple => 3` δεν χρειάζεται εισαγωγικά γύρω από το `apple`. Ένα hash της Perl **δεν έχει σειρά εισαγωγής** (σε αντίθεση με ένα array της PHP, που τη διατηρεί)· επαναλάβετε με `sort keys %h` όταν θέλετε μια σταθερή σειρά. Για να διατρέξετε κλειδί και τιμή μαζί, το `foreach ($h as $k => $v)` της PHP γίνεται είτε `for my $k (keys %h)` είτε, όταν θέλετε και τα δύο μαζί, `while (my ($k, $v) = each %h)`.

Περιβάλλον

Αυτό είναι το κεφάλαιο που πρέπει να διαβάσετε δύο φορές. Η PHP δεν έχει αντίστοιχο, και σχεδόν κάθε ιστορία «η Perl είναι παράξενη» ανάγεται σε αυτό.

Κάθε έκφραση στην Perl τρέχει είτε σε **περιβάλλον λίστας** είτε σε **βαθμωτό περιβάλλον**, που καθορίζεται από το πού εμφανίζεται, όχι από το τι είναι. Ο ίδιος πίνακας συμπεριφέρεται διαφορετικά σε καθένα. Η ανάθεση ενός πίνακα σε ένα *βαθμωτό* δίνει το μήκος του· η ανάθεσή του σε μια *λίστα* (ένα σύνολο μεταβλητών σε παρενθέσεις) αντιγράφει στοιχεία.

```
use v5.44;
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 ($x, $y) = @list;        # 10 and 20
say "$x $y";                # 10 20
```

Οι παρενθέσεις στα αριστερά είναι όλη η διαφορά. Το `my $x = @list` ρωτάει «πόσα;» και παίρνει 3 - εδώ ζει το `count($arr)`, απλώς γραμμένο μέσω περιβάλλοντος αντί για συνάρτηση. Το `my ($x) = @list` ρωτάει «δώσε μου τα στοιχεία» και παίρνει `10`. Ένας προγραμματιστής της PHP διαβάζει και τα δύο ως «ανάθεσε το array», που είναι ακριβώς η παγίδα.

Μια υπορουτίνα μπορεί να ρωτήσει σε ποιο περιβάλλον κλήθηκε, με `wantarray`, και να επιστρέψει διαφορετικά πράγματα ανάλογα:

```
use v5.44;
sub items {
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    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
```

Οι ενσωματωμένες το χρησιμοποιούν κι αυτές. Το `scalar @array` επιβάλλει βαθμωτό περιβάλλον για να πάρει ένα πλήθος - αυτό είναι το ρητό `count()` που ψάχνατε. Μια αντιστοίχιση regex με `/g` επιστρέφει τη λίστα των αντιστοιχιών σε περιβάλλον λίστας και ένα αληθές/ψευδές σε βαθμωτό περιβάλλον.

Η παγίδα: δεν υπάρχει μία μόνο απάντηση στο «τι κάνει αυτός ο πίνακας» - εξαρτάται από την περιβάλλουσα έκφραση. Όταν κάτι επιστρέφει έναν αριθμό που περιμένατε να είναι λίστα (ή το αντίστροφο), το περιβάλλον είναι ο πρώτος ύποπτος. Φτάστε στο `scalar` για να επιβάλετε ένα πλήθος και σε παρενθέσεις στα αριστερά για να επιβάλετε αντιγραφή στοιχείων.

Αλήθεια τιμών

Οι ψευδείς τιμές της Perl είναι μια σύντομη, συγκεκριμένη λίστα: ο αριθμός `0`, η κενή συμβολοσειρά `""`, η συμβολοσειρά `"0"`, και το `undef` (το `null` της Perl). **Όλα τα άλλα είναι αληθή.** Αυτό είναι κοντά στους κανόνες της PHP αλλά όχι πανομοιότυπο, και οι διαφορές είναι σιωπηλές.

```
use v5.44;
for my $v (0, 1, "", "0", "00", "0.0", "false") {
    say "'$v' is ", ($v ? "true" : "false");
}
```

Αυτό τυπώνει, ακριβώς:

```
'0' is false
'1' is true
'' is false
'0' is false
'00' is true
'0.0' is true
'false' is true
```

Η παγίδα: τα `"00"` και `"0.0"` είναι **αληθή** στην Perl, παρόλο που η PHP αντιμετωπίζει το `"0"` και μόνο το `"0"` ως την ψευδή συμβολοσειρά επίσης - η Perl συμφωνεί στο `"0"` αλλά όχι στα όμοιά του. Η συμβολοσειρά `"false"` είναι αληθής (είναι μια μη κενή, μη-`"0"` συμβολοσειρά), το ίδιο όπως στην PHP. Και σε αντίθεση με τον κανόνα της PHP ότι το κενό array είναι ψευδές, ένας κενός πίνακας της Perl δεν είναι ο ίδιος μια ψευδής *τιμή*: σε λογικό περιβάλλον ένας πίνακας δίνει το μήκος του, οπότε το `if (@list)` είναι αληθές ακριβώς όταν η λίστα είναι μη κενή, που είναι ο έλεγχος που θέλετε. Για το «χρησιμοποίησε αυτό αν είναι ορισμένο, αλλιώς μια προεπιλογή», χρησιμοποιήστε το defined-or `//`, το αντίστοιχο του `??` της PHP, που πέφτει πίσω μόνο στο `undef` (όχι στο `0` ή στο `""`):

```
use v5.44;
my %cfg = (host => "localhost");
my $port = $cfg{port} // 8080;   # key absent -> undef -> 8080
say $port;                       # 8080
```

Ροή ελέγχου

Οι δεσμευμένες λέξεις είναι οικείες· η γραφή διαφέρει με δύο μικρούς τρόπους. Το `elseif` είναι `elsif` (ένα e), και οι συνθήκες χρειάζονται παρενθέσεις ενώ τα μπλοκ χρειάζονται άγκιστρα - δεν υπάρχει εναλλακτική σύνταξη `:`/`endif`.

```
if ($n < 0) {
    $sign = "neg";
} elseif ($n == 0) {
    $sign = "zero";
} else {
    $sign = "pos";
}

foreach ($nums as $x) {
    echo $x;
}

while ($running) {
    step();
}
```

```
use v5.44;
my $sign;
if    ($n < 0)  { $sign = "neg"  }
elsif ($n == 0) { $sign = "zero" }
else            { $sign = "pos"  }

for my $x (@nums) { say $x }

while ($running) { step() }
```

Η Perl προσθέτει δύο πράγματα που δεν έχει η PHP. Οι **μεταθεματικοί τροποποιητές** επιτρέπουν σε μια απλή εντολή να φέρει τη δική της συνθήκη ή βρόχο, κάτι που διαβάζεται φυσικά για μονόγραμμα:

```
use v5.44;
say $_ for @nums;                # loop, one statement
say "negative" if $n < 0;        # guarded statement
$sum += $_ for @nums;            # accumulate
```

Και τα `unless`/`until` είναι οι αρνημένες μορφές των `if`/`while`, για όταν η θετική διατύπωση διαβάζεται αδέξια: `say "ok" unless $error`.

Η παγίδα: ο τριαδικός τελεστής είναι `?:`, ο ίδιος με της PHP, και αλυσιδώνεται με τον ίδιο τρόπο: `my $sign = $n < 0 ? "neg" : $n == 0 ? "zero" : "pos"`.

Η Perl δεν έχει εντολή switch/match στον πυρήνα· ένας αλυσιδωτός τριαδικός τελεστής ή ένα hash αποστολής (`$handler{$key}->()`) είναι το ιδίωμα, και ένα μπλοκ for-πάνω-σε-μία-τιμή είναι το συνηθισμένο υποκατάστατο για ένα switch.

Υπορουτίνες

Το function της PHP γίνεται το sub της Perl. Η σύγχρονη Perl έχει υπογραφές που μοιάζουν πολύ με τις λίστες παραμέτρων της PHP, συμπεριλαμβανομένων των προεπιλεγμένων τιμών.

```
function greet($name, $greeting = "Hello") {
    return "$greeting, $name!";
}

function total(...$nums) {
    return array_sum($nums);
}
```

```
use v5.44;
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
```

Το variadic `...$nums` γίνεται μια τελική παράμετρος πίνακα `@nums`, που συγκεντρώνει όλα τα υπόλοιπα ορίσματα. Τα ονοματισμένα ορίσματα της PHP (ή η συνηθισμένη σύμβαση `$opts` με array) αντιστοιχίζονται σε μια παράμετρο hash, αφού τα ζεύγη `key => value` ισοπεδώνονται σε μια λίστα που μια υπογραφή `%opts` συγκεντρώνει:

```
use v5.44;
sub connect_to (%opts) {
    return "host=$opts{host} port=$opts{port}";
}
say connect_to(host => "localhost", port => 8080);
# host=localhost port=8080
```

Η παγίδα: μια sub επιστρέφει την τιμή της τελευταίας της έκφρασης ακόμη και χωρίς `return`, και η επιστροφή είναι **ευαίσθητη στο περιβάλλον** (δείτε *Περιβάλλον*). Αν κάνετε `return @list`, ο καλών παίρνει τη λίστα σε περιβάλλον λίστας αλλά το *πλήθος* σε βαθμωτό περιβάλλον. Να είστε εσκεμμένοι σχετικά με το τι επιστρέφετε όταν μπορεί να χρησιμοποιηθεί με οποιονδήποτε τρόπο.

Αναφορές

Οι πίνακες της PHP εμφωλεύονται απευθείας: ένα `array` μπορεί να κρατήσει ένα άλλο `array` και απλώς συνεχίζετε να βάζετε δείκτες. Οι πίνακες και τα hashes της Perl δεν εμφωλεύονται απευθείας· τα εμφωλεύετε μέσω **αναφορών**, βαθμωτών που δείχνουν σε ένα σύνολο. Αυτός είναι ο ίδιος μηχανισμός με τις αναφορές `&$x` της PHP και τις λαβές αντικειμένων της, αλλά στην Perl τον χρησιμοποιείτε κάθε φορά που χτίζετε εμφωλευμένα δεδομένα. Πάρτε μια αναφορά με `\`, ακολουθήστε τη με `->`.

```
$data = ["name" => "Ada", "langs" => ["Perl", "PHP"]];
echo $data["langs"][1];      // PHP
$data["langs"][] = "Ruby";
```

```
use v5.44;
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
```

Το `{ ... }` φτιάχνει μια ανώνυμη αναφορά hash και το `[ ... ]` μια ανώνυμη αναφορά πίνακα - τα κυριολεκτικά δομικά στοιχεία των εμφωλευμένων δεδομένων, που αντιστοιχούν στις κυριολεξίες `[...]` και `[k => v]` που ήδη γράφετε στην PHP. Το βέλος `->` ακολουθεί μια αναφορά κατά ένα βήμα· ανάμεσα σε δύο δείκτες είναι προαιρετικό, οπότε τα `$data->{langs}[1]` και `$data->{langs}->[1]` είναι ίδια. Το βέλος είναι επίσης ακριβώς το βέλος κλήσης μεθόδου που ξέρετε από την αντικειμενοστρέφεια της PHP - ίδιο σύμβολο, και για το «φτάσε μέσα» και για το «κάλεσε μια μέθοδο».

Οι ανώνυμες subs είναι closures, ακριβώς όπως τα `fn` και `function() use (...)` της PHP, αλλά χωρίς να χρειάζεται ρήτρα `use` - συλλαμβάνουν αυτόματα τις λεξιλογικές μεταβλητές:

```
use v5.44;
my $double = sub ($n) { $n * 2 };
say $double->(21);          # 42
```

Η παγίδα: οι μεταθεματικές μορφές αποαναφοράς (`->@*`, `->%*`, `->$*`) είναι ο σύγχρονος, ευανάγνωστος τρόπος να αποαναφέρετε ένα ολόκληρο σύνολο, και το `"@{ ... }"` είναι ο τρόπος να παρεμβάλετε έναν αποαναφερμένο πίνακα μέσα σε μια συμβολοσειρά. Οι παλαιότερες μορφές `@$ref` και `${$ref}{key}` εξακολουθούν να δουλεύουν και εμφανίζονται σε υπάρχοντα κώδικα, αλλά προτιμήστε το βέλος και τις μεταθεματικές μορφές σε νέο κώδικα.

Κανονικές εκφράσεις

Στην PHP το regex είναι η οικογένεια συναρτήσεων `preg_*` με συμβολοσειρές μοτίβων όπως `'/\d+/'`. Στην Perl είναι **σύνταξη**: το `=~` συνδέει ένα μοτίβο με μια συμβολοσειρά, το `m//` αντιστοιχίζει, το `s///` αντικαθιστά - χωρίς οριοθέτες-μέσα-σε-συμβολοσειρά, χωρίς κλήση συνάρτησης. Οι ομάδες σύλληψης καταλήγουν στα `$1`, `$2`, και οι ονομαστικές συλλήψεις στο `%+`.

```
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 v5.44;
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
```

Μια αντιστοίχιση `/g` σε περιβάλλον λίστας επιστρέφει *όλες* τις συλλήψεις, το άμεσο αντίστοιχο του `preg_match_all`:

```
use v5.44;
my @nums = "x1y2z3" =~ /(\d)/g;     # 1 2 3
say "@nums";
```

Η παγίδα: το `s///` τροποποιεί τον στόχο του *επιτόπια* και επιστρέφει τον αριθμό των αντικαταστάσεων, όχι μια νέα συμβολοσειρά - σε αντίθεση με το `preg_replace`, που επιστρέφει μια φρέσκια συμβολοσειρά. Γι" αυτό το παράδειγμα αντιγράφει πρώτα με (`my $t = "a,b,c"`) και μετά αντικαθιστά στο αντίγραφο - αλλιώς θα επεξεργαζόσασταν μια κυριολεξία συμβολοσειράς. Για να πάρετε μια νέα συμβολοσειρά χωρίς να αγγίξετε την αρχική (το αντανακλαστικό `preg_replace`), χρησιμοποιήστε τη σημαία `/r`: `my $t = $s =~ s/,/;/gr`.

I/O αρχείων και ροών

Το ιδίωμα είναι το open με τρία ορίσματα και ένα λεξιλογικό filehandle, το αντίστοιχο του `fopen` της PHP μέσα σε ένα `$handle`. Ο τελεστής με γωνιώδεις αγκύλες `<$fh>` διαβάζει μια γραμμή, με τον ίδιο ρόλο όπως το `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 v5.44;
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;
```

Για να ρουφήξετε (slurp) όλες τις γραμμές μονομιάς, διαβάστε το handle σε περιβάλλον λίστας - το ηθικό αντίστοιχο του `file()`:

```
use v5.44;
open my $fh, '<', "/tmp/sample.txt" or die $!;
my @lines = <$fh>;          # all lines, including newlines
say scalar @lines;          # 3
```

Η παγίδα: μια γραμμή που διαβάστηκε με `<$fh>` κρατάει τον τελικό χαρακτήρα νέας γραμμής· καλέστε `chomp` για να τον αφαιρέσετε (το αντανακλαστικό `rtrim`, αλλά το `chomp` αφαιρεί μόνο τον διαχωριστή εγγραφών). Και το `open` επιστρέφει ψευδές σε αποτυχία αντί να εγείρει εξαίρεση ή να προειδοποιεί, οπότε το ιδίωμα `or die "...: $!"` είναι υποχρεωτικό - το `$!` είναι το μήνυμα σφάλματος συστήματος, το errno της Perl (το `error_get_last()` της PHP).

Αρθρώματα και πακέτα

Τα `require`/`include` της PHP και ο autoloader του Composer γίνονται το `use` της Perl. Ο διαχωριστής χώρου ονομάτων είναι το `::`, όχι το `\`, και ένα άρθρωμα `Foo::Bar` βρίσκεται στο `Foo/Bar.pm` στη διαδρομή ένταξης `@INC` (το `include_path` της Perl). Η λίστα εισαγωγής ενός αρθρώματος είναι μια επιλογή ονομάτων που θα φέρετε στον χώρο ονομάτων σας.

```
require 'vendor/autoload.php';
use App\Util\ListHelper;
$sum = ListHelper::sum(range(1, 10));
```

```
use v5.44;
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
```

Ο τεράστιος επίπεδος χώρος ονομάτων τυπικών συναρτήσεων της PHP (`count`, `in_array`, `array_map`, `implode`, `sprintf`, ...) αντιστοιχίζεται στην Perl εν μέρει σε ενσωματωμένους τελεστές και εν μέρει σε λίγα καθημερινά αρθρώματα. Οι συνηθισμένες μεταφράσεις:

PHP	Perl
`count($a)`	`scalar @a`
`in_array($x, $a)`	`grep { $_ eq $x } @a`
`array_map($f, $a)`	`map { ... } @a`
`array_filter($a, $f)`	`grep { ... } @a`
`implode(",", $a)`	`join ",", @a`
`explode(",", $s)`	`split /,/, $s`
`sprintf(...)`	`sprintf(...)` (ίδιο)
`isset($h[$k])`	`exists $h{$k}` / `defined $h{$k}`
`array_sum($a)`	`sum0 @a` (`List::Util`)

Τα `List::Util`, `Scalar::Util`, `POSIX`, `File::Spec`, και `Data::Dumper` είναι τα καθημερινά αρθρώματα· τα περισσότερα από αυτά που θα τραβούσατε από το Packagist έρχονται στον πυρήνα ή στο CPAN, το αρχείο πακέτων της Perl.

Η παγίδα: το `use` τρέχει κατά τη *μεταγλώττιση*, όχι στο σημείο του αρχείου όπου εμφανίζεται, οπότε η σειρά των γραμμών `use` σε σχέση με τον κώδικά σας δεν έχει σημασία όπως μπορεί να έχει η τοποθέτηση του `require` στην PHP. Η εισαγωγή είναι επίσης προαιρετική ανά όνομα: το `use List::Util qw(sum0 max)` φέρνει ακριβώς τα `sum0` και `max` στον χώρο ονομάτων σας, τίποτε άλλο - δεν υπάρχει καθολικός πίνακας συναρτήσεων όπως η PHP εκθέτει κάθε ενσωματωμένη παντού.

(Τα δεδομένα αιτήματος web - τα superglobals `$_GET`, `$_POST`, `$_SERVER` της PHP - δεν έχουν αντίστοιχο σε επίπεδο γλώσσας στην Perl· αυτή είναι η δουλειά ενός επιπέδου CGI ή PSGI, όχι της βασικής γλώσσας, και είναι εκτός σκοπού για αυτόν τον οδηγό.)

Αντικείμενα

For new code, Perl 5.44 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)
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
$p->move(1, 1);
echo $p->to_string();        // (4, 5)
```

```
use v5.44;
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)
```

Ο κατασκευαστής δημιουργείται για εσάς, όπως οι προαγόμενες ιδιότητες κατασκευαστή της PHP: το `:param` σημειώνει ένα πεδίο που η αυτόματα κατασκευασμένη new δέχεται ως ονοματισμένο όρισμα, και το `= 0` είναι η προεπιλογή του. Το βέλος κλήσης μεθόδου `->` είναι το ίδιο που χρησιμοποιείτε στην PHP. Η κληρονομικότητα είναι `:isa`, το αντίστοιχο του `extends`:

```
use v5.44;
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 5.44, 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. 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*.

Χειρισμός σφαλμάτων

Το try/catch της PHP έχει ένα άμεσο αντίστοιχο στη σύγχρονη Perl, και διαβάζεται σχεδόν πανομοιότυπα. Η πλευρά που εγείρει είναι το die (το throw της PHP):

```
function risky($n) {
    if ($n < 0) {
        throw new Exception("negative!");
    }
    return sqrt($n);
}

try {
    risky(-4);
} catch (Exception $e) {
    echo "caught: " . $e->getMessage() . "\n";
}
```

```
use v5.44;
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!
}
```

Η παλαιότερη, ακόμη συνηθισμένη μορφή τυλίγει κώδικα σε eval { } και εξετάζει το $@ (το σφάλμα που πιάστηκε) μετά - το ίδιο μοτίβο με κώδικα PHP γραμμένο πριν οι εξαιρέσεις γίνουν ιδιωματικές:

```
use v5.44;
my $result = eval { risky(-1) };
if ($@) {
    print "caught: $@";     # caught: negative!
}
```

Η παγίδα: το die με μια συμβολοσειρά που τελειώνει σε \n παράγει ακριβώς αυτό το μήνυμα· το die με μια συμβολοσειρά που *δεν* τελειώνει σε χαρακτήρα νέας γραμμής παίρνει αυτόματα την προσθήκη " at SCRIPT line N.\n". Τελειώστε τις συμβολοσειρές σφάλματός σας με \n όταν θέλετε ένα καθαρό μήνυμα και παραλείψτε το όταν θέλετε το ίχνος αρχείου-και-γραμμής. Σημειώστε επίσης τις δύο μεταβλητές σφάλματος: το $@ είναι η τελευταία εξαίρεση που πιάστηκε (το $e της PHP), ενώ το $! είναι το τελευταίο σφάλμα *συστήματος* (ένα αποτυχημένο open, η συμβολοσειρά errno).

Ιδιωματισμοί

Μερικά μοτίβα που θα βλέπετε καθημερινά και θα θέλετε στα δάχτυλά σας.

Ταξινομήστε μια λίστα με βάση ένα υπολογισμένο κλειδί (ο Schwartzian transform - το `usort` της PHP με ένα προϋπολογισμένο κλειδί, γινόμενο σε μία έκφραση: διακόσμησε, ταξινόμησε, αφαίρεσε τη διακόσμηση):

```
use v5.44;
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
```

Φτιάξτε ένα hash από μια λίστα, το αντίστοιχο του `array_combine`/`array_column` (εδώ, λέξη προς το μήκος της):

```
use v5.44;
my @words = qw(apple pear fig);
my %len = map { $_ => length $_ } @words;
say "$_ => $len{$_}" for sort keys %len;
# apple => 5
# fig => 3
# pear => 4
```

Και το `pperl` είναι ένα εργαλείο γραμμής εντολών στην εξειδικευμένη θέση των `awk`/`sed` - την ίδια δουλειά για την οποία ίσως κατέφευγες σε ένα γρήγορο σενάριο PHP CLI. Το μονόγραμμο με σιωπηρό βρόχο είναι το καθημερινό αντανακλαστικό του προγραμματιστή της Perl:

```
pperl -lane '$s += $F[-1]; END { print $s }' data.txt
```

Η παγίδα: διαβάστε τον *οδηγό Μονόγραμμων* για τους διακόπτες (`-n`, `-p`, `-a`, `-l`, `-e`) και τις δεκάδες συνταγές που είναι χτισμένες πάνω τους· είναι ένα από τα πράγματα με τη μεγαλύτερη μόχλευση που μπορείτε να μάθετε ερχόμενοι από υπόβαθρο PHP-και-κελύφους.

Παγίδες ερχόμενοι από την PHP

Οι παγίδες, συγκεντρωμένες για μια τελική σάρωση πριν αρχίσετε να γράφετε:

- **Τα sigils αλλάζουν με τη χρήση.** Ένα στοιχείο του `@a` είναι `$a[0]`· μία τιμή του `%h` είναι `$h{k}`. Η PHP κρατάει το `$` παντού· η Perl αλλάζει το sigil ανάλογα με το σχήμα που παίρνετε.
- **Ένα `array` της PHP είναι δύο τύποι στην Perl.** Αποφασίστε εξαρχής: ακέραια κλειδιά και σειρά σημαίνει `@array`· κλειδιά συμβολοσειρών σημαίνει `%hash`. Δεν είναι εναλλάξιμα.
- **Δύο οικογένειες σύγκρισης.** `==`/`<`/`<=>` για αριθμούς, `eq`/`lt`/ `cmp` για συμβολοσειρές. Επιλέξτε με βάση τον τύπο του τελεστέου, όχι με βάση τη μεταβλητή. Το `==` σε συμβολοσειρές ή το `eq` σε αριθμούς είναι το κλασικό σφάλμα.

- **Το `.` ενώνει συμβολοσειρές, το `+` προσθέτει μόνο αριθμούς.** Το `+` της Perl ποτέ δεν κάνει ένωση πινάκων ή συνένωση· το μπέρδεμα `.` και `+` αλλοιώνει τα δεδομένα σας σιωπηλά ή προειδοποιεί.
- **Το `sort` ταξινομεί ως συμβολοσειρές από προεπιλογή.** Περάστε `{ $a <=> $b }` για αριθμητική σειρά.
- **Τα hashes δεν έχουν σειρά.** Σε αντίθεση με ένα array της PHP, η σειρά επανάληψης δεν είναι η σειρά εισαγωγής· κάντε `sort keys %h` όταν χρειάζεστε σταθερότητα.
- **Περιβάλλον.** Το `my $x = @a` είναι το *πλήθος*· το `my ($x) = @a` είναι το *πρώτο στοιχείο*. Οι παρενθέσεις είναι όλη η διαφορά. Ξαναδιαβάστε το *Περιβάλλον* όταν μια τιμή επιστρέφει με λάθος σχήμα.
- **Εκπλήξεις αλήθειας.** Το `"0"` είναι ψευδές, αλλά τα `"00"` και `"0.0"` είναι **αληθή**. Μόνο τα `0`, `""`, `"0"`, και `undef` είναι ψευδή. Ένας κενός πίνακας δεν είναι ψευδής τιμή - το `if (@a)` ελέγχει αν είναι μη κενός.
- **Το `s///` επεξεργάζεται επιτόπια** και επιστρέφει ένα πλήθος, σε αντίθεση με το `preg_replace`· χρησιμοποιήστε τη σημαία `/r` για να πάρετε μια νέα συμβολοσειρά αντ” αυτού.
- **Το open δεν εγείρει εξαίρεση.** Χρησιμοποιήστε `open ... or die "...: $!"`.
- **Κάντε `chomp` στις αναγνώσεις γραμμών σας.** Το `<$fh>` κρατάει τον τελικό χαρακτήρα νέας γραμμής.
- **Τα superglobals του web δεν ανήκουν στον πυρήνα.** Τα `$_GET`/`$_POST` ανήκουν σε ένα επίπεδο CGI ή PSGI, όχι στη γλώσσα.
- **Non-ASCII source needs `use utf8;`.** Under `use v5.44;` a literal non-ASCII byte in your source is a compile error without it. Keep string data ASCII unless you specifically add `use utf8;`.
- **Το χαρακτηριστικό `class` τυπώνει πειραματικές ειδοποιήσεις στο stderr** στο `pperl`. Σωστή έξοδος, θορυβώδης ροή σφαλμάτων.

Από Ruby σε Perl

Ξέρετε Ruby. Περνάτε blocks στα `each` και `map` χωρίς να το σκέφτεστε, φτάνετε σε ένα `Hash` αντανακλαστικά, και το `arr.select { |x| ... }` είναι μυϊκή μνήμη. Αυτός ο οδηγός αντιστοιχίζει καθεμία από αυτές τις συνήθειες στην Perl, ώστε να γίνετε παραγωγικοί στο `pperl` μέσα σε ένα απόγευμα αντί να ξαναμάθετε προγραμματισμό από το μηδέν.

Δύο μετατοπίσεις φτάνουν μέσα στα πρώτα πέντε λεπτά, οπότε γνωρίστε τες τώρα:

- **Τα sigils σημειώνουν τον τύπο, όχι την εμβέλεια.** Η Ruby χρησιμοποιεί `@var` για κατάσταση στιγμιότυπου, `$var` για καθολικές, και ένα σκέτο `var` για τοπικές. Η Perl χρησιμοποιεί `$x` για μία τιμή, `@x` για μια διατεταγμένη λίστα, `%x` για μια αντιστοίχιση κλειδιού/τιμής - το sigil σας λέει το *σχήμα* των δεδομένων, και είναι το ίδιο είτε η μεταβλητή είναι τοπική είτε καθολική. Η πρώτη ενότητα δεν αφορά τίποτε άλλο.
- **Δεν είναι τα πάντα αντικείμενο.** Στη Ruby τα `"foo".upcase` και `[1, 2].sum` είναι κλήσεις μεθόδων επειδή οι αριθμοί, οι συμβολοσειρές και οι πίνακες είναι όλα αντικείμενα. Η Perl είναι πρακτική: ένα βαθμωτό δεν είναι αντικείμενο, δεν υπάρχει καθολική βασική κλάση, και αυτές οι λειτουργίες είναι απλές συναρτήσεις

- `uc "foo"`, `sum @a`. Καλείτε μεθόδους μόνο σε πράγματα που κάνατε bless σε μια κλάση.

Every example here runs on `pperl`, a Rust reimplementation of Perl 5.44. Each Perl snippet was executed on `pperl`; the `# ...` comments show its real output. Start your scripts with:

```
use v5.44;
```

Αυτή η μία γραμμή ενεργοποιεί το `strict` (καμία τυχαία καθολική μεταβλητή), τα `warnings`, την ενσωματωμένη `say` (ένα `print` με τελικό χαρακτήρα νέας γραμμής, όπως το `puts` της Ruby), τις υπογραφές υπορουτινών, και το σύγχρονο σύνολο χαρακτηριστικών που χρησιμοποιείται σε όλον αυτόν τον οδηγό.

Πώς να το διαβάσετε αυτό

- *Μεταβλητές και sigils* - η μετάβαση $/@/%.
- *Συμβολοσειρές* - παρεμβολή και συνένωση.
- *Αριθμοί* - ένας επιφανειακός τύπος, διαίρεση κινητής υποδιαστολής.
- *Πίνακες και λίστες* - ο `Array` σας, συν `map`/`grep` που δέχονται block.
- *Hashes* - το `Hash` σας, και πού πάνε τα σύμβολα.
- *Περιβάλλον* - η έννοια για την οποία η Ruby δεν έχει λέξη. Διαβάστε αυτήν.
- *Αλήθεια* - τι μετράει ως ψευδές, και διαφέρει έντονα.
- *Ροή ελέγχου* - `if`, βρόχοι, οι μεταθεματικές μορφές.
- *Υπορουτίνες* - το `def` γίνεται `sub`.
- *Αναφορές* - εμφωλευμένα δεδομένα, και blocks ως τιμές.
- *Κανονικές εκφράσεις* - το regex ως σύνταξη, όπως της Ruby.
- *I/O αρχείων και ροών* - open, αναγνώσεις γραμμών, slurp.
- *Αρθρώματα και πακέτα* - το `require` γίνεται `use`.
- *Αντικείμενα* - η σύγχρονη σύνταξη `class`.
- *Χειρισμός σφαλμάτων* - το `begin`/`rescue` γίνεται `eval` ή `try`/`catch`.
- *Ιδιώματα* - τα καθημερινά μονόγραμμα.
- *Παγίδες ερχόμενοι από τη Ruby* - σαρώστε τες πριν γράψετε.

Μεταβλητές και sigils

Στη Ruby ένα sigil σας λέει την *εμβέλεια*: το `@x` είναι μεταβλητή στιγμιότυπου, το `$x` καθολική, το `x` τοπική. Στην Perl το **sigil σας λέει το σχήμα**: `$` για μία τιμή, `@` για μια διατεταγμένη λίστα, `%` για μια αντιστοίχιση κλειδιού/τιμής. Το `my` δηλώνει μια μεταβλητή με λεξιλογική εμβέλεια, το αντίστοιχο της Perl για μια συνηθισμένη τοπική σύνδεση - είναι ο τρόπος που παίρνετε μια τοπική στην Perl, εκεί που η Ruby απλώς χρησιμοποιεί ένα σκέτο όνομα.

```
name  = "Ada"
langs = ["Ruby", "Perl"]
ages  = {"Ada" => 36, "Yukihiro" => 60}
```

```
use v5.44;
my $name  = "Ada";
my @langs = ("Ruby", "Perl");
my %ages  = (Ada => 36, Yukihiro => 60);
say $name;            # Ada
say $langs[0];        # Ruby
say $ages{Ada};       # 36
```

Η παγίδα: όταν φτάνετε *μέσα* σε ένα σύνολο για ένα στοιχείο, το sigil ακολουθεί την τιμή που παίρνετε πίσω, όχι τον περιέκτη. Ένα μεμονωμένο στοιχείο του `@langs` είναι βαθμωτό, οπότε είναι `$langs[0]`, όχι `@langs[0]`· μία τιμή από το `%ages` είναι `$ages{Ada}`. Η μορφή `@`/`%` είναι όλη η συλλογή· η μορφή `$` είναι ένα κομμάτι της. (Και ξεχάστε την αντιστοίχιση της Ruby: ένα `$x` της Perl δεν είναι ποτέ «καθολική» όπως είναι το `$x` της Ruby - το `$` εδώ σημαίνει απλώς «μία τιμή».)

Συμβολοσειρές

Τα διπλά εισαγωγικά παρεμβάλλουν μεταβλητές, τα μονά όχι, ακριβώς όπως τα `"..."` έναντι `'...'` της Ruby. Η μορφή `#{expr}` της Ruby έχει ένα άμεσο αντίστοιχο για σκέτες μεταβλητές και ένα ελαφρώς μακρύτερο για αυθαίρετες εκφράσεις. Η συνένωση είναι `.`, όχι `+`.

```
name     = "Ada"
greeting = "Hello, #{name}"
total    = "Total: #{6 * 7}"
shout    = "AB" * 3
```

```
use v5.44;
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
```

Η παγίδα: ένα σκέτο `"$name"` παρεμβάλλει μια μεταβλητή ακριβώς όπως το `#{name}` της Ruby, αλλά για να παρεμβάλετε μια *έκφραση* την τυλίγετε στη μορφή `@{[ ... ]}` (φτιάχνει έναν πρόσκαιρο πίνακα και τον ενσωματώνει). Η επανάληψη συμβολοσειράς είναι `x`, όχι `*`, και ο τελεστής ένωσης είναι `.`, όχι `+`· το `+` σε συμβολοσειρές τις εξαναγκάζει σε αριθμούς και προειδοποιεί. Τα στοιχεία πίνακα και οι τιμές hash παρεμβάλλονται και μέσα σε διπλά εισαγωγικά: το `"First: $langs[0]"` δουλεύει όπως είναι γραμμένο.

Αριθμοί

Η Perl έχει έναν αριθμητικό τύπο στην επιφάνεια· δεν ξεχωρίζετε `Integer` από `Float`. Η διαίρεση πάντα παράγει έναν αριθμό κινητής υποδιαστολής, όπως η διαίρεση `Float` της Ruby όταν κάποιος από τους τελεστέους είναι float.

```
puts 7.0 / 2        # 3.5
puts 7 / 2          # 3   (integer division in Ruby)
puts 2 ** 10        # 1024
puts 10 % 3         # 1
```

```
use v5.44;
say 7 / 2;          # 3.5   (always float)
say int(7 / 2);     # 3
say 2 ** 10;        # 1024
say 10 % 3;         # 1
```

Η παγίδα: οι τελεστές σύγκρισης και ισότητας έρχονται σε δύο οικογένειες, και επιλέγετε με βάση τον *τύπο των τελεστέων*, όχι των μεταβλητών. Οι αριθμοί χρησιμοποιούν ==, !=, <, >, <=>· οι συμβολοσειρές χρησιμοποιούν eq, ne, lt, gt, cmp. Αυτό δεν έχει αντίστοιχο στη Ruby - το == της Ruby αποστέλλει με βάση την κλάση του αντικειμένου, ενώ το == της Perl σημαίνει πάντα *αριθμητική* ισότητα. Το `"10" == "10.0"` είναι αληθές (αριθμητικά), αλλά το `"10" eq "10.0"` είναι ψευδές (συμβολοσειρά). Η χρήση == σε συμβολοσειρές, ή eq σε αριθμούς, είναι ένα συχνό και σιωπηλό σφάλμα.

Πίνακες και λίστες

Ένας πίνακας της Perl είναι ο `Array` σας στη Ruby. Η μεγάλη μετατόπιση είναι ότι οι κλήσεις μεθόδων γίνονται συναρτήσεις: τα `arr.push(x)`, `arr.pop`, `arr.length` της Ruby γίνονται `push @arr, x`, `pop @arr`, `scalar @arr` - ο πίνακας είναι όρισμα, όχι αποδέκτης.

```
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 v5.44;
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)
```

Οι επαναλήπτες της Ruby που δέχονται block είναι το πλουσιότερο μέρος αυτής της αντιστοίχισης. Τα `map`, `select`, και `sort_by` γίνονται `map`, `grep`, και `sort` της Perl, που δέχονται ένα block `{ ... }` ακριβώς όπως οι μέθοδοι της Ruby. Μέσα στο block, το `$_` είναι το τρέχον στοιχείο - η σιωπηρή μεταβλητή block της Perl, ο ρόλος που παίζει η παράμετρος `|x|` της Ruby.

```
squares = nums.map    { |x| x * x }
evens   = nums.select { |x| x.even? }
desc    = nums.sort   { |a, b| b <=> a }
```

```
use v5.44;
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
```

Η παγίδα: το `sort` συγκρίνει ως συμβολοσειρές από προεπιλογή, οπότε το `sort (10, 9, 100)` δίνει `10 100 9`. Για αριθμούς πρέπει να περάσετε έναν συγκριτή: `sort { $a <=> $b } @nums`. Τα `$a` και `$b` είναι τα δύο στοιχεία που συγκρίνονται (το αντίστοιχο του `|a, b|` της Ruby), και το `<=>` είναι ο αριθμητικός τρίτιμος τελεστής - το ίδιο ακριβώς διαστημόπλοιο που ξέρετε από τη Ruby (το `cmp` είναι το αδελφάκι του για συμβολοσειρές).

Hashes

Ένα hash της Perl είναι το Hash σας στη Ruby. Δηλώνεται με %, και προσπελαύνεται μία τιμή τη φορά με `$h{key}`. Το παχύ κόμμα => που ήδη χρησιμοποιείτε στις κυριολεξίες hash της Ruby δουλεύει κι εδώ, και βάζει αυτόματα εισαγωγικά στο bareword στα αριστερά του.

```
h = {apple: 3, pear: 5, plum: 2}
puts h[:apple]                          # 3
puts h.keys.sort.join(",")              # apple,pear,plum
puts h.key?(:pear)                      # true
h.delete(:plum)
h.sort.each { |k, v| puts "#{k} => #{v}" }
```

```
use v5.44;
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}";
}
```

Η παγίδα: η Perl **δεν έχει τύπο συμβόλου**. Εκεί που η Ruby φτάνει στο `:apple` ως ένα ελαφρύ, εσωτερικευμένο κλειδί, η Perl απλώς χρησιμοποιεί τη συμβολοσειρά `"apple"` - και το παχύ κόμμα => της βάζει αυτόματα εισαγωγικά, οπότε τα `apple => 3` και `$h{apple}` δεν χρειάζονται εισαγωγικά. Ένα σύμβολο της Ruby και μια συμβολοσειρά της Ruby είναι διαφορετικά αντικείμενα· στην Perl υπάρχει μόνο η συμβολοσειρά, οπότε η διάκριση `:sym` έναντι `"sym"` απλώς εξαφανίζεται. Ο έλεγχος μέλους είναι `exists $h{key}`, όχι το `key?` της Ruby, και ένα hash δεν έχει σειρά - επαναλάβετε με `sort keys %h` όταν θέλετε μια σταθερή.

Περιβάλλον

Αυτό είναι το κεφάλαιο που πρέπει να διαβάσετε δύο φορές. Η Ruby δεν έχει αντίστοιχο, και σχεδόν κάθε ιστορία «η Perl είναι παράξενη» ανάγεται σε αυτό.

Κάθε έκφραση στην Perl τρέχει είτε σε **περιβάλλον λίστας** είτε σε **βαθμωτό περιβάλλον**, που καθορίζεται από το πού εμφανίζεται, όχι από το τι είναι. Ο ίδιος πίνακας συμπεριφέρεται διαφορετικά σε καθένα. Η ανάθεση ενός πίνακα σε ένα *βαθμωτό* δίνει το μήκος του· η ανάθεσή του σε μια *λίστα* (ένα σύνολο μεταβλητών σε παρενθέσεις) αντιγράφει στοιχεία. Η Ruby ποτέ δεν υπερφορτώνει την ανάθεση έτσι - το `a = arr` πάντα απλώς δίνει ψευδώνυμο στον πίνακα - οπότε αυτό είναι πραγματικά καινούργιο.

```
use v5.44;
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
```

Οι παρενθέσεις στα αριστερά είναι όλη η διαφορά. Το `my $x = @list` ρωτάει «πόσα;» και παίρνει 3. Το `my ($x) = @list` ρωτάει «δώσε μου τα στοιχεία» και παίρνει `10`. Το αντανακλαστικό της Ruby διαβάζει και τα δύο ως «ανάθεσε τον πίνακα», που είναι ακριβώς η παγίδα.

Μια υπορουτίνα μπορεί να ρωτήσει σε ποιο περιβάλλον κλήθηκε, με `wantarray`, και να επιστρέψει διαφορετικά πράγματα ανάλογα:

```
use v5.44;
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
```

Οι ενσωματωμένες το χρησιμοποιούν κι αυτές. Το `scalar @array` επιβάλλει βαθμωτό περιβάλλον για να πάρει ένα πλήθος· μια αντιστοίχιση regex με `/g` επιστρέφει τη λίστα των αντιστοιχιών σε περιβάλλον λίστας και ένα αληθές/ψευδές σε βαθμωτό περιβάλλον. Το κλασικό ιδίωμα για το «μέτρα τις αντιστοιχίες» επιβάλλει μια ανάθεση λίστας σε βαθμωτό περιβάλλον με τη λεγόμενη μορφή *countof*:

```
use v5.44;
my $n = () = "a1b2c3" =~ /\d/g;
say $n;                     # 3
```

Η παγίδα: δεν υπάρχει μία μόνο απάντηση στο «τι κάνει αυτός ο πίνακας» - εξαρτάται από την περιβάλλουσα έκφραση. Όταν κάτι επιστρέφει έναν αριθμό που περιμένατε να είναι

λίστα (ή το αντίστροφο), το περιβάλλον είναι ο πρώτος ύποπτος. Φτάστε στο `scalar` για να επιβάλετε ένα πλήθος και σε παρενθέσεις στα αριστερά για να επιβάλετε αντιγραφή στοιχείων.

Αλήθεια τιμών

Αυτή είναι η πιο απότομη ρήξη από τη Ruby σε όλον τον οδηγό. Στη Ruby μόνο τα `nil` και `false` είναι ψευδή - τα `0` και `""` είναι και τα δύο **αληθή**. Στην Perl οι ψευδείς τιμές είναι μια σύντομη, συγκεκριμένη λίστα: ο αριθμός `0`, η κενή συμβολοσειρά `""`, η συμβολοσειρά `"0"`, και το undef (το `nil` της Perl). Όλα τα άλλα είναι αληθή.

```
use v5.44;
for my $v (0, 1, "", "0", "00", "0.0", "false") {
    say "'$v' is ", ($v ? "true" : "false");
}
```

Αυτό τυπώνει, ακριβώς:

```
'0' is false
'1' is true
'' is false
'0' is false
'00' is true
'0.0' is true
'false' is true
```

Η παγίδα: τα `0` και `""` είναι **ψευδή** στην Perl αλλά **αληθή** στη Ruby - η πιο ανατρεπτική συνήθεια που πρέπει να ξεμάθετε. Ένας φύλακας της Ruby όπως το `if count` πυροδοτείται για `count == 0`· το `if ($count)` της Perl όχι. Από την άλλη πλευρά, τα `"00"` και `"0.0"` είναι **αληθή** στην Perl παρόλο που μοιάζουν αριθμητικά μηδέν, επειδή μόνο η ακριβής συμβολοσειρά `"0"` είναι ψευδής. Και σε αντίθεση με τη Ruby, όπου ίσως στηρίζεστε σε ελέγχους `nil`, η Perl ξεχωρίζει το «ψευδές» από το «απροσδιόριστο» με το defined-or `//`, που πέφτει πίσω μόνο στο undef (όχι στο `0` ή στο `""`):

```
use v5.44;
my %cfg = (host => "localhost");
my $port = $cfg{port} // 8080;   # key absent -> undef -> 8080
say $port;                       # 8080
```

Ροή ελέγχου

Οι δεσμευμένες λέξεις είναι οικείες· η γραφή διαφέρει. Το `elsif` είναι το `elsif` της Perl (και το `elsif` της Ruby - μία από τις λίγες ακριβείς αντιστοιχίες), και τα μπλοκ είναι σε άγκιστρα, χωρίς `then` και χωρίς τερματικό `end`.

```
if    n < 0  then sign = "neg"
elsif n == 0 then sign = "zero"
else              sign = "pos"
end

nums.each { |x| puts x }
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
while running
  step
end
```

```
use v5.44;
my $sign;
if    ($n < 0)  { $sign = "neg"  }
elsif ($n == 0) { $sign = "zero" }
else            { $sign = "pos"  }

for my $x (@nums) { say $x }

while ($running) { step() }
```

Οι **μεταθεματικοί τροποποιητές** της Perl είναι το άμεσο αντίστοιχο των τελικών `puts x if cond` και `do_it while cond` της Ruby - μια απλή εντολή που φέρει τη δική της συνθήκη ή βρόχο:

```
use v5.44;
say $_ for @nums;                # loop, one statement
say "negative" if $n < 0;        # guarded statement
$sum += $_ for @nums;            # accumulate
```

Και τα `unless`/`until` είναι οι αρνημένες μορφές των `if`/`while`, ακριβώς όπως τα `unless`/`until` της Ruby: `say "ok" unless $error`.

Η παγίδα: ο τριαδικός τελεστής χρησιμοποιεί `?`/`:`, όχι το `cond ? a : b` της Ruby - που είναι στην πραγματικότητα πανομοιότυπο, οπότε αυτό είναι δωρεάν. Δεν υπάρχει μορφή έκφρασης `x if c else y`· αλυσιδώστε τριαδικούς τελεστές αντ” αυτού: `my $sign = $n < 0 ? "neg" : $n == 0 ? "zero" : "pos"`.

Υπορουτίνες

Το `def` γίνεται `sub`. Η σύγχρονη Perl έχει υπογραφές που μοιάζουν πολύ με τις λίστες παραμέτρων της Ruby, συμπεριλαμβανομένων των προεπιλογών. Και τα καλά νέα που θέλει να ακούσει ένας προγραμματιστής της Ruby: **η τελευταία αποτιμημένη έκφραση είναι η τιμή επιστροφής**, ακριβώς όπως στη Ruby - το `return` είναι προαιρετικό.

```
def greet(name, greeting = "Hello")
  "#{greeting}, #{name}!"
end

def total(*nums)
  nums.sum
end
```

```
use v5.44;
sub greet ($name, $greeting = "Hello") {
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    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
```

Τα ονοματισμένα ορίσματα της Ruby αντιστοιχίζονται σε μια παράμετρο hash, αφού τα ζεύγη `key => value` ισοπεδώνονται σε μια λίστα που μια υπογραφή `%opts` συγκεντρώνει:

```
use v5.44;
sub connect_to (%opts) {
    return "host=$opts{host} port=$opts{port}";
}
say connect_to(host => "localhost", port => 8080);
# host=localhost port=8080
```

Η παγίδα: μια sub επιστρέφει την τιμή της τελευταίας της έκφρασης ακόμη και χωρίς `return` (το αντανακλαστικό της Ruby μεταφέρεται), αλλά στην Perl αυτή η επιστροφή είναι επίσης **ευαίσθητη στο περιβάλλον** (δείτε *Περιβάλλον*). Αν κάνετε `return @list`, ο καλών παίρνει τη λίστα σε περιβάλλον λίστας αλλά το *πλήθος* σε βαθμωτό περιβάλλον. Να είστε εσκεμμένοι σχετικά με το τι επιστρέφετε όταν μπορεί να χρησιμοποιηθεί με οποιονδήποτε τρόπο.

Αναφορές

Οι πίνακες και τα hashes της Perl δεν εμφωλεύονται απευθείας· τα εμφωλεύετε μέσω **αναφορών**, ενός βαθμωτού που δείχνει σε ένα σύνολο. Στη Ruby κάθε περιέκτης είναι ήδη αναφορά, οπότε η εμφώλευση είναι εκεί αόρατη σε εσάς· στην Perl παίρνετε μια αναφορά με `\` (ή φτιάχνετε μία με `[ ]`/`{ }`) και την ακολουθείτε με `->`.

```
data = {name: "Ada", langs: ["Perl", "Ruby"]}
puts data[:langs][1]      # Ruby
data[:langs].push("Python")
```

```
use v5.44;
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
```

Το `{ ... }` φτιάχνει μια ανώνυμη αναφορά hash και το `[ ... ]` μια ανώνυμη αναφορά πίνακα - τα κυριολεκτικά δομικά στοιχεία των εμφωλευμένων δεδομένων, που αντιστοιχούν στις κυριολεξίες `{}` και `[]` της Ruby. Το βέλος `->` ακολουθεί μια αναφορά κατά ένα βήμα· ανάμεσα σε δύο δείκτες είναι προαιρετικό, οπότε τα `$data->{langs}[1]` και `$data->{langs}->[1]` είναι ίδια.

Το όφελος για έναν προγραμματιστή της Ruby είναι τα blocks-ως-τιμές. Ένα block, `proc`, ή `lambda` της Ruby γίνεται μια ανώνυμη sub της Perl: ένα closure αποθηκευμένο σε ένα βαθμωτό, που καλείται με `->`. Κλείνει πάνω από τις λεξιλογικές του μεταβλητές ακριβώς όπως ένα block της Ruby:

```
use v5.44;
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
```

Η παγίδα: μια ανώνυμη sub της Perl είναι μια τιμή που πρέπει να *αποθηκεύσετε και να καλέσετε ρητά* με `->( )` - δεν υπάρχει σιωπηρό `yield`, κανένα block που να περνιέται αόρατα δίπλα στα ορίσματα. Όταν μια ενσωματωμένη θέλει ένα block (`map`/`grep`/`sort`, ή `first`/`reduce` από το `List::Util`), γράφετε το `{ ... }` ενσωματωμένα· όταν θέλετε να περάσετε ένα καλέσιμο τριγύρω ως δεδομένα, φτιάχνετε ένα `sub { ... }` και το καλείτε μέσω του βαθμωτού του. Οι μεταθεματικές μορφές αποαναφοράς (`->@*`, `->%*`, `->$*`) είναι ο σύγχρονος, ευανάγνωστος τρόπος να αποαναφέρετε ένα ολόκληρο σύνολο.

Κανονικές εκφράσεις

Καλά νέα: το regex της Perl θα σας φανεί σαν στο σπίτι σας. Η Ruby δανείστηκε την κυριολεξία regex `/.../`, τον τελεστή `=~`, και τις μεταβλητές σύλληψης `$1`/`$2` κατευθείαν από την Perl. Ο τελεστής αντιστοίχισης `m//`, η αντικατάσταση `s///`, και οι ονομαστικές συλλήψεις στο `%+` το συμπληρώνουν.

```
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 v5.44;
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
```

Μια αντιστοίχιση `/g` σε περιβάλλον λίστας επιστρέφει *όλες* τις συλλήψεις, το άμεσο αντίστοιχο του `String#scan` της Ruby:

```
use v5.44;
my @nums = "x1y2z3" =~ /(\d)/g;      # 1 2 3
say "@nums";
```

Η παγίδα: εκεί που το `gsub` της Ruby επιστρέφει μια νέα συμβολοσειρά και αφήνει την αρχική ήσυχη, το `s///` της Perl τροποποιεί τον στόχο του *επιτόπια* και επιστρέφει τον αριθμό των αντικαταστάσεων. Γι" αυτό το παράδειγμα αντιγράφει πρώτα με `(my $t = "a,b,c")` και μετά αντικαθιστά στο αντίγραφο. Για να πάρετε μια νέα συμβολοσειρά χωρίς να αγγίξετε την αρχική - τη συμπεριφορά `gsub` που περιμένετε - προσθέστε τη σημαία `/r`: `my $t = $s =~ s/,/;/gr`.

I/O αρχείων και ροών

Το ιδίωμα είναι το open με τρία ορίσματα και ένα λεξιλογικό filehandle, το αντίστοιχο του `File.open(path, mode) do |f| ... end` της Ruby. Ο τελεστής με γωνιώδεις αγκύλες `<$fh>` διαβάζει μια γραμμή, με τον ίδιο ρόλο όπως τα `f.each_line` ή `f.gets` της Ruby.

```
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 v5.44;
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;
```

Για να ρουφήξετε (slurp) όλες τις γραμμές μονομιάς, διαβάστε το handle σε περιβάλλον λίστας:

```
use v5.44;
open my $fh, '<', "/tmp/sample.txt" or die $!;
my @lines = <$fh>;          # all lines, including newlines
say scalar @lines;          # 3
```

Η παγίδα: μια γραμμή που διαβάστηκε με `<$fh>` κρατάει τον τελικό χαρακτήρα νέας

γραμμής· καλέστε `chomp` για να τον αφαιρέσετε - το `line.chomp` της Ruby είναι το ίδιο ρήμα, και η Perl δάνεισε το όνομα. Και το open επιστρέφει ψευδές σε αποτυχία αντί να εγείρει εξαίρεση όπως κάνει το `File.open` της Ruby, οπότε το ιδίωμα `or die "...: $!"` είναι υποχρεωτικό· το `$!` είναι το μήνυμα σφάλματος συστήματος, το αντίστοιχο της Perl για το μήνυμα στην εξαίρεση `Errno` της Ruby.

Αρθρώματα και πακέτα

Τα `require`/`include` γίνονται `use`. Ο διαχωριστής χώρου ονομάτων είναι το `::`, που ήδη ξέρετε από σταθερές της Ruby όπως το `Set::Foo`· η Perl το χρησιμοποιεί παντού, και για πακέτα και για τα ονόματα μέσα τους. Ένα άρθρωμα `Foo::Bar` βρίσκεται στο `Foo/Bar.pm` στη διαδρομή ένταξης `@INC` (το `$LOAD_PATH` της Perl). Η λίστα εισαγωγής ενός αρθρώματος είναι μια επιλογή ονομάτων, όπως όταν τραβάτε συγκεκριμένες μεθόδους εντός εμβέλειας.

```ruby
require "set"
nums = [1, 2, 3, 4].sum
biggest = [3, 7, 2].max
```

```perl
use v5.44;
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
```

Η τυπική βιβλιοθήκη είναι μεγάλη, και πολλά από αυτά για τα οποία θα κατέφευγες στα RubyGems έρχονται στον πυρήνα ή στο CPAN, το αρχείο πακέτων της Perl. Τα `List::Util`, `Scalar::Util`, `POSIX`, `File::Spec`, και `Data::Dumper` είναι τα καθημερινά.

Η παγίδα: το `use` τρέχει κατά τη *μεταγλώττιση*, όχι στο σημείο του αρχείου όπου εμφανίζεται, οπότε η σειρά των γραμμών `use` σε σχέση με τον κώδικά σας δεν έχει σημασία όπως μπορεί να έχει η τοποθέτηση του `require`. Η εισαγωγή είναι προαιρετική ανά όνομα: το `use List::Util qw(sum0 max)` φέρνει ακριβώς τα `sum0` και `max` στον χώρο ονομάτων σας, τίποτε άλλο - δεν υπάρχει monkey-patching ενσωματωμένων τύπων όπως η Ruby σας αφήνει να ξανανοίξετε το `Array`.

Αντικείμενα

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 5.44 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).

```ruby
class Point
  attr_accessor :x, :y
  def initialize(x: 0, y: 0)
    @x = x
    @y = y
  end
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```ruby
  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)
```

```perl
use v5.44;
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)
```

Ο κατασκευαστής δημιουργείται για εσάς, το αντίστοιχο του `attr_accessor` συν `initialize` της Ruby συμπτυγμένων μέσα στις δηλώσεις πεδίων: το `:param` σημειώνει ένα πεδίο που η αυτόματα κατασκευασμένη `new` δέχεται ως ονοματισμένο όρισμα, και το `= 0` είναι η προεπιλογή του. Η κληρονομικότητα είναι `:isa`, το αντίστοιχο του `class Point3D < Point` της Ruby:

```perl
use v5.44;
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);
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
say $q->to_string, " z=", $q->z;   # (1, 2) z=3
```

The gotcha: the `class` feature is still marked experimental in Perl 5.44, 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*.

Χειρισμός σφαλμάτων

Το `begin/rescue/ensure` της Ruby έχει δύο αντίστοιχα. Το κλασικό τυλίγει κώδικα σε `eval { }` και εξετάζει το `$@` (το σφάλμα που πιάστηκε) μετά - σημειώστε ότι το `eval` εδώ είναι *παγίδευση* εξαίρεσης, όχι το `eval` μιας συμβολοσειράς της Ruby:

```
begin
  risky(-1)
rescue => e
  puts "caught: #{e.message}"
end
```

```
use v5.44;
my $result = eval { risky(-1) };
if ($@) {
    print "caught: $@";     # caught: negative!
}
```

Η σύγχρονη μορφή είναι το `try/catch/finally`, που διαβάζεται σχεδόν ακριβώς όπως το `begin/rescue/ensure`:

```
use v5.44;
use feature 'try';

try {
    risky(-4);
}
catch ($e) {
    print "caught: $e";     # caught: negative!
}
finally {
    say "done";             # done   (runs either way, like ensure)
}
```

όπου η πλευρά που εγείρει είναι το `die` - το `raise` της Perl:

```
use v5.44;
sub risky ($n) {
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    die "negative!\n" if $n < 0;
    return sqrt $n;
}
```

Η παγίδα: το `die` με μια συμβολοσειρά που τελειώνει σε `\n` παράγει ακριβώς αυτό το μήνυμα· το `die` με μια συμβολοσειρά που *δεν* τελειώνει σε χαρακτήρα νέας γραμμής παίρνει αυτόματα την προσθήκη `" at SCRIPT line N.\n"`. Τελειώστε τις συμβολοσειρές σφάλματός σας με `\n` όταν θέλετε ένα καθαρό μήνυμα και παραλείψτε το όταν θέλετε το ίχνος αρχείου-και-γραμμής. Το μπλοκ `finally`, όπως το `ensure` της Ruby, εκτελείται είτε ρίχτηκε εξαίρεση είτε όχι. Σημειώστε επίσης τις δύο μεταβλητές σφάλματος: το `$@` είναι η τελευταία εξαίρεση που πιάστηκε (το `rescue => e` της Ruby), ενώ το `$!` είναι το τελευταίο σφάλμα *συστήματος* (ένα αποτυχημένο open). Υπό το `pperl`, το χαρακτηριστικό `try/catch/finally` τυπώνει μια ειδοποίηση «experimental» στο stderr· το stdout δεν επηρεάζεται.

Ιδιωματισμοί

Μερικά μοτίβα που θα βλέπετε καθημερινά και θα θέλετε στα δάχτυλά σας.

Ταξινομήστε μια λίστα με βάση ένα υπολογισμένο κλειδί. Η Ruby έχει το `sort_by` ενσωματωμένο· η φτιαγμένη-με-το-χέρι μορφή της Perl είναι ο Schwartzian transform - διακόσμησε, ταξινόμησε, αφαίρεσε τη διακόσμηση - και αξίζει να την αναγνωρίζετε με την πρώτη ματιά:

```
use v5.44;
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
```

Φτιάξτε ένα hash από μια λίστα, το αντίστοιχο του `words.to_h { |w| [w, w.length] }` της Ruby:

```
use v5.44;
my @words = qw(apple pear fig);
my %len = map { $_ => length $_ } @words;
say "$_ => $len{$_}" for sort keys %len;
# apple => 5
# fig => 3
# pear => 4
```

Και το `pperl` είναι ένα εργαλείο γραμμής εντολών στην εξειδικευμένη θέση των awk/sed - τον ρόλο που παίζουν τα `ruby -e` και `ruby -ne` για κόλλημα κελύφους. Το μονόγραμμο με σιωπηρό βρόχο είναι το καθημερινό αντανακλαστικό του προγραμματιστή της Perl:

```
pperl -lane '$s += $F[-1]; END { print $s }' data.txt
```

Η παγίδα: διαβάστε τον *οδηγό Μονόγραμμων* για τους διακόπτες (`-n`, `-p`, `-a`, `-l`, `-e`) και τις δεκάδες συνταγές που είναι χτισμένες πάνω τους· είναι ένα από τα

πράγματα με τη μεγαλύτερη μόχλευση που μπορείτε να μάθετε ερχόμενοι από υπόβαθρο κελύφους-και-Ruby.

Παγίδες ερχόμενοι από τη Ruby

Οι παγίδες, συγκεντρωμένες για μια τελική σάρωση πριν αρχίσετε να γράφετε:

- **Τα sigils σημειώνουν σχήμα, όχι εμβέλεια.** Τα `$x`/`@x`/`%x` λένε μία/λίστα/αντιστοίχιση· δεν είναι οι δείκτες στιγμιότυπου/καθολικής/τοπικής της Ruby. Ένα στοιχείο του `@a` είναι `$a[0]`· μία τιμή του `%h` είναι `$h{k}`.
- **Δεν είναι τα πάντα αντικείμενο.** Τα `uc "foo"` και `sum @a` είναι συναρτήσεις, όχι μέθοδοι σε μια συμβολοσειρά ή πίνακα. Καλείτε μεθόδους μόνο σε αναφορές που έχουν γίνει bless.
- **Η αλήθεια είναι η μεγάλη υπόθεση.** Τα `0` και `""` είναι **ψευδή** στην Perl αλλά **αληθή** στη Ruby. Το `if ($count)` είναι ψευδές όταν το `$count` είναι `0`. Μόνο τα `0`, `""`, `"0"`, και `undef` είναι ψευδή· τα `"00"` και `"0.0"` είναι **αληθή**.
- **Καθόλου σύμβολα.** Το `:name` δεν έχει αντίστοιχο στην Perl· τα κλειδιά hash είναι σκέτες συμβολοσειρές, και το `=>` βάζει αυτόματα εισαγωγικά στο bareword στα αριστερά του.
- **Το `.` ενώνει συμβολοσειρές, το `+` προσθέτει αριθμούς.** Ποτέ μην κάνετε `+` δύο συμβολοσειρές για να τις συνενώσετε· αυτό τις εξαναγκάζει σε αριθμούς και προειδοποιεί. Η επανάληψη συμβολοσειράς είναι `x`, όχι `*`.
- **Δύο οικογένειες σύγκρισης.** `==`/`<`/`<=>` για αριθμούς, `eq`/`lt`/`cmp` για συμβολοσειρές. Επιλέξτε με βάση τον τύπο του τελεστέου, όχι με βάση τη μεταβλητή - σε αντίθεση με το `==` της Ruby που αποστέλλει με βάση την κλάση.
- **Το `sort` ταξινομεί ως συμβολοσειρές από προεπιλογή.** Περάστε `{ $a <=> $b }` για αριθμητική σειρά.
- **Περιβάλλον.** Το `my $x = @a` είναι το *μήκος*· το `my ($x) = @a` είναι το *πρώτο στοιχείο*. Οι παρενθέσεις είναι όλη η διαφορά. Ξαναδιαβάστε το *Περιβάλλον* όταν μια τιμή επιστρέφει με λάθος σχήμα.
- **Το `s///` επεξεργάζεται επιτόπια** και επιστρέφει ένα πλήθος, σε αντίθεση με το `gsub` της Ruby· χρησιμοποιήστε τη σημαία `/r` για να πάρετε μια νέα συμβολοσειρά αντ” αυτού.
- **Το `open` δεν εγείρει εξαίρεση.** Χρησιμοποιήστε `open ... or die "...: $!"`.
- **Τα blocks δεν είναι σιωπηρά.** Ένα αποθηκευμένο `sub { ... }` καλείται ρητά με `->( )`· δεν υπάρχει `yield` και κανένα αόρατο όρισμα block.
- **Non-ASCII source needs `use utf8;`.** Under `use v5.44;` a literal non-ASCII byte in your source is a compile error without it. Keep string data ASCII unless you specifically add `use utf8;`.
- **Το χαρακτηριστικό `class` τυπώνει πειραματικές ειδοποιήσεις στο stderr** στο `pperl`. Σωστή έξοδος, θορυβώδης ροή σφαλμάτων.

Από Lua σε Perl

Ξέρετε Lua. Χτίζετε ένα table χωρίς να το σκέφτεστε, φτάνετε στο `setmetatable` για να πάρετε αντικείμενα και τελεστές, και το `for k, v in pairs(t)` είναι μυϊκή μνήμη. Αυτός ο οδηγός αντιστοιχίζει καθεμία από αυτές τις συνήθειες στην Perl, ώστε να γίνετε παραγωγικοί στο `pperl` μέσα σε ένα απόγευμα αντί να ξαναμάθετε προγραμματισμό από το μηδέν.

Δύο σοκ φτάνουν μέσα στα πρώτα πέντε λεπτά, οπότε γνωρίστε τα τώρα:

- **Ο ένας τύπος table της Lua γίνεται δύο τύποι στην Perl.** Η Lua έχει ένα μοναδικό σύνολο, το table, που το χρησιμοποιείτε και ως πίνακα (`{1, 2, 3}`) και ως αντιστοίχιση (`{k = "v"}`). Η Perl το χωρίζει σε δύο διακριτά πράγματα: έναν διατεταγμένο `@array` με ακέραια κλειδιά και ένα `%hash` με κλειδιά συμβολοσειρών. Η απόφαση «είναι αυτό λίστα ή αναζήτηση;» εξαρχής είναι η κεντρική νέα συνήθεια, και οι ενότητες *Πίνακες* και *Hashes* δεν αφορούν τίποτε άλλο.
- **Οι μεταβλητές φορούν sigils.** Τα ονόματα της Lua είναι σκέτα: `x`, `t`, `f`. Στην Perl ένα βαθμωτό είναι `$x`, ένας πίνακας είναι `@x`, ένα hash είναι `%x`. Το sigil είναι μέρος του πώς διαβάζετε τη μεταβλητή, όχι απλώς διακόσμηση. Αυτή είναι η μεγαλύτερη επιφανειακή διαφορά από τη Lua, και η πρώτη ενότητα δεν αφορά τίποτε άλλο.

Every example here runs on `pperl`, a Rust reimplementation of Perl 5.44. Each Perl snippet was executed on `pperl`; the `# ...` comments show its real output. Start your scripts with:

```
use v5.44;
```

Αυτή η μία γραμμή ενεργοποιεί το `strict` (κάθε μεταβλητή πρέπει να δηλώνεται με `my`, η θεραπεία για τις επιρρεπείς σε ατυχήματα σιωπηρές καθολικές της Lua), τα `warnings`, την ενσωματωμένη `say` (ένα `print` με τελικό χαρακτήρα νέας γραμμής, όπως το `print` της Lua), τις υπογραφές υπορουτινών, και το σύγχρονο σύνολο χαρακτηριστικών που χρησιμοποιείται σε όλον αυτόν τον οδηγό.

Πώς να το διαβάσετε αυτό

- *Μεταβλητές και sigils* - η μετάβαση `$`/`@`/`%`, και το `local x` που γίνεται `my $x`.
- *Συμβολοσειρές* - παρεμβολή, και η συνένωση `..` που γίνεται `.`.
- *Αριθμοί* - ένας επιφανειακός τύπος, διαίρεση κινητής υποδιαστολής.
- *Πίνακες και λίστες* - το μισό από το table σας, 0-based όχι 1-based.
- *Hashes* - το άλλο μισό από το table σας.
- *Περιβάλλον* - η έννοια για την οποία η Lua δεν έχει λέξη. Διαβάστε αυτήν.
- *Αλήθεια* - τι μετράει ως ψευδές, και διαφέρει έντονα από τη Lua.
- *Ροή ελέγχου* - `if`, βρόχοι, οι μεταθεματικές μορφές.
- *Υπορουτίνες* - το `function` γίνεται `sub`· πραγματικά closures.
- *Αναφορές* - εμφωλευμένα δεδομένα και closures.
- *Κανονικές εκφράσεις* - πραγματικό regex, όχι μοτίβα της Lua.
- *I/O αρχείων και ροών* - open, αναγνώσεις γραμμών, slurp.

- *Αρθρώματα και πακέτα* - το `require` γίνεται `use`.
- *Αντικείμενα* - τα metatables γίνονται η σύγχρονη σύνταξη `class`.
- *Χειρισμός σφαλμάτων* - το `pcall` γίνεται `eval` ή `try`/`catch`.
- *Ιδιώματα* - τα καθημερινά μονόγραμμα.
- *Παγίδες ερχόμενοι από τη Lua* - σαρώστε τες πριν γράψετε.

Μεταβλητές και 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 v5.44` turns that into a compile error: every variable is `my` unless you say otherwise.

```lua
local name = "Ada"
local langs = {"Lua", "Perl"}
local ages = {Ada = 36, Linus = 56}
```

```perl
use v5.44;
my $name = "Ada";
my @langs = ("Lua", "Perl");
my %ages  = (Ada => 36, Linus => 56);
say $name;            # Ada
say $langs[0];        # Lua
say $ages{Ada};       # 36
```

Η παγίδα: όταν φτάνετε *μέσα* σε ένα σύνολο για ένα στοιχείο, το sigil ακολουθεί την τιμή που παίρνετε πίσω, όχι τον περιέκτη. Ένα μεμονωμένο στοιχείο του `@langs` είναι βαθμωτό, οπότε είναι `$langs[0]`, όχι `@langs[0]`· μία τιμή από το `%ages` είναι `$ages{Ada}`. Η μορφή `@`/`%` είναι όλη η συλλογή· η μορφή `$` είναι ένα κομμάτι της. Στη Lua το όνομα table `t` δεν αλλάζει ποτέ είτε γράφετε `t`, `t[1]`, ή `t.key`· στην Perl το αρχικό σημάδι αλλάζει ανάλογα με το τι προσπελαύνετε.

Συμβολοσειρές

Τα διπλά εισαγωγικά παρεμβάλλουν μεταβλητές, τα μονά όχι. Δεν χρειάζεται κανένα αντανακλαστικό `string.format` για τη συνηθισμένη περίπτωση, επειδή η προεπιλεγμένη συμβολοσειρά ήδη παρεμβάλλει. Η μεγάλη αλλαγή: η συνένωση είναι `.`, εκεί που η Lua χρησιμοποιεί `..`.

```lua
local name = "Ada"
local greeting = "Hello, " .. name
local shout = string.rep("AB", 3)
```

```perl
use v5.44;
my $name = "Ada";
my $greeting = "Hello, $name";   # Hello, Ada
my $literal  = 'Hello, $name';   # Hello, $name  (no interpolation)
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
my $joined   = "AB" . "CD";      # ABCD
my $shout    = "AB" x 3;         # ABABAB
```

Η παγίδα: ο τελεστής συνένωσης `..` της Lua δεν υπάρχει στην Perl· η μονή τελεία `.` ενώνει συμβολοσειρές και ο τελεστής `x` τις επαναλαμβάνει (το `string.rep` της Lua). Τα στοιχεία πίνακα και οι τιμές hash παρεμβάλλονται και μέσα σε διπλά εισαγωγικά, οπότε το `"First: $langs[0]"` δουλεύει όπως είναι γραμμένο, που εξαλείφει το μεγαλύτερο μέρος της αλυσίδωσης με `..` στην οποία καταφεύγει ένας προγραμματιστής της Lua.

Αριθμοί

Η Perl έχει έναν αριθμητικό τύπο στην επιφάνεια· δεν επιλέγετε μεταξύ ακεραίου και float, όπως ακριβώς η Lua 5.3+ παρουσιάζει έναν μοναδικό `number` στον περισσότερο κώδικα. Η διαίρεση πάντα παράγει έναν αριθμό κινητής υποδιαστολής.

```
print(7 / 2)        -- 3.5
print(math.floor(7 / 2))  -- 3
print(2 ^ 10)       -- 1024.0
print(10 % 3)       -- 1
```

```
use v5.44;
say 7 / 2;          # 3.5
say int(7 / 2);     # 3
say 2 ** 10;        # 1024
say 10 % 3;         # 1
```

Η παγίδα: οι τελεστές σύγκρισης και ισότητας έρχονται σε δύο οικογένειες, και επιλέγετε με βάση τον *τύπο των τελεστέων*, όχι των μεταβλητών. Οι αριθμοί χρησιμοποιούν `==`, `!=`, `<`, `>`, `<=>`· οι συμβολοσειρές χρησιμοποιούν `eq`, `ne`, `lt`, `gt`, `cmp`. Το `"10" == "10.0"` είναι αληθές (αριθμητικά), αλλά το `"10" eq "10.0"` είναι ψευδές (συμβολοσειρά). Η Lua έχει μόνο το `==` και δεν εξαναγκάζει τίποτε πέρα από τη γραμμή συμβολοσειράς/αριθμού· η Perl αντ” αυτού ζητάει από *εσάς* να επιλέξετε τη σύγκριση, και η χρήση `==` σε συμβολοσειρές (ή `eq` σε αριθμούς) είναι ένα συχνό σιωπηλό σφάλμα.

Πίνακες και λίστες

Εδώ είναι το μισό του μεγάλου διαχωρισμού του table. Μια ακολουθία της Lua (`{10, 20, 30}`, το table χρησιμοποιούμενο ως πίνακας) γίνεται ένας `@array` της Perl. Η πρώτη και πιο εμφανής διαφορά: **οι πίνακες της Perl είναι 0-based**, εκεί που οι πίνακες της Lua συμβατικά ξεκινούν από τον δείκτη 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 v5.44;
my @t = ("x", "y", "z");
say $t[0];          # x   (Perl: first element is index 0)
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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)
```

Οι μεταλλάκτες έχουν μετονομαστεί αλλά είναι οικείοι. Τα `table.insert`/`table.remove` γίνονται push/pop/shift/unshift:

```
use v5.44;
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)
```

Η Lua καταφεύγει σε ρητούς βρόχους `for i = 1, #t do ... end` για να μετασχηματίσει ή να φιλτράρει· η Perl έχει `map` (μετασχηματισμός) και `grep` (φιλτράρισμα), με το `$_` ως το τρέχον στοιχείο:

```
use v5.44;
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
```

Η παγίδα: πέρα από την ευρετηρίαση 0-based, το `#t` δεν έχει μία μόνο γραφή στην Perl - χρησιμοποιείτε `scalar @a` για το μήκος ενός πίνακα και `keys %h` για ενός hash. Και το `sort` συγκρίνει ως συμβολοσειρές από προεπιλογή, οπότε το `sort (10, 9, 100)` δίνει `10 100 9`. Για αριθμούς πρέπει να περάσετε έναν συγκριτή: `sort { $a <=> $b } @nums`. Τα `$a` και `$b` είναι τα δύο στοιχεία που συγκρίνονται, και το `<=>` είναι ο αριθμητικός τρίτιμος τελεστής (το `cmp` είναι το αδελφάκι του για συμβολοσειρές).

Hashes

Εδώ είναι το άλλο μισό του διαχωρισμού του table. Ένα table της Lua χρησιμοποιούμενο ως αντιστοίχιση (`{name = "Ada", age = 36}`, ή `t["key"]`) γίνεται ένα `%hash` της Perl. Δηλώνεται με `%`, και προσπελαύνεται μία τιμή τη φορά με `$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 v5.44;
my %h = (apple => 3, pear => 5, plum => 2);
say $h{apple};                          # 3
say join ",", sort keys %h;             # apple,pear,plum
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
say exists $h{pear} ? "yes" : "no";     # yes
delete $h{plum};
for my $k (sort keys %h) {
    say "$k => $h{$k}";                 # apple => 3 ... pear => 5
}
```

Η παγίδα: η Lua συγχωνεύει τον πίνακα και την αντιστοίχιση σε ένα table και κρατάει και ένα μέρος πίνακα και ένα μέρος hash μέσα του· η Perl τα κρατάει αυστηρά χωριστά, οπότε αποφασίζετε εξαρχής αν ένα πράγμα είναι ένας `@array` (ακέραιες θέσεις, διατεταγμένος) ή ένα `%hash` (κλειδιά συμβολοσειρών, αδιάτακτο). Ο έλεγχος μέλους είναι `exists $h{key}`, το αντίστοιχο του ελέγχου `h[k] ~= nil` της Lua, και το παχύ κόμμα `=>` είναι ο ιδιωματικός διαχωριστής κλειδιού/τιμής που βάζει αυτόματα εισαγωγικά στο bareword στα αριστερά του (οπότε το `apple => 3` δεν χρειάζεται εισαγωγικά). Ένα hash δεν έχει σειρά, οπότε επαναλάβετε με `sort keys %h` όταν θέλετε μια σταθερή· το `pairs` της Lua είναι αδιάτακτο για τον ίδιο λόγο.

Περιβάλλον

Αυτό είναι το κεφάλαιο που πρέπει να διαβάσετε δύο φορές. Η Lua δεν έχει γενικό αντίστοιχο, και σχεδόν κάθε ιστορία «η Perl είναι παράξενη» ανάγεται σε αυτό. Αλλά η Lua *όντως* σας δίνει τη μία γέφυρα που το κάνει να βγάζει νόημα: τις **πολλαπλές τιμές επιστροφής**.

Στη Lua, το `local a, b = f()` απλώνει τις πολλές τιμές επιστροφής μιας συνάρτησης σε πολλές μεταβλητές, και το `#t` ή το `select("#", ...)` συμπτύσσει μια λίστα σε ένα πλήθος. Η Perl γενικεύει αυτή την ιδέα σε μια ιδιότητα *κάθε* έκφρασης. Κάθε έκφραση τρέχει είτε σε **περιβάλλον λίστας** είτε σε **βαθμωτό περιβάλλον**, που καθορίζεται από το πού εμφανίζεται. Ο ίδιος πίνακας συμπεριφέρεται διαφορετικά σε καθένα: ανατεθειμένος σε ένα *βαθμωτό* δίνει το μήκος του, ανατεθειμένος σε μια *λίστα* (μεταβλητές σε παρενθέσεις) αντιγράφει στοιχεία.

```
use v5.44;
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
```

Οι παρενθέσεις στα αριστερά είναι όλη η διαφορά. Το `my $x = @list` ρωτάει «πόσα;» και παίρνει 3. Το `my ($x) = @list` ρωτάει «δώσε μου τα στοιχεία» και παίρνει 10. Η μορφή λίστας `my ($a, $b) = @list` είναι ακριβώς το `local a, b = ...` της Lua - η γέφυρα που ήδη ξέρετε.

Μια υπορουτίνα που επιστρέφει πολλές τιμές είναι το ίδιο με μια συνάρτηση της Lua με πολλές τιμές `return`· το περιβάλλον του καλούντος αποφασίζει τι επιστρέφεται:

```
use v5.44;
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
```

Μια sub μπορεί ακόμη και να ρωτήσει σε ποιο περιβάλλον κλήθηκε, με `wantarray`, και να απαντήσει διαφορετικά - κάτι που οι συναρτήσεις της Lua δεν μπορούν να κάνουν:

```
use v5.44;
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
```

Η παγίδα: δεν υπάρχει μία μόνο απάντηση στο «τι κάνει αυτός ο πίνακας» - εξαρτάται από την περιβάλλουσα έκφραση, εκεί που η Lua επιλύει το άπλωμα πολλαπλών τιμών μόνο στο συγκεκριμένο σημείο κλήσης. Όταν κάτι επιστρέφει έναν αριθμό που περιμένατε να είναι λίστα (ή το αντίστροφο), το περιβάλλον είναι ο πρώτος ύποπτος. Φτάστε στο `scalar` για να επιβάλετε ένα πλήθος και σε παρενθέσεις στα αριστερά για να επιβάλετε αντιγραφή στοιχείων.

Αλήθεια τιμών

Αυτή είναι μια απότομη ρήξη από τη Lua, οπότε επιβραδύνετε. Στη Lua **μόνο τα `nil` και `false` είναι ψευδή** - ο αριθμός `0` και η κενή συμβολοσειρά `""` είναι και τα δύο *αληθή*. Η Perl τραβάει τη γραμμή σε ένα εντελώς διαφορετικό σημείο. Οι ψευδείς τιμές της είναι: ο αριθμός `0`, η κενή συμβολοσειρά `""`, η συμβολοσειρά `"0"`, και το `undef` (το `nil` της Perl). **Όλα τα άλλα είναι αληθή.**

```
use v5.44;
for my $v (0, 1, "", "0", "00", "0.0", "false") {
    say "'$v' is ", ($v ? "true" : "false");
}
```

Αυτό τυπώνει, ακριβώς:

```
'0' is false
'1' is true
'' is false
'0' is false
'00' is true
'0.0' is true
'false' is true
```

Η παγίδα: το ένστικτο ενός προγραμματιστή της Lua ότι το `0` είναι αληθές θα αντιστρέψει

σιωπηλά τη σημασία κάθε `if ($count)`. Στην Perl τα `0` και `""` *είναι* ψευδή, οπότε το `if ($n)` είναι ψευδές όταν το `$n` είναι μηδέν - συνήθως αυτό που θέλετε, αλλά το αντίθετο της Lua. Από την άλλη μεριά, τα `"00"` και `"0.0"` είναι **αληθή** παρόλο που μοιάζουν αριθμητικά μηδέν, επειδή μόνο η ακριβής συμβολοσειρά `"0"` είναι ψευδής. Για μια επιλογή «χρησιμοποίησε αυτό αν είναι ορισμένο, αλλιώς μια προεπιλογή» - το ιδίωμα `x = y or default` της Lua, που σκοντάφτει στα `0` και `false` - χρησιμοποιήστε το defined-or `//` της Perl, που πέφτει πίσω μόνο στο undef (το `nil` της Perl):

```
use v5.44;
my %cfg = (host => "localhost");
my $port = $cfg{port} // 8080;   # key absent -> undef -> 8080
say $port;                       # 8080
```

Ροή ελέγχου

Οι δεσμευμένες λέξεις είναι οικείες· η γραφή διαφέρει. Το `elseif` της Lua είναι το `elsif` της Perl, δεν υπάρχει then/do/end, και τα μπλοκ είναι σε άγκιστρα αντί να οριοθετούνται με δεσμευμένες λέξεις.

```
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

while running do
    step()
end
```

```
use v5.44;
my $sign;
if    ($n < 0)  { $sign = "neg"  }
elsif ($n == 0) { $sign = "zero" }
else            { $sign = "pos"  }

for my $x (@nums) { say $x }

while ($running) { step() }
```

Η Perl προσθέτει δύο πράγματα που δεν έχει η Lua. Οι **μεταθεματικοί τροποποιητές** επιτρέπουν σε μια απλή εντολή να φέρει τη δική της συνθήκη ή βρόχο, κάτι που διαβάζεται φυσικά για μονόγραμμα:

```
use v5.44;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
say $_ for @nums;                    # loop, one statement
say "negative" if $n < 0;            # guarded statement
$sum += $_ for @nums;                # accumulate
```

Και τα unless/until είναι οι αρνημένες μορφές των if/while, για όταν η θετική διατύπωση διαβάζεται αδέξια: `say "ok" unless $error`.

Η παγίδα: η Perl δεν έχει εντολή `x if c else y`· η εκφραση υπό συνθήκη είναι ο τριαδικός τελεστής σε στυλ C με ?/:, τον οποίο η Lua συνήθως απομιμείται με `c and x or y`. Η μορφή της Perl είναι άμεση και αλυσιδώνεται: `my $sign = $n < 0 ? "neg" : $n == 0 ? "zero" : "pos"`. Ο έλεγχος βρόχου είναι last/next εκεί που η Lua έχει break και (στην 5.4) goto continue.

Υπορουτίνες

Το function γίνεται sub. Η σύγχρονη Perl έχει υπογραφές που ονοματίζουν τις παραμέτρους τους, συμπεριλαμβανομένων των προεπιλογών, αντί για το θεσιακό `...` και τον χειροκίνητο χειρισμό arg της Lua.

```lua
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 v5.44;
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;
    return $sum;
}
say total(1, 2, 3, 4);       # 10
```

Το ιδίωμα ονοματισμένων ορισμάτων της Lua (το πέρασμα ενός μοναδικού table, `f{host=..., port=...}`) αντιστοιχίζεται σε μια παράμετρο hash, αφού τα ζεύγη key => value ισοπεδώνονται σε μια λίστα που μια υπογραφή %opts συγκεντρώνει:

```
use v5.44;
sub connect_to (%opts) {
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    return "host=$opts{host} port=$opts{port}";
}
say connect_to(host => "localhost", port => 8080);
# host=localhost port=8080
```

Η παγίδα: μια sub επιστρέφει την τιμή της τελευταίας της έκφρασης ακόμη και χωρίς `return` (όπως η μόνο-ρητή επιστροφή της Lua, μόνο που η Perl έχει επίσης τη σιωπηρή μορφή της τελευταίας τιμής), και η επιστροφή είναι **ευαίσθητη στο περιβάλλον** (δείτε *Περιβάλλον*). Αν κάνετε `return @list`, ο καλών παίρνει τη λίστα σε περιβάλλον λίστας αλλά το *πλήθος* σε βαθμωτό περιβάλλον - ένας βαθμός ελευθερίας που η πολλαπλή επιστροφή της Lua δεν έχει.

Αναφορές

Οι πίνακες και τα hashes της Perl δεν εμφωλεύονται απευθείας· τα εμφωλεύετε μέσω **αναφορών**, ενός βαθμωτού που δείχνει σε ένα σύνολο. Στη Lua κάθε table είναι ήδη αναφορά (η ανάθεση ενός table το μοιράζεται, ποτέ δεν το αντιγράφει), οπότε αυτή η έννοια είναι εκεί αόρατη σε εσάς· στην Perl παίρνετε μια αναφορά ρητά με `\`, και την ακολουθείτε με `->`.

```
local data = {name = "Ada", langs = {"Perl", "Lua"}}
print(data.langs[2])        -- Lua
table.insert(data.langs, "Ruby")
```

```
use v5.44;
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
```

Το `{ ... }` φτιάχνει μια ανώνυμη αναφορά hash και το `[ ... ]` μια ανώνυμη αναφορά πίνακα - τα κυριολεκτικά δομικά στοιχεία των εμφωλευμένων δεδομένων. Το βέλος `->` ακολουθεί μια αναφορά κατά ένα βήμα· ανάμεσα σε δύο δείκτες είναι προαιρετικό, οπότε τα `$data->{langs}[1]` και `$data->{langs}->[1]` είναι ίδια.

Οι ανώνυμες subs είναι πραγματικά closures, ακριβώς όπως τα closures της Lua πάνω από upvalues - μία από τις πιο καθαρές μεταφορές σε όλον τον οδηγό:

```
use v5.44;
sub counter ($start) {
    my $n = $start;
    return sub { return $n++ };
}
my $c = counter(10);
say $c->();                 # 10
say $c->();                 # 11
say $c->();                 # 12
```

Η παγίδα: οι μεταθεματικές μορφές αποαναφοράς (`->@*`, `->%*`, `->$*`) είναι ο σύγχρονος, ευανάγνωστος τρόπος να αποαναφέρετε ένα ολόκληρο σύνολο, και το `"@{ ... }"` είναι

ο τρόπος να παρεμβάλετε έναν αποαναφερμένο πίνακα μέσα σε μια συμβολοσειρά. Οι παλαιότερες μορφές `@$ref` και `${$ref}{key}` εξακολουθούν να δουλεύουν και εμφανίζονται σε υπάρχοντα κώδικα, αλλά προτιμήστε το βέλος και τις μεταθεματικές μορφές σε νέο κώδικα.

Κανονικές εκφράσεις

Αυτό είναι αναβάθμιση, όχι μετάφραση. Η Lua έχει **μοτίβα** - `string.match`, `string.gsub`, κλάσεις χαρακτήρων όπως το `%d` - τα οποία εσκεμμένα *δεν* είναι κανονικές εκφράσεις και τους λείπει η εναλλαγή, οι ποσοδείκτες, και οι οπισθαναφορές. Η Perl έχει πλήρες regex ως **σύνταξη**: το `=~` συνδέει ένα μοτίβο με μια συμβολοσειρά, το `m//` αντιστοιχίζει, το `s///` αντικαθιστά. Οι ομάδες σύλληψης καταλήγουν στα `$1`, `$2`, και οι ονομαστικές συλλήψεις στο `%+`.

```
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 v5.44;
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
```

Μια αντιστοίχιση `/g` σε περιβάλλον λίστας επιστρέφει *όλες* τις αντιστοιχίες, το άμεσο αντίστοιχο της επανάληψης `string.gmatch`:

```
use v5.44;
my @nums = "x1y2z3" =~ /(\d)/g;     # 1 2 3
say "@nums";
```

Η παγίδα: οι μετα-χαρακτήρες διαφέρουν από τα μοτίβα της Lua - η Perl χρησιμοποιεί `\d` εκεί που η Lua χρησιμοποιεί %d, ποσοδείκτες */+/?, εναλλαγή `|`, και ονομαστικές ομάδες `(?<name>...)`, καμία από τις οποίες δεν υποστηρίζουν τα μοτίβα της Lua. Και το `s///` τροποποιεί τον στόχο του *επιτόπια* και επιστρέφει τον αριθμό των αντικαταστάσεων, όχι μια νέα συμβολοσειρά, εκεί που το `gsub` της Lua επιστρέφει μια νέα συμβολοσειρά. Γι" αυτό το παράδειγμα αντιγράφει πρώτα με `(my $t = "a,b,c")` και μετά αντικαθιστά στο αντίγραφο. Για να πάρετε μια νέα συμβολοσειρά χωρίς να αγγίξετε την αρχική, χρησιμοποιήστε τη σημαία `/r`: `my $t = $s =~ s/,/;/gr`.

I/O αρχείων και ροών

Το ιδίωμα είναι το open με τρία ορίσματα και ένα λεξιλογικό filehandle, το αντίστοιχο του `io.open` της Lua. Ο τελεστής με γωνιώδεις αγκύλες `<$fh>` διαβάζει μια γραμμή, τον ρόλο του `fh:read("l")` ή του επαναλήπτη `fh:lines()` της Lua.

```lua
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()
```

```perl
use v5.44;
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;
```

Για να ρουφήξετε (slurp) όλες τις γραμμές μονομιάς, διαβάστε το handle σε περιβάλλον λίστας:

```perl
use v5.44;
open my $fh, '<', "/tmp/sample.txt" or die $!;
my @lines = <$fh>;          # all lines, including newlines
say scalar @lines;          # 3
```

Η παγίδα: μια γραμμή που διαβάστηκε με `<$fh>` κρατάει τον τελικό χαρακτήρα νέας γραμμής (το `fh:lines()` της Lua τον αφαιρεί, το `fh:read("L")` τον κρατάει)· καλέστε `chomp` για να τον αφαιρέσετε. Και το open επιστρέφει ψευδές σε αποτυχία αντί να εγείρει εξαίρεση, όπως ακριβώς το `io.open` της Lua επιστρέφει `nil, err`, οπότε το ιδίωμα `or die "...: $!"` είναι υποχρεωτικό - το `$!` είναι το μήνυμα σφάλματος συστήματος, το `errno` της Perl.

Αρθρώματα και πακέτα

Το `require` της Lua γίνεται το `use` της Perl. Ο διαχωριστής χώρου ονομάτων είναι το `::`, όχι το `.`, και ένα άρθρωμα `Foo::Bar` βρίσκεται στο `Foo/Bar.pm` στη διαδρομή ένταξης `@INC` (το `package.path` της Lua). Σε αντίθεση με το `require` της Lua, που επιστρέφει ένα table που αναθέτετε εσείς, το `use` εισάγει επιλεγμένα ονόματα απευθείας στον χώρο ονομάτων σας.

```lua
local sqrt = math.sqrt
local sum = 0
for i = 1, 10 do sum = sum + i end
```

```perl
use v5.44;
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
```

Εδώ είναι μια ευχάριστη έκπληξη ερχόμενοι από τη σκόπιμα μικροσκοπική τυπική βιβλιοθήκη της Lua: η Perl έρχεται με ένα μεγάλο ενσωματωμένο σύνολο, και πολλά από αυτά που θα έφτιαχνες με το χέρι ή θα τραβούσες από το LuaRocks είναι στον πυρήνα ή ένα `use` μακριά στο CPAN, το αρχείο πακέτων της Perl. Τα `List::Util`, `Scalar::Util`, `POSIX`, `File::Spec`, και `Data::Dumper` είναι τα καθημερινά.

Η παγίδα: το `use` τρέχει κατά τη *μεταγλώττιση*, όχι στη γραμμή όπου εμφανίζεται, οπότε η τοποθέτησή του σε σχέση με τον κώδικά σας δεν έχει σημασία όπως έχει μια κλήση `require`. Η εισαγωγή είναι προαιρετική ανά όνομα: το `use List::Util qw(sum0 max)` φέρνει ακριβώς τα `sum0` και `max` στον χώρο ονομάτων σας, τίποτε άλλο.

Αντικείμενα

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 5.44 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).

```lua
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

local p = Point.new(3, 4)
print(p:to_string())        -- (3, 4)
p:move(1, 1)
print(p:to_string())        -- (4, 5)
```

```
use v5.44;
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)
```

Ο κατασκευαστής δημιουργείται για εσάς, αντικαθιστώντας το γραμμένο-με-το-χέρι `Point.new` της Lua: το `:param` σημειώνει ένα πεδίο που η αυτόματα κατασκευασμένη `new` δέχεται ως ονοματισμένο όρισμα, και το `= 0` είναι η προεπιλογή του. Η κληρονομικότητα, την οποία η Lua χτίζει από μια αλυσίδα metatables `__index`, είναι το attribute `:isa`:

```
use v5.44;
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 5.44, and on `pperl` 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*.

Χειρισμός σφαλμάτων

Η προστατευμένη κλήση `pcall(f)` της Lua - που επιστρέφει `false, err` αντί να ξετυλίγει - έχει δύο αντίστοιχα στην Perl. Το κλασικό τυλίγει κώδικα σε `eval { }` και εξετάζει το `$@` (το σφάλμα που πιάστηκε) μετά:

```lua
local ok, err = pcall(function() risky(-1) end)
if not ok then
    print("caught: " .. err)
end
```

```perl
use v5.44;
my $result = eval { risky(-1) };
if ($@) {
    print "caught: $@";     # caught: negative!
}
```

Η σύγχρονη μορφή είναι το `try`/`catch`, που διαβάζεται πιο πολύ σαν ένα συμβατικό μπλοκ εξαίρεσης παρά σαν τον χορό λογικής-επιστροφής του `pcall`:

```perl
use v5.44;
use feature 'try';

try {
    risky(-4);
}
catch ($e) {
    print "caught: $e";     # caught: negative!
}
```

όπου η πλευρά που εγείρει είναι το `die`, το `error` της Perl:

```perl
use v5.44;
sub risky ($n) {
    die "negative!\n" if $n < 0;
    return sqrt $n;
}
```

Η παγίδα: το `die` με μια συμβολοσειρά που τελειώνει σε `\n` παράγει ακριβώς αυτό το μήνυμα· το `die` με μια συμβολοσειρά που *δεν* τελειώνει σε χαρακτήρα νέας γραμμής παίρνει αυτόματα την προσθήκη `" at SCRIPT line N.\n"` (το `error` της Lua κάνει παρόμοιο προθεματισμό θέσης εκτός αν περάσετε επίπεδο `0`). Τελειώστε τις συμβολοσειρές σφάλματός σας με `\n` όταν θέλετε ένα καθαρό μήνυμα και παραλείψτε το όταν θέλετε το ίχνος αρχείου-και-γραμμής. Σημειώστε επίσης τις δύο μεταβλητές σφάλματος: το `$@` είναι η τελευταία εξαίρεση που πιάστηκε, ενώ το `$!` είναι το τελευταίο σφάλμα *συστήματος* (ένα αποτυχημένο open, η συμβολοσειρά `errno`).

Ιδιωματισμοί

Μερικά μοτίβα που θα βλέπετε καθημερινά και θα θέλετε στα δάχτυλά σας.

Ταξινομήστε μια λίστα με βάση ένα υπολογισμένο κλειδί (ο Schwartzian transform - διακόσμησε, ταξινόμησε, αφαίρεσε τη διακόσμηση σε μία σωλήνωση, το αντανακλαστικό της Perl εκεί που η Lua περνάει μια συνάρτηση συγκριτή στο `table.sort`):

```perl
use v5.44;
my @words = qw(apple fig pear);
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
my @by_length =
    map  { $_->[1] }
    sort { $a->[0] <=> $b->[0] }
    map  { [ length $_, $_ ] } @words;
say "@by_length";           # fig pear apple
```

Φτιάξτε ένα hash από μια λίστα, μετατρέποντας μια ακολουθία σε πίνακα αναζήτησης σε ένα βήμα:

```
use v5.44;
my @words = qw(apple pear fig);
my %len = map { $_ => length $_ } @words;
say "$_ => $len{$_}" for sort keys %len;
# apple => 5
# fig => 3
# pear => 4
```

Και το `pperl` είναι ένα εργαλείο γραμμής εντολών στην εξειδικευμένη θέση των `awk`/`sed`, έναν ρόλο που η Lua δεν στοχεύει. Το μονόγραμμο με σιωπηρό βρόχο είναι το καθημερινό αντανακλαστικό του προγραμματιστή της Perl:

```
pperl -lane '$s += $F[-1]; END { print $s }' data.txt
```

Η παγίδα: διαβάστε τον *οδηγό Μονόγραμμων* για τους διακόπτες (`-n`, `-p`, `-a`, `-l`, `-e`) και τις δεκάδες συνταγές που είναι χτισμένες πάνω τους· είναι ένα από τα πράγματα με τη μεγαλύτερη μόχλευση που μπορείτε να μάθετε ερχόμενοι από υπόβαθρο scripting όπου αυτή η εξειδικευμένη θέση ήταν δουλειά κάποιου άλλου.

Παγίδες ερχόμενοι από τη Lua

Οι παγίδες, συγκεντρωμένες για μια τελική σάρωση πριν αρχίσετε να γράφετε:

- **Ένα table γίνεται δύο τύποι.** Ένα table της Lua είναι ένας `@array` (με ακέραια κλειδιά, διατεταγμένος) *ή* ένα `%hash` (με κλειδιά συμβολοσειρών, αδιάτακτο) στην Perl, ποτέ και τα δύο μαζί. Αποφασίστε ποιο έχετε πριν το χρησιμοποιήσετε.
- **Οι πίνακες είναι 0-based.** Το `t[1]` της Lua είναι το `$t[0]` της Perl· το τελευταίο στοιχείο είναι `$t[-1]`. Αυτό είναι το πιο συχνό σφάλμα ολίσθησης-κατά-ένα κατά τη μεταφορά.
- **Τα sigils αλλάζουν με τη χρήση.** Ένα στοιχείο του `@a` είναι `$a[0]`· μία τιμή του `%h` είναι `$h{k}`. Το sigil του περιέκτη και το sigil του στοιχείου διαφέρουν - το σκέτο όνομα `t` της Lua δεν έχει τέτοια μετατόπιση.
- **Η συνένωση είναι `.`, όχι `...`** Το `..` της Lua δεν υπάρχει· το `.` ενώνει συμβολοσειρές και το `x` τις επαναλαμβάνει.
- **Η αλήθεια είναι αντεστραμμένη στο μηδέν.** Στη Lua τα `0` και `""` είναι *αληθή*· στην Perl είναι **ψευδή**. Μόνο τα `0`, `""`, `"0"`, και `undef` είναι ψευδή, οπότε τα `"00"` και `"0.0"` είναι παραδόξως **αληθή**.
- **Δύο οικογένειες σύγκρισης.** `==`/`<`/`<=>` για αριθμούς, `eq`/`lt`/`cmp` για συμβολοσειρές. Επιλέξτε με βάση τον τύπο του τελεστέου· το μοναδικό `==` της Lua δεν έχει

αντίστοιχο διαχωρισμό.

- **Το `sort` ταξινομεί ως συμβολοσειρές από προεπιλογή.** Περάστε `{ $a <=> $b }` για αριθμητική σειρά.
- **Περιβάλλον.** Το `my $x = @a` είναι το *μήκος*· το `my ($x) = @a` είναι το *πρώτο στοιχείο*. Η μορφή λίστας είναι ο φίλος σας `local a, b = f()`· η βαθμωτή μορφή είναι καινούργια. Ξαναδιαβάστε το *Περιβάλλον* όταν μια τιμή επιστρέφει με λάθος σχήμα.
- **Το regex δεν είναι τα μοτίβα της Lua.** Η Perl έχει πλήρες regex (`\d`, `|`, `*`, ονομαστικές ομάδες)· τα μοτίβα της Lua (%d, χωρίς εναλλαγή) είναι μια διαφορετική, μικρότερη γλώσσα. Το `s///` επεξεργάζεται επιτόπια και επιστρέφει ένα πλήθος· χρησιμοποιήστε `/r` για μια νέα συμβολοσειρά.
- **Το open δεν εγείρει εξαίρεση.** Χρησιμοποιήστε `open ... or die "...: $!"`, όπως το `io.open` της Lua επιστρέφει `nil, err`.
- **Κάντε chomp στις αναγνώσεις γραμμών σας.** Το `<$fh>` κρατάει τον τελικό χαρακτήρα νέας γραμμής που το `lines()` της Lua θα είχε αφαιρέσει.
- **Non-ASCII source needs `use utf8;`.** Under `use v5.44;` a literal non-ASCII byte in your source is a compile error without it. Keep string data ASCII unless you specifically add `use utf8;`.
- **Το χαρακτηριστικό `class` τυπώνει πειραματικές ειδοποιήσεις στο stderr** στο `pperl`. Σωστή έξοδος, θορυβώδης ροή σφαλμάτων. Οι μεταμέθοδοι τελεστών αντιστοιχίζονται στην pragma `overload`, όχι στο `class`.

Από Sed σε Perl

Ξέρετε το `sed`. Σκέφτεστε με όρους `s/old/new/`, στοχεύετε γραμμές με `/pattern/` και `$`, καταφεύγετε στο `-n` και στο `p` για επιλεκτική εκτύπωση, και στο `-i` για επιτόπια επεξεργασία αρχείων. Αυτός ο οδηγός αντιστοιχίζει καθεμία από αυτές τις συνήθειες στην Perl, ώστε να κρατήσετε κάθε αντανακλαστικό που έχετε και να αποκτήσετε τα πράγματα που το sed κάνει δύσκολα.

Η μία ιδέα που πρέπει να κρατήσετε από την πρώτη γραμμή: **η Perl είναι σχεδόν υπερσύνολο του sed για εργασίες κειμένου.** Ό,τι κάνει το sed, το κάνει και η Perl - συνήθως με την *ίδια* σύνταξη `s///` - συν μια πραγματική μηχανή regex, πραγματικές μεταβλητές και αυθαίρετο κώδικα στην αντικατάσταση. Οι δύσχρηστες γωνιές του sed (ο χώρος συγκράτησης, τα κόλπα πολλαπλών γραμμών, οτιδήποτε μοιάζει με αριθμητική) γίνονται συνηθισμένη Perl που θα γράφατε χωρίς σκέψη. Αυτός ο οδηγός ξεκινά με τις ένα-προς-ένα αντιστοιχίσεις και στη συνέχεια δείχνει πού η Perl απλώς αφαιρεί έναν περιορισμό του sed.

Every runnable example here was executed on `pperl`, a Rust reimplementation of Perl 5.44; the `# ...` comments show its real output. Start scripts with:

```
use v5.44;
```

Αυτή η μία γραμμή ενεργοποιεί το `strict`, τα `warnings` και την ενσωματωμένη `say` (μια `print` με τελική νέα γραμμή), το σύγχρονο σύνολο χαρακτηριστικών που χρησιμοποιείται παντού.

Σημείωση

One-liner switches on this pperl build sed's natural Perl twin is the one-liner: `perl -ne` and `perl -pe`. Those switches are **correct perl5 5.44** 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.

Πώς να το διαβάσετε

- *Ο κύκλος γραμμής* - ο έμμεσος βρόχος του sed γίνεται `while (<$fh>)` και `$_`.
- *Αντικατάσταση:* `s///` - ο ίδιος χειριστής, με πραγματικό regex.
- *Μεταγραμματισμός: το* `y///` *γίνεται* `tr///` - η εντολή y, μετονομασμένη.
- *Διευθύνσεις και εύρη* - αριθμοί γραμμών, μοτίβα και `/a/ .. /b/`.
- *Διαγραφή, εκτύπωση, έξοδος:* d, p, q - `next`, `print`, `last`.
- *Ο χώρος συγκράτησης γίνεται μεταβλητή* - εκεί που το sed είναι δύσκολο και η Perl τετριμμένη.
- *Επιτόπια επεξεργασία και I/O* - `-i`, ανάγνωση αρχείων, σωληνώσεις.
- *One-liners και διακόπτες* - `-n`, `-p`, `-e`, `BEGIN`/`END`.
- *Παγίδες ερχόμενοι από το sed* - ρίξτε μια ματιά πριν γράψετε.

Ο κύκλος γραμμής

Το sed τρέχει το σενάριό σας μία φορά ανά γραμμή εισόδου, με την τρέχουσα γραμμή στον χώρο μοτίβου του, και εκτυπώνει τον χώρο μοτίβου στο τέλος του κύκλου (εκτός αν δοθεί `-n`). Το αντίστοιχο της Perl είναι ένας ρητός βρόχος με τη γραμμή στο `$_`, την έμμεση μεταβλητή θέματος. Ο διακόπτης `-p` αναπαράγει το «αυτόματη εκτύπωση κάθε γραμμής»· το `-n` αναπαράγει το «μην εκτυπώσεις τίποτα εκτός αν το πω».

```
sed 's/bar/BAR/' input.txt
```

```
perl -pe 's/bar/BAR/' input.txt
```

Η εκτελέσιμη μορφή πλήρους σεναρίου του ίδιου βρόχου - αυτή που τρέχει στο `pperl` σήμερα - ανοίγει το αρχείο και το διατρέχει ρητά. Το `$_` είναι η τρέχουσα γραμμή, ακριβώς όπως ο χώρος μοτίβου του sed:

```
use v5.44;
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)
}
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
# Apple
# bAnAnA
# cherry
```

(Το αρχείο εισόδου περιείχε apple, banana, cherry.) Η παγίδα: μια γραμμή που διαβάζεται με <$fh> κρατά την τελική νέα γραμμή της, ακριβώς όπως ο χώρος μοτίβου του sed. Η print χωρίς όρισμα εκτυπώνει το $_ συμπεριλαμβανομένης αυτής της νέας γραμμής, οπότε μεταφέρεται καθαρά· κάντε chomp μόνο όταν σκοπεύετε να την αφαιρέσετε. Ο αριθμός γραμμής βρίσκεται στο $., το αντίστοιχο στο sed του μετρητή γραμμών στον οποίο καταφεύγετε με το $= σε σενάρια GNU sed.

Αντικατάσταση: s///

Αυτός είναι ο τίτλος: **το s/// του sed και το s/// της Perl είναι ο ίδιος χειριστής.** Και οι σημαίες ταιριάζουν - το g είναι καθολικό, το i αγνοεί πεζά/κεφαλαία - και η Perl προσθέτει σημαίες για τις οποίες το sed δεν έχει αντίστοιχο.

```
sed 's/colou\?r/COLOR/g' input.txt
```

```
perl -pe 's/colou?r/COLOR/g' input.txt
```

Εξ ορισμού το s/// επεξεργάζεται τον στόχο του επιτόπια και επιστρέφει το πλήθος των αντικαταστάσεων. Οι ομάδες σύλληψης είναι $1, $2, ... και μπορείτε να εκτελέσετε **αυθαίρετο κώδικα** στην αντικατάσταση με τη σημαία /e - κάτι που το sed δεν μπορεί να κάνει καθόλου:

```
use v5.44;
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
```

Για να αναδιατάξετε συλλήψεις (τις οπισθοαναφορές \1, \2 του sed στην αντικατάσταση), χρησιμοποιήστε $1, $2 και, όταν θέλετε μια *νέα* συμβολοσειρά αντί για επιτόπια επεξεργασία, τη σημαία /r, η οποία επιστρέφει το αποτέλεσμα και αφήνει το πρωτότυπο ανέγγιχτο:

```
use v5.44;
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
```

Η παγίδα: το γυμνό s/// τροποποιεί τον στόχο του και επιστρέφει ένα *πλήθος*, όχι τη νέα συμβολοσειρά. Γι" αυτό τα παραπάνω παραδείγματα είτε αντιγράφουν πρώτα ((my $t = $s) =~ s/.../.../) είτε χρησιμοποιούν το /r. Γράφοντας my $new = ($s =~ s/a/b/) βάζετε το πλήθος των αντικαταστάσεων στο $new, όχι την επεξεργασμένη συμβολοσειρά - μια κλασική έκπληξη της πρώτης μέρας. Σημειώστε επίσης ότι οι οριοθέτες της Perl είναι ελεύθεροι: τα s{...}{...} και s#...#...# αποφεύγουν τις ανάποδες καθέτους στις καθέτους ενός μοτίβου διαδρομής.

Μεταγραμματισμός: το y/// γίνεται tr///

Το `y/abc/xyz/` του sed είναι το `tr/abc/xyz/` της Perl. Η Perl κρατά το `y///` ως ακριβές ψευδώνυμο, οπότε το `y/a-z/A-Z/` λειτουργεί αυτολεξεί, αλλά το `tr` είναι το όνομα που θα δείτε γραμμένο στον κώδικα Perl.

```
sed 'y/abc/xyz/' input.txt
```

```
use v5.44;
my $s = "hello";
(my $t = $s) =~ tr/a-z/A-Z/;   # uppercase, like y/a-z/A-Z/
say $t;                        # HELLO
```

Το `tr` κάνει επίσης πράγματα που το `y` δεν μπορεί. Με κενή αντικατάσταση *μετρά* αντιστοιχίες, και ο τροποποιητής `s` συμπτύσσει διαδοχές ενός χαρακτήρα - το συνηθισμένο ιδίωμα `tr -s`:

```
use v5.44;
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
```

Η παγίδα: το `tr///` **δεν** είναι regex - η αριστερή πλευρά είναι ένα κυριολεκτικό σύνολο χαρακτήρων, το ίδιο με το `y` του sed. Το `tr/a-z//` σημαίνει το εύρος `a` έως `z`, αλλά το `tr/.*//` σημαίνει τους δύο κυριολεκτικούς χαρακτήρες `.` και `*`, όχι «οποιαδήποτε συμβολοσειρά». Για εργασία με μοτίβα, καταφύγετε στο `s///`. Και όπως το `s///`, το `tr///` επεξεργάζεται επιτόπια και επιστρέφει ένα πλήθος, οπότε αντιγράψτε πρώτα ή χρησιμοποιήστε τη σημαία `/r` για έναν μη καταστροφικό μεταγραμματισμό.

Διευθύνσεις και εύρη

Μια εντολή sed μπορεί να έχει πρόθεμα μια διεύθυνση: έναν αριθμό γραμμής, `$` για την τελευταία γραμμή, ένα `/pattern/`, ή ένα εύρος `addr1,addr2`. Στην Perl μια διεύθυνση είναι απλώς μια συνθήκη στην εντολή - ένα μεταθεματικό `if`. Ο αριθμός γραμμής είναι `$.`· ο έλεγχος μοτίβου είναι `/pat/` έναντι του `$_`· η τελευταία γραμμή είναι `eof`.

```
sed -n '2p' input.txt              # print line 2
sed -n '/banana/p' input.txt       # print matching lines
```

```
use v5.44;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (<$fh>) {
    print if $. == 2;              # sed -n '2p'
}
# banana
```

Ένα εύρος sed `/start/,/end/` γίνεται ο χειριστής εύρους **flip-flop** της Perl `/start/ .. /end/`, ο οποίος είναι αληθής από τη γραμμή που ταιριάζει με το αριστερό μοτίβο έως

τη γραμμή που ταιριάζει με το δεξί, συμπεριλαμβανομένων:

```
use v5.44;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (<$fh>) {
    print if /banana/ .. /cherry/;  # sed '/banana/,/cherry/p' with␣
 ↪-n
}
# banana
# cherry
```

Η παγίδα: σε έναν λογικό έλεγχο το `..` είναι ο χειριστής flip-flop (εύρους), **όχι** ο χειριστής εύρους λίστας (που είναι το `1 .. 10` που χτίζει μια λίστα). Η Perl τα ξεχωρίζει βάσει περιβάλλοντος - στη βαθμωτή/λογική θέση ενός `if`, το `..` είναι ο επιλογέας εύρους γραμμών που αντικατοπτρίζει το `addr1`, `addr2` του sed. Είναι με κατάσταση: θυμάται αν έχει δει το μοτίβο εκκίνησης, οπότε κάνει ακριβώς ό,τι κάνει ένα εύρος sed κατά μήκος του βρόχου.

Διαγραφή, εκτύπωση, έξοδος: d, p, q

Οι μονογράμματες εντολές κύκλου του sed αντιστοιχίζονται σε δεσμευμένες λέξεις βρόχου της Perl. Το d (διαγραφή χώρου μοτίβου, μετάβαση στον επόμενο κύκλο) είναι `next`. Το q (έξοδος) είναι `last`. Το p (εκτύπωση) είναι `print`. Επειδή ο βρόχος της Perl είναι ρητός, διαβάζονται ακριβώς ως αυτό που κάνουν.

```
sed '/banana/d' input.txt          # drop matching lines
sed '/banana/q' input.txt          # stop after the match
```

```
use v5.44;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (<$fh>) {
    next if /banana/;              # sed '/banana/d'
    print;
}
# apple
# cherry
```

```
use v5.44;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (<$fh>) {
    print;
    last if /banana/;              # sed '/banana/q'
}
# apple
# banana
```

Η παγίδα: υπό την αυτόματη εκτύπωση τύπου `-p` του sed, το d πρέπει να παρακάμψει την αυτόματη εκτύπωση, κάτι που το κάνει ματαιώνοντας τον κύκλο. Σε έναν ρητό βρόχο Perl ελέγχετε εσείς την εκτύπωση, οπότε η «διαγραφή» είναι απλώς το *να μην εκτυπώσετε* τη γραμμή - `next if /pat/` πριν την `print`, ή φυλάξτε την `print` με `unless`. Δεν υπάρχει κρυφή αυτόματη εκτύπωση να αντιπαλέψετε· ό,τι εκτυπώνετε με `print` είναι

ό,τι βγαίνει.

Ο χώρος συγκράτησης γίνεται μεταβλητή

Αυτή είναι η ενότητα όπου το sed παύει να είναι βολικό και η Perl παύει να είναι δύσκολη. Το sed έχει ακριβώς έναν βοηθητικό ενταμιευτή - τον χώρο συγκράτησης - και μετακινεί δεδομένα σε αυτόν με h, H, g, G, x. Οτιδήποτε χρειάζεται δύο κομμάτια αποθηκευμένης κατάστασης μετατρέπεται σε ακροβατικά χώρου συγκράτησης. Στην Perl απλώς δηλώνετε μια μεταβλητή. Ή έναν πίνακα. Ή έναν κατακερματισμό.

Η κατεξοχήν άσκηση χώρου συγκράτησης είναι η αντιστροφή ενός αρχείου (tac), που στο sed είναι ένα αδιαφανές ξόρκι `1!G;h;$!d`. Στην Perl είναι μία προφανής γραμμή:

```
sed -n '1!G;h;$!d' input.txt       # reverse the file (tac)
```

```
use v5.44;
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
}
print @hold;
# cherry
# banana
# apple
```

Η παγίδα: δεν υπάρχει παγίδα - αυτό ακριβώς είναι το νόημα. Μόλις έχετε πραγματικές μεταβλητές, εξαφανίζεται ολόκληρη η κατηγορία προβλημάτων του sed που υπάρχουν μόνο επειδή υπάρχει ένας μοναδικός χώρος συγκράτησης. Χρειάζεται να θυμάστε την προηγούμενη γραμμή; `my $prev`. Χρειάζεστε ένα τρέχον άθροισμα; `my $sum`. Χρειάζεται να αφαιρέσετε διπλότυπα; Ένας κατακερματισμός `my %seen`. Όταν ένα σενάριο sed καταφεύγει στα h/H/x, η μετάφραση σε Perl είναι σχεδόν πάντα «εισάγετε μια επώνυμη μεταβλητή και γράψτε το προφανές».

Επιτόπια επεξεργασία και I/O

Το `-i` του sed ξαναγράφει ένα αρχείο επιτόπια. Η ιδιωματική Perl είναι ο ίδιος διακόπτης σε ένα one-liner `-p`. Η ανάγνωση πολλών αρχείων είναι η προεπιλογή του sed· η Perl το γράφει <>, ο χειριστής διαμαντιού, ο οποίος διαβάζει με τη σειρά κάθε αρχείο που ονομάζεται στο @ARGV.

```
sed -i 's/red/crimson/' file.txt   # edit in place
```

```
perl -i -pe 's/red/crimson/' file.txt
```

Το εκτελέσιμο αντίστοιχο σήμερα είναι μια ρητή ανάγνωση-τροποποίηση-εγγραφή: ρουφήξτε τις γραμμές, μετασχηματίστε τις, γράψτε τις πίσω. Αυτό ακριβώς κάνει το `-i` στο παρασκήνιο, και τρέχει στο `pperl` τώρα:

```
use v5.44;
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
# grEEn
# bluE
```

Για σωληνώσεις - sed στη μέση μιας σωλήνωσης κελύφους - η Perl ανοίγει έναν περιγραφέα σωλήνωσης απευθείας: το `open my $fh, '-|', 'ls', '-l'` διαβάζει την έξοδο μιας εντολής, και το `open my $fh, '|-', 'sort'` γράφει στην είσοδο μιας εντολής. Η παγίδα: η open επιστρέφει ψευδές σε αποτυχία αντί να εγείρει σφάλμα, οπότε το ιδίωμα `or die "...: $!"` είναι υποχρεωτικό - το `$!` είναι η συμβολοσειρά σφάλματος του συστήματος, η σιωπηλή αποτυχία του sed γινόμενη ρητή. Και η ανάγνωση πολλών αρχείων με <> μηδενίζει το `$.` ανά αρχείο μόνο αν ελέγξετε ρητά το eof· το σκέτο `$.` συνεχίζει να μετρά σε όλο το συνενωμένο ρεύμα.

One-liners και διακόπτες

Το one-liner είναι το οικείο έδαφος του sed, και της Perl επίσης. Η οικογένεια διακοπτών που το κάνει να λειτουργεί: το `-e` παρέχει το πρόγραμμα, το `-n` το τυλίγει σε έναν βρόχο χωρίς αυτόματη εκτύπωση (το `-n` του sed), το `-p` το τυλίγει σε έναν βρόχο με αυτόματη εκτύπωση (η προεπιλογή του sed), το `-i` επεξεργάζεται επιτόπια, το `-l` χειρίζεται τα τέλη γραμμών, και τα `-a`/`-F` αυτοδιαχωρίζουν κάθε γραμμή σε `@F` (έδαφος του awk, χρήσιμο εδώ για επεξεργασίες πεδίων). Τα μπλοκ `BEGIN { }` και `END { }` τρέχουν πριν και μετά τον βρόχο, το άμεσο αντίστοιχο των κόλπων αρχικοποίησης-και-τερματισμού `1i`/`$a` του sed.

```
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
```

Αυτά εμφανίζονται χωρίς έξοδο επειδή οι διακόπτες έμμεσου βρόχου δεν αναγνωρίζονται ακόμη από αυτό το build του `pperl` (δείτε τη σημείωση στην αρχή). Καθένας αντιστοιχίζεται σε μια επαληθευμένη μορφή ρητού βρόχου: το `perl -ne 'CODE'` είναι `while (<$fh>) { CODE }`, και το `perl -pe 'CODE'` είναι το ίδιο με μια τελική `print`. Η εκτελέσιμη μορφή «μέτρησε γραμμές σφάλματος»:

```
use v5.44;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
my $n = 0;
while (<$fh>) {
    $n++ if /banana/;          # the body of -ne
}
say $n;                        # 1
```

Η παγίδα: μέχρι να προστεθούν οι διακόπτες στο `pperl`, γράψτε τον ρητό βρόχο (τρέχει σήμερα και είναι ούτως ή άλλως αυτό στο οποίο αναπτύσσονται οι διακόπτες), και συμβουλευτείτε τον *οδηγό One-Liners* για τον πλήρη κατάλογο διακοπτών και δεκάδες συνταγές - είναι το πιο αποδοτικό πράγμα που μπορείτε να διαβάσετε ερχόμενοι από υπόβαθρο sed. Η Perl έχει επίσης πλήρεις υπορουτίνες, αναφορές, αρθρώματα και OO αν το σενάριό σας ξεπεράσει ένα one-liner· δείτε τον *οδηγό Αντικειμενοστραφούς Προγραμματισμού* για αυτήν την πλευρά της γλώσσας.

Παγίδες ερχόμενοι από το sed

Οι παγίδες, συγκεντρωμένες για μια τελική ματιά πριν γράψετε:

- **Το `s///` επιστρέφει ένα πλήθος, όχι τη νέα συμβολοσειρά.** Το `my $x = $s =~ s/a/b/` βάζει το *πλήθος* αντικαταστάσεων στο `$x`. Αντιγράψτε πρώτα (`my $t = $s) =~ s/.../.../`) ή χρησιμοποιήστε τη σημαία `/r` για να πάρετε μια νέα συμβολοσειρά.
- **Το `tr///` (το y του sed) είναι ένα κυριολεκτικό σύνολο χαρακτήρων, όχι regex.** Το `tr/.*//` ταιριάζει τους δύο χαρακτήρες `.` και `*`. Για μοτίβα, χρησιμοποιήστε `s//`.
- **Το `..` σε λογικό περιβάλλον είναι το εύρος flip-flop, όχι εύρος λίστας.** Σε ένα `if`, το `/a/ .. /b/` είναι ο επιλογέας `addr1`, `addr2` του sed· το `1 .. 10` σε περιβάλλον λίστας χτίζει μια λίστα. Ίδιος χειριστής, που αποφασίζεται βάσει περιβάλλοντος.
- **Το `<$fh>` κρατά την τελική νέα γραμμή,** όπως ο χώρος μοτίβου του sed. Η `print` τη μεταφέρει· κάντε chomp μόνο όταν εννοείτε να την αφαιρέσετε.
- **Δεν υπάρχει κρυφή αυτόματη εκτύπωση.** Η προεπιλογή `-p` του sed είναι κάτι που γράφετε εσείς οι ίδιοι με `print`. Το «διέγραψε μια γραμμή» είναι απλώς το να μην την εκτυπώσετε (`next if /pat/`), όχι μια ειδική εντολή.
- **Η `open` δεν εγείρει σφάλμα σε αποτυχία.** Χρησιμοποιήστε `open my $fh, '<', $path or die "...: $!"`. Το `$!` είναι η συμβολοσειρά σφάλματος του συστήματος.
- **Ο χώρος συγκράτησης χάθηκε - χρησιμοποιήστε μια μεταβλητή.** Κάθε σενάριο sed που καταφεύγει στα h/H/x μεταφράζεται σε μια επώνυμη μεταβλητή, πίνακα ή κατακερματισμό της Perl. Αυτό είναι απλοποίηση, όχι απώλεια.
- **Οι διακόπτες έμμεσου βρόχου (`-n`/`-p`/`-a`/`-F`/`-i`/`-l`) δεν αναγνωρίζονται ακόμη από αυτό το build του `pperl`.** Μόνο το `-e` λειτουργεί σήμερα. Γράψτε τον ρητό βρόχο `while (<$fh>) { ... }`, που είναι ακριβώς αυτό στο οποίο αναπτύσσονται οι διακόπτες, μέχρι να προστεθούν.

Από Awk σε Perl

Ξέρετε το awk. Σκέφτεστε με όρους `pattern { action }`, διαχωρίζετε κάθε γραμμή σε `$1`, `$2`, `$3` χωρίς να το ζητάτε, μετράτε πράγματα με `a[key]++`, και πλαισιώνετε όλη την εργασία με `BEGIN { }` και `END { }`. Η Perl γράφτηκε εν μέρει ως «awk, αλλά μεγαλύτερο» - σχεδόν ό,τι κάνετε στο awk έχει μια άμεση μορφή στην Perl, και ύστερα η Perl συνεχίζει: πραγματικές δομές δεδομένων, πλήρες regex, αρθρώματα, και μια γλώσσα που δεν μένει από χώρο όταν το σενάριο μεγαλώνει.

Η μία ιδέα που πρέπει να κρατήσετε από την πρώτη γραμμή: **η Perl είναι σχεδόν υπερσύνολο του awk.** Ο βρόχος εγγραφών, τα πεδία, οι συσχετιστικοί πίνακες, οι δεσμευμένες λέξεις BEGIN/END, οι `print` και `printf`, οι `length`/`substr`/`split`

- αντιστοιχίζονται όλα, συχνά με το *ίδιο όνομα*. Αυτό που αλλάζει είναι ότι τα πεδία γίνονται ένας πραγματικός πίνακας `@F` που μπορείτε να κάνετε `push` και `pop`, ο συσχετιστικός πίνακας γίνεται ένας κατακερματισμός της Perl που μπορείτε να εμφωλεύσετε, και η μηχανή regex είναι PCRE αντί για POSIX ERE. Αυτός ο οδηγός ξεκινά με τις ένα-προς-ένα αντιστοιχίσεις και επισημαίνει τα λίγα σημεία όπου η αναλογία διαρρέει.

Every runnable example here was executed on `pperl`, a Rust reimplementation of Perl 5.44; the `# ...` comments show its real output. Start scripts with:

```
use v5.44;
```

Αυτή η μία γραμμή ενεργοποιεί το `strict`, τα `warnings` και την ενσωματωμένη `say` (μια `print` με τελική νέα γραμμή), το σύγχρονο σύνολο χαρακτηριστικών που χρησιμοποιείται παντού.

Σημείωση

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 5.44** 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.

Πώς να το διαβάσετε

- *Ο βρόχος γραμμής/εγγραφής* - το έμμεσο `{ ... }` του awk γίνεται `while (<$fh>)` και `$_`.
- *Πεδία και διαχωρισμός* - τα `$1`/`$NF`/`NF`/`FS` γίνονται `@F`, `$F[0]`, `split`, `-F`.
- *Regex και αντικατάσταση* - τα `gsub`/`sub`/ `match` γίνονται `s///`/`=~`, ένα γνήσιο υπερσύνολο.
- *Διευθύνσεις και εύρη* - τα `/re/`, `cond { }` και `/a/,/b/` γίνονται μεταθεματικό `if` και το flip-flop.
- *I/O, σωληνώσεις, επιτόπια επεξεργασία* - `getline`, ανακατεύθυνση και `-i`.
- *One-liners και διακόπτες* - `-n`, `-a`, `-F`, BEGIN/END.

- *Ροή ελέγχου* - `next`, `for`, οι μεταθεματικές μορφές.
- *Μεταβλητές και δεδομένα* - οι συσχετιστικοί πίνακες γίνονται κατακερματισμοί, η ευτυχής αντιστοίχιση.
- *Παγίδες ερχόμενοι από το awk* - ρίξτε μια ματιά πριν γράψετε.

Ο βρόχος γραμμής/εγγραφής

Το awk τρέχει το πρόγραμμά σας μία φορά ανά εγγραφή εισόδου, με την εγγραφή στο `$0` και τη δράση στο `{ ... }`. Το αντίστοιχο της Perl είναι ένας ρητός βρόχος με την εγγραφή στο `$_`, την έμμεση μεταβλητή θέματος. Ο διακόπτης `-n` αναπαράγει το «τρέξε αυτό το μπλοκ ανά γραμμή» του awk· το `NR` (αριθμός εγγραφής) του awk είναι το `$.` της Perl.

```
awk '{ print }' input.txt
```

```
perl -ne 'print' input.txt
```

Η εκτελέσιμη μορφή πλήρους σεναρίου του ίδιου βρόχου - αυτή που τρέχει στο `pperl` σήμερα - ανοίγει το αρχείο και το διατρέχει ρητά. Το `$_` είναι η τρέχουσα εγγραφή, ακριβώς όπως το `$0` του awk:

```
use v5.44;
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
# line 3 : carol 95 100
# line 4 : alice 50 55
```

(Το αρχείο εισόδου περιείχε `alice 85 90`, `bob 70 60`, `carol 95 100`, `alice 50 55`.) Η παγίδα: μια γραμμή που διαβάζεται με `<$fh>` κρατά την τελική νέα γραμμή της, σε αντίθεση με το `$0` του awk, από το οποίο έχει αφαιρεθεί ο διαχωριστής εγγραφών. Καλέστε `chomp` για να την αφαιρέσετε (το chomp είναι το πιο κοντινό πράγμα στο «το awk ήδη αφαίρεσε τη νέα γραμμή για σένα»). Ο αριθμός εγγραφής είναι `$.` = το `NR` του awk· όταν διαβάζετε πολλά αρχεία με `<>`, το `$.` συνεχίζει να μετρά σε όλα τους, οπότε ο μηδενισμός του `FNR` (μετρητής ανά αρχείο) του awk είναι κάτι που οργανώνετε εσείς ελέγχοντας το `eof`.

Πεδία και διαχωρισμός

Αυτή είναι η καρδιά του awk, και η Perl σάς δίνει την ίδια εικόνα με μία ανατροπή: τα πεδία ζουν σε έναν πραγματικό πίνακα `@F`, οπότε το `$1` του awk είναι το `$F[0]` της Perl (μηδενοβάσει), το `$0` είναι το `$_`, και το `NF` είναι το πλήθος `scalar @F`. Υπό τον διακόπτη `-a` η Perl αυτοδιαχωρίζει κάθε γραμμή σε `@F` ακριβώς όπως το awk αυτοδιαχωρίζει σε `$1..$NF`· το εκτελέσιμο αντίστοιχο είναι ένα ρητό `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
```

Η μορφή πλήρους σεναρίου κάνει τον διαχωρισμό η ίδια. Το γυμνό `split` χωρίς ορίσματα διαχωρίζει το `$_` βάσει κενών χαρακτήρων και απορρίπτει τα αρχικά κενά - ακριβώς το προεπιλεγμένο `FS` του awk:

```
use v5.44;
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
```

Η αριθμητική κατά μήκος των πεδίων είναι η καθημερινή δράση `print $1, $2+$3` του awk, και διαβάζεται σχεδόν πανομοιότυπα:

```
use v5.44;
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
# bob: 130
# carol: 195
# alice: 105
```

Το `$NF` (το τελευταίο πεδίο) του awk είναι ο αρνητικός δείκτης `$F[-1]` της Perl, και ένα μη προεπιλεγμένο `FS` είναι το όρισμα μοτίβου του `split`:

```
use v5.44;
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
```

Για να ενώσετε ξανά τα πεδία με έναν διαχωριστή (το OFS του awk), ορίστε το $, (τον διαχωριστή πεδίων εξόδου που η print/say εισάγει ανάμεσα στα ορίσματα της λίστας της):

```
use v5.44;
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
```

Η παγίδα: τα πεδία είναι **μηδενοβάσει**. Το $1 του awk είναι $F[0], το $2 είναι $F[1], και το $0 (όλη η εγγραφή) είναι $_, όχι στοιχείο του @F. Το NF είναι scalar @F (ή $#F + 1)· το τελευταίο πεδίο είναι $F[-1], όχι $F[NF]. Και το split με **ένα κενό ως συμβολοσειρά** (split ' ') είναι η ειδική λειτουργία κενών-και-περικοπής· το split / / (ένα regex με ένα κενό) είναι κυριολεκτικό και *δεν* περικόπτει τα αρχικά κενά. Το σκέτο split χωρίς όρισμα είναι η προεπιλεγμένη λειτουργία τύπου awk που θέλετε τις περισσότερες φορές.

Regex και αντικατάσταση

Τα sub, gsub, match και ~ του awk αντιστοιχίζονται όλα στη μία σύνταξη regex της Perl: το =~ δένει ένα μοτίβο σε μια συμβολοσειρά, το s/// αντικαθιστά, το s///g είναι καθολικό (το gsub έναντι του sub του awk). Η μηχανή της Perl είναι PCRE, ένα γνήσιο υπερσύνολο του POSIX ERE του awk, οπότε κάθε μοτίβο awk λειτουργεί και η Perl προσθέτει οπισθοαναφορές, lookaround, επώνυμες συλλήψεις και τις σημαίες /e, /r, /x για τις οποίες το awk δεν έχει αντίστοιχο.

```
awk '/^alice/ { sub(/alice/, "ALICE"); print }' input.txt
```

```
perl -ne 'if (/^alice/) { s/alice/ALICE/; print }' input.txt
```

Η μορφή πλήρους σεναρίου:

```
use v5.44;
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
```

Χρησιμοποιήστε s///g για το gsub (κάθε αντιστοιχία) του awk, και οι ομάδες σύλληψης καταλήγουν στα $1, $2 ακριβώς όπως το match του awk γεμίζει τα RSTART/RLENGTH συν

τον πίνακα του gawk. Η σημαία `/e` εκτελεί **αυθαίρετη Perl** στην αντικατάσταση, κάτι που το awk δεν μπορεί να κάνει:

```
use v5.44;
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
```

Οι `length`, `substr`, `index` και οι συναρτήσεις πεζών/κεφαλαίων αντιστοιχίζονται κατ" όνομα - προσέξτε την ευρετηρίαση:

```
use v5.44;
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))
```

Η παγίδα: η `substr` είναι **μηδενοβάσει** ενώ η `substr` του awk είναι βάσει του ένα, οπότε το `substr($s,1,3)` του awk είναι `substr($s,0,3)` στην Perl. Και το `s///` επεξεργάζεται τον στόχο του επιτόπια και επιστρέφει το **πλήθος** των αντικαταστάσεων, όχι τη νέα συμβολοσειρά - οπότε το `my $new = ($s =~ s/a/b/)` βάζει έναν αριθμό στο `$new`. Αντιγράψτε πρώτα `((my $t = $s) =~ s/.../.../)` ή χρησιμοποιήστε τη σημαία `/r` για να πάρετε μια φρέσκια συμβολοσειρά. Οι κλάσεις χαρακτήρων POSIX (`[[:digit:]]`) λειτουργούν, αλλά οι συντομογραφίες της Perl (`\d`, `\w`, `\s`) είναι συντομότερες και αυτό που θα δείτε στον κώδικα Perl.

Διευθύνσεις και εύρη

Ένας κανόνας awk είναι `pattern { action }`: η δράση τρέχει μόνο σε εγγραφές που ταιριάζουν με το μοτίβο. Στην Perl αυτό το μοτίβο είναι απλώς μια συνθήκη στην εντολή - ένα μεταθεματικό `if` που ελέγχει το `$_`. Ένα γυμνό `/re/` ταιριάζει έναντι του `$_`, ένας αριθμητικός έλεγχος χρησιμοποιεί το `$.` (το `NR` του awk), και το εύρος `/start/,/end/` του awk είναι ο χειριστής **flip-flop** `/start/ .. /end/` της Perl.

```
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
```

Το flip-flop πλήρους σεναρίου, αληθές από τη γραμμή που ταιριάζει με το αριστερό μοτίβο έως τη γραμμή που ταιριάζει με το δεξί, συμπεριλαμβανομένων - ακριβώς το εύρος δύο μοτίβων του awk:

```
use v5.44;
open my $fh, '<', '/tmp/scores.txt' or die "open: $!";
while (<$fh>) {
    print if /bob/ .. /carol/;   # awk: /bob/,/carol/
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
}
# bob 70 60
# carol 95 100
```

Η παγίδα: ένα γυμνό `pattern` χωρίς δράση στο awk σημαίνει «εκτύπωσε την εγγραφή»· στην Perl ο βρόχος είναι ρητός, οπότε γράφετε εσείς την `print` (ή `say`). Και σε έναν λογικό έλεγχο το `..` είναι ο χειριστής flip-flop (εύρους), **όχι** ο χειριστής εύρους λίστας (το `1 .. 10` χτίζει μια λίστα). Η Perl τα ξεχωρίζει βάσει περιβάλλοντος: στη βαθμωτή θέση ενός `if`, το `..` είναι ο επιλογέας εύρους γραμμών με κατάσταση που αντικατοπτρίζει το `addr1,addr2` του awk.

I/O, σωληνώσεις, επιτόπια επεξεργασία

Το awk διαβάζει αυτόματα κάθε αρχείο που ονομάζεται στη γραμμή εντολών του και έχει το `getline` για ρητές αναγνώσεις. Το `<>` (ο χειριστής διαμαντιού) της Perl διαβάζει με τη σειρά κάθε αρχείο στο `@ARGV` - η ίδια προεπιλογή - και το `<$fh>` είναι το ρητό `getline`. Η εγγραφή σε μια εντολή ή η ανάγνωση από μία (τα `print | "cmd"` και `"cmd" | getline` του awk) είναι ένας περιγραφέας σωλήνωσης.

```
awk '{ print > "out.txt" }' input.txt      # redirect output
awk '{ print | "sort" }'     input.txt     # pipe to a command
```

```
use v5.44;
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;
```

Για σωληνώσεις, το `open my $fh, '|-', 'sort'` γράφει στην είσοδο μιας εντολής και το `open my $fh, '-|', 'ls', '-l'` διαβάζει την έξοδο μιας εντολής - το `| "cmd"` του awk και προς τις δύο κατευθύνσεις.

Το awk δεν έχει ενσωματωμένη επιτόπια επεξεργασία· καταφεύγετε σε ανακάτεμα tmp-αρχείου ή στο GNU `gawk -i inplace`. Η Perl το γράφει `-i` σε ένα one-liner:

```
perl -i -pe 's/red/crimson/' file.txt      # edit in place
```

Το εκτελέσιμο αντίστοιχο σήμερα είναι μια ρητή ανάγνωση-τροποποίηση-εγγραφή - ρουφήξτε τις γραμμές, μετασχηματίστε, γράψτε πίσω - που είναι ακριβώς αυτό που κάνει το `-i` στο παρασκήνιο:

```
use v5.44;
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;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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
```

Η παγίδα: η open επιστρέφει ψευδές σε αποτυχία αντί να ματαιώσει, οπότε το ιδίωμα `or die "...: $!"` είναι υποχρεωτικό - το `$!` είναι η συμβολοσειρά σφάλματος του συστήματος, η σιωπηλή αποτυχία του awk γινόμενη ρητή. Και το `print {$out} ...` χρειάζεται τα άγκιστρα γύρω από τον περιγραφέα όταν είναι μεταβλητή· η γυμνή μορφή `print FH ...` (χωρίς κόμμα) είναι για επώνυμους περιγραφείς, κάτι που αποτελεί συχνό σκόνταμμα της πρώτης μέρας.

One-liners και διακόπτες

Το one-liner είναι το οικείο έδαφος του awk, και της Perl επίσης. Η οικογένεια διακοπτών που αναπαράγει το awk: το `-e` παρέχει το πρόγραμμα, το `-n` το τυλίγει σε έναν βρόχο ανά εγγραφή (το έμμεσο `{ }` του awk), το `-p` κάνει το ίδιο με μια αυτόματη εκτύπωση στο τέλος κάθε κύκλου, το `-a` αυτοδιαχωρίζει σε `@F` (ο αυτόματος διαχωρισμός πεδίων του awk), το `-F` ορίζει το μοτίβο διαχωρισμού (το `FS` του awk), και το `-l` χειρίζεται τα τέλη γραμμών (η ευκολία `ORS`/`RS` του awk). Τα `BEGIN { }` και `END { }` είναι οι **ίδιες δεσμευμένες λέξεις** με το awk, τρέχοντας πριν και μετά τον βρόχο.

```
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
```

Αυτά εμφανίζονται χωρίς έξοδο επειδή οι διακόπτες έμμεσου βρόχου δεν αναγνωρίζονται ακόμη από αυτό το build του `pperl` (δείτε τη σημείωση στην αρχή). Καθένας αντιστοιχίζεται σε μια επαληθευμένη μορφή ρητού βρόχου: το `perl -ne 'CODE'` είναι `while (<$fh>) { CODE }`, το `-ane` προσθέτει `my @F = split` στην αρχή, και τα μπλοκ `BEGIN`/`END` λειτουργούν σε ένα πλήρες σενάριο ακριβώς όπως στο awk. Η εκτελέσιμη μορφή «άθροισε μια στήλη με BEGIN/END»:

```
use v5.44;
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>) {
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    my @F = split;
    $n++;
    $sum += $F[1];                  # awk: sum += $2
}
# start
# lines=4 sum=300
```

Η παγίδα: μέχρι να προστεθούν οι διακόπτες στο `pperl`, γράψτε τον ρητό βρόχο (τρέχει σήμερα και είναι ούτως ή άλλως αυτό στο οποίο αναπτύσσονται οι διακόπτες), και συμβουλευτείτε τον *οδηγό One-Liners* για τον πλήρη κατάλογο διακοπτών και δεκάδες συνταγές - είναι το πιο αποδοτικό πράγμα που μπορείτε να διαβάσετε ερχόμενοι από υπόβαθρο awk. Η Perl έχει επίσης πλήρεις υπορουτίνες, αναφορές, αρθρώματα και OO αν το σενάριό σας ξεπεράσει ένα one-liner· δείτε τον *οδηγό Αντικειμενοστραφούς Προγραμματισμού* και τους άλλους οδηγούς coming-from.

Ροή ελέγχου

Οι δεσμευμένες λέξεις ελέγχου του awk αντιστοιχούν στις αντίστοιχες της Perl σχεδόν λέξη προς λέξη. Το `next` του awk (παράλειψη προς την επόμενη εγγραφή) είναι το `next` της Perl· το `break`/`continue` στους βρόχους του awk είναι το `last`/`next` της Perl· το `for (k in arr)` του awk γίνεται ένα `for` της Perl πάνω στα `keys`. Οι αγκύλες και το `;` είναι ίδια· η Perl προσθέτει τις μεταθεματικές μορφές που διαβάζονται σαν μοτίβα του awk.

```
awk '{ if ($2 < 70) next; print }' input.txt
```

```
use v5.44;
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
```

Η παγίδα: οι χειριστές σύγκρισης της Perl έρχονται σε δύο οικογένειες και επιλέγετε βάσει του *τύπου των τελεστέων*, ενώ το awk έχει μόνο `</==` και για τους δύο. Οι αριθμοί χρησιμοποιούν `==`, `<`, `>`, `<=>`· οι συμβολοσειρές χρησιμοποιούν `eq`, `lt`, `gt`, `cmp`. Το awk αποφασίζει αριθμητική-έναντι-συμβολοσειρά σύγκριση βάσει των τιμών κατά τον χρόνο εκτέλεσης· η Perl σάς κάνει να επιλέξετε τον χειριστή. Η χρήση του `==` σε συμβολοσειρές (ή του `eq` σε αριθμούς) είναι ένα συχνό, σιωπηλό σφάλμα - το `"abc" == "xyz"` είναι αληθές στην Perl (και τα δύο μετατρέπονται αριθμητικά σε 0), κάτι που σχεδόν ποτέ δεν είναι αυτό που εννοούσατε.

Μεταβλητές και δεδομένα

Αυτή είναι η πιο ευτυχής αντιστοίχιση στον οδηγό: ο συσχετιστικός πίνακας του awk είναι ένας **κατακερματισμός** της Perl, και τα καθημερινά ιδιώματα είναι σχεδόν πανομοιότυπα. Το `count[$1]++` του awk είναι το `$count{$F[0]}++` της Perl· το `for (k in count)` είναι `for my $k (keys %count)`· το `if (k in count)` είναι `exists`

`$count{$k}`. Η μόνη μετατόπιση είναι το sigil - ολόκληρος ο κατακερματισμός είναι `%count`, αλλά μία τιμή είναι `$count{$key}`.

```
awk '{ total[$1] += $2 + $3 }
     END { for (k in total) print k, total[k] }' input.txt
```

```
use v5.44;
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
```

Το ιδίωμα δύο αρχείων `NR==FNR` - φόρτωσε κλειδιά από το πρώτο αρχείο, μετά έλεγξε την ιδιότητα μέλους στο δεύτερο - είναι ένας ρητός βρόχος ανά αρχείο στην Perl, που είναι σαφέστερος από το one-liner του awk που αντικαθιστά:

```
use v5.44;
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
```

Η παγίδα: ένας κατακερματισμός δεν έχει εγγενή σειρά (το `for (k in arr)` του awk είναι επίσης χωρίς σειρά, οπότε αυτό δεν θα σας εκπλήξει), οπότε επαναλαμβάνετε με `sort keys %h` όταν θέλετε σταθερή σειρά. Η ιδιότητα μέλους είναι `exists $h{$k}`, όχι το `k in arr` του awk - και απλώς το *να διαβάσετε* το `$h{$k}` για να το ελέγξετε θα αυτο-δημιουργήσει το κλειδί σε ύπαρξη σε κάποια περιβάλλοντα, ενώ το `exists` ποτέ δεν το κάνει. Οι αριθμοί και οι συμβολοσειρές αυτο-εξαναγκάζονται στην Perl όπως στο awk, αλλά θυμηθείτε τον διαχωρισμό `==` έναντι `eq` από τη *Ροή ελέγχου*.

Παγίδες ερχόμενοι από το awk

Οι παγίδες, συγκεντρωμένες για μια τελική ματιά πριν γράψετε:

- **Τα πεδία είναι μηδενοβάσει.** Το `$1` του awk είναι `$F[0]`, το `$2` είναι `$F[1]`, και όλη η εγγραφή `$0` είναι `$_`. Το `NF` είναι `scalar @F`· το τελευταίο πεδίο είναι `$F[-1]`, όχι `$F[NF]`.
- **Το `<$fh>` κρατά την τελική νέα γραμμή,** σε αντίθεση με το `$0` του awk. Κάντε `chomp` για να την αφαιρέσετε.
- **Η `substr` είναι μηδενοβάσει,** ενώ του awk είναι βάσει του ένα: το `substr($s,1,3)` του awk είναι `substr($s,0,3)` στην Perl.
- **Το `s///` επιστρέφει ένα πλήθος, όχι τη νέα συμβολοσειρά.** Το `my $x = $s =~ s/a/b/` βάζει το *πλήθος* αντικαταστάσεων στο `$x`. Αντιγράψτε πρώτα, ή χρησιμοποιήστε τη σημαία `/r` για να πάρετε μια νέα συμβολοσειρά.
- **Δύο οικογένειες σύγκρισης.** Οι αριθμοί χρησιμοποιούν `==`/`<`/`<=>`· οι συμβολοσειρές χρησιμοποιούν `eq`/`lt`/`cmp`. Το μοναδικό `==`/`<` του awk χωρίζεται στα δύο, επιλεγόμενο βάσει χειριστή, όχι βάσει τιμής. Το `"abc" == "xyz"` είναι αληθές (και τα δύο μετατρέπονται αριθμητικά σε 0).
- **Το γυμνό `split` είναι η προεπιλεγμένη λειτουργία τύπου awk.** Το `split` (χωρίς ορίσματα) διαχωρίζει το `$_` βάσει κενών χαρακτήρων και αφαιρεί τα αρχικά κενά. Το `split / /` (ένα regex με ένα κενό) είναι κυριολεκτικό και *δεν* αφαιρεί - χρησιμοποιήστε το σκέτο `split` για το προεπιλεγμένο `FS` του awk.
- **Η ιδιότητα μέλους είναι `exists $h{$k}`, όχι `k in arr`.** Ο έλεγχος ενός κλειδιού μέσω ανάγνωσης του `$h{$k}` μπορεί να το αυτο-δημιουργήσει· το `exists` ποτέ δεν το κάνει.
- **Η `open` δεν ματαιώνει σε αποτυχία.** Χρησιμοποιήστε `open my $fh, '<', $path or die "...: $!"`. Το `$!` είναι η συμβολοσειρά σφάλματος του συστήματος.
- **Οι διακόπτες έμμεσου βρόχου (`-n`/`-p`/`-a`/`-F`/`-i`/`-l`) δεν αναγνωρίζονται ακόμη από αυτό το build του `pperl`.** Μόνο το `-e` λειτουργεί σήμερα. Γράψτε τον ρητό βρόχο `while (<$fh>) { my @F = split; ... }`, που είναι ακριβώς αυτό στο οποίο αναπτύσσονται οι διακόπτες, μέχρι να προστεθούν.

Από Shell σε Perl

Ξέρετε το κέλυφος. Κολλάτε προγράμματα μεταξύ τους με σωληνώσεις, καταφεύγετε στα `sed`, `awk`, `grep` και `cut` σε μια σωλήνωση χωρίς σκέψη, συλλαμβάνετε την έξοδο εντολών με `$(...)`, και προγραμματίζετε ροή ελέγχου με `for f in *`, `while read line` και `case`. Αυτός ο οδηγός αντιστοιχίζει καθεμία από αυτές τις συνήθειες στην Perl, ώστε να κρατήσετε τα αντανακλαστικά σωλήνωσης και να αποκτήσετε τα πράγματα που το κέλυφος κάνει επώδυνα.

Όταν ένα σενάριο κελύφους μεγαλώσει πέρα από περίπου 50 γραμμές παράθεσης σε εισαγωγικά και σωληνώσεων `sed`/`awk`/`grep`, η Perl είναι το φυσικό επόμενο βήμα: τα κάνει όλα σε μία γλώσσα, με πραγματικές δομές δεδομένων αντί για συμβολοσειρές οριοθετημένες με κενά. Το πλαίσιο που πρέπει να κρατήσετε: **η Perl είναι η σωλήνωση κελύφους, αλλά η γλώσσα του φίλτρου είναι μια πραγματική γλώσσα.** Εξακολουθείτε να κάνετε open σε σωληνώσεις, εξακολουθείτε να κάνετε glob, εξακολουθείτε να τρέχετε

εντολές και να διαβάζετε την έξοδό τους. Αυτό που αλλάζει είναι ότι ανάμεσα στις σωληνώσεις έχετε τώρα `@arrays`, `%hashes`, πραγματικό regex, και αριθμητική που δεν χρειάζεται `$(( ))`. Ο ένας τύπος δεδομένων του κελύφους είναι η συμβολοσειρά· η Perl σάς δίνει πραγματικούς περιέκτες.

Every runnable example here was executed on `pperl`, a Rust reimplementation of Perl 5.44; the `# ...` comments show its real output. Start scripts with:

```
use v5.44;
```

Αυτή η μία γραμμή ενεργοποιεί το `strict`, τα `warnings` και την ενσωματωμένη `say` (μια `print` με τελική νέα γραμμή), το σύγχρονο σύνολο χαρακτηριστικών που χρησιμοποιείται παντού.

Σημείωση

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 5.44** 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.

Πώς να το διαβάσετε

- *Ο βρόχος εγγραφών* - το `while read line` γίνεται `while (<$fh>)` και `$_`.
- *Πεδία και διαχωρισμός* - τα πεδία `cut`/`IFS`/`awk` γίνονται `split` και ένας πραγματικός πίνακας.
- *Regex και αντικατάσταση* - οι παραμετρικές αναπτύξεις `${var#...}`/`${var/a/b}` του κελύφους γίνονται πραγματικό regex.
- *Διευθύνσεις και εύρη* - `grep -n`, `awk NR==2` και εύρη γραμμών.
- *I/O, σωληνώσεις και επιτόπια επεξεργασία* - η καρδιά: open, `-|`/`|-`, ανακατεύθυνση, here-docs, `$(...)`.
- *One-liners και διακόπτες* - `-n`, `-p`, `-e`, `BEGIN`/`END`.
- *Ροή ελέγχου* - `for`/`while`/`case`/`if`, έλεγχοι αρχείων, κατάσταση εξόδου.
- *Μεταβλητές και δεδομένα* - το `$var` γίνεται `my $var`, συν τους `@arrays` και `%hashes` που το κέλυφος ποτέ δεν είχε.
- *Παγίδες ερχόμενοι από το κέλυφος* - ρίξτε μια ματιά πριν γράψετε.

Ο βρόχος εγγραφών

Το ιδίωμα γραμμή-τη-φορά του κελύφους είναι `while read line; do ... done`, που διαβάζει μία γραμμή ανά επανάληψη στο `$line` (και την περικόπτει σύμφωνα με το `IFS`). Το αντίστοιχο της Perl είναι `while (<$fh>)`, που διαβάζει μία γραμμή ανά επανάληψη στο `$_`, την έμμεση μεταβλητή θέματος. Η κρίσιμη διαφορά: η Perl κρατά την τελική νέα

γραμμή, οπότε κάνετε εσείς chomp, ακριβώς εκεί όπου το `read` του κελύφους θα την είχε καταπιεί.

```
while read line; do
    echo "got: $line"
done < data.txt
```

```
use v5.44;
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";
}
close $fh;
# got: apple
# got: banana
# got: cherry
```

(Το αρχείο εισόδου περιείχε `apple`, `banana`, `cherry`.) Διαβάστε με το γυμνό `<$fh>` και η γραμμή καταλήγει στο `$_`, την προεπιλογή στην οποία λειτουργούν οι περισσότερες ενσωματωμένες συναρτήσεις γραμμών. Ο αριθμός γραμμής βρίσκεται στο `$.`. Αυτή είναι η μορφή στην οποία καταφεύγετε όταν μεταφράζετε ένα φίλτρο:

```
use v5.44;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (<$fh>) {                 # line lands in $_
    chomp;                      # chomp $_ by default
    say "$. : $_";              # $. is the line number
}
close $fh;
# 1 : apple
# 2 : banana
# 3 : cherry
```

Η παγίδα: το `read` διαχωρίζει βάσει του `IFS` και περικόπτει· το `<$fh>` δεν κάνει κανένα από τα δύο. Μια ανάγνωση γραμμής στην Perl είναι όλη η ακατέργαστη γραμμή συμπεριλαμβανομένου του `\n`, οπότε το `chomp` είναι το αντανακλαστικό σας, και ο διαχωρισμός πεδίων είναι ένα ξεχωριστό, ρητό βήμα (η επόμενη ενότητα). Ό,τι χάνετε στην αυτόματη περικοπή το κερδίζετε στο να μην παραμορφώνει σιωπηλά το `IFS` τα δεδομένα σας.

Πεδία και διαχωρισμός

Στο κέλυφος φτάνετε στα πεδία με `cut -d: -f1`, με `awk '{print $1}'`, ή ορίζοντας το `IFS` και αφήνοντας το `read` να διαχωρίσει για εσάς. Η μία απάντηση της Perl είναι το `split`: δώστε του έναν διαχωριστή και μια συμβολοσειρά, πάρτε πίσω έναν πραγματικό πίνακα. Τα θεσιακά πεδία που το κέλυφος αριθμεί `$1`, `$2` γίνονται στοιχεία πίνακα `$f[0]`, `$f[1]` - **μηδενοβάσει**, σε αντίθεση με το βάσει-του-ένα `$1` του awk.

```
line='root:x:0:0:root:/root:/bin/bash'
user=$(echo "$line" | cut -d: -f1)
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
shell=$(echo "$line" | cut -d: -f7)
echo "$user $shell"
```

```
use v5.44;
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
```

Για να διαχωρίσετε σε διαδοχές κενών χαρακτήρων - τον προεπιλεγμένο διαχωρισμό λέξεων του κελύφους, και την προεπιλογή του awk - χρησιμοποιήστε το ειδικό μοτίβο `split ' '` (μια κυριολεκτική συμβολοσειρά ενός κενού, που επίσης παραλείπει τα αρχικά κενά). Για να ενώσετε ξανά τα πεδία, `join`:

```
use v5.44;
my @words = split ' ', "  the   quick brown  ";
say scalar @words;            # 3   (leading/inner blanks collapsed)
say join "|", @words;         # the|quick|brown
```

Η παγίδα: το `split /:/` με ένα κυριολεκτικό μοτίβο κρατά κενά τελικά πεδία μόνο μέχρι ένα σημείο - εξ ορισμού η Perl απορρίπτει τα κενά τελικά πεδία, οπότε το `split /:/, "a::"` δίνει `("a")`, όχι `("a", "", "")`. Δώστε ένα αρνητικό όριο (`split /:/, $s, -1`) για να τα κρατήσετε, το αντίστοιχο του να ενδιαφέρεστε για τα κενά τελικά πεδία του cut. Και το `split ' '` (η μαγική μορφή με κενά) **δεν** είναι το ίδιο με το `split / /` (ένα μοναδικό κυριολεκτικό κενό): η πρώτη συμπτύσσει διαδοχές και περικόπτει, η δεύτερη όχι.

Regex και αντικατάσταση

Οι παραμετρικές αναπτύξεις `${}` του κελύφους γίνονται πραγματικό regex - και αυτό είναι μια μεγάλη αναβάθμιση. Τα `${var#prefix}`, `${var%suffix}`, `${var/a/b}` και `${var//a/b}` είναι το καθένα μια ειδικού σκοπού, περιορισμένη επεξεργασία συμβολοσειράς. Η Perl αντικαθιστά ολόκληρη την οικογένεια με έναν χειριστή, το `s///`, που δέχεται μια πραγματική κανονική έκφραση και το πλήρες σύνολο σημαιών.

```
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 v5.44;
my $f = "report.txt.bak";
(my $base = $f) =~ s/\.bak$//;       # ${f%.bak}
say $base;                           # report.txt
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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
```

Επειδή είναι ένα πραγματικό regex, τα πράγματα που το κέλυφος δεν μπορεί να κάνει καθόλου είναι τετριμμένα: οι ομάδες σύλληψης καταλήγουν στα `$1`, `$2`· η σημαία `/e` εκτελεί κώδικα Perl στην αντικατάσταση· η σημαία `/r` επιστρέφει ένα αντίγραφο αντί για επιτόπια επεξεργασία· και τα `/i`, `/x` τροποποιούν το ταίριασμα.

```
use v5.44;
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)
```

Η παγίδα: το γυμνό `s///` επεξεργάζεται τον στόχο του και επιστρέφει το *πλήθος* των αντικαταστάσεων, όχι τη νέα συμβολοσειρά - οπότε τα παραδείγματα αντιγράφουν πρώτα με `(my $t = $s) =~ ...` ή χρησιμοποιούν τη σημαία `/r`. Γράφοντας `my $new = ($s =~ s/a/b/)` βάζετε έναν αριθμό στο `$new`. Αυτή είναι η πιο συχνή έκπληξη της πρώτης μέρας. Σημειώστε επίσης ότι οι οριοθέτες είναι ελεύθεροι: τα `s{...}{...}` και `s#...#...#` σας γλιτώνουν από τις ανάποδες καθέτους στις καθέτους ενός μοτίβου διαδρομής, όπως το `${path##*/}` του κελύφους ήδη τις αποφεύγει.

Διευθύνσεις και εύρη

Η επιλογή γραμμών βάσει αριθμού ή μοτίβου - `grep`, `grep -n`, `awk 'NR==2'`, `sed -n '/a/,/b/p'` - γίνεται ένα μεταθεματικό `if` σε μια συνθήκη μέσα στον βρόχο. Ο αριθμός γραμμής είναι `$.`· ο έλεγχος μοτίβου είναι `/pat/` έναντι του `$_`.

```
grep -n an data.txt              # number + matching lines
awk 'NR==2' data.txt             # the second line
```

```
use v5.44;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (<$fh>) {
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    chomp;
    say "$. : $_" if /an/;       # grep -n 'an'
}
close $fh;
# 2 : banana
```

Ένα εύρος γραμμών - ο επιλογέας `/start/,/end/` του sed/awk - γίνεται ο χειριστής εύρους **flip-flop** της Perl `/start/ .. /end/`, αληθής από τη γραμμή που ταιριάζει με το αριστερό μοτίβο έως τη γραμμή που ταιριάζει με το δεξί, συμπεριλαμβανομένων:

```
use v5.44;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (<$fh>) {
    print if /banana/ .. /cherry/;   # the /start/,/end/ range
}
close $fh;
# banana
# cherry
```

Η παγίδα: σε έναν λογικό έλεγχο το `..` είναι ο χειριστής flip-flop (εύρους), **όχι** ο χειριστής εύρους λίστας (που είναι το `1 .. 10` που χτίζει μια λίστα). Η Perl τα ξεχωρίζει βάσει περιβάλλοντος - στη λογική θέση ενός `if`, το `..` είναι ο επιλογέας εύρους γραμμών με κατάσταση· σε περιβάλλον λίστας χτίζει μια ακολουθία. Οι ίδιες δύο τελείες, που αποφασίζονται από το πού τις γράφετε.

I/O, σωληνώσεις και επιτόπια επεξεργασία

Αυτή είναι η καρδιά του οδηγού, επειδή είναι η καρδιά του προγραμματισμού κελύφους. Ό,τι κάνετε με `|`, `>`, `>>`, `2>&1`, `<<EOF` και `$( ... )` έχει μια άμεση γραφή στην Perl, και μόλις ένα σενάριο ξεπεράσει μια μοναδική σωλήνωση αυτή είναι η ενότητα που αποδίδει.

Υποκατάσταση εντολής. Τα `$(cmd)` και τα backticks του κελύφους γίνονται backticks της Perl ή `qx{...}`. Η σύλληψη εξαρτάται από το περιβάλλον: σε βαθμωτό περιβάλλον παίρνετε όλη την έξοδο ως μία συμβολοσειρά· σε περιβάλλον λίστας παίρνετε ένα στοιχείο ανά γραμμή.

```
out=$(echo hello)
mapfile -t lines < <(seq 1 3)
```

```
use v5.44;
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
```

Όταν δεν χρειάζεστε την έξοδο, η `system()` τρέχει μια εντολή και σας δίνει την κατάσταση εξόδου της (δείτε τη *Ροή ελέγχου* για το `$?`). Δώστε στη `system` μια *λίστα*

και δεν εμπλέκεται κανένα κέλυφος - η ασφαλής μορφή που αποφεύγει την κόλαση της παράθεσης σε εισαγωγικά:

```
use v5.44;
system('mkdir', '-p', '/tmp/sh_demo');   # list form: no shell, no␣
↪quoting
say "made it" if -d '/tmp/sh_demo';      # made it
```

Σωληνώσεις. Ένα στάδιο σωλήνωσης - η ανάγνωση της εξόδου μιας εντολής ή η τροφοδοσία της εισόδου μιας εντολής - είναι ένα open με λειτουργία σωλήνωσης. Το open `my $fh, '-|', LIST` διαβάζει το stdout μιας εντολής· το open `my $fh, '|-', LIST` γράφει στο stdin μιας εντολής. Η μορφή λίστας (εντολή συν ορίσματα ως ξεχωριστές συμβολοσειρές) δεν τρέχει κανένα κέλυφος, οπότε δεν υπάρχει τίποτα προς παράθεση σε εισαγωγικά.

```
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 v5.44;
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 v5.44;
open my $sort, '|-', 'sort' or die "pipe: $!";  # write to command␣
↪stdin
print {$sort} "$_\n" for qw(banana apple cherry);
close $sort;
# apple
# banana
# cherry
```

Ανακατεύθυνση. Τα `>` και `>>` είναι οι λειτουργίες open `'>'` (περικοπή) και `'>>'` (προσθήκη)· ένα open τριών ορισμάτων με έναν λεξιλογικό περιγραφέα είναι το ιδίωμα. Το `print {$fh} ...` γράφει σε έναν περιγραφέα. Για να συγχωνεύσετε το stderr ενός παιδιού στο συλληφθέν ρεύμα (2>&1), κρατήστε την ανακατεύθυνση στην εντολή που δίνετε στα backticks:

```
use v5.44;
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;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
# to-out
# to-err
```

Here-docs. Τα `<<EOF` (με παρεμβολή) και `<<'EOF'` (κυριολεκτικά) του κελύφους αντιστοιχίζονται ένα-προς-ένα στα `<<"END"` και `<<'END'` της Perl. Τα διπλά εισαγωγικά στον τερματιστή παρεμβάλλουν μεταβλητές· τα μονά εισαγωγικά όχι.

```
use v5.44;
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
```

Επιτόπια επεξεργασία. Το `sed -i` ξαναγράφει ένα αρχείο επιτόπια· η ιδιωματική Perl είναι ο ίδιος διακόπτης σε ένα one-liner `-p`. Το εκτελέσιμο αντίστοιχο σήμερα είναι η ρητή ανάγνωση-τροποποίηση-εγγραφή που εκτελεί το `-i` στο παρασκήνιο - ρούφηγμα, μετασχηματισμός, εγγραφή πίσω:

```
use v5.44;
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;

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
```

Η παγίδα: η open επιστρέφει ψευδές σε αποτυχία αντί να εγείρει σφάλμα, οπότε το ιδίωμα `or die "...: $!"` είναι υποχρεωτικό - το `$!` είναι η συμβολοσειρά σφάλματος του συστήματος, η σιωπηλή αποτυχία `2>/dev/null` του κελύφους γινόμενη ηχηρή και ρητή. Και όταν χτίζετε μια εντολή από μεταβλητές, προτιμήστε τη **μορφή λίστας** των system/open `('ls', '-l', $dir)` έναντι μιας μοναδικής συμβολοσειράς με παρεμβολή: η μορφή λίστας παρακάμπτει εντελώς το κέλυφος, οπότε δεν υπάρχει διαχωρισμός λέξεων, ούτε ανάπτυξη glob, ούτε παράθεση σε εισαγωγικά να την κάνετε λάθος - η θεραπεία για την κόλαση παράθεσης του κελύφους.

One-liners και διακόπτες

Το one-liner είναι το οικείο έδαφος του προγραμματιστή κελύφους, και της Perl επίσης: το φίλτρο που ρίχνετε σε μια σωλήνωση. Η οικογένεια διακοπτών που το κάνει να λειτουργεί: το `-e` παρέχει το πρόγραμμα, το `-n` το τυλίγει σε έναν βρόχο χωρίς αυτόματη εκτύπωση, το `-p` το τυλίγει σε έναν βρόχο με αυτόματη εκτύπωση, το `-i` επεξεργάζεται αρχεία επιτόπια, το `-l` χειρίζεται τα τέλη γραμμών, και τα `-a`/`-F` αυτοδιαχωρίζουν κάθε γραμμή σε `@F` (το άμεσο αντίστοιχο του awk, πεδία στο `@F` αντί για `$1`..`$NF`). Τα `BEGIN { }` και `END { }` τρέχουν πριν και μετά τον βρόχο, όπως τα BEGIN/END του awk.

```
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
```

Αυτά εμφανίζονται χωρίς έξοδο επειδή οι διακόπτες έμμεσου βρόχου δεν αναγνωρίζονται ακόμη από αυτό το build του `pperl` (δείτε τη σημείωση στην αρχή). Καθένας αντιστοιχίζεται σε μια επαληθευμένη μορφή ρητού βρόχου: το `perl -ne 'CODE'` είναι `while (<$fh>) { CODE }`, και το `perl -pe 'CODE'` είναι το ίδιο με μια τελική `print`. Η εκτελέσιμη μορφή «πρώτο πεδίο κάθε γραμμής»:

```
use v5.44;
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
# daemon
# bin
```

Η παγίδα: μέχρι να προστεθούν οι διακόπτες στο `pperl`, γράψτε τον ρητό βρόχο (τρέχει σήμερα και είναι ούτως ή άλλως αυτό στο οποίο αναπτύσσονται οι διακόπτες), και συμβουλευτείτε τον *οδηγό One-Liners* για τον πλήρη κατάλογο διακοπτών και δεκάδες συνταγές - είναι το πιο αποδοτικό πράγμα που μπορείτε να διαβάσετε ερχόμενοι από υπόβαθρο κελύφους, όπου το one-liner είναι το καθημερινό αντανακλαστικό.

Ροή ελέγχου

Τα `for`, `while`, `case` και `if`/`test` του κελύφους αντιστοιχίζονται σε βρόχους και συνθήκες της Perl, και αρκετές από τις αντιστοιχίσεις είναι ευχάριστα άμεσες.

Βρόχοι. Το `for f in *; do` γίνεται `for my $f (glob "*")`· το glob αρχείων `*.txt` είναι `glob "*.txt"` ή η μορφή readline `<*.txt>`.

```
for f in /tmp/sh_*.txt; do
    echo "file: $f"
done
```

```
use v5.44;
for my $f (sort glob "/tmp/sh_*.txt") {
    say "file: $f";
}
# file: /tmp/sh_data.txt
# file: /tmp/sh_passwd.txt
```

Συνθήκες και `test`. Αυτή είναι μια ευτυχής αντιστοίχιση: οι έλεγχοι αρχείων `[ -e ]`, `[ -d ]`, `[ -f ]` του κελύφους είναι οι **ίδιοι χειριστές `-X`** στην Perl. Η σύγκριση συμβολοσειρών και αριθμών χωρίζεται σε δύο οικογένειες, όπως γίνεται ανάμεσα στα `[ "$a" = "$b" ]` και `[ "$a" -eq "$b" ]`: η Perl χρησιμοποιεί eq/ne για συμβολοσειρές και ==/!= για αριθμούς. Τα `-z`/`-n` (κενή / μη κενή συμβολοσειρά) του κελύφους γίνονται `!length` / `length`.

```
if [ -f "$f" ] && [ -n "$name" ]; then
    echo ok
fi
[ "$a" = "$b" ]        # string equality
[ "$x" -eq "$y" ]      # numeric equality
```

```
use v5.44;
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`. Το `case` του κελύφους δεν έχει δίδυμο μίας δεσμευμένης λέξης· η ιδιωματική Perl είναι μια αλυσίδα `for`/`if` ή ένας τριαδικός καταρράκτης (ή, για πολλούς κλάδους, ένας κατακερματισμός αποστολής που αντιστοιχίζει κλειδιά σε αναφορές κώδικα).

```
case "$cmd" in
    start)   echo starting ;;
    stop)    echo stopping ;;
    *)       echo "unknown: $cmd" ;;
esac
```

```
use v5.44;
for my $cmd (qw(start stop bogus)) {
    my $r =
        $cmd eq 'start' ? "starting"
      : $cmd eq 'stop'  ? "stopping"
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
      :                     "unknown: $cmd";
    say "$cmd -> $r";
}
# start -> starting
# stop -> stopping
# bogus -> unknown: bogus
```

Κατάσταση εξόδου. Μετά από μια `system()`, η κατάσταση του παιδιού βρίσκεται στο `$?`, ακριβώς η μεταβλητή του κελύφους. Όμως το `$?` κρατά την ακατέργαστη κατάσταση `wait`, όχι τον σκέτο κωδικό εξόδου: ολισθήστε δεξιά κατά 8 για να πάρετε τον κωδικό που το κέλυφος θα εμφάνιζε ως `$?`. Οι δεσμευμένες λέξεις βρόχου `last` και `next` είναι το `break` και το `continue` του κελύφους.

```
use v5.44;
system('sh', '-c', 'exit 3');
say "raw \$? = $?";           # raw $? = 768
say "exit code = ", $? >> 8;  # exit code = 3
```

Η παγίδα: το `$? >> 8` είναι ο κωδικός εξόδου· τα χαμηλά 8 bits κρατούν το σήμα που σκότωσε το παιδί (αν υπάρχει). Το κέλυφος το κρύβει πίσω από το δικό του `$?`· στην Perl κάνετε εσείς την ολίσθηση. Το να το ξεχάσετε είναι ο λόγος που το `if ($? == 1)` μετά από μια εντολή που τερμάτισε με 1 είναι σχεδόν πάντα λάθος - η ακατέργαστη τιμή είναι 256, όχι 1.

Μεταβλητές και δεδομένα

Μια μεταβλητή κελύφους είναι πάντα μια συμβολοσειρά - `var=value`, χρησιμοποιούμενη ως `$var` ή `${var}`, και τοποθετημένη σε εισαγωγικά ως `"$var"` για να επιβιώσει του διαχωρισμού λέξεων. Το βαθμωτό της Perl είναι `my $var`, με λεξιλογική εμβέλεια, και **δεν χρειάζεται κανένας χορός παράθεσης σε εισαγωγικά**: ένα βαθμωτό της Perl κρατά μία τιμή και δεν ξαναδιαχωρίζεται κατά τη χρήση. Εκεί που το κέλυφος αναγκάζει τα πάντα να περάσουν μέσα από συμβολοσειρές, η Perl προσθέτει τους δύο περιέκτες που το κέλυφος ποτέ δεν είχε: διατεταγμένους `@arrays` και κλειδιού/τιμής `%hashes`.

```
name="Ada"
greeting="Hello, ${name}"
langs="Perl Python"          # a "list" is really a string + IFS
```

```
use v5.44;
my $name = "Ada";
my $greeting = "Hello, $name";   # interpolation, like "${name}"
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
```

Ορίσματα σεναρίου. Οι θεσιακές παράμετροι του κελύφους αντιστοιχίζονται στο `@ARGV`. Το ναρκοπέδιο παράθεσης σε εισαγωγικά του `"$@"` έναντι του `$*` απλώς δεν υπάρχει: το `@ARGV` είναι ένας πραγματικός πίνακας, κάθε όρισμα ένα καθαρό στοιχείο

ανεξάρτητα από τα κενά ή τα glob που περιείχε. Τα `$1`..`$9` γίνονται `$ARGV[0]`..`$ARGV[8]`, το `$#` (το πλήθος) γίνεται `scalar @ARGV`, και το `$0` (το όνομα του σεναρίου) γίνεται `$0`.

```
echo "count: $#"
echo "first: $1"
echo "all: $@"
```

```
use v5.44;
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
```

Συσχετιστικοί πίνακες. Ένας κατακερματισμός της Perl είναι το `declare -A` του κελύφους καμωμένο σωστά - δηλωμένος με %, προσπελάσιμος μία τιμή τη φορά με `$h{key}`, και πολύ πιο εργονομικός από τους πρόχειρα προσαρτημένους συσχετιστικούς πίνακες του κελύφους.

```
use v5.44;
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
```

Η παγίδα: μια μεταβλητή κελύφους είναι εξ ορισμού καθολική και μια μεταβλητή `my` της Perl έχει λεξιλογική εμβέλεια στο μπλοκ της - η αντίθετη προεπιλογή. Το `local` του κελύφους (δυναμική εμβέλεια μέσα σε μια συνάρτηση) είναι το πλησιέστερο στο `local` της Perl, αλλά για συνηθισμένες μεταβλητές θέλετε το `my`, που είναι αληθινή λεξιλογική εμβέλεια. Και η παρεμβολή συμβαίνει μόνο σε διπλά εισαγωγικά: το `"$name"` παρεμβάλλει, το `'$name'` είναι οι κυριολεκτικοί τέσσερις χαρακτήρες, ακριβώς όπως στο κέλυφος.

> Η Perl έχει επίσης πλήρεις υπορουτίνες, αναφορές, αρθρώματα και OO αν το σενάριό σας ξεπεράσει ένα one-liner - δείτε τον *οδηγό Αντικειμενοστραφούς Προγραμματισμού* και τους άλλους οδηγούς coming-from.

Παγίδες ερχόμενοι από το κέλυφος

Οι παγίδες, συγκεντρωμένες για μια τελική ματιά πριν γράψετε:

- **Το `<$fh>` κρατά την τελική νέα γραμμή.** Το `read` του κελύφους περικόπτει· η Perl όχι. Το `chomp` είναι το αντανακλαστικό σας σε κάθε ανάγνωση γραμμής.
- **Το `split` είναι ξεχωριστό βήμα, και είναι μηδενοβάσει.** Τα πεδία είναι `$f[0]`, `$f[1]`, όχι το βάσει-του-ένα `$1` του awk. Το `split ' '` (μαγικά κενά) περικόπτει και συμπτύσσει· το `split / /` (ένα κυριολεκτικό κενό) όχι.

- **Το `s///` επιστρέφει ένα πλήθος, όχι τη νέα συμβολοσειρά.** Το `my $x = $s =~ s/a/b/` βάζει το *πλήθος* αντικαταστάσεων στο `$x`. Αντιγράψτε πρώτα (`my $t = $s) =~ s/.../.../`) ή χρησιμοποιήστε τη σημαία `/r`.
- **Το `$? >> 8` είναι ο κωδικός εξόδου.** Το ακατέργαστο `$?` μετά τη `system` είναι η πλήρης κατάσταση wait· ο κωδικός εξόδου είναι τα υψηλά 8 bits. Το `if ($? == 1)` είναι σχεδόν πάντα σφάλμα· θέλετε `if (($? >> 8) == 1)`.
- **Η `open` δεν εγείρει σφάλμα σε αποτυχία.** Χρησιμοποιήστε `open my $fh, '<', $path or die "...: $!"`. Το `$!` είναι η συμβολοσειρά σφάλματος του συστήματος - η σιωπηλή αποτυχία του κελύφους γινόμενη ρητή.
- **Προτιμήστε τη μορφή λίστας των `system`/`open` έναντι μιας συμβολοσειράς με παρεμβολή.** Το `system('ls', '-l', $dir)` δεν τρέχει κανένα κέλυφος: δεν υπάρχει διαχωρισμός λέξεων, ούτε έκπληξη glob, ούτε κόλαση παράθεσης. Η μορφή συμβολοσειράς (`system "ls -l $dir"`) επανεισάγει κάθε παγίδα παράθεσης κελύφους από την οποία ήρθατε εδώ να ξεφύγετε.
- **Το `my` είναι λεξιλογικό· οι μεταβλητές κελύφους είναι καθολικές.** Δηλώστε με `my` και περιορίστε την εμβέλεια στο μπλοκ· καταφύγετε στο `local` μόνο για τη συμπεριφορά δυναμικής εμβέλειας του κελύφους, που σπάνια είναι αυτό που θέλετε.
- **Η σύγκριση συμβολοσειρών έναντι αριθμών είναι δύο οικογένειες.** eq/ne για συμβολοσειρές, ==/!= για αριθμούς - ο διαχωρισμός `[ = ]` έναντι `[ -eq ]`, αλλά οι χειριστές είναι αντεστραμμένοι σε σχέση με το κέλυφος, οπότε διαβάστε τους προσεκτικά.
- **Οι διακόπτες έμμεσου βρόχου (`-n`/`-p`/`-a`/`-F`/`-i`/`-l`) δεν αναγνωρίζονται ακόμη από αυτό το build του `pperl`.** Μόνο το `-e` λειτουργεί σήμερα. Γράψτε τον ρητό βρόχο `while (<$fh>) { ... }`, που είναι ακριβώς αυτό στο οποίο αναπτύσσονται οι διακόπτες, μέχρι να προστεθούν.

Από Tcl σε Perl

Ξέρετε την Tcl. Γράφετε `set x 5`, υποκαθιστάτε με `$x`, καταφεύγετε στο `[expr {...}]` για να κάνετε αριθμητική, χτίζετε λίστες με `lappend` και τις διατρέχετε με `foreach`, και λυγίζετε συμβολοσειρές με `regexp` και `regsub`. Αυτός ο οδηγός αντιστοιχίζει καθεμία από αυτές τις συνήθειες στην Perl, ώστε να κρατήσετε κάθε αντανακλαστικό που έχετε και να αποκτήσετε τα πράγματα που η Tcl κάνει δύσκολα.

Η Tcl και η Perl μοιράζονται τις ίδιες ρίζες: προγραμματισμό Unix, μια κοσμοθεωρία με προτεραιότητα στη συμβολοσειρά, και την υποκατάσταση με `$` ως τον τρόπο που φτάνετε σε μια μεταβλητή. Τα περισσότερα από όσα ξέρετε αντιστοιχίζονται σχεδόν άμεσα - και η Perl σάς δίνει πραγματικές δομές δεδομένων (`@arrays` και `%hashes` ως πρωτεύοντες τύπους, όχι λίστες επιστρατευμένες για τον σκοπό) και ένα τεράστιο οικοσύστημα αρθρωμάτων όπου να προσγειωθείτε. Δύο συγκεκριμένα σημεία τριβής παίρνουν το βάρος σε αυτόν τον οδηγό: οι διακριτοί τύποι πίνακα/κατακερματισμού της Perl εκεί που η Tcl έχει λίστες και συσχετιστικούς πίνακες, και το regex της Perl (ένα υπερσύνολο εκείνου της Tcl). Το υλικό γραμμή-τη-φορά και one-liner είναι ελαφρύτερο εδώ, επειδή η Tcl δεν είναι γλώσσα one-liner· θα περάσετε τον χρόνο σας σε μεταβλητές, δεδομένα, ροή ελέγχου και regex.

Every runnable example here was executed on `pperl`, a Rust reimplementation of Perl 5.44; the `# ...` comments show its real output. Start scripts with:

```
use v5.44;
```

Αυτή η μία γραμμή ενεργοποιεί το `strict`, τα `warnings` και την ενσωματωμένη `say` (μια `print` με τελική νέα γραμμή, ο πρόχειρος δίδυμος του `puts` της Tcl), το σύγχρονο σύνολο χαρακτηριστικών που χρησιμοποιείται παντού.

Σημείωση

One-liner switches on this pperl build Perl's one-liner switches - `perl -ne`, `perl -pe`, `-a`, `-i`, `-l` - are **correct perl5 5.44** 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.

Πώς να το διαβάσετε

- *Μεταβλητές και υποκατάσταση* - το `set` γίνεται `my`, το `[expr]` εξαφανίζεται, η υποκατάσταση `$x` παραμένει.
- *Λίστες και πίνακες* - τα `lindex`/`lappend`/`lsort` γίνονται λειτουργίες πίνακα της Perl.
- *Συσχετιστικοί πίνακες και κατακερματισμοί* - το `set a(k) v` γίνεται `$a{k}`· το `dict` γίνεται κατακερματισμός.
- *Procs και αναφορές* - το `proc` γίνεται `sub`· τα `upvar`/`uplevel` γίνονται αναφορές.
- *Regex και αντικατάσταση* - τα `regexp`/ `regsub`/`string map` γίνονται `m//`/`s///`/`tr//`.
- *Ροή ελέγχου* - τα `if`/`while`/`foreach`/`switch` γίνονται της Perl, συν την παράθεση `{}` έναντι `""`.
- *Ο βρόχος γραμμής/εγγραφής* - ανάγνωση ενός αρχείου γραμμή προς γραμμή με `$_` και `$.`.
- *Πεδία και διαχωρισμός* - `split`/`join`, το αντίστοιχο του `split` της Tcl.
- *I/O, σωληνώσεις και επιτόπια επεξεργασία* - τα `open`/`gets`/`puts`/`exec` γίνονται `open`/`<$fh>`/`print`/`qx`.
- *One-liners και διακόπτες* - `-n`, `-p`, `-e`, σε συντομία.
- *Παγίδες ερχόμενοι από την Tcl* - ρίξτε μια ματιά πριν γράψετε.

Μεταβλητές και υποκατάσταση

Στην Tcl, το `set x 5` δημιουργεί μια μεταβλητή και το `$x` υποκαθιστά την τιμή της. Στην Perl, το `my $x = 5;` δηλώνει μια μεταβλητή με λεξιλογική εμβέλεια και το `$x` την υποκαθιστά - η σύνταξη υποκατάστασης που ήδη ξέρετε μεταφέρεται αυτολεξεί. Η ανακούφιση είναι ότι η Perl έχει εγγενείς χειριστές: δεν υπάρχει `[expr {...}]`. Η αριθμητική είναι απλώς αριθμητική.

```
set x 5
set name "world"
set sum [expr {$x + 3}]
puts "x is $x"
puts "sum is $sum"
```

```
use v5.44;
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
```

Η παράθεση σε εισαγωγικά αντιστοιχίζεται καθαρά, και αυτή είναι μια ευτυχής αντιστοίχιση: το "..." της Tcl (όπου συμβαίνει υποκατάσταση) είναι η συμβολοσειρά σε διπλά εισαγωγικά της Perl, και το {...} της Tcl (κυριολεκτικό, χωρίς υποκατάσταση) είναι η συμβολοσειρά σε μονά εισαγωγικά της Perl.

```
use v5.44;
my $name = "world";
say "hello, $name";         # hello, world   (interpolates)
say 'no $interp here';      # no $interp here (literal)
```

Η παγίδα: η Tcl είναι «τα πάντα είναι συμβολοσειρά» και η Perl είναι κοντά σε αυτό - ένα βαθμωτό αυτο-εξαναγκάζεται μεταξύ αριθμού και συμβολοσειράς βάσει περιβάλλοντος - αλλά η Perl χωρίζει τη *σύγκριση* σε δύο οικογένειες χειριστών, και επιλέγετε βάσει της πράξης, όχι της τιμής. Οι αριθμοί χρησιμοποιούν ==, !=, <, >· οι συμβολοσειρές χρησιμοποιούν eq, ne, lt, gt.

```
use v5.44;
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
```

Το == συγκρίνει τα "10" και "10.0" ως αριθμούς (ίσα)· το eq τα συγκρίνει ως συμβολοσειρές (διαφορετικά). Η χρήση του == εκεί που εννοούσατε eq, ή το αντίστροφο, είναι το ένα σιωπηλό σφάλμα για το οποίο σας προετοιμάζει η κοσμοθεωρία με προτεραιότητα στη συμβολοσειρά.

Λίστες και πίνακες

Μια λίστα της Tcl είναι ένας πίνακας της Perl, και τα ρήματα ταιριάζουν ένα προς ένα. Η μεγάλη μετατόπιση: στην Tcl *τα πάντα είναι λίστα*, συμπεριλαμβανομένων των συμβολοσειρών που διαχωρίζετε στο λεπτό· στην Perl ένας πίνακας (@a) είναι ένας διακριτός, πρωτεύων τύπος με το δικό του sigil. Όταν φτάνετε σε ένα στοιχείο, το sigil ακολουθεί το *στοιχείο*, οπότε είναι $a[0] (ένα βαθμωτό), όχι @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 v5.44;
my @items = ("apple", "pear", "fig");
say $items[0];                  # apple        (lindex $items 0)
say scalar @items;              # 3            (llength $items)
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";
}
```

Το `lsearch` - «βρες τον δείκτη ενός στοιχείου που ταιριάζει» - δεν έχει μία ενσωματωμένη συνάρτηση· η ιδιωματική Perl είναι το `grep` πάνω στο εύρος δεικτών, με το `$#items` ως τελευταίο δείκτη:

```
use v5.44;
my @items = ("apple", "pear", "fig", "plum");
my ($idx) = grep { $items[$_] eq "fig" } 0 .. $#items;   # lsearch
say "fig at $idx";              # fig at 2
```

Η παγίδα: το `scalar @items` είναι ο τρόπος που παίρνετε το μήκος (το `llength` της Tcl)· γράφοντας `my $n = @items` λειτουργεί επειδή η ανάθεση ενός πίνακα σε βαθμωτό δίνει το πλήθος του, αλλά γράψτε το `scalar @items` όταν θέλετε το μήκος μέσα σε μια μεγαλύτερη έκφραση. Και το `sort` συγκρίνει ως *συμβολοσειρές* εξ ορισμού, οπότε το `sort (10, 9, 100)` δίνει `10 100 9`· δώστε έναν αριθμητικό συγκριτή `sort { $a <=> $b } @nums` για αριθμητική σειρά, όπου τα `$a` και `$b` είναι τα δύο στοιχεία και το `<=>` είναι ο αριθμητικός τριαδικός χειριστής.

Συσχετιστικοί πίνακες και κατακερματισμοί

Ένας συσχετιστικός πίνακας της Tcl - `set a(red) 1` - είναι ένας κατακερματισμός της Perl. Η πρόσβαση στα στοιχεία αντιστοιχίζεται άμεσα: το `$a(red)` της Tcl είναι το `$a{red}` της Perl, και ολόκληρη η συλλογή είναι `%a` (η Tcl δεν έχει σύνταξη για τον πίνακα ως μία τιμή· η Perl έχει). Το `dict` της σύγχρονης Tcl αντιστοιχίζεται στον ίδιο κατακερματισμό της Perl, ή σε μια αναφορά κατακερματισμού όταν χρειάζεται να τον εμφωλεύσετε ή να τον περάσετε τριγύρω.

```
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)" }
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
unset a(blue)
```

```
use v5.44;
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)
```

Αυτό εκτυπώνει:

```
1
yes
red => 1
green => 2
blue => 3
```

Η παγίδα: η ιδιότητα μέλους είναι `exists $a{key}`, όχι μια ξεχωριστή εντολή, και το παχύ κόμμα `=>` είναι ο ιδιωματικός διαχωριστής κλειδιού/τιμής σε ένα κυριολεκτικό κατακερματισμού. Το `=>` επίσης αυτο-θέτει σε εισαγωγικά τη bareword στα αριστερά του, οπότε το `red => 1` δεν χρειάζεται εισαγωγικά γύρω από το `red`. Ένας κατακερματισμός δεν έχει εγγενή σειρά (ούτε ένας πίνακας της Tcl)· επαναλαμβάνετε με `sort keys %a` όταν θέλετε μια σταθερή, ακριβώς όπως καταφεύγετε στο `lsort [array names a]`.

Procs και αναφορές

Το `proc name {args} {body}` γίνεται το `sub name { ... }` της Perl. Ο κλασικός τρόπος της Tcl να διαβάζετε ορίσματα - θεσιακά, κατ" όνομα - έχει έναν άμεσο δίδυμο στην Perl: το `my ($a, $b) = @_;` αποσυσκευάζει τον πίνακα ορισμάτων `@_`. Η σύγχρονη Perl έχει επίσης υπογραφές, που διαβάζονται σαν τη λίστα `{args}` της Tcl και προσθέτουν προεπιλογές.

```
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 v5.44;
sub add {
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    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!
```

Το πέρασμα-κατ”-όνομα της Tcl - το upvar, όπου ένα proc φτάνει προς τα πάνω και μεταλλάσσει μια μεταβλητή του καλούντος - γίνεται μια **αναφορά** της Perl: πάρτε μία με \, ακολουθήστε την με $$ref. Περνάτε την αναφορά ρητά αντί να ονομάζετε τη μεταβλητή, κάτι που κάνει τη ροή δεδομένων ορατή στο σημείο κλήσης.

```
use v5.44;
sub increment {
    my ($ref) = @_;
    $$ref++;                # mutate through the reference (Tcl: upvar)
}
my $count = 10;
increment(\$count);         # pass a reference to $count
say $count;                 # 11
```

Η παγίδα: το @_ είναι ο πίνακας ορισμάτων, και το my ($a, $b) = @_; *αντιγράφει* τα ορίσματα σε επώνυμες τοπικές μεταβλητές - η γραμμή που σχεδόν πάντα θέλετε στην αρχή μιας υπορουτίνας. Η καταφυγή σε ψευδωνυμοποίηση τύπου upvar σπάνια χρειάζεται στην Perl· όταν όντως χρειάζεται να μεταλλάξετε δεδομένα του καλούντος, περάστε μια αναφορά (\$x, \@arr, \%h) σκόπιμα αντί να ψευδωνυμίζετε κατ” όνομα. Οι αναφορές είναι επίσης ο τρόπος που η Perl εμφωλεύει δεδομένα (έναν πίνακα κατακερματισμών, έναν κατακερματισμό πινάκων)· ο ίδιος μηχανισμός που αντικαθιστά το upvar είναι αυτός που αντικαθιστά τις εμφωλευμένες λίστες κωδικοποιημένες σε συμβολοσειρά της Tcl.

Regex και αντικατάσταση

Αυτή είναι μια ουσιαστική ενότητα. Τα regexp και regsub της Tcl χρησιμοποιούν Προηγμένες Κανονικές Εκφράσεις· το regex της Perl είναι ένα **υπερσύνολο**, οπότε τα μοτίβα που ξέρετε μεταφέρονται και κερδίζουν χαρακτηριστικά. Η Perl κάνει το regex *σύνταξη*, όχι εντολή: το =~ δένει ένα μοτίβο σε μια συμβολοσειρά, το m// ταιριάζει, το s/// αντικαθιστά, και το tr/// μεταγραμματίζει.

```
set s "order 42 shipped"
if {[regexp {(\d+)} $s -> num]} {
    puts "number: $num"
}
set t [regsub -all {o} "foo boo zoo" 0]
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
puts $t
set u [string map {a A b B} "abcabc"]
puts $u
```

```
use v5.44;
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/O/g;    # regsub -all  ->  s///g
say $t;                                # fOO bOO zOO
(my $u = "abcabc") =~ tr/ab/AB/;       # string map (char-for-char)
say $u;                                # ABcABc
```

Και η σύλληψη αντιστοιχίζεται: η μεταβλητή αντιστοιχίας `-> num` της Tcl γίνεται τα `$1`, `$2` της Perl, και ούτω καθεξής, με τις επώνυμες συλλήψεις να καταλήγουν στο `%+`:

```
use v5.44;
if ("2026-06-21" =~ /(?<y>\d{4})-(?<m>\d{2})/) {
    say "$+{y} / $+{m}";        # 2026 / 06
}
```

Το `string match` της Tcl είναι ταίριασμα glob, όχι regex (`string match *.txt $name`). Στην Perl, αγκυρώστε ένα regex αντί γι" αυτό - το `/\.txt$/` είναι το αντίστοιχο - επειδή η μία γλώσσα μοτίβων της Perl καλύπτει και τις δύο δουλειές.

```
use v5.44;
my $name = "report.txt";
say "matches" if $name =~ /\.txt$/;     # string match *.txt $name
# matches
```

Η παγίδα: το `s///` τροποποιεί τον στόχο του *επιτόπια* και επιστρέφει το *πλήθος* των αντικαταστάσεων, όχι τη νέα συμβολοσειρά - γι" αυτό τα παραπάνω παραδείγματα αντιγράφουν πρώτα με `(my $t = "...")` πριν αντικαταστήσουν. Για να πάρετε μια νέα συμβολοσειρά και να αφήσετε το πρωτότυπο ανέγγιχτο, χρησιμοποιήστε τη σημαία `/r`: `my $new = $s =~ s/o/O/gr`. Και το `tr///` (το αντίστοιχο του `string map` για μεμονωμένους χαρακτήρες) **δεν** είναι regex: η αριστερή του πλευρά είναι ένα κυριολεκτικό σύνολο χαρακτήρων, οπότε το `tr/ab/AB/` αντιστοιχίζει μόνο τα `a` και `b`. Για εργασία με μοτίβα, καταφύγετε στο `s///`.

Ροή ελέγχου

Οι δομές ελέγχου αντιστοιχίζονται σχεδόν μία προς μία· η γραφή αλλάζει και τα άγκιστρα μένουν. Το `if {cond} {body}` της Tcl γίνεται το `if (cond) { body }` της Perl, το `elseif` γίνεται `elsif`, και το `foreach` γίνεται `for` (οι δύο γραφές είναι ψευδώνυμα στην Perl· το `foreach` λειτουργεί επίσης).

```
if {$n < 0} {
    set sign neg
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
} 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 v5.44;
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}
```

Το `switch` της Tcl δεν έχει μία δεσμευμένη λέξη στην Perl· η ιδιωματική αντικατάσταση είναι ένας **κατακερματισμός αποστολής** που αντιστοιχίζει κάθε περίπτωση σε μια αναφορά κώδικα, που είναι ταχύτερος και πιο ευέλικτος από μια αλυσίδα `if`:

```
use v5.44;
my $cmd = "start";
my %dispatch = (
    start => sub { "starting" },
    stop  => sub { "stopping" },
);
say $dispatch{$cmd}->();     # starting
```

Η παγίδα: η διάκριση `{}` έναντι `""` στην οποία βασίζεστε στην Tcl για αναβολή έναντι υποκατάστασης αντιστοιχίζεται στα `''` έναντι `""` της Perl για συμβολοσειρές (που καλύφθηκαν παραπάνω), αλλά τα άγκιστρα *μπλοκ* της Perl δεν είναι παράθεση - ένα `{ ... }` της Perl μετά από `if`/`while`/`for` είναι ένα μπλοκ κώδικα, πάντα αποτιμώμενο, ποτέ μια αναβληθείσα συμβολοσειρά. Δεν υπάρχει βήμα «το σώμα είναι συμβολοσειρά μέχρι να το τρέξω»· η Perl αναλύει το μπλοκ κατά τη μεταγλώττιση. Οι μεταθεματικοί τροποποιητές είναι ένα μπόνους για το οποίο η Tcl δεν έχει μορφή: τα `say $_ for @items;` και `say "neg" if $n < 0;` φέρουν τον βρόχο ή τη συνθήκη μετά την εντολή.

Ο βρόχος γραμμής/εγγραφής

Η Tcl διαβάζει ένα αρχείο με `set fh [open f r]; while {[gets $fh line] >= 0} {...}`. Το αντίστοιχο της Perl είναι ένας βρόχος `while (<$fh>)`, με κάθε γραμμή να καταλήγει στο `$_` (την έμμεση μεταβλητή θέματος) ή σε ένα επώνυμο βαθμωτό. Ο αριθμός γραμμής βρίσκεται στο `$.`.

```
set fh [open /tmp/tcl_data.txt r]
while {[gets $fh line] >= 0} {
    puts "got: $line"
}
close $fh
```

```
use v5.44;
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
```

(Το αρχείο εισόδου περιείχε `apple`, `banana`, `cherry`.) Η χρήση του `$_` και του μετρητή γραμμών `$.` διαβάζεται ακόμη πιο κοντά σε έναν βρόχο εγγραφών:

```
use v5.44;
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
# 3 : cherry
```

Η παγίδα: το `gets` της Tcl αφαιρεί την τελική νέα γραμμή για εσάς· το `<$fh>` της Perl την κρατά, οπότε καλέστε `chomp` όταν δεν τη θέλετε. Χωρίς όρισμα, τα `chomp` και `say`/`print` έχουν αμφότερα προεπιλογή το `$_`, γι” αυτό ο δεύτερος βρόχος ποτέ δεν ονομάζει τη μεταβλητή της γραμμής.

Πεδία και διαχωρισμός

Το `split $line $sep` της Tcl επιστρέφει μια λίστα· το `split /sep/, $line` της Perl επιστρέφει έναν πίνακα, και το `join` το ξανασυναρμολογεί. Η διαφορά: ένας διαχωριστής split της Perl είναι ένα *regex*, οπότε παίρνετε όλη τη γλώσσα μοτίβων δωρεάν.

```
set line "a:b:c"
set parts [split $line ":"]
puts [llength $parts]
puts [lindex $parts 1]
puts [join $parts ","]
```

```
use v5.44;
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 ",")
```

Η παγίδα: το `split ' '` με μια κυριολεκτική συμβολοσειρά ενός κενού είναι μια ειδική περίπτωση που διαχωρίζει σε *οποιαδήποτε διαδοχή κενών χαρακτήρων* και απορρίπτει τα αρχικά κενά πεδία - η συμπεριφορά τύπου awk «διαχώρισε σε λέξεις», και συνήθως αυτό που θέλετε για ελεύθερης μορφής είσοδο. Το `split / /` (ένα regex με ένα κενό) είναι το κυριολεκτικό «διαχώρισε σε κάθε μεμονωμένο κενό» και δεν συμπτύσσει διαδοχές. Καταφύγετε στο `split ' '` για να ζητάτε λέξεις:

```
use v5.44;
my @words = split ' ', "  the  quick brown  ";
say "@words";                   # the quick brown
```

I/O, σωληνώσεις και επιτόπια επεξεργασία

Τα open/gets/puts/read/close της Tcl αντιστοιχίζονται στα open/<$fh>/print/slurp/close της Perl. Το ιδίωμα είναι το open τριών ορισμάτων με έναν λεξιλογικό περιγραφέα. Το exec της Tcl - τρέξε μια εντολή και σύλλαβε την έξοδό της - γίνεται το qx{...} (backticks) της Perl ή `system` όταν δεν χρειάζεστε την έξοδο.

```
set out [exec echo hello from shell]
puts "exec said: $out"
```

```
use v5.44;
my $out = qx{echo hello from shell};    # exec  ->  qx// (backticks)
chomp $out;
say "exec said: $out";                  # exec said: hello from␣
↪shell
```

Η Tcl δεν έχει ενσωματωμένη επιτόπια επεξεργασία αρχείων· διαβάζετε, μετασχηματίζετε και γράφετε πίσω. Ο διακόπτης `-i` της Perl είναι η μορφή one-liner ακριβώς αυτού, και το εκτελέσιμο αντίστοιχο πλήρους σεναρίου είναι μια ανάγνωση-τροποποίηση-εγγραφή που μπορείτε να γράψετε σήμερα:

```
use v5.44;
my $path = '/tmp/tcl_inplace.txt';      # holds: red green blue

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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
# grEEn
# bluE
```

Η παγίδα: η open επιστρέφει ψευδές σε αποτυχία αντί να εγείρει σφάλμα (η Tcl εκτοξεύει ένα σφάλμα που πιάνετε με catch), οπότε το ιδίωμα or die "...: $! " είναι υποχρεωτικό - το $! είναι η συμβολοσειρά σφάλματος του συστήματος. Για σωληνώσεις, η Perl ανοίγει έναν περιγραφέα σωλήνωσης απευθείας: το open my $fh, '-|', 'ls', '-l' διαβάζει την έξοδο μιας εντολής και το open my $fh, '|-', 'sort' γράφει στην είσοδο μιας εντολής, η δομημένη μορφή του open "|command" της Tcl.

One-liners και διακόπτες

Η Tcl δεν είναι γλώσσα one-liner, οπότε αυτή η ενότητα είναι σύντομη - αλλά η Perl είναι, και μόλις αφήσετε την Tcl πίσω αξίζει να ξέρετε ότι το αντανακλαστικό υπάρχει. Η οικογένεια διακοπτών: το -e παρέχει το πρόγραμμα, το -n το τυλίγει σε έναν βρόχο χωρίς αυτόματη εκτύπωση, το -p το τυλίγει σε έναν βρόχο με αυτόματη εκτύπωση, το -i επεξεργάζεται επιτόπια, και τα -a/-F αυτοδιαχωρίζουν κάθε γραμμή σε @F. Τα μπλοκ BEGIN { } και END { } τρέχουν πριν και μετά τον βρόχο.

```
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
```

Αυτά εμφανίζονται χωρίς έξοδο επειδή οι διακόπτες έμμεσου βρόχου δεν αναγνωρίζονται ακόμη από αυτό το build του pperl (δείτε τη σημείωση στην αρχή). Καθένας αντιστοιχίζεται σε μια επαληθευμένη μορφή ρητού βρόχου: το perl -ne 'CODE' είναι while (<$fh>) { CODE }, και το perl -pe 'CODE' είναι το ίδιο με μια τελική print. Η εκτελέσιμη μορφή «μέτρησε γραμμές που ταιριάζουν»:

```
use v5.44;
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
```

Η παγίδα: μέχρι να προστεθούν οι διακόπτες στο `pperl`, γράψτε τον ρητό βρόχο (τρέχει σήμερα και είναι ούτως ή άλλως αυτό στο οποίο αναπτύσσονται οι διακόπτες), και συμβουλευτείτε τον *οδηγό One-Liners* για τον πλήρη κατάλογο διακοπτών και συνταγές. Η Perl έχει επίσης πλήρεις υπορουτίνες, αναφορές, αρθρώματα και OO αν το σενάριό σας ξεπεράσει μια εργασία συγκόλλησης: τα `namespace eval` και οι κλάσεις `[incr Tcl]` της Tcl αντιστοιχίζονται σε πακέτα της Perl και στη σύγχρονη σύνταξη `class`· δείτε τον *οδηγό Αντικειμενοστραφούς Προγραμματισμού* για αυτήν την πλευρά της γλώσσας.

Παγίδες ερχόμενοι από την Tcl

Οι παγίδες, συγκεντρωμένες για μια τελική ματιά πριν γράψετε:

- **Δύο οικογένειες σύγκρισης.** Τα `==`/`</<=>` συγκρίνουν αριθμητικά· τα `eq`/`lt`/`cmp` συγκρίνουν ως συμβολοσειρές. Το `"10" == "10.0"` είναι αληθές, το `"10" eq "10.0"` είναι ψευδές. Επιλέξτε βάσει της πράξης, όχι της τιμής - το ένα σιωπηλό σφάλμα που προσκαλεί η κοσμοθεωρία με προτεραιότητα στη συμβολοσειρά.
- **Το sigil ακολουθεί το στοιχείο.** Ένα στοιχείο του `@items` είναι `$items[0]` (ένα βαθμωτό), όχι `@items[0]`· μία τιμή του `%a` είναι `$a{red}`. Το sigil του περιέκτη και το sigil του στοιχείου διαφέρουν.
- **Το `sort` ταξινομεί ως συμβολοσειρές εξ ορισμού,** όπως η σύγκριση με `eq`. Δώστε `{ $a <=> $b }` για αριθμητική σειρά.
- **Το `s///` επιστρέφει ένα πλήθος, όχι τη νέα συμβολοσειρά,** και επεξεργάζεται επιτόπια. Αντιγράψτε πρώτα (`(my $t = $s) =~ s/.../.../`) ή χρησιμοποιήστε τη σημαία `/r` για μια φρέσκια συμβολοσειρά.
- **Το `tr///` (το αντίστοιχο του `string map`) είναι ένα κυριολεκτικό σύνολο χαρακτήρων, όχι regex.** Το `tr/ab/AB/` αντιστοιχίζει μεμονωμένους χαρακτήρες. Για μοτίβα, χρησιμοποιήστε `s///`.
- **Τα `split ' '` και `split / /` διαφέρουν.** Η συμβολοσειρά κυριολεκτικού κενού διαχωρίζει σε διαδοχές κενών χαρακτήρων και απορρίπτει τα αρχικά κενά (ο tokeniser λέξεων)· το regex ενός κενού όχι. Χρησιμοποιήστε `split ' '` για ελεύθερης μορφής είσοδο.
- **Το `<$fh>` κρατά την τελική νέα γραμμή,** σε αντίθεση με το `gets` της Tcl. Καλέστε `chomp` για να την αφαιρέσετε.
- **Η `open` δεν εγείρει σφάλμα σε αποτυχία.** Χρησιμοποιήστε `open my $fh, '<', $path or die "...: $!"`. Το `$!` είναι η συμβολοσειρά σφάλματος του συστήματος· δεν υπάρχει `catch` να το τυλίξει.
- **Οι αναφορές αντικαθιστούν το `upvar`.** Για να μεταλλάξετε μια μεταβλητή του καλούντος, περάστε μια αναφορά (`\$x`, `\@arr`, `\%h`) σκόπιμα αντί να ψευδωνυμίζετε κατ” όνομα. Οι αναφορές είναι επίσης ο τρόπος που η Perl εμφωλεύει δεδομένα.
- **Ένα μπλοκ `{ ... }` είναι κώδικας, όχι αναβληθείσα συμβολοσειρά.** Η Perl αναλύει τα μπλοκ κατά τη μεταγλώττιση· δεν υπάρχει βήμα «το σώμα είναι συμβολοσειρά μέχρι να το τρέξω». Μόνο τα `''`/`""` φέρουν τη διάκριση υποκατάστασης `{}`/`""` της Tcl.
- **Οι διακόπτες έμμεσου βρόχου (`-n`/`-p`/`-a`/`-F`/`-i`/`-l`) δεν αναγνωρίζονται ακόμη από αυτό το build του `pperl`.** Μόνο το `-e` λειτουργεί σήμερα. Γράψτε τον

ρητό βρόχο `while (<$fh>) { ... }`, που είναι ακριβώς αυτό στο οποίο αναπτύσσονται οι διακόπτες, μέχρι να προστεθούν.

4.10 Ο οριστικός οδηγός για γραφικά με την 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 Ποιο κεφάλαιο θέλω;

Θέλω να ...	Ξεκινήστε εδώ
καταλάβω πώς λειτουργούν στην πραγματικότητα τα pixel, οι διαδρομές και το χρώμα	*foundations*
υλοποιήσω ο ίδιος μια γραμμή, έναν κύκλο, ένα γέμισμα ή ένα θόλωμα	*raster-algorithms*
σχεδιάσω πάνω, φορτώσω ή αποθηκεύσω μια εικόνα bitmap	*raster-imager*
χρησιμοποιήσω μια βάση κώδικα GD ή ImageMagick, ή τα ειδικά τους χαρακτηριστικά	*raster-gd-magick*
παραγάγω διανυσματικές διαδρομές ανεξάρτητες από την ανάλυση, ή έξοδο SVG	*vector-cairo-svg*
διαβάσω μεταδεδομένα εικόνας, ή σχεδιάσω ένα γράφημα	*metadata-and-charts*
σχεδιάσω σε ένα ζωντανό παράθυρο στο Linux	*gui-toolkits*
προσεγγίσω μια βιβλιοθήκη γραφικών σε C που δεν έχει σύνδεση με το CPAN	*ffi-graphics-primer*
δω πώς λειτουργεί η εγγενής σύνδεση `Cairo` που συνοδεύει την pperl	*ffi-cairo-showcase*
φτιάξω μια πραγματική εφαρμογή σχεδίασης	*capstone-drawing-app*
φτιάξω ένα πραγματικό 2D παιχνίδι	*capstone-2d-game*
βρω 3D, μηχανές παιχνιδιών ή μια πλήρη σύνδεση SDL2	*whats-next*

4.10.2 Οι τρεις άξονες αυτού του οδηγού

- **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 Μια σημείωση για τον χρόνο εκτέλεσης

These chapters target `pperl`, a faithful Rust reimplementation of Perl 5.44. The CPAN modules in the toolkit chapters are used exactly as they are on any Perl 5, and the FFI chapters use pperl's own `Peta::FFI`, which has no perl5 equivalent.

4.10.4 Γνωστά κενά

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 Δείτε επίσης

- *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 pperl provides C-backed modules natively.

Θεμέλια: pixel, διαδρομές και χρώμα

Πριν από οποιαδήποτε βιβλιοθήκη, τέσσερις ιδέες στηρίζουν όλο το θέμα: η διαφορά ανάμεσα σε ένα raster και ένα vector, το σύστημα συντεταγμένων μέσα στο οποίο ζει ένα σχέδιο, τι είναι στην πραγματικότητα ένα χρώμα από τη στιγμή που πρέπει να το αποθηκεύσετε, και πώς δύο χρώματα συνδυάζονται όταν το ένα είναι εν μέρει διαφανές. Κάθε κεφάλαιο μετά από αυτό στηρίζεται σε αυτά χωρίς να τα ξαναεξηγεί. Τίποτα από αυτά δεν είναι ειδικό για την Perl, αλλά όλα τα παραδείγματα είναι σε Perl, γιατί το ζητούμενο είναι να γίνουν οι ιδέες απτές σε κώδικα που μπορείτε να εκτελέσετε.

Raster έναντι vector

Μια εικόνα **raster** είναι ένα πλέγμα από χρωματιστά κελιά. Έχει σταθερό πλάτος και ύψος σε pixel, και κάθε pixel κρατά ένα χρώμα. Η μεγέθυνσή της πέρα από την ανάλυσή της δεν επινοεί τίποτα: τα κελιά απλώς μεγαλώνουν. Οι φωτογραφίες, τα στιγμιότυπα οθόνης και οτιδήποτε έχει συνεχή τόνο είναι raster. Τα `Imager`, GD και ImageMagick είναι εργαλεία raster.

Μια εικόνα **vector** είναι ένα σύνολο σχημάτων που περιγράφονται μαθηματικά: «μια γραμμή από εδώ ως εκεί», «ένας κύκλος αυτής της ακτίνας», «γέμισε αυτή τη διαδρομή». Δεν υπάρχει ανάλυση μέχρι να τη σχεδιάσετε, οπότε κλιμακώνεται καθαρά σε οποιοδήποτε μέγεθος. Οι γραμματοσειρές, τα λογότυπα και τα διαγράμματα είναι vector. Τα `Cairo` και SVG είναι εργαλεία vector.

Τα δύο συναντιούνται στη **rasterization** (μετατροπή σε ψηφίδες): τη μετατροπή μιας περιγραφής vector σε pixel. Μια βιβλιοθήκη vector πρέπει και πάλι να αποφασίσει ποια pixel αγγίζει μια διαγώνια γραμμή, και αυτή η απόφαση είναι *ο πρώτος αλγόριθμος στο επόμενο κεφάλαιο*. Οτιδήποτε σχεδιάζετε καταλήγει ως pixel σε μια οθόνη· το μόνο ερώτημα είναι αν δεσμεύεστε σε μια ανάλυση νωρίς (raster) ή αργά (vector).

Ένα raster στην Perl είναι, στην απλούστερη μορφή του, ένας πίνακας από γραμμές:

```
my $w = 4;
my $h = 3;
# each pixel is [r, g, b], each channel 0..255
my @img = map { [ map { [0, 0, 0] } 1 .. $w ] } 1 .. $h;

$img[1][2] = [255, 0, 0];   # row 1, column 2, red
```

Αυτή η εικόνα του ένθετου πίνακα είναι όλο κι όλο ένα raster. Οι πραγματικές βιβλιοθήκες αποθηκεύουν το ίδιο πλέγμα σε έναν πακεταρισμένο ενταμιευτή C για ταχύτητα, αλλά το νοητικό μοντέλο δεν αλλάζει: ένα pixel διευθυνσιοδοτείται από μια ακέραια γραμμή και στήλη, και κρατά ένα χρώμα.

Το σύστημα συντεταγμένων

Οι συντεταγμένες των γραφικών δεν είναι οι συντεταγμένες του μαθήματος των μαθηματικών. Η αρχή `(0, 0)` είναι η **επάνω αριστερή** γωνία, το x αυξάνεται προς τα δεξιά, και το y αυξάνεται **προς τα κάτω**. Αυτό παγιδεύει τον καθένα μία φορά: ένα μεγαλύτερο y είναι πιο χαμηλά στην οθόνη, όχι πιο ψηλά.

```
# (0,0) top-left            x ->
#   +-----------------
#   | (0,0)      (w-1, 0)
# y |
# | |
# v | (0, h-1)   (w-1, h-1)
```

Ένα pixel είναι ένα μικρό τετράγωνο, και υπάρχει μια διαρκής αμφισημία για το αν η συντεταγμένη `(3, 5)` ονομάζει τη γωνία του τετραγώνου ή το κέντρο του. Οι βιβλιοθήκες raster αντιμετωπίζουν συνήθως τις ακέραιες συντεταγμένες ως κέντρα pixel· οι βιβλιοθήκες vector τις αντιμετωπίζουν ως τις γραμμές του πλέγματος ανάμεσα στα pixel, γι” αυτό και μια κατακόρυφη γραμμή πλάτους μίας μονάδας στο `x = 3` μπορεί να βγει να πατάει σε δύο στήλες pixel με τη μισή ένταση. Όταν μια λεπτή γραμμή ή η ακμή ενός ορθογωνίου φαίνεται θολή ή μετατοπισμένη κατά ένα pixel, αυτός είναι σχεδόν πάντα ο λόγος, και η διόρθωση είναι μια μετατόπιση μισού pixel. Τα κεφάλαια που το συναντούν το αναφέρουν.

Τα μεγέθη και οι θέσεις είναι ακέραιοι στον χώρο raster και πραγματικοί αριθμοί στον χώρο vector. Αυτή είναι η πρακτική διαφορά που νιώθετε: στο `Imager` ορίζετε το pixel

`(120, 47)`· στο `Cairo` μπορείτε να μετακινηθείτε στο `(120.5, 47.25)` και ο rasterizer υπολογίζει την κάλυψη.

Χρώμα: τι κρατά ένα pixel

Ένα χρώμα σε μια οθόνη είναι τρεις αριθμοί: πόσο **κόκκινο**, **πράσινο** και **μπλε** φως να εκπεμφθεί. Προσθέστε και τα τρία με πλήρη ένταση και παίρνετε λευκό· αφήστε τα όλα σβηστά και παίρνετε μαύρο. Αυτό είναι το μοντέλο **RGB**, και είναι προσθετικό γιατί προσθέτετε φως, δεν αναμειγνύετε μπογιά.

Κάθε κανάλι αποθηκεύεται σχεδόν πάντα σε 8 bit, οπότε `0 .. 255` ανά κανάλι, και ένα χρώμα είναι τρία bytes. Αυτή είναι η δεκαεξαδική σημειογραφία #RRGGBB από τον ιστό: το `#ff0000` είναι `(255, 0, 0)`, καθαρό κόκκινο.

```
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
```

Το RGB είναι ο τρόπος με τον οποίο αποθηκεύονται τα pixel, αλλά είναι ένα απαίσιο μοντέλο για έναν άνθρωπο που επιλέγει ένα χρώμα. Το «κάνε το λίγο πιο πορτοκαλί αλλά κράτησέ το στην ίδια φωτεινότητα» δεν είναι μια προφανής κίνηση στο RGB. Το **HSV** (hue, saturation, value) είναι το μοντέλο γι” αυτό: το **hue** (απόχρωση) είναι η θέση στον χρωματικό τροχό σε μοίρες `0 .. 360` (0 κόκκινο, 120 πράσινο, 240 μπλε), το **saturation** (κορεσμός) είναι πόσο ζωηρό έναντι γκρι `0 .. 1`, και το **value** (τιμή) είναι πόσο φωτεινό `0 .. 1`. Η περιστροφή της απόχρωσης κρατώντας σταθερά τον κορεσμό και την τιμή διατρέχει το ουράνιο τόξο με σταθερή ζωηρότητα και φωτεινότητα.

Κάθε επιλογέας χρώματος και εργαλείο διαβάθμισης δουλεύει σε HSV ή σε έναν στενό συγγενή του, και έπειτα μετατρέπει σε RGB για αποθήκευση. Αξίζει να έχετε τη μετατροπή σε κοινή θέα, γιατί θα την ξαναϋλοποιήσετε περισσότερες από μία φορές:

```
# 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,
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
 ↪300;
# red, yellow, green, cyan, blue, magenta
```

Υπάρχουν και άλλοι χώροι που θα συναντήσετε: η κλίμακα του γκρι (ένα κανάλι, μια φωτεινότητα), το CMYK (αφαιρετικό, για εκτύπωση) και αντιληπτικά ομοιόμορφοι χώροι όπως το CIELAB όπου ίσα αριθμητικά βήματα φαίνονται ως ίσα αντιληπτικά βήματα. Για σχεδίαση στην οθόνη σε δύο διαστάσεις, το RGB για αποθήκευση και το HSV για επιλογή καλύπτουν σχεδόν τα πάντα.

Άλφα και σύνθεση

Ένα τέταρτο κανάλι, το **alpha** (άλφα), καταγράφει την αδιαφάνεια: `0` πλήρως διαφανές, 255 (ή `1.0`) πλήρως αδιαφανές. Ένα pixel με άλφα είναι **RGBA**. Το άλφα είναι αυτό που σας επιτρέπει να σχεδιάσετε μια εικόνα πάνω σε μια άλλη και να αφήσετε τα διαφανή μέρη της επάνω να αφήνουν να φανεί ό,τι είναι από κάτω.

Ο συνδυασμός ενός pixel προσκηνίου πάνω σε ένα pixel φόντου είναι η **σύνθεση** (compositing), και ο τυπικός κανόνας είναι το **source-over**. Με το άλφα ως κλάσμα `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 = $fg->[3] / 255;                    # source alpha
    my @out = map {
        int($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
```

Το κόκκινο με μισή αδιαφάνεια πάνω στο λευκό δίνει ροζ, ακριβώς όπως θα περιμένατε να φαίνεται μια μπογιά αραιωμένη με νερό. Αυτός ο ένας τύπος, εφαρμοσμένος ανά pixel, είναι ο τρόπος με τον οποίο κάθε στρώμα, πινελιά και sprite με απαλές ακμές προσγειώνεται στον καμβά από κάτω του. Όταν στοιβάζετε πολλά ημιδιαφανή στρώματα, τον εφαρμόζετε επανειλημμένα, από κάτω προς τα πάνω.

Η λεπτομέρεια που δαγκώνει αργότερα είναι το **premultiplied alpha** (προπολλαπλασιασμένο άλφα): η αποθήκευση κάθε καναλιού χρώματος ήδη πολλαπλασιασμένου με το άλφα του. Κάνει την επαναλαμβανόμενη σύνθεση φθηνότερη και το φιλτράρισμα σωστό στις ακμές, και είναι ο λόγος για τον οποίο τα ακατέργαστα bytes των pixel μιας βιβλιοθήκης ορισμένες φορές δεν είναι το RGB που περιμένετε. Το `Cairo` χρησιμοποιεί εσωτερικά προπολλαπλασιασμένο άλφα· το *κεφάλαιο για το Cairo* σημειώνει πού φαίνεται.

Η μία ιδέα του 3D, ονομασμένη και αναβεβλημένη

Θα ακούσετε για τον **Z-buffer** τη στιγμή που προκύπτει το 3D, και αξίζει μία παράγραφος εδώ ώστε ο όρος να μην είναι μυστήριο. Στο 2D, το «τι είναι από πάνω» αποφασίζεται από τη σειρά σχεδίασης: το τελευταίο πράγμα που σχεδιάστηκε κερδίζει, που είναι ακριβώς η σύνθεση source-over παραπάνω. Στο 3D, τα αντικείμενα έχουν βάθος (μια συντεταγμένη z), και το πράγμα που είναι πλησιέστερα στον θεατή θα έπρεπε να κερδίζει ανεξάρτητα από τη σειρά σχεδίασης. Ένας **Z-buffer** είναι ένα δεύτερο πλέγμα, ίδιου μεγέθους με την εικόνα, που κρατά το βάθος της πλησιέστερης επιφάνειας που έχει φανεί ως τώρα σε κάθε pixel· πριν σχεδιάσετε ένα pixel ελέγχετε αν η νέα επιφάνεια είναι πλησιέστερα από ό,τι έχει καταγραφεί, και το παραλείπετε αν όχι.

Αυτή είναι όλη η έννοια, και ως εκεί φτάνει αυτός ο οδηγός 2D με αυτήν. Η αποθήκευση βάθους, η προβολή και η υπόλοιπη τρίτη διάσταση ανήκουν στην *προγραμματισμένη συνέχεια*. Σε δύο διαστάσεις, η σειρά σχεδίασης είναι ο δικός σας ενταμιευτής βάθους.

Πού πηγαίνει αυτό στη συνέχεια

Τώρα έχετε το λεξιλόγιο που μιλά ο υπόλοιπος οδηγός: raster και vector, συντεταγμένες με αρχή επάνω αριστερά, RGB και HSV, άλφα και σύνθεση source-over. Το *επόμενο κεφάλαιο* ξοδεύει αυτό το λεξιλόγιο στους κλασικούς αλγόριθμους σχεδίασης, υλοποιημένους από το μηδέν, ώστε όταν μια βιβλιοθήκη σχεδιάζει μια γραμμή ή γεμίζει ένα πολύγωνο να ξέρετε ακριβώς τι κάνει για λογαριασμό σας.

Πρωτογενή σχεδίασης από το μηδέν

Κάθε βιβλιοθήκη γραφικών σας δίνει τα `line`, `circle`, `fill` και `blur`. Αυτό το κεφάλαιο τα υλοποιεί σε απλή Perl ώστε, όταν μια βιβλιοθήκη τα σχεδιάζει για εσάς, τίποτα να μην είναι μαγικό. Αυτοί είναι οι κλασικοί αλγόριθμοι, και είναι κλασικοί γιατί είναι οι σωστές απαντήσεις: γρήγοροι, ακέραιοι όπου μπορούν, και σωστοί στις άβολες περιπτώσεις. Δεν θα αποστείλετε αυτές τις υλοποιήσεις (μια βιβλιοθήκη C θα είναι πάντα ταχύτερη), αλλά η γνώση τους σας λέει γιατί μια βιβλιοθήκη συμπεριφέρεται όπως συμπεριφέρεται, και σας δίνει τα εργαλεία για τη μία περίπτωση που η βιβλιοθήκη δεν προέβλεψε.

Παντού, ο καμβάς είναι το raster ένθετου πίνακα από το *κεφάλαιο των θεμελίων*: ένα πλέγμα από pixel `[r, g, b]` που διευθυνσιοδοτούνται με ακέραιο `(x, y)`, με αρχή επάνω αριστερά.

```
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;
}
```

Γραμμές: Bresenham

Μια γραμμή από `(x0, y0)` ως `(x1, y1)` πρέπει να επιλέξει, για κάθε στήλη που διασχίζει, ποια γραμμή pixel να ανάψει. Η αφελής προσέγγιση υπολογίζει το `y = m*x + b` με κινητή υποδιαστολή ανά βήμα· ο **αλγόριθμος του Bresenham** το κάνει μόνο με ακέραια πρόσθεση, παρακολουθώντας έναν όρο σφάλματος που λέει πόσο μακριά έχει παρεκκλίνει η ιδανική γραμμή από τα pixel που έχουν επιλεγεί ως τώρα.

Η ουσία είναι ότι σε κάθε βήμα είτε μένετε στην ίδια γραμμή του εγκάρσιου άξονα είτε μετακινείστε κατά μία, και ένα τρέχον ακέραιο σφάλμα αποφασίζει ποιο. Η έκδοση παρακάτω χειρίζεται κάθε κλίση και κατεύθυνση σε έναν βρόχο, δουλεύοντας με απόλυτα deltas και πρόσημα βήματος:

```
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
    }
}
```

Καθόλου πολλαπλασιασμός, καθόλου διαίρεση, καθόλου κινητή υποδιαστολή στον βρόχο. Γι” αυτό ο Bresenham επιβίωσε από το 1962: είναι η σχεδίαση γραμμών που θα κατασκευάζατε σε υλικό. Κάθε λεπτή, aliased γραμμή μιας βιβλιοθήκης είναι αυτός ή ένας άμεσος απόγονός του.

Κύκλοι: η μέθοδος του μέσου σημείου

Ένας κύκλος έχει οκταπλή συμμετρία: υπολογίστε ένα ογδόημα και κατοπτρίστε το στα άλλα επτά. Ο **αλγόριθμος κύκλου μέσου σημείου** διατρέχει ένα ογδόημα με την ίδια ιδέα ακέραιου σφάλματος όπως ο Bresenham, αποφασίζοντας σε κάθε βήμα αν θα μετακινηθεί ευθεία ή διαγώνια, ελέγχοντας μια μεταβλητή απόφασης ως προς το μηδέν.

```
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);
        }
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
        $y++;
        if ($err < 0) { $err += 2 * $y + 1; }
        else          { $x--; $err += 2 * ($y - $x) + 1; }
    }
}
```

Υπολογίστε ένα όγδοο, σχεδιάστε οκτώ. Το ίδιο κόλπο συμμετρίας, με προσαρμοσμένο τον έλεγχο σφάλματος, δίνει ελλείψεις και τόξα.

Γέμισμα πολυγώνων: η μέθοδος scanline

Η σχεδίαση του περιγράμματος ενός σχήματος είναι ένα πρόβλημα· το γέμισμα του εσωτερικού του είναι ένα άλλο. Το **γέμισμα scanline** σαρώνει μια οριζόντια γραμμή προς τα κάτω στην εικόνα, και για κάθε scanline βρίσκει πού διασχίζει τις ακμές του πολυγώνου. Αυτές οι διασταυρώσεις έρχονται σε ζεύγη, και το εσωτερικό είναι ανάμεσα σε κάθε ζεύγος: ο **κανόνας άρτιου-περιττού**. Ταξινομήστε τις διασταυρώσεις x, και έπειτα γεμίστε ανάμεσα στην πρώτη και τη δεύτερη, την τρίτη και την τέταρτη, και ούτω καθεξής.

```
# 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) % @$poly];
            my ($ay, $by) = ($a->[1], $b->[1]);
            # does this edge straddle the scanline?
            next if ($ay <= $y) == ($by <= $y);
            # 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);
            }
        }
    }
}
```

Ο έλεγχος `($ay <= $y) == ($by <= $y)` είναι ο έλεγχος διασταύρωσης γραμμένος ώστε να μετρά κάθε κορυφή ακριβώς μία φορά, οπότε μια scanline που περνά από μια κοινή κορυφή δεν διπλομετρά και δεν διαρρέει το γέμισμα. Αυτή η ημιανοιχτή σύμβαση είναι η λεπτομέρεια που ξεχωρίζει ένα σωστό γέμισμα από ένα με περιστασιακές αδέσποτες γραμμές, και είναι ο λόγος για τον οποίο οι βιβλιοθήκες και αυτός ο κώδικας συμφωνούν στο πού «ανήκει» μια ακμή.

Εξομάλυνση: κάλυψη αντί για αναμμένο-ή-σβηστό

Η γραμμή και ο κύκλος παραπάνω είναι **aliased**: κάθε pixel είναι πλήρως αναμμένο ή πλήρως σβηστό, και μια διαγώνια ακμή μοιάζει με σκάλα. Η **εξομάλυνση** το μαλακώνει ορίζοντας την ένταση ενός pixel ίση με το πόσο από αυτό καλύπτει στην πραγματικότητα το σχήμα. Ένα pixel που η ακμή διασχίζει με κάλυψη 40 τοις εκατό παίρνει 40 τοις εκατό του χρώματος του σχήματος αναμεμειγμένο με 60 τοις εκατό του φόντου, που είναι ακριβώς η *σύνθεση source-over* από το προηγούμενο κεφάλαιο με άλφα ίσο με την κάλυψη.

Ο φθηνότερος τρόπος να προσεγγίσετε την κάλυψη είναι η **υπερδειγματοληψία**: σχεδιάστε σε πολλαπλάσια της ανάλυσης με τον aliased αλγόριθμο, και έπειτα υπολογίστε τον μέσο όρο κάθε μπλοκ pixel υψηλής ανάλυσης σε ένα. Ο μέσος όρος ενός μπλοκ 2x2 όπου τρία υπο-pixel είναι αναμμένα δίνει κάλυψη 75 τοις εκατό δωρεάν:

```
# 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 ];
        }
    }
    return $out;
}
```

Οι πραγματικές βιβλιοθήκες υπολογίζουν την κάλυψη αναλυτικά αντί με υπερδειγματοληψία ωμής βίας, αλλά το αποτέλεσμα είναι πανομοιότυπο: η σκάλα γίνεται μια ομαλή ράμπα εντάσεων. Όταν το `Cairo` σας δίνει μια ευκρινή εξομαλυμένη καμπύλη και το `Imager` προσφέρει μια σημαία `aa` στα πρωτογενή του, αυτό είναι που κάνουν.

Τεσελίωση: διάσπαση σχημάτων σε τρίγωνα

Το υλικό και πολλοί αλγόριθμοι στην πραγματικότητα ξέρουν να γεμίζουν μόνο **τρίγωνα**· οτιδήποτε πιο σύνθετο διασπάται πρώτα σε τρίγωνα, μια διαδικασία που ονομάζεται **τεσελίωση** ή **τριγωνοποίηση**. Ένα κυρτό πολύγωνο τριγωνοποιείται τετριμμένα με βεντάλια από μία κορυφή:

```
# 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;
}
```

Τα κοίλα πολύγωνα χρειάζονται έναν πραγματικό αλγόριθμο (το ear clipping είναι ο τυπικός: βρίσκετε επανειλημμένα ένα τρίγωνο που προεξέχει χωρίς καμία άλλη κορυφή μέσα του, το ψαλιδίζετε, επαναλαμβάνετε). Η κυρτή βεντάλια αρκεί για να δείτε την ιδέα, και είναι η ιδέα που μεταφέρεται κατευθείαν στο 3D, όπου κάθε επιφάνεια είναι τελικά ένα πλέγμα από τρίγωνα. Σε δύο διαστάσεις συναντάτε την τεσελίωση όποτε γεμίζετε μια σύνθετη διαδρομή vector· στην *προγραμματισμένη συνέχεια 3D* είναι το θεμέλιο των πάντων.

Συνέλιξη: θόλωμα, όξυνση και ακμές

Μια ολόκληρη οικογένεια εφέ εικόνας προέρχεται από μία πράξη: τη **συνέλιξη**, το ολίσθημα ενός μικρού πλέγματος βαρών (ενός **πυρήνα**) πάνω στην εικόνα και την αντικατάσταση κάθε pixel με το σταθμισμένο άθροισμα των γειτόνων του. Ένας πυρήνας με όλα ίσα βάρη υπολογίζει μέσο όρο, που θολώνει. Ένας πυρήνας που αφαιρεί τους γείτονες και προσθέτει ένα μεγάλο κέντρο οξύνει.

```
# 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) {
                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);
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
                        $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
```

Το box blur διαιρεί με το 9 για να κρατήσει σταθερή τη φωτεινότητα· ο πυρήνας όξυνσης αθροίζεται στο 1 οπότε δεν χρειάζεται διαιρέτη. Βάλτε στη θέση του έναν πυρήνα που αντιδρά στη μεταβολή της έντασης και παίρνετε ανίχνευση ακμών, που είναι εκεί όπου η επεξεργασία εικόνας σβήνει προς την υπολογιστική όραση. Αυτό το όριο είναι σκόπιμο: αυτός ο οδηγός χρησιμοποιεί τη συνέλιξη ως εργαλείο **σχεδίασης** (θόλωμα μιας σκιάς, όξυνση ενός υποκλιμακωμένου sprite) και σταματά εκεί. Η ανάλυση, η ανίχνευση χαρακτηριστικών και η αναγνώριση είναι ένα διαφορετικό θέμα, εκτός πεδίου εδώ.

Τι κάνουν οι βιβλιοθήκες για εσάς

Όλα τα παραπάνω είναι αυτό που μια βιβλιοθήκη γραφικών υλοποιεί σε βελτιστοποιημένη C, συνήθως με SIMD και προσεκτικό χειρισμό ακμών που δεν θα θέλατε να γράψετε στο χέρι. Η γνώση των αλγορίθμων σημαίνει ότι μπορείτε να διαβάσετε την τεκμηρίωση μιας βιβλιοθήκης και να ξέρετε τι σημαίνουν τα «anti-aliased», «κανόνας γεμίσματος άρτιου-περιττού», «πίνακας συνέλιξης» και «τριγωνοποιημένη διαδρομή», και μπορείτε να κατέβετε στον δικό σας βρόχο pixel για το σπάνιο εφέ που δεν αποστέλλει καμία βιβλιοθήκη. Το *επόμενο κεφάλαιο* πιάνει το `Imager`, όπου αυτά τα πρωτογενή απέχουν μία κλήση μεθόδου και τρέχουν με ταχύτητα C.

Σχεδίαση raster με το Imager

Το `Imager` είναι το βαρύ όχημα του raster: δημιουργήστε μια εικόνα, σχεδιάστε πάνω της, φιλτράρετέ την, αποθηκεύστε την. Συντηρείται ενεργά, συνδέει τα δικά του αντίγραφα των βιβλιοθηκών μορφών ώστε PNG, JPEG, TIFF, GIF και WEBP να λειτουργούν όλα εκ των έξω, και τα πρωτογενή σχεδίασής του είναι αυτά από το *κεφάλαιο των αλγορίθμων* που τρέχουν με ταχύτητα C. Αν ξεκινάτε ένα έργο raster σε Perl σήμερα και δεν έχετε λόγο να προτιμήσετε κάτι άλλο, ξεκινήστε εδώ.

Κάθε μέθοδος σχεδίασης επιστρέφει ψευδές σε αποτυχία και ορίζει το `errstr`, οπότε το ιδίωμα είναι να ελέγχετε και να αναφέρετε:

```
use Imager;

my $img = Imager->new(xsize => 400, ysize => 300, channels => 4)
    or die Imager->errstr;
```

Το `channels => 4` ζητά RGBA (red, green, blue, alpha)· χρησιμοποιήστε 3 για αδιαφανές RGB. Το κανάλι άλφα είναι αυτό που επιτρέπει στη *σύνθεση* από το κεφάλαιο των θεμελίων να λειτουργεί όταν σχεδιάζετε μια εικόνα πάνω σε μια άλλη.

Χρώματα

Οπουδήποτε μια μέθοδος παίρνει `color =>`, μπορείτε να περάσετε ένα όνομα, μια δεκαεξαδική συμβολοσειρά, ένα αντικείμενο `Imager::Color`, ή μια απλή αναφορά πίνακα:

```
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
```

Η μορφή της αναφοράς πίνακα είναι η λιγότερο τυπική για ένα εφάπαξ χρώμα και είναι αυτή που χρησιμοποιούν τα περισσότερα από τα παρακάτω παραδείγματα.

Τα πρωτογενή

Σημειώστε προσεκτικά τα ονόματα των παραμέτρων, γιατί δεν έχουν όλα το ίδιο σχήμα. Τα κουτιά χρησιμοποιούν συντεταγμένες γωνιών `xmin`/`ymin`/`xmax`/`ymax`· οι κύκλοι και τα τόξα χρησιμοποιούν ένα κέντρο `x`/`y` με μια ακτίνα `r`· τα τόξα προσθέτουν μοίρες αρχής και τέλους `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);
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
# flood fill from a seed point out to a border
$img->flood_fill(x => 5, y => 5, color => [245, 245, 245]);
```

Δύο παγίδες που αξίζει να αναφερθούν μία φορά. Το box γεμίζει μόνο όταν περνάτε filled => 1 (ή ένα μοτίβο fill =>)· χωρίς αυτό παίρνετε ένα περίγραμμα. Το polyline σχεδιάζει μια ανοιχτή διαδρομή και δεν δέχεται fill, σε αντίθεση με το polygon που κλείνει και μπορεί να γεμίσει. Όταν ένα σχήμα βγαίνει ως περίγραμμα ενώ το περιμένατε γεμάτο, ή ένα γέμισμα σιωπηλά δεν κάνει τίποτα, μία από αυτές τις δύο είναι ο λόγος.

Φόρτωση, κλιμάκωση και αποθήκευση

Η ανάγνωση και η εγγραφή συμπεραίνουν τη μορφή από την επέκταση του ονόματος αρχείου, ή μπορείτε να την επιβάλετε με 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;
```

Το scale επιστρέφει μια νέα εικόνα· δεν τροποποιεί την αρχική. Η μορφή type => 'min' κλιμακώνει ώστε η εικόνα να χωρά μέσα στο δοσμένο κουτί διατηρώντας τον λόγο διαστάσεων· το 'max' γεμίζει το κουτί (περικόπτοντας την υπερχείλιση όταν έπειτα περικόψετε)· το 'nonprop' αγνοεί τον λόγο διαστάσεων και τεντώνει ακριβώς στα pixel που ονομάσατε.

Φίλτρα

Το filter εφαρμόζει ένα εφέ επιτόπου, ονομασμένο μέσω type =>. Η *συνέλιξη* από το κεφάλαιο των αλγορίθμων είναι εδώ ως ενσωματωμένο θόλωμα Gauss και unsharp masking, συν ένα γενικό conv για τον δικό σας πυρήνα:

```
# 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);
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
$img->filter(type => 'mosaic', size => 8);
$img->filter(type => 'hardinvert');
```

Το πλήρες σύνολο περιλαμβάνει τα gaussian, gaussian2 (ξεχωριστές αποκλίσεις x και y), conv, unsharpmask, contrast, mosaic, noise, hardinvert, autolevels, postlevels, και τους διαδικαστικούς γεννήτορες υφής fountain, gradgen, radnoise και turbnoise. Καταφύγετε στο ονομασμένο φίλτρο πριν γράψετε τον δικό σας βρόχο pixel· κατεβείτε σε μια λίστα συντελεστών conv ή σε χειροκίνητη πρόσβαση μόνο όταν τίποτα ενσωματωμένο δεν ταιριάζει.

Ένα πλήρες πρόγραμμα

Συνθέτοντάς τα: μια μικρή αφίσα με ένα φόντο, μερικά σχήματα, ένα πέρασμα θολώματος και μια αποθηκευμένη μικρογραφία.

```
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;
```

Πότε το Imager είναι το λάθος εργαλείο

Το Imager είναι raster, οπότε έχει τον περιορισμό του raster: έχει σταθερή ανάλυση, και η κλιμάκωση πέρα από αυτήν υποβαθμίζει. Για έξοδο ανεξάρτητη από την ανάλυση (ένα λογότυπο που πρέπει να εκτυπώνεται ευκρινώς σε οποιοδήποτε μέγεθος, ένα διάγραμμα προορισμένο για τον ιστό ως markup) θέλετε ένα εργαλείο vector, που καλύπτεται στο *Cairo και SVG*. Για μια υπάρχουσα βάση κώδικα χτισμένη πάνω σε GD ή ImageMagick, ή για τα ειδικά τους δυνατά σημεία, δείτε το *επόμενο κεφάλαιο*. Και όταν η σχεδίαση πρέπει να εμφανιστεί σε ένα ζωντανό παράθυρο αντί για ένα αρχείο, αυτό είναι το *κεφάλαιο των εργαλειοθηκών GUI*. Για οτιδήποτε είναι «φτιάξε ή τροποποίησε ένα bitmap στον δίσκο», το Imager είναι η απάντηση.

GD και ImageMagick όταν τα χρειάζεστε

Το `Imager` είναι η προεπιλογή, αλλά αξίζει να γνωρίζετε άλλες δύο βιβλιοθήκες raster. Το GD είναι παλαιότερο, μικρότερο και πανταχού παρόν: ένα μεγάλο μέρος της υπάρχουσας Perl σχεδιάζει με αυτό, και το μοντέλο ευρετηρίου χρωμάτων του αξίζει να το κατανοήσετε ακόμη και αν δεν το επιλέξετε ποτέ για κάτι νέο. Το ImageMagick, μέσω του `Image::Magick`, είναι το βαρέων βαρών: η ευρύτερη υποστήριξη μορφών και ο βαθύτερος σάκος μετασχηματισμών, με κόστος μια μεγάλη εξάρτηση. Αυτό το κεφάλαιο δείχνει το καθένα, και λέει πότε νικά το `Imager`.

GD

Το GD τυλίγει το libgd. Η καθοριστική ιδιοτροπία του είναι το **ευρετήριο χρωμάτων**: σε μια κλασική εικόνα (παλέτας) δεσμεύετε κάθε χρώμα μία φορά και έπειτα σχεδιάζετε με τον ακέραιο δείκτη που επιστρέφει, όχι με το ίδιο το χρώμα. Αυτό είναι ένα κατάλοιπο των εικόνων GIF των 256 χρωμάτων, και εξακολουθεί να φαίνεται στο API παρότι οι εικόνες truecolor είναι πλέον ο κανόνας.

```
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);
```

Τα πρωτογενή είναι θεσιακά, και δύο από αυτά διαφέρουν από αυτό που ίσως περιμένετε. Τα `arc` και `ellipse` παίρνουν ένα κέντρο συν **πλάτος και ύψος** (τις πλήρεις εκτάσεις, όχι μια ακτίνα), και γωνίες σε μοίρες:

```
# 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
```

Τα πολύγωνα σχεδιάζονται από ένα αντικείμενο `GD::Polygon`, όχι από ακατέργαστες συντεταγμένες:

```
my $poly = GD::Polygon->new;
$poly->addPt(50, 10);
$poly->addPt(150, 10);
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
$poly->addPt(100, 90);
$im->polygon($poly, $blue);
$im->filledPolygon($poly, $red);
```

Η έξοδος είναι μία μέθοδος ανά μορφή που επιστρέφει τα κωδικοποιημένα bytes, οπότε τα εκτυπώνετε σε ένα filehandle μόνοι σας:

```
open my $fh, '>', 'out.png' or die $!;
binmode $fh;
print $fh $im->png;            # or ->jpeg($quality), ->gif
close $fh;
```

Πότε νικά το GD: συντηρείτε κώδικα που ήδη το χρησιμοποιεί (ένα μεγάλο μέρος της παλαιότερης Perl το κάνει), θέλετε τη μικρότερη εξάρτηση που εξακολουθεί να σχεδιάζει, ή χρειάζεστε συγκεκριμένα η συμπεριφορά του libgd να ταιριάζει με ένα άλλο εργαλείο σε μια σωλήνωση που επίσης χρησιμοποιεί libgd. Για νέα δουλειά σχεδίασης χωρίς τέτοιον περιορισμό, το API ονομασμένων χρωμάτων και ορισμάτων hash του `Imager` είναι πιο φιλικό και το σύνολο φίλτρων του πλουσιότερο.

ImageMagick (Image::Magick)

Το `Image::Magick`, που ονομάζεται και PerlMagick, αποστέλλεται μαζί με το ίδιο το ImageMagick και εκθέτει την πλήρη μηχανή του. Τα δυνατά του σημεία είναι το **εύρος μορφών** (διαβάζει και γράφει μορφές που δεν αγγίζει τίποτα άλλο στο CPAN) και οι **μετασχηματισμοί** (εκατοντάδες πράξεις από `Resize` και `Blur` έως `Distort`, `Morphology` και κόλπα ειδικά για κάθε μορφή). Η αδυναμία του είναι η εξάρτηση: χρειάζεστε μια αντίστοιχη βιβλιοθήκη ImageMagick εγκατεστημένη, και είναι μεγάλη.

Το API είναι προσανατολισμένο στα ρήματα: δημιουργείτε ένα αντικείμενο, το διαβάζετε με `Read` ή το διαστασιολογείτε, και έπειτα καλείτε ονομασμένες πράξεις, καθεμία από τις οποίες επιστρέφει μια συμβολοσειρά σφάλματος που είναι κενή σε επιτυχία.

```
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');
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
my $err = $im->Write('out.png');
warn "$err" if $err;
```

Το πρωτογενές `circle` παίρνει το κέντρο ακολουθούμενο από οποιοδήποτε σημείο στην περίμετρο (οπότε το `100,100 100,40` είναι κέντρο `(100,100)` με ακτίνα 60), που είναι η σύμβαση του ImageMagick αντί για ακτίνα. Κάθε όρισμα γεωμετρίας είναι η μικρογλώσσα συμβολοσειρών του ImageMagick (`'200x200'`, `'100x100+10+10'`), οπότε αξίζει μια ματιά στην τεκμηρίωση γεωμετρίας του ίδιου του ImageMagick για τις λεπτομέρειες.

Πότε νικά το ImageMagick: χρειάζεστε μια μορφή που δεν διαβάζει τίποτα άλλο, κάνετε βαριά ομαδική μεταμόρφωση (δημιουργία μικρογραφιών χιλιάδων αρχείων, σωληνώσεις μετατροπής μορφών), ή θέλετε μία από τις πολλές πράξεις του που θα ήταν ένα ολόκληρο έργο να ξαναϋλοποιηθεί. Για διαδραστική σχεδίαση ή ένα στενό αποτύπωμα εξαρτήσεων, είναι περισσότερη μηχανή από όση χρειάζεστε, και το `Imager` ή το `GD` ταιριάζουν καλύτερα.

Επιλογή

Θέλετε ...	Καταφύγετε σε
Νέα σχεδίαση raster, φιλικό API, καλά φίλτρα	`Imager`
Μικρότερη εξάρτηση, ή μια υπάρχουσα βάση κώδικα GD	`GD`
Ευρύτερες μορφές, βαριές ομαδικές μεταμορφώσεις	`Image::Magick`

Και τα τρία είναι raster και μοιράζονται την οροφή του raster: σταθερή ανάλυση, υποβάθμιση κατά τη μεγέθυνση. Όταν χρειάζεστε έξοδο που κλιμακώνεται καθαρά, το *επόμενο κεφάλαιο* περνά στα διανυσματικά γραφικά με `Cairo` και SVG.

Διανυσματικά γραφικά: Cairo και SVG

Τα εργαλεία raster δεσμεύονται σε μια ανάλυση· τα εργαλεία vector περιγράφουν σχήματα και αποφασίζουν τα pixel αργότερα, ή ποτέ. Αυτό το κεφάλαιο καλύπτει τις δύο διαδρομές vector στην Perl. Το `Cairo` είναι μια πλήρης εξομαλυμένη μηχανή 2D που rasterize διαδρομές σε μια επιφάνεια pixel (ή σε PDF και SVG): σχεδιάζετε με συντεταγμένες πραγματικών αριθμών και υπολογίζει την κάλυψη. Το SVG παράγει markup SVG, μια κειμενική περιγραφή σχημάτων που ένας φυλλομετρητής ή αποδότης σχεδιάζει αργότερα σε όποιο μέγεθος του αρέσει. Χρησιμοποιήστε το `Cairo` όταν χρειάζεστε pixel τώρα με την ποιότητα ενός σωστού rasterizer· χρησιμοποιήστε το SVG όταν θέλετε έξοδο ανεξάρτητη από την ανάλυση ως αρχείο που καταναλώνουν άλλα εργαλεία.

Το `Cairo` επιστρέφει επίσης στο *κεφάλαιο 9*: η pperl αποστέλλει μια εγγενή επανυλοποίησή του, οπότε ο ίδιος κώδικας τρέχει χωρίς εγκατεστημένο το Cairo του CPAN, και εκείνο το κεφάλαιο δείχνει πώς λειτουργεί η εγγενής σύνδεση. Εδώ είναι το συνηθισμένο άρθρωμα του CPAN.

Cairo: διαδρομές και το μοντέλο σχεδίασης

Το μοντέλο του `Cairo` είναι μια **διαδρομή** χτισμένη από τμήματα, που έπειτα είτε **χαράσσεται** (σχεδιάζεται το περίγραμμα) είτε **γεμίζει** (βάφεται το εσωτερικό). Ορίζετε ένα χρώμα πηγής, χτίζετε μια διαδρομή με `move_to` και `line_to` και καμπύλες, και την επικυρώνετε. Τα χρώματα είναι floats στο `0 .. 1`, όχι `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;

# 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');
```

Τρία πράγματα να αφομοιώσετε. Οι συντεταγμένες είναι πραγματικοί αριθμοί, οπότε το `move_to(100.5, 47.25)` έχει νόημα και η *μετατόπιση μισού pixel* από το κεφάλαιο των θεμελίων είναι ο τρόπος με τον οποίο παίρνετε ευκρινείς λεπτές γραμμές. Οι γωνίες τόξου είναι σε **ακτίνια**, όχι σε μοίρες, οπότε ένας πλήρης κύκλος είναι `2 * pi`. Και το άλφα του `set_source_rgba` σας δίνει τη *σύνθεση* δωρεάν: γεμίστε ή χαράξτε με άλφα μικρότερο του 1 και το Cairo αναμειγνύει πάνω σε ό,τι υπάρχει ήδη εκεί.

Τα `stroke` και `fill` καταναλώνουν την τρέχουσα διαδρομή. Αν θέλετε να γεμίσετε και να χαράξετε το ίδιο σχήμα, χρησιμοποιήστε το `fill_preserve` (ή το `stroke_preserve`), που κρατά τη διαδρομή για μια δεύτερη πράξη:

```
$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
```

Προπολλαπλασιασμένο άλφα

Η επιφάνεια `argb32` αποθηκεύει pixel με *προπολλαπλασιασμένο άλφα*: κάθε κανάλι χρώματος είναι ήδη πολλαπλασιασμένο με το άλφα του. Αυτό είναι αόρατο όταν σχεδιάζετε μέσω του API του Cairo, αλλά έχει σημασία τη στιγμή που διαβάζετε ακατέργαστα bytes pixel από την επιφάνεια ή τροφοδοτείτε τον ενταμιευτή του Cairo σε μια άλλη βιβλιοθήκη, γιατί τα bytes δεν είναι το ευθύ RGBA που ίσως υποθέτετε. Όταν διασχίζετε αυτό το όριο, αποπολλαπλασιάστε πρώτα. Μέσα στην ίδια τη σχεδίαση του Cairo δεν προκύπτει ποτέ.

SVG: markup, όχι pixel

Το SVG είναι καθαρή Perl και κάνει κάτι διαφορετικό: χτίζει ένα έγγραφο XML που περιγράφει σχήματα, και παράγει κείμενο. Τίποτα δεν rasterize· ο καταναλωτής (ένας φυλλομετρητής, το Inkscape, ένας μετατροπέας PDF) το σχεδιάζει. Αυτό το κάνει ιδανικό για έξοδο που πρέπει να παραμείνει ευκρινής σε οποιαδήποτε μεγέθυνση, ή που άλλα εργαλεία θα επεξεργαστούν περαιτέρω.

```
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);
$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;
```

Το hash στυλ αντιστοιχίζεται στα γνωρίσματα παρουσίασης του SVG: `stroke`, `fill`, `stroke-width`, `opacity` και τα υπόλοιπα. Ένα `group` (με ψευδώνυμο `g`) με ένα στυλ είναι ο τρόπος με τον οποίο αποφεύγετε την επανάληψη γνωρισμάτων σε κάθε στοιχείο· τα παιδιά κληρονομούν και μπορούν να υπερκαλύψουν. Το κείμενο προσαρτάται με ένα αλυσιδωτό `cdata` γιατί το περιεχόμενο χαρακτήρων είναι ένας κόμβος-παιδί, όχι ένα γνώρισμα.

Για ένα πολύγωνο από υπολογισμένα σημεία, το `get_path` συναρμολογεί τη συμβολοσειρά σημείων για εσάς:

```
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' });
```

Το `xmlify` (γράφεται επίσης `to_xml` ή `render`) επιστρέφει το ολοκληρωμένο έγγραφο ως συμβολοσειρά. Γράψτε το σε ένα αρχείο `.svg` και ανοίγει σε οποιονδήποτε φυλλομετρητή· δώστε το σε έναν μετατροπέα για PDF ή PNG.

Cairo ή SVG;

Θέλετε ...	Καταφύγετε σε
Pixel τώρα, εξομαλυμένα, από συντεταγμένες πραγματικών αριθμών	`Cairo`
Έξοδο vector ως αρχείο για φυλλομετρητές ή άλλα εργαλεία	`SVG`
Να αποδώσετε το ίδιο σχέδιο σε PNG, PDF και SVG από ένα API	`Cairo`
Καμία μεταγλωττισμένη εξάρτηση καθόλου (καθαρή Perl)	`SVG`

Τα δύο επίσης συνθέτονται: παραγάγετε δομή ως markup SVG, και έπειτα αποδώστε την σε pixel με έναν rasterizer όταν χρειάζεστε ένα bitmap. Για εφαρμοσμένη έξοδο vector στη συγκεκριμένη μορφή διαγραμμάτων και γραφημάτων, και για την ανάγνωση των μεταδεδομένων που βρίσκονται ήδη μέσα σε αρχεία εικόνας, το *επόμενο κεφάλαιο* καλύπτει τα πραγματιστικά στηρίγματα της δουλειάς raster και vector.

Μεταδεδομένα εικόνας και εφαρμοσμένη δημιουργία 2D γραφημάτων

Δύο πραγματιστικές γωνιές της δουλειάς 2D ολοκληρώνουν τον άξονα των εργαλειοθηκών. Πρώτον, η ανάγνωση των δεδομένων που βρίσκονται ήδη μέσα σε αρχεία εικόνας: διαστάσεις, μορφή, και το μπλοκ EXIF που γράφει μια κάμερα. Δεύτερον, ο πιο συνηθισμένος λόγος για τον οποίο ο κόσμος σχεδιάζει καθόλου, που είναι η μετατροπή αριθμών σε ένα γράφημα. Κανένα από τα δύο δεν είναι εντυπωσιακό, και τα δύο προκύπτουν διαρκώς.

Ανάγνωση μεταδεδομένων: Image::Info

Πριν σχεδιάσετε πάνω ή αλλάξετε το μέγεθος μιας εικόνας, συχνά χρειάζεται απλώς να ξέρετε γι” αυτήν: πόσο μεγάλη είναι, τι μορφή, τι κατέγραψε η κάμερα. Το `Image::Info` είναι καθαρή Perl, διαβάζει τις κεφαλίδες χωρίς να αποκωδικοποιεί ολόκληρη την εικόνα, και επιστρέφει μια απλή αναφορά hash.

```
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};
```

Το `dim` είναι μια διευκόλυνση που βγάζει το πλάτος και το ύψος από το hash. Τα υπόλοιπα κλειδιά εξαρτώνται από τη μορφή και από το τι κουβαλά το αρχείο: ανάλυση, τύπος χρώματος, και τα κοινά πεδία EXIF για φωτογραφίες. Για ένα γρήγορο «τι είναι αυτό το αρχείο» είναι όλο κι όλο ό,τι χρειάζεστε, και κοστίζει σχεδόν τίποτα γιατί δεν αποκωδικοποιεί ποτέ τα pixel.

Ανάγνωση και εγγραφή μεταδεδομένων: Image::ExifTool

Όταν χρειάζεστε την πλήρη επιφάνεια μεταδεδομένων, και ειδικά όταν χρειάζεται να τη **γράψετε**, το `Image::ExifTool` είναι το οριστικό εργαλείο. Χειρίζεται τα μεταδεδομένα ουσιαστικά κάθε μορφής εικόνας και μέσων, διαβάζει χιλιάδες tags, και μπορεί να τα επεξεργαστεί επιτόπου χωρίς να αγγίξει τα δεδομένα της εικόνας.

```
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
```

Το `Image::Info` απαντά στο «πόσο μεγάλη, τι μορφή» φθηνά· το `Image::ExifTool` είναι αυτό στο οποίο καταφεύγετε όταν τα ίδια τα μεταδεδομένα είναι η δουλειά (αφαίρεση tags τοποθεσίας για ιδιωτικότητα, ομαδική προσθήκη ετικετών σε μια βιβλιοθήκη φωτογραφιών, ανάγνωση πεδίων σημειώσεων του κατασκευαστή). Και τα δύο αφήνουν τα pixel ήσυχα.

Δημιουργία γραφημάτων: η ειλικρινής κατάσταση της τεχνολογίας

Η δημιουργία γραφημάτων είναι εφαρμοσμένη σχεδίαση 2D: άξονες, ράβδοι, γραμμές και ετικέτες συντεθειμένες από τα πρωτογενή που ήδη γνωρίζετε. Το τοπίο δημιουργίας γραφημάτων της Perl είναι πιο ισχνό απ” ό,τι ήταν κάποτε. Αρκετά κάποτε δημοφιλή αρθρώματα είναι πλέον ασυντήρητα, οπότε αυτή η ενότητα ξεκινά με αυτά που εξακολουθούν να λειτουργούν και ονομάζει τα υπόλοιπα καθαρά.

GD::Graph για γρήγορα γραφήματα raster

Το `GD::Graph` χτίζει πάνω στο GD και εξακολουθεί να συντηρείται. Είναι η συντομότερη διαδρομή προς ένα ραβδόγραμμα, γραμμικό γράφημα ή κυκλικό γράφημα ως PNG. Του δίνετε δεδομένα ως ένα σύνολο παράλληλων πινάκων (ο πρώτος είναι οι ετικέτες x, οι υπόλοιποι είναι σειρές δεδομένων) και το αποδίδει.

```
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(
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    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;
```

Τα `GD::Graph::bars`, `::lines`, `::pie` και `::points` είναι τα βαριά όχηματα. Το `plot` επιστρέφει ένα `GD::Image`, οπότε ισχύουν όλα από το *κεφάλαιο του GD* σχετικά με την έξοδο. Αυτή είναι η πραγματιστική προεπιλογή για ένα γράφημα στον δίσκο.

SVG για γραφήματα vector

Όταν το γράφημα πρέπει να παραμείνει ευκρινές σε οποιοδήποτε μέγεθος, ή να ζει σε μια ιστοσελίδα ως markup, χτίστε το ως *SVG* απευθείας. Δεν υπάρχει αφαίρεση γραφήματος να μάθετε: υπολογίζετε τα ορθογώνια των ράβδων και τις γραμμές των αξόνων μόνοι σας και τα εκπέμπετε. Αυτό είναι περισσότερος κώδικας από το `GD::Graph`, αλλά είναι ανεξάρτητο από την ανάλυση και δεν χρειάζεται καμία μεταγλωττισμένη εξάρτηση.

```
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' });
}
$svg->line(x1 => 30, y1 => $base, x2 => 380, y2 => $base,
           style => { stroke => 'black' });

print $svg->xmlify;
```

Για οτιδήποτε πέρα από ένα απλό ραβδόγραμμα ή γραμμικό γράφημα το χειρόγραφο SVG μεγαλώνει γρήγορα, αλλά ακριβώς για τις κοινές περιπτώσεις είναι καθαρό και χωρίς εξαρτήσεις.

Οι παλαιές επιλογές, ονομασμένες

Δύο αρθρώματα δημιουργίας γραφημάτων που θα βρείτε να αναφέρονται στο διαδίκτυο είναι ουσιαστικά ασυντήρητα και σημειώνονται εδώ μόνο για να μην αποτελεί έκπληξη η κατάστασή τους. Το **`Chart::Clicker`** παράγει ελκυστικά γραφήματα αποδοσμένα με Cairo αλλά δεν έχει δει κυκλοφορία εδώ και χρόνια και κουβαλά ένα μεγάλο δέντρο εξαρτήσεων (συμπεριλαμβανομένου του Moose)· αν το τρέχετε ήδη, λειτουργεί, αλλά δεν είναι επιλογή για νέο κώδικα. Το **`Chart::Gnuplot`** οδηγεί ένα εξωτερικό εκτελέσιμο `gnuplot` και είναι επίσης αδρανές· αν έχετε το `gnuplot` εγκατεστημένο και θέλετε τη δύναμη σχεδιασμού του, το να το καλέσετε απευθείας (ή μέσω ενός λεπτού δικού σας περιτυλίγματος) είναι πιο συντηρήσιμο από το να εξαρτάστε από το εγκαταλελειμμένο άρθρωμα. Ειδικά για επιστημονικό σχεδιασμό γραφημάτων, τα ίδια τα αρθρώματα γραφικών του οικοσυστήματος PDL είναι η καλύτερα υποστηριζόμενη διαδρομή και ανήκουν στο PDL αντί εδώ.

Πού αφήνει αυτό τον άξονα των εργαλειοθηκών

Μπορείτε τώρα να δημιουργήσετε, να φορτώσετε, να σχεδιάσετε πάνω, να φιλτράρετε, να αποθηκεύσετε, να διανυσματοποιήσετε, να επιθεωρήσετε και να απεικονίσετε σε γράφημα εικόνες 2D εξ ολοκλήρου στον δίσκο. Το ένα πράγμα που λείπει είναι μια **οθόνη**: το να φέρετε ένα σχέδιο σε ένα ζωντανό, διαδραστικό παράθυρο. Αυτό είναι το *επόμενο κεφάλαιο*, και είναι επίσης η γέφυρα προς τα δύο ολοκληρωτικά έργα, που και τα δύο χρειάζονται ένα παράθυρο για να εκτελεστούν.

2D σε μια οθόνη: καμβάδες GUI

Όλα ως τώρα σχεδίαζαν σε ένα αρχείο. Αυτό το κεφάλαιο σχεδιάζει σε ένα **παράθυρο**: έναν διαδραστικό καμβά που ξαναβάφεται, ανταποκρίνεται στο ποντίκι και στο πληκτρολόγιο, και μπορεί να κινείται. Κάθε εργαλειοθήκη GUI λειτουργεί με τον ίδιο τρόπο σε γενικές γραμμές: δημιουργείτε ένα παράθυρο, του δίνετε έναν χειριστή βαφής που σχεδιάζει το περιεχόμενό του, και η εργαλειοθήκη καλεί αυτόν τον χειριστή όποτε το παράθυρο χρειάζεται επανασχεδίαση. Η σχεδίαση μέσα στον χειριστή χρησιμοποιεί πρωτογενή που είναι πλέον οικεία: γραμμές, ελλείψεις, πολύγωνα, γεμίσματα.

Το τοπίο των GUI της Perl έχει μια σαφή καλύτερη επιλογή για γραφικά και αρκετές άλλες με συμβιβασμούς. Αυτό το κεφάλαιο ξεκινά με το `Prima`, γιατί συντηρείται ενεργά, είναι αυτοτελές, και έχει έναν πραγματικό πυρήνα γραφικών· έπειτα καλύπτει τα `Tk` και `Gtk3` για όταν έχετε λόγο να τα προτιμήσετε· και είναι ειλικρινές για το `Wx`, που δεν είναι πλέον μια υγιής επιλογή.

Prima: η αυτοτελής επιλογή

Το `Prima` είναι μια εργαλειοθήκη GUI γραμμένη για την Perl με τη δική της μηχανή γραφικών. Δεν τυλίγει μια εξωτερική εργαλειοθήκη που πρέπει να εγκαταστήσετε και να ταιριάξετε εκδόσεις μαζί της· σχεδιάζει εγγενώς (στο X11 στο Linux) και αποστέλλει τα δικά του πρωτογενή, χειρισμό εικόνας και γραμματοσειρές. Συντηρείται, και είναι η σύσταση για νέα δουλειά GUI με πολλά γραφικά στην Perl.

Ένα ελάχιστο παράθυρο με έναν χειριστή βαφής:

```
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(cl::White);
        $canvas->bar(0, 0, $canvas->size);   # bar is a filled
↪rectangle
        $canvas->color(cl::Red);
        $canvas->fill_ellipse(200, 150, 120, 90);
        $canvas->color(cl::Black);
        $canvas->line(0, 0, 400, 300);
    },
);

Prima->run;
```

Δύο σημειώσεις ονοματοδοσίας που παγιδεύουν τους νεοφερμένους. Το **γεμάτο ορθογώνιο** του Prima είναι το `bar`· το σκέτο `rectangle` σχεδιάζει το περίγραμμα. Το γέμισμα πολυγώνου είναι το `fillpoly` (μία λέξη), που παίρνει μια αναφορά πίνακα συντεταγμένων. Τα χρώματα προέρχονται από τις σταθερές `cl::` (`cl::Red`, `cl::White` κ.ο.κ.) ή από έναν πακεταρισμένο ακέραιο RGB, και ορίζονται ως η κατάσταση σχεδίασης με το `color` πριν σχεδιάσετε, αντί να περνιούνται σε κάθε πρωτογενές.

Τα πρωτογενή του `Prima::Drawable` είναι το πλήρες σύνολο που περιμένετε: `line`, `ellipse` και `fill_ellipse`, `arc`, `sector` και `fill_sector`, `polyline`, `fillpoly`, `bar`, `rectangle`, `flood_fill`, και `put_image` για blit ενός bitmap. Η κατάσταση σχεδίασης περιλαμβάνει τα `color`, `backColor`, `lineWidth` και μοτίβα γεμίσματος.

Σχεδίαση εκτός οθόνης και αποθήκευση

Τα ίδια πρωτογενή λειτουργούν σε ένα `Prima::Image`, μια επιφάνεια εκτός οθόνης, οπότε μπορείτε να σχεδιάσετε χωρίς παράθυρο και να αποθηκεύσετε το αποτέλεσμα. Έτσι ένα πρόγραμμα Prima εξάγει αυτό που σχεδίασε:

```
my $img = Prima::Image->new(width => 200, height => 200, type =>
↪im::RGB);
$img->begin_paint;
$img->color(cl::Blue);
$img->fill_ellipse(100, 100, 80, 80);
$img->end_paint;
$img->save('out.png');
```

Τα `begin_paint` και `end_paint` περικλείουν τη σχεδίαση πάνω στην εικόνα εκτός οθόνης, και το `save` τη γράφει σε μια μορφή που συμπεραίνεται από την επέκταση. Αυτή η διττή φύση (ίδιος κώδικας σχεδίασης για την οθόνη και για ένα αρχείο) είναι ο λόγος για τον οποίο τα *ολοκληρωτικά κεφάλαια* χτίζουν πάνω στο Prima.

Tk: ο κλασικός Canvas

Το Tk (Perl/Tk) είναι η μακροχρόνια επιλογή, και το widget Canvas του έχει ένα διαφορετικό μοντέλο που αξίζει να γνωρίζετε: είναι **retained-mode**. Αντί για έναν χειριστή βαφής που ξανασχεδιάζει τα πάντα, δημιουργείτε αντικείμενα καμβά (μια γραμμή, ένα οβάλ, ένα πολύγωνο) που το widget θυμάται και ξανασχεδιάζει για εσάς, και τα μετακινείτε ή τα αναδιαμορφώνετε με το id τους. Αυτό ταιριάζει σε διαγράμματα και επεξεργαστές όπου τα αντικείμενα διατηρούνται και υφίστανται χειρισμό.

```
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;
```

Το Tk είναι σταθερό και λειτουργεί, αν και βρίσκεται σε συντήρηση αντί για ενεργή ανάπτυξη και κουβαλά μερικά κληρονομημένα CVE στο ενσωματωμένο Tcl/libpng του. Για ένα διάγραμμα με μόνιμα αντικείμενα ή ένα γρήγορο κλασικό GUI είναι μια λογική επιλογή· για μια κίνηση με επανασχεδίαση σε κάθε καρέ το μοντέλο immediate-mode του Prima ταιριάζει καλύτερα.

Gtk3: σχεδίαση μέσω Cairo

Το Gtk3 συνδέει το GTK 3 μέσω introspection του GObject. Η ιστορία σχεδίασής του είναι ότι ένα widget Gtk3::DrawingArea εκπέμπει ένα σήμα draw στο οποίο δίνεται ένα context *Cairo*, οπότε σχεδιάζετε με το API του Cairo από το κεφάλαιο των vector μέσα στον χειριστή. Αν η εφαρμογή σας είναι ήδη μια εφαρμογή GTK, αυτός είναι ο φυσικός τρόπος να προσθέσετε προσαρμοσμένα γραφικά· αν δεν είναι, το να τραβήξετε μέσα ολόκληρη τη στοίβα GTK (τις βιβλιοθήκες ανάπτυξης του GTK 3 και τα δεδομένα introspection του GObject) είναι μια μεγάλη εξάρτηση μόνο για γραφικά.

```
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;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    return 0;
});

$window->show_all;
Gtk3->main;
```

Όλα όσα ξέρετε για το Cairo ισχύουν αμετάβλητα μέσα σε αυτόν τον χειριστή. Το `Gtk3` είναι χρησιμοποιήσιμο αλλά σε αργή συντήρηση, και το βάρος των εξαρτήσεων σημαίνει ότι κερδίζει τη θέση του κυρίως όταν το GTK είναι ήδη στην εικόνα.

Wx: αναφορά, όχι σύσταση

Θα βρείτε το `Wx` (wxPerl) να αναφέρεται ως ο τρόπος να χτίσετε cross-platform GUI με εγγενή εμφάνιση στην Perl, και ιστορικά ήταν. Περιλαμβάνεται εδώ ώστε να είναι σαφής η τρέχουσα κατάστασή του: το `Wx` και η εξάρτηση κατασκευής του `Alien::wxWidgets` δεν έχουν κυκλοφορήσει έκδοση εδώ και χρόνια, και ένα μεγάλο ποσοστό των αυτοματοποιημένων αναφορών κατασκευής αποτυγχάνει έναντι του τρέχοντος wxWidgets. Δεν είναι επιλογή για νέα δουλειά. Αν χρειάζεστε ένα GUI με προσαρμοσμένη σχεδίαση σήμερα, το `Prima` είναι η συντηρούμενη αυτοτελής επιλογή και το `Tk` το συντηρούμενο κλασικό· καταφύγετε σε ένα από αυτά τα δύο αντ” αυτού.

Επιλογή

Θέλετε ...	Καταφύγετε σε
Συντηρούμενο, αυτοτελές, νέο GUI με πολλά γραφικά	`Prima`
Έναν καμβά retained-mode με μόνιμα αντικείμενα	`Tk`
Προσαρμοσμένη σχεδίαση μέσα σε μια υπάρχουσα εφαρμογή GTK	`Gtk3`
Κίνηση ανά καρέ και έναν βρόχο πραγματικού χρόνου	`Prima`

Με ένα παράθυρο και έναν χειριστή βαφής στα χέρια, ο άξονας των εργαλειοθηκών ολοκληρώνεται. Ο *επόμενος άξονας* πηγαίνει πιο μακριά από οποιοδήποτε μεμονωμένο άρθρωμα του CPAN: χρησιμοποιεί το εγγενές FFI της pperl για να προσεγγίσει βιβλιοθήκες γραφικών C απευθείας, και έπειτα ξοδεύει όλα όσα υπάρχουν στον οδηγό σε δύο έργα που μπορείτε να εκτελέσετε, χτισμένα και τα δύο πάνω στο θεμέλιο του Prima από αυτό το κεφάλαιο.

Φτάνοντας πιο μακριά: το Peta::FFI για γραφικά

Τα κεφάλαια των εργαλειοθηκών κάλυψαν αυτό που το CPAN ήδη τυλίγει. Αυτό το κεφάλαιο, και ο υπόλοιπος οδηγός, πηγαίνει πιο μακριά: προσεγγίζει μια βιβλιοθήκη γραφικών σε C που δεν έχει καθόλου σύνδεση με το CPAN, χρησιμοποιώντας το εγγενές FFI της pperl. Αυτή είναι μια δυνατότητα αποκλειστική της pperl. Η σύνδεση μιας βιβλιοθήκης C με τον παραδοσιακό τρόπο σημαίνει τη συγγραφή ενός shim σε C, τη μεταγλώττισή του και την αποστολή μιας εργαλειοαλυσίδας κατασκευής μαζί με τον κώδικά σας· η pperl καλεί τη βιβλιοθήκη C απευθείας, τον ίδιο μηχανισμό που σας δίνει ήδη το `Math::GMP` χωρίς καμία μεταγλωττισμένη σύνδεση να κατασκευαστεί.

Αυτό δεν είναι το `FFI::Platypus`. Το `Peta::FFI` είναι ενσωματωμένο στον χρόνο εκτέλεσης, και οι επιμελημένες συνδέσεις του είναι εγγενής Rust, όχι δηλώσεις από την πλευρά της Perl. Η διάκριση έχει σημασία για τα γραφικά, γιατί μια πραγματική σύνδεση γραφικών χρειάζεται να διαχειρίζεται αντικείμενα δεσμευμένα από τη C (μια επιφάνεια, ένα context) σε πολλές κλήσεις, και το μοντέλο της pperl το χειρίζεται με τον τρόπο που το κάνει η ίδια η βιβλιοθήκη C.

Η έγκυρη αναφορά για το στρώμα FFI είναι η *σελίδα Auto-FFI*· αυτό το κεφάλαιο την εφαρμόζει ειδικά στα γραφικά.

Τα τρία επίπεδα

Το `Peta::FFI` κατανοείται καλύτερα ως τρία επίπεδα, από το ακατέργαστο ως το επιμελημένο.

Επίπεδο 1, ακατέργαστες κλήσεις. Ονομάζετε μια βιβλιοθήκη, μια συνάρτηση και μια συμβολοσειρά υπογραφής τύπων, και η pperl δρομολογεί την κλήση μέσω του `libffi`. Τίποτα δεν παράγεται εκ των προτέρων· οποιαδήποτε υπογραφή μπορείτε να γράψετε λειτουργεί.

```
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);
```

Η υπογραφή `"(args)ret"` χρησιμοποιεί ένα γράμμα ανά τύπο: `v` void, `i` int, `l` long, `L` unsigned long ή size_t, `d` double, `f` float, `p` μια συμβολοσειρά C ως είσοδο, και `P` έναν μεταβλητό ενταμιευτή εξόδου. Αυτό το σύνολο καλύπτει μια εκπληκτική ποσότητα ενός API της C: οτιδήποτε του οποίου τα ορίσματα και η επιστροφή είναι βαθμωτά, συμβολοσειρές ή ένας ενταμιευτής bytes.

Επίπεδο 2, ένα στρώμα περιγραφής δεδομένων. Πολλές συναρτήσεις C παίρνουν ή επιστρέφουν **structs**, όχι βαθμωτά, και οι ακατέργαστοι κωδικοί τύπων δεν μπορούν να περιγράψουν τη διάταξη ενός struct. Η προγραμματισμένη απάντηση είναι ένα δηλωτικό στρώμα όπου περιγράφετε τα πεδία μιας εγγραφής μία φορά και αφήνετε το FFI να τη μεταφέρει. Αυτή είναι η αναδυόμενη μέση οδός· δεν είναι ακόμη ένα άρθρωμα που αποστέλλεται, και αυτός ο οδηγός είναι ρητός σε κάθε σημείο για το πού θα το χρειαζόταν μια σύνδεση.

Επίπεδο 3, επιμελημένες συνδέσεις. Το ανώτατο επίπεδο είναι μια χειρόγραφη εγγενής σύνδεση: ένα άρθρωμα Rust που ανοίγει τη βιβλιοθήκη μία φορά, επιλύει κάθε σύμβολο που χρειάζεται, και παρουσιάζει ένα καθαρό API της Perl, διαχειριζόμενο τα αντικείμενα που έχουν δεσμευτεί από τη C πίσω από τιμές της Perl. Το `Math::GMP` είναι ακριβώς αυτό: `use Math::GMP` και παίρνετε ακεραίους αυθαίρετης ακρίβειας με υπερφόρτωση τελεστών, και πουθενά δεν αγγίξατε το `dlopen` ή μια συμβολοσειρά υπογραφής. Το *επόμενο κεφάλαιο* παίρνει μία από αυτές που η pperl ήδη αποστέλλει για μια βιβλιοθήκη γραφικών, το `Cairo`, και δείχνει πώς λειτουργεί.

Τι μπορούν και τι δεν μπορούν να προσεγγίσουν οι ακατέργαστες κλήσεις

Η διαχωριστική γραμμή για τα γραφικά είναι το struct. Ένα μεγάλο μέρος ενός API γραφικών σε C είναι προσεγγίσιμο στο Επίπεδο 1 αυτή τη στιγμή, γιατί είναι βαθμωτά και αδιαφανείς χειριστές:

- Οι συναρτήσεις που παίρνουν και επιστρέφουν **αριθμούς** (ορίζουν ένα πλάτος γραμμής, μια ακτίνα, μια συνιστώσα χρώματος) αντιστοιχίζονται απευθείας στα `i`, `d`, `f` και τα συναφή.
- Οι συναρτήσεις που παίρνουν μια **συμβολοσειρά C** (φορτώνουν μια γραμματοσειρά με το όνομά της, ανοίγουν ένα αρχείο) χρησιμοποιούν το `p`.
- Οι συναρτήσεις που γεμίζουν έναν **ενταμιευτή που παρέχει ο καλών** (διαβάζουν pixel σε μνήμη που σας ανήκει) χρησιμοποιούν το `P`.
- Ένας **αδιαφανής χειριστής** που επιστρέφεται από τη βιβλιοθήκη (ένας δείκτης επιφάνειας, ένας δείκτης context) είναι απλώς μια διεύθυνση. Η pperl αναπαριστά ήδη έναν χειριστή βιβλιοθήκης ως ακέραιο, και το επιμελημένο πρότυπο στο επόμενο κεφάλαιο αποθηκεύει αυτούς τους χειριστές αντικειμένων με τον ίδιο τρόπο: ως ακέραιο μέσα σε μια blessed αναφορά της Perl.

Πού σταματούν οι ακατέργαστες κλήσεις:

- **Structs που περνιούνται ή επιστρέφονται κατά τιμή.** Μια συνάρτηση που παίρνει ένα `Color { r, g, b, a }` κατά τιμή, ή επιστρέφει ένα `Rectangle`, δεν μπορεί να περιγραφεί από τους κωδικούς βαθμωτών τύπων. Αυτό είναι το κενό του Επιπέδου 2.
- **Callbacks.** Μια βιβλιοθήκη που καλεί πίσω τον κώδικά σας (ένας βρόχος συμβάντων που επικαλείται έναν χειριστή, ένα σήμα σχεδίασης) χρειάζεται έναν δείκτη συνάρτησης C που ξαναμπαίνει στην Perl. Αυτό είναι πέρα από το ακατέργαστο στρώμα.
- **Unions.** Ένας τύπος συμβάντος που είναι ένα union από πολλά σχήματα struct (η κλασική εγγραφή συμβάντος παραθύρων) συνδυάζει και τα δύο παραπάνω προβλήματα.

Αυτά δεν είναι μόνιμα τείχη· είναι το σύνορο προς το οποίο μεγαλώνουν τα στρώματα του FFI. Η αξία αυτού του οδηγού είναι ότι σας λέει ακριβώς σε ποια πλευρά της γραμμής πέφτει μια δεδομένη εργασία, ώστε ούτε να αποφεύγετε αυτό που λειτουργεί ούτε να υπόσχεστε αυτό που δεν λειτουργεί.

Γιατί μια επιμελημένη σύνδεση, όχι ακατέργαστες κλήσεις, για πραγματική δουλειά

Θα μπορούσατε να οδηγήσετε μια βιβλιοθήκη γραφικών εξ ολοκλήρου μέσω κλήσεων `call` του Επιπέδου 1, και για ένα γρήγορο πείραμα αυτό είναι εντάξει. Για οτιδήποτε πραγματικό, μια επιμελημένη σύνδεση του Επιπέδου 3 είναι η σωστή μονάδα, για τρεις λόγους.

Πρώτον, **η διάρκεια ζωής του αντικειμένου**. Ένα context γραφικών δεσμεύεται από τη βιβλιοθήκη και πρέπει να αποδεσμευτεί ακριβώς μία φορά. Μια επιμελημένη σύνδεση δένει αυτή τη διάρκεια ζωής στο DESTROY μιας τιμής της Perl, οπότε το αντικείμενο C αποδεσμεύεται όταν το αντικείμενο της Perl εξαφανίζεται, χωρίς χειροκίνητη λογιστική. Οι ακατέργαστες κλήσεις το αφήνουν εξ ολοκλήρου σε εσάς.

Η πρώτης τάξεως **εργονομία** της Perl είναι ο δεύτερος λόγος: το `$surface->fill` διαβάζεται καλύτερα και είναι δυσκολότερο να το κάνει κανείς λάθος από ένα `call` με χειρόγραφη υπογραφή σε κάθε σημείο χρήσης.

Τρίτον, **η υπογραφή γράφεται μία φορά**. Σε μια επιμελημένη σύνδεση οι συμβολοσειρές `"(ppd)i"` ζουν σε ένα άρθρωμα Rust, ελεγμένες μία φορά· κάθε καλών απλώς χρησιμοποιεί τη μέθοδο. Η διασπορά ακατέργαστων υπογραφών μέσα στον κώδικα της εφαρμογής είναι ο τρόπος με τον οποίο προκύπτει μια αναντιστοιχία που διαφθείρει τη στοίβα.

Η *επίδειξη του Cairo* κάνει και τα τρία απτά με τη σύνδεση `Cairo` που αποστέλλει η pperl, μια πραγματική βιβλιοθήκη vector της οποίας το API είναι σχεδόν εξ ολοκλήρου βαθμωτά και ένας αδιαφανής χειριστής, που είναι ακριβώς το σχήμα που προσεγγίζεται καθαρά σήμερα.

Το σχήμα αυτών που ακολουθούν

Το επόμενο κεφάλαιο κατασκευάζει μια επιμελημένη σύνδεση Cairo με τον τρόπο που κατασκευάζεται το `Math::GMP`, ώστε να δείτε μια βιβλιοθήκη γραφικών να σηκώνεται στην pperl από άκρη σε άκρη. Έπειτα τα δύο ολοκληρωτικά έργα χρησιμοποιούν τη συντηρούμενη στοίβα εργαλειοθηκών από τον άξονα B, αγκυρωμένη στο Prima, για να φτιάξουν μια εφαρμογή σχεδίασης και ένα 2D παιχνίδι που μπορείτε να εκτελέσετε, με το νήμα του FFI υφασμένο εκεί όπου κερδίζει τη θέση του και ειλικρινά επισημασμένο εκεί όπου αρχίζει το σύνορο. Το *τελευταίο κεφάλαιο* σχεδιάζει τον χάρτη αυτών που απομένουν.

Μια εγγενής σύνδεση Cairo, που αποστέλλεται

Το *κεφάλαιο των vector* σχεδίασε με το `Cairo` ως ένα συνηθισμένο άρθρωμα του CPAN. Αυτό το κεφάλαιο δείχνει την άλλη πλευρά του ίδιου κώδικα: η pperl αποστέλλει μια **εγγενή επανυλοποίηση του `Cairo`**, οπότε το `use Cairo` λειτουργεί στην pperl χωρίς εγκατεστημένη τη διανομή Cairo του CPAN και χωρίς να εμπλέκεται μεταγλωττιστής C. Η Perl που γράφετε είναι πανομοιότυπη· αυτό που τη στηρίζει είναι το ίδιο το FFI της pperl που προσεγγίζει το `libcairo` απευθείας. Το Cairo αποτελεί το ιδανικό δουλεμένο παράδειγμα γιατί το API της C του είναι σχεδόν εξ ολοκλήρου βαθμωτά και ένας αδιαφανής χειριστής, που είναι ακριβώς το σχήμα που προσεγγίζεται καθαρά σήμερα.

Αυτή είναι μια πραγματική, απεσταλμένη σύνδεση, όχι ένα σκίτσο. Η ανάγνωσή της σας λέει τι είναι μια επιμελημένη εγγενής σύνδεση γραφικών, πώς ταιριάζουν τα κομμάτια, και πώς να αναγνωρίσετε (ή να ζητήσετε) την ίδια μεταχείριση για μια άλλη βιβλιοθήκη. Η κατασκευή μιας τέτοιας σύνδεσης είναι εργασία του χρόνου εκτέλεσης· αυτό που γράφετε ως χρήστης είναι η Perl στην απέναντι πλευρά της, και αυτή η Perl είναι απλώς το *API του* `Cairo` που ήδη είδατε.

Το API της Perl: ακριβώς αυτό του CPAN

Η εγγενής σύνδεση εκθέτει τα **πραγματικά ονόματα πακέτων και μεθόδους του `Cairo`**: `Cairo::ImageSurface`, `Cairo::Context`, `Cairo::PdfSurface` και τα υπόλοιπα, με τις υπογραφές μεθόδων του upstream. Κώδικας γραμμένος για το `Cairo` του CPAN τρέχει αμετάβλητος· μεταφερμένες σουίτες δοκιμών Cairo περνούν επάνω του. Το πρόγραμμα από το κεφάλαιο των vector είναι το πρόγραμμα εδώ:

```
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
```

Τίποτα σε αυτό το πρόγραμμα δεν υπαινίσσεται πώς είναι υλοποιημένο. Δεν υπάρχει dlopen, καμία συμβολοσειρά υπογραφής, κανένα αντικείμενο σύνδεσης: μόνο Cairo. Το υπόλοιπο αυτού του κεφαλαίου είναι αυτό που κάνει το use Cairo να το κάνει αυτό στην pperl, και το πρότυπο είναι το ίδιο που κρύβεται πίσω από το Math::GMP.

Βήμα 1: άνοιξε τη βιβλιοθήκη μία φορά, επίλυσε τα σύμβολα

Η σύνδεση ανοίγει το libcairo.so.2 μία μόνο φορά, όταν φορτώνεται το άρθρωμα, και επιλύει κάθε συνάρτηση C που χρειάζεται σε έναν πίνακα δεικτών συναρτήσεων που κρατιέται για όλη τη ζωή της διεργασίας. Αυτό είναι το ίδιο σχήμα με τη σύνδεση Math::GMP, που ανοίγει το libgmp μία φορά και επιλύει τα σύμβολά του mpz_*. Για το Cairo τα επιλυμένα σύμβολα είναι αυτά πίσω από τις μεθόδους παραπάνω:

```
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)
    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*)
```

Αν το libcairo δεν είναι εγκατεστημένο στο σύστημα, η σύνδεση το αναφέρει μία φορά με ένα χρήσιμο μήνυμα αντί να καταρρεύσει. Καθεμία από αυτές τις συναρτήσεις C έχει μια υπογραφή που οι κωδικοί τύπων του FFI ήδη εκφράζουν: τα ορίσματα είναι

ο αδιαφανής χειριστής (μια διεύθυνση, άρα ένας ακέραιος), doubles και μια διαδρομή συμβολοσειράς. Δεν υπάρχει struct κατά τιμή στο πυρηνικό API σχεδίασης, γι" αυτό το Cairo είναι το καθαρό παράδειγμα.

Το Cairo είναι μια μεγάλη βιβλιοθήκη, οπότε η πραγματική σύνδεση οργανώνεται ανά πακέτο: τα `Cairo::Surface`, `Cairo::Context`, `Cairo::Path`, `Cairo::Pattern`, `Cairo::Matrix`, `Cairo::Font` και `Cairo::Region` επιλύουν το καθένα τα σύμβολα που χρειάζεται. Αυτό είναι μια διευκόλυνση της υλοποίησης· ο χρήστης βλέπει ένα `Cairo`.

Βήμα 2: ο αδιαφανής χειριστής ως blessed ακέραιος

Το Cairo επιστρέφει ένα `cairo_surface_t*` και ένα `cairo_t*`: δείκτες σε structs της C των οποίων τα εσωτερικά δεν αγγίζετε ποτέ. Η σύνδεση δεν χρειάζεται να κατανοεί τη διάταξή τους, μόνο να κρατά τον δείκτη και να τον επιστρέφει σε κάθε κλήση. Η pperl αναπαριστά έναν τέτοιον χειριστή με τον ίδιο τρόπο που το `Math::GMP` κρατά τον δείκτη του `mpz_t`: ως έναν **ακέραιο μέσα σε μια blessed αναφορά**.

```
Cairo::ImageSurface object  =  bless(\ (surface_ptr as integer),
↪'Cairo::ImageSurface')
Cairo::Context object       =  bless(\ (context_ptr as integer),
↪'Cairo::Context')
```

Όταν καλείται μια μέθοδος, η σύνδεση διαβάζει τον ακέραιο πίσω από τη blessed αναφορά, τον μετατρέπει στον δείκτη C, και καλεί τη συνάρτηση. Το bless σάς δίνει αποστολή μεθόδων (`$cr->stroke`), κληρονομικότητα (κάθε συγκεκριμένη κλάση επιφάνειας είναι υποκλάση `Cairo::Surface`) και, κρίσιμα, ένα hook `DESTROY`. Η μόνη διαφορά από το `Math::GMP` είναι ότι το αδιαφανές πράγμα είναι μια επιφάνεια σχεδίασης αντί για έναν μεγάλο ακέραιο.

Βήμα 3: η διάρκεια ζωής δεμένη στο DESTROY

Ένα αντικείμενο γραφικών C πρέπει να αποδεσμεύεται ακριβώς μία φορά. Αν το διαρρεύσετε καίτε μνήμη σε κάθε σχεδίαση· αν το αποδεσμεύσετε δύο φορές διαφθείρετε τον σωρό. Η σύνδεση δένει τη ζωή του αντικειμένου C σε αυτήν του αντικειμένου της Perl: όταν η blessed αναφορά βγαίνει εκτός εμβέλειας, η Perl καλεί το `DESTROY`, και η σύνδεση καλεί τον αντίστοιχο καταστροφέα C, σεβόμενη την καταμέτρηση αναφορών του cairo.

```
Cairo::Context DESTROY  ->  cairo_destroy(context_ptr)
Cairo::Surface DESTROY  ->  cairo_surface_destroy(surface_ptr)
```

Το `Math::GMP` κάνει ακριβώς αυτό: ο καταστροφέας του καλεί το `mpz_clear` και αποδεσμεύει το `mpz_t`. Εσείς, ο χρήστης, δεν καλείτε ποτέ μια συνάρτηση αποδέσμευσης· τα αντικείμενα σχεδίασης καθαρίζονται όταν εξαφανίζονται οι μεταβλητές της Perl που τα κρατούν. Η σύνδεση επιβάλλει τη σειρά που απαιτεί το cairo, αποδεσμεύοντας ένα context πριν από την επιφάνεια στην οποία σχεδιάζει.

Βήμα 4: οι μέθοδοι

Κάθε μέθοδος της Perl είναι μια μικρή εγγενής συνάρτηση που διαβάζει τον χειριστή, διαβάζει τα βαθμωτά ορίσματα, και κάνει μία κλήση C. Το `Cairo::Context::move_to` διαβάζει τον δείκτη context και δύο doubles και καλεί το `cairo_move_to`· το `Cairo::Context::set_source_rgba` διαβάζει τον δείκτη και τέσσερα doubles και καλεί το `cairo_set_source_rgba`· το `Cairo::Surface::write_to_png` διαβάζει τον δείκτη και μια συμβολοσειρά διαδρομής. Η αντιστοίχιση είναι μία μέθοδος σε μία κλήση C:

```
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)
```

Οι υπογραφές σε επίπεδο C ζουν μέσα στη σύνδεση και ρυθμίζονται εκεί μία φορά. Ένας χρήστης που καλεί `$cr->set_source_rgba(1, 0, 0, 0.5)` δεν γράφει ποτέ υπογραφή και δεν μπορεί να κάνει κάποια λάθος. Αυτό το μοναδικό σημείο αλήθειας για τις κλήσεις C είναι ο τρίτος από τους *τρεις λόγους* για τους οποίους μια επιμελημένη σύνδεση νικά τις διάσπαρτες ακατέργαστες κλήσεις: η διάρκεια ζωής του αντικειμένου, η εργονομία, και η υπογραφή γραμμένη μία φορά.

Τι προσεγγίζει αυτό, και τι όχι

Με αυτά τα τέσσερα κομμάτια (άνοιγμα μία φορά, χειριστής ως blessed ακέραιος, το DESTROY αποδεσμεύει, μέθοδοι πάνω από τον επιλυμένο πίνακα συναρτήσεων) η pperl παραδίδει ένα πραγματικό, ασφαλές ως προς τη μνήμη `Cairo` χωρίς εγκατεστημένη διανομή CPAN, χωρίς μεταγλωττιστή C, και χωρίς βήμα κατασκευής. Ολόκληρο το μοντέλο σχεδίασης του *κεφαλαίου των vector* είναι διαθέσιμο, και τα αντικείμενα διαχειρίζονται τον εαυτό τους.

Αυτό που το πρότυπο δεν προσεγγίζει από μόνο του είναι το ίδιο σύνορο που ονόμασε ο *εισαγωγικός οδηγός*. Το Cairo αποφεύγει όλο αυτό στο πυρηνικό API σχεδίασής του, γι" αυτό είναι η πρώτη βιβλιοθήκη γραφικών που σηκώνεται με αυτόν τον τρόπο. Μια βιβλιοθήκη που περνά χρώματα ή σημεία **κατά τιμή ως structs**, που **καλεί πίσω** τον κώδικά σας, ή που σας δίνει **συμβάντα τύπου union** χρειάζεται περισσότερα από όσα έχει αυτό το πρότυπο: οι περιπτώσεις κατά τιμή και union περιμένουν το *στρώμα περιγραφής δεδομένων*, και τα callbacks χρειάζονται υποστήριξη δεικτών συναρτήσεων που ξαναμπαίνει στην Perl. Το *ολοκληρωτικό έργο του 2D παιχνιδιού* χτυπά ακριβώς σε αυτό το τείχος με τις βιβλιοθήκες παραθύρων και συμβάντων, και το λέει στο σημείο που το κάνει.

Το συμπέρασμα

Μια βιβλιοθήκη γραφικών της οποίας το API της C είναι βαθμωτά συν αδιαφανείς χειριστές μπορεί να σηκωθεί στην pperl ως μια επιμελημένη εγγενής σύνδεση, και το `Cairo` είναι η πρώτη που αποστέλλεται: άνοιγμα της βιβλιοθήκης μία φορά, κράτημα κάθε αντικειμένου C ως blessed ακεραίου, αποδέσμευσή του στο `DESTROY`, και

στήριξη κάθε μεθόδου με την κλήση C της ρυθμισμένη σε ένα σημείο. Το αποτέλεσμα είναι το upstream API του `Cairo`, να τρέχει εγγενώς. Άλλες βιβλιοθήκες του ίδιου σχήματος είναι καθ" οδόν. Το *επόμενο κεφάλαιο* σταματά να περιγράφει και αρχίζει να κατασκευάζει: μια εφαρμογή σχεδίασης πάνω στη συντηρούμενη στοίβα εργαλειοθηκών, συναρμολογώντας ολόκληρο τον οδηγό σε κάτι που τρέχει.

Ολοκληρωτικό έργο: μια εφαρμογή σχεδίασης

Αυτό το κεφάλαιο συναρμολογεί τον οδηγό σε ένα πρόγραμμα που εκτελείτε: μια μικρή εφαρμογή σχεδίασης. Κάντε κλικ και σύρετε για να ζωγραφίσετε, διαλέξτε ένα χρώμα, καθαρίστε τον καμβά, και αποθηκεύστε το αποτέλεσμα σε ένα PNG. Είναι χτισμένη πάνω στο *Prima*, τη συντηρούμενη αυτοτελή εργαλειοθήκη από το κεφάλαιο του GUI, γιατί μας δίνει ένα παράθυρο, συμβάντα ποντικιού, και μια επιφάνεια εκτός οθόνης που αποθηκεύει τον εαυτό της, όλα χωρίς μια εξωτερική εργαλειοθήκη να εγκαταστήσουμε.

Ο σχεδιασμός είναι αυτός που χρησιμοποιεί κάθε πρόγραμμα ζωγραφικής. Υπάρχει μια **εικόνα εκτός οθόνης** που κρατά την πραγματική ζωγραφιά, και ένα **παράθυρο** που την εμφανίζει. Το ποντίκι σχεδιάζει πάνω στην εικόνα εκτός οθόνης· ο χειριστής βαφής του παραθύρου απλώς αντιγράφει αυτή την εικόνα στην οθόνη. Ο διαχωρισμός των δύο σημαίνει ότι η ζωγραφιά επιβιώνει από τις επανασχεδιάσεις του παραθύρου (αλλαγή μεγέθους, επικάλυψη) δωρεάν, και η αποθήκευση είναι τετριμμένη γιατί το πραγματικό σχέδιο ζει ήδη σε ένα αντικείμενο εικόνας.

Το μοντέλο: μια εικόνα εκτός οθόνης

Η ίδια η ζωγραφιά είναι ένα `Prima::Image`, πάνω στο οποίο σχεδιάζουμε με τα ίδια πρωτογενή που θα χρησιμοποιούσατε σε έναν καμβά παραθύρου. Το ξεκινάμε λευκό:

```
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(cl::White);
$canvas->bar(0, 0, $W, $H);          # bar is Prima's filled␣
 ↪rectangle
$canvas->end_paint;

my $brush_color = cl::Black;         # current paint color
my $last;                            # last mouse point while␣
 ↪dragging
```

Τα `begin_paint` και `end_paint` περικλείουν κάθε σχεδίαση πάνω στην εικόνα εκτός οθόνης. Το `bar` γεμίζει ένα ορθογώνιο (θυμηθείτε από το *κεφάλαιο του GUI* ότι το γεμάτο ορθογώνιο του Prima είναι το `bar`, όχι το `rectangle`). Κρατάμε το τρέχον χρώμα του πινέλου και το προηγούμενο σημείο συρσίματος ως συνηθισμένες μεταβλητές της Perl.

Το παράθυρο: εμφάνιση και είσοδος

Η δουλειά του παραθύρου είναι να δείχνει το `$canvas` και να μεταφράζει την κίνηση του ποντικιού σε πινελιές. Ο χειριστής `onPaint` του αντιγράφει την εικόνα εκτός οθόνης στην οθόνη με το `put_image`· οι χειριστές ποντικιού του σχεδιάζουν πάνω στην εικόνα εκτός οθόνης και ζητούν από το παράθυρο να ξαναβαφτεί.

```
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; },
);
```

Η μετατόπιση `y - 40` κρατά μια λωρίδα στο κάτω μέρος του παραθύρου για τα κουμπιά· η περιοχή σχεδίασης είναι όλα όσα βρίσκονται πάνω από αυτήν. Το `onMouseMove` πυροδοτείται συνεχώς, οπότε σχεδιάζουμε ένα σύντομο ευθύγραμμο τμήμα από το προηγούμενο σημείο ως το τρέχον και θυμόμαστε το τρέχον σημείο. Το γρήγορο σύρσιμο παράγει και πάλι μια συνεχή πινελιά γιατί πάντα ενώνουμε διαδοχικά σημεία αντί να στίζουμε μεμονωμένα pixel.

Το πινέλο: σχεδίαση πάνω στην εικόνα εκτός οθόνης

Το `stroke` σχεδιάζει ένα τμήμα του πινέλου πάνω στο εκτός οθόνης `$canvas`. Επειδή η ζωγραφιά είναι ένα κανονικό `Prima::Image`, χρησιμοποιούμε τις ίδιες κλήσεις σχεδίασης όπως οπουδήποτε αλλού, περικλεισμένες από τα `begin_paint` και `end_paint`:

```
sub stroke {
    my ($from, $to) = @_;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    $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;
}
```

Ο ορισμός του `lineWidth` πριν από το `line` δίνει μια παχύτερη πινελιά· το μικρό `fill_ellipse` στο άκρο στρογγυλεύει τις ενώσεις ώστε ένα γρήγορο ζιγκ-ζαγκ να μη δείχνει κενά στις γωνίες. Αυτή είναι η λεπτομέρεια που γειτονεύει με την εξομάλυνση από το *κεφάλαιο των αλγορίθμων* να εμφανίζεται στην πράξη: οι παχιές γραμμές που συναντιούνται σε γωνία αφήνουν μια εγκοπή εκτός αν στρογγυλέψετε την ένωση.

Χειριστήρια: χρώμα, καθαρισμός και αποθήκευση

Μια σειρά κουμπιών κατά μήκος του κάτω μέρους δίνει στο εργαλείο τα ρήματά του. Το καθένα είναι ένα `Prima::Button` με ένα `onClick`. Τα κουμπιά χρώματος ορίζουν το `$brush_color`· το clear ξαναβάφει την εικόνα εκτός οθόνης λευκή· το save τη γράφει στον δίσκο.

```
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;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
$window->insert(Button =>
    text => 'Save', origin => [$x, 5], size => [60, 30],
    onClick => sub {
        $canvas->save('painting.png')
            and print "saved painting.png\n";
    },
);

Prima->run;
```

Το `insert(Button => ...)` προσθέτει ένα widget-παιδί στο παράθυρο. Το `$canvas->save` γράφει την εικόνα εκτός οθόνης σε ένα PNG, με τη μορφή να συμπεραίνεται από την επέκταση, επιστρέφοντας αληθές σε επιτυχία. Επειδή το σχέδιο ζει σε ένα αντικείμενο εικόνας όλη την ώρα, η αποθήκευση είναι μία μόνο κλήση χωρίς τίποτα να ανακατασκευαστεί.

Εκτελώντας την

Συναρμολογήστε τα παραπάνω κομμάτια σε ένα αρχείο και εκτελέστε το με το `pperl`. Ανοίγει ένα παράθυρο με έναν λευκό καμβά και μια λωρίδα κουμπιών· σύρετε για να ζωγραφίσετε, κάντε κλικ σε ένα χρώμα για να αλλάξετε, Clear για επαναφορά, Save για να γράψετε το `painting.png`. Κάθε κομμάτι ανάγεται πίσω στον οδηγό: η εικόνα εκτός οθόνης και τα πρωτογενή της από το *κεφάλαιο του GUI*, η ένωση πινελιών από το *κεφάλαιο των αλγορίθμων*, το μοντέλο χρώματος από τα *θεμέλια*, και η διαδρομή αποθήκευσης που είναι μία μέθοδος γιατί η ζωγραφιά ήταν πάντα μια πραγματική εικόνα.

Πού θα έμπαινε το νήμα του FFI

Αυτό το ολοκληρωτικό έργο παραμένει στη συντηρούμενη στοίβα του CPAN, και επίτηδες: προορίζεται να τρέξει σήμερα, χωρίς τροποποιήσεις. Το *νήμα του FFI* μπαίνει όταν ξεπερνάτε αυτό που προσφέρει η στοίβα, για παράδειγμα όταν θέλετε μια βιβλιοθήκη μηχανής πινέλου σε C χωρίς σύνδεση Perl για πινελιές ευαίσθητες στην πίεση. Αυτό προσεγγίζεται ακριβώς όπως έδειξε η *επίδειξη του Cairo*: μια επιμελημένη σύνδεση που παρουσιάζει τη βιβλιοθήκη C ως αντικείμενα της Perl. Για ένα πρόγραμμα ζωγραφικής, τα ίδια τα πρωτογενή του Prima είναι υπεραρκετά, και η ειλικρινής μηχανική επιλογή είναι να τα χρησιμοποιήσετε. Το *επόμενο ολοκληρωτικό έργο* είναι εκεί όπου το σύνορο του FFI έρχεται πραγματικά στο προσκήνιο, γιατί τα παράθυρα και τα συμβάντα πραγματικού χρόνου σπρώχνουν πέρα από αυτά που προσεγγίζουν οι τρέχουσες συνδέσεις.

Ολοκληρωτικό έργο: ένα 2D παιχνίδι

Το δεύτερο ολοκληρωτικό έργο είναι ένα μικρό αλλά πλήρες 2D παιχνίδι: μια ρακέτα, μια μπάλα, ένα σκορ, ένας βρόχος παιχνιδιού που τρέχει με σταθερό ρυθμό καρέ, και έλεγχος από το πληκτρολόγιο. Είναι χτισμένο πάλι πάνω στο *Prima*, γιατί το Prima μας δίνει τα τρία πράγματα που χρειάζεται ένα παιχνίδι (έναν χρονομετρητή για τον βρόχο, συμβάντα πληκτρολογίου, και γρήγορη σχεδίαση) σε ένα συντηρούμενο πακέτο που τρέχει σήμερα. Στο τέλος του κεφαλαίου κοιτάζουμε ειλικρινά την εναλλακτική, μια

αποκλειστική βιβλιοθήκη παιχνιδιών όπως το SDL2, και γιατί το να την προσεγγίσει κανείς καλά είναι ακόμη δουλειά συνόρου για το FFI της pperl.

Ένα παιχνίδι είναι ένας βρόχος: διάβασε την είσοδο, ενημέρωσε τον κόσμο, σχεδίασε τον κόσμο, περίμενε το επόμενο καρέ, επανάλαβε. Όλα τα υπόλοιπα είναι λεπτομέρεια. Χτίζουμε αυτόν τον βρόχο πρώτα, και έπειτα κρεμάμε ένα παιχνίδι επάνω του.

Ο βρόχος παιχνιδιού: ένας χρονομετρητής

Ένας βρόχος πραγματικού χρόνου χρειάζεται έναν παλμό. Το Prima παρέχει το `Prima::Timer`, που πυροδοτεί έναν χειριστή `onTick` σε ένα διάστημα· ρυθμίζοντάς το σε περίπου 16 χιλιοστά του δευτερολέπτου δίνει περίπου 60 καρέ ανά δευτερόλεπτο. Κάθε τικ ενημερώνει τον κόσμο και ζητά από το παράθυρο να ξαναβαφτεί.

```
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 },
    paddle => { x => 200, w => 80, h => 12 },
    score  => 0,
    over    => 0,
);
```

Ολόκληρος ο κόσμος του παιχνιδιού είναι ένα απλό hash. Αυτό είναι σκόπιμο: η λογική ενημέρωσης είναι συνηθισμένη Perl πάνω σε συνηθισμένα δεδομένα, και η σχεδίαση διαβάζει τα ίδια δεδομένα. Τίποτα στον βρόχο δεν είναι ειδικό για γραφικά μέχρι το βήμα της σχεδίασης.

Ενημέρωση: μετακίνηση του κόσμου

Κάθε τικ προωθεί την μπάλα, την αναπηδά από τους τοίχους και τη ρακέτα, και τερματίζει το παιχνίδι αν η μπάλα προσπεράσει τη ρακέτα. Αυτό είναι καθαρή αριθμητική πάνω στο hash του κόσμου, καμία σχεδίαση:

```
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;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    # 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;
}
```

Ο έλεγχος σύγκρουσης είναι η ευθυγραμμισμένη με τους άξονες επικάλυψη από το σύστημα συντεταγμένων των *θεμελίων*: η μπάλα έχει διασχίσει τη γραμμή της ρακέτας και βρίσκεται μέσα στο οριζόντιο εύρος της. Η αναπήδηση είναι η άρνηση μιας συνιστώσας της ταχύτητας. Ένα ολόκληρο είδος παιχνιδιών είναι αυτό και όχι πολλά περισσότερα.

Σχεδίαση: απόδοση του κόσμου

Η σχεδίαση διαβάζει το hash του κόσμου και το ζωγραφίζει. Το Prima καλεί το `onPaint` με έναν καμβά· τον καθαρίζουμε, και έπειτα σχεδιάζουμε τη ρακέτα, την μπάλα και το σκορ χρησιμοποιώντας τα πρωτογενή από το *κεφάλαιο του GUI*:

```
sub draw {
    my ($canvas) = @_;
    $canvas->color(cl::Black);
    $canvas->bar(0, 0, $W, $H);              # clear to black

    my $p = $game{paddle};
    $canvas->color(cl::White);
    $canvas->bar($p->{x}, $H - 30, $p->{x} + $p->{w}, $H - 30 + $p->
↪{h});

    my $b = $game{ball};
    $canvas->color(cl::Yellow);
    $canvas->fill_ellipse($b->{x}, $b->{y}, $b->{r} * 2, $b->{r} *␣
↪2);

    $canvas->color(cl::White);
    $canvas->text_out("Score: $game{score}", 10, $H - 20);
    $canvas->text_out('GAME OVER', $W / 2 - 40, $H / 2) if $game
↪{over};
}
```

Το `text_out` σχεδιάζει μια συμβολοσειρά· τα `fill_ellipse` και `bar` είναι η μπάλα και η ρακέτα. Ο καθαρισμός σε μαύρο κάθε καρέ και η επανασχεδίαση των πάντων είναι το στυλ immediate-mode: η οθόνη είναι μια συνάρτηση της κατάστασης του κόσμου, επανυπολογισμένη σε κάθε καρέ, που είναι απλό στη συλλογιστική και αρκετά γρήγορο

για ένα παιχνίδι αυτού του μεγέθους.

Είσοδος: το πληκτρολόγιο

Η ρακέτα ακολουθεί τα πλήκτρα βέλους. Το Prima παραδίδει τα πατήματα πλήκτρων μέσω ενός χειριστή onKeyDown, τον οποίο χρησιμοποιούμε για να ολισθαίνουμε τη ρακέτα αριστερά και δεξιά, περικομμένη μέσα στο παράθυρο:

```
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};
}
```

Οι σταθερές `kb::Left` και `kb::Right` ονομάζουν τα πλήκτρα βέλους. Η περικοπή κρατά τη ρακέτα στην οθόνη. Για ομαλή συνεχή κίνηση ένα παιχνίδι συχνά παρακολουθεί ποια πλήκτρα κρατιούνται αυτή τη στιγμή και κινείται στο βήμα ενημέρωσης· ένα βήμα ανά πάτημα είναι απλούστερο και μια χαρά εδώ.

Συνδέοντάς τα μεταξύ τους

Το παράθυρο δένει τους τρεις χειριστές στον βρόχο. Ο χρονομετρητής οδηγεί την ενημέρωση και την επαναβαφή· ο χειριστής βαφής σχεδιάζει· ο χειριστής πλήκτρων κατευθύνει:

```
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;
```

Συναρμολογήστε τα κομμάτια σε ένα αρχείο και εκτελέστε το με το `pperl`. Ανοίγει ένα παράθυρο με μια μπάλα που αναπηδά και μια ρακέτα που κατευθύνετε με τα πλήκτρα βέλους· κάθε επιτυχημένη αναπήδηση βαθμολογεί έναν πόντο, και η αστοχία τερματίζει το παιχνίδι. Κάθε μέρος ανάγεται μέσα από τον οδηγό: ο βρόχος είναι ένας χρονομετρητής, ο κόσμος είναι απλά δεδομένα, η ενημέρωση είναι αριθμητική πάνω σε συντεταγμένες από τα *θεμέλια*, και το βήμα σχεδίασης είναι *πρωτογενή του Prima*.

Το σύνορο: το SDL2 και οι αποκλειστικές βιβλιοθήκες παιχνιδιών

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 pperl stands.

Η ιστορία του SDL στην Perl είναι ισχνή. Η παλιά διανομή SDL συνδέει το SDL 1.2, που είναι στο τέλος του κύκλου ζωής του, και είναι η ίδια ουσιαστικά ασυντήρητη. Δεν υπάρχει στέρεη, συντηρούμενη σύνδεση SDL2 στο CPAN· η μία απόπειρα βασισμένη σε FFI είναι επιπέδου alpha, με μια ελλιπή σουίτα δοκιμών και ασταθές threading. Έτσι το SDL2 είναι ακριβώς το είδος μοντέρνας βιβλιοθήκης C που υπάρχει το *νήμα του FFI* για να προσεγγίσει, και είναι επίσης ακριβώς εκεί που δαγκώνει το τρέχον σύνορο του FFI.

Ένα μέρος του SDL2 είναι προσεγγίσιμο με ακατέργαστες κλήσεις `Peta::FFI` σήμερα. Η δημιουργία ενός παραθύρου και ενός renderer επιστρέφει αδιαφανείς χειριστές (δείκτες, άρα ακεραίους), και πολλές κλήσεις παίρνουν μόνο βαθμωτά:

```
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",
               "pperl", 100, 100, 480, 360, 4);      # returns a␣
↪handle
# ... and so on for the renderer, using the returned handle
```

Αυτά λειτουργούν: τα βαθμωτά ορίσματα, τα ορίσματα συμβολοσειρών, και οι αδιαφανείς χειριστές που επιστρέφονται ως ακέραιοι είναι ακριβώς αυτά που καλύπτουν οι ακατέργαστοι κωδικοί τύπων. Αλλά ένας πραγματικός βρόχος παιχνιδιού χρειάζεται το μέρος που δεν λειτουργεί ακόμη. Το `SDL_PollEvent` γεμίζει ένα `SDL_Event`, που είναι ένα **union** από πολλά σχήματα struct, και οι ακατέργαστοι κωδικοί τύπων δεν μπορούν να περιγράψουν ένα struct, πόσο μάλλον ένα union. Η ανάγνωση του ποιο πλήκτρο πατήθηκε σημαίνει αποκωδικοποίηση των bytes εκείνου του struct, που είναι αυτό που κατασκευάζεται να κάνει το *στρώμα περιγραφής δεδομένων του Επιπέδου 2*. Και μια πλήρως ιδιωματική σύνδεση θα ήθελε callbacks, τα οποία το ακατέργαστο στρώμα επίσης δεν προσεγγίζει.

Έτσι η ειλικρινής θέση είναι η εξής: μια **πλήρης, άνετη** σύνδεση SDL2 στην pperl είναι δουλειά συνόρου, που περιμένει την υποστήριξη struct και callback του FFI. Είναι προσεγγίσιμη, είναι στον οδικό χάρτη, και είναι ακριβώς το είδος σύνδεσης που θα παραγάγει το πρότυπο της *επίδειξης του Cairo* μόλις προσγειωθεί το στρώμα περιγραφής δεδομένων. Μέχρι τότε, το Prima είναι η ειλικρινής, εκτελέσιμη απάντηση του ολοκληρωτικού έργου, και η διαδρομή του SDL2 είναι μια πραγματική αλλά μερική διαδρομή σαφώς επισημασμένη ως σύνορο.

Πού τελειώνει ο οδηγός

Έχετε τώρα χτίσει δύο προγράμματα που τρέχουν: μια εφαρμογή σχεδίασης και ένα παιχνίδι, και τα δύο πάνω στη συντηρούμενη στοίβα εργαλειοθηκών, με το νήμα του FFI υφασμένο εκεί όπου κερδίζει τη θέση του και επισημασμένο εκεί όπου ο δρόμος τελειώνει προς το παρόν. Το *τελευταίο κεφάλαιο* σχεδιάζει αυτόν τον χάρτη ρητά: τι

θα πρόσθετε μια δεύτερη έκδοση, και ποια σύνορα, με την τρίτη διάσταση επικεφαλής μεταξύ τους, αφήνονται σκόπιμα για τη συνέχεια.

Γνωστά κενά και τι ακολουθεί

Αυτός ο οδηγός κάλυψε δύο διαστάσεις διεξοδικά και σταμάτησε εκεί σκόπιμα. Αυτό το καταληκτικό κεφάλαιο σχεδιάζει το όριο ρητά: τι θα πρόσθετε μια δεύτερη έκδοση, τι ανήκει σε ξεχωριστούς οδηγούς, και πού τελειώνει προς το παρόν το σύνορο του FFI. Η ονομασία αυτών σας κρατά από το να ψάχνετε για κάλυψη που δεν είναι εδώ, και χαράζει τον δρόμο μπροστά.

Η τρίτη διάσταση (προγραμματισμένη v2)

Καθετί τρισδιάστατο αναβάλλεται για μια συνέχεια, δεν παραλείπεται κατά λάθος. Το *κεφάλαιο των θεμελίων* ονόμασε τη μία έννοια που μεταφέρεται (τον Z-buffer, που αποφασίζει τι είναι πλησιέστερα στον θεατή), και το *κεφάλαιο των αλγορίθμων* σημείωσε ότι η τεσελίωση σε τρίγωνα είναι η γέφυρα προς το 3D, αλλά κανένα δεν πήγε παρακάτω. Μια δεύτερη έκδοση θα κάλυπτε:

- Το **OpenGL**, μέσω μιας συντηρούμενης σύνδεσης, για απόδοση με επιτάχυνση υλικού: τη μοντέρνα σωλήνωση shader, τους ενταμιευτές κορυφών και δεικτών, και τα μαθηματικά (προβολή, ο μετασχηματισμός model-view) που μετατρέπουν μια σκηνή 3D σε μια εικόνα 2D.
- Τον **ενταμιευτή βάθους στα αλήθεια**, όχι απλώς την έννοια: πώς η σειρά σχεδίασης παύει να έχει σημασία και το βάθος αποφασίζει την ορατότητα.
- Ένα **ολοκληρωτικό έργο 3D παιχνιδιού**, το τρίτο έργο που ονόμασε το αρχικό σχέδιο, χτισμένο πάνω σε όποια διαδρομή 3D αποδειχθεί πιο ειλικρινής να τρέξει εκείνη τη στιγμή.

Το 3D είναι ένα μεγάλο θέμα και το να του αποδοθεί δικαιοσύνη χρειάζεται τον δικό του άξονα· η δίπλωση μιας ισχνής έκδοσης μέσα σε αυτόν τον οδηγό θα είχε αδικήσει και τις δύο διαστάσεις.

Σκόπιμα εκτός πεδίου

Τρεις περιοχές εξαιρούνται σκόπιμα και θα ήταν καθεμία ένας ξεχωριστός οδηγός, όχι ένα κεφάλαιο εδώ:

- **Προγραμματισμός GPU.** Η συγγραφή shaders υπολογισμού ή γραφικών και η εκτέλεση εργασιών γενικού σκοπού στη GPU είναι ένας διακριτός κλάδος με τη δική του εργαλειοαλυσίδα. Αγγίζει τα γραφικά αλλά δεν είναι σχεδίαση, και κερδίζει τη δική του μεταχείριση.
- **Υπολογιστική όραση και ανάλυση εικόνας.** Αυτός ο οδηγός χρησιμοποίησε τη *συνέλιξη* ως εργαλείο σχεδίασης (θόλωμα, όξυνση) και σταμάτησε ακριβώς στη γραμμή όπου τα ίδια μαθηματικά γίνονται ανίχνευση και αναγνώριση. Η ανίχνευση χαρακτηριστικών, η κατάτμηση και η αναγνώριση είναι ανάλυση, όχι σχεδίαση, και ανήκουν αλλού.
- **Φιλοξενούμενες υπηρεσίες παραγωγής εικόνων.** Οι συνδέτες προς εξωτερικά API παραγωγής εικόνων είναι δουλειά δικτύου και ενσωμάτωσης υπηρεσιών, όχι προγραμματισμός γραφικών, και είναι εκτός πεδίου ανεξάρτητα από τη διάσταση.

Το σύνορο του FFI

Ο *άξονας του «φτάνοντας πιο μακριά»* ήταν ειλικρινής σε κάθε βήμα για το πού προσεγγίζει προς το παρόν το εγγενές FFI της pperl. Συγκεντρωμένο εδώ, το σύνορο είναι:

- **Το στρώμα περιγραφής δεδομένων του Επιπέδου 2.** Οι ακατέργαστες κλήσεις `Peta::FFI` χειρίζονται βαθμωτά, συμβολοσειρές, ενταμιευτές και αδιαφανείς χειριστές, που καλύπτει ένα μεγάλο μέρος πολλών API γραφικών σε C και όλο ένα library όπως το *Cairo*. Αυτό που δεν περιγράφουν ακόμη είναι ένα **struct** που περνιέται ή επιστρέφεται κατά τιμή, ή ένα **union** όπως μια εγγραφή συμβάντος παραθύρων. Το δηλωτικό στρώμα περιγραφής δεδομένων κατασκευάζεται για να κλείσει ακριβώς αυτό το κενό, και είναι το κομμάτι που λείπει ανάμεσα στις σημερινές ακατέργαστες κλήσεις και σε μια άνετη σύνδεση για βιβλιοθήκες με πολλά structs.
- **Callbacks.** Μια βιβλιοθήκη που καλεί πίσω τον κώδικά σας (ένας βρόχος συμβάντων που επικαλείται έναν χειριστή, ένα σήμα σχεδίασης) χρειάζεται έναν δείκτη συνάρτησης C που ξαναμπαίνει στην Perl. Αυτό είναι πέρα από το ακατέργαστο στρώμα σήμερα.
- **Μια πλήρης σύνδεση SDL2.** Το *ολοκληρωτικό έργο του παιχνιδιού* έδειξε ότι το SDL2 είναι εν μέρει προσεγγίσιμο τώρα (δημιουργία παραθύρου και renderer, οι βαθμωτές κλήσεις) και μπλοκαρισμένο στα δύο παραπάνω στοιχεία για τα υπόλοιπα (τα union συμβάντων, και ιδανικά τα callbacks). Είναι η φυσική πρώτη μεγάλη επιμελημένη σύνδεση γραφικών μόλις προσγειωθεί το στρώμα περιγραφής δεδομένων, ακολουθώντας το πρότυπο της *επίδειξης του Cairo*.

Κανένα από αυτά δεν είναι τείχος. Είναι η σειρά με την οποία μεγαλώνει το FFI, και η δουλειά αυτού του οδηγού ήταν να σας πει σε ποια πλευρά της τρέχουσας γραμμής κάθεται κάθε εργασία.

Τι μπορείτε να κάνετε τώρα

Σε αντιπαραβολή με το σύνορο, το έδαφος που έχει ήδη καλυφθεί είναι ευρύ. Με αυτόν τον οδηγό μπορείτε, σήμερα, στην pperl:

- Να σχεδιάζετε εικόνες raster με πρωτογενή και φίλτρα, και να τις διαβάζετε, να τις κλιμακώνετε και να τις αποθηκεύετε (*Imager*, *GD και ImageMagick*).
- Να παραγάγετε έξοδο vector ανεξάρτητη από την ανάλυση, ως pixel μέσω του Cairo ή ως markup μέσω του SVG (*Cairo και SVG*).
- Να διαβάζετε και να γράφετε μεταδεδομένα εικόνας, και να σχεδιάζετε γραφήματα (*μεταδεδομένα και γραφήματα*).
- Να ανοίγετε ένα παράθυρο, να ανταποκρίνεστε στο ποντίκι και στο πληκτρολόγιο, και να κινείτε (*εργαλειοθήκες GUI*).
- Να προσεγγίζετε μια βιβλιοθήκη γραφικών C με βαθμωτά και χειριστές μέσω μιας επιμελημένης εγγενούς σύνδεσης, χωρίς μεταγλωττιστή C και χωρίς βήμα κατασκευής (*επίδειξη του Cairo*).
- Να χτίζετε και να εκτελείτε πραγματικά προγράμματα: μια *εφαρμογή σχεδίασης* και ένα *2D παιχνίδι*.

Αυτή είναι μια πλήρης δυνατότητα 2D γραφικών στην Perl, από το pixel ως ένα πρόγραμμα που τρέχει. Η τρίτη διάσταση, η GPU και η πλευρά της όρασης είναι οι δρόμοι που φεύγουν από εδώ, και βρίσκονται στον χάρτη.

4.11 pack και unpack - ένας οδηγός εκμάθησης

Οι `pack` και `unpack` είναι οι σειριοποιητές χαμηλού επιπέδου της Perl. Μετατρέπουν μεταξύ τιμών της Perl και των ωμών συμβολοσειρών bytes που τα δυαδικά πρότυπα - πρωτόκολλα δικτύου, headers αρχείων, δομές C, εγγραφές σταθερού πλάτους - μεταφέρουν πραγματικά επί της γραμμής.

Οι περισσότεροι προγραμματιστές της Perl τις συναντούν σπάνια και τις ξεχνούν γρήγορα. Αυτός ο οδηγός εκμάθησης υπάρχει ώστε όταν τις συναντήσετε, να γνωρίζετε ήδη το σχήμα του προβλήματος.

4.11.1 Πότε χρειάζεστε τις pack και unpack

Καταφύγετε σε αυτές όταν:

- **Μιλάτε ένα πρωτόκολλο επί της γραμμής** - ερωτήματα DNS, headers TCP/IP, πλαίσια MQTT, οτιδήποτε έχει «big-endian 16-bit length» στις προδιαγραφές του.
- **Αναλύετε ένα δυαδικό πρότυπο αρχείου** - PNG, GIF, WAV, ELF, GZip, Intel HEX, όλα τα πρότυπα των οποίων τα πρώτα τέσσερα bytes είναι ένας μαγικός αριθμός ακολουθούμενος από γνωστά ακέραια πεδία.
- **Τεμαχίζετε εγγραφές κειμένου σταθερού πλάτους** - αρχεία καταγραφής, λογιστικά κατάστιχα, αναφορές από mainframes. Η `unpack` είναι πιο καθαρή από αλυσιδωτές κλήσεις `substr` και χειρίζεται την αμφισημία κενών πεδίων που η `split` καταρρέει.
- **Περνάτε δεδομένα σε μια κλήση συστήματος** - οι `syscall`, `ioctl`, και ο σπάνιος κατασκευαστής `sockaddr_*` αναμένουν bytes διατεταγμένα ως struct της C.

Για σειριοποίηση κειμένου (JSON, YAML, CSV, XML), καταφύγετε στο κατάλληλο άρθρωμα. Η `pack` είναι για δυαδικά.

4.11.2 Πώς είναι οργανωμένος αυτός ο οδηγός εκμάθησης

Κάθε κεφάλαιο διδάσκει ένα θέμα και στέκεται αυτόνομα. Μπορείτε να τα διαβάσετε με τη σειρά, ή να μεταπηδήσετε σε αυτό που απαντά στο ερώτημά σας.

Bytes και πλάτη

Στο τέλος αυτού του κεφαλαίου θα μπορείτε να επιλέγετε τη σωστή οδηγία ακεραίων για ένα δεδομένο πεδίο, να μετράτε τα bytes που παράγει ένα πρότυπο, και να ξεχωρίζετε πότε χρειάζεστε σταθερό πλάτος έναντι native πλάτους.

Τα δυαδικά πρότυπα ορίζονται με βάση *μετρητές bytes*. Το «ένα μη προσημασμένο πεδίο μήκους 16-bit ακολουθούμενο από πακέτα 4-byte» είναι μια οδηγία για bytes, όχι για αριθμούς. Η πρώτη δουλειά ενός προτύπου `pack` είναι να ταιριάξει με αυτούς τους μετρητές ακριβώς.

Οι οδηγίες ακεραίων σταθερού πλάτους

Η ραχοκοκαλιά κάθε προτύπου:

Bytes	Μη προσημασμένος	Προσημασμένο
1	C	c
2	S	s
4	L	l
8	Q	q

Κάθε οδηγία καταναλώνει μία τιμή από τη λίστα και παράγει ακριβώς τόσα 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)
```

Ένας μετρητής επανάληψης πακετάρει τόσες επαναλήψεις της ίδιας οδηγίας στη σειρά:

```
length pack "C4", 1, 2, 3, 4        # 4
length pack "L3", 10, 20, 30        # 12
```

Συνδυάστε τις ελεύθερα· το συνολικό μήκος είναι το άθροισμα:

```
length pack "C S L", 65, 1000, 2_000_000   # 1 + 2 + 4 = 7
```

Το C είναι η προεπιλεγμένη οδηγία ενός byte

Για bytes πρωτοκόλλου - πεδία σημαιών, αριθμούς εκδόσεων, opcodes, ετικέτες τύπου - το `C` (μη προσημασμένο 8-bit, από 0 έως 255) είναι σχεδόν πάντα η σωστή επιλογή. Το `c` (προσημασμένο, από −128 έως 127) εμφανίζεται μόνο όταν οι προδιαγραφές λένε «signed». Το `W` είναι παραλλαγή του `C` που επιτρέπει τιμές πάνω από 255 όταν το πρότυπό σας βρίσκεται σε λειτουργία `U0` (UTF-8) - αγνοήστε το μέχρι να συναντήσετε αυτή την περίπτωση.

Οι διευθύνσεις IP είναι το σχολικό παράδειγμα:

```
my $raw = pack "C4", 192, 168, 1, 42;
#         ^-- four unsigned bytes, in the order the list gives them
```

Native έναντι σταθερού πλάτους

Οι οδηγίες `s` / `S` / `l` / `L` ορίζονται να είναι ακριβώς 2 και 4 bytes ανεξάρτητα από τον μεταγλωττιστή C. Τα native-width αντίστοιχά τους προσθέτουν `!`:

```
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.
```

Τα `i` και `I` είναι *πάντα* native (ό,τι επιστρέφει το `sizeof(int)` σε αυτή τη μηχανή, με ελάχιστο τα 32-bit). Το `i!` είναι ψευδώνυμο του `i`.

Πρακτικός κανόνας:

- Για διαλειτουργικά δυαδικά (μορφές επί της γραμμής, αρχεία σε δίσκο, δεδομένα που διασχίζουν μηχανές) χρησιμοποιήστε **σταθερό πλάτος** (`l`, `L`, ...) συν ρητό endianness - ή τις *φορητές συντομεύσεις*.
- Για εργασία εντός διεργασίας που ταιριάζει σε κάποιο τοπικό struct της C που πρόκειται να παραδώσετε στο `ioctl` ή στο `syscall`, χρησιμοποιήστε **native πλάτος** (`l!`, `L!`, `i`, `I`).

Η μείξη των δύο είναι αποδεκτή, αλλά σταματήστε και ρωτήστε τον εαυτό σας γιατί κάθε φορά.

Μέτρηση του μήκους ενός προτύπου

Πριν στείλετε μια εγγραφή, ελέγξτε ότι ο μετρητής bytes είναι αυτός που λένε οι προδιαγραφές. Η `length pack(TEMPLATE, ...)` με placeholder τιμές είναι ο απλούστερος τρόπος:

```
my $sz = length pack "n N L C", 0, 0, 0, 0;
# 2 + 4 + 4 + 1 = 11
```

Για πρότυπα των οποίων το μήκος εξαρτάται από native μεγέθη, η `length pack TEMPLATE, 0, 0, ...` σας δίνει την απάντηση στην τρέχουσα μηχανή.

Επεξεργασμένο παράδειγμα: ένα ελάχιστο header πακέτου

Ένα υποθετικό πρωτόκολλο ορίζει:

- έκδοση 8-bit
- σημαίες 8-bit
- μήκος 16-bit big-endian (ακολουθεί φορτίο)

Τρία πεδία, τέσσερα bytes συνολικά. Το πρότυπο γράφεται από μόνο του:

```
my $hdr = pack "C C n", $version, $flags, length $payload;
my $pkt = $hdr . $payload;

length $hdr    # 4
```

Δύο πράγματα προς προσοχή:

1. Δεν γράψαμε αναλυτικά `C1 C1 n1` - ένα γυμνό γράμμα οδηγίας έχει υπονοούμενο μετρητή επανάληψης 1.
2. Το μήκος στο τρίτο πεδίο υπολογίζεται από το `$payload` στην ίδια έκφραση. Ένα μεταγενέστερο κεφάλαιο δείχνει τη *μορφή* `/` που συνδέει τα δύο μεταξύ τους ώστε να μη μπορείτε να ξεχάσετε.

Οριακές περιπτώσεις άξιες να θυμάστε

- **Τιμές εκτός εύρους**: η `pack "C", 256` αποκόπτει σιωπηλά (παίρνετε `\x00`). Δεν υπάρχει προειδοποίηση. Ελέγξτε τα εύρη οι ίδιοι αν τα δεδομένα δεν είναι αξιόπιστα.
- **Πλήρης κύκλος προσημασμένου έναντι μη προσημασμένου**: μια τιμή πακεταρισμένη ως `C` και αποπακεταρισμένη ως `c` (ή το αντίστροφο) αλλάζει πρόσημο για τιμές ⩾ 128:

```
unpack "c", pack "C", 200        # -56
```

- **Πολύ λίγες τιμές**: οι τιμές που λείπουν πακετάρονται ως `0` για αριθμητικές οδηγίες, `""` για συμβολοσειρές. Καμία προειδοποίηση.

Το επόμενο κεφάλαιο χειρίζεται το ερώτημα που εγείρει κάθε ακέραιος πολλαπλών bytes: ποια άκρη του αριθμού πάει πρώτη;

Endianness

Στο τέλος αυτού του κεφαλαίου θα μπορείτε να επιλέγετε μεταξύ native, big-endian, και little-endian σειράς bytes· να χρησιμοποιείτε τις φορητές συντομεύσεις `n` / `N` / `v` / `V`· και να εφαρμόζετε τους τροποποιητές `<` / `>` σε οποιονδήποτε ακέραιο ή float.

Ένας ακέραιος πολλαπλών bytes πρέπει να γραφεί κάπως. Το `0x12345678` είναι τέσσερα bytes - αλλά με ποια σειρά; Δύο συμβάσεις κυριαρχούν:

- **Big-endian** - το byte υψηλότερης τάξης πρώτο: `12 34 56 78`. Χρησιμοποιείται από τα περισσότερα πρωτόκολλα δικτύου, από τις CPU PowerPC και SPARC, και από πρότυπα αρχείων που νοιάζονται για τη φορητότητα (PNG, Java `.class`, τα περισσότερα headers TCP/IP).
- **Little-endian** - το byte χαμηλότερης τάξης πρώτο: `78 56 34 12`. Χρησιμοποιείται από x86, x86-64, ARM (στις συνηθισμένες λειτουργίες του), και από πρότυπα αρχείων από τη γενιά της Intel (BMP, WAV, TIFF στη συνηθισμένη παραλλαγή του, ολόκληρο το FAT).

Μπερδέψτε τα και θα διαβάσετε ασυναρτησίες. Η `pack` σας δίνει τρεις τρόπους να καθηλώσετε τη σειρά bytes.

Οι φορητές συντομεύσεις: n / N / v / V

Τέσσερα γράμματα καλύπτουν τις πιο συνηθισμένες περιπτώσεις μορφής επί της γραμμής:

Οδηγία	Bytes	Endian	Σημασία
n	2	big-endian	Δίκτυο (αριθμοί θυρών)
N	4	big-endian	Δίκτυο (διευθύνσεις IPv4)
v	2	little-endian	«VAX»
V	4	little-endian	«VAX»

Η «σειρά bytes δικτύου» είναι big-endian. Αν διαβάζετε ένα RFC, η απάντηση είναι σχεδόν πάντα `n` ή `N`.

```
my $port = pack "n", 443;          # "\x01\xbb"
my $ip   = pack "N", 0x7F000001;   # "\x7f\x00\x00\x01" - 127.0.0.1
```

Αυτά τα τέσσερα είναι μόνο μη προσημασμένα. Με τον τροποποιητή ! γίνονται προσημασμένα:

```
my $neg = pack "n!", -1;            # "\xff\xff"
```

Αν το πεδίο σας ταιριάζει σε ένα από αυτά τα τέσσερα σχήματα, χρησιμοποιήστε τα. Είναι φορητά, είναι ευανάγνωστα, και η σημασία τους δεν εξαρτάται από τη CPU που τρέχει τον κώδικα.

Οι γενικοί τροποποιητές: < και >

Οποιαδήποτε οδηγία ακεραίου (`s`, `S`, `l`, `L`, `i`, `I`, `q`, `Q`, `j`, `J`) και κάθε float (`f`, `d`, `F`, `D`) δέχεται τροποποιητή endianness:

Τροποποιητής	Σημασία	Μνημονικό
>	Big-endian	Η «μεγάλη άκρη» αγγίζει το γράμμα της οδηγίας
<	Little-endian	Η «μικρή άκρη» αγγίζει την οδηγία

Διαβάστε το `l>` ως «long, μεγάλη άκρη» και το `l<` ως «long, μικρή άκρη».

```
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
```

Για τις συνηθισμένες περιπτώσεις μη προσημασμένων 16- και 32-bit, τα `n` / `N` / `v` / `V` είναι συντομότερα και προτιμώνται. Χρησιμοποιήστε τα < / > όταν:

- το πεδίο είναι προσημασμένο - τα `n` και `N` είναι μη προσημασμένα από προεπιλογή
- το πεδίο είναι 64-bit - δεν υπάρχουν αντίστοιχα `n`/`N` για τα `q`/`Q`
- το πεδίο είναι float - τα `n`/`N` καλύπτουν μόνο ακεραίους
- θέλετε να εφαρμόσετε endianness σε ολόκληρη μια ομάδα (δείτε παρακάτω)

Εφαρμογή endianness σε ομάδα

Αν κάθε ακέραιος σε μια υπο-δομή μοιράζεται την ίδια σειρά bytes, γράψτε τον τροποποιητή endianness μία φορά σε μια ομάδα:

```
my $rec = pack "(s l l)<", $flags, $x, $y;
# same as "s<l<l<"
```

Ένας τροποποιητής σε επίπεδο ομάδας διαχέεται σε κάθε οδηγία με σειρά bytes που βρίσκεται μέσα, συμπεριλαμβανομένων των εμφωλευμένων ομάδων. Οι οδηγίες που δεν δέχονται τροποποιητή σειράς bytes (όπως η `C`) αγνοούνται σιωπηλά.

Native σειρά: όταν *είναι* αυτό που θέλετε

Οι γυμνές οδηγίες `s` / `S` / `l` / `L` / `i` / `I` / `q` / `Q` παράγουν τη native σειρά bytes της CPU του υπολογιστή. Αυτό είναι χρήσιμο όταν:

- Κατασκευάζετε ένα struct της C στη μνήμη για μια κλήση `ioctl` στην *ίδια* μηχανή.
- Γράφετε ένα πρόχειρο αρχείο που μόνο αυτό το πρόγραμμα θα διαβάσει πίσω.
- Οι προδιαγραφές του πρωτοκόλλου λένε ρητά «host byte order» (σπάνιο).

Είναι **λάθος** όταν τα δεδομένα φεύγουν από τη διεργασία: ένα αρχείο που γράφεται σε laptop και διαβάζεται σε διακομιστή δεν πρέπει να χρησιμοποιεί native σειρά εκτός αν συμβαίνει και οι δύο μηχανές να συμφωνούν.

Ανίχνευση της σειράς bytes του υπολογιστή

Χρήσιμο μία φορά, ως έλεγχος ορθότητας. Πακετάρετε το 1 ως ακέραιο 16-bit και κοιτάξτε το πρώτο byte:

```
my $little_endian = unpack("c", pack("s", 1)) == 1;
my $big_endian    = unpack("c", pack("s", 1)) == 0;
```

Σε x86 αληθεύει το πρώτο· σε big-endian υπολογιστή αληθεύει το δεύτερο.

Και τα float έχουν endianness

Το IEEE 754 δεν καθηλώνει τη σειρά bytes. Ένα double που γράφεται σε little-endian μηχανή βγαίνει αντεστραμμένο σε μια big-endian:

```
# send a float in big-endian order - safe across machines that use␣
↪IEEE 754
my $buf = pack "d>", 2.718281828;
```

Το ασύμβατο hardware (μορφές float που δεν είναι IEEE, εξωτικές διατάξεις long double) δεν μπορεί να επιδιορθωθεί μόνο με τα `<` / `>`. Για φορητότητα σε ασυνήθιστες πλατφόρμες, στείλτε τα float ως κείμενο ή χρησιμοποιήστε μια μορφή ακριβή σε byte όπως η έξοδος του `sprintf "%a"`.

Επεξεργασμένο παράδειγμα: ανάλυση ενός header BMP

Η μορφή bitmap BMP ξεκινάει με τη μαγική ASCII `BM` ακολουθούμενη από τρία πεδία little-endian 32-bit (μέγεθος αρχείου, δεσμευμένο, offset δεδομένων pixel):

```
my ($magic, $filesize, $reserved, $pixoff) =
    unpack "A2 V V V", $bmp_header;

die "not a BMP" unless $magic eq "BM";
```

Το `V` είναι little-endian 32-bit μη προσημασμένο - ακριβώς αυτό που απαιτούν οι προδιαγραφές. Το `A2` διαβάζει δύο bytes ως συμβολοσειρά κειμένου χωρίς αποκοπή. Κάθε πεδίο έχει το σωστό σχήμα· δεν απαιτείται εναλλαγή bytes, δεν υπάρχει συμπεριφορά εξαρτώμενη από την πλατφόρμα.

Τι να θυμάστε

- **Μορφή επί της γραμμής; Χρησιμοποιήστε `n` / `N` / `v` / `V`** - ή `<` / `>` αν χρειάζεστε προσημασμένο, 64-bit, ή float.
- **Παραμένετε εντός μίας διεργασίας; Το native είναι εντάξει** - αλλά επισημάνετε την επιλογή σκόπιμα.
- **Το IEEE 754 δεν αρκεί.** Αν ένα float διασχίζει μια μηχανή, καθηλώστε τη σειρά bytes.

Με τους μετρητές bytes και τη σειρά bytes υπό έλεγχο, οι υπόλοιπες οδηγίες - συμβολοσειρές, τοποθέτηση, ομάδες - προκύπτουν φυσιολογικά.

Συμβολοσειρές, bits, και nybbles

Στο τέλος αυτού του κεφαλαίου θα μπορείτε να πακετάρετε και αποπακετάρετε συμβολοσειρές σταθερού πλάτους, C-style NUL-τερματισμένες συμβολοσειρές, συμβολοσειρές bits, και δεκαεξαδικές συμβολοσειρές - τις τρεις οικογένειες οδηγιών που μοιάζουν με κείμενο.

Τα δυαδικά πρότυπα μεταφέρουν δεδομένα συμβολοσειρών σε διάφορες ξεχωριστές γεύσεις: ωμά bytes, κείμενο με συμπλήρωση κενών, NUL-τερματισμένες συμβολοσειρές C, πεδία bits, αριθμοί κωδικοποιημένοι σε δεκαεξαδικό. Η `pack` έχει μία οδηγία για το καθένα.

Οι τρεις οδηγίες συμβολοσειρών: `a` / `A` / `Z`

Και οι τρεις πακετάρουν ακριβώς μία τιμή σε σταθερό πλάτος. Διαφέρουν στο με τι συμπληρώνουν και - κρισίμως - στο τι η `unpack` αφαιρεί κατά την επιστροφή:

Γράμμ	Byte συμπλήρωσης	Η unpack επιστρέφει	Τυπική χρήση
`a`	`"\0"` (NUL)	Όλα τα bytes αμετάβλητα	Αυθαίρετα δυαδικά
`A`	`" "` (κενό)	Τελικοί λευκοί χαρακτήρες και NUL αφαιρούνται	Κείμενο ASCII σταθερού πλάτους
`Z`	`"\0"` (NUL)	Bytes μέχρι το πρώτο NUL	C-style NUL-τερματισμένο

Το πλάτος είναι ο **μετρητής επανάληψης**, όχι μετρητής τιμών:

```
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
```

Το Z εγγυάται τελικό NUL - με μια επιφύλαξη

Το Z δεσμεύει πάντα χώρο για τουλάχιστον ένα τερματικό NUL:

```
pack "Z*", "hello"        # "hello\0"   - a free NUL appended
pack "Z5", "hello"        # "hell\0"    - truncated to make room!
```

Αν ένα πρόγραμμα C στην άλλη πλευρά αναμένει ένα μηδενοτερματισμένο `char[32]`, χρησιμοποιήστε Z32. Θα πακετάρει έως 31 bytes δεδομένων συν τον τερματιστή.

Επιλογή μεταξύ a και A για κείμενο

- Τα δεδομένα είναι ASCII, με συμπλήρωση κενών στον δίσκο → A.
- Τα δεδομένα είναι αυθαίρετα bytes (ενδέχεται να περιέχουν κενά ή NUL ως έγκυρο περιεχόμενο) → a.

Η κλασική παγίδα: πακετάρετε ένα όνομα με A20 και η μορφή επί της γραμμής χρησιμοποιεί συμπλήρωση με NUL. Η `unpack "A20"` αφαιρεί και κενά *και* NUL, οπότε φαίνεται εντάξει - αλλά η `pack "A20", "bob"` θα κάνει πλήρη κύκλο σε `"bob "` συμπληρωμένο με κενά. Αν οι προδιαγραφές λένε συμπλήρωση με NUL, χρησιμοποιήστε a στην πλευρά του pack.

Επεξεργασμένο παράδειγμα: εγγραφές κειμένου σταθερού πλάτους

Ένα αρχείο κατάστιχου διατάσσει τα δεδομένα σε στήλες:

```
0         1         2         3         4
01234567890123456789012345678901234567890123456789
2026-04-22 coffee at the station           3.50
2026-04-23 train to Brussels              42.00
```

Η στήλη 1-10 είναι η ημερομηνία, 12-38 η περιγραφή, 40-47 το ποσό. Τα κενά είναι μεμονωμένα κενά (byte 11 και byte 39). Η unpack με `A10 x A27 x A*` ξεφλουδίζει κάθε εγγραφή:

```
while (<$ledger>) {
    chomp;
    my ($date, $desc, $amount) = unpack "A10 x A27 x A*", $_;
    print "  $date | $desc | $amount\n";
}
```

Το x παρακάμπτει ένα byte· το A27 καταβροχθίζει τη στήλη περιγραφής και αφαιρεί τελικά κενά· το A* καταβροχθίζει άπληστα τα υπόλοιπα bytes ως ποσό. Η πλευρά εξόδου χρησιμοποιεί ευρύτερα πεδία ώστε τα κενά να επιβιώνουν στον πλήρη κύκλο:

```
my $line = pack "A11 A28 A8 A*", $date, $desc,
                                 sprintf("%.2f", $amt_left),
                                 sprintf("%12.2f", $amt_right);
```

Παρατηρήστε το επιπλέον byte σε κάθε πλάτος A - αυτό το μοναδικό επιπλέον byte είναι το κενό στήλης. Ένας συνεπής προϋπολογισμός bytes ανά στήλη είναι αυτό που κάνει τις εγγραφές σταθερού πλάτους διαχειρίσιμες εξαρχής.

Συμβολοσειρές bits: b και B

Μια συμβολοσειρά bits είναι μια συμβολοσειρά χαρακτήρων `"0"` και `"1"` που πακετάρει σε πραγματικά bits. Δύο οδηγίες, που διαφέρουν μόνο στο πώς διαβάζετε κάθε byte:

Οδηγία	Σειρά bits μέσα σε κάθε byte
b	LSB πρώτο - bit 0, 1, 2, 3, 4, 5, 6, 7
B	MSB πρώτο - bit 7, 6, 5, 4, 3, 2, 1, 0

Ο μετρητής επανάληψης είναι ο αριθμός των **bits**, όχι bytes:

```
pack "B8", "10001100"      # "\x8c" - MSB first, bit 7 set, bit 3 &␣
 ↪2 set
pack "b8", "00110001"      # "\x8c" - LSB first, same byte
```

Το B ταιριάζει με τη συνηθισμένη σύμβαση «από αριστερά προς τα δεξιά είναι από υψηλό προς χαμηλό» που βλέπετε σε δυαδικά dumps· το b ταιριάζει με τη σύμβαση που χρησιμοποιούν ορισμένοι καταχωρητές hardware και η ενσωματωμένη `vec`. Χρησιμοποιήστε B όταν γράφετε έναν «κανονικό» δυαδικό αριθμό, b όταν αντικατοπτρίζετε ένα φύλλο δεδομένων που αριθμεί το bit 0 στα δεξιά.

Μέτρηση ενεργοποιημένων bits

Μία από τις πιο απρόσμενες χρήσεις της unpack: μέτρηση των ενεργοποιημένων bits σε έναν ενταμιευτή με μία κλήση. Το %32b* αποπακετάρει τα bits και ζητάει ένα άθροισμα 32-bit, που είναι η μέτρηση:

```
my $n_bits = unpack "%32b*", $mask;
```

Το πρόθεμα %N υπάρχει μόνο στην unpack - δείτε το *κεφάλαιο για την τοποθέτηση* για τις άλλες χρήσεις.

Δεκαεξαδικές συμβολοσειρές: h και H

Οι δεκαεξαδικές συμβολοσειρές είναι η αναπαράσταση κειμένου που διαβάζουν οι περισσότεροι προγραμματιστές - δύο δεκαεξαδικά ψηφία ανά byte. Δύο οδηγίες, που διαφέρουν στη σειρά των nybbles:

Οδηγία	Σειρά nybbles
h	Χαμηλό nybble πρώτο
H	Υψηλό nybble πρώτο

Το H είναι το «κανονικό» δεκαεξαδικό dump - διάβασμα από αριστερά προς δεξιά, υψηλό ψηφίο μετά χαμηλό:

```
pack "H*", "deadbeef"      # "\xde\xad\xbe\xef"
unpack "H*", "\xde\xad\xbe\xef"   # "deadbeef"
```

Το h αντιστρέφει τα nybbles κάθε byte - σπάνια αυτό που θέλετε εκτός αν οι προδιαγραφές το λένε ρητά. Αν ο πλήρης κύκλος που περιμένετε *δεν* είναι αυτός που παίρνετε, δοκιμάστε το άλλο γράμμα.

```
pack "h*", "ef"            # "\xfe"  (nybbles swapped)
pack "H*", "ef"            # "\xef"
```

Επιλογή μεταξύ τους

Έχετε	Καταφύγετε σε
ASCII σε στήλη σταθερού πλάτους στον δίσκο	A
Αυθαίρετα bytes σε θυρίδα σταθερού πλάτους	a
Μια C-style NUL-τερματισμένη συμβολοσειρά	Z
Έναν δυαδικό αριθμό ως συμβολοσειρά από 0 / 1	B
Το ίδιο, με τη σειρά bits της σύμβασης καταχωρητών	b
Δεκαεξαδική συμβολοσειρά στη συνηθισμένη σειρά ανάγνωσης	H
Δεκαεξαδική συμβολοσειρά με εναλλαγμένα nybbles	h

Οι οδηγίες bits, nybbles, και συμβολοσειρών μοιράζονται έναν κανόνα: ο μετρητής επανάληψης ορίζει το πλάτος της *μοναδικής* τιμής που πακετάρεται, όχι τον αριθμό των τιμών. Αυτό είναι το ένα γεγονός που πρέπει να μεταφέρετε παρακάτω.

Στη συνέχεια: ομάδες ( ) και μετρητές επανάληψης, τα εργαλεία για την εφαρμογή ενός μοτίβου οδηγιών σε λίστα τιμών αγνώστου μήκους.

Ομαδοποίηση και μετρητές

Στο τέλος αυτού του κεφαλαίου θα μπορείτε να επαναλαμβάνετε ένα μοτίβο οδηγιών σε μια λίστα τιμών, να χρησιμοποιείτε τις μορφές * και [...] του μετρητή επανάληψης, και να γράφετε αυτο-περιγραφόμενες εγγραφές με πρόθεμα μήκους με την οδηγία /.

Ένας γυμνός μετρητής επανάληψης εφαρμόζεται σε *ένα* γράμμα οδηγίας: το C4 πακετάρει τέσσερα bytes. Τη στιγμή που η επαναλαμβανόμενη μονάδα είναι περισσότερες από μία οδηγίες - «πακετάρισμα ενός short και δύο bytes, πολλές φορές» - χρειάζεστε ομάδα.

- «pack a short and two bytes, many times» - you need a group.

( ) - μια ομάδα είναι ένα υπο-πρότυπο

Οι παρενθέσεις συγκεντρώνουν μια ακολουθία οδηγιών ώστε ένας μετρητής επανάληψης ή τροποποιητής endianness να εφαρμόζεται στο σύνολο. Συγκρίνετε:

```
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
```

Μια ομάδα δεν έχει δικό της κόστος σε bytes - είναι συντακτικός μηχανισμός. Το συνολικό πακεταρισμένο μήκος είναι αυτό που θα παρήγαγαν οι οδηγίες μέσα της χωρίς τις παρενθέσεις.

Ομαδοποίηση με endianness

Η πιο πρακτική χρήση μιας ομάδας - μια υπο-δομή της οποίας κάθε ακέραιος μοιράζεται μία σειρά bytes:

```
my $rec = pack "(l s s)<", $id, $x, $y;
# same as "l<s<s<"
```

Το < διαχέεται σε κάθε οδηγία με σειρά bytes που βρίσκεται μέσα, συμπεριλαμβανομένων των εμφωλευμένων ομάδων. Οι οδηγίες που δεν δέχονται τροποποιητή σειράς bytes (όπως οι C, a, Z) μένουν σιωπηλά ανεπηρέαστες.

Μετρητές επανάληψης αναλυτικά

Μετά από οποιαδήποτε οδηγία ή ομάδα, μπορείτε να γράψετε:

Μορφή	Σημασία
N	Εφαρμόστε την οδηγία/ομάδα N φορές
*	Εφαρμογή όσο διαρκούν οι τιμές. Για x, X, @: ισοδυναμεί με 0. Για u: 45.
[N]	Όπως το N
[templ]	Ο μετρητής επανάληψης είναι το μήκος σε πακεταρισμένα bytes του προτύπου σε αγκύλες

Η μορφή [template] είναι το εργαλείο για να εκφράσετε «τόσα bytes όσα παίρνει ένα *foo*»:

```
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
```

Είναι ιδιαίτερα χρήσιμη για ευθυγράμμιση (δείτε το *κεφάλαιο για την τοποθέτηση*) και για πρότυπα των οποίων τα πλάτη πρέπει να ακολουθούν το εξαρτώμενο από την πλατφόρμα μέγεθος ενός native ακεραίου:

```
pack "a[l!]", $native_long_buf      # room for one native long
```

Το * εφαρμόζεται ανά ομάδα-τιμής, όχι «καταπίνει τα πάντα»

Ένα μεμονωμένο * μετράει «υπόλοιπες τιμές» *για αυτή την οδηγία*. Δύο A* στη σειρά δεν ανταγωνίζονται:

```
pack "A*A*", "hello", "world"     # "helloworld"
```

Το πρώτο A* πακετάρει όλο το "hello"· το δεύτερο πακετάρει όλο το "world". Κάθε * καταναλώνει *μία* τιμή από τη λίστα, στο πλήρες μήκος αυτής της τιμής. Αυτός είναι ο γενικός κανόνας: **κάθε οδηγία αντιστοιχεί σε ένα κομμάτι δεδομένων**, ανεξάρτητα από τον μετρητή επανάληψης.

Η οδηγία / - μήκος και δεδομένα μαζί

Οι μορφές επί της γραμμής συχνά αποθηκεύουν ένα μετρητή ακριβώς πριν από το πράγμα που μετριέται: «ένα μήκος 2-byte, μετά τόσα bytes φορτίου». Η οδηγία / συνδέει τα δύο σε ένα βήμα.

Στην pack: *length-item*/*sequence-item*

Γράψτε *δύο* οδηγίες χωρισμένες με κάθετο. Η πρώτη πακετάρει το μήκος· η δεύτερη πακετάρει το φορτίο. Η pack υπολογίζει το μήκος για εσάς:

```
my $msg = pack "n/a*", "hello, world";
# "\x00\x0chello, world"
#  ^^^^^^^^ big-endian 16-bit length = 12
#           ^^^^^^^^^^^^ the payload itself
```

Το *length-item* μπορεί να είναι οποιαδήποτε αριθμητική οδηγία - C, n, N, w, S<, και ούτω καθεξής - ή οδηγία συμβολοσειράς όπως A4 όταν το πρωτόκολλο γράφει το μήκος ως ASCII:

```
my $buf = pack "A4/A*", "Humpty-Dumpty";
# "13  Humpty-Dumpty"  - 4-char ASCII length, then the string
```

Στην unpack: */item*

Η μορφή της unpack είναι απλούστερη: ένα γυμνό / πριν από το στοιχείο. Ο μετρητής λαμβάνεται από την πιο πρόσφατη ακέραια οδηγία:

```
my ($payload) = unpack "n/a*", $msg;     # "hello, world"
```

Διαβάζοντας αυτό το πρότυπο: «διάβασε έναν ακέραιο n, ονόμασέ τον L· μετά διάβασε L bytes ως συμβολοσειρά τύπου a*.» Το ίδιο το μήκος δεν εμφανίζεται στη λίστα εξόδου.

Το συνηθισμένο λάθος: A* μετά από /

Δεν μπορείτε να βάλετε άλλο A* ή a* μετά από πεδίο που εισάγει / στην unpack και να περιμένετε ότι θα συμπεριφερθεί - το * είναι άπληστο:

```
# 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;
```

Στο δεύτερο πρότυπο, το C/A* διαβάζει έναν μετρητή bytes και μετά τόσα bytes. Όλα μετά την κάθετο τον σέβονται, και το τελικό C παίρνει το επόμενο byte όπως αναμενόταν.

Επεξεργασμένο παράδειγμα: ζεύγη κλειδιού-τιμής

Ένα πρωτόκολλο αποθηκεύει ένα λεξικό ως `count` ακολουθούμενο από `count` ζεύγη της μορφής `(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;
```

Διαβάζοντάς το πίσω:

```
my %parsed = unpack "S/(S/A* S/A*)", $blob;
```

Το πρότυπο της pack λέει: «ένας μετρητής 16-bit (ζευγών), μετά επανέλαβε το υπο-πρότυπο `S/A* S/A*` για κάθε ζεύγος.» Το πρότυπο της unpack διαβάζει τον μετρητή και τον εφαρμόζει απευθείας στην ομάδα - ο μετρητής δεν εμφανίζεται πλέον στη λίστα εξόδου.

Οριακές περιπτώσεις και περιορισμοί

- **Το `/` δεν έχει νόημα με στοιχείο σταθερού μήκους.** Η δεύτερη οδηγία πρέπει να είναι μεταβλητού πλάτους: `a*`, `A*`, `Z*`, `/A$n`, ή κάτι ανάλογο. Η Perl θα απορρίψει δεύτερο στοιχείο σταθερού μήκους.
- **Το `()*` με την pack δεν μπορεί να αντιστοιχιστεί από `()*` στην unpack.** Η pack έχει τις τιμές, οπότε μπορεί να πει «επανέλαβε μέχρι να τελειώσει». Η unpack δεν γνωρίζει πόσες επαναλήψεις είναι κωδικοποιημένες στον ενταμιευτή εκτός αν μια οδηγία μετρητή προηγείται της ομάδας.
- **Οι εμφωλευμένες ομάδες** είναι θεμιτές και συνηθισμένες:

  ```
  pack "((CC)(S))<", @records
  ```

  Οι τροποποιητές endianness διαχέονται σε κάθε επίπεδο εμφώλευσης.
- **Ο μετρητής επανάληψης σε μια ομάδα** εφαρμόζεται σε ολόκληρη την ομάδα και παίρνει τόσες επαναλήψεις τιμών. Το `(CS)3` καταναλώνει *έξι* τιμές της λίστας, όχι τρεις.

Επόμενο κεφάλαιο: οι οδηγίες που μετακινούνται μέσα σε ένα πρότυπο χωρίς να παράγουν τιμή - `x`, `X`, `@`, `..`

Τοποθέτηση και συμπλήρωση

Στο τέλος αυτού του κεφαλαίου θα μπορείτε να παρακάμπτετε bytes, να εισάγετε συμπλήρωση, να μεταπηδάτε σε απόλυτα offsets, και να ευθυγραμμίζετε πεδία σε αυθαίρετα όρια - οι τέσσερις οδηγίες `x`, `X`, `@`, και `..`

Ορισμένες οδηγίες παράγουν ένα byte χωρίς να καταναλώνουν τιμή από τη λίστα (bytes συμπλήρωσης). Ορισμένες καταναλώνουν μια τιμή χωρίς να παράγουν byte (εντολές

τοποθέτησης). Μαζί σας επιτρέπουν να μοντελοποιείτε τα κενά και τις ευθυγραμμίσεις που περιέχουν τα πραγματικά δυαδικά πρότυπα.

x - παράλειψη προς τα εμπρός / εισαγωγή NUL

Το x είναι το απλούστερο από τα τέσσερα. Στην pack εισάγει ένα byte NUL χωρίς να καταναλώνει τιμή· στην unpack παρακάμπτει ένα byte χωρίς να παράγει τιμή.

```
pack   "C x C", 65, 66       # "A\0B"
unpack "C x C", "A\0B"       # (65, 66)
```

Με μετρητή επανάληψης, εισάγει ή παρακάμπτει τόσα bytes:

```
pack "C x3 C", 65, 66        # "A\0\0\0B"
```

Το x* σημαίνει «παράλειψη μέχρι το τέλος της συμβολοσειράς» στην unpack, και ισοδυναμεί με x0 (no-op) στην pack.

Το x δεν σημαίνει «ένα κενό»

Το μπερδεμένο με το x είναι ότι συμπληρώνει με NUL, όχι με κενό. Αν θέλετε συμπλήρωση με κενά για κείμενο, χρησιμοποιήστε το A με ευρύτερο μετρητή επανάληψης:

```
# 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 - επιστροφή προς τα πίσω

Το X μετακινεί την κεφαλή εγγραφής/ανάγνωσης προς τα πίσω κατά ένα byte. Με μετρητή επανάληψης, προς τα πίσω κατά τόσα bytes. Σε αντίθεση με το x, το X δεν έχει επίδραση στα bytes εξόδου της pack - απλώς επανατυλίσσει ώστε μια επόμενη οδηγία να τα αντικαταστήσει.

```
pack "CC X C", 65, 66, 67    # "AC" - the second C overwrites 66␣
↪with 67
```

Το X είναι κυρίως χρήσιμο με την unpack: ματιά μπροστά, μετά επανατύλιξη για επανανάγνωση:

```
# 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;
```

Το X πριν την αρχή της συμβολοσειράς είναι μοιραίο σφάλμα.

@ - μετάβαση σε απόλυτη θέση

Το @N ορίζει την τρέχουσα θέση στο byte N **εντός της εσώτατης ομάδας ()** (ή από την αρχή της συμβολοσειράς αν δεν υπάρχει περικλείουσα ομάδα). Στην pack, αυτό συμπληρώνει με μηδενικά από την τρέχουσα θέση προς τα εμπρός, ή αποκόπτει αν η τρέχουσα θέση είναι μετά το N:

```
pack "C @4 C", 65, 66        # "A\0\0\0B"  - byte 0 is 'A', byte 4␣
↪is 'B'
```

Αυτό είναι το αντίστοιχο του offsetof(struct, field) σε επίπεδο προτύπου: αν ένα header αρχείου C σας πει ότι το πεδίο 2 βρίσκεται στο offset 12, το @12 σας παρκάρει εκεί ανεξάρτητα από τι έκαναν οι προηγούμενες οδηγίες.

Το @ μέσα σε ομάδα

Σε κάθε επανάληψη μιας ομάδας, το @ επανεκκινεί στο 0. Αυτό μπορεί να σας εκπλήξει:

```
pack '@1A((@2A)@3A)', qw(X Y Z)
# "\0X\0\0YZ"
```

Βήμα προς βήμα: το εξωτερικό @1A βάζει "X" στο byte 1 (με NUL στο byte 0). Η εσωτερική ομάδα ξεκινάει νέο σύστημα συντεταγμένων στη θέση του "X"· μέσα της, το @2A βάζει "Y" στο byte 2 της ομάδας (με δύο ακόμα NUL ενδιάμεσα), μετά το @3A βάζει "Z" στο byte 3 της ομάδας. Διαβάστε τις προδιαγραφές προσεκτικά πριν χρησιμοποιήσετε @ μέσα σε ομάδες.

. - απόλυτη θέση από τα δεδομένα

Το . είναι ο εξάδελφος του @ που οδηγείται από τιμές. Αντί για σταθερό μετρητή επανάληψης, η θέση προέρχεται από την επόμενη τιμή στη λίστα:

```
pack "C . C", 65, 4, 66     # "A\0\0\0B" - position 4 from the next␣
↪value
```

Χρήσιμο όταν το offset υπολογίζεται κατά τον χρόνο εκτέλεσης - ας πούμε, από ένα πεδίο header που μόλις διαβάσατε:

```
my $buf = pack "C .N C", $type, $offset, $value;
# $offset bytes of NUL fill before $value
```

Στην unpack, το . επιστρέφει την τρέχουσα θέση αντί να συμπληρώνει με μηδενικά:

```
my ($a, $pos) = unpack "C .", "hello";
# $a = 104 ('h'), $pos = 1
```

Με .*, το offset μετριέται από την αρχή ολόκληρης της συμβολοσειράς· με .N για ακέραιο N ≥ 1, από την αρχή της περικλείουσας N-οστής ομάδας· με .0, σχετικά με την τρέχουσα θέση.

Ο τροποποιητής !: ευθυγράμμιση

Τα `x!N` και `X!N` μετατρέπουν τα `x` και `X` σε εντολές ευθυγράμμισης: προωθούν (ή επανατυλίσσουν) στο πλησιέστερο πολλαπλάσιο του `N`. Έτσι σέβεστε τους κανόνες συμπλήρωσης struct της C.

```
# struct { char c; double d; char cc[2]; }
my $s = pack "c x![d] d c2", $c, $d, $c1, $c2;
```

Διαβάζοντας το `x![d]`: «ευθυγράμμιση προς τα εμπρός σε πολλαπλάσιο του πλάτους που θα έπαιρνε ένα πακεταρισμένο d» - δηλαδή 8 bytes. Αυτός είναι ακριβώς ο κανόνας που εφαρμόζουν οι περισσότεροι μεταγλωττιστές C σε ένα πεδίο `double` μετά από ένα `char`. Χρησιμοποιήστε `[l!]`, `[d]`, `[Q]` για να ονομάσετε την απαίτηση ευθυγράμμισης χωρίς να χρειάζεται να γνωρίζετε το αριθμητικό πλάτος.

Ένα γυμνό `x!0` ή `x!1` είναι no-op.

Αθροίσματα ελέγχου - `%N` μόνο στην unpack

Το `%N` είναι πρόθεμα σε μία οδηγία, όχι αυτοτελής οδηγία. Λέει «αντί να επιστραφούν οι τιμές που θα παρήγαγε η επόμενη οδηγία, επίστρεψε το `N`-bit άθροισμά τους». Συνηθισμένες χρήσεις:

```
my $bytesum = unpack "%32C*", $buf;    # sum every byte
my $setbits = unpack "%32b*", $mask;   # count set bits
```

Υπάρχει μόνο στην `unpack`. Δεν υπάρχει `%` στην `pack` - δεν μπορείτε να πακετάρετε άθροισμα ελέγχου απευθείας· υπολογίστε το οι ίδιοι και πακετάρετε το αποτέλεσμα με όποια ακέραια οδηγία χρειάζεστε.

Επεξεργασμένο παράδειγμα: ανάγνωση φορτίου με μεταβλητό offset

Ορισμένες μορφές ξεκινούν με header σταθερού μεγέθους αλλά τοποθετούν τα πραγματικά δεδομένα σε ένα offset που περιέχει το header. Η κλασική συνταγή:

```
# 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;
```

Δύο ξεχωριστές κλήσεις `unpack` - η πρώτη διαβάζει το offset, η δεύτερη το χρησιμοποιεί στο πρότυπο. Οι συμβολοσειρές προτύπων δεν είναι κανονικές εκφράσεις· δεν έχουν τρόπο να κοιτάξουν μπροστά και να χρησιμοποιήσουν μια υπολογισμένη τιμή εντός της ίδιας κλήσης. Αποπακετάρετε σε στάδια όταν το σχήμα των δεδομένων εξαρτάται από αυτά που μόλις διαβάσατε.

Με την τοποθέτηση και την ευθυγράμμιση στην εργαλειοθήκη, τα δύο επεξεργασμένα παραδείγματα στα επόμενα κεφάλαια θα διαβάζονται ως απλές εφαρμογές: ανάλυση ενός

πραγματικού header δικτύου και μιας πραγματικής μορφής αρχείου.

Πρωτόκολλα δικτύου - ένα ερώτημα DNS

Στο τέλος αυτού του κεφαλαίου θα μπορείτε να κατασκευάζετε και να αναλύετε ένα header ερωτήματος DNS και ένα question section χρησιμοποιώντας τις `pack` και `unpack` - και να ακολουθείτε την ίδια προσέγγιση σε οποιοδήποτε πρωτόκολλο που ορίζεται από RFC.

Ένα «πρωτόκολλο δικτύου» είναι, εν τέλει, μια ακολουθία πεδίων ακριβών σε byte που περιγράφονται σε αγγλική πρόζα. Αν μπορείτε να διαβάσετε τον πίνακα πεδίων, μπορείτε να γράψετε το πρότυπο. Αυτό το κεφάλαιο διατρέχει ένα συγκεκριμένο παράδειγμα - το πακέτο ερωτήματος DNS - από το διάγραμμα επί της γραμμής του RFC σε ένα πλήρως λειτουργικό ζεύγος υπορουτινών κωδικοποίησης / αποκωδικοποίησης.

Το πρόβλημα

Θέλουμε να ρωτήσουμε έναν διακομιστή DNS για την εγγραφή A του `example.com`. Το φορτίο UDP που στέλνουμε είναι ένα **μήνυμα DNS** (RFC 1035), που αποτελείται από:

1. Ένα **header** 12 byte.
2. Ένα **question section** - τον τομέα για τον οποίο ρωτάμε, και τον τύπο εγγραφής που θέλουμε.

(Ένας πραγματικός πελάτης αναλύει επίσης το **answer section** στην απάντηση. Σταματάμε στην κωδικοποίηση και στην παρουσίαση ενός σκαριφήματος της αποκωδικοποίησης.)

Το header

Από την ενότητα 4.1.1 του RFC 1035, το header είναι έξι διαδοχικά πεδία 16-bit big-endian:

Offset	Πεδίο	Σημασία
0	ID	Αυθαίρετο 16-bit αναγνωριστικό που επιλέγουμε
2	Σημαίες	Opcode, bit RD, σημαίες απάντησης, rcode
4	QDCOUNT	Πλήθος εγγραφών στο question section
6	ANCOUNT	Πλήθος εγγραφών πόρων στην απάντηση
8	NSCOUNT	Εγγραφές authority
10	ARCOUNT	Πρόσθετες εγγραφές

Έξι μη προσημασμένα big-endian shorts σημαίνει έξι οδηγίες `n`:

```
sub dns_header {
    my (%opts) = @_;
    pack "n6",
        $opts{id}      // 0,
        $opts{flags}   // 0,
        $opts{qd}      // 0,
        $opts{an}      // 0,
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
        $opts{ns}      // 0,
        $opts{ar}      // 0;
}
```

Αυτό είναι 12 bytes, ακριβώς όπως απαιτούν οι προδιαγραφές. Το `n6` είναι συντομογραφία του `n n n n n n`.

Το question section

Ένα question είναι:

1. Ένα **όνομα**, κωδικοποιημένο ως ακολουθία ετικετών με πρόθεμα μήκους που τερματίζεται από μια ετικέτα μηδενικού μήκους.
2. Ένα 16-bit **QTYPE** (1 = εγγραφή A).
3. Ένα 16-bit **QCLASS** (1 = IN, Internet).

Για το `example.com` το όνομα κωδικοποιείται ως:

```
\x07 e x a m p l e   \x03 c o m   \x00
```

Κάθε ετικέτα ξεκινάει με ένα byte μήκους· ολόκληρο το όνομα τελειώνει με μια ετικέτα μηδενικού μήκους. Δύο πράγματα να προσέξετε: το μήκος είναι ένα byte (`C`), όχι δύο· και υπάρχει ένα τελικό `NUL` αλλά *δεν* είναι ακριβώς αυτό που παράγει το `Z` - τερματίζει τη λίστα ετικετών, όχι μια μεμονωμένη συμβολοσειρά.

Μπορούμε να το κατασκευάσουμε χειροκίνητα από ένα όνομα τομέα:

```
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;
}
```

Δύο οδηγίες στο σώμα του βρόχου: `C/a*` - ένα μήκος ενός byte ακολουθούμενο από τα bytes της ετικέτας, υπολογισμένο αυτόματα (δείτε το *κεφάλαιο grouping-and-counts* για τη μορφή `/`). Μετά τον βρόχο, ένα κυριολεκτικό `"\0"` τερματίζει τη λίστα.

Με την `encode_name` στη θέση της, ολόκληρο το question section είναι δύο συνενώσεις και μία `pack`:

```
sub dns_question {
    my ($name, $qtype, $qclass) = @_;
    return encode_name($name) . pack "n n", $qtype, $qclass;
}
```

Συναρμολόγηση του πακέτου

```
use constant {
    QR_QUERY    => 0,
    OPCODE_QUERY=> 0,
    RD          => 1 << 8,          # recursion desired
    TYPE_A      => 1,
    CLASS_IN    => 1,
};

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 συνολικά: 12 header + 13 για το όνομα (το `\x07example\x03com\x00` είναι 13) + 2 + 2 για τα QTYPE και QCLASS. Τρέξτε `length $pkt` για επιβεβαίωση.

Ανάλυση του header απάντησης

Η απάντηση έχει το ίδιο σχήμα header - μόνο τα bits σημαιών αλλάζουν. Η αποκωδικοποίηση είναι το αντίστροφο πρότυπο:

```
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,
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
        qd    => $qd,
        an    => $an,
        ns    => $ns,
        ar    => $ar,
    );
    return \%hdr;
}
```

Το n6 ανασύρει τα έξι shorts με μία κίνηση· τα επιμέρους bits σημαιών προκύπτουν από shifts και masks πάνω στο `$flags`. Αυτό το μοτίβο είναι καθολικό: χρησιμοποιήστε pack / unpack για τη διάταξη σε επίπεδο byte, σκέτη Perl για αποκωδικοποίηση σε επίπεδο bit των πεδίων σημαιών.

Ανάλυση ενός ονόματος

Τα ονόματα στο answer section χρησιμοποιούν την ίδια κωδικοποίηση προθέματος μήκους, συν έναν μηχανισμό **συμπίεσης δεικτών** που θα παραλείψουμε. Για ένα φρέσκο όνομα (χωρίς συμπίεση) ο αποκωδικοποιητής κατοπτρίζει τον κωδικοποιητή:

```
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);
}
```

Τρεις κλήσεις unpack, καθεμία με πρόθεμα `x$offset` για να παρακαμφθεί μέχρι το σωστό byte. Το όνομα εναλλάσσεται μεταξύ μηκών ενός byte και ετικετών μεταβλητού πλάτους, και ο βρόχος συνεχίζει να διαβάζει μέχρι να συναντήσει μια ετικέτα μηδενικού μήκους. Το τέχνασμα `x$offset a$len` - σύνθεση προτύπου από τιμές της Perl, κλήση της unpack - είναι το πρότυπο μοτίβο όταν το πλάτος του επόμενου πεδίου εξαρτάται από την τιμή του προηγούμενου.

Τι να μεταφέρετε παρακάτω

- **Ένας πίνακας διάταξης bytes ενός RFC αντιστοιχίζεται απευθείας σε συμβολοσειρά προτύπου.** Big-endian shorts → `n`, big-endian longs → `N`, bytes → `C`.
- **Τα πεδία με πρόθεμα μήκους χρησιμοποιούν `C/a*` / `n/a*` / …** - μία έκφραση, χωρίς μέτρημα off-by-one με το χέρι.
- **Τα πεδία επιπέδου bit μέσα σε ένα byte αποκωδικοποιούνται σε σκέτη Perl** αφού η unpack σας έχει παραδώσει το byte ή το short που τα περιέχει.

- **Τα δεδομένα μεταβλητού πλάτους απαιτούν αποπακετάρισμα σε στάδια.** Διαβάστε ένα μήκος, μετά κατασκευάστε το πρότυπο για τα δεδομένα από αυτό το μήκος, μετά καλέστε ξανά την unpack.

Το επόμενο κεφάλαιο εφαρμόζει την ίδια προσέγγιση σε μια πραγματική μορφή αρχείου - ένα header εικόνας GIF - και εισάγει το ιδίωμα του «ελέγχου μαγικού αριθμού».

Πρότυπα αρχείων - ένα header GIF

Στο τέλος αυτού του κεφαλαίου θα μπορείτε να αναγνωρίζετε ένα δυαδικό αρχείο από τον μαγικό του αριθμό, να αναλύετε το header σταθερής διάταξής του, και να αποκωδικοποιείτε bytes σημαιών συσκευασμένα σε bit. Το παράδειγμα είναι η μορφή εικόνας GIF, αλλά η προσέγγιση μεταφέρεται απευθείας σε PNG, WAV, BMP, ELF, ZIP, και κάθε άλλη μορφή χτισμένη γύρω από σταθερό header.

Το header GIF και ο logical screen descriptor

Ένα αρχείο GIF ξεκινάει με δύο μπλοκ σταθερού μεγέθους, δεκατρία bytes συνολικά. Από τις προδιαγραφές GIF89a:

Header (6 bytes)

Offset	Bytes	Πεδίο
0	3	Υπογραφή - `"GIF"`
3	3	Έκδοση - `"87a"` ή `"89a"`

Logical Screen Descriptor (7 bytes)

Offset	Bytes	Πεδίο	Σειρά bytes
6	2	Λογικό πλάτος οθόνης	little-endian
8	2	Λογικό ύψος οθόνης	little-endian
10	1	Συσκευασμένες σημαίες	-
11	1	Δείκτης χρώματος φόντου	-
12	1	Αναλογία διαστάσεων pixel	-

Το byte συσκευασμένων σημαιών έχει πέντε υπο-πεδία:

```
Bit:  7  6 5 4  3  2 1 0
     [GCT][CR  ][Sort][GCT size]
```

- bit 7 - σημαία καθολικού πίνακα χρωμάτων
- bits 6-4 - ανάλυση χρώματος (μείον 1)
- bit 3 - σημαία ταξινόμησης
- bits 2-0 - μέγεθος καθολικού πίνακα χρωμάτων (πραγματικό μέγεθος = `2^(value+1)`)

Κάθε προδιαγραφή μορφής μοιάζει περίπου με αυτό: πεδία σταθερού πλάτους σε γνωστή σειρά, με κάποιο περιστασιακό byte από συσκευασμένα bits.

Διαβάζοντάς το

Το πρότυπο γράφεται από μόνο του πεδίο προς πεδίο:

```
sub parse_gif_header {
    my ($fh) = @_;
    binmode $fh;

    my $header;
    read $fh, $header, 13
        or die "short read: $!";

    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,
    };
}
```

Οδηγία προς οδηγία:

- `A3` `A3` - έξι bytes ASCII, χωρισμένα σε υπογραφή και έκδοση. `A` (όχι `a` ή `Z`) επειδή οι προδιαγραφές εγγυώνται ακριβώς τρεις χαρακτήρες χωρίς συμπλήρωση - το `A` τους επιστρέφει αμετάβλητους και έχει την ασφαλέστερη συμπεριφορά αποκοπής κατά το unpack.
- `v` `v` - δύο little-endian 16-bit μη προσημασμένοι ακέραιοι. Η μορφή GIF είναι little-endian από άκρη σε άκρη· το `v` ταιριάζει ακριβώς.
- `C` `C` `C` - τρία μη προσημασμένα bytes.

Συνολικό πλάτος: 3 + 3 + 2 + 2 + 1 + 1 + 1 = 13 bytes. Διαβάζουμε ακριβώς τόσα πριν αποπακετάρουμε· οτιδήποτε λιγότερο είναι σφάλμα.

Ο έλεγχος του μαγικού αριθμού

Δύο γραμμές παραπάνω παρουσιάζουν ένα μοτίβο που θα επαναλάβετε για κάθε μορφή αρχείου:

```
die "not a GIF" unless $sig eq "GIF";
die "unknown version: $ver" unless $ver eq "87a" or $ver eq "89a";
```

Ελέγξτε πρώτα τον μαγικό αριθμό, αποπακετάρετε τα υπόλοιπα μετά. Η αντιστροφή της σειράς επιτρέπει σε ασυναρτησίες να περάσουν σε μεταγενέστερα στάδια όπου τα μηνύματα σφάλματος είναι λιγότερο σαφή. Για μορφές με μεγαλύτερο μαγικό αριθμό (το PNG έχει 8 bytes, το ELF 4, το «RIFF...WAVE» σάντουιτς του WAV έχει 12), ισχύει η ίδια ιδέα: αποπακετάρετε αρκετά για να αναγνωρίσετε το αρχείο, ματαιώστε σε αναντιστοιχία, μετά αποπακετάρετε τα υπόλοιπα.

Αποκωδικοποίηση του byte συσκευασμένων σημαιών

Οι pack και unpack δεν μπορούν να αποκωδικοποιήσουν πεδία bit κάτω από byte απευθείας (με την εξαίρεση των οδηγιών b / B σε σχήμα συμβολοσειράς, δείτε το *κεφάλαιο strings*). Η πρακτική προσέγγιση είναι shifts και masks σε σκέτη Perl πάνω στο byte που επέστρεψε η unpack:

```
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»

- Το μοτίβο - «αποπακετάρετε το byte, ξεχωρίστε τα bits με shifts» - είναι τόσο συνηθισμένο ώστε αξίζει όνομα. Οτιδήποτε μικρότερο από byte ανήκει εκτός του προτύπου.

Επεξεργασμένη εκτέλεση

```
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};
```

Σε ένα πραγματικό αρχείο GIF89a διαστάσεων 640×480, αυτό εκτυπώνει:

```
GIF89a 640x480
global colour table: yes (256 entries)
```

Εγγραφή ενός header

Το αντίστροφο είναι ακόμα συντομότερο - κάθε οδηγία που αποπακετάραμε, μπορούμε να πακετάρουμε. Μία λεπτομέρεια: το byte σημαιών πρέπει να επανασυναρμολογηθεί από τα υπο-πεδία του πρώτα.

```
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},
        $hdr->{width}, $hdr->{height},
        $flags, $hdr->{bg}, $hdr->{aspect};
    print {$fh} $bytes;
}
```

Ίδιο πρότυπο στην πλευρά της pack, ίδια 13 bytes στην έξοδο, ταιριάζει με τον αναγνώστη GIF στο άλλο άκρο. Ο πλήρης κύκλος με ένα διάνυσμα δοκιμής είναι ο γρηγορότερος τρόπος να αποκτήσετε σιγουριά ότι έχετε το πρότυπο σωστό: αποπακετάρετε ένα υπάρχον αρχείο, ξανα-πακετάρετε το αποτέλεσμα, συγκρίνετε byte-προς-byte.

Κλιμάκωση σε μεγαλύτερες μορφές

Η προσέγγιση είναι συνθετική. Μια μορφή όπως το WAV έχει πολλαπλά chunks, καθένα με το δικό του header (`id` + `size` + `data`). Γράφετε:

- Έναν βοηθό χαμηλού επιπέδου «ανάγνωση ενός header chunk» - δύο κλήσεις unpack σε 8 bytes: 4-byte FourCC και 4-byte little-endian μέγεθος.
- Έναν διανομέα που κοιτάει τον FourCC και καλεί έναν αναλυτή ανά chunk.
- Αναλυτές ανά chunk που χρησιμοποιούν unpack στο σώμα του chunk σύμφωνα με τη δική του διάταξη bytes.

Κάθε στρώμα είναι μια μικρή unpack πάνω σε ένα μικροσκοπικό πρότυπο. Τη στιγμή που το πλάτος ενός πεδίου εξαρτάται από μια τιμή που μόλις διαβάσατε, σταματάτε, συνθέτετε το νέο πρότυπο από αυτή την τιμή, και καλείτε ξανά την unpack - το ίδιο σταδιακό μοτίβο που χρησιμοποιήσαμε για τα ονόματα DNS.

Τι μάθατε

- **Μαγικός αριθμός πρώτα, σώμα δεύτερο.** Αποτυχία γρήγορα σε λάθος μορφή.
- **Οι little-endian μορφές αρχείων χρησιμοποιούν v / V**, οι big-endian χρησιμοποιούν n / N· τα < / > χειρίζονται τις προσημασμένες ή 64-bit περιπτώσεις.

- **Τα πεδία υπο-byte είναι shifts σε σκέτη Perl**, όχι οδηγίες προτύπου.
- **Οι μεγαλύτερες μορφές αποσυντίθενται σε πολλά μικρά πρότυπα.** Η δουλειά του προγραμματιστή είναι να κρατά κάθε πρότυπο μικρό και προφανές.

Με τις σελίδες αναφοράς ως πηγή αναζήτησης και αυτά τα επτά κεφάλαια στο ενεργητικό σας, η επόμενη δυαδική μορφή που θα διασχίσει το γραφείο σας θα πρέπει να διαβάζεται ως λυμένο πρόβλημα. Οι σελίδες pack και unpack περιέχουν τον πλήρη πίνακα οδηγιών όταν χρειάζεται να επιβεβαιώσετε ένα πλάτος ή έναν τροποποιητή.

- **Bytes και πλάτη** - μέτρηση bytes, οδηγίες ακεραίων σταθερού πλάτους, η διάκριση native έναντι fixed.
- **Endianness** - σειρά bytes, οι τροποποιητές < / >, οι φορητές συντομεύσεις n / N / v / V.
- **Συμβολοσειρές** - a / A / Z (δυαδική, κείμενο, συμβολοσειρά C), συμβολοσειρές bits, δεκαεξαδικές συμβολοσειρές.
- **Ομαδοποίηση και μετρητές** - ομάδες `()`, μετρητές επανάληψης, μορφές `[…]`, η οδηγία `/` ως πρόθεμα μετρητή στοιχείων.
- **Τοποθέτηση** - x / X / @ / . για μετακίνηση μέσα σε ένα πρότυπο χωρίς παραγωγή τιμής.
- **Πρωτόκολλα δικτύου** - κατασκευή και ανάλυση ενός header ερωτήματος DNS.
- **Πρότυπα αρχείων** - ανάλυση ενός header GIF, του κανονικού αρχείου «μαγικός αριθμός συν σταθερά πεδία».

Δύο κεφάλαια (*network-protocols*, *file-formats*) είναι πλήρως επεξεργασμένα παραδείγματα. Διατρέξτε ένα από αυτά αφού έχετε διαβάσει αρκετά από τα άλλα κεφάλαια για να αναγνωρίζετε τις οδηγίες.

4.11.3 Μια πρώτη ολοκληρωμένη διαδρομή

Το συντομότερο χρήσιμο παράδειγμα των pack και unpack που συνεργάζονται - η πακετάρισμα μιας διεύθυνσης IP σε μορφή τετράδας με τελείες σε τέσσερα ωμά bytes και η εκ νέου ανάγνωσή της:

```
my $raw    = pack   "C4", 192, 168, 1, 42;   # "\xc0\xa8\x01\x2a"
my @octets = unpack "C4", $raw;              # (192, 168, 1, 42)
```

Το C πακετάρει ένα μη προσημασμένο byte· το C4 πακετάρει τέσσερα διαδοχικά. Στην έξοδο, το ίδιο πρότυπο τα ανασύρει σε μια λίστα τεσσάρων ακεραίων. Κάθε πιο πολύπλοκο δυαδικό πρότυπο χτίζεται πάνω σε αυτή την ιδέα.

4.11.4 Συμβάσεις στα παραδείγματα

- Η αναμενόμενη έξοδος εμφανίζεται ως ενσωματωμένο σχόλιο # … στην ίδια γραμμή με την έκφραση που την παρήγαγε, ή ακριβώς κάτω από το μπλοκ κώδικα.
- Τα παραδείγματα τρέχουν σε διερμηνέα προσανατολισμένο σε bytes - χωρίς use utf8, χωρίς pragma use open που αλλάζει στρώματα. Αν εμφανιστεί ένα filehandle, υποθέστε ότι το binmode $fh έχει ήδη κληθεί.

- Οι σταθερές πολλαπλών bytes όπως `"\xc0\xa8\x01\x2a"` γράφονται με δεκαεξαδικές διαφυγές αντί για ωμές τιμές bytes ώστε το εκτυπωμένο κείμενο να συμφωνεί με αυτό που βλέπετε στον κώδικα.

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 Πού να πάτε στη συνέχεια

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.

Επόμενο

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.

Γιατί έχει σημασία

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>) {
    # $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.

Επόμενο

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 `fork`s. In the child, `stdout` is connected to the write end of a pipe, and the command is `exec`ed.
- 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.

Επόμενο

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: $!";
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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;

# $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.

Επόμενο

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.

Δύο πρακτικές σημειώσεις:

- `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.

Πού να πάτε στη συνέχεια

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 Διασταυρούμενοι σύνδεσμοι αναφοράς

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` - η συνάρτηση πίσω από το `<$fh>`
- `sysopen` - low-level open with flag bits
- `sysread` - unbuffered read
- `syswrite` - unbuffered write

4.13 Αναφορές - ένας οδηγός εκμάθησης

Μια **αναφορά** στην Perl είναι ένα βαθμωτό που δείχνει σε μια άλλη τιμή: έναν πίνακα, ένα κατακερματισμό, μια υπορουτίνα, ένα άλλο βαθμωτό, ακόμα και μια άλλη αναφορά. Οι αναφορές είναι ο μοναδικός μηχανισμός που μετατρέπει τους τρεις τύπους κοντέινερ της Perl - βαθμωτά, πίνακες, κατακερματισμούς - σε αυθαίρετα εμφωλευμένες δομές δεδομένων.

Χωρίς αναφορές δεν μπορείτε να κατασκευάσετε κατακερματισμό πινάκων, πίνακα κατακερματισμών, δέντρο, γράφο, εγγραφή με μικτά πεδία, πίνακα callback, ή closure. Με αυτές, μπορείτε. Αυτός ο οδηγός σας οδηγεί από τους δύο τελεστές δημιουργίας αναφορών, μέσα από τη σύνταξη για την ανάκτηση της υποκείμενης τιμής, μέχρι τους ανώνυμους κατασκευαστές, τις αναφορές κώδικα, και το τέχνασμα ασθενών αναφορών που εμποδίζει τη διαρροή κυκλικών δομών.

4.13.1 Σε ποιους απευθύνεται

Γράφετε ήδη λειτουργικό κώδικα Perl. Χρησιμοποιείτε πίνακες και κατακερματισμούς άνετα, αλλά τη στιγμή που το πρόβλημα απαιτεί έναν κατακερματισμό του οποίου οι τιμές είναι πίνακες - ή έναν πίνακα εγγραφών, ή ένα callback που μπορείτε να αποθηκεύσετε σε πίνακα - η σύνταξη παύει να είναι προφανής. Αυτό το κενό συμπληρώνουν οι αναφορές, και αυτό το κενό κλείνει αυτός ο οδηγός.

4.13.2 Πώς είναι οργανωμένος αυτός ο οδηγός εκμάθησης

Κάθε κεφάλαιο εισάγει μία έννοια και στέκεται αυτόνομα. Διαβάστε τα με τη σειρά την πρώτη φορά· μεταπηδήστε στο σχετικό κεφάλαιο αργότερα.

Βασικά - λήψη αναφορών και χρήση τους

Υπάρχουν μόνο δύο τρόποι για να **φτιάξετε** μια αναφορά σε κάτι που έχει ήδη όνομα, και μόνο δύο τρόποι για να **χρησιμοποιήσετε** μια αναφορά μόλις την έχετε. Αυτό το κεφάλαιο καλύπτει και τους δύο. Οι ανώνυμοι κατασκευαστές (`[...]`, `{...}`) καλύπτονται στις *Ανώνυμες αναφορές*.

Δημιουργία αναφοράς με ανάστροφη κάθετο

Βάλτε μια ανάστροφη κάθετο μπροστά από οποιαδήποτε μεταβλητή, και παίρνετε ένα βαθμωτό που αναφέρεται σε αυτήν:

```
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
```

Μια αναφορά είναι η ίδια βαθμωτό. Μπορείτε να την αντιγράψετε, να την περάσετε, να την αποθηκεύσετε σε άλλη δομή δεδομένων, χωρίς να αγγίξετε αυτό στο οποίο δείχνει:

```
my $copy   = $aref;    # two scalars, one underlying array
my @slots;
$slots[3]  = $href;    # hash reference stored as an array element
my $pulled = $slots[3];
```

Και το `$aref` και το `$copy` δείχνουν στον **ίδιο** πίνακα. Η αλλαγή του πίνακα μέσω του ενός είναι ορατή μέσω του άλλου. Για να αντιγράψετε πραγματικά τα περιεχόμενα, δείτε τις *Ανώνυμες αναφορές*.

Πώς εκτυπώνεται μια αναφορά

Μετατρέψτε μια αναφορά σε συμβολοσειρά και παίρνετε μια ετικέτα τύπου συν μια διεύθυνση:

```
print "$aref\n";        # ARRAY(0x55d3c1a02b40)
print "$href\n";        # HASH(0x55d3c1a02be0)
```

Αν δείτε ποτέ τέτοια έξοδο κατά λάθος, εκτυπώσατε την αναφορά εκεί που εννοούσατε να εκτυπώσετε τα περιεχόμενα.

Χρήση αναφοράς - η μορφή με αγκύλες

Όπου θα μπορούσατε να γράψετε όνομα μεταβλητής, μπορείτε να βάλετε την αναφορά σε αγκύλες αντί αυτής. Ο κανόνας είναι μηχανικός:

Λειτουργία επί του @array	Ίδια λειτουργία μέσω $aref
`@array`	`@{$aref}`
`scalar @array`	`scalar @{$aref}`
`$array[3]`	`${$aref}[3]`
`$array[3] = 17`	`${$aref}[3] = 17`
`for my $x (@array) { ... }`	`for my $x (@{$aref}) { ... }`

Οι κατακερματισμοί ακολουθούν το ίδιο μοτίβο:

Λειτουργία επί του %hash	Ίδια λειτουργία μέσω $href
`%hash`	`%{$href}`
`keys %hash`	`keys %{$href}`
`$hash{red}`	`${$href}{red}`
`$hash{red} = 17`	`${$href}{red} = 17`

Και τα βαθμωτά:

```
my $x   = 10;
my $ref = \$x;
print ${$ref};          # 10
${$ref} = 99;
print $x;               # 99
```

Οι αγκύλες μπορούν συνήθως να παραλειφθούν όταν αυτό που βρίσκεται μέσα είναι μια απλή βαθμωτή μεταβλητή: το `@$aref` σημαίνει το ίδιο με το `@{$aref}`, και το `$$sref` σημαίνει το ίδιο με το `${$sref}`. Ξεκινήστε με τη μορφή με αγκύλες μέχρι η σύντομη μορφή να παύσει να μοιάζει αμφίσημη· τόσο το `$$hash{key}` όσο και το `${$hash}{key}` λειτουργούν, αλλά το `$$table{$country}[0]` παρανοείται ευκολότερα από το `${$table}{$country}[0]`.

Χρήση αναφοράς - η μορφή με βέλος

Το πιο συνηθισμένο πράγμα που κάνετε με μια αναφορά είναι να ανακτήσετε ένα στοιχείο από τον πίνακα ή κατακερματισμό στον οποίο δείχνει. Η μορφή με αγκύλες λειτουργεί (`${$aref}[3]`) αλλά είναι δυσανάγνωστη. Η μορφή με βέλος είναι η συντομογραφία:

```
$aref->[3]              # same as ${$aref}[3]
$href->{red}            # same as ${$href}{red}
```

Το βέλος δεν είναι καλλωπιστικό. Είναι αυτό που διακρίνει δύο διαφορετικές σημασίες:

```
$aref->[3]              # fourth element of the array $aref points at
$aref[3]                # fourth element of the unrelated array @aref
```

Τα `$aref` και `@aref` είναι ξεχωριστές μεταβλητές, τόσο άσχετες όσο τα `$x` και `@x`. Το ξεχασμένο βέλος ανακτά σιωπηλά από έναν διαφορετικό πίνακα που πιθανότατα δεν υπάρχει. Η εκτέλεση υπό `use strict` μετατρέπει το λάθος σε σφάλμα μεταγλώττισης αντί για σιωπηλή λάθος απάντηση κατά τον χρόνο εκτέλεσης.

Ρωτώντας τι δείχνει μια αναφορά

Η ενσωματωμένη `ref` επιστρέφει μια συμβολοσειρά που περιγράφει το είδος του αναφερόμενου:

```
ref $aref               # ARRAY
ref $href               # HASH
ref $sref               # SCALAR
ref \&printer           # CODE
ref $n                  # empty string - $n isn't a reference
```

Συνδυάστε με λογικό περιβάλλον για να ελέγξετε «είναι αυτό αναφορά;»:

```
if (ref $maybe) {
    # it's a reference of some kind
}
if (ref $maybe eq 'ARRAY') {
    # it's specifically an array reference
}
```

Δείτε την `ref` για την πλήρη λίστα συμβολοσειρών επιστροφής (συμπεριλαμβανομένων `CODE`, `GLOB`, `REF`, `Regexp`, και του ονόματος πακέτου όταν η αναφορά έχει γίνει blessed σε μια κλάση).

Δύο συνηθισμένα λάθη αρχαρίων

- **Αντιμετώπιση μιας αναφοράς σαν να ήταν το υποκείμενο κοντέινερ.** Το `push $aref, 1` δεν προσθέτει στον πίνακα· είναι σφάλμα τύπου. Πρέπει να αποαναφέρετε: `push @{$aref}, 1` ή (ισοδύναμα) `push @$aref, 1`.
- **Παράλειψη του βέλους μεταξύ μεταβλητής και του δείκτη της.** Το `$aref[3]` δεν είναι το ίδιο με το `$aref->[3]`. Το πρώτο διαβάζει από τον `@aref`, το δεύτερο διαβάζει μέσω της αναφοράς.

Πού να πάτε στη συνέχεια

- *Πίνακες πινάκων* - η δισδιάστατη περίπτωση, και ο κανόνας του βέλους-μεταξύ-δεικτών.
- *Ανώνυμες αναφορές* - `[...]` και `{...}` για κατασκευή δομών χωρίς κατονομασμένα ενδιάμεσα.

Πίνακες πινάκων

Οι πίνακες της Perl κρατούν βαθμωτά, και τα βαθμωτά δεν μπορούν να περιέχουν άμεσα άλλους πίνακες. Για να κατασκευάσετε έναν δισδιάστατο πίνακα - μήτρα, πλέγμα, πίνακα γραμμών - αποθηκεύετε **αναφορές** σε εσωτερικούς πίνακες ως στοιχεία του εξωτερικού πίνακα. Αυτό το κεφάλαιο δείχνει πώς.

Κατασκευή μιας μήτρας

Η πιο άμεση μορφή χρησιμοποιεί ανώνυμους κατασκευαστές πινάκων (`[...]`, καλύπτονται αναλυτικά στις *Ανώνυμες αναφορές*) ώστε να μη χρειάζεται να ονομάσετε κάθε εσωτερικό πίνακα:

```
my @m = (
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
);
```

Ο `@m` έχει τρία στοιχεία. Κάθε στοιχείο είναι ένα βαθμωτό που κρατάει αναφορά πίνακα. Υπάρχουν τέσσερις πίνακες στο παιχνίδι: ο εξωτερικός `@m` και τρεις ανώνυμοι εσωτερικοί πίνακες.

Ανάγνωση και εγγραφή στοιχείων

Το `$m[1]` είναι η αναφορά στη δεύτερη γραμμή. Για να φτάσετε στο τρίτο στοιχείο αυτής της γραμμής, εφαρμόστε το βέλος:

```
$m[1]->[2]             # 6
$m[0]->[0]             # 1
$m[2]->[2] = 99;       # overwrite the bottom-right cell
```

Μεταξύ **δύο δεικτών**, το βέλος είναι προαιρετικό. Αυτός ο κανόνας είναι που κάνει την πολυδιάστατη πρόσβαση ευανάγνωστη:

```
$m[1][2]               # same as $m[1]->[2]
$m[2][2] = 99;         # same as $m[2]->[2] = 99
```

Το βέλος παραμένει **απαραίτητο** μεταξύ μιας βαθμωτής μεταβλητής και του πρώτου της δείκτη: το `$aref->[0]` δεν μπορεί να συντομευθεί σε `$aref[0]` επειδή αυτό θα σήμαινε «πρώτο στοιχείο του `@aref`». Η συντόμευση εφαρμόζεται μόνο **εντός** μιας αλυσίδας δεικτών.

Για τρεις διαστάσεις είναι η ίδια ιδέα:

```
$cube[2][3][5]          # readable
${${$cube[2]}[3]}[5]    # the same expression fully spelled out
```

Η δεύτερη μορφή είναι αυτό που θα έπρεπε να γράψετε χωρίς τη συντόμευση με βέλος. Κανείς δεν το κάνει αυτό.

Επανάληψη πάνω σε μια μήτρα

Για να διασχίσετε γραμμές, κάντε βρόχο πάνω στον εξωτερικό πίνακα· κάθε επανάληψη σας δίνει μια αναφορά πίνακα. Για να διασχίσετε κελιά, αποαναφέρετε αυτή την αναφορά μέσα στον βρόχο:

```
for my $row (@m) {
    for my $cell (@{$row}) {
        print "$cell ";
    }
    print "\n";
}
```

Αν χρειάζεστε επίσης τους δείκτες, χρησιμοποιήστε αριθμημένο βρόχο πάνω στον εξωτερικό πίνακα και στον αποαναφερόμενο εσωτερικό:

```
for my $i (0 .. $#m) {
    for my $j (0 .. $#{$m[$i]}) {
        print "m[$i][$j] = $m[$i][$j]\n";
    }
}
```

Το `$#{$m[$i]}` είναι ο τελεστής τελευταίου δείκτη εφαρμοσμένος στον αποαναφερόμενο εσωτερικό πίνακα. Είναι η μορφή `$#array` με το `{$m[$i]}` να αντικαθιστά το όνομα του πίνακα.

Επέκταση γραμμών και στηλών

Προσαρτήστε σε μια γραμμή κάνοντας push στον εσωτερικό πίνακα μέσω αναφοράς:

```
push @{$m[0]}, 10;      # row 0 is now (1, 2, 3, 10)
```

Προσαρτήστε ολόκληρη γραμμή στη μήτρα κάνοντας push μιας αναφοράς πίνακα στον εξωτερικό πίνακα:

```
push @m, [10, 11, 12]; # @m now has four rows
```

Και οι δύο λειτουργίες χρησιμοποιούν την `push` - μία φορά σε αποαναφερόμενο εσωτερικό πίνακα, μία φορά απευθείας στον εξωτερικό πίνακα.

Αυτο-δημιουργία

Η ανάθεση στο `$m[$i][$j]` λειτουργεί ακόμα και αν το `$m[$i]` δεν υπάρχει ακόμα:

```
my @grid;
$grid[5][7] = 'X';
```

Αυτό δημιουργεί τον `@grid` με έξι κενές θυρίδες και μία αναφορά στη θυρίδα 5. Αυτή η αναφορά δείχνει σε έναν φρέσκο ανώνυμο πίνακα του οποίου η όγδοη θυρίδα κρατάει `'X'`. Η Perl δημιουργεί αυτόματα και την εξωτερική θυρίδα και τον εσωτερικό πίνακα. Αυτό λέγεται **αυτο-δημιουργία** και είναι ο λόγος που σχεδόν ποτέ δεν χρειάζεται να γράψετε `$m[$i] = []` πριν τη χρήση.

Η αυτο-δημιουργία ενεργοποιείται κατά την *ανάθεση* και κατά την *αποαναφορά σε περιβάλλον lvalue*. Οι σκέτες αναγνώσεις **δεν** αυτο-δημιουργούν:

```
my @grid;
my $x = $grid[5][7];   # $x is undef; no inner array created
exists $grid[5];       # false
```

Χρησιμοποιήστε την `defined` για να ξεχωρίσετε την περίπτωση «δεν υπήρξε ποτέ» από την περίπτωση «υπάρχει και κρατάει undef», και την `exists` για να ελέγξετε την ίδια τη θυρίδα χωρίς να την αυτο-δημιουργήσετε.

Συνηθισμένο λάθος: κοινή χρήση ενός μοναδικού εσωτερικού πίνακα

Αυτό μοιάζει με τρεις γραμμές αλλά είναι στην πραγματικότητα μία γραμμή επαναλαμβανόμενη:

```
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
```

Για να πάρετε τρεις ανεξάρτητες γραμμές, κατασκευάστε καθεμία ξεχωριστά. Η μορφή ανώνυμου κατασκευαστή το χειρίζεται κατανέμοντας πάντα φρέσκα:

```
my @good = ([0, 0, 0], [0, 0, 0], [0, 0, 0]);
$good[0][1] = 9;
print "$good[2][1]\n"; # 0
```

Ένας βρόχος μέσω `map` δίνει το ίδιο αποτέλεσμα για γραμμές μεταβλητού μήκους:

```
my @grid = map { [ (0) x 10 ] } 1 .. 10;  # 10×10 of zeros
```

Συνηθισμένο λάθος: αποθήκευση του πλήθους αντί των στοιχείων

Η ανάθεση ενός πίνακα σε μια μεμονωμένη βαθμωτή θυρίδα αποτιμά τον πίνακα σε **βαθμωτό περιβάλλον**, που δίνει το *πλήθος* των στοιχείων του, όχι το περιεχόμενό του:

```
for my $i (1 .. 10) {
    my @array = somefunc($i);
    $AoA[$i] = @array;          # WRONG - stores the count
}
```

Μετά από αυτόν τον βρόχο κάθε `$AoA[$i]` είναι ένας αριθμός. Για να αποθηκεύσετε τα στοιχεία, τυλίξτε τα σε έναν κατασκευαστή ανώνυμου πίνακα ώστε να πάρετε μια αναφορά σε ένα φρέσκο αντίγραφο:

```
$AoA[$i] = [ @array ];          # right - a new array, copied
```

Αν πραγματικά θέλατε το πλήθος, πείτε το με `scalar` και ένα όνομα που το αντικατοπτρίζει:

```
$counts[$i] = scalar @array;    # explicit - no surprise later
```

Συνηθισμένο λάθος: ένας επαναχρησιμοποιημένος πίνακας πίσω από κάθε αναφορά

Η παγίδα του επαναλαμβανόμενου πίνακα από την προηγούμενη ενότητα εμφανίζεται επίσης όταν ο κοινόχρηστος πίνακας επαναχρησιμοποιείται μεταξύ επαναλήψεων του βρόχου:

```
our @array;
for my $i (1 .. 10) {
    @array = somefunc($i);
    $AoA[$i] = \@array;         # WRONG - same array every time
}
```

Κάθε `\@array` είναι μια αναφορά στον *ίδιο* πίνακα, οπότε και οι δέκα θυρίδες καταλήγουν να δείχνουν σε ό,τι επέστρεψε τελευταίο η `somefunc`. Και πάλι, το `[ @array ]` το διορθώνει μέσω αντιγραφής.

Η δήλωση του πίνακα με `my` **μέσα** στον βρόχο είναι ασφαλής, ωστόσο:

```
for my $i (1 .. 10) {
    my @array = somefunc($i);
    $AoA[$i] = \@array;         # fine - fresh array each iteration
}
```

Το `my @array` χτίζει έναν νέο πίνακα σε κάθε πέρασμα, οπότε κάθε `\@array` αναφέρεται σε έναν ξεχωριστό. Η αναφορά επιβιώνει του σώματος του βρόχου επειδή ο πίνακας στον οποίο δείχνει παραμένει ζωντανός όσο παραμένει και η αναφορά. Η μορφή κατασκευαστή `[ @array ]` είναι ακόμη η σαφέστερη επιλογή· καταφύγετε στη μορφή `\@array` μόνο όταν καταλαβαίνετε γιατί λειτουργεί εδώ.

Πού να πάτε στη συνέχεια

- *Κατακερματισμοί και μείξεις* - κατακερματισμοί πινάκων και πίνακες κατακερματισμών, τα δύο σχήματα που καλύπτουν τις περισσότερες εγγραφές του πραγματικού κόσμου.
- *Ανώνυμες αναφορές* - οι κατασκευαστές `[...]` / `{...}` που χρησιμοποιήθηκαν παραπάνω, αναλυτικά.

Κατακερματισμοί και μικτές δομές

Οι τιμές ενός κατακερματισμού είναι βαθμωτά, οπότε μια τιμή δεν μπορεί κυριολεκτικά να **είναι** πίνακας ή άλλος κατακερματισμός. Μπορεί ωστόσο να είναι **αναφορά** σε έναν - και αυτό αρκεί για να κατασκευάσετε κάθε σχήμα εγγραφής που χρειάζονται τα πραγματικά προγράμματα: κατακερματισμοί πινάκων, πίνακες κατακερματισμών, κατακερματισμοί κατακερματισμών, και εμφωλευμένοι συνδυασμοί.

Αυτό το κεφάλαιο διατρέχει τα τρία πιο συνηθισμένα σχήματα. Τα μοτίβα συντίθενται: μόλις έχετε έναν κατακερματισμό πινάκων και έναν πίνακα κατακερματισμών, έχετε δωρεάν «γραμμές εγγραφών με πεδία πολλαπλών τιμών».

Κατακερματισμός πινάκων - ομαδοποίηση τιμών κάτω από ένα κλειδί

Το παράδειγμα που δίνει το κίνητρο: διαβάζετε γραμμές της μορφής `city, country` και θέλετε να συλλέξετε, για κάθε χώρα, τη λίστα των πόλεών της.

```
Chicago, USA
Frankfurt, Germany
Berlin, Germany
Washington, USA
Helsinki, Finland
New York, USA
```

Το φυσικό σχήμα είναι ένας κατακερματισμός του οποίου τα κλειδιά είναι ονόματα χωρών και οι τιμές είναι αναφορές σε πίνακες πόλεων:

```
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";
}
```

Δύο πράγματα προς προσοχή:

- Το `push @{$cities_by_country{$country}}, $city` λειτουργεί από την πρώτη κιόλας επανάληψη για κάθε νέα χώρα. Η εγγραφή hash δεν υπάρχει ακόμη, οπότε η Perl δημιουργεί έναν ανώνυμο πίνακα για αυτήν, αποθηκεύει μια αναφορά, και σπρώχνει μέσα την πόλη. Αυτό είναι πάλι αυτο-δημιουργία
  - και ο λόγος που σχεδόν ποτέ δεν χρειάζεστε το `$h{$k} = [] unless exists $h{$k}` ως ξεχωριστή γραμμή.
- Η αποαναφορά `@{$cities_by_country{$country}}` είναι μηχανική: «ο πίνακας στον οποίο δείχνει αυτή η τιμή του κατακερματισμού». Όπου θα γράφατε `@cities` αν ήταν κατονομασμένος πίνακας, γράφετε τη μορφή με αγκύλες αντί αυτής.

Η έξοδος είναι:

```
Finland: Helsinki.
Germany: Berlin, Frankfurt.
USA: Chicago, New York, Washington.
```

Πίνακας κατακερματισμών - πίνακας εγγραφών

Όταν κάθε στοιχείο σε μια λίστα κουβαλάει αρκετά κατονομασμένα πεδία, αποθηκεύστε κάθε εγγραφή ως αναφορά κατακερματισμού και βάλτε τις αναφορές σε έναν πίνακα:

```
my @people = (
    { name => 'Ada',   born => 1815, field => 'computing' },
    { name => 'Hedy',  born => 1914, field => 'signals'   },
    { name => 'Grace', born => 1906, field => 'compilers' },
);
```

Το `$people[0]` είναι αναφορά κατακερματισμού· το `$people[0]->{name}` ή (με τη συντόμευση του βέλους-μεταξύ-δεικτών) το `$people[0]{name}` είναι `'Ada'`.

Τυπικές λειτουργίες:

```
# 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';
```

Μέσα στις `sort`, `grep`, και `map` η εγγραφή είναι αναφορά κατακερματισμού, οπότε φτάνετε στα πεδία της μέσω του βέλους. Τα `$a` και `$b` μέσα σε έναν συγκριτή `sort` είναι εδώ αναφορές κατακερματισμών, όχι κατακερματισμοί.

Κατακερματισμός κατακερματισμών - πίνακες αναζήτησης με κλειδιά δομημένες τιμές

Όταν κάθε εγγραφή έχει ως κλειδί ένα μοναδικό αναγνωριστικό, ένας κατακερματισμός κατακερματισμών είναι συνήθως πιο όμορφος από έναν πίνακα κατακερματισμών:

```
my %people = (
    ada   => { born => 1815, field => 'computing' },
    hedy  => { born => 1914, field => 'signals'   },
    grace => { born => 1906, field => 'compilers' },
);

print $people{ada}{born};              # 1815
$people{grace}{field} = 'COBOL';       # mutate one record
```

Επαναλάβετε σε καθορισμένη σειρά ταξινομώντας τα κλειδιά:

```
for my $key (sort keys %people) {
    my $rec = $people{$key};
    print "$key: born $rec->{born}, $rec->{field}\n";
}
```

Η σύνδεση της εσωτερικής αναφοράς σε μια κατονομασμένη λεξιλογική (`$rec`) είναι συνήθεια που αξίζει να αποκτήσετε - κάνει τον κώδικα να διαβάζεται από πάνω προς τα κάτω αντί να επαναλαμβάνει το `$people{$key}{field}` πέντε φορές.

Μικτό - εγγραφές με πεδία πολλαπλών τιμών

Συνθέστε τα όλα μαζί: κάθε εγγραφή είναι ένας κατακερματισμός, και ένα από τα πεδία της είναι αναφορά σε πίνακα.

```
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";
}
```

Το `@{$rec->{genres}}` είναι το ίδιο μοτίβο όπως πριν: φτάστε μέσα στον κατακερματισμό, πάρτε την αναφορά πίνακα, αποαναφέρετέ την με `@{...}`.

Ένα πεδίο μπορεί εξίσου εύκολα να κρατά μια **αναφορά κώδικα** ή ένα εμφωλευμένο **hash αναζήτησης**. Τίποτα στην πρόσβαση δεν αλλάζει· ακολουθείτε το ίδιο βέλος μέσα σε ό,τι δείχνει το κάθε πεδίο:

```
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'
```

Το `$rec->{CALLBACK}` είναι μια αναφορά κώδικα, οπότε το ακολουθούν `->($arg)` την

καλεί. Το `$rec->{LOOKUP}` είναι μια αναφορά hash, οπότε το `{red}` δεικτοδοτεί μέσα από αυτήν (το βέλος ανάμεσα στις δύο δεικτοδοτήσεις παραλείπεται).

Η αντιγραφή ενός πίνακα αναφορών αντιγράφει τον πίνακα, όχι τους αναφερόμενους

Όταν μια εγγραφή κρατά έναν πίνακα αναφορών, η αντιγραφή αυτού του πίνακα με `[ @list ]` φτιάχνει έναν *νέο* πίνακα του οποίου τα στοιχεία είναι οι *ίδιες* αναφορές. Το αντίγραφο και το πρωτότυπο μοιράζονται τους αναφερόμενούς τους, οπότε μια μεταβολή μέσα από μία διαδρομή είναι ορατή μέσα από κάθε διαδρομή που κρατά αυτές τις ίδιες αναφορές στοιχείων:

```
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
```

Το `@also` είναι ένας ξεχωριστός πίνακας, οπότε το σπρώξιμο πάνω του δεν μεγαλώνει το πρωτότυπο. Αλλά τα στοιχεία του είναι ακριβώς οι ίδιες αναφορές hash, οπότε η αύξηση του age μέσω του `$TV{simpsons}{kids}[0]` αλλάζει το hash στο οποίο δείχνει επίσης το `$also[0]`. Για να πάρετε ανεξάρτητες εγγραφές πρέπει να αντιγράψετε ένα επίπεδο βαθύτερα, χτίζοντας φρέσκα hashes αντί να επαναχρησιμοποιείτε τις αναφορές τους.

Έλεγχος του τι έχετε

Πριν ενεργήσετε σε μια εμφωλευμένη θυρίδα συχνά είναι χρήσιμο να γνωρίζετε αν η θυρίδα υπάρχει πραγματικά εκεί. Χρησιμοποιήστε την `exists` αντί της `defined` όταν θέλετε να αποφύγετε την αυτο-δημιουργία κενών ενδιάμεσων δομών:

```
if (exists $library{$title} and exists $library{$title}{genres}) {
    push @{$library{$title}{genres}}, 'essential';
}
```

Η ανάγνωση του `$library{$title}{genres}` σε περιβάλλον lvalue - η αριστερή πλευρά μιας ανάθεσης, ή ως όρισμα της `push` - αυτο-δημιουργεί επίπεδα που λείπουν. Η ανάγνωσή του σε περιβάλλον rvalue όχι. Οι έλεγχοι `exists` παραπάνω κρατούν τη σάρωση μόνο για ανάγνωση.

Για να ελέγξετε τον τύπο μιας αναφοράς χωρίς να βγάλετε τιμές από αυτή, χρησιμοποιήστε την `ref`:

```
ref $library{'The Road'}              # HASH
ref $library{'The Road'}{genres}      # ARRAY
```

Πού να πάτε στη συνέχεια

- *Ανώνυμες αναφορές* - οι κατασκευαστές `[...]` / `{...}` που χρησιμοποιήθηκαν σε όλο αυτό το κεφάλαιο, και πότε μια κατονομασμένη μεταβλητή είναι η καλύτερη επιλογή.
- *Αναφορές υπορουτινών* - η εναπομένουσα γεύση αναφοράς, απαραίτητη για πίνακες callback και αποστολή.

Ανώνυμες αναφορές - `[...]` και `{...}`

Δεν θέλετε πάντα να δώσετε όνομα σε έναν εσωτερικό πίνακα ή κατακερματισμό πριν τον αναφέρετε. Η Perl παρέχει δύο κατασκευαστές που χτίζουν ένα νέο κοντέινερ και σας παραδίδουν αναφορά σε αυτό σε μία έκφραση:

- Το `[...]` χτίζει έναν φρέσκο ανώνυμο **πίνακα** και επιστρέφει αναφορά σε αυτόν.
- Το `{...}` χτίζει έναν φρέσκο ανώνυμο **κατακερματισμό** και επιστρέφει αναφορά σε αυτόν.

Αυτοί είναι ο κύριος τρόπος συναρμολόγησης εμφωλευμένων δομών δεδομένων.

Πίνακας: `[...]`

```
my $aref = [1, 'foo', undef, 13];
```

Η δεξιά πλευρά αποτιμά τη λίστα `(1, 'foo', undef, 13)`, κατανέμει έναν νέο πίνακα, αντιγράφει αυτά τα τέσσερα στοιχεία μέσα, και αποδίδει ένα βαθμωτό που κρατάει αναφορά σε αυτόν τον πίνακα. Αυτό είναι **ακριβώς** ισοδύναμο με τη μορφή δύο βημάτων με κατονομασμένο ενδιάμεσο:

```
my @array = (1, 'foo', undef, 13);
my $aref  = \@array;
```

...εκτός από το ότι καμία μεταβλητή `@array` δεν διαρρέει στην περιβάλλουσα εμβέλεια. Ο ανώνυμος πίνακας δεν έχει όνομα· υπάρχει μόνο η αναφορά, και ο πίνακας παραμένει ζωντανός όσο παραμένει και η αναφορά.

Οι κενές μορφές είναι χρήσιμες ως αρχικές τιμές:

```
my $aref = [];          # reference to a new empty array
push @{$aref}, 1, 2;    # now [1, 2]
```

Κατακερματισμός: `{...}`

```
my $href = { APR => 4, AUG => 8 };
```

Ίδια ιδέα για τους κατακερματισμούς. Η έκφραση είναι μια λίστα ζευγών κλειδιού/τιμής (το `=>` είναι απλώς κόμμα που αυτο-εισαγωγοποιεί την αριστερή του πλευρά), ένας νέος κατακερματισμός κατανέμεται, τα ζεύγη εισάγονται, και παίρνετε πίσω μια αναφορά:

```
my %hash = (APR => 4, AUG => 8);
my $href = \%hash;
```

είναι το ισοδύναμο δύο βημάτων. Και όπως με τους πίνακες, η κενή μορφή είναι συνηθισμένο σημείο εκκίνησης:

```
my $href = {};
$href->{red} = 17;
```

`{...}` έναντι μπλοκ - μια γωνία του αναλυτή

Ένα γυμνό `{...}` σε θέση πρότασης αναλύεται ως **μπλοκ**, όχι ως κατασκευαστής κατακερματισμού:

```
{ foo => 1 };           # block, with `foo => 1` as its expression
```

Στις περισσότερες θέσεις - στα δεξιά του =, μέσα σε λίστα, ως όρισμα σε συνάρτηση - η Perl εντοπίζει αυτόματα την ερμηνεία του κατασκευαστή κατακερματισμού. Αν χρειαστεί ποτέ να επιβάλετε την ερμηνεία του κατασκευαστή κατακερματισμού, βάλτε + μπροστά:

```
my $x = +{ foo => 1 }; # unambiguous: always a hash reference
```

Σπάνια θα το χρειαστείτε αυτό. Μέσα σε `return +{...}` από υπορουτίνα και σε μερικές άλλες θέσεις «τύπου πρότασης», είναι η λύση.

Εμφώλευση - κατασκευαστές μέσα σε κατασκευαστές

Επειδή κάθε κατασκευαστής αποδίδει βαθμωτό, και τα βαθμωτά πάνε παντού, συνθέτετε δομές με εμφώλευση:

```
my $tree = {
    name     => 'root',
    children => [
        { name => 'left',  children => [] },
        { name => 'right', children => [
            { name => 'r1', children => [] },
        ] },
    ],
};

print $tree->{children}[1]{children}[0]{name};   # r1
```

Κάθε `[...]` και `{...}` κατανέμει φρέσκα. Σε αντίθεση με την παγίδα «κοινού εσωτερικού πίνακα» από τους *Πίνακες πινάκων*, οι εμφωλευμένοι κατασκευαστές δεν μοιράζονται ποτέ αποθήκευση κατά λάθος.

Κατονομασμένα ενδιάμεσα έναντι ανώνυμων κατασκευαστών

Και οι δύο μορφές λειτουργούν. Επιλέξτε με βάση την αναγνωσιμότητα:

- **Ανώνυμος κατασκευαστής** όταν το κοντέινερ έχει έναν σκοπό και ο μοναδικός του ρόλος είναι «να είναι η τιμή αυτού του πεδίου». Εμφωλευμένα κυριολεκτικά δεδομένων, μίας χρήσης πίνακες αναζήτησης, εγγραφές πίνακα callback.

- **Κατονομασμένο ενδιάμεσο** όταν χτίζετε το κοντέινερ σταδιακά, ή όταν θέλετε να χρησιμοποιήσετε τελεστές πίνακα/κατακερματισμού πάνω του απευθείας χωρίς να επαναλαμβάνετε `@{...}` / `%{...}`:

```
my @row;
for my $x (@source) {
    push @row, transform($x);
}
$grid[$i] = \@row;
```

Αντιγραφή έναντι κοινής χρήσης

Οι ανώνυμοι κατασκευαστές πάντα κατανέμουν· δεν μοιράζονται ποτέ. Αλλά η ανάθεση μιας αναφοράς σε άλλο βαθμωτό δεν αντιγράφει τίποτα:

```
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
```

Για να πάρετε ένα ρηχό αντίγραφο του υποκείμενου κοντέινερ, αποαναφέρετε μέσα σε έναν φρέσκο κατασκευαστή:

```
my $copy_aref = [ @{$a} ];      # new array, same elements
my $copy_href = { %{$h} };      # new hash, same keys/values
```

Η νέα αναφορά είναι ανεξάρτητη από την αρχική. Οι **τιμές** μέσα είναι ακόμα κοινές - αν αυτές οι τιμές είναι οι ίδιες αναφορές, οι δύο δομές δείχνουν ακόμα σε κοινά υποδέντρα. Αυτό είναι ρηχή αντιγραφή. Για βαθιά αντιγραφή, καταφύγετε σε άρθρωμα όπως το `Storable` (`dclone`).

Πού να πάτε στη συνέχεια

- *Αναφορές υπορουτινών* - η τρίτη γεύση ανώνυμου πράγματος, χρησιμοποιώντας `sub { ... }` αντί για `[...]` ή `{...}`.
- *Ασθενείς αναφορές* - απαραίτητες όταν μια εμφωλευμένη δομή δείχνει πίσω στον εαυτό της.

Αναφορές υπορουτινών

Η αναφορά υπορουτίνας είναι ένα βαθμωτό που δείχνει σε ένα κομμάτι κώδικα. Μόλις έχετε μία, μπορείτε να την αποθηκεύσετε σε μεταβλητή, να την περάσετε σε άλλη υπορουτίνα ως callback, να κρατάτε πίνακα από αυτές για αποστολή, ή να την καλέσετε. Οι αναφορές κώδικα είναι αυτό που κάνει πίνακες αποστολής, callbacks, επαναλήπτες, και closures εφικτά στην Perl.

Δύο τρόποι να πάρετε αναφορά κώδικα

\&name - λήψη αναφοράς σε κατονομασμένη υπορουτίνα:

```
sub greet { print "hello $_[0]\n" }

my $cref = \&greet;
$cref->('world');      # hello world
```

Το sigil & είναι αυτό που σημαδεύει την `greet` ως υπορουτίνα. Η ανάστροφη κάθετος μπροστά παίρνει την αναφορά της. Χωρίς το &, θα καλούσατε την υπορουτίνα και θα παίρνατε αναφορά στο αποτέλεσμα.

sub { ... } - μια **ανώνυμη** υπορουτίνα. Αποτιμάται σε αναφορά σε ένα φρέσκο, χωρίς όνομα κομμάτι κώδικα:

```
my $cref = sub { print "hello $_[0]\n" };
$cref->('world');
```

Οι ανώνυμες sub είναι το άμεσο ανάλογο των `[...]` και `{...}` από το προηγούμενο κεφάλαιο: ένας κατασκευαστής που σας δίνει πίσω αναφορά σε κάτι νέο και χωρίς όνομα.

Κλήση μέσω αναφοράς

Το `$cref->(@args)` είναι αυτό στο οποίο καταφεύγετε:

```
$cref->('world');
$cref->($x, $y);
$cref->();             # call with no arguments
```

Υπάρχει επίσης μια παλαιότερη σύνταξη, `&{$cref}(...)`, παράλληλη με τη μορφή `@{$aref}` / `%{$href}`. Παραμένει έγκυρη αλλά η μορφή με βέλος είναι η πρότυπη σε κάθε σύγχρονο πλαίσιο.

Μέσα στην καλούμενη υπορουτίνα, τα ορίσματα φτάνουν στο `@_` ως συνήθως:

```
my $max = sub {
    my ($a, $b) = @_;
    $a > $b ? $a : $b;
};
print $max->(3, 7);    # 7
```

Callbacks - πέρασμα κώδικα σε άλλη ρουτίνα

Οποιαδήποτε υπορουτίνα που δέχεται αναφορά κώδικα μπορεί να δεχτεί είτε αναφορά κατονομασμένης υπορουτίνας είτε ανώνυμη:

```
sub each_line {
    my ($filename, $cb) = @_;
    open my $fh, '<', $filename or die "$filename: $!";
    while (my $line = <$fh>) {
        $cb->($line);
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    }
}

# 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] });
```

Οι ενσωματωμένες ανώνυμες sub είναι η ιδιωματική μορφή όταν το σώμα του callback είναι σύντομο και ειδικό για το σημείο κλήσης.

Πίνακες αποστολής

Ένας κατακερματισμός αναφορών κώδικα αντικαθιστά μεγάλες αλυσίδες `if` / `elsif`:

```
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";
}
```

Κάθε τιμή στον `%dispatch` είναι αναφορά κώδικα. Η αναζήτηση με το όνομα εντολής και η κλήση μέσω της αναφοράς αντικαθιστά τη διακλάδωση με μία αναζήτηση. Η προσθήκη νέας εντολής είναι μία εγγραφή κατακερματισμού.

Closures - κώδικας που αποτυπώνει λεξιλογικές

Μια ανώνυμη υπορουτίνα που αναφέρεται σε μια λεξιλογική (`my`) μεταβλητή από την περιβάλλουσα εμβέλειά της **αποτυπώνει** αυτή τη μεταβλητή. Η μεταβλητή επιβιώνει όσο και η αναφορά της υπορουτίνας, και κάθε κλήση βλέπει την τρέχουσα τιμή:

```
sub make_counter {
    my $n = 0;
    return sub { ++$n };
}

my $a = make_counter();
my $b = make_counter();
print $a->();          # 1
print $a->();          # 2
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
print $b->();           # 1   - independent from $a
print $a->();           # 3
```

Κάθε κλήση της make_counter δημιουργεί μια **νέα** λεξιλογική $n και μια **νέα** ανώνυμη υπορουτίνα που αποτυπώνει αυτή την $n. Η υπορουτίνα που επιστράφηκε στο $a και αυτή που επιστράφηκε στο $b κλείνουν πάνω από διακριτές μεταβλητές, οπότε οι μετρητές τους δεν παρεμβαίνουν.

Έτσι εκφράζει η Perl τα περισσότερα από αυτά που άλλες γλώσσες αποκαλούν «αντικείμενα με ιδιωτική κατάσταση» χωρίς καμία μηχανή κλάσεων: οι κλεισμένες λεξιλογικές **είναι** η ιδιωτική κατάσταση.

Ένα closure αποτυπώνει κατά αναφορά, όχι κατά αντιγραφή. Ένας βρόχος όπως αυτός αποτυπώνει την **ίδια** μεταβλητή βρόχου:

```
my @subs;
for my $i (1 .. 3) {
    push @subs, sub { print "i=$i\n" };
}
$_->() for @subs;       # i=1 / i=2 / i=3
```

...επειδή το my $i στο for my $i δηλώνεται εκ νέου με my σε κάθε επανάληψη, οπότε κάθε closure αποτυπώνει μια *διαφορετική* $i. Το συνηθισμένο λάθος είναι η χρήση μεταβλητής βρόχου for που δεν δηλώνεται φρέσκα με my:

```
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
```

Η λύση είναι for my $i (...) - προτιμάτε πάντα το ενσωματωμένο my.

Ενδοσκόπηση

Η ref σε αναφορά κώδικα επιστρέφει τη συμβολοσειρά CODE:

```
ref $cref               # CODE
ref \&greet             # CODE
```

Η defined λειτουργεί ως συνήθως για να ξεχωρίσει το «καμία τιμή» από το «οποιαδήποτε τιμή, συμπεριλαμβανομένης αναφοράς κώδικα»:

```
if (defined $dispatch{$cmd}) { ... }
```

Πού να πάτε στη συνέχεια

- *Ασθενείς αναφορές* - σχετικές όταν μια δομή αναφορών κώδικα κρατάει αναφορές πίσω σε αντικείμενα που την κατέχουν (εκπομποί συμβάντων, αλυσίδες παρατηρητών).

Ασθενείς αναφορές

Η Perl παρακολουθεί τη διάρκεια ζωής αντικειμένων με **μέτρηση αναφορών**. Κάθε τιμή γνωρίζει πόσες αναφορές δείχνουν σε αυτήν· όταν ο μετρητής πέφτει στο μηδέν, η τιμή απελευθερώνεται. Αυτό λειτουργεί καλά για δεδομένα σε σχήμα δέντρου, όπου οι αναφορές ρέουν προς μία κατεύθυνση, από γονέα σε παιδί. Αποτυγχάνει για **κύκλους**, όπου μια αλυσίδα αναφορών εν τέλει δείχνει πίσω σε κάτι νωρίτερα στην αλυσίδα. Τα μέλη του κύκλου κρατούν το ένα το άλλο ζωντανό ακόμα και όταν τίποτα έξω από τον κύκλο δεν αναφέρεται πλέον σε αυτά.

Μια **ασθενής αναφορά** είναι αναφορά που δεν αυξάνει τη μέτρηση αναφορών του αναφερόμενου. Σας επιτρέπει να δείχνετε σε κάτι χωρίς να το κρατάτε ζωντανό. Οι ασθενείς αναφορές είναι η πρότυπη απάντηση της Perl σε κύκλους αναφορών.

Πώς διαρρέει ένας κύκλος

Σκεφτείτε έναν κόμβο διπλά συνδεδεμένης λίστας που κρατάει αναφορές `next` και `prev`:

```
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;
```

Τα `$a` και `$b` έχουν το καθένα μέτρηση αναφορών 2:

- Το `$a` αναφέρεται από τη λεξιλογική `$a` **και** από το `$b->{prev}`.
- Το `$b` αναφέρεται από τη λεξιλογική `$b` **και** από το `$a->{next}`.

Όταν οι λεξιλογικές βγαίνουν εκτός εμβέλειας, η συνεισφορά τους πέφτει:

- Το `$a` πέφτει σε μέτρηση 1 (απομένει μόνο το `$b->{prev}`).
- Το `$b` πέφτει σε μέτρηση 1 (απομένει μόνο το `$a->{next}`).

Κανένα από τα δύο δεν φτάνει στο μηδέν. Και οι δύο κόμβοι είναι μη προσπελάσιμοι από οπουδήποτε στο πρόγραμμά σας, ωστόσο κρατούν ο ένας τον άλλο ζωντανό για όλη τη διάρκεια ζωής του διερμηνέα. Αυτή είναι διαρροή.

Διόρθωση της διαρροής με την weaken

Η `Scalar::Util::weaken` μετατρέπει μια υπάρχουσα αναφορά σε ασθενή. Η αναφορά συνεχίζει να λειτουργεί για αποαναφορά· απλώς δεν μετράει για τη διατήρηση του στόχου ζωντανού.

Το πρότυπο μοτίβο: όποια κατεύθυνση θεωρείτε «δείκτη προς τα πίσω» ή «δείκτη γονέα», κάντε **αυτή** ασθενή.

```
use Scalar::Util qw(weaken);

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
my $a = new_node('A');
my $b = new_node('B');
$a->{next} = $b;
$b->{prev} = $a;
weaken $b->{prev};              # the back-pointer
```

Τώρα η μέτρηση αναφορών του `$a` παραμένει στο 1 (συνεισφέρει μόνο η λεξιλογική `$a`· το `$b->{prev}` είναι ασθενές και δεν μετράει). Όταν το `$a` βγει εκτός εμβέλειας, το `$a` απελευθερώνεται - και επειδή η απελευθέρωση του `$a` ρίχνει το `$b->{next}`, απελευθερώνεται και το `$b`. Ολόκληρη η δομή καταρρέει.

Όταν μια ασθενής αναφορά γίνεται undef

Αν ο αναφερόμενος απελευθερωθεί ενώ υπάρχει ακόμα ασθενής αναφορά, η ασθενής αναφορά **γίνεται αυτόματα `undef`**. Δεν βλέπετε ποτέ κρεμασμένο δείκτη:

```
use Scalar::Util qw(weaken);

my $target = { name => 'alive' };
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
```

Το `defined $weak` είναι ο πρότυπος τρόπος για να ελέγξετε αν ο στόχος βρίσκεται ακόμα εκεί. Δείτε την `defined`.

Πού ανήκουν οι ασθενείς αναφορές

Ο κανόνας: χρησιμοποιήστε ασθενή αναφορά για τον «μη κατέχοντα» δείκτη σε οποιοδήποτε ζεύγος αναφορών που διαφορετικά θα σχημάτιζε κύκλο.

- **Διπλά συνδεδεμένη λίστα.** Το `next` κατέχει· το `prev` είναι ασθενές.
- **Δέντρο γονέα/παιδιού με δείκτες γονέα.** Το `children` κατέχει· το `parent` σε κάθε παιδί είναι ασθενές.
- **Μοτίβο παρατηρητή.** Το υποκείμενο κρατάει ασθενείς αναφορές στους παρατηρητές του· οι παρατηρητές κρατούν ισχυρές αναφορές στο υποκείμενο. Όταν ένας παρατηρητής αφήσει την αναφορά του, εξαφανίζεται· η εγγραφή του υποκειμένου για αυτόν γίνεται `undef` και μπορεί να σαρωθεί.
- **Caches που δεν πρέπει να κρατούν τις εγγραφές ζωντανές.** Ένα cache του οποίου οι τιμές είναι ασθενείς αναφορές δεν επεκτείνει τη διάρκεια ζωής των αποθηκευμένων αντικειμένων.

isweak - επιθεώρηση μιας αναφοράς

Η `Scalar::Util::isweak` σας λέει αν ένα δεδομένο βαθμωτό κρατάει ασθενή αναφορά:

```
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
```

Χρήσιμη κυρίως όταν αποσφαλματώνετε ένα πρόβλημα διάρκειας ζωής. Ο κώδικας παραγωγής σπάνια διακλαδίζει βάσει της `isweak`.

Συνηθισμένα λάθη

- **Ασθενοποίηση του λάθος άκρου.** Η ασθενοποίηση του «εμπρόσθιου» δείκτη (αυτού που κατέχει τον διάδοχο) απελευθερώνει τον διάδοχο αμέσως, πριν τον χρησιμοποιήσετε ποτέ. Το ασθενές πάει στον **δείκτη προς τα πίσω**.
- **Ασθενοποίηση λεξιλογικής.** Το `weaken $ref` όπου η `$ref` είναι μια φρέσκα λεξιλογική που αναφέρεται στον `$target` κάνει τη λεξιλογική ασθενή. Αν τίποτα άλλο δεν κρατάει ισχυρή αναφορά στον `$target`, αυτός εξαφανίζεται στην επόμενη πρόταση και η `$ref` γίνεται `undef`. Οι ασθενείς αναφορές ανήκουν σε θυρίδες δομών δεδομένων, όχι τυπικά σε τοπικές λεξιλογικές.
- **Η υπόθεση ότι οι ασθενείς αναφορές είναι αυτόματες.** Η Perl **δεν** εντοπίζει κύκλους και δεν τους ασθενοποιεί για εσάς. Πρέπει να συμμετάσχετε ρητά.

Πέρα από το Scalar::Util

Το `Scalar::Util` έρχεται μαζί με την Perl· είναι πάντα διαθέσιμο. Για πιο εξεζητημένες ανάγκες - `WeakRef`, `Hash::Util::FieldHash` - καταφύγετε στο CPAN. Για συνηθισμένο κώδικα, η `weaken` είναι όλο όσο χρειάζεστε.

Πού να πάτε στη συνέχεια

Ολοκληρώσατε τον οδηγό εκμάθησης αναφορών. Καλά επόμενα αναγνώσματα:

- Η σελίδα αναφοράς της `ref` - η πλήρης λίστα των συμβολοσειρών που επιστρέφει, συμπεριλαμβανομένων των περιπτώσεων blessed αντικειμένων.
- Η `bless` - πώς μια απλή αναφορά κατακερματισμού γίνεται αντικείμενο μιας συγκεκριμένης κλάσης.
- Ο οδηγός εκμάθησης αντικειμενοστρέφειας, μόλις είστε έτοιμοι για κλάσεις χτισμένες πάνω στις αναφορές που τώρα κατανοείτε.
- **Βασικά** - ο τελεστής `\`, αποαναφορά βαθμωτού/πίνακα/κατακερματισμού με `$$x`, `@$x`, `%$x`, αποσαφήνιση με αγκύλες, και το βέλος `->` για πρόσβαση σε στοιχεία.
- **Πίνακες πινάκων** - δισδιάστατα δεδομένα, μοτίβα επανάληψης, ο κανόνας του βέλους-μεταξύ-δεικτών που κάνει το `$m[i][j]` να διαβάζεται ως πίνακας.

- **Κατακερματισμοί και μείξεις** - κατακερματισμοί πινάκων, πίνακες κατακερματισμών, και το συνηθισμένο σχήμα πίνακα-εγγραφών.
- **Ανώνυμες αναφορές** - `[...]` και `{...}` ως ενσωματωμένοι κατασκευαστές, πότε να τις προτιμάτε έναντι κατονομασμένων μεταβλητών.
- **Αναφορές υπορουτινών** - `\&name`, ανώνυμα `sub { ... }`, κλήση με `->(...)`, και πώς οι closures αποτυπώνουν λεξιλογικές μεταβλητές.
- **Ασθενείς αναφορές** - `Scalar::Util::weaken` και γιατί οι κυκλικές δομές τη χρειάζονται.

4.13.3 Σχετικές σελίδες αναφοράς

- `ref` - ρωτήστε μια τιμή τι είδους αναφορά είναι.
- `defined` - ο πρότυπος τρόπος για να ελέγξετε «υπάρχει ήδη αυτή η αναφορά;» μετά από αυτο-δημιουργία.
- `bless` - το βήμα που μετατρέπει μια απλή αναφορά σε αντικείμενο.
- *Αναφορές (αναφορά τύπων δεδομένων)* - η περιεκτική σελίδα αναφοράς μίας σελίδας που συμπληρώνει αυτόν τον οδηγό.
  - η περιεκτική σελίδα αναφοράς μίας σελίδας που συμπληρώνει αυτό το 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"
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
# 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

Το filehandle είναι το όριο ανάμεσα στον κόσμο κειμένου του προγράμματός σας και στον έξω κόσμο των bytes. Το να κάνετε το Unicode σωστά σημαίνει να στήσετε αυτό το όριο σωστά κατά την `open` (ή με `binmode`), ώστε οι συνηθισμένες `print` και `readline` να μιλούν ήδη τη σωστή κωδικοποίηση.

Στρώματα PerlIO

Κάθε filehandle φέρει μια στοίβα από *layers* - μικρά κομμάτια μηχανισμού που μετασχηματίζουν τα bytes καθώς διασχίζουν το όριο. Τα σχετικά για το Unicode είναι:

- `:raw` - κανένας μετασχηματισμός. Bytes μέσα, bytes έξω.
- `:utf8` - ελέγχει ότι τα bytes που διαβάζονται σχηματίζουν έγκυρο UTF-8 και σημειώνει τη συμβολοσειρά ως κείμενο. Εμπιστεύεται τον χρήστη να μην παράγει μη έγκυρο UTF-8 στις εγγραφές.
- `:encoding(ENCODING)` - αποκωδικοποίηση κατά την ανάγνωση, κωδικοποίηση κατά την εγγραφή, για οποιαδήποτε ονομασμένη κωδικοποίηση. Απορρίπτει την κακοσχηματισμένη είσοδο εξ ορισμού.
- `:crlf` - μεταφράζει `"\r\n"` σε `"\n"` κατά την ανάγνωση, `"\n"` σε `"\r\n"` κατά την εγγραφή. Στο Linux αυτό είναι ανενεργό· στα Windows είναι η προεπιλογή για handles κειμένου.

Το `:encoding(UTF-8)` είναι αυτό που σχεδόν πάντα θέλετε. Προτιμήστε το από το σκέτο layer `:utf8` - η μορφή `:encoding(...)` επικυρώνει, το `:utf8` όχι.

Άνοιγμα ενός αρχείου με κωδικοποίηση

Η μορφή τριών ορισμάτων της `open` παίρνει ένα mode που μπορεί να περιλαμβάνει 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
}
```

Κάθε `readline` από το `$in` επιστρέφει μια συμβολοσειρά κειμένου. Κάθε `print` προς το `$out` κωδικοποιείται σε UTF-8 στο καλώδιο. Το σώμα του βρόχου βλέπει μόνο χαρακτήρες - καμία decode, καμία encode, καμία μέτρηση bytes κατά λάθος.

Εκ των υστέρων προσθήκη με binmode

Αν το filehandle είναι ήδη ανοιχτό - για παράδειγμα οι standard streams, ή ένα handle που σας έδωσε ένα άλλο άρθρωμα - προσαρτήστε το layer με `binmode`:

```
binmode STDIN,  ":encoding(UTF-8)";
binmode STDOUT, ":encoding(UTF-8)";
binmode STDERR, ":encoding(UTF-8)";
```

Κάντε το αυτό μία φορά, στην κορυφή του προγράμματος, πριν από οποιοδήποτε I/O σε αυτά τα handles. Μετά από αυτό τα standard streams συμπεριφέρονται όπως ένα αρχείο ανοιγμένο με ένα layer κωδικοποίησης.

Η pragma use open

Το να γράφετε το ίδιο layer `:encoding(UTF-8)` σε κάθε `open` είναι θορυβώδες. Η pragma open ορίζει τα προεπιλεγμένα layers για όλα τα επόμενα ανοίγματα στην περικλείουσα εμβέλεια:

```
use open ":std", ":encoding(UTF-8)";
```

Η ετικέτα `:std` επίσης ξανα-στρώνει τα `STDIN`, `STDOUT` και `STDERR` επιτόπου, οπότε παίρνετε το αποτέλεσμα τριών κλήσεων `binmode` και μια προεπιλογή για κάθε μεταγενέστερη `open` σε μία γραμμή.

Αυτό είναι το μονόγραμμο στην κορυφή ενός σύγχρονου σεναρίου με επίγνωση Unicode:

```
use utf8;
use open ":std", ":encoding(UTF-8)";
```

- `use utf8` - οι κυριολεξίες σε αυτό το αρχείο πηγαίου κώδικα είναι UTF-8.
- `use open ":std", ":encoding(UTF-8)"` - τα stdin/stdout/stderr και κάθε μεταγενέστερο open έχουν προεπιλογή UTF-8 στο καλώδιο.

Μπορείτε επίσης να ωθήσετε το I/O σε UTF-8 από έξω από το πρόγραμμα με τη μεταβλητή περιβάλλοντος `PERL_UNICODE`. Τα γράμματα `I`, `O`, `E` επιλέγουν stdin, stdout και stderr μεμονωμένα, το `S` επιλέγει και τα τρία ταυτόχρονα με τον τρόπο που το κάνει η pragma `use open ":std"`, και το `A` αποκωδικοποιεί το `@ARGV`, οπότε το `PERL_UNICODE=SA` είναι περίπου το ισοδύναμο γραμμής εντολών του δίγραμμου προλόγου παραπάνω. Για να

αποκωδικοποιήσετε τα ορίσματα μέσα στο πρόγραμμα αντ” αυτού, γράψτε `@ARGV = map { decode('UTF-8', $_, 1) } @ARGV`.

Ο αντίστοιχος διακόπτης γραμμής εντολών `-C` (`-CSA`) δεν αναγνωρίζεται ακόμη από την pperl, που είναι μια απόκλιση από τη στάνταρ perl· χρησιμοποιήστε τη μεταβλητή `PERL_UNICODE` ή την pragma `use open` μέχρι να υλοποιηθεί.

Binary I/O παράλληλα με I/O κειμένου

Κάποια handles φέρουν bytes, όχι κείμενο - αρχεία εικόνων, συμπιεσμένα streams, πρωτόκολλα δικτύου που πλαισιώνουν τη δική τους κωδικοποίηση. Ανοίξτε εκείνα `:raw` και μην προσαρτάτε ποτέ κωδικοποίηση:

```
open my $img, "<:raw", "photo.jpg" or die "open: $!";
```

Μέσα σε ένα πρόγραμμα, κάποια handles μπορεί να είναι κειμένου (`:encoding(UTF-8)`) και άλλα binary (`:raw`). Κρατήστε τις ετικέτες ξεκάθαρες: ένα binary handle σας δίνει συμβολοσειρές bytes, ένα handle κειμένου σας δίνει συμβολοσειρές κειμένου, και το `$!` είναι το ίδιο όπως και να “χει.

Διαδραστική είσοδος

Ένα terminal δεν σας λέει σε ποια κωδικοποίηση πληκτρολογεί. Στο σύγχρονο Linux είναι σχεδόν πάντα UTF-8· στα Windows το προεπιλεγμένο code page της κονσόλας είναι ακόμη συχνά CP1252 ή CP437. Η συνήθης υπόθεση για διαπλατφορμικά σενάρια είναι UTF-8, με `binmode` εφαρμοσμένο κατά την εκκίνηση:

```
binmode STDIN, ":encoding(UTF-8)";
```

Αν το πρόγραμμά σας πρέπει να υποστηρίζει παλαιά terminals, αποκωδικοποιήστε με βάση τη μεταβλητή περιβάλλοντος `LANG` ή `LC_CTYPE`, ή αποδεχτείτε μια σημαία `--encoding`. Δεν υπάρχει φορητή αυτόματη ανίχνευση.

Έλεγχος των layers

Για να δείτε ποια layers υπάρχουν σε ένα handle, χρησιμοποιήστε την `PerlIO::get_layers`:

```
use PerlIO;
my @layers = PerlIO::get_layers(\*STDOUT);
print "@layers\n";                          # e.g. unix perlio␣
↪encoding(utf-8-strict)
```

Χρήσιμο όταν ένα πρόγραμμα παράγει σιωπηλά διπλά-κωδικοποιημένη έξοδο και θέλετε να βρείτε ποιο handle έχει το απρόσμενο layer πάνω του.

Regex και ιδιότητες

Μόλις οι συμβολοσειρές σας αποκωδικοποιηθούν, η regex λειτουργεί σε χαρακτήρες - και η μηχανή regex της Perl γνωρίζει για το Unicode. Το `\w` ταιριάζει κάθε χαρακτήρα λέξης Unicode, η αντιστοίχιση χωρίς διάκριση πεζών-κεφαλαίων αναδιπλώνεται μεταξύ γραφών, και το `\p{…}` σας δίνει πρόσβαση στην πλήρη βάση δεδομένων χαρακτήρων

Unicode. Αυτό το κεφάλαιο καλύπτει τις συνήθειες που χρειάζεστε για να το κάνετε να δουλεύει αξιόπιστα.

Χαρακτήρες, όχι bytes

Σε μια συμβολοσειρά κειμένου, κάθε μετα-χαρακτήρας regex μιλά σε χαρακτήρες:

```
use utf8;
my $s = "café";                    # 4 characters
$s =~ /^.{4}$/;                    # matches
$s =~ /^.{5}$/;                    # does not
```

Σε μια binary συμβολοσειρά, οι ίδιοι μετα-χαρακτήρες μετρούν bytes. Αυτό συνήθως δεν είναι αυτό που θέλετε - αν βρεθείτε να γράφετε μια regex έναντι bytes που περιέχουν κείμενο, αποκωδικοποιήστε πρώτα.

Τα \w, \d, \s και οι φίλοι τους

Στην προεπιλεγμένη (Unicode) λειτουργία, αυτές οι συντομογραφίες ταιριάζουν τα ισοδύναμά τους στο Unicode:

- `\w` - γράμματα, ψηφία και κάτω παύλα από κάθε γραφή.
- `\d` - δεκαδικά ψηφία από κάθε γραφή (όχι μόνο `0`–`9`).
- `\s` - λευκός χαρακτήρας κάθε είδους, συμπεριλαμβανομένου του 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
```

Όταν θέλετε την παλιά σημασία μόνο-ASCII, προσθέστε τον τροποποιητή `/a` - δείτε τους *Τροποποιητές* παρακάτω.

Οι κλάσεις ιδιοτήτων \p{…}

Το `\p{PROPERTY}` ταιριάζει οποιονδήποτε χαρακτήρα με μια ονομασμένη ιδιότητα Unicode. Τα ονόματα ιδιοτήτων αντλούνται από τη βάση δεδομένων χαρακτήρων Unicode· τα συνήθως χρήσιμα είναι:

- `\p{Letter}`, συντομογραφία `\p{L}` - οποιοδήποτε γράμμα.
- `\p{Lu}`, `\p{Ll}`, `\p{Lt}` - κεφαλαία, πεζά, title-case γράμματα.
- `\p{Number}`, `\p{N}`, `\p{Nd}` - όλοι οι αριθμοί, δεκαδικά ψηφία.
- `\p{Punct}`, `\p{P}` - σημεία στίξης.
- `\p{Space}`, `\p{Zs}` - λευκός χαρακτήρας, διαχωριστές διαστήματος.
- `\p{ASCII}` - τα 128 σημεία κώδικα ASCII.

- `\p{Script=Greek}` - κάθε χαρακτήρας που ανήκει στην ελληνική γραφή. Οποιοδήποτε όνομα γραφής από τα δεδομένα Unicode λειτουργεί εδώ.
- `\p{Block=Cyrillic}` - κάθε χαρακτήρας στο κυριλλικό block. (Η γραφή και το block διαφέρουν: η *γραφή* είναι το σύστημα γραφής στο οποίο ανήκει ένας χαρακτήρας· το *block* είναι το εύρος σημείων κώδικα στο οποίο βρίσκεται.)

Το `\P{…}` είναι η άρνηση - οποιοσδήποτε χαρακτήρας χωρίς αυτή την ιδιότητα.

```
use utf8;
my $s = "Γειά σου, κόσμε";
my @greek_letters = $s =~ /(\p{Script=Greek})/g;
scalar @greek_letters;                    # 12
```

Ο πλήρης κατάλογος ιδιοτήτων συνοδεύει την Perl· το `perldoc perluniprops` απαριθμεί κάθε όνομα που αποδέχεται η μηχανή.

Τροποποιητές: `/a`, `/u`, `/l`, `/aa`

Τέσσερις τροποποιητές αλλάζουν το πώς ερμηνεύονται οι κλάσεις χαρακτήρων:

- `/u` - **Unicode mode.** Το `\w` ταιριάζει χαρακτήρες λέξης Unicode, το `\d` ταιριάζει δεκαδικά ψηφία Unicode, η αναδίπλωση πεζών-κεφαλαίων χρησιμοποιεί κανόνες Unicode. Αυτή είναι η προεπιλογή σε συμβολοσειρές κειμένου.
- `/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.** Το `/a` συν η αναδίπλωση πεζών-κεφαλαίων περιορίζεται σε ASCII. Το `/foo/aai` ταιριάζει FOO αλλά όχι FÖO. Χρησιμοποιήστε το όταν θέλετε αντιστοίχιση αναγνωριστικού έναντι μιας γνωστά-ASCII προδιαγραφής (ονόματα κεφαλίδων HTTP, λέξεις-κλειδιά εντολών).
- `/l` - **locale mode.** Παραπέμπει στο τρέχον locale POSIX. Σχεδόν ποτέ αυτό που θέλετε σε ένα πρόγραμμα με επίγνωση Unicode· αναφέρεται μόνο για να μπορείτε να το αναγνωρίσετε όταν το χρησιμοποιεί μια άλλη βάση κώδικα.

```
use utf8;
"café" =~ /\w+/;                          # matches whole word␣
↪(Unicode)
"café" =~ /\w+/a;                         # matches "caf" (ASCII␣
↪only)
"FOO"  =~ /foo/ai;                        # matches (case fold␣
↪anywhere)
"FÖO"  =~ /foo/aa;                        # does not - strict ASCII
```

Εμπειρικός κανόνας: η προεπιλεγμένη λειτουργία είναι σωστή για κείμενο· περάστε στο `/aa` όταν αναλύετε ένα πρωτόκολλο καθορισμένο σε ASCII και θέλετε να απορρίψετε λαθραίους μη-ASCII χαρακτήρες.

Αντιστοίχιση χωρίς διάκριση πεζών-κεφαλαίων

Το `/i` αναδιπλώνει και τις δύο πλευρές της αντιστοιχίας χρησιμοποιώντας τους πίνακες αναδίπλωσης πεζών-κεφαλαίων Unicode. Αυτό χειρίζεται τα προφανή ευρωπαϊκά ζεύγη (é/É, ß/SS), και τα λιγότερο προφανή (İ/i, fi/fi).

```
use utf8;
"Ångström" =~ /ångström/i;              # matches
"STRASSE"  =~ /straße/i;                # matches - ß folds to SS
```

Για αναδίπλωση πεζών-κεφαλαίων σε επίπεδο κυριολεκτικών byte (την παλιά σημασία ASCII του `/i`), χρησιμοποιήστε `/iaa`.

Ονομασμένα σημεία κώδικα

Το `\N{...}` ονομάζει έναν χαρακτήρα Unicode, είτε με το επίσημο όνομά του είτε με τον αριθμό του σημείου κώδικα:

```
use charnames ();
"\N{LATIN SMALL LETTER E WITH ACUTE}" eq "\x{e9}";   # true
"\N{U+00E9}" eq "\x{e9}";                             # the same
 ↪character
```

Το `\N{U+HEX}` λειτουργεί χωρίς καμία pragma· τα μακρά ονόματα περνούν μέσα από το άρθρωμα `charnames`, που φορτώνεται αυτόματα μέσα σε ένα escape `\N{...}` (ή ρητά με `use charnames qw(:full)`). Σε ένα μοτίβο, το `\N{...}` είναι χρήσιμο όταν θέλετε ο πηγαίος κώδικας να τεκμηριώνει ποιον χαρακτήρα εννοείτε χωρίς να βασίζεστε στο ότι ο αναγνώστης θα αναγνωρίσει ένα δεκαεξαδικό escape:

```
"café" =~ /caf\N{LATIN SMALL LETTER E WITH ACUTE}/;  # matches
```

Διαχωρισμός και ένωση

Οι `split` και `substr` μετρούν και οι δύο στις μονάδες της συμβολοσειράς στην οποία λειτουργούν. Σε μια συμβολοσειρά κειμένου, η μέτρηση είναι χαρακτήρες:

```
use utf8;
my $s = "a,é,b,Ω";
my @parts = split /,/, $s;              # ("a", "é", "b", "Ω")
substr($s, 2, 1);                       # "é"
```

Σε μια binary συμβολοσειρά, οι ίδιες λειτουργίες μετρούν bytes. Αυτό είναι το πιο συνηθισμένο λεπτό σφάλμα - μια παλιά βάση κώδικα που αποκωδικοποίησε την είσοδο σε μία συνάρτηση αλλά δεικτοδότησε το αποτέλεσμα σε μια άλλη, φτιαγμένη πριν εισαχθεί το βήμα αποκωδικοποίησης.

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 *cafÃ©*

Symptom: text looks almost right, but every non-ASCII character has been replaced by a pair of funny-looking bytes. `café` becomes `cafÃ©`, `Straße` becomes `StraÃŸe`.

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 `cafÃ©`, as `caf?`, 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.

Συνταγές

Τα εννοιολογικά κεφάλαια καλύπτουν την πειθαρχία· αυτή η σελίδα είναι το παράρτημα γρήγορης αναφοράς. Κάθε συνταγή είναι μια σύντομη, αυτοτελής απάντηση σε μια συνηθισμένη εργασία Unicode - το είδος του πράγματος για το οποίο το κλασικό βιβλίο συνταγών Unicode της Perl καταφεύγει σε ένα άρθρωμα CPAN, αλλά που οι ενσωματωμένες συναρτήσεις και η μηχανή regex της pperl χειρίζονται απευθείας.

Κάθε παράδειγμα εδώ υποθέτει ότι οι συμβολοσειρές σας είναι ήδη κείμενο - αποκωδικοποιημένες κατά την είσοδο, όπως περιγράφει το κεφάλαιο *Συμβολοσειρές και κωδικοποιήσεις*. Όλες οι παρακάτω λειτουργίες μιλούν χαρακτήρες, οπότε μια binary συμβολοσειρά που τους δοθεί θα μετρήσει bytes και θα σας εκπλήξει.

Ταξινόμηση και σύγκριση χωρίς διάκριση πεζών-κεφαλαίων με `fc`

Η ενσωματωμένη συνάρτηση `fc` επιστρέφει το *foldcase* μιας συμβολοσειράς - την ίδια αναδίπλωση πεζών-κεφαλαίων Unicode που χρησιμοποιεί ο τροποποιητής μοτίβου `/i`. Δύο συμβολοσειρές είναι ίσες υπό αναδίπλωση πεζών-κεφαλαίων ακριβώς όταν η `fc` επιστρέφει το ίδιο αποτέλεσμα και για τις δύο, κάτι που κάνει την `fc` το σωστό εργαλείο για ταξινόμηση και σύγκριση χωρίς διάκριση πεζών-κεφαλαίων.

```
use v5.36;     # or: use feature 'fc';
use utf8;

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
# sort case-insensitively
my @sorted = sort { fc($a) cmp fc($b) } @list;

# both are true:
fc("tschüß")  eq fc("TSCHÜSS");
fc("Σίσυφος") eq fc("ΣΊΣΥΦΟΣ");
```

Η αναδίπλωση χειρίζεται τις περιπτώσεις που η απλή `lc` δεν μπορεί: το γερμανικό sharp s αναδιπλώνεται σε `ss`, και το ελληνικό τελικό σίγμα ς αναδιπλώνεται μαζί με το μεσαίο σ. Καταφύγετε στην `fc` αντί στην `lc` όποτε η σύγκριση πρέπει να αγνοεί την περίπτωση πεζών-κεφαλαίων με την πλήρη έννοια του Unicode.

Αυτό είναι μόνο αναδίπλωση πεζών-κεφαλαίων. Δεν αγνοεί τους τόνους, και δεν επιβάλλει την αλφαβητική σειρά ενός locale - για αυτό θα χρειαζόσασταν μια βιβλιοθήκη συνταξιοθεσίας όπως το `Unicode::Collate`, που δεν είναι ακόμη χρησιμοποιήσιμο στην pperl (δείτε το τέλος αυτής της σελίδας).

Κανονικοποίηση αλλαγών γραμμής με \R

Το `\R` ταιριάζει μία μεμονωμένη αλλαγή γραμμής Unicode: το γράφημα CRLF δύο χαρακτήρων, ή οποιονδήποτε από τους κατακόρυφους λευκούς χαρακτήρες (LF, CR, form feed, vertical tab, next line, line separator, paragraph separator). Είναι ο φορητός τρόπος για να χειριστείτε αρχεία κειμένου που φτάνουν από διαφορετικά λειτουργικά συστήματα.

```
my $text = "line one\r\nline two\rline three\n";
$text =~ s/\R/\n/g;          # normalise every linebreak to \n
```

Μετά την αντικατάσταση, το `$text` χρησιμοποιεί ένα μεμονωμένο `\n` παντού ανεξάρτητα από το πώς η είσοδος κωδικοποίησε τους τερματισμούς γραμμής της. Χρησιμοποιήστε το `\R` στη θέση μιας ρητής εναλλαγής `\r?\n`: καλύπτει τις περιπτώσεις που το χειρόγραφο μοτίβο ξεχνά.

Αντιστοίχιση μιας συστάδας γραφημάτων με \X

Ένας ορατός στον χρήστη χαρακτήρας - ένα βασικό γράμμα μαζί με τυχόν συνδυαστικά σημάδια - είναι μια *συστάδα γραφημάτων*. Ο μετα-χαρακτήρας `.` ταιριάζει ένα σημείο κώδικα, που μπορεί να είναι μόνο μέρος μιας τέτοιας συστάδας· το `\X` ταιριάζει μια ολόκληρη συστάδα γραφημάτων.

```
use utf8;

# "é" written as e + COMBINING ACUTE ACCENT is two code points,
# but one grapheme.
my $str = "cafe\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;
```

Το \X είναι η σωστή μονάδα όποτε ο «χαρακτήρας» σημαίνει «αυτό που βλέπει ο αναγνώστης» αντί «σημείο κώδικα». Οι παρακάτω συνταγές χτίζουν πάνω του.

Τα πρώτα N γραφήματα, αντιστροφή, και μήκος ανά γράφημα

Και οι τρεις αυτές εργασίες ανάγονται στην εφαρμογή του \X αντί για δεικτοδότηση ανά σημείο κώδικα. Λειτουργούν σωστά ανεξάρτητα από τη μορφή κανονικοποίησης στην οποία βρίσκεται η συμβολοσειρά.

Πάρτε τα πρώτα πέντε γραφήματα:

```
my ($first_five) = $str =~ /^ ( \X{5} ) /x;
```

Αντιστρέψτε μια συμβολοσειρά ανά γράφημα. Η αντιστροφή ανά σημείο κώδικα θα αποσπούσε ένα συνδυαστικό σημάδι από το βασικό του γράμμα και θα κατέστρεφε το κείμενο· η συλλογή των γραφημάτων πρώτα διατηρεί κάθε συστάδα ακέραιη:

```
$str = join "", reverse $str =~ /\X/g;
```

Μετρήστε το μήκος μιας συμβολοσειράς σε γραφήματα. Η λέξη brûlée είναι έξι γραφήματα αλλά μπορεί να φτάνει τα οκτώ σημεία κώδικα, οπότε η length και μια μέτρηση γραφημάτων μπορεί να διαφέρουν:

```
my $count = 0;
$count++ while $str =~ /\X/g;
```

Λεπτομερής ρύθμιση των προειδοποιήσεων UTF-8

Η Perl χωρίζει τις προειδοποιήσεις UTF-8 σε τρεις υποκατηγορίες, ώστε να μπορείτε να σιγάσετε ένα είδος προειδοποίησης κακοσχηματισμένου Unicode χωρίς να σιγάσετε τις άλλες. Η pperl αναγνωρίζει και τα τρία ονόματα κατηγοριών:

```
no warnings "nonchar";      # the forbidden non-characters
no warnings "surrogate";    # UTF-16 surrogate code points
no warnings "non_unicode";  # code points above 0x10_FFFF
```

Απενεργοποιήστε μόνο την υποκατηγορία που έχετε σκόπιμο λόγο να επιτρέψετε - για παράδειγμα no warnings "non_unicode" σε κώδικα που χειρίζεται σημεία κώδικα πέρα από το εύρος Unicode επίτηδες - και αφήστε τις υπόλοιπες σε ισχύ ώστε γνήσια σφάλματα κωδικοποίησης να εξακολουθούν να εμφανίζονται.

Αποκωδικοποίηση του @ARGV

Τα ορίσματα γραμμής εντολών φτάνουν ως bytes. Αν ένας χρήστης περάσει ένα μη-ASCII όρισμα, πρέπει να το αποκωδικοποιήσετε πριν το αντιμετωπίσετε ως κείμενο, ακριβώς όπως θα αποκωδικοποιούσατε οποιαδήποτε άλλη είσοδο. Η φορητή μορφή εφαρμόζει τη decode πάνω στο @ARGV:

```
use Encode qw(decode);
@ARGV = map { decode('UTF-8', $_, 1) } @ARGV;
```

Μετά από αυτή τη γραμμή, κάθε στοιχείο του @ARGV είναι μια συμβολοσειρά κειμένου και συμπεριφέρεται υπό την length, τη regex, και τις λειτουργίες πεζών-κεφαλαίων σε χαρακτήρες αντί για bytes. Το τρίτο όρισμα 1 (ο έλεγχος Encode::FB_CROAK) κάνει ένα κακοσχηματισμένο όρισμα μοιραίο σφάλμα αντί για σιωπηλή αντικατάσταση, κάτι που είναι συνήθως αυτό που θέλετε για είσοδο που δεν μπορείτε να ζητήσετε ξανά.

Προσαρμοσμένη ιδιότητα ανά εύρος σημείων κώδικα

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\tEND 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
```

Αυτός είναι ο φορητός τρόπος για να ορίσετε μια ad-hoc κλάση χαρακτήρων μία φορά και να την επαναχρησιμοποιήσετε με το όνομά της σε διάφορα μοτίβα, χωρίς να αναγράφετε το εύρος σε κάθε regex.

Αρθρώματα σε εκκρεμότητα στην pperl

Μια χούφτα συνταγές στο κλασικό βιβλίο συνταγών Unicode της Perl καταφεύγουν στα βοηθητικά αρθρώματα Unicode που η Perl παρέχει ως μεταγλωττισμένες επεκτάσεις: Unicode::Normalize (κανονικοποίηση NFC, NFD, NFKC, NFKD), Unicode::Collate και Unicode::Collate::Locale (αλφαβητική και εξαρτώμενη από locale ταξινόμηση, σύγκριση χωρίς διάκριση τόνων), Unicode::GCString (substr με επίγνωση γραφημάτων και πλάτος στηλών εκτύπωσης), Unicode::UCD (προγραμματική αναζήτηση κατηγορίας χαρακτήρα και αριθμητικής τιμής), και Unicode::LineBreak (κοπή γραμμών σύμφωνα με τους κανόνες Unicode).

Αυτά τα αρθρώματα δεν είναι ακόμη διαθέσιμα στην pperl: η φόρτωση ενός σήμερα αποτυγχάνει με dynamic loading not available in this perl. Μέχρι να γίνουν, η οικογένεια regex \X παραπάνω καλύπτει το μήκος, την αντιστροφή και την εξαγωγή σταθερού πλήθους με επίγνωση γραφημάτων (αν και όχι το πλάτος στηλών εκτύπωσης), και δεν υπάρχει εδώ υποκατάστατο για την κανονικοποίηση ή τη συνταξιοθεσία locale. Οι ονομασμένοι χαρακτήρες και οι ιδιότητες ορισμένες από τον χρήστη, αντιθέτως, λειτουργούν σήμερα, όπως δείχνουν οι παραπάνω συνταγές.

- **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 Λογική Boole για προγραμματιστές της Perl - οδηγός εκμάθησης

Σχεδόν κάθε γραμμή λειτουργικής Perl διαμορφώνεται από τη λογική Boole. Τα `open ... or die`, `$h{$k} //= []`, `unless ($a == $b && $c eq $d)`, `/^(yes|no)$/`, `$flags & 0x4` - είναι όλα η ίδια χούφτα πράξεων εφαρμοσμένη σε διαφορετικά πεδία. Αυτός ο οδηγός εκμάθησης διατρέχει αυτές τις πράξεις από τη βάση προς τα πάνω: τι σημαίνει αλήθεια στην Perl, τι επιστρέφουν στην πραγματικότητα οι τελεστές, και τις δεκαέξι δυαδικές συναρτήσεις αληθείας και πώς γράφεται η καθεμία στην Perl, τους δύο νόμους που μετατρέπουν περίπλοκες συνθήκες σε απλές, και τα τρία σημεία όπου εμφανίζεται περισσότερο η λογική: αριθμητική κατά bit, κανονικές εκφράσεις και ροή ελέγχου.

4.15.1 Σε ποιον απευθύνεται

Γράφετε ήδη λειτουργική Perl. Χρησιμοποιείτε τα `&&`, `||`, `if`, `unless` χωρίς να το σκέφτεστε - όταν όμως μια συνθήκη μεγαλώσει πέρα από τρεις ή τέσσερις προτάσεις, παύετε να την εμπιστεύεστε και καταφεύγετε σε παρενθέσεις για τις οποίες υποπτεύεστε ότι δεν χρειάζονται. Αυτός ο οδηγός εκμάθησης κλείνει αυτό το κενό. Αφού τον διαβάσετε, θα γνωρίζετε με ακρίβεια ποια λογική πράξη εκτελεί μια δεδομένη έκφραση, γιατί δύο φαινομενικά διαφορετικές μορφές είναι η ίδια έκφραση, και πώς να ανάγετε ένα μπερδεμένο `unless` σε ένα καθαρό `if` με σιγουριά αντί με δοκιμή και σφάλμα.

4.15.2 Πεδίο

Κλασική δίτιμη λογική Boole - αληθές και ψευδές, τίποτε άλλο. Το `undef` της Perl εισάγει μια τρίτη τιμή (το *άγνωστο*) και μια ολόκληρη οικογένεια τρίτιμων ιδιωμάτων (`defined`, `//`, `//=`). Αυτά αναφέρονται όπου τέμνονται με τη δίτιμη λογική, αλλά μια κατάλληλη πραγμάτευση της τρίτιμης λογικής θα διπλασίαζε το μέγεθος αυτού του οδηγού εκμάθησης χωρίς να τον ολοκληρώνει· αυτός είναι ξεχωριστός οδηγός.

4.15.3 Πώς είναι οργανωμένος αυτός ο οδηγός εκμάθησης

Κάθε κεφάλαιο είναι μία έννοια. Διαβάστε τα με τη σειρά στο πρώτο πέρασμα· μεταβείτε στο σχετικό κεφάλαιο αργότερα όταν αναζητάτε ένα γεγονός.

Αλήθεια τιμών

Πριν από τη λογική, οι τιμές πάνω στις οποίες ενεργεί η λογική. Η Perl δεν έχει αποκλειστικό λογικό τύπο. Κάθε βαθμωτό είναι είτε *αληθές* είτε *ψευδές* όταν χρησιμοποιείται σε λογικό περιβάλλον, και ο κανόνας είναι σύντομος:

> Ένα βαθμωτό είναι **ψευδές** αν είναι μία από τις τέσσερις τιμές undef, ο ακέραιος `0`, η συμβολοσειρά `""`, ή η συμβολοσειρά `"0"`. Οτιδήποτε άλλο είναι **αληθές**.

Τέσσερις ψευδείς τιμές, κάθε άλλο βαθμωτό - συμπεριλαμβανομένων των συμβολοσειρών `"00"`, `"0.0"`, `"0E0"`, και `"false"` - είναι αληθές.

```
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"
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
```

Το ιδίωμα `"0 but true"` δεν είναι αστείο. Ορισμένα API της Perl επιστρέφουν μια τιμή που πρέπει να συγκρίνεται ίση με μηδέν αριθμητικά (ώστε ένας αριθμητικός έλεγχος να κάνει το σωστό) αλλά να είναι αληθής σε λογικό περιβάλλον. Η κλασική περίπτωση είναι το `ioctl()` - η επιτυχία που θα επέστρεφε κυριολεκτικά μηδέν θα ελεγχόταν ως ψευδής. Η συμβολοσειρά `"0 but true"` είναι αριθμητικά μηδέν (με μια προειδοποίηση που καταστέλλεται κατόπιν ιστορικής συμφωνίας) και λογικά αληθής επειδή δεν ανήκει στο τετραστοιχειακό σύνολο των ψευδών.

```
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"
```

Δύο γεύσεις σύγκρισης

Η Perl διαθέτει παράλληλες οικογένειες τελεστών σύγκρισης επειδή έχει παράλληλες έννοιες ισότητας. Οι αριθμοί συγκρίνονται αριθμητικά· οι συμβολοσειρές συγκρίνονται λεξικογραφικά. Η λανθασμένη οικογένεια στον λανθασμένο τύπο είναι ένα από τα πιο κοινά σφάλματα στην αρχάρια Perl.

Έλεγχος	Αριθμητικός	Συμβολοσειρά
ίσο	==	eq
άνισο	!=	ne
μικρότερο από	<	lt
μεγαλύτερο από	>	gt
μικρότερο ή ίσο	<=	le
μεγαλύτερο ή ίσο	>=	ge
τριμερής (sort)	<=>	cmp

Το == σε συμβολοσειρές πάντα πετυχαίνει για οποιεσδήποτε δύο μη αριθμητικές συμβολοσειρές - και οι δύο μετατρέπονται σε 0, και το 0 == 0 είναι αληθές:

```
"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
```

Τα <=> και cmp επιστρέφουν -1, 0, ή +1 - η τυπική μορφή συγκριτή ταξινόμησης. Δεν είναι λογικοί τελεστές με τη στενή έννοια, αλλά εμφανίζονται μέσα σε λογικά περιβάλλοντα αρκετά συχνά ώστε να αξίζει να τους γνωρίζετε.

Γιατί αυτό έχει σημασία για τη λογική

Κάθε λογικός τελεστής στην Perl - &&, ||, //, !, xor, and, or, not - χρησιμοποιεί το παραπάνω τετραστοιχειακό σύνολο των ψευδών ως κανόνα εισόδου του. Όποτε ένα παρακάτω κεφάλαιο λέει «αν το $x είναι ψευδές», αυτό ακριβώς εννοεί: το $x είναι undef, 0, "", ή "0". Όποτε λέει «αν το $x είναι αληθές», το $x είναι οτιδήποτε άλλο.

Δύο συνέπειες που αξίζει να έχετε υπόψη:

1. **Ένα βαθμωτό μπορεί να είναι αληθές χωρίς να είναι λογικά αληθής τιμή.** Συμβολοσειρές, αναφορές, blessed αντικείμενα, μεγάλοι αριθμοί - όλα αληθή. Όταν ρωτάτε «είναι αυτό αληθές;» δεν ρωτάτε «είναι αυτό ίσο με 1;» - ρωτάτε «δεν ανήκει αυτό στο σύνολο των ψευδών;».

2. **Το αποτέλεσμα ενός λογικού τελεστή δεν είναι πάντα 1 ή 0.** Το επόμενο κεφάλαιο καλύπτει αυτό λεπτομερώς. Συνοπτικά: τα && και || επιστρέφουν έναν από τους *τελεστέους* τους, όχι μια κανονικοποιημένη λογική τιμή. Αυτό είναι χαρακτηριστικό, όχι ιδιοτροπία, και πολλά ιδιώματα της Perl εξαρτώνται από αυτό.

Δείτε επίσης

- *perlop - Αριθμητική σύγκριση* και *Σύγκριση συμβολοσειρών* - οι σελίδες αναφοράς για τα ==/eq και τις υπόλοιπες οικογένειες σύγκρισης που εισήχθησαν παραπάνω.

Τελεστές

Η Perl γράφει τους λογικούς τελεστές δύο φορές, σε δύο διαφορετικές κλάσεις προτεραιότητας. Οι συμβολικές μορφές σε στιλ C (`&&`, `||`, `!`) δένουν σφιχτά· οι λεκτικές μορφές (`and`, `or`, `not`) δένουν πολύ χαλαρά. Υπάρχει επίσης το `//` για το ιδίωμα *defined-or*, το `xor` για το αποκλειστικό «ή», και ο τριαδικός `?:` που είναι λογική επιλογή σε μεταμφίεση.

Ο πλήρης κατάλογος

Τελεστής	Διαβάζεται ως	Επιστρέφει	Προτεραιότητα
`!`	όχι	κανονικοποιημένο 1 ή κενή συμβολοσειρά	υψηλή
`&&`	και	αριστερό αν ψευδές, αλλιώς δεξί (ο *τελεστέος*)	υψηλή
`||`	ή	αριστερό αν αληθές, αλλιώς δεξί (ο *τελεστέος*)	υψηλή
`//`	defined-or	αριστερό αν ορισμένο, αλλιώς δεξί (ο *τελεστέος*)	υψηλή
`xor`	αποκλειστικό ή	κανονικοποιημένο 1 ή κενή συμβολοσειρά	πολύ χαμηλή
`not`	όχι	κανονικοποιημένο 1 ή κενή συμβολοσειρά	πολύ χαμηλή
`and`	και	αριστερό αν ψευδές, αλλιώς δεξί (ο *τελεστέος*)	πολύ χαμηλή
`or`	ή	αριστερό αν αληθές, αλλιώς δεξί (ο *τελεστέος*)	πολύ χαμηλή
`?:`	τριαδικός	μεσαίο αν η συνθήκη αληθής, αλλιώς δεξί (τελεστέος)	χαμηλή

Δύο πράγματα σε αυτόν τον πίνακα κάνουν το μεγαλύτερο μέρος της δουλειάς:

- **Τα `&&` και `||` επιστρέφουν τελεστέο, όχι λογική τιμή.** Αυτή είναι η ιδιότητα που κάνει τα ιδιώματα της Perl να λειτουργούν όπως λειτουργούν. Δείτε παρακάτω.
- **Οι συμβολικές και λεκτικές μορφές διαφέρουν στην προτεραιότητα, όχι στο νόημα.** Τα `&&` και `and` υπολογίζουν την ίδια λογική συνάρτηση. Διαφέρουν στο πόσο σφιχτά δένουν με τους άλλους τελεστές γύρω τους. Γι" αυτό βλέπετε `or die` μετά από κλήσεις συναρτήσεων αλλά `||` μέσα σε εκφράσεις - το κεφάλαιο για το ιδίωμα `or die` εξηγεί.

Βραχυκύκλωμα και ο κανόνας επιστροφής τελεστέου

Τα `&&` και `||` αποτιμώνται από αριστερά προς τα δεξιά και σταματούν μόλις προσδιοριστεί το αποτέλεσμα.

- `A && B` - αν το `A` είναι ψευδές, επιστρέφει το `A` χωρίς να αποτιμήσει το `B`. Αλλιώς επιστρέφει το `B`.
- `A || B` - αν το `A` είναι αληθές, επιστρέφει το `A` χωρίς να αποτιμήσει το `B`. Αλλιώς επιστρέφει το `B`.

Καθοριστικά, η **τιμή επιστροφής είναι ο ίδιος ο τελεστέος**, όχι ένα κανονικοποιημένο αληθές/ψευδές. Αυτό κάνει μοτίβα όπως `$config{port} || 8080` να λειτουργούν: αν το `$config{port}` έχει οριστεί (αληθές), η έκφραση είναι η τιμή του `$config{port}`· αν όχι, είναι το `8080`.

```
my $port = $config{port} || 8080;
my $name = $user_input   || "anonymous";
my $list = shift         || [];
```

Η ίδια ιδιότητα τροφοδοτεί το `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
```

Το open επιστρέφει αληθές στην επιτυχία, undef στην αποτυχία. Στην επιτυχία το βραχυκύκλωμα σταματά στα αριστερά και το die ποτέ δεν εκτελείται. Στην αποτυχία αποτιμάται ο δεξής τελεστέος, που είναι η κλήση die.

`//` - defined-or

Το `||` χειρίζεται τα `0`, `""`, και `"0"` το ίδιο με το undef. Μερικές φορές αυτό είναι λάθος: ένα ρυθμισμένο `0` είναι πραγματική τιμή, όχι ελλείπουσα. Το `//` βραχυκυκλώνει με βάση τον **ορισμό** αντί της αλήθειας:

- `A // B` - αν το A είναι ορισμένο, επιστρέφει το A. Αλλιώς επιστρέφει το B.

```
my $port    = $config{port} || 8080;   # 0 means "use default" - wrong
my $port    = $config{port} // 8080;   # 0 means "0", undef means "use default"
my $verbose = $opt{verbose} // 0;       # 0 is a real choice
```

Οι σύνθετες μορφές ανάθεσης `||=` και `//=` σας απαλλάσσουν από την επανάληψη της μεταβλητής στα αριστερά:

```
$config{port} //= 8080;   # set only if currently undef
$cache{$key}  ||= compute($key);   # set if currently false
```

Το xor και η απουσία μορφής υψηλής προτεραιότητας

Δεν υπάρχει ^^ στην Perl. Το xor υπάρχει μόνο ως λεκτική μορφή πολύ χαμηλής προτεραιότητας, και σε αντίθεση με τα and/or *δεν* βραχυκυκλώνει (δεν μπορεί - και οι δύο πλευρές πρέπει να αποτιμηθούν για να προσδιοριστεί το αποτέλεσμα). Επιστρέφει κανονικοποιημένο 1 ή `""`, όχι τελεστέο.

```
if ($admin xor $guest) { ... }   # exactly one of the two
```

Για xor πάνω σε τιμές αντί για αλήθεια, το κατά bit ^ εφαρμόζεται σε ακεραίους - καλύπτεται στο *κεφάλαιο εφαρμογών*.

`!` και `not`

Και τα δύο αρνούνται. Το `!` είναι η συμβολική μορφή υψηλής προτεραιότητας· το `not` είναι η λεκτική μορφή χαμηλής προτεραιότητας. Και τα δύο επιστρέφουν κανονικοποιημένη τιμή: 1 για «ο τελεστέος ήταν ψευδής», κενή συμβολοσειρά `""` για «ο τελεστέος ήταν αληθής».

```
my $missing = ! $config{port};        # 1 if port is unset/0/""/"0"
return if not @items;                 # cleaner reading than `if !
 ↪@items`
```

Ο τριαδικός `?:`

`COND ? X : Y` είναι λογική επιλογή: αν το `COND` είναι αληθές η τιμή είναι `X`, αλλιώς `Y`. Δεν είναι απλώς συντακτική ζάχαρη για `if`/`else` επειδή είναι *έκφραση* - έχει τιμή, μπορεί να ανατεθεί, να επιστραφεί, να περάσει ως όρισμα.

```
my $label = $count == 1 ? "1 item" : "$count items";
return $ok ? \%result : undef;
```

Προσοχή στην εμφώλευση: το `$a ? $b : $c ? $d : $e` αναλύεται ως `$a ? $b : ($c ? $d : $e)` - το αλυσιδωτό `?:` είναι δεξιά προσεταιριστικό, που συνήθως είναι αυτό που θέλετε για μια κλιμάκωση `if/elsif/else` αλλά αξίζει να το γνωρίζετε αντί να μαντεύετε.

Γιατί υπάρχουν οι λεκτικές μορφές: προτεραιότητα, όχι στιλ

Το `||` δένει σφιχτότερα από το `=`. Το `or` δένει χαλαρότερα από το `=` (ή το `,`, ή οτιδήποτε άλλο συμβατικού βάρους). Αυτός είναι ο μοναδικός λόγος που υπάρχουν οι λεκτικές μορφές.

```
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.
```

Εμπειρικός κανόνας:

- Μέσα σε μια έκφραση, όπου θέλετε η τιμή να ρέει σε μια μεταβλητή ή σε άλλον τελεστή: χρησιμοποιήστε `&&`, `||`, `!`, `//`.
- Μετά από μια πρόταση, όπου θέλετε μια παρενέργεια (`die`, `warn`, `return`) να ενεργοποιηθεί υπό συνθήκη: χρησιμοποιήστε `or`, `and`, `not`.

Τι πρέπει να θυμάστε από αυτό το κεφάλαιο

- Τα && και || δεν είναι κατηγορήματα που επιστρέφουν 1 ή 0 - επιστρέφουν έναν από τους τελεστέους τους. Αυτό είναι το θεμέλιο των ||, ||=, or die, και των περισσότερων σύντομων ιδιωμάτων προεπιλογής.
- Το // είναι το || για ορισμό, όχι αλήθεια. Χρησιμοποιήστε το όταν το 0 είναι έγκυρη τιμή.
- Οι λεκτικές μορφές υπάρχουν για να δένουν πιο χαλαρά από την ανάθεση, όχι για στιλ.
- Το xor υπάρχει, δεν έχει συμβολική μορφή, δεν βραχυκυκλώνει.

Δείτε επίσης

- *perlop - Λογικοί* - ο σύντροφος αναφοράς, με το πλαίσιο των γραμμών προτεραιότητας και τις σημειώσεις ειδικά για το PetaPerl.
- *perlop - Προτεραιότητα* - ο πλήρης πίνακας· οι γραμμές 5, 15, 16, 22, 23, 24 είναι οι τελεστές αυτής της σελίδας.
- *perlop - Τριαδικός* - ?: ως λογική επιλογή.

Πίνακες αληθείας

Μια δυαδική λογική συνάρτηση παίρνει δύο λογικές εισόδους και επιστρέφει μία λογική έξοδο. Υπάρχουν ακριβώς **δεκαέξι** τέτοιες συναρτήσεις - δηλαδή 2^(2^2): καθένας από τους τέσσερις συνδυασμούς εισόδου (0,0) (0,1) (1,0) (1,1) αντιστοιχίζεται ανεξάρτητα σε μία από δύο εξόδους, έτσι 2 × 2 × 2 × 2 = 16. Ούτε περισσότερες, ούτε λιγότερες.

Οι πέντε που έχουν δικό τους τελεστή Perl (&&, ||, xor, συν τις δύο τετριμμένες - πάντα-αληθές και πάντα-ψευδές) είναι αυτές που γνωρίζουν όλοι. Οι άλλες έντεκα είναι σιωπηλά κι αυτές εκεί, εκφρασμένες ως μικρές συνθέσεις. Αυτό το κεφάλαιο απαριθμεί και τις δεκαέξι, τις ονομάζει, και δείχνει πώς γράφεται η καθεμία στην Perl.

Η σύμβαση

Κάθε συνάρτηση περιγράφεται πλήρως από την έξοδό της για τις τέσσερις γραμμές εισόδου, γραμμένες από πάνω προς τα κάτω με αυτή τη σειρά:

```
a  b   →  output
0  0   →  bit 1
0  1   →  bit 2
1  0   →  bit 3
1  1   →  bit 4
```

Διαβάζοντας αυτά τα τέσσερα bit εξόδου ως αριθμό 4 bit, δίνεται στη συνάρτηση ο δείκτης της 0..15. Έτσι το AND είναι 0001 (μόνο η τελευταία γραμμή είναι αληθής), το OR είναι 0111, το XOR είναι 0110. Τα τρία τέταρτα του πίνακα είναι απλώς αναζήτηση της σωστής γραμμής.

Και οι δεκαέξι

#	Αλήθεια 00 01 10 11	Όνομα	Σύμβολ	Perl απευθείας	Perl ιδιωματικά
0	0 0 0 0	ψευδές (αντίφαση)	⊥	0 / ""	-
1	0 0 0 1	AND	∧	$a && $b	$a and $b
2	0 0 1 0	a AND NOT b	a ∧ ¬b	-	$a && !$b
3	0 0 1 1	a (αριστερή προβολή)	a	$a	-
4	0 1 0 0	NOT a AND b	¬a ∧ b	-	!$a && $b
5	0 1 0 1	b (δεξιά προβολή)	b	$b	-
6	0 1 1 0	XOR	⊕	$a xor $b	!$a != !$b / ($a?1:0) != ($b?1:0)
7	0 1 1 1	OR	∨	$a || $b	$a or $b
8	1 0 0 0	NOR	↓ (Peirce)	-	!($a || $b) / !$a && !$b
9	1 0 0 1	XNOR / ισοδυναμία	↔	-	!$a == !$b / ($a?1:0) == ($b?1:0)
10	1 0 1 0	NOT b	¬b	!$b	not $b
11	1 0 1 1	το b συνεπάγεται το a	b → a	-	!$b || $a
12	1 1 0 0	NOT a	¬a	!$a	not $a
13	1 1 0 1	το a συνεπάγεται το b	a → b	-	!$a || $b
14	1 1 1 0	NAND	↑ (Sheffer	-	!($a && $b) / !$a || !$b
15	1 1 1 1	αληθές (ταυτολογία)	⊤	1	-

Πέντε γραμμές έχουν απευθείας τελεστή Perl (&&, ||, xor, !, συν τις δύο σταθερές). Οι άλλες έντεκα είναι σύντομες συνθέσεις αυτών. Δεν υπάρχει θεμελιώδης λόγος που η Perl γράφει κάποιες απευθείας και άλλες όχι - είναι ιστορικό, κληρονομημένο από τη C. Κάθε στήλη στα δεξιά είναι κάτι που μπορείτε να πληκτρολογήσετε σήμερα.

Αξίζει να ξεχωρίσουμε δύο πράγματα από τον πίνακα.

Συνεπαγωγή: →

Η συνεπαγωγή είναι ο τελεστής που κανείς δεν νομίζει ότι χρησιμοποιεί, και όλοι χρησιμοποιούν συνεχώς. Το a → b διαβάζεται «το a συνεπάγεται το b» και είναι αληθές σε κάθε περίπτωση **εκτός** όταν το a είναι αληθές και το b είναι ψευδές. Αυτό είναι όλο.

a	b	a → b
0	0	1
0	1	1
1	0	0
1	1	1

Η γραφή σε Perl είναι `!$a || $b`. Αυτή η ταυτότητα -

a → b ≡ ¬a ∨ 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"
```

Η μορφή `unless ... !X || Y` είναι άβολη. Το επόμενο κεφάλαιο για τους νόμους του De Morgan δείχνει πώς να την αναστρέψετε σε κάτι αναγνώσιμο.

Μια δεύτερη χρήσιμη ταυτότητα: η συνεπαγωγή είναι **η άρνηση του XOR** *μόνο όταν γράφεται με έναν συγκεκριμένο τρόπο*. Μην το αποστηθίζετε αυτό - αποστηθίστε αντ" αυτού τον παρακάτω, που είναι ο πραγματικός πίνακας αναζήτησης για την εναλλαγή μεταξύ κοινών μορφών:

Στόχος	Ένας τρόπος	Ισοδύναμος τρόπος
`a → b`	`¬a ∨ b`	`¬(a ∧ ¬b)`
`a ↔ b` (XNOR)	`(a ∧ b) ∨ (¬a ∧ ¬b)`	`¬(a ⊕ b)`
`a ⊕ b` (XOR)	`(a ∧ ¬b) ∨ (¬a ∧ b)`	`¬(a ↔ b)`
`¬a`	`a ↑ a` (αυτο-NAND)	`a ⊕ 1`

Η τελευταία στήλη είναι μια προεπισκόπηση για το κεφάλαιο της λειτουργικής πληρότητας.

Η ισοδυναμία και το XOR είναι δυϊκά

Το XOR είναι αληθές ακριβώς όταν τα a και b διαφέρουν. Η ισοδυναμία (XNOR) είναι αληθής ακριβώς όταν συμφωνούν. Είναι αρνήσεις μεταξύ τους, και αυτό ακριβώς λένε οι γραμμές του πίνακα αληθείας (η γραμμή 6 είναι `0 1 1 0`, η γραμμή 9 είναι `1 0 0 1` - άρνηση κατά bit).

Στην Perl η πιο καθαρή γραφή και για τα δύο είναι να κανονικοποιηθεί κάθε τελεστέος σε `0` ή `1` πρώτα και στη συνέχεια να γίνει σύγκριση με `==` ή `!=`:

```
my $a01 = $a ? 1 : 0;
my $b01 = $b ? 1 : 0;

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
my $xor  = $a01 != $b01;   # true if they differ
my $xnor = $a01 == $b01;   # true if they agree
```

Γιατί η κανονικοποίηση; Επειδή τα `$a` και `$b` μπορεί να είναι οποιεσδήποτε αληθείς τιμές - οι συμβολοσειρές `"yes"` και `5` είναι αμφότερες αληθείς αλλά όχι ίσες. Η απευθείας σύγκρισή τους με `==` ή `eq` απαντά σε διαφορετική ερώτηση. Το λογικό XOR ρωτά «έχουν την ίδια *αλήθεια τιμής*;», όχι «είναι ίσες;».

Μια σύντομη εναλλακτική χωρίς τις προσωρινές τιμές είναι `!$a != !$b` - κάθε `!` επιστρέφει το κανονικό `1` ή `""`, και στη συνέχεια το `!=` τις συγκρίνει. Δουλεύει, αλλά διαβάζεται πλάγια· η ρητή μορφή `? 1 : 0` είναι πιο αναγνώσιμη.

Διαβάζοντας τον πίνακα αντίστροφα

Μερικές φορές θα γνωρίζετε το μοτίβο αληθείας που θέλετε και θα χρειαστεί να βρείτε τον τελεστή. Παράδειγμα: «Θέλω μια συνάρτηση που να είναι αληθής εκτός αν τα a και b είναι αμφότερα ψευδή.» Διαβάστε τις γραμμές: `a=0,b=0` → `0`, `a=0,b=1` → `1`, `a=1,b=0` → `1`, `a=1,b=1` → `1`. Αυτό είναι `0 1 1 1` - γραμμή 7 - OR.

Αυτό ακούγεται τετριμμένο εδώ, αλλά είναι ακριβώς πώς αποσφαλματώνετε μια μπερδεμένη συνθήκη: γράψτε τον πίνακα αληθείας της, διαβάστε τα τέσσερα bit, βρείτε τη γραμμή, και έχετε το κανονικό όνομα και την απλούστερη γραφή.

Τι πρέπει να θυμάστε από αυτό το κεφάλαιο

- Υπάρχουν ακριβώς δεκαέξι δυαδικές συναρτήσεις αληθείας· κάθε λογική έκφραση σε οποιαδήποτε γλώσσα υπολογίζει μία από αυτές.
- Πέντε από τις δεκαέξι έχουν αποκλειστικό τελεστή Perl. Οι άλλες έντεκα είναι σύντομες συνθέσεις και ο παραπάνω πίνακας είναι η αναζήτηση.
- Η συνεπαγωγή `a → b` είναι `¬a ∨ b`. Εμφανίζεται συνεχώς κάτω από την επιφάνεια προτάσεων `unless` και `if`.
- Το XOR και το XNOR είναι αρνήσεις μεταξύ τους. Η πιο καθαρή γραφή σε Perl και για τα δύο ξεκινά με την κανονικοποίηση των τελεστέων σε `0` ή `1`.

Οι νόμοι του De Morgan

Δύο ταυτότητες, που οφείλονται στον Augustus De Morgan, γραμμένες εδώ και πάνω από έναν αιώνα σε κάθε εγχειρίδιο τυπικής λογικής, και το πιο χρήσιμο μεμονωμένο ζεύγος εξισώσεων που μπορεί να γνωρίζει ένας ενεργός προγραμματιστής.

$$\neg(a \wedge b) \equiv \neg a \vee \neg b$$

$$\neg(a \vee b) \equiv \neg a \wedge \neg b$$

Με λόγια: το πέρασμα ενός `not` μέσα από μια έκφραση σε παρενθέσεις αντιστρέφει κάθε τελεστή στο εσωτερικό (το ∧ γίνεται ∨, το ∨ γίνεται ∧) και αρνείται κάθε τελεστέο. Ισοδύναμα: η εξαγωγή ενός `not` προς τα έξω αντιστρέφει κάθε τελεστή και αφαιρεί τις αρνήσεις από τους τελεστέους.

Γιατί έχει σημασία στην Perl

Το `unless` είναι η κανονική περίπτωση. Το `unless (X)` είναι `if (!X)` - και τη στιγμή που το X είναι το ίδιο σύνθετο, έχετε ένα προπορευόμενο `not` πάνω από μια έκφραση σε παρενθέσεις: ακριβώς το σκηνικό του De Morgan.

```
unless ($a == $x && $b == $y) { ... }
```

Η συνθήκη κάτω από το `unless` είναι `!($a == $x && $b == $y)`. Εφαρμόστε De Morgan: αυτό είναι `!($a == $x) || !($b == $y)`. Κάθε `!=` είναι η άρνηση του αντίστοιχου `==`, επομένως η καθαρή μορφή είναι:

```
if ($a != $x || $b != $y) { ... }
```

Συνέβησαν τρεις μετασχηματισμοί, καθένας μηχανικός:

1. Το `unless` αντιστράφηκε σε `if` (το εξωτερικό `not`).
2. Το `&&` αντιστράφηκε σε `||` (De Morgan, στο εσωτερικό).
3. Το `==` αντιστράφηκε σε `!=` σε κάθε τελεστέο (οι αρνήσεις που τοποθέτησε ο De Morgan στους τελεστέους ακυρώνονται στην αντίθετη σύγκριση).

Αυτό *δεν* είναι στιλιστική προτίμηση. Κάθε προγραμματιστής που έχει αποσφαλματώσει μια λανθασμένη συνθήκη έχει κοιτάξει επίμονα ένα `unless (! ... && ...)` για ένα λεπτό παραπάνω· το επίπεδο `if` διαβάζεται σε ένα πέρασμα.

Ένα επεξεργασμένο παράδειγμα

Μια πραγματική μορφή, ελαφρώς μπερδεμένη:

```
unless ($x !~ /^\d+$/ && $y !~ /^[a-z]+$/) {
    print "at least one of \$x and \$y looks valid\n";
}
```

Αυτό λέει: *εκτός αν* το `$x` είναι μη αριθμητικό και το `$y` είναι μη αλφαβητικό, τύπωσε. Πέντε αρνήσεις στοιβαγμένες σε δύο γραμμές. Διατρέξτε το:

Η συνθήκη κάτω από το `unless` είναι

```
!( $x !~ /^\d+$/  &&  $y !~ /^[a-z]+$/ )
```

Ο De Morgan αντιστρέφει το `&&` σε `||` και αρνείται κάθε πλευρά:

```
!($x !~ /^\d+$/)  ||  !($y !~ /^[a-z]+$/)
```

Το `!($x !~ ...)` είναι απλώς `$x =~ ...` (διπλή άρνηση μιας αντιστοιχίας regex). Το ίδιο για τη δεξιά πλευρά:

```
$x =~ /^\d+$/  ||  $y =~ /^[a-z]+$/
```

Έτσι η καθαρισμένη εκδοχή είναι:

```
if ($x =~ /^\d+$/ || $y =~ /^[a-z]+$/) {
    print "at least one of \$x and \$y looks valid\n";
}
```

Πέντε αρνήσεις κατέβηκαν στο μηδέν. Η συνθήκη τώρα διαβάζεται ακριβώς όπως το έλεγε το σχόλιο: *αν το* `$x` *είναι αριθμητικό ή το* `$y` *είναι αλφαβητικό.*

Το ίδιο τέχνασμα για το OR

Η δεύτερη μορφή του νόμου είναι η συμμετρική περίπτωση:

```
unless ($a == 0 || $b == 0) { divide($a, $b) }
```

γίνεται

```
if ($a != 0 && $b != 0) { divide($a, $b) }
```

- same mechanical steps, only this time the inner operator is `||` flipping to `&&`.

NAND, NOR - η ίδια ταυτότητα, διαβασμένη από την άλλη πλευρά

Η γραμμή NAND στον πίνακα αληθείας από το προηγούμενο κεφάλαιο έχει δύο καταχωρίσεις στη στήλη «Perl ιδιωματικά»:

```
14   NAND    !($a && $b)    /    !$a || !$b
```

Αυτές είναι ίσες *λόγω του πρώτου νόμου του De Morgan*. Το NAND είναι ακριβώς η άρνηση του AND, και το πέρασμα της άρνησης προς τα μέσα αντιστρέφει τον τελεστή και αρνείται τους τελεστέους. Η γραμμή NOR λειτουργεί με τον ίδιο τρόπο για τον δεύτερο νόμο:

```
8    NOR     !($a || $b)    /    !$a && !$b
```

Έτσι ο De Morgan δεν είναι ξεχωριστό γεγονός που πρέπει να θυμάστε δίπλα στα NAND και NOR - *είναι* η ταυτότητα που μετατρέπει τη μία γραφή στην άλλη.

Μηχανική, σε μία παράγραφο

Η άρνηση κατανέμεται πάνω στα ∧ και ∨ αντιστρέφοντας τον σύνδεσμο και αρνούμενη κάθε τελεστέο. Η άρνηση επίσης αυτο-ακυρώνεται: ¬¬x ≡ x. Οι δύο κανόνες μαζί σας επιτρέπουν να πιέσετε ένα not αυθαίρετα βαθιά, ή να το εξάγετε αυθαίρετα προς τα έξω, από οποιαδήποτε έκφραση δομημένη από ∧, ∨, και ¬. Αυτό είναι όλο το περιεχόμενο των νόμων του De Morgan - κάθε «περίπλοκο unless» που ξαναγράφετε είναι μία εφαρμογή αυτών των δύο κανόνων συν την ακύρωση διπλής άρνησης.

Πρακτική διαδικασία

Όταν κοιτάζετε επίμονα μια συνθήκη που δεν σας αρέσει:

1. Αν η εξωτερική δεσμευμένη λέξη είναι `unless`, προσθέστε ένα `not` νοητικά μπροστά και κάντε το `if`.

2. Αν το προπορευόμενο `not` βρίσκεται πάνω από ένα `&&` ή `||` σε παρενθέσεις, κατανείμετέ το: αντιστρέψτε τον σύνδεσμο, αρνηθείτε κάθε τελεστέο.
3. Αν μια υποέκφραση είναι `!(x == y)`, αντικαταστήστε με `x != y`. Το ίδιο για `!~` ↔ `=~`, `!defined` ↔ `defined`, κ.λπ.
4. Σταματήστε όταν δεν υπάρχουν άλλες προπορευόμενες αρνήσεις να πιέσετε ή άλλες διπλές αρνήσεις να ακυρώσετε.

Στην πράξη θα κάνετε και τα τέσσερα βήματα στο μυαλό σας σε ένα πέρασμα. Το νόημα της διαδικασίας είναι ότι υπάρχει μία - δεν αναζητάτε τη σωστή μορφή, την παράγετε.

Τι πρέπει να θυμάστε από αυτό το κεφάλαιο

- Δύο ταυτότητες: το πέρασμα του ¬ μέσα από ∧ το αντιστρέφει σε ∨ (και αντίστροφα) και αρνείται κάθε τελεστέο.
- Τα περισσότερα «άσχημα `unless`» απέχουν μία μηχανική αντιστροφή De Morgan από ένα καθαρό `if`.
- Τα NAND και NOR δεν είναι επιπλέον γεγονότα· είναι οι νόμοι του De Morgan διαβασμένοι ως ζεύγος ισοδύναμων γραφών.

Δείτε επίσης

- *perlop - Λογικοί* - ο σύντροφος αναφοράς τελεστών. Οι μετασχηματισμοί De Morgan εδώ ισχύουν σε κάθε έκφραση Perl που χρησιμοποιεί `&&`, `||`, `!`, και τη δεσμευμένη λέξη `unless`.

Λειτουργική πληρότητα

Ένα σύνολο λογικών τελεστών είναι *λειτουργικά πλήρες* όταν καθεμία από τις δεκαέξι δυαδικές συναρτήσεις αληθείας μπορεί να κατασκευαστεί από τελεστές αυτού του συνόλου, μέσω σύνθεσης. Τα κλασικά πλήρη σύνολα που ήδη γνωρίζετε είναι τα `{∧, ∨, ¬}` και `{→, ⊥}`. Το εκπληκτικό - και χρήσιμο - αποτέλεσμα είναι ότι **ένας μόνο τελεστής αρκεί**, αλλά μόνο δύο από τις δεκαέξι δυαδικές συναρτήσεις έχουν αυτή την ιδιότητα: το NAND και το NOR.

Το NAND πήρε το όνομά του από τον Henry M. Sheffer (1913) και γράφεται ↑· το NOR πήρε το όνομά του από τον Charles Sanders Peirce (που το περιέγραψε νωρίτερα) και γράφεται ↓. Είναι οι δύο *γραμμές Sheffer*.

Δομώντας τα πάντα από NAND

Το NAND από μόνο του σας δίνει και τις δεκαέξι συναρτήσεις. Οι παρακάτω κατασκευές δείχνουν τις τέσσερις πιο χρήσιμες - NOT, AND, OR, XOR - με όρους του ↑. Σκεφτείτε το `a ↑ b` ως `!($a && $b)`.

Στόχος	Με όρους NAND	Γιατί
¬a	a ↑ a	Το !(a ∧ a) είναι !a
a ∧ b	(a ↑ b) ↑ (a ↑ b)	NAND και μετά άρνηση του αποτελέσματος, δηλ. !!(a ∧ b) ≡ a ∧ b
a ∨ b	(a ↑ a) ↑ (b ↑ b)	De Morgan: a ∨ b ≡ ¬(¬a ∧ ¬b) ≡ ¬a ↑ ¬b
a ⊕ b	(a ↑ (a ↑ b)) ↑ (b ↑ (a ↑ b))	Το XOR είναι το AND των (a ∨ b) και ¬(a ∧ b) - αφήνεται ως άσκηση

Οι ίδιες κατασκευές υπάρχουν με NOR· είναι τα δυϊκά κατά De Morgan.

Στην Perl

Δεν θα γράψετε τον κώδικά σας με αυτόν τον τρόπο σε παραγωγή - το (a ↑ a) ↑ (b ↑ b) είναι χειρότερη γραφή του OR από ό,τι είναι το ||. Αλλά αξίζει να δείτε τις κατασκευές να τρέχουν στην πραγματικότητα, επειδή ο αφηρημένος ισχυρισμός («τα πάντα ανάγονται σε έναν τελεστή») γίνεται συγκεκριμένος όταν τον πληκτρολογείτε.

```
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;
    }
}
```

Έξοδος:

```
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
```

Κάθε γραμμή ταιριάζει με τον πίνακα αληθείας από το προηγούμενο κεφάλαιο.

Γιατί έχει σημασία

Τρία πράγματα, με σειρά από το πρακτικό προς το βαθύ:

1. **Εξηγεί το πανηγύρι των ταυτοτήτων.** Το NAND, το NOR, και οι έντεκα μη απευθείας γραμμές του κεφαλαίου του πίνακα αληθείας δεν είναι ένας ζωολογικός κήπος ξεχωριστών γεγονότων. Όλες είναι εκφράσιμες από έναν μόνο τελεστή· οι έντεκα μη απευθείας γραφές στην Perl είναι απλώς αναγνώσιμες συντομεύσεις για σύντομες συνθέσεις NAND ή NOR.

2. **Έτσι λειτουργεί στην πραγματικότητα το πυρίτιο.** Η λογική στατικού CMOS παράγει NAND και NOR φθηνότερα από AND και OR - μια πύλη AND δύο εισόδων κατασκευάζεται ως NAND ακολουθούμενη από αναστροφέα. Κάθε άλλη πύλη σε ένα τσιπ είναι σύνθεση από NAND (ή NOR). Το μοτίβο επαναλαμβάνεται στα εσωτερικά επιλυτών και μηχανών περιορισμών: οι επιλυτές SAT κανονικοποιούν σε CNF ακριβώς επειδή το OR-από-(κυριολεκτικούς) και AND-από-(προτάσεις) είναι το ελάχιστο λεξιλόγιο που χρειάζονται.

3. **Διδάσκει την πειθαρχία της κανονικής μορφής.** Κάθε λογική έκφραση έχει πολλές ισοδύναμες μορφές, αλλά μόνο λίγες είναι *κανονικές*. Η CNF (συζευκτική κανονική μορφή: ένα AND από OR πιθανώς-αρνημένων κυριολεκτικών) και η DNF (διαζευκτική κανονική μορφή: ένα OR από AND) είναι οι δύο πιο συνηθισμένες. Η αναγωγή ενός μπερδέματος σε CNF ή DNF είναι μια καθαρά μηχανική διαδικασία βασισμένη στον De Morgan συν την επιμεριστικότητα, και μόλις η έκφρασή σας είναι σε κανονική μορφή, μπορείτε να τη συγκρίνετε με άλλη έκφραση κανονικής μορφής για να ελέγξετε ισοδυναμία - χωρίς πίνακες αληθείας, χωρίς δοκιμές.

Δεν θα υλοποιήσετε επιλυτή SAT την επόμενη εβδομάδα. Αλλά την επόμενη φορά που θα βρεθείτε να γράφετε `if ((!$a && $b) || (!$c && $b) || ($a && !$c))` και θα αναρωτιέστε αν η προηγούμενη έκδοση ήταν ισοδύναμη, έχετε ένα εργαλείο: επιμερίστε, παραγοντοποιήστε, κανονικοποιήστε. Δύο εκφράσεις που ανάγονται στην ίδια DNF υπολογίζουν την ίδια συνάρτηση.

Τι πρέπει να θυμάστε από αυτό το κεφάλαιο

- Το NAND και το NOR είναι το καθένα μεμονωμένα επαρκές για να εκφράσει κάθε λογική συνάρτηση. Κανένας άλλος τελεστής δύο εισόδων δεν έχει αυτή την ιδιότητα.
- Οι κατασκευές είναι σύντομες και μπορείτε να τις επαληθεύσετε στην Perl σε δέκα γραμμές.
- Το ζητούμενο δεν είναι να γράφετε κώδικα σε NAND, αλλά να γνωρίζετε ότι και οι δεκαέξι δυαδικές συναρτήσεις ζουν πάνω σε ένα θεμέλιο. Οι φαινομενικά ανόμοιες γραμμές του κεφαλαίου του πίνακα αληθείας είναι μια οικογένεια.

Εφαρμογές

Τα προηγούμενα κεφάλαια αφορούσαν τη λογική Boole στο αφηρημένο. Αυτό αφορά πού εμφανίζεται πιο συχνά σε λειτουργική Perl: αριθμητική κατά bit, κανονικές εκφράσεις, και τα ιδιώματα βραχυκυκλώματος που παίρνουν τη θέση των ρητών συνθηκών.

Κατά bit: λογική σε bit ακεραίων

Οι τελεστές κατά bit εφαρμόζουν λογικές πράξεις σε κάθε ζεύγος bit σε δύο ακεραίους παράλληλα. Ένας ακέραιος 32 bit είναι, από την οπτική γωνία των τελεστών, τριάντα δύο παράλληλες τιμές ενός bit.

Τελ.	Διαβάζεται ως	Κανόνας ανά bit
&	AND κατά bit	κάθε bit εξόδου = a∧b
|	OR κατά bit	κάθε bit εξόδου = a∨b
^	XOR κατά bit	κάθε bit εξόδου = a ⊕ b
~	NOT κατά bit	κάθε bit εξόδου = ¬a
<<	αριστερή ολίσθηση	ολίσθηση bit προς τα αριστερά, γέμισμα με μηδενικά από τα δεξιά
>>	δεξιά ολίσθηση	ολίσθηση bit προς τα δεξιά

Οι ίδιοι λογικοί τελεστές - ∧, ∨, ⊕, ¬ - εμφανίζονται εδώ σε καθαρά αριθμητική μορφή. Αυτό δεν είναι σύμπτωση· είναι ο ορισμός.

Θέση, καθάρισμα, εναλλαγή, έλεγχος μιας σημαίας

Οι τέσσερις βασικές πράξεις σημαίας σε ένα μόνο 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
```

Καθεμία από αυτές είναι μια εφαρμογή ενός bit ενός λογικού τελεστή. Η θέση είναι `bit ∨ flag`, το καθάρισμα είναι `bit ∧ ¬flag`, η εναλλαγή είναι `bit ⊕ flag`, ο έλεγχος είναι `bit ∧ flag`.

Ανταλλαγή με XOR: ανταλλαγή χωρίς προσωρινή τιμή

Το XOR έχει δύο ιδιότητες που συνδυάζονται σε ένα αξιομνημόνευτο τέχνασμα: `a ⊕ a = 0`, και `a ⊕ b ⊕ b = a`. Εφαρμόστε τις διαδοχικά και μπορείτε να ανταλλάξετε δύο ακεραίους χωρίς τρίτη μεταβλητή:

```
my ($a, $b) = (0xFEED, 0xBEEF);

$a ^= $b;      # a := a ⊕ b
$b ^= $a;      # b := b ⊕ (a ⊕ b) = a
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
$a ^= $b;      # a := (a ⊕ b) ⊕ a  = b

print "a=$a b=$b\n";   # a=48879 b=65261  (0xBEEF, 0xFEED)
```

Το τέχνασμα δεν είναι στην πραγματικότητα χρήσιμο στην Perl - το `($a, $b) = ($b, $a)` είναι ταχύτερο, σαφέστερο, και λειτουργεί σε οποιοδήποτε βαθμωτό συμπεριλαμβανομένων συμβολοσειρών και αναφορών. Αλλά εμφανίζεται σε δύο σημεία που έχουν σημασία:

- **Ενσωματωμένος κώδικας χωρίς ελεύθερο καταχωρητή.** Ένας μικροελεγκτής με τρεις τιμές να ζογκλάρει σε δύο καταχωρητές καταφεύγει σε αυτό.
- **Λαογραφία συνεντεύξεων.** Το να γνωρίζετε ότι υπάρχει και *γιατί* λειτουργεί (το XOR είναι το ίδιο το αντίστροφό του) αξίζει τα τριάντα δευτερόλεπτα που απαιτούνται για να το διαβάσετε.

Ο λόγος που λειτουργεί είναι ακριβώς η λογική ταυτότητα από το κεφάλαιο του πίνακα αληθείας: `x ⊕ y ⊕ y = x`. Καθεμία από τις τρεις παραπάνω γραμμές είναι μία εφαρμογή αυτής της ταυτότητας.

Συνηθισμένα τεχνάσματα bit

Μια χούφτα μοτίβων που θα δείτε σε κώδικα ευαίσθητο στην απόδοση:

```
$x &  ($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
1 << $n                 # the integer with only bit $n set
$x & (1 << $n)          # is bit $n set in $x?
```

Καθένα από αυτά είναι μια άσκηση στην παρακολούθηση τού τι κάνουν τα bit - καθαρός λογικός συλλογισμός εφαρμοσμένος 32 (ή 64) φορές παράλληλα.

Κανονικές εκφράσεις: λογική σε σύνολα συμβολοσειρών

Μια regex ταιριάζει σε ένα *σύνολο* συμβολοσειρών. Το κενό σύνολο, το μονοσύνολο `{"foo"}`, το άπειρο σύνολο «οτιδήποτε ξεκινά με ψηφίο» - όλα είναι σύνολα, και η regex δηλώνει ένα από αυτά.

Μόλις δείτε μια regex ως σύνολο, οι λογικές πράξεις έχουν γεωμετρικό νόημα:

Λογική πράξη	Σε σύνολα	Σε σύνταξη regex
OR (∨)	ένωση	εναλλαγή: `foo|bar`
AND (∧)	τομή	ζεύγος πρόσθιων διεκδικήσεων: `(?=.{4})(?=\d+$)`
NOT (¬)	συμπλήρωμα	αρνητική πρόσθια διεκδίκηση: `(?!foo)`
AND-NOT (a ∧ ¬b)	διαφορά	`(?=A)(?!B)A`

Δύο συγκεκριμένα σημεία που αξίζει να ξεχωρίσουμε.

Η εναλλαγή είναι OR πάνω σε σύνολα

```
$s =~ /yes|no|maybe/;
```

Ταιριάζει στην ένωση τριών μονοσυνόλων. Δεν υπάρχει τίποτε περισσότερο σε αυτό· το `|` στη σύνταξη regex είναι ακριβώς το λογικό ∨.

Μέσα σε μια κλάση χαρακτήρων το νόημα είναι το ίδιο:

```
$s =~ /[abc]/;        # union of {"a"}, {"b"}, {"c"}
$s =~ /[^abc]/;       # complement: anything NOT in {"a","b","c"}
```

Το `[^...]` είναι η σύνταξη regex για το ¬ εφαρμοσμένο σε ένα σύνολο χαρακτήρων.

Οι διεκδικήσεις γύρω είναι AND και NOT

Μια regex χωρίς διεκδικήσεις γύρω καταναλώνει χαρακτήρες καθώς ταιριάζει· τα δύο άκρα μιας εναλλαγής `A|B` δεν μπορούν να ταιριάξουν αμφότερα στο ίδιο εύρος (μόνο ένας κλάδος κερδίζει). Για να εκφράσετε *και* το `A` *και* το `B` στην ίδια θέση, χρειάζεστε διεκδίκηση γύρω: μια διεκδίκηση μηδενικού πλάτους που απαιτεί μια ιδιότητα χωρίς κατανάλωση.

```
# matches only if the rest of the string is BOTH digits AND ≤ 4␣
 ↪chars
$s =~ /^(?=\d+$)(?=.{1,4}$)/;
#       \_____/\_______/
#         A         B          intersection: A ∧ B
```

Το `(?=...)` είναι θετική πρόσθια διεκδίκηση· το `(?!...)` είναι αρνητική πρόσθια διεκδίκηση (το λογικό NOT). Συνδυάζοντάς τα σας δίνεται η πλήρης άλγεβρα συνόλων πάνω σε κατηγορήματα regex:

```
# "starts with a digit but is not the literal '0'":
# (digit) ∧ ¬(literal "0")
$s =~ /^(?=\d)(?!0$)/;
```

Γιατί βοηθά αυτό το πλαίσιο

Μόλις διαβάσετε τη regex με αυτόν τον τρόπο, η αναγνωσιμότητα σύνθετων μοτίβων βελτιώνεται δραματικά. Μια regex με δύο πρόσθιες διεκδικήσεις δεν είναι «δύο regex κολλημένες μαζί» - είναι η τομή δύο συνόλων. Μια αρνητική πρόσθια διεκδίκηση ακολουθούμενη από μια θετική αντιστοιχία δεν είναι «πρώτα απόρριψη, μετά αντιστοιχία» - είναι διαφορά συνόλων. Η λογική άλγεβρα που ήδη γνωρίζετε είναι η άλγεβρα των regex· μόνο ο συμβολισμός αλλάζει.

Ροή ελέγχου: λογική βραχυκυκλώματος ως υπό συνθήκη εκτέλεση

Το κεφάλαιο για τους τελεστές εισήγαγε τον κανόνα επιστροφής τελεστέου για τα &&, ||, και //. Αυτός ο κανόνας, συν η προτεραιότητα, παράγει μια μικρή οικογένεια ιδιωμάτων που αντικαθιστούν τη ρητή `if`/`else` για σύντομη λογική υπό συνθήκη:

```
# 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*

- Καθένα από αυτά είναι μια λογική έκφραση επιλεγμένη για τις *παρενέργειές* της - το βραχυκύκλωμα αποφασίζει αν θα εκτελεστεί καν ο δεξής τελεστέος. Το `open ... or die` λειτουργεί ακριβώς επειδή το `or` δεν αποτιμά τη δεξιά πλευρά του όταν η αριστερή είναι αληθής.

Δύο συμβουλές.

Χρησιμοποιήστε // και //= όταν το 0 ή το "" είναι έγκυρη τιμή. Αυτή η μεμονωμένη αλλαγή έχει αποτρέψει περισσότερα σφάλματα από οποιοδήποτε άλλο σύγχρονο ιδίωμα της Perl· το || που πέφτει σε προεπιλογή πάνω σε ρυθμισμένο μηδέν είναι ένας από τους κλασικούς τρόπους απώλειας δεδομένων.

Καταφύγετε στο `?:` όταν διαφορετικά θα δομούσατε μια τριάδα `if`/`else`/βαθμωτή-ανάθεση. Ο τριαδικός διατηρεί την έκφραση ως έκφραση και την ανάθεση σε μία γραμμή:

```
# verbose
my $kind;
if ($n == 1) { $kind = 'singular' }
else         { $kind = 'plural'   }

# idiomatic
my $kind = $n == 1 ? 'singular' : 'plural';
```

Μην εμφωλεύετε τον `?:` πάνω από δύο επίπεδα βαθιά. Πέρα από τα δύο, η αλυσιδωτή μορφή είναι δυσκολότερη να διαβαστεί από την `if`/`elsif`/`else`. Ο τριαδικός είναι για *επιλογή*· οι διαδοχικές αποφάσεις ανήκουν σε ένα μπλοκ.

Τι πρέπει να θυμάστε από αυτό το κεφάλαιο

- Τα κατά bit &, |, ^, ~ είναι οι λογικοί τελεστές εφαρμοσμένοι ανά bit παράλληλα· οι θέση/καθάρισμα/εναλλαγή/έλεγχος σημαίας είναι οι τέσσερις εφαρμογές ενός bit.
- Μια regex δηλώνει ένα σύνολο συμβολοσειρών· το `|` είναι ένωση, το `[^...]` είναι συμπλήρωμα, τα `(?=)` και `(?!)` φτιάχνουν τομή και διαφορά.
- Τα `&&`, `||`, `//`, `?:` σας δίνουν το μεγαλύτερο μέρος της υπό συνθήκη ροής ελέγχου χωρίς να γράφετε `if`. Χρησιμοποιήστε `//` όταν το `0` είναι πραγματικό· χρησιμοποιήστε `?:` για σύντομη επιλογή μέσα σε μια έκφραση.

Δείτε επίσης

- *perlop - Κατά bit* - ο σύντροφος αναφοράς για την ενότητα κατά bit, με την πλήρη σημασιολογία ολίσθησης και τις μορφές συμβολοσειράς bytes με κατάληξη `&. |. ^. ~.`.
- *perlop - Σύνδεση* - τα `=~` και `!~`, οι τελεστές που συνδέουν συμβολοσειρές με αντιστοίχιση regex.
- *Οδηγός κανονικών εκφράσεων*
  - Η πλήρης αναφορά γλώσσας regex βρίσκεται στον οδηγό regex:
- *perlop - Λογικοί* - οι τελεστές βραχυκυκλώματος πίσω από τα ιδιώματα ροής ελέγχου.

4.15.4 Τι θα μπορείτε να κάνετε μετά την ανάγνωση

- Να κοιτάξετε οποιαδήποτε λογική έκφραση σε πηγαίο κώδικα Perl και να ονομάσετε τη συνάρτηση αληθείας που υπολογίζει.
- Να μετατρέπετε μεταξύ των τεσσάρων ισοδύναμων μορφών μιας συνεπαγωγής (`a → b`, `!a || b`, `!(a && !b)`, της αντιθετοαντιστροφής) χωρίς να χρειάζεται να σκεφτείτε.
- Να ανάγετε ένα `unless` με δύο αρνημένες υποπροτάσεις σε ένα επίπεδο `if` σε ένα μηχανικό βήμα.
- Να αναγνωρίζετε πότε μια εναλλαγή regex είναι ένα λογικό OR πάνω σε σύνολα και πότε ένα ζεύγος διεκδικήσεων είναι ένα λογικό AND, και να χρησιμοποιείτε αυτό το πλαίσιο για να διαβάζετε σύνθετες κανονικές εκφράσεις.
- Να χρησιμοποιείτε τους τελεστές κατά bit για να θέτετε, να καθαρίζετε, να εναλλάσσετε και να ελέγχετε επιμέρους σημαίες χωρίς να συμβουλεύεστε αναφορά.
- Να επιλέγετε το σωστό ιδίωμα βραχυκυκλώματος (`||`, `//`, `||=`, `//=`, `or die`) για την περίσταση, γνωρίζοντας γιατί λειτουργεί το καθένα.

4.16 Επικοινωνία μεταξύ διεργασιών - ένας οδηγός εκμάθησης

Δύο προγράμματα που χρειάζεται να επικοινωνήσουν έχουν μια χούφτα κανάλια να επιλέξουν: έναν σωλήνα, ένα ονομασμένο FIFO στον δίσκο, μια υποδοχή πάνω από το δίκτυο, ένα ζεύγος υποδοχών στη μνήμη ή ένα κομμάτι κοινόχρηστης μνήμης System V. Αυτός ο οδηγός εκμάθησης είναι ένα βιβλίο συνταγών. Κάθε σελίδα σας δίνει ένα πλήρες,

εκτελέσιμο πρόγραμμα - έναν παραγωγό και έναν καταναλωτή, ή έναν πελάτη και έναν διακομιστή - που μπορείτε να αντιγράψετε, να εκτελέσετε και να προσαρμόσετε.

Οι μηχανισμοί κάθε καναλιού - τι επιστρέφει η `socket`, πώς η `fork` διασπά μια διεργασία, τι κάνει το `%SIG` - τεκμηριώνονται ήδη στην αναφορά ανά ενσωματωμένη συνάρτηση και στο κεφάλαιο των σημάτων. Αυτός ο οδηγός εκμάθησης δεν τα επαναλαμβάνει. Τα συναρμολογεί σε λειτουργικά προγράμματα IPC και συνδέει κάθε κλήση πίσω στη σελίδα αναφοράς της.

Η PetaPerl τρέχει μόνο σε Linux, οπότε όλα εδώ είναι POSIX. Δεν υπάρχει IPC Win32, ούτε εναλλακτική ονομασμένου σωλήνα σε Windows, ούτε `Win32::Process`. Οι συνταγές χρησιμοποιούν τις κλήσεις συστήματος που πραγματικά παρέχει ένας πυρήνας Linux.

4.16.1 Επιλογή καναλιού

Επιλέξτε το ελαφρύτερο κανάλι που μεταφέρει αυτό που χρειάζεστε:

- **Ένα παιδί που εκτελεί μια εντολή** - θέλετε την έξοδο της `sort` ή την είσοδο του `gzip`. Χρησιμοποιήστε ένα *άνοιγμα σωλήνα*, ή τα `IPC::Open2` / `IPC::Open3` όταν χρειάζεστε και τις δύο κατευθύνσεις. Δείτε *Υποδιεργασίες και συνεργαζόμενες διεργασίες*.
- **Δύο σχετιζόμενες διεργασίες, ένα μηχάνημα** - ένας γονέας και ένα παιδί που μόλις δημιούργησε με fork, που μοιράζονται μια ροή bytes. Χρησιμοποιήστε `pipe` ή `socketpair`. Δείτε *Υποδιεργασίες* και *Υποδοχές UNIX-domain και UDP*.
- **Μη σχετιζόμενες διεργασίες, ένα μηχάνημα** - δύο προγράμματα που ξεκίνησαν ανεξάρτητα και συναντιούνται σε μια γνωστή διαδρομή. Χρησιμοποιήστε ένα *FIFO* ή μια υποδοχή UNIX-domain. Δείτε *FIFO* και *Υποδοχές UNIX-domain και UDP*.
- **Διεργασίες σε διαφορετικά μηχανήματα** - ένας πελάτης και ένας διακομιστής δικτύου. Χρησιμοποιήστε μια υποδοχή TCP. Δείτε *Υποδοχές TCP*.
- **Ένα datagram, ρίξε-και-ξέχνα** - καμία σύνδεση, διατήρηση των ορίων μηνύματος. Χρησιμοποιήστε μια υποδοχή UDP. Δείτε *Υποδοχές UNIX-domain και UDP*.
- **Ένας κοινόχρηστος μετρητής ή ενταμιευτής χωρίς αντιγραφή** - σηματοφόροι και κοινόχρηστη μνήμη System V. Χρησιμοποιήστε `semget` και `shmget`. Δείτε *IPC System V*.

4.16.2 Πώς είναι οργανωμένος αυτός ο οδηγός εκμάθησης

Κάθε σελίδα στέκεται από μόνη της. Διαβάστε εκείνη που απαντά στην ερώτησή σας.

Υποδιεργασίες και συνεργαζόμενες διεργασίες

Η πιο συνηθισμένη εργασία IPC είναι η απλούστερη: εκτέλεση μιας εξωτερικής εντολής και επικοινωνία μαζί της. Αυτή η σελίδα καλύπτει το φάσμα, από ένα μονόδρομο άνοιγμα σωλήνα μέχρι μια αμφίδρομη συνεργαζόμενη διεργασία - ένα παιδί στο οποίο τόσο γράφετε όσο και διαβάζετε από αυτό.

Οι μηχανισμοί ενός μονόδρομου ανοίγματος σωλήνα (`"-|"` και `"|-"`) καλύπτονται πλήρως στη σελίδα *Σωλήνες*. Αυτή η σελίδα συνεχίζει εκεί όπου σταματά εκείνη: όταν μία κατεύθυνση δεν αρκεί, και όταν χρειάζεστε ένα δικό σας χαμηλού επιπέδου `pipe` και `fork`.

Μία κατεύθυνση: το άνοιγμα σωλήνα

Για να διαβάσετε την έξοδο μιας εντολής, ή για να τροφοδοτήσετε μια εντολή με την είσοδό της, ένα άνοιγμα σωλήνα είναι το μόνο που χρειάζεστε:

```
open my $fh, "-|", "sort", "-u", "names.txt" or die "pipe: $!";
while (my $line = <$fh>) { ... }
close $fh;
```

Αυτή είναι όλη η ιστορία για τη μονόδρομη εργασία, και αφηγείται στη σελίδα *Σωλήνες* - συμπεριλαμβανομένης της μορφής λίστας έναντι συμβολοσειράς, του γιατί η μορφή λίστας παρακάμπτει το κέλυφος, και του πώς να αποκωδικοποιήσετε την κατάσταση εξόδου του παιδιού. Ξεκινήστε από εκεί αν μία κατεύθυνση αρκεί.

Και οι δύο κατευθύνσεις: IPC::Open2

Όταν χρειάζεται να γράψετε σε μια εντολή *και* να διαβάσετε την απάντησή της - ένα φίλτρο που οδηγείτε διαδραστικά, μια συνεργαζόμενη διεργασία - ένας σωλήνας δεν αρκεί. Το `IPC::Open2` ανοίγει δύο: έναν προς το `stdin` του παιδιού, έναν από το `stdout` του.

```
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);
```

Η open2 επιστρέφει το αναγνωριστικό διεργασίας του παιδιού. Η σειρά των ορισμάτων είναι η παγίδα που αξίζει να απομνημονεύσετε: **πρώτα ο περιγραφέας ανάγνωσης, δεύτερος ο περιγραφέας εγγραφής**, μετά η εντολή. `open2(OUT, IN, ...)` - ο περιγραφέας από τον οποίο διαβάζετε προηγείται του περιγραφέα στον οποίο γράφετε.

Το αδιέξοδο γύρω από το οποίο πρέπει να σχεδιάσετε

Μια συνεργαζόμενη διεργασία μπορεί να φτάσει σε αδιέξοδο, και το `IPC::Open2` δεν κάνει τίποτα για να το αποτρέψει. Το σχήμα της παγίδας:

- Γράφετε ένα μεγάλο μπλοκ στο παιδί.
- Το παιδί διαβάζει ένα μέρος, παράγει έξοδο, και αυτή η έξοδος γεμίζει τον σωλήνα πίσω προς εσάς.
- Το παιδί μπλοκάρει γράφοντας σε έναν γεμάτο σωλήνα· εσείς μπλοκάρετε γράφοντας σε ένα παιδί που έχει σταματήσει να διαβάζει. Κανείς δεν προχωρά.

Η παραπάνω συνταγή το παρακάμπτει κλείνοντας το `$to_child` πριν από την ανάγνωση - στέλνοντας ολόκληρη την είσοδο, μετά σηματοδοτώντας το τέλος αρχείου, μετά

αποστραγγίζοντας την απάντηση. Αυτό λειτουργεί όταν η είσοδος χωρά άνετα στον ενταμιευτή του σωλήνα. Για απεριόριστη ροή και στις δύο κατευθύνσεις ταυτόχρονα, χρειάζεστε I/O χωρίς μπλοκάρισμα ή έναν βρόχο `select`· το σχήμα των δύο σωλήνων από μόνο του δεν αρκεί.

Σύλληψη και του τυπικού σφάλματος: `IPC::Open3`

Το `IPC::Open2` σας συνδέει με το `stdin` και το `stdout`. Όταν χρειάζεστε επίσης το `stderr` του παιδιού σε δικό του περιγραφέα, το `IPC::Open3` προσθέτει έναν τρίτο:

```
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);
```

Δύο διαφορές από την open2 που πρέπει να σημειώσετε:

- **Η σειρά των ορισμάτων αντιστρέφεται.** Η open3 παίρνει πρώτα τον περιγραφέα εγγραφής, μετά τον αναγνώστη, μετά τον περιγραφέα σφάλματος: `open3(IN, OUT, ERR, ...)`. Αυτό είναι το αντίθετο της σειράς αναγνώστη-πρώτα της open2 - μια μακροχρόνια ασυνέπεια που δαγκώνει τον καθένα μία φορά.
- **Ο περιγραφέας σφάλματος χρειάζεται `gensym`.** Περάστε έναν φρέσκο ανώνυμο περιγραφέα από την `Symbol::gensym` για το `stderr`· ένα bareword ή μια απροσδιόριστη λεξιλογική μεταβλητή δεν θα το συλλέξει καθαρά.

Φτιάχνοντας το δικό σας: `pipe` και `fork`

Όταν η εντολή δεν είναι ένα εξωτερικό πρόγραμμα αλλά κώδικας Perl που θέλετε να εκτελέσετε σε μια ξεχωριστή διεργασία, χτίστε το κανάλι μόνοι σας. Ένα χαμηλού επιπέδου `pipe` δίνει ένα συνδεδεμένο ζεύγος αναγνώστη/εγγραφέα πριν από το fork· μετά το `fork` κάθε πλευρά κρατά το άκρο που χρειάζεται και κλείνει το άλλο:

```
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␣
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
 ↪end
    $writer->autoflush(1);
    print $writer "down-the-pipe\n";
    close $writer;
    exit 0;
}

# 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);
```

Η πειθαρχία είναι ίδια με του διακομιστή που κάνει fork στη σελίδα *Υποδοχές TCP*: **κάθε πλευρά κλείνει το άκρο που δεν χρησιμοποιεί.** Αν ο γονέας κρατήσει το άκρο εγγραφής ανοιχτό, το `close` του παιδιού δεν παράγει ποτέ τέλος αρχείου στο άκρο ανάγνωσης, και η ανάγνωση του γονέα μπλοκάρει για πάντα.

Το ρητό `fork()` δημιουργεί μια ξεχωριστή διεργασία λειτουργικού συστήματος· η αυτόματη παραλληλοποίηση της PetaPerl είναι μια δεξαμενή νημάτων εντός της διεργασίας. Ένα παιδί που δημιουργήθηκε με fork εκτελεί κανονικά τις δικές του αυτόματα παραλληλοποιημένες αριθμητικές αναγωγές μέσα του, και ο παραλληλισμός του γονέα και του παιδιού δεν παρεμβαίνουν μεταξύ τους - είναι ανεξάρτητες διεργασίες με ανεξάρτητες δεξαμενές νημάτων. Για την εκτέλεση εργασίας σε πολλούς πυρήνες CPU *μέσα* σε μία διεργασία αντί σε ξεχωριστές διεργασίες, δείτε τον οδηγό *Ταυτόχρονη εκτέλεση*.

Διασταυρούμενοι σύνδεσμοι αναφοράς

- `open` - το μονόδρομο άνοιγμα σωλήνα (" - | ", " | - ")
- `pipe` - ένα χαμηλού επιπέδου συνδεδεμένο ζεύγος περιγραφέων
- `fork` - διασπά σε δύο διεργασίες
- `waitpid` - συγκομίζει το παιδί και ανακτά την κατάστασή του
- `exec` - αντικαθιστά την εικόνα του παιδιού με μια εντολή

Επόμενο

Για δύο μη σχετιζόμενα προγράμματα που συναντιούνται σε μια διαδρομή του συστήματος αρχείων αντί μέσω κοινού γονέα, δείτε *FIFO*.

FIFO

Ένα FIFO - ένας *ονομασμένος σωλήνας* - είναι ένας σωλήνας με όνομα στο σύστημα αρχείων. Δύο προγράμματα που δεν μοιράζονται κανέναν γονέα, ξεκινημένα ανεξάρτητα από διαφορετικά κελύφη, μπορούν ωστόσο να επικοινωνήσουν: το ένα ανοίγει τη διαδρομή για εγγραφή, το άλλο την ανοίγει για ανάγνωση, και τα bytes ρέουν μεταξύ τους ακριβώς όπως μέσα από έναν ανώνυμο `pipe`. Η διαδρομή είναι ένα σημείο συνάντησης· κανένα δεδομένο δεν αποθηκεύεται ποτέ στο ίδιο το αρχείο.

Αυτό είναι το κανάλι στο οποίο καταφεύγετε όταν οι διεργασίες είναι *μη σχετιζόμενες* - όταν ένα απλό `pipe` είναι αδύνατο επειδή δεν υπήρξε ποτέ ένα κοινό `fork` από το οποίο να κληρονομηθεί ο περιγραφέας.

Δημιουργία του FIFO

Ένα FIFO δημιουργείται μία φορά, ως ειδικό αρχείο, με την `POSIX::mkfifo`. Μετά από αυτό ανοίγεται όπως κάθε άλλη διαδρομή:

```
use POSIX qw(mkfifo);

my $path = "/tmp/demo.fifo";
mkfifo($path, 0600) or die "mkfifo $path: $!";
```

Το δεύτερο όρισμα είναι ο τρόπος δικαιωμάτων, τα ίδια `0600` / `0644` που θα περνούσατε στην `chmod` - αποφασίζει ποιος μπορεί να ανοίξει το FIFO. Η κλήση αποτυγχάνει αν η διαδρομή υπάρχει ήδη, οπότε μια μακρόχρονη αρχικοποίηση τυπικά κάνει πρώτα `unlink` ένα παλιό FIFO.

Παραγωγός και καταναλωτής

Η χαρακτηριστική συμπεριφορά ενός FIFO είναι ότι **η open μπλοκάρει μέχρι να είναι παρόντα και τα δύο άκρα**. Ένας αναγνώστης που ανοίγει τη διαδρομή περιμένει μέχρι κάποιος εγγραφέας να την ανοίξει επίσης, και αντιστρόφως. Αυτή η συνάντηση είναι το χαρακτηριστικό: συγχρονίζει δύο προγράμματα που ποτέ δεν συμφώνησαν ως προς τον χρονισμό.

Αυτό το μοναδικό πρόγραμμα παρουσιάζει και τα δύο άκρα κάνοντας fork, αλλά τα δύο μισά θα λειτουργούσαν εξίσου καλά ως δύο ξεχωριστά scripts που εκτελούνται από δύο τερματικά:

```
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);
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    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
```

Οι δύο κλήσεις open είναι ο συγχρονισμός. Όποια διεργασία φτάσει πρώτη στην open της σταθμεύει εκεί μέχρι να φτάσει η άλλη· κατόπιν προχωρούν και οι δύο μαζί. Μόλις ανοίξουν, οι περιγραφείς ανάγνωσης και εγγραφής συμπεριφέρονται σαν τα δύο άκρα οποιουδήποτε σωλήνα - αναγνώσεις γραμμών, `readline`, `print` με ενταμιευτή, όλα αμετάβλητα.

Δύο πρακτικές σημειώσεις:

- **Αυτόματη εκκένωση στον εγγραφέα.** Όπως με κάθε σωλήνα και υποδοχή, μια εγγραφή με ενταμιευτή μπορεί να μην φτάσει ποτέ στον αναγνώστη μέχρι να γεμίσει ο ενταμιευτής ή να κλείσει ο περιγραφέας. Ορίστε `autoflush(1)` αν ο αναγνώστης περιμένει κάθε γραμμή τη στιγμή που παράγεται.
- **Ένα FIFO παραμένει στον δίσκο.** Η `mkfifo` αφήνει πίσω της ένα ειδικό αρχείο που επιζεί και των δύο διεργασιών. Αφαιρέστε το με `unlink` όταν τελειώσει η συνομιλία, με τον ίδιο τρόπο που θα καθαρίζατε ένα αρχείο κλειδώματος.

Διασταυρούμενοι σύνδεσμοι αναφοράς

- `open` - ανοίγει τη διαδρομή του FIFO για ανάγνωση ή εγγραφή
- `pipe` - ο συγγενής ανώνυμου σωλήνα για σχετιζόμενες διεργασίες
- `unlink` - αφαιρεί το FIFO όταν τελειώσετε
- `readline` - η συνάρτηση πίσω από το `<$fh>`

Επόμενο

Όταν δύο διεργασίες χρειάζονται μια σύνδεση αντί για μια μονόδρομη διαδρομή - αίτηση και απάντηση, και στις δύο κατευθύνσεις ταυτόχρονα - καταφύγετε σε μια υποδοχή. Δείτε *Υποδοχές UNIX-domain και UDP* και *Υποδοχές TCP*.

Υποδοχές TCP

Μια υποδοχή TCP μεταφέρει μια αξιόπιστη, διατεταγμένη ροή bytes μεταξύ δύο διεργασιών που μπορεί να βρίσκονται σε διαφορετικά μηχανήματα. Αυτή η σελίδα χτίζει τα δύο μισά μιας συνομιλίας TCP από τις χαμηλού επιπέδου ενσωματωμένες συναρτήσεις: έναν **πελάτη** που συνδέεται και ανταλλάσσει μια γραμμή, και έναν

διακομιστή που κάνει fork που αποδέχεται συνδέσεις και χειρίζεται την καθεμία στο δικό του παιδί.

Το άρθρωμα ευκολίας υψηλού επιπέδου `IO::Socket::INET` δεν είναι διαθέσιμο στην PetaPerl, οπότε κάθε συνταγή εδώ χρησιμοποιεί απευθείας το ενσωματωμένο επίπεδο `socket` με βοηθούς διευθύνσεων από το άρθρωμα `Socket`. Αυτό δεν είναι απώλεια: η ενσωματωμένη μορφή είναι αυτό που περιτυλίγει το `IO::Socket::INET`, και το να τη βλέπετε ανοιχτά καθιστά τη χειραψία ευανάγνωστη.

Οι βοηθοί διευθύνσεων

Μια διεύθυνση υποδοχής είναι ένα πακεταρισμένο C `struct sockaddr_in`. Το άρθρωμα `Socket` χτίζει και αποσυναρμολογεί αυτή τη δομή για εσάς:

```
use Socket;
my $packed = sockaddr_in($port, $ip);     # build
my ($port, $ip) = sockaddr_in($packed);   # take apart
```

Το `$ip` είναι το ίδιο μια πακεταρισμένη διεύθυνση τεσσάρων bytes. Η `inet_aton` μετατρέπει μια συμβολοσειρά τετράδας με τελείες σε τέτοια· η `inet_ntoa` την επαναφέρει. Η σταθερά `INADDR_LOOPBACK` είναι η πακεταρισμένη μορφή του `127.0.0.1`, και το `INADDR_ANY` σημαίνει «κάθε τοπική διεπαφή».

```
use Socket;
my $ip = inet_aton("127.0.0.1");
print inet_ntoa($ip), "\n";          # 127.0.0.1
```

Ένας πελάτης TCP

Ο πελάτης δημιουργεί μια υποδοχή, τη συνδέει σε μια διεύθυνση διακομιστή, και κατόπιν διαβάζει και γράφει τον περιγραφέα όπως κάθε άλλη ροή.

```
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;
```

Η μία γραμμή που πιάνει τον καθένα είναι το `$sock->autoflush(1)`. Χωρίς αυτό, το `print $sock "ping\n"` παραμένει στον ενταμιευτή εξόδου της Perl, ο διακομιστής δεν το βλέπει ποτέ, και οι δύο πλευρές μπλοκάρουν για πάντα περιμένοντας η μία την άλλη. Σε κάθε υποδοχή όπου γράφετε μια αίτηση και μετά διαβάζετε μια απάντηση, ορίστε autoflush πριν από την πρώτη εγγραφή.

Ένας διακομιστής που κάνει fork

Ένας διακομιστής δένεται σε μια θύρα, ακούει, και κατόπιν επαναλαμβάνει αποδεχόμενος συνδέσεις. Η κλασική δομή χειρίζεται κάθε πελάτη σε ένα παιδί που έχει δημιουργηθεί με fork, ώστε ένας αργός πελάτης να μην μπορεί να καθυστερήσει τον επόμενο:

```
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.
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;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    }

    # Parent: hand the connection to the child and move on.
    close $client;                  # parent does not need this␣
↪client
}
```

Τρεις λεπτομέρειες κάνουν αυτόν τον διακομιστή σωστό αντί απλώς λειτουργικό:

- **`SO_REUSEADDR` πριν από το `bind`.** Αφού ένας διακομιστής τερματίσει, ο πυρήνας κρατά τη θύρα σε `TIME_WAIT` για κάποιο διάστημα. Χωρίς αυτή την επιλογή, μια επανεκκίνηση αποτυγχάνει με «Address already in use». Ορίστε την και η επανεκκίνηση δένεται αμέσως.
- **Ο συγκομιστής `$SIG{CHLD}`.** Κάθε παιδί που δημιουργείται με fork γίνεται zombie όταν τερματίζει, μέχρι ο γονέας να καλέσει `waitpid`. Ο χειριστής συγκομίζει όλα τα ολοκληρωμένα παιδιά χωρίς μπλοκάρισμα με `WNOHANG`, ώστε ο βρόχος αποδοχής να μην σταματά ποτέ για να περιμένει.
- **Και οι δύο πλευρές κλείνουν τον περιγραφέα που δεν χρησιμοποιούν.** Το παιδί κλείνει την υποδοχή που ακούει· ο γονέας κλείνει την αποδεκτή υποδοχή πελάτη. Αν κάποια από τις δύο παραλείψει το `close` της, ο υποκείμενος περιγραφέας αρχείου παραμένει ανοιχτός στη λάθος διεργασία και η σύνδεση δεν βλέπει ποτέ EOF.

Εκτέλεση και των δύο μισών μαζί

Για μια αυτοτελή επίδειξη, κάντε fork τον διακομιστή στο παρασκήνιο και εκτελέστε τον πελάτη στον γονέα. Ο διακομιστής δένεται σε μια εφήμερη θύρα (θύρα `0`) και αναφέρει τη θύρα που ανέθεσε ο πυρήνας:

```
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;
}
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
# --- 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);
```

Η `getsockname($server)` αναφέρει την τοπική διεύθυνση στην οποία είναι δεμένη η υποδοχή. Το αποπακετάρισμά της με `sockaddr_in` ανακτά τη θύρα που επέλεξε ο πυρήνας, στην οποία στη συνέχεια συνδέεται ο πελάτης - ο τυπικός τρόπος να γράψετε ένα test που δεν ενσωματώνει σταθερά μια θύρα.

Διασταυρούμενοι σύνδεσμοι αναφοράς

Κάθε κλήση έχει μια σελίδα αναφοράς ανά ενσωματωμένη συνάρτηση με την πλήρη μορφή ορισμάτων και την τιμή επιστροφής:

- `socket` - δημιουργεί το άκρο
- `bind` - το προσαρτά σε μια τοπική διεύθυνση
- `listen` - το σημειώνει ως αποδεχόμενο συνδέσεις
- `accept` - παίρνει την επόμενη εκκρεμή σύνδεση
- `connect` - προσεγγίζει έναν διακομιστή
- `setsockopt` - ορίζει το `SO_REUSEADDR` και τα συναφή
- `getsockname` - ανακαλύπτει τη δεμένη θύρα
- `fork` και `waitpid` - δημιουργεί και συγκομίζει το παιδί ανά πελάτη
- `%SIG` - ο χειριστής συγκομιδής `CHLD`

Επόμενο

Για υποδοχές ροής μόνο τοπικές χωρίς τη στοίβα δικτύου, και για datagrams χωρίς σύνδεση, δείτε *Υποδοχές UNIX-domain και UDP*.

Υποδοχές UNIX-domain και UDP

Δεν φτάνει κάθε υποδοχή μέσα από ένα δίκτυο. Αυτή η σελίδα καλύπτει δύο ελαφρύτερους συγγενείς της υποδοχής TCP από την *προηγούμενη σελίδα*: ένα **ζεύγος υποδοχών UNIX-domain** για δύο σχετιζόμενες διεργασίες σε ένα μηχάνημα, και μια **υποδοχή UDP** για datagrams χωρίς σύνδεση.

Και τα δύο χρησιμοποιούν την ίδια οικογένεια ενσωματωμένων συναρτήσεων `socket` και τους ίδιους βοηθούς διευθύνσεων `Socket`. Τα περιτυλίγματα υψηλού επιπέδου

`IO::Socket::*` δεν είναι διαθέσιμα στην PetaPerl, οπότε οι συνταγές παραμένουν στο ενσωματωμένο επίπεδο.

Ένα ζεύγος υποδοχών για γονέα και παιδί

Η `socketpair` δημιουργεί δύο συνδεδεμένες υποδοχές σε μία κλήση, χωρίς διευθύνσεις, χωρίς `bind`, και χωρίς `connect`. Είναι το ιδανικό κανάλι μεταξύ ενός γονέα και ενός παιδιού που πρόκειται να δημιουργήσει με fork: πλήρως αμφίδρομο (και τα δύο άκρα μπορούν να διαβάζουν και να γράφουν), τοπικό, και ήδη συνδεδεμένο τη στιγμή που επιστρέφει.

```
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;
}

# 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);
```

Γιατί να καταφύγετε στην `socketpair` αντί για ένα απλό `pipe`; Ένας σωλήνας είναι μονόδρομος - ο αναγνώστης διαβάζει, ο εγγραφέας γράφει, και αυτό είναι καθορισμένο κατά τη δημιουργία. Ένα ζεύγος υποδοχών είναι αμφίδρομο: κάθε άκρο και διαβάζει και γράφει, οπότε μια ανταλλαγή αίτησης-και-απάντησης χρειάζεται μία κλήση `socketpair` αντί για δύο σωλήνες καλωδιωμένους σε αντίθετες κατευθύνσεις.

Ο κανόνας autoflush από το TCP ισχύει αμετάβλητος: ορίστε τον σε κάθε άκρο πριν από την πρώτη εγγραφή, αλλιώς μια αίτηση με ενταμιευτή δεν φτάνει ποτέ στην άλλη πλευρά και οι δύο διεργασίες μπλοκάρουν.

Μια ανταλλαγή datagram UDP

Το UDP είναι χωρίς σύνδεση. Δεν υπάρχει `connect`, ούτε `accept`, ούτε ροή - μόνο διακριτά μηνύματα, καθένα διευθυνσιοδοτημένο προς έναν προορισμό και καθένα διατηρώντας τα δικά του όρια. Ένας παραλήπτης δένεται σε μια θύρα· ένας αποστολέας του εκτοξεύει ένα datagram με `send`· ο παραλήπτης το συλλέγει με `recv`, η οποία αναφέρει επίσης ποιος το έστειλε.

```
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

waitpid($kid, 0);
```

Τι σας δίνει το UDP και τι όχι:

- **Τα όρια μηνύματος διατηρούνται.** Ένα `send` είναι ένα `recv`. Ένα `recv` επιστρέφει ακριβώς ένα datagram, ποτέ δύο συνενωμένα και ποτέ το μισό ενός. Το `65535` είναι το μέγιστο μήκος που είστε διατεθειμένοι να αποδεχτείτε· ένα συντομότερο datagram επιστρέφει το πραγματικό του μήκος.
- **Δεν υπάρχει `connect` ούτε χειραψία.** Ο αποστολέας δεν ανοίγει σύνδεση· διευθυνσιοδοτεί κάθε datagram με το τέταρτο όρισμα της `send`. Η `recv` αναφέρει τη διεύθυνση του αποστολέα ώστε μια απάντηση να μπορεί να επιστρέψει με τον ίδιο τρόπο.
- **Η παράδοση δεν είναι εγγυημένη.** Τα datagrams UDP μπορούν να απορριφθούν, να διπλασιαστούν ή να αναδιαταχθούν από το δίκτυο. Πάνω από τη διεπαφή loopback, όπως σε αυτή τη συνταγή, αυτό ουσιαστικά δεν συμβαίνει ποτέ· πάνω από ένα πραγματικό δίκτυο πρέπει να χειριστείτε μόνοι σας την απώλεια αν έχει σημασία.

Διασταυρούμενοι σύνδεσμοι αναφοράς

- `socketpair` - δύο συνδεδεμένες τοπικές υποδοχές
- `socket` - δημιουργεί ένα άκρο datagram ή ροής
- `bind` - προσαρτά τον παραλήπτη σε μια τοπική θύρα
- `send` - εκτοξεύει ένα datagram προς μια διεύθυνση
- `recv` - συλλέγει ένα datagram και τον αποστολέα του
- `getsockname` - ανακαλύπτει τη δεμένη θύρα

Επόμενο

Για κοινόχρηστη μνήμη και σηματοφόρους - κανάλια που κατέχει ο πυρήνας αντί για ροές μεταξύ διεργασιών - δείτε *IPC System V*.

IPC System V

Το IPC System V δίνει σε δύο μη σχετιζόμενες διεργασίες έναν κοινόχρηστο πόρο που κατέχει ο πυρήνας: ένα **σύνολο σηματοφόρων** για συντονισμό, ή ένα **τμήμα κοινόχρηστης μνήμης** για τη μεταβίβαση bytes χωρίς αντιγραφή μέσα από σωλήνα. Κάθε πόρος προσδιορίζεται από ένα ακέραιο κλειδί και επιζεί μέχρι μια διεργασία να τον αφαιρέσει ρητά.

Τα αρθρώματα ευκολίας `IPC::SysV`, `IPC::Semaphore` και `IPC::SharedMem` δεν είναι διαθέσιμα στην PetaPerl, οπότε οι συνταγές εδώ χρησιμοποιούν τη χαμηλού επιπέδου οικογένεια ενσωματωμένων συναρτήσεων `semget` / `shmget` με τις τιμές των σημαιών γραμμένες ως απλούς αριθμούς. Αυτές οι ενσωματωμένες συναρτήσεις είναι το επίπεδο που περιτυλίγουν τα αρθρώματα· η απευθείας χρήση τους κοστίζει μόνο μερικές ονομασμένες σταθερές που παρέχετε μόνοι σας.

Οι σταθερές σημαιών που χρειάζεστε

Μια χούφτα μικροί ακέραιοι ελέγχουν το πώς συμπεριφέρονται αυτές οι κλήσεις. Είναι οι τυπικές τιμές του Linux:

```
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
};
```

Τα χαμηλά εννέα bits της λέξης σημαιών είναι συνηθισμένα bits δικαιωμάτων, τα ίδια `0600` / `0644` που θα περνούσατε στην `chmod`. Το `IPC_CREAT | 0600` σημαίνει «δημιούργησέ το, ανάγνωση/εγγραφή για τον ιδιοκτήτη».

Μια παράδοση μέσω κοινόχρηστης μνήμης

Η κοινόχρηστη μνήμη είναι το ταχύτερο κανάλι: μια διεργασία γράφει bytes σε ένα τμήμα, μια άλλη τα διαβάζει κατευθείαν, χωρίς αντιγραφή από τον πυρήνα ενδιάμεσα. Αυτό το πρόγραμμα δημιουργεί ένα τμήμα, γράφει ένα ωφέλιμο φορτίο, κάνει fork, και το παιδί διαβάζει πίσω τα ίδια bytes:

```
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: $!";
```

Δύο σημεία αποφασίζουν αν αυτός ο κώδικας διαρρέει:

- **Οι `shmread` / `shmwrite` παίρνουν ρητό μήκος.** Δεν σταματούν σε ένα NUL ή σε έναν χαρακτήρα νέας γραμμής· τους λέτε ακριβώς πόσα bytes να μετακινήσουν, ξεκινώντας από μια μετατόπιση. Περάστε `length $payload` ώστε ο αναγνώστης και ο εγγραφέας να συμφωνούν ως προς το μέγεθος.
- **Το `IPC_RMID` δεν είναι αυτόματο.** Ένα κοινόχρηστο τμήμα παραμένει στον πυρήνα αφού κάθε διεργασία που είχε προσαρτηθεί σε αυτό έχει τερματίσει. Μπορείτε να δείτε ορφανά τμήματα με `ipcs -m` στο κέλυφος. Πάντα να συνδυάζετε ένα `shmget` με ένα αντίστοιχο `shmctl IPC_RMID`.

Ένας σηματοφόρος για συντονισμό

Ένα σύνολο σηματοφόρων είναι ένας μικρός πίνακας μετρητών που φυλά ο πυρήνας. Η κλασική χρήση είναι ένα κλείδωμα: μια διεργασία μειώνει τον μετρητή για να εισέλθει σε ένα κρίσιμο τμήμα και τον αυξάνει για να εξέλθει. Αυτή η συνταγή δημιουργεί ένα σύνολο ενός στοιχείου, το αρχικοποιεί, και το αφαιρεί:

```
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: $!";
```

Οι ίδιες οι λειτουργίες αναμονής/σηματοδότησης περνούν μέσα από την `semop`, η οποία παίρνει μια πακεταρισμένη συμβολοσειρά τριάδων `(sem_num, sem_op, sem_flg)` χτισμένη με `pack`. Η μείωση κατά ένα περιμένει μέχρι ένα διακριτικό να είναι ελεύθερο· η αύξηση κατά ένα το απελευθερώνει. Όπως με την κοινόχρηστη μνήμη, ένα σύνολο σηματοφόρων που κατέχει ο πυρήνας πρέπει να αφαιρεθεί με `IPC_RMID` αλλιώς παραμένει - ορατό υπό το `ipcs -s`.

Διασταυρούμενοι σύνδεσμοι αναφοράς

- `shmget` - δημιουργεί ή ανοίγει ένα κοινόχρηστο τμήμα
- `shmread` / `shmwrite` - μετακινούν bytes προς τα μέσα και προς τα έξω
- `shmctl` - έλεγχος, συμπεριλαμβανομένου του `IPC_RMID`
- `semget` - δημιουργεί ή ανοίγει ένα σύνολο σηματοφόρων
- `semop` - οι λειτουργίες αναμονής/σηματοδότησης
- `semctl` - ορισμός, ανάγνωση και αφαίρεση
- `pack` - χτίζει τη συμβολοσειρά λειτουργίας της `semop`

Επόμενο

Η κοινόχρηστη μνήμη και οι σηματοφόροι είναι τα βαρέα κανάλια. Για τα ελαφρύτερα, προσανατολισμένα σε ροή, δείτε *Υποδιεργασίες και συνεργαζόμενες διεργασίες* και *Υποδοχές TCP*.

- **Υποδιεργασίες και συνεργαζόμενες διεργασίες** - εκτέλεση μιας εντολής και επικοινωνία μαζί της, σε μία ή και στις δύο κατευθύνσεις, με το άνοιγμα σωλήνα και με τα `IPC::Open2` / `IPC::Open3`.

- **FIFO** - ένας ονομασμένος σωλήνας στο σύστημα αρχείων που επιτρέπει σε δύο μη σχετιζόμενα προγράμματα να συναντηθούν σε μια διαδρομή.
- **Υποδοχές TCP** - ένας πλήρης πελάτης και ένας διακομιστής echo που κάνει fork, πάνω από τη διεπαφή loopback.
- **Υποδοχές UNIX-domain και UDP** - τοπικές υποδοχές ροής μέσω της `socketpair`, και datagrams UDP χωρίς σύνδεση.
- **IPC System V** - σηματοφόροι και κοινόχρηστη μνήμη με τις χαμηλού επιπέδου ενσωματωμένες συναρτήσεις `semget` / `shmget`.

4.16.3 Τι δεν καλύπτει αυτός ο οδηγός εκμάθησης

- **Τα σήματα ως μηχανισμός συντονισμού πέρα από τα βασικά.** Η αποστολή ενός σήματος με `kill` και η σύλληψή του μέσω του `%SIG` καλύπτεται εκεί όπου τη χρειάζεται κάθε συνταγή· ο πλήρης πίνακας διάθεσης σημάτων βρίσκεται στην αναφορά `%SIG`.
- **Παραλληλισμός εντός διεργασίας.** Η εκτέλεση εργασίας σε πολλούς πυρήνες CPU μέσα σε ένα μόνο πρόγραμμα είναι αυτόματη παραλληλοποίηση, ένα διαφορετικό θέμα. Δείτε τον οδηγό *Ταυτόχρονη εκτέλεση*.
- **Αρθρώματα δικτύωσης υψηλού επιπέδου.** Αυτός ο οδηγός εκμάθησης παραμένει στο επίπεδο της ενσωματωμένης `socket` ώστε οι μηχανισμοί να είναι ορατοί. Οι συνταγές είναι το θεμέλιο πάνω στο οποίο θα χτιζόταν ένα περιτύλιγμα υψηλότερου επιπέδου.

ΚΕΦΑΛΑΙΟ 5

Αναφορά

Reference for pperl, a faithful Rust reimplementation of perl5 5.44. 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*
- *Διαχείριση locale*
- *Diagnostics*
- *Security and taint mode*

5.1.1 Ειδικές μεταβλητές

Η Perl διαθέτει ένα σταθερό σύνολο ενσωματωμένων μεταβλητών που συμπληρώνονται, διαβάζονται, ή τηρούνται από την ίδια τη γλώσσα. Μερικές κρατούν κατάσταση από την τελευταία λειτουργία (οι μεταβλητές αντιστοίχισης regex, `$!`, `$@`)· μερικές ρυθμίζουν τη συμπεριφορά ενσωματωμένων (`$/`, `$\`, `$,`)· μερικές εκθέτουν τον τρέχοντα διερμηνέα και το περιβάλλον του (`$^V`, `$^O`, `%ENV`, `@INC`)· μερικές δεν είναι πραγματικά μεταβλητές αλλά hooks σε υπηρεσίες χρόνου εκτέλεσης (`%SIG`).

PetaPerl implements every special variable that Perl 5.44 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.

Αυτή η αναφορά είναι χωρισμένη σε μία σελίδα ανά οικογένεια. Μεταβλητές που μοιράζονται πλαίσιο (σχεδόν ποτέ δεν θέτετε τη `$,` χωρίς να σκεφτείτε τη `$\` και τη `$|`) βρίσκονται στην ίδια σελίδα· μεταβλητές που δεν μοιράζονται (ID διεργασιών και συλλήψεις regex) δεν βρίσκονται.

Επιλέξτε οικογένεια

Προεπιλεγμένες μεταβλητές

Δύο μεταβλητές κρατούν τους σιωπηρούς τελεστέους της Perl. Η `$_` είναι το προεπιλεγμένο βαθμωτό που αμέτρητες ενσωματωμένες διαβάζουν όταν κληθούν χωρίς όρισμα. Η `@_` είναι ο πίνακας ορισμάτων κάθε υπορουτίνας. Μαζί ευθύνονται για τις περισσότερες εκπλήξεις «από πού ήρθε αυτή η τιμή;» σε κώδικα Perl.

`$_` - το προεπιλεγμένο βαθμωτό

Η `$_` είναι το *θέμα*: η τιμή για την οποία γίνεται λόγος αυτή τη στιγμή. Ένα τεράστιο πλήθος ενσωματωμένων της Perl τη συμβουλεύεται όταν κληθεί χωρίς ρητό όρισμα:

```
for my $line (<$fh>) {
    chomp;                      # chomp $line  - but $line was␣
↪aliased to $_
    print if /\bERROR\b/;       # print $_ if $_ =~ /\bERROR\b/
}
```

Η πλήρης λίστα ενσωματωμένων που χρησιμοποιούν εξ ορισμού τη `$_` περιλαμβάνει `chomp`, `chop`, `lc`, `lcfirst`, `length`, `pos`, `print`, `printf`, `quotemeta`, `say`, `split` (προεπιλεγμένο μοτίβο), `study`, `uc`, `ucfirst`, `unlink`, τους τελεστές `m//` και `s///`, τον τελεστή `tr///`, και τους τελεστές ελέγχου αρχείου `<>`/`-X`. Οι σελίδες λεπτομερειών τους κατονομάζουν τη `$_` ως προεπιλογή· αυτή η σελίδα είναι η κανονική περιγραφή.

Ψευδωνυμοποίηση μέσα σε `for` και `while`

Ο βρόχος `for` (και `foreach`) **ψευδωνυμεί** τη `$_` σε κάθε στοιχείο της λίστας, διαδοχικά:

```
my @items = (1, 2, 3);
for (@items) {
    $_ *= 2;                    # mutates @items in place
}
# @items is now (2, 4, 6)
```

Η τροποποίηση της `$_` μέσα στον βρόχο τροποποιεί το ίδιο το στοιχείο λίστας. Αυτή είναι πανομοιότυπη συμπεριφορά με τις `map`, `grep`, `any`, `all` - και τις `first` και `reduce` από τη `List::Util`.

Η `while (<$fh>)` **δεν** ψευδωνυμεί - διαβάζει κάθε γραμμή στη `$_` ως νέο αντίγραφο. Δεν μπορείτε να γράψετε πίσω στο αρχείο αναθέτοντας στη `$_` μέσα σε βρόχο `while (<$fh>)`.

Τοπικοποίηση $_

Μια υπορουτίνα που θέλει τη δική της $_ χωρίς να διαταράξει αυτή του καλούντος πρέπει να την κάνει `local`:

```
sub categorise {
    local $_ = shift;            # decouple from caller's $_
    return 'small'  if /^\d{1,3}$/;
    return 'medium' if /^\d{4,6}$/;
    return 'large';
}

my @sizes;
for ('5', '12345', '999999999') {
    push @sizes, categorise($_); # caller's $_ stays the loop alias
}
```

Χωρίς τη `local`, η κλήση `categorise()` με `shift` θα λειτουργούσε - αλλά κάθε =~, `/.../`, ή chomp μέσα σε αυτήν θα διάβαζε ή θα έγραφε σιωπηρά το ψευδώνυμο βρόχου, σπάζοντας την επανάληψη του καλούντος.

Η $_ ως lvalue

Η $_ είναι μια πραγματική βαθμωτή μεταβλητή· μπορείτε να αναθέσετε σε αυτήν, να πάρετε αναφορά σε αυτήν, και να τη μεταβιβάσετε ως 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
```

Προσέξτε ότι το δεύτερο `$$ref` θα γράψει σε **ό,τι είναι η $_ αυτή τη στιγμή** - που μέσα σε βρόχο `for` είναι το τρέχον στοιχείο λίστας. Αναφορές στη $_ που λήφθηκαν εκτός βρόχου και χρησιμοποιούνται μέσα σε αυτόν σπάνια κάνουν αυτό που εννοούσε ο συγγραφέας.

Λεξιλογική $_

Οι `our $_` και `my $_` ήταν πειραματικά χαρακτηριστικά που αφαιρέθηκαν. Η $_ είναι πάντα η καθολική πακέτου· χρησιμοποιήστε `local $_` για απομόνωση.

@_ - ορίσματα υπορουτίνας

Μέσα σε μια υπορουτίνα, η @_ είναι ο πίνακας ορισμάτων που μεταβιβάστηκαν σε αυτή την κλήση. Κάθε στοιχείο είναι **ψευδώνυμο** προς την έκφραση του καλούντος - η @_ δεν είναι αντίγραφο:

```
sub double {
    $_[0] *= 2;                 # mutates the caller's variable
}

my $x = 5;
double($x);
# $x is now 10
```

Γι” αυτό οι περισσότερες subs ξεκινούν με `my (...) = @_;` - η αποσυσκευασία δημιουργεί αντίγραφα, μετά τα οποία η ρουτίνα δεν μπορεί κατά λάθος να τροποποιήσει την κατάσταση του καλούντος:

```
sub greet {
    my ($name, $greeting) = @_;  # decoupled from caller
    $greeting //= 'Hello';
    return "$greeting, $name!";
}
```

Η ψευδωνυμοποίηση είναι περιστασιακά χρήσιμη - είναι ο τρόπος με τον οποίο η `chomp(@lines)` τροποποιεί κάθε στοιχείο του πίνακα του καλούντος - αλλά σε καθημερινό κώδικα το ιδίωμα αποσυσκευασμένου-αντιγράφου είναι ο κανόνας.

`wantarray` και το περιβάλλον κλήσης

Μια sub που θέλει να συμπεριφέρεται διαφορετικά ανάλογα με το αν κλήθηκε σε βαθμωτό, λίστα, ή κενό περιβάλλον εξετάζει τη `wantarray`:

```
sub items {
    return wantarray ? (1, 2, 3) : 3;
}

my @list = items();             # (1, 2, 3)
my $cnt  = items();             # 3
items();                         # void - return value discarded
```

Η `@_` και η `wantarray` είναι τα δύο κομμάτια της εικόνας σύμβασης κλήσης. Η `@_` λέει *τι* μεταβιβάστηκε· η `wantarray` λέει *τι αναμένεται πίσω*.

`@_` μετά από `shift`, `pop`, `unshift`, `push`

Η τροποποίηση της ίδιας της `@_` (όχι των στοιχείων της) δεν έχει αποτέλεσμα ψευδωνυμοποίησης στον καλούντα - είναι μια κανονική τοπική μεταβλητή πίνακα. Η `shift @_` αφαιρεί το πρώτο όρισμα, αφήνοντας τα υπόλοιπα:

```
sub method_call {
    my $self = shift;           # standard OO pattern
    my %args = @_;              # rest of args as a hash
    ...
}
```

Η `shift` χωρίς ορίσματα μέσα σε υπορουτίνα χρησιμοποιεί εξ ορισμού τη `@_`, γι" αυτό βλέπετε `my $self = shift;` παντού.

`$_ = shift` - περιμένετε, τι;

Ένα συνηθισμένο παραπλανητικό ιδίωμα:

```
sub trim {
    $_ = shift;                 # - set caller's $_ - DO NOT DO
↪THIS
    s/^\s+//;
    s/\s+$//;
    return $_;
}
```

Αυτό το `$_ = shift` γράφει στην καθολική πακέτου `$_`, η οποία είναι *το θέμα βρόχου του καλούντος* αν ο καλών βρίσκεται μέσα σε `for`. Η λύση είναι η `local`, όπως παραπάνω:

```
sub trim {
    local $_ = shift;
    s/^\s+//;
    s/\s+$//;
    return $_;
}
```

Δείτε επίσης

- `map`, `grep` - οι πιο συνηθισμένοι καταναλωτές ψευδωνυμοποίησης `$_`.
- `shift`, `pop` - χρησιμοποιούν εξ ορισμού τη `@_` μέσα σε υπορουτίνα.
- `wantarray` - η σύντροφος της `@_` στην εικόνα σύμβασης κλήσης.
- `local` - ο ασφαλής τρόπος ρύθμισης της `$_` μέσα σε sub στην οποία ο καλών μπορεί να βρίσκεται σε επανάληψη.
- *Σύνδεση regex* - η =~ είναι αυτό που χρησιμοποιείτε όταν δεν θέλετε τη σιωπηρή `$_` ως στόχο.

Ταυτότητα διεργασίας

Έξι μεταβλητές εκθέτουν τι θεωρεί το λειτουργικό σύστημα γι" αυτή τη διεργασία: το όνομά της, το PID της, και τα πραγματικά-και-effective IDs χρήστη και ομάδας. Είναι πιο χρήσιμες σε σενάρια που χρειάζεται να δρουν βάσει *ποιος τα εκτελεί*, να απορρίπτουν προνόμια μετά από προνομιακή λειτουργία, ή να διακρίνουν γονέα από παιδί μετά από `fork`.

Μεταβλητή	Τι κρατά	Τροποποιήσιμο
`$0`	Όνομα προγράμματος (και εμφάνιση `ps`)	ναι
`$$`	PID τρέχουσας διεργασίας	όχι (μόνο ανάγνωση)
`$<`	Πραγματικό UID	ναι (προνομιακό)
`$>`	Effective UID	ναι (προνομιακό)
`$(`	Πραγματικό GID (και συμπληρωματικά)	ναι (προνομιακό)
`$)`	Effective GID (και συμπληρωματικά)	ναι (προνομιακό)

`$0` - όνομα προγράμματος

Το όνομα του προγράμματος που εκτελείται, όπως μεταβιβάστηκε στο `argv[0]`:

```
print "Started by: $0\n";        # e.g. "/usr/local/bin/myscript.pl"
```

Η ανάθεση στη `$0` αλλάζει πώς εμφανίζεται η διεργασία στην έξοδο `ps` και παρόμοιες λίστες - βολικό για σενάρια μεγάλης διάρκειας που κάνουν fork και θέλουν κάθε παιδί να είναι αναγνωρίσιμο:

```
$0 = "myserver: worker pid $$";   # appears in `ps` as that string
```

Η αλλαγή γράφει στην περιοχή μνήμης διεργασίας που διαβάζει η `ps` σε Linux. Παλαιότερες παραλλαγές Unix και τα Windows έχουν διαφορετικούς κανόνες· η PetaPerl ακολουθεί ακριβώς τη συμπεριφορά της perl5. Η ανάγνωση της `$0` επιστρέφει πάντα την τρέχουσα τιμή, συμπεριλαμβανομένων τυχόν τροποποιήσεων.

Η `$0` **δεν** είναι το όνομα προγράμματος όπως το βλέπει *το σενάριο* - για αυτό, δείτε `@ARGV` (που έχει τη `$0` ως συνοδό στη θέση `[-1]` του αρχικού `argv`, όχι ως `@ARGV[0]`).

`$$` - αναγνωριστικό διεργασίας

Μόνο ανάγνωση. Το PID της τρέχουσας διεργασίας.

```
print "Running as PID $$\n";

open my $log, '>>', "/var/log/myapp.log" or die $!;
print $log "[$$] starting...\n";
```

Μετά τη `fork`, το παιδί βλέπει διαφορετικό `$$` από τον γονέα - αυτός είναι ο τυπικός τρόπος να ξέρετε σε ποια πλευρά της fork βρίσκεστε:

```
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;
```

Το ψευδώνυμο English `$PROCESS_ID` (και το πλεονασματικό `$PID`) λειτουργούν ίδια με τη `$$`.

Για να πάρετε το PID του *γονέα*, χρησιμοποιήστε τη `getppid`.

Πραγματικά έναντι effective IDs - τι σημαίνουν

Κάθε διεργασία Unix έχει *δύο* IDs χρήστη και *δύο* IDs ομάδας:

- Το **πραγματικό** UID/GID είναι ποιος είστε - συνήθως ο χρήστης που εκκίνησε το πρόγραμμα.
- Το **effective** UID/GID είναι αυτό που ελέγχει ο πυρήνας για αποφάσεις δικαιωμάτων - τυπικά ίδιο με το πραγματικό, αλλά διαφορετικό για προγράμματα *setuid* και *setgid*.

Όταν εκτελείτε ένα εκτελέσιμο setuid-root, ο πυρήνας θέτει το effective UID σε root (ο ιδιοκτήτης του αρχείου) αλλά αφήνει το πραγματικό UID ως δικό σας. Το πρόγραμμα μπορεί στη συνέχεια να ενεργεί ως root για την προνομιακή λειτουργία, και είτε να *απορρίψει προνόμια* θέτοντας το effective πίσω στο πραγματικό, είτε να *ελέγξει ποιος πραγματικά το εκκίνησε* διαβάζοντας το πραγματικό UID.

`$<` - πραγματικό UID

```
print "Invoked by user with UID $<\n";
```

Η `$<` είναι το πραγματικό UID. Σχεδόν ποτέ δεν αλλάζει κατά τη διάρκεια ζωής μιας διεργασίας· οι περιπτώσεις που μπορεί είναι πολύ περιορισμένες (μόνο ο `root` μπορεί να το αλλάξει, και μόνο μέσω συγκεκριμένων κλήσεων συστήματος).

Μνημονικό: *το UID **από** όπου ήρθατε*.

`$>` - effective UID

```
print "Privileges currently effective: UID $>\n";
```

Η `$>` είναι το effective UID - αυτό που μετράει για ελέγχους δικαιωμάτων. Σε πρόγραμμα setuid, αρχικά είναι το UID του ιδιοκτήτη αρχείου· σε κανονικό πρόγραμμα, ισούται με τη `$<`.

Μνημονικό: *το UID **προς** το οποίο πήγατε*.

Το κλασικό μοτίβο setuid

Ένα βοηθητικό setuid-root που χρειάζεται να κάνει *μία* προνομιακή λειτουργία και μετά να απορρίψει προνόμια ώστε ο υπόλοιπος κώδικας να τρέξει ως ο πραγματικός χρήστης:

```
#!/usr/bin/perl
use strict;
use warnings;

# Privileged work first - we still have effective root.
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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);
```

Η μόνιμη απόρριψη προνομίων **πρέπει** να θέσει το effective πριν το πραγματικό (ή και τα δύο ταυτόχρονα μέσω `POSIX::setuid()`)· στα περισσότερα συστήματα αν θέσετε πρώτα το πραγματικό θα κλειδωθείτε εκτός της επαναφοράς του effective. Ελέγξτε τη `$!` μετά από κάθε ανάθεση - ο πυρήνας απορρίπτει μη εξουσιοδοτημένες αλλαγές:

```
$> = $<;
die "could not drop privileges: $!" if $> != $<;
```

Εναλλαγή πραγματικού και effective

Όταν θέλετε να *αλλάξετε προσωρινά* το effective ID για μία μόνο λειτουργία και μετά να το επαναφέρετε:

```
my $saved_uid = $>;
$> = $<;                        # become real user
my $ok = open(my $fh, '<', $user_path);
$> = $saved_uid;                # back to effective
$ok or die "open $user_path: $!";
```

Αυτό λειτουργεί μόνο σε συστήματα που υποστηρίζουν `setreuid()` - τα περισσότερα σύγχρονα Unix, συμπεριλαμβανομένου του Linux. Η `POSIX::seteuid()` από το άρθρωμα `POSIX` είναι η ρητή και φορητή γραφή.

`$(` - πραγματικό GID (με συμπληρωματικές ομάδες)

Η `$(` επιστρέφει το πραγματικό GID ακολουθούμενο από κάθε *συμπληρωματική ομάδα* στην οποία ανήκει η διεργασία, όλα διαχωρισμένα με κενά:

```
print "Real GID list: $(\n";     # e.g. "1000 1000 4 27 100 999"
                                  #       primary + supplementaries
my @groups = split ' ', $(;
my $primary = $groups[0];
```

Ο πρώτος αριθμός είναι το κύριο GID· τα υπόλοιπα είναι συμπληρωματικές ομάδες (από `/etc/group`). Η ανάθεση στη `$(` θέτει το GID· η σύνταξη δέχεται έναν μόνο αριθμό:

```
$( = 100;                          # become primary GID 100
```

$) - effective GID (με συμπληρωματικές ομάδες)

Ίδια μορφή με τη $(, αλλά για το effective GID:

```
print "Effective GID list: $)\n";

# Drop both real and effective GID to a specific gid:
$( = $) = "$gid $gid";
```

Οι έλεγχοι δικαιωμάτων χρησιμοποιούν το effective GID και τις συμπληρωματικές· η λίστα πραγματικού GID είναι ενημερωτική.

Πότε τα χρησιμοποιείτε πραγματικά

Ο καθημερινός κώδικας Perl δεν αγγίζει UIDs και GIDs. Οι ρεαλιστικές περιπτώσεις χρήσης είναι:

1. **Σενάρια setuid** που χρειάζεται να απορρίψουν προνόμια. Τυπικό ιδίωμα που φαίνεται παραπάνω.
2. **Εκκίνηση δαίμονα** που τρέχει ως root για δέσμευση προνομιακής θύρας, και μετά απορρίπτει προνόμια πριν εξυπηρετήσει αιτήσεις:

   ```
   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;
   $> = $u;
   $< = $u;
   ```

3. **Έλεγχος προνομίων** σε σενάριο που πρέπει να αρνείται να τρέξει ως root: `die "do not run me as root\n" if $< == 0`.

Αλληλεπίδραση με tainted-mode

Υπό `-T`, η Perl παρατηρεί όταν τα πραγματικά και effective IDs διαφέρουν (δηλ. το σενάριο είναι setuid) και ενεργοποιεί επιπρόσθετους ελέγχους: ορισμένες λειτουργίες αρνούνται tainted ορίσματα, και η διαδρομή αναζήτησης για `system()` και `exec()` περιορίζεται. Η σχετικότητα εδώ είναι ότι αν η `-T` είναι ενεργή αποφασίζεται βάσει $< έναντι $> κατά την εκκίνηση.

Δείτε επίσης

- `getppid` - το PID του *γονέα*, σε ζεύγος με τη `$$`.
- `fork` - μετά την οποία η `$$` διαφέρει σε γονέα και παιδί.
- `getpwuid`, `getgrgid` - μετατρέπουν τα αριθμητικά IDs σε ονόματα χρηστών και ομάδων.
- `POSIX::setuid` - ο φορητός, ρητός τρόπος αλλαγής UIDs μαζί.
- *Μεταβλητές σφάλματος* - η `$!` μετά από ανάθεση `$<` / `$>` σας λέει γιατί ο πυρήνας αρνήθηκε.

Κατάσταση σφάλματος

Τέσσερις μεταβλητές αναφέρουν σφάλματα από τέσσερα διαφορετικά στρώματα. Δεν είναι εναλλάξιμες και συμπληρώνονται από εντελώς διαφορετικούς μηχανισμούς· η κακή χρήση μίας αντί άλλης είναι η πιο συνηθισμένη αιτία σφαλμάτων «γιατί δεν λειτουργεί ο χειρισμός σφαλμάτων μου» σε κώδικα Perl.

Μεταβλ	Πηγή	Διάρκεια ζωής
`$!`	`errno` βιβλιοθήκης C από την τελευταία κλήση συστήματος	Τίθεται σε αποτυχία κλήσης συστήματος· χωρίς νόημα διαφορετικά
`$@`	Η εξαίρεση που συλλήφθηκε από την τελευταία `eval`	Τίθεται κατά την έξοδο `eval` (κενή συμβολοσειρά σε επιτυχία)
`$?`	Κατάσταση εξόδου της τελευταίας διεργασίας-παιδιού	Τίθεται από κάθε `wait`/κλείσιμο σωλήνα/`system`
`$^E`	Πληροφορίες σφάλματος εκτεταμένες από το OS	Ίδια με τη `$!` σε Unix· πλουσιότερη σε VMS/Win32

Το καλύτερο πράγμα που μπορείτε να κάνετε για τον χειρισμό σφαλμάτων είναι να εσωτερικεύσετε *ποια μεταβλητή απαντάει σε ποια ερώτηση*:

- «Απέτυχε η open/read/connect/κλήση συστήματος μου;» → ελέγξτε πρώτα την τιμή επιστροφής, μετά τη `$!`.
- «Έπιασε το μπλοκ `eval` μου μια εξαίρεση;» → ελέγξτε τη `$@`.
- «Ποιος ήταν ο κωδικός εξόδου του προγράμματος που μόλις έτρεξα;» → αποκωδικοποιήστε τη `$?`.
- «Βρίσκομαι σε Windows ή VMS και χρειάζομαι κείμενο σφάλματος ειδικό του OS;» → δείτε επίσης τη `$^E`.

`$!` - το σφάλμα συστήματος

Η `$!` είναι η διεπαφή της Perl με το `errno` της βιβλιοθήκης C. Έχει *διπλή φύση*: διαβάστε τη σε αριθμητικό περιβάλλον για να πάρετε τον ακέραιο `errno`, σε περιβάλλον συμβολοσειράς για να πάρετε την αντίστοιχη συμβολοσειρά σφάλματος:

```
open(my $fh, '<', '/no/such/file') or do {
    print  "errno = ", $! + 0, "\n";        # numeric: 2
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    print  "errstr = $!\n";                  # string: "No such file␣
↪or directory"
};
```

Κρίσιμο: η `$!` **έχει νόημα μόνο αμέσως μετά από αποτυχία**. Το `errno` της C δεν καθαρίζεται σε επιτυχία, οπότε μεταγενέστερη ανάγνωση μπορεί να δει παλιά τιμή από κάποια προηγούμενη κλήση συστήματος. Η αξιόπιστη μορφή:

```
if (open my $fh, '<', $path) {
    # success - $! is meaningless here
    process($fh);
} else {
    # ONLY here is $! meaningful
    die "open $path: $!";
}
```

Η εγγραφή αποτυχίας στην ίδια γραμμή με τη λειτουργία που την προκάλεσε είναι η ασφαλέστερη γραφή - δεν αφήνει ευκαιρία σε ενδιάμεση λειτουργία Perl να αλλοιώσει το `errno`.

`%!` - συμβολικοί έλεγχοι errno

Η `%!` είναι ένα hash του οποίου τα κλειδιά είναι ονόματα συμβόλων errno όπως `ENOENT`, `EACCES`, `EAGAIN`. Ένα κλειδί ελέγχεται ως αληθές ακριβώς όταν η `$!` ισούται τη δεδομένη στιγμή με αυτό το 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
}
```

Αυτό είναι φορητό - οι αριθμητικές τιμές των συμβόλων errno διαφέρουν μεταξύ λειτουργικών συστημάτων, αλλά τα κλειδιά `%!` είναι πάντα τα ονόματα συμβόλων. Στο παρασκήνιο, η `%!` παρέχεται από το άρθρωμα `Errno`, το οποίο φορτώνεται κατ" απαίτηση την πρώτη φορά που χρησιμοποιείτε το hash.

Ανάθεση στη `$!`

Η εγγραφή στη `$!` θέτει το `errno` στη βιβλιοθήκη C - χρήσιμο για κατασκευή μηνύματος σφάλματος που θα δει ο κώδικας παρακάτω:

```
$! = 13;                            # EACCES
print "$!\n";                       # "Permission denied"
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
# Slightly more idiomatic - set errno by symbol:
use Errno qw(:POSIX);
$! = EACCES;
die "synthetic permission failure: $!";
```

$@ - η μεταβλητή εξαίρεσης

Τίθεται από την `eval` όταν το μπλοκ (ή η συμβολοσειρά) που εκτέλεσε εκτόξευσε εξαίρεση. Περιέχει είτε την τιμή που μεταβιβάστηκε στη `die` (που συνήθως είναι συμβολοσειρά, αλλά μπορεί να είναι οποιαδήποτε αναφορά) είτε το μήνυμα σφάλματος ανάλυσης/χρόνου εκτέλεσης από τον διερμηνέα.

```
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;
}
```

Η επιτυχής αποτίμηση θέτει τη `$@` σε κενή συμβολοσειρά `''`. Ο έλεγχος `if ($@)` είναι επομένως αξιόπιστος: η κενή συμβολοσειρά είναι ψευδής, οποιαδήποτε μη κενή τιμή είναι αληθής.

Το κλασικό σφάλμα αλλοίωσης $@

Υπάρχει κίνδυνος εδώ. Οτιδήποτε εκτελείται *μεταξύ* της εξόδου `eval` και του ελέγχου `if ($@)` μπορεί να αλλοιώσει τη `$@`. Ο πιο συνηθισμένος ένοχος είναι η μέθοδος `DESTROY` ενός αντικειμένου - όταν τα αντικείμενα βγαίνουν εκτός εμβέλειας στο κλείσιμο του μπλοκ `eval`, η `DESTROY` τους εκτελείται, και αν η `DESTROY` κάνει η ίδια κάποιο `eval`, η εξαίρεσή σας χάνεται:

```
eval {
    my $obj = MyClass->new;
    $obj->might_die;
    1;
};
# If MyClass::DESTROY does an eval, $@ is now empty.
warn "got: $@";                 # may print nothing!
```

Η ιστορικά συνιστώμενη λύση είναι να αποτυπώσετε τη `$@` αμέσως, ή να χρησιμοποιήσετε το άρθρωμα `Try::Tiny`, που χειρίζεται αυτό και πολλές σχετικές παγίδες:

```
my $err;
{
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    local $@;                    # save and isolate
    eval {
        risky_op();
        1;
    } or $err = $@ || 'unknown error';
}
if ($err) {
    warn "got: $err";
}
```

Το εγγενές χαρακτηριστικό `try/catch` της Perl 5.34+ είναι η καθαρότερη σύγχρονη γραφή - δεν χρησιμοποιεί καθόλου τη `$@` και δεν επηρεάζεται από το πρόβλημα καταστροφέα:

```
use feature 'try';
try {
    risky_op();
} catch ($err) {
    warn "got: $err";
}
```

Η PetaPerl υποστηρίζει εγγενώς τα `try/catch`. Ο νέος κώδικας πρέπει να τα προτιμά.

eval και $!

Η `eval` **δεν** διατηρεί τη `$!`. Αν το μπλοκ `eval` έκανε μια κλήση συστήματος που απέτυχε και μετά `die`, η `$!` είναι ό,τι έθεσε η *τελευταία* λειτουργία - που μέσα σε καταστροφέα ή χειριστή `local $SIG{__DIE__}` μπορεί να είναι οτιδήποτε. Αν το μήνυμα σφάλματός σας χρειάζεται τη `$!`, αποτυπώστε τη μέσα στην `eval`:

```
eval {
    open(my $fh, '<', $path) or die "open $path: $!\n";
    ...
};
```

Η `\n` στο τέλος είναι σύμβαση - καταστέλλει το επίθεμα «at line N» που η Perl προσαρτά στα μηνύματα die, κάτι που είναι σωστό για σφάλματα προσανατολισμένα στον χρήστη.

$? - το σφάλμα παιδιού

Μετά από `system`, `waitpid`, `wait`, εντολή με backticks, ή κλείσιμο σωλήνα, η `$?` περιέχει τη 16-bit κατάσταση `wait()` της διεργασίας-παιδιού. **Δεν** είναι απλώς ο κωδικός εξόδου:

```
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,␣
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
↪or 0
my $coredump  = $status & 0x80;      # set if the child dumped core

if ($status == -1) {
    die "system: $!";                # could not even fork/exec
} elsif ($signal) {
    die "$cmd died with signal $signal";
} elsif ($exit_code) {
    die "$cmd failed (exit $exit_code)";
}
```

Η ολίσθηση >> 8 είναι η πιο αναφερόμενη παγίδα στην Perl. Μόνο το άνω byte είναι ο κωδικός εξόδου· το κάτω byte περιέχει τις σημαίες σήματος-και-core-dump. Αν ξεχάσετε την ολίσθηση παράγονται τιμές πολλαπλασιασμένες με 256.

Η σύγχρονη Perl παρέχει τις μακροεντολές `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG`, `WIFEXITED` του αρθρώματος `POSIX`, που είναι οι φορητοί ονομαστικοί accessors:

```
use POSIX ':sys_wait_h';
if (WIFEXITED($?))   { my $code = WEXITSTATUS($?); ... }
if (WIFSIGNALED($?)) { my $sig  = WTERMSIG($?);    ... }
```

Η `$?` **δεν** διατηρείται μεταξύ άλλων λειτουργιών. Αν χρειάζεστε την τιμή, αποτυπώστε τη σε τοπική μεταβλητή στην επόμενη γραμμή - αλλιώς τα επόμενα backticks ή `system` θα την αντικαταστήσουν.

Η $? σε μπλοκ END

Μέσα σε μπλοκ `END`, η `$?` περιέχει τον σκοπούμενο κωδικό εξόδου. Η ανάθεση σε αυτήν υπερισχύει της κατάστασης εξόδου του σεναρίου:

```
END {
    $? = 0 if $? == 255 && $shutdown_was_clean;
}
```

$^E - το εκτεταμένο σφάλμα OS

Σε Unix, η `$^E` είναι ακριβώς ίδια με τη `$!`. Σε Windows, περιέχει την τιμή `GetLastError()`, η οποία είναι μερικές φορές πιο ενημερωτική από το `errno` (στο οποίο τα Win32 αντιστοιχίζουν ένα αδρό υποσύνολο). Σε VMS, είναι ο εγγενής κωδικός κατάστασης VMS.

Η PetaPerl είναι μόνο για Linux. Η `$^E` και η `$!` είναι εναλλάξιμες εδώ· η μεταβλητή υπάρχει για φορητό κώδικα που μπορεί να τρέχει και σε μη-Unix perl.

Συνδυάζοντάς τα - ένα αναλυτικό παράδειγμα

```
sub copy_file {
    my ($src, $dst) = @_;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    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
        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;
}
```

Τρεις μεταβλητές σφάλματος σε πέντε παραγράφους. Η `$!` αναφέρει κάθε αποτυχία κλήσης συστήματος στη γραμμή που την προκάλεσε. Η `$@` συλλέγει κάθε εξαίρεση που πέρασε το όριο `eval`. Η `$?` δεν βρίσκεται σε αυτό το παράδειγμα γιατί δεν υπάρχει διεργασία-παιδί - αλλά αν η συνάρτηση εκτελούσε `cp`, θα αποκωδικοποιούσατε τη `$?` με την ίδια μορφή όπως στη σελίδα `system`.

Δείτε επίσης

- `die` - συμπληρώνει τη `$@`. Ο συνηθισμένος τρόπος για να εγείρετε εξαίρεση από τον δικό σας κώδικα.
- `eval` - πιάνει εξαιρέσεις· θέτει τη `$@`.
- `system`, `waitpid`, `wait` - συμπληρώνουν τη `$?`.
- `try/catch` - η σύγχρονη εναλλακτική στο `eval`/`$@` που παρακάμπτει τον κίνδυνο αλλοίωσης.
- `Errno` - η πηγή των κλειδιών `%!`· παρέχει ονομαστικές σταθερές errno.
- `Try::Tiny` - ανθεκτικό περιτύλιγμα `eval` που διατηρεί σωστά τη `$@`.
- `%SIG` - οι `$SIG{__DIE__}` και `$SIG{__WARN__}` είναι τα hooks που ενεργοποιούνται όταν εκτελούνται η `die` και η `warn`· μπορούν να ξαναγράψουν τη `$@`.
- *Λογικοί τελεστές* - το ιδίωμα `or die` που κινεί σχεδόν κάθε αναφορά `$!`.

Κατάσταση filehandle και εκτύπωσης

Επτά μεταβλητές που κινούν το I/O της Perl. Είναι *καθολικές* - η ρύθμιση της `$/` σε μια ρουτίνα αλλάζει πώς κάθε άλλη ρουτίνα στην ίδια διεργασία διαβάζει από filehandle. Η πειθαρχία που αποτρέπει αυτό να γίνει καταστροφή είναι η `local`: κάθε αλλαγή σε μεταβλητή αυτής της σελίδας πρέπει να βρίσκεται μέσα σε μπλοκ `local` εκτός αν το πρόγραμμα πραγματικά θέλει η νέα τιμή να είναι μόνιμη.

Μεταβλr	Διαβάζεται ως	Προεπιλογή	Χρησιμοποιείται από
`$/`	διαχωριστής εγγραφών εισόδου	`"\n"`	`<$fh>`, `readline`
`$\`	διαχωριστής εγγραφών εξόδου	`undef`	`print`, `printf`, `say` (υποχρεωτικό `\n`)
`$,`	διαχωριστής πεδίων εξόδου	`undef`	`print`, `printf` (μεταξύ ορισμάτων)
`$"`	διαχωριστής λίστας	`" "`	παρεμβολή πίνακα `"@arr"`
`` `$ ``	`` ` ``	αυτόματη εκκένωση εξόδου	`0`
`$.`	αριθμός τρέχουσας γραμμής	`0`	τελευταίο filehandle που διαβάστηκε
`$;`	διαχωριστής δεικτών	`"\034"`	εξομοίωση πολλαπλών διαστάσεων `$h{$x, $y}`

`$/` - ο διαχωριστής εγγραφών εισόδου

Ελέγχει τι αντιμετωπίζουν η `<$fh>` και η `readline` ως τέλος μιας εγγραφής. Η προεπιλογή `"\n"` σημαίνει ανάγνωση *γραμμή-γραμμή*· η ρύθμιση σε άλλες τιμές παράγει άλλες λειτουργίες ανάγνωσης.

Λειτουργία γραμμής (η προεπιλογή)

```
while (my $line = <$fh>) {
    chomp $line;                # strips the terminator $/
    process($line);
}
```

Η `chomp` αφαιρεί ό,τι κρατάει τη δεδομένη στιγμή η `$/` - όταν αλλάξετε τη `$/`, η `chomp` την ακολουθεί.

Λειτουργία slurp

Η `undef $/` κάνει τη `<>` να διαβάσει ολόκληρο το αρχείο μονομιάς:

```
my $whole_file = do {
    local $/;
    <$fh>;
};
```

Η `local` είναι ουσιαστική. Χωρίς αυτή, κάθε άλλο κομμάτι κώδικα που στη συνέχεια διαβάζει από filehandle στην ίδια διεργασία θα κάνει σιωπηρά slurp.

Λειτουργία παραγράφου

Η `local $/ = "";` διαβάζει παράγραφο-παράγραφο, χωρίζοντας σε ακολουθίες δύο ή περισσότερων newlines:

```
{
    local $/ = "";
    while (my $para = <$fh>) {
        # $para ends with exactly two newlines (or one at EOF)
        process_paragraph($para);
    }
}
```

Αυτή είναι η συμπεριφορά `RS=""` του awk. Κενές γραμμές στην αρχή του αρχείου παραλείπονται σιωπηρά.

Λειτουργία σταθερής εγγραφής

Η `local $/ = \N` διαβάζει `N` bytes ανά κλήση:

```
local $/ = \4096;                # binary 4-KiB chunks
while (defined(my $chunk = <$fh>)) {
    process($chunk);
}
```

Η αναφορά είναι προς αριθμό, όχι στον ίδιο τον αριθμό. Αυτή είναι η σωστή μορφή για ανάγνωση δυαδικών δεδομένων, πακέτων δικτύου, ή μαζική αντιγραφή χωρίς σάρωση για `\n`.

Τερματιστές πολλαπλών χαρακτήρων

```
local $/ = "\r\n";               # CRLF mode
local $/ = "\nEND-OF-RECORD\n";  # custom marker
```

Ο τερματιστής είναι κυριολεκτική συμβολοσειρά, όχι regex.

`$\` - ο διαχωριστής εγγραφών εξόδου

Προσαρτάται σε κάθε κλήση `print`. Η προεπιλογή είναι `undef` (τίποτα δεν προσαρτάται). Η ρύθμισή της αλλάζει κάθε επόμενο `print`:

```
{
    local $\ = "\n";             # all prints get a trailing newline
    print "first";
    print "second";
}
# Output: "first\nsecond\n"
```

Αυτή είναι η φυσική αντιστοιχία για τη `-l` στη γραμμή εντολών - η `-l` θέτει τη `$\` σε `\n`. Σε κώδικα σεναρίου, η `local $\ = "\n"` μέσα σε σφιχτό βρόχο αναφοράς αξίζει περιστασιακά· αλλιώς η `say` (που επιβάλλει `\n` ανεξάρτητα από τη `$\`) είναι η καθαρότερη γραφή.

$, - ο διαχωριστής πεδίων εξόδου

Εισάγεται μεταξύ ορισμάτων `print`. Η προεπιλογή είναι undef:

```
print "a", "b", "c";              # "abc"

{
    local $, = ", ";
    print "a", "b", "c";          # "a, b, c"
}

{
    local $, = "\t";
    print @items;                 # tab-separated record
}
```

Η `$,` εφαρμόζεται ανά print, όχι ανά γραμμή - δεν διαχωρίζει *διαδοχικές* κλήσεις `print`. Συνδυάστε με τη `$\` για πλήρη έξοδο τύπου TSV:

```
local ($,, $\) = ("\t", "\n");
print 'col1', 'col2', 'col3';     # "col1\tcol2\tcol3\n"
```

$" - ο διαχωριστής λίστας

Ενώνει στοιχεία πίνακα όταν ένας πίνακας παρεμβάλλεται σε συμβολοσειρά σε διπλά εισαγωγικά. Η προεπιλογή είναι `" "` (ένα κενό):

```
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"
}
```

Η `$"` συμβουλεύεται *μόνο* κατά την παρεμβολή. Η `print @arr` (χωρίς εισαγωγικά) χρησιμοποιεί τη `$,`, όχι τη `$"`.

$| - αυτόματη εκκένωση εξόδου

Επιβάλλει εκκένωση μετά από κάθε εγγραφή στο **τρέχον επιλεγμένο** filehandle:

```
$| = 1;                           # autoflush STDOUT
print "Loading..."; sleep 1;     # appears immediately, not at exit
```

Η διευκρίνιση «τρέχον επιλεγμένο» έχει σημασία. Εξ ορισμού, η `select` επιστρέφει STDOUT, οπότε η `$| = 1` κάνει αυτόματη εκκένωση στο STDOUT. Για αυτόματη εκκένωση

σε διαφορετικό handle:

```
my $old = select($log_fh);
$| = 1;
select($old);                    # restore selection
```

Η ελαφρώς ωραιότερη σύγχρονη γραφή χρησιμοποιεί την `IO::Handle`:

```
$log_fh->autoflush(1);
```

Η `$|` διαβάζεται ως λογική τιμή: η `print $|` εμφανίζει `1` ή `0`.

`$.` - αριθμός τρέχουσας γραμμής

Ο αριθμός γραμμής της τελευταίας ανάγνωσης τύπου <>. Κάθε filehandle έχει τον δικό του μετρητή· η ανάγνωση από διαφορετικό handle ψευδωνυμεί τη `$.` στον μετρητή εκείνου του handle:

```
while (<$fh>) {
    print "$.: $_";              # numbered listing
}

# After the loop, $. is the total number of lines read.
```

Η `$.` μηδενίζεται όταν *κλείσει* το filehandle. Ξανα-άνοιγμα (χωρίς ρητό κλείσιμο ενδιάμεσα) **δεν** μηδενίζει.

Για επανάληψη πολλαπλών αρχείων μέσω <>, οι γραμμές αριθμούνται σωρευτικά μεταξύ αρχείων εκτός αν κλείσετε ρητά το ARGV σε κάθε όριο αρχείου - δείτε `eof` για το τυπικό ιδίωμα `close ARGV if eof;`.

`$;` - ο διαχωριστής δεικτών

Χρησιμοποιείται για εξομοίωση πολυδιάστατων hashes - ένα βαθιά παλαιό χαρακτηριστικό που διατηρείται για συμβατότητα προς τα πίσω:

```
$h{$x, $y} = $value;             # equivalent to $h{"$x\034$y"}
```

Η προεπιλεγμένη τιμή είναι `"\034"` (FS, «file separator», ένα σπάνιο byte ελέγχου ASCII). Πραγματικές πολυδιάστατες δομές πρέπει να χρησιμοποιούν ένθετες αναφορές hash (`$h{$x}{$y}`)· η `$;` υπάρχει επειδή η Perl 4 δεν είχε σωστές αναφορές.

Πότε στην πράξη κάνετε local σε αυτά

Η πιο συνηθισμένη μορφή, μακράν:

```
sub slurp_text {
    my ($path) = @_;
    open my $fh, '<', $path or die "open $path: $!";
    local $/;                    # slurp inside this sub only
    return <$fh>;
}
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
sub paragraphs_of {
    my ($path) = @_;
    open my $fh, '<', $path or die "open $path: $!";
    local $/ = "";
    return <$fh>;                # list context: list of paragraphs
}

sub csv_print {
    my (@row) = @_;
    local ($,, $\) = (",", "\n");
    print @row;
}
```

Το κόστος του να ξεχάσετε τη `local` είναι αόρατο κατά τη γραφή και καταστροφικό κατά τη χρήση - μια υπορουτίνα που «απλώς θέτει τη `$/`» μπορεί να σπάσει άσχετο κώδικα που διαβάζει από εντελώς διαφορετικά filehandles. Αντιμετωπίστε τις `$/`, `$\`, `$,`, `$"` ως *καθολικές σημαίες λειτουργίας*: θέτετέ τις πάντα με `local`.

Δείτε επίσης

- `open`, `readline`, `<$fh>` - οι καταναλωτές της `$/`.
- `print`, `printf`, `say` - οι καταναλωτές των `$\`, `$,`.
- `chomp` - αφαιρεί ό,τι κρατάει τη δεδομένη στιγμή η `$/` · ζευγαρώνει με αλλαγές `$/`.
- `eof` - το ιδίωμα `close ARGV if eof` που κάνει τη `$.` να μετράει ανά αρχείο σε `<>`.
- `select` - επιλέγει το filehandle στο οποίο εφαρμόζονται η `$|` και τα χωρίς πρόθεμα `$,`/`$\`.
- `IO::Handle` - παρέχει `autoflush`, `output_record_separator`, κ.λπ. ως κλήσεις μεθόδων.
- `local` - ο ασφαλής τρόπος αλλαγής οποιασδήποτε μεταβλητής σε αυτή τη σελίδα.
- *Παρεμβολή* - πώς η `$"` διαμορφώνει την επέκταση πίνακα μέσα σε συμβολοσειρές σε διπλά εισαγωγικά.

Μεταβλητές αντιστοίχισης regex

Μετά από κάθε επιτυχημένη αντιστοίχιση μοτίβου, η Perl συμπληρώνει ένα σταθερό σύνολο μεταβλητών με πληροφορίες για το *τι* ταίριαξε και *πού*. Αυτές είναι μόνο-ανάγνωσης - τις παρατηρείτε, δεν γράφετε σε αυτές. Η διευκρίνιση «επιτυχημένη» είναι κρίσιμη: μια ανεπιτυχής αντιστοίχιση αφήνει τις μεταβλητές να κρατούν *ό,τι τις έθεσε η προηγούμενη επιτυχημένη αντιστοίχιση στην ίδια δυναμική εμβέλεια*. Φιλτράρετε πάντα την πρόσβαση με το λογικό αποτέλεσμα της ίδιας της αντιστοίχισης.

Μεταβλητή	Κρατά
`$1..$N`	Κείμενο που συλλήφθηκε από την *N*-οστή ομάδα σύλληψης
`$&`	Ολόκληρη η υποσυμβολοσειρά που ταίριαξε
`$``	Η συμβολοσειρά πριν την αντιστοίχιση
`$'`	Η συμβολοσειρά μετά την αντιστοίχιση
`$+`	Κείμενο που συλλήφθηκε από την ομάδα με τον μεγαλύτερο αριθμό
`$^N`	Κείμενο που συλλήφθηκε από την *πιο πρόσφατα κλεισμένη* ομάδα
`@-`	Offset αρχής: `$-[0]` = αντιστοίχιση, `$-[N]` = αρχή `$N`
`@+`	Offset τέλους: `$+[0]` = τέλος αντιστοίχισης, `$+[N]` = τέλος `$N`
`%+`	Hash ονομαστικών συλλήψεων: `$+{name}` = `(?<name>...)`
`%-`	Hash *όλων* των συλλήψεων κατά όνομα (τιμές arrayref)
`@{^CAPTURE}`	Πίνακας συλλήψεων: `${^CAPTURE}[0]` = `$1`, κ.λπ.
`${^MATCH}`	Ίδιο με τη `$&`, συμπληρώνεται μόνο με τη σημαία `/p`
`${^PREMATCH}`	Ίδιο με τη `$``, μόνο με `/p`
`${^POSTMATCH}`	Ίδιο με τη `$'`, μόνο με `/p`

Το βασικό μοτίβο

```
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
}
```

Το `if` είναι αυτό που κάνει την πρόσβαση ασφαλή. Χωρίς αυτό, σε μια *μη ταιριαστή* συμβολοσειρά, οι `$1`/`$&` θα κρατούσαν ακόμα τιμές από κάποια προηγούμενη επιτυχημένη αντιστοίχιση - δεν υπάρχει κανόνας «αποτυχία αντιστοίχισης → καθαρισμός».

Αριθμημένες συλλήψεις - `$1..$N`

Κάθε ομάδα σύλληψης `(...)` γεμίζει μία μεταβλητή. Η αρίθμηση είναι από αριστερά προς τα δεξιά βάσει *αρχικής* παρένθεσης:

```
"alpha-beta=42" =~ /^(\w+)-(\w+)=(\d+)$/;
print "$1 / $2 / $3\n";           # alpha / beta / 42
```

Οι μη συλλαμβάνουσες ομάδες `(?:...)` και οι διεκδικήσεις `(?=...)`/`(?!...)` **δεν** καταναλώνουν αριθμό· είναι αόρατες στην αρίθμηση.

Για αντικαταστάσεις `s///`, οι `$1..$N` είναι ορατές μέσα στην αντικατάσταση (μαζί με τις οπισθαναφορές τύπου `$1` μόνο αντικατάστασης σε αντικαταστάσεις σε μονά εισαγωγικά):

```
my $s = "John Smith";
$s =~ s/(\w+)\s+(\w+)/$2, $1/;    # "Smith, John"
```

Ονομαστικές συλλήψεις - `(?<name>...)` και `%+`

Οι ονομαστικές συλλήψεις είναι πιο ξεκάθαρες από τη μέτρηση παρενθέσεων, ειδικά σε μοτίβα με πολλές ομάδες:

```
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";
}
```

Η `%+` είναι το hash ονομαστικών συλλήψεων. Οι αριθμημένες μορφές εξακολουθούν να λειτουργούν (οι ονομαστικές συλλήψεις λαμβάνουν επίσης αριθμούς με τη σειρά εμφάνισης), οπότε η `$1` θα ήταν η `$+{date}` εδώ.

Η `%-` είναι παρόμοια αλλά οι τιμές της είναι αναφορές πίνακα, που κρατούν *κάθε* σύλληψη κάτω από αυτό το όνομα (σχετικό όταν χρησιμοποιούνται ομάδες επαναφοράς διακλάδωσης `(?|...|...)` ή όταν ένα όνομα επαναχρησιμοποιείται μεταξύ κλάδων εναλλαγής):

```
"abc" =~ /(?|(?<x>a)|(?<x>b)|(?<x>c))/;
print "%-{x} has @{$-{x}}\n";    # captured value(s) under name 'x'
```

Ο περισσότερος κώδικας διαβάζει μόνο τη `%+`. Η `%-` είναι για τις οριακές περιπτώσεις.

Όρια αντιστοίχισης - `@-` και `@+`

Τα offset αρχής και τέλους της αντιστοίχισης στη συμβολοσειρά που ταίριαξε:

```
"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
```

Οι `$-[N]` και `$+[N]` δίνουν τις ίδιες πληροφορίες που θα εξήγαγε η `substr($var, $-[N], $+[N] - $-[N])` - είναι ο τρόπος ανακατασκευής *θέσεων*, όχι τιμών, μετά από αντιστοίχιση.

Η κλασική χρήση: αντικατάσταση υποσυμβολοσειρών που ταίριαξαν διατηρώντας το πρωτότυπο (χωρίς τον τελεστή `s///`):

```
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";
```

$&, $`, $' - αντιστοίχιση, πριν-αντιστοίχιση, μετά-αντιστοίχιση

```
"hello world" =~ /wo\w+/;
print "before: '$`'\n";          # 'hello '
print "match:  '$&'\n";          # 'world'
print "after:  '$\''\n";         # ''
```

Αυτές οι τρεις είναι οι παλαιότερες μεταβλητές αντιστοίχισης της Perl και ιστορικά οι πιο ακριβές - δείτε *Απόδοση: η ιστορία της* $& παρακάτω.

Η $+ είναι το κείμενο που συλλήφθηκε από την ομάδα *με τον μεγαλύτερο αριθμό* που συμμετείχε στην αντιστοίχιση:

```
"abc" =~ /(a)(b)?(c)?/;          # $1='a', $2='b', $3='c', $+='c'
"a"   =~ /(a)(b)?(c)?/;          # $1='a', $2=undef, $3=undef, $+='a
 ↪'
```

Η $^N είναι παρόμοια αλλά κρατά το κείμενο της πιο πρόσφατα *κλεισμένης* ομάδας - χρήσιμη μέσα σε σύνθετα μοτίβα που χρειάζονται την τιμή «της ομάδας που μόλις ολοκληρώθηκε»:

```
my $s = "tag:value";
$s =~ /(\w+):(\w+) (?{ $cb = $^N })/;
# $cb is the text most recently captured (here: "value")
```

Απόδοση: η ιστορία της $&

Ιστορική επιφύλαξη που θα βρείτε σε παλαιό κώδικα και σε βιβλία:

> Μην αναφέρετε τις $&, $`, ή $' *πουθενά* στον κώδικά σας, συμπεριλαμβανομένων αρθρωμάτων που κάνετε require - προκαλούν κάθε επιτυχημένη αντιστοίχιση να αντιγράψει ολόκληρη τη συμβολοσειρά που ταίριαξε, επιβραδύνοντας το πρόγραμμα.

Αυτό ίσχυε μέχρι την Perl 5.10. Από την Perl 5.18 και μετά, ο χρόνος εκτέλεσης παρακολουθεί *ποια* από τις τρεις μεταβλητές αναφέρει πραγματικά ο κώδικάς σας και αντιγράφει μόνο ό,τι χρειάζεται. Από την Perl 5.20 ένα σχήμα copy-on-write τις κάνει ουσιαστικά χωρίς κόστος.

Η PetaPerl ακολουθεί τη σύγχρονη συμπεριφορά: η $& και οι σχετικές είναι ασφαλείς για χρήση οπουδήποτε. Η σημαία /p και οι ${^MATCH} / ${^PREMATCH} / ${^POSTMATCH} υπάρχουν για την εποχή μεταξύ 5.10 και 5.20 όταν η αντιγραφή μέσω /p ήταν χρήσιμη βελτιστοποίηση. Δεν υπάρχει λόγος να γράψετε νέο κώδικα μαζί τους.

Εμβέλεια - είναι δυναμικά οριοθετημένες

Οι μεταβλητές αντιστοίχισης δεν είναι λεξιλογικές· συμπεριφέρονται σαν δυναμικά οριοθετημένες μεταβλητές. Κάθε επιτυχημένη αντιστοίχιση κάνει local μια *καθολική κατάσταση αντιστοίχισης* στην τρέχουσα δυναμική εμβέλεια. Κρίσιμο:

- Μια *ανεπιτυχής* αντιστοίχιση **δεν** καθαρίζει τις μεταβλητές.
- Μια επιτυχημένη αντιστοίχιση *μέσα σε εσωτερικό μπλοκ* τις υπερισχύει, αλλά μόνο μέσα σε εκείνο το μπλοκ - όταν εξέλθει το εσωτερικό μπλοκ, η κατάσταση

αντιστοίχισης της εξωτερικής εμβέλειας αποκαθίσταται.

```
"alpha" =~ /(\w+)/;              # $1 = 'alpha'
{
    "1234" =~ /(\d+)/;           # $1 = '1234' (inside this block)
    print "inner: $1\n";          # 1234
}
print "outer: $1\n";              # alpha - restored
```

Αυτό είναι περιστασιακά εκπληκτικό: μια συνάρτηση που καλείτε μέσα από μπλοκ χειρισμού regex δεν μολύνει τη $1 σας εκτός αν εκείνη η συνάρτηση κάνει δική της επιτυχημένη αντιστοίχιση στην *ίδια* δυναμική εμβέλεια (κάτι σπάνιο εκτός αν εκτελούνται στο ίδιο λεξιλογικό μπλοκ).

${^LAST_SUCCESSFUL_PATTERN}

Μια αναφορά μόνο-ανάγνωσης προς το regex που παρήγαγε την τρέχουσα κατάσταση αντιστοίχισης - χρήσιμη για διαγνωστικά:

```
"hello" =~ /(\w+)/;
print "last pattern was: ${^LAST_SUCCESSFUL_PATTERN}\n";
```

Όταν αποτυγχάνουν οι αντιστοιχίσεις - pos

Η θέση σε μια συμβολοσειρά μετά από αντιστοίχιση αγκυρωμένη με /g κρατείται *όχι* σε μία από τις μεταβλητές αυτής της σελίδας, αλλά στη 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.
```

Η pos είναι ανά συμβολοσειρά, ρυθμίσιμη, και είναι αυτό στο οποίο αγκυρώνεται η \G.

@{^CAPTURE} - οι συλλήψεις ως πίνακας

Οι αριθμημένες συλλήψεις, εκτεθειμένες επίσης ως πίνακας μηδενοβάσει:

```
"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
```

Αυτό είναι περιστασιακά ευκολότερο στην επανάληψη από τα $1, $2, ..., αλλά ο περισσότερος κώδικας χρησιμοποιεί τις αριθμημένες ή ονομαστικές μορφές απευθείας.

Δείτε επίσης

- `m//`, `s///` - οι τελεστές που συμπληρώνουν κάθε μεταβλητή σε αυτή τη σελίδα.
- `qr//` - μεταγλωττίζει ένα μοτίβο· το αντικείμενο regex που προκύπτει μπορεί αργότερα να αντιστοιχιστεί και να παράγει τις ίδιες συλλήψεις.
- `pos` - το offset ανά συμβολοσειρά για `/g` και `\G`.
- *Σύνδεση regex* - η =~, ο τελεστής που αποφασίζει σε ποια συμβολοσειρά εκτελείται η αντιστοίχιση.
- *Regular expressions guide*
  - the regex language itself.
- *Ομάδες και συλλήψεις* - το κεφάλαιο για τις μεταβλητές σύλληψης.
  - Οι αποτυχημένες αντιστοιχίσεις δεν επαναφέρουν τις μεταβλητές σύλληψης

Ενδοσκόπηση διερμηνέα

Μεταβλητές που εκθέτουν τον τρέχοντα διερμηνέα Perl και τη ρύθμισή του. Ο περισσότερος κώδικας ποτέ δεν τις αγγίζει· οι περιπτώσεις όπου έχουν σημασία είναι έλεγχοι έκδοσης, λογική υπό συνθήκη OS, hooks profiler, και επιτόπια επεξεργασία.

Μεταβλητή	Κρατά
`$]`	Perl version as a numeric string (`5.044000`)
`$^V`	Perl version as a `version` object (`v5.44.0`)
`$^O`	Όνομα λειτουργικού συστήματος (`linux`, `darwin`, ...)
`$^X`	Διαδρομή προς το τρέχον εκτελέσιμο Perl/PetaPerl
`$^T`	Χρόνος βάσης διεργασίας (δευτερόλεπτα Unix epoch)
`$^W`	Κεντρικός διακόπτης προειδοποιήσεων (λογική τιμή)
`$^I`	Επέκταση επιτόπιας επεξεργασίας (τίθεται από `-i`)
`$^P`	Μάσκα bits σημαιών αποσφαλματωτή
`$^R`	Αποτέλεσμα τελευταίου μπλοκ κώδικα regex
`$^S`	Κατάσταση χειρισμού εξαιρέσεων (εντός `eval`; `undef`/0/1)
`${^GLOBAL_PHASE}`	Τρέχουσα φάση διερμηνέα (`START`, `RUN`, `END`, ...)

`$]` - αριθμητική έκδοση

Μια αναπαράσταση κινητής υποδιαστολής της έκδοσης του διερμηνέα. Μορφή: `5.` PPPSSS όπου PPP είναι η κύρια και SSS η υποέκδοση:

```
print "$]\n";                    # e.g. "5.044000"
warn "old perl"  if "$]" < 5.038;
warn "new perl"  if "$]" >= 5.040;
```

The `"$]"` stringification before comparison is recommended - it sidesteps subtle floating-point representation issues that can make `$] == 5.044` mysteriously fail.

Το ψευδώνυμο English `$OLD_PERL_VERSION` είναι τεκμηριωμένο αλλά αποθαρρύνεται· η ίδια η `$]` είναι εντάξει.

$^V - αντικείμενο έκδοσης

Η έκδοση διερμηνέα ως αντικείμενο `version`, κατάλληλο για ανάγνωση και πλουσιότερη σύγκριση:

```
print "$^V\n";                  # "v5.44.0"
warn "need 5.40+" if $^V lt v5.40.0;
```

Η κυριολεκτική `v5.40.0` είναι v-string - η εγγενής σύνταξη της Perl για σταθερές εκδόσεων. Χρησιμοποιήστε τη `$^V` όταν συγκρίνετε με κυριολεκτικά v-string· χρησιμοποιήστε τη `$]` όταν συγκρίνετε με σκέτους αριθμούς.

In PetaPerl, `$^V` reports the underlying Perl-compatibility level (5.44), not the PetaPerl release version.

$^O - όνομα λειτουργικού συστήματος

```
print "$^O\n";                  # "linux"
if ($^O eq 'MSWin32') {
    use_windows_path_separator();
}
```

Η PetaPerl είναι μόνο για Linux· η `$^O` είναι πάντα `"linux"`. Η μεταβλητή υπάρχει για συμβατότητα πηγαίου κώδικα με φορητό κώδικα που διακλαδίζεται βάσει αυτής. Η πλήρης λίστα τιμών που μπορεί να παράξει η upstream Perl περιλαμβάνει `linux`, `darwin`, `freebsd`, `openbsd`, `MSWin32`, `cygwin`, `solaris`, και αρκετές ιστορικές καταχωρήσεις.

$^X - διαδρομή εκτελέσιμου

Η διαδρομή προς τον τρέχοντα εκτελούμενο διερμηνέα, κατάλληλη για επανεκκίνηση:

```
print "$^X\n";                  # e.g. "/usr/local/bin/pperl"

# Rerun the script under the same interpreter, with extra args:
exec $^X, $0, @ARGV, '--debug';
```

Η `$^X` είναι αυτό που κάνει φορητή τη `exec $^X, $0, ...` - δεν χρειάζεται να γνωρίζετε αν το σενάριο εκκινήθηκε μέσω `pperl`, `/usr/bin/env perl`, ή απόλυτης διαδρομής.

$^T - χρόνος βάσης

Το δευτερόλεπτο Unix epoch κατά το οποίο ξεκίνησε να εκτελείται το πρόγραμμα:

```
my $uptime = time() - $^T;
print "running for $uptime seconds\n";
```

Η κλασική χρήση είναι οι δοκιμές αρχείων `-M` και `-A`, που εξ ορισμού αναφέρουν ηλικίες αρχείων *σχετικές με* τη `$^T` αντί με τη `time()`. Αν θέσετε `$^T = time()` περιοδικά, αυτές οι δοκιμές αρχείων επανυπολογίζουν βάσει του νέου σημείου αναφοράς - χρήσιμο

σε δαίμονες μεγάλης διάρκειας που χρειάζεται να αγνοούν αρχεία παλαιότερα από «αφότου ξεκίνησα αυτή τη δέσμη».

$^W - ο διακόπτης προειδοποιήσεων

Η κεντρική σημαία προειδοποιήσεων, που τίθεται από τον διακόπτη γραμμής εντολών `-w`:

```
$^W = 1;                        # all warnings on
$^W = 0;                        # all warnings off
```

Η `$^W` είναι η καθολική σημαία εποχής Perl-1. Η σύγχρονη λεξιλογική pragma `use warnings` είναι ο συνιστώμενος τρόπος ελέγχου προειδοποιήσεων, και λειτουργεί ανεξάρτητα - ένα μπλοκ `use warnings` παραμένει ενεργό ανεξάρτητα από τη `$^W`, και ένα μπλοκ `no warnings` παραμένει ανενεργό ανεξάρτητα από τη `$^W`. Η μεταβλητή έχει σημασία κυρίως σε κώδικα που προηγείται της λεξιλογικής pragma ή που θέλει να εναλλάξει την καθολική προεπιλογή κατά τον χρόνο εκτέλεσης.

Η `${^WARNING_BITS}` είναι ο λεξιλογικός χάρτης bits που διαβάζει και γράφει η `use warnings`. Η μορφή της είναι εσωτερική· μην βασίζεστε σε συγκεκριμένες θέσεις bits.

$^I - επέκταση επιτόπιας επεξεργασίας

Τίθεται από τον διακόπτη γραμμής εντολών `-i`. Όταν η `$^I` είναι ορισμένη, η ανάγνωση από `<>` (η μορφή `while (<>)`) ξανα-ανοίγει κάθε αρχείο για ανάγνωση και εγγραφή και ανακατευθύνει τα επόμενα `print` στο ξαναγραμμένο αρχείο:

```
# perl -i.bak -ne 's/\bold\b/new/g; print' *.txt
# Equivalent in script form:
local $^I = '.bak';
while (<>) {
    s/\bold\b/new/g;
    print;
}
```

Η `$^I = ''` (κενή συμβολοσειρά) σημαίνει «ξαναγράψε επιτόπια χωρίς αντίγραφο ασφαλείας». Η `$^I = '.bak'` αποθηκεύει το πρωτότυπο ως `<filename>.bak`. Η ρύθμιση `$^I = undef` απενεργοποιεί την επιτόπια επεξεργασία.

$^P - μάσκα bits σημαιών αποσφαλματωτή

Διαβάζεται από τον αποσφαλματωτή Perl· αν δεν είναι μηδέν, η `DB::DB` καλείται πριν από κάθε δήλωση. Χειρίζεται από τη `perl -d` και την οικογένεια αρθρωμάτων αποσφαλματωτή και profiler `Devel::*`.

Ο κώδικας χρήστη σπάνια θέτει τη `$^P`. Τα bits σημαιών τεκμηριώνονται στο upstream perlvar· η πιο συνηθισμένη τιμή είναι `0x73f` (πλήρης αποσφαλμάτωση).

$^R - αποτέλεσμα μπλοκ κώδικα regex

Το αποτέλεσμα της τελευταίας διεκδίκησης κώδικα regex `(?{...})` ή `(?(N)...)`. Ζει μόνο όσο διαρκεί η περιβάλλουσα αντιστοίχιση:

```
"abc123" =~ /(?<word>\w+?)(\d+)(?{ length($1) + length($2) })/;
print "$^R\n";                  # 6 - the code block's last␣
↪expression
```

Η PetaPerl υποστηρίζει μπλοκ κώδικα regex `(?{ })`· δείτε το *κεφάλαιο απόδοσης* του οδηγού regex για τις λεπτομέρειες γλώσσας.

$^S - κατάσταση χειρισμού εξαιρέσεων

Αναφέρει αν ο διερμηνέας βρίσκεται αυτή τη στιγμή εντός ενός `eval`. Κρατά μία από τρεις τιμές:

$^S	Σημασία
undef	Ανάλυση ενός module, ενός `eval` ή του κύριου προγράμματος
1 (αληθές)	Εκτέλεση εντός ενός μπλοκ `eval` ή `try`
0 (ψευδές)	Οπουδήποτε αλλού

Η κατάσταση undef είναι αυτή που έχει σημασία: μπορεί να εμφανιστεί εντός ενός χειριστή `$SIG{__DIE__}` ή `$SIG{__WARN__}`, όπου ο διερμηνέας μεταγλωττίζει αντί να εκτελεί. Ένας χειριστής που χρειάζεται να συμπεριφέρεται διαφορετικά κατά τη μεταγλώττιση την ελέγχει:

```
local $SIG{__DIE__} = sub {
    return if $^S;              # real runtime exception - let it␣
↪propagate
    warn "compile-time failure: $_[0]";
};
```

Το παλαιό μακρύ όνομα (`use English`) είναι `$EXCEPTIONS_BEING_CAUGHT`, το οποίο είναι ελαφρώς παραπλανητικό: η τιμή undef δεν σας λέει αν οι εξαιρέσεις πιάνονται, διότι η μεταγλώττιση του κύριου προγράμματος δεν τις πιάνει.

${^GLOBAL_PHASE} - τρέχουσα φάση διερμηνέα

Η τρέχουσα φάση κύκλου ζωής του διερμηνέα Perl:

Φάση	Πότε
CONSTRUCT	Μέσα στον κατασκευαστή διερμηνέα σε επίπεδο C
START	Μπλοκ BEGIN, use, require, μεταγλώττιση ανώτατου επιπέδου
CHECK	Μετά τη μεταγλώττιση, πριν το INIT
INIT	Μπλοκ INIT
RUN	Εκτέλεση κύριου προγράμματος
END	Μπλοκ END
DESTRUCT	Καθολική καταστροφή

```
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;
}
```

Ο φύλακας «μην εκτελέσεις κατά τη διάρκεια καθολικής καταστροφής» μέσα στη DESTROY είναι η κλασική περίπτωση χρήσης της μεταβλητής. Κατά τη DESTRUCT, η σειρά αποδόμησης αντικειμένων δεν είναι ορισμένη· οι αναφορές ενός αντικειμένου μπορεί να είναι ήδη άκυρες.

Δείτε επίσης

- `use VERSION` - ο συμβατικός τρόπος για απαίτηση ελάχιστης έκδοσης Perl, αντικαθιστώντας τη `if "$]" < ...`.
- `require VERSION` - ίδιο με `use VERSION` αλλά κατά τον χρόνο εκτέλεσης.
- `Config` - η πλήρης ρύθμιση μεταγλώττισης Perl, πολύ πλουσιότερη από τη `$^O` και τη `$]`.
- `use warnings` - η λεξιλογική pragma που αντικαθιστά τη `$^W` για νέο κώδικα.
- *Διακόπτες γραμμής εντολών* - τεκμηριώνει τους `-i` (`$^I`), `-w` (`$^W`), `-d` (`$^P`).

Είσοδος προγράμματος - @ARGV, %ENV, @INC

Τρεις καθολικές μεταβλητές μεταφέρουν είσοδο *στο* πρόγραμμα: ορίσματα γραμμής εντολών, μεταβλητές περιβάλλοντος, και τη διαδρομή αναζήτησης αρθρωμάτων. Μοιράζονται αρκετό πλαίσιο ώστε οι αποφάσεις για μία συνήθως ενημερώνουν τις άλλες - η `@ARGV` αλληλεπιδρά με τη `$0`, τα κλειδιά σχετικά με Perl στη `%ENV` επηρεάζουν τη `@INC`, και το ερώτημα σειράς της `@INC` αντικατοπτρίζεται σε παρόμοιες ανησυχίες για καταναλωτές `@ARGV` και το `PATH` του κελύφους.

Μεταβλητή	Κρατά
`@ARGV`	Ορίσματα γραμμής εντολών (χωρίς το όνομα σεναρίου)
`$0`	Όνομα σεναρίου (σε ζεύγος με τη `@ARGV`)
`%ENV`	Περιβάλλον διεργασίας
`@INC`	Διαδρομή αναζήτησης αρθρωμάτων (χρησιμοποιείται από `use`/`require`)
`%INC`	Ήδη φορτωμένα αρθρώματα
`$INC`	Τρέχων δείκτης `@INC` κατά τη διάρκεια hook (5.37.7+)

Η @ARGV και η σχέση της με τη $0

Μετά την εκκίνηση της Perl, το *όνομα σεναρίου* βρίσκεται στη `$0` και τα *ορίσματα του σεναρίου* στη `@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
```

Η `$ARGV[0]` είναι το *πρώτο όρισμα του σεναρίου*, **όχι** το ίδιο το όνομα σεναρίου. Αυτό διαφέρει από το `argv` της C (όπου `argv[0]` είναι το όνομα προγράμματος)· η Perl εκθέτει το όνομα προγράμματος ξεχωριστά μέσω `$0`.

Τυπικός χειρισμός ορισμάτων

Η χειρόγραφη ανάλυση ορισμάτων είναι εντάξει για τετριμμένα σενάρια:

```
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;
```

Για οτιδήποτε πέρα από απλοϊκά σενάρια, το άρθρωμα `Getopt::Long` χειρίζεται κάθε τυπικό μοτίβο (μεγάλες επιλογές, συντομογραφία, υποχρεωτικά ορίσματα, επαναλαμβανόμενες σημαίες, διαχωριστής `--`, αυτόματη βοήθεια):

```
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";
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
my @files = @ARGV;                  # what's left after option␣
↪processing
```

Η GetOptions τροποποιεί τη @ARGV επιτόπια - όταν επιστρέψει, η @ARGV περιέχει μόνο τα ορίσματα που δεν είναι επιλογές.

Η @ARGV και ο τελεστής «διαμάντι»

Ο τελεστής <> (διαμάντι) επαναλαμβάνει γραμμές από τα αρχεία που ονομάζονται στη @ARGV, χρησιμοποιώντας εφεδρικά το STDIN αν η @ARGV είναι κενή:

```
while (<>) {
    # one line from one of the @ARGV files (or STDIN)
    chomp;
    process($_);
}
```

Αυτή είναι η προεπιλεγμένη σύμβαση κάθε προγράμματος φίλτρου τύπου awk. Η <> ανοίγει κάθε αρχείο διαδοχικά, θέτει τη $ARGV στο όνομά του κατά την ανάγνωση, και το κλείνει στο EOF.

Η -i και οι άλλοι διακόπτες - (-n, -p, -l, -a) χτίζουν όλοι τη συμπεριφορά τους πάνω σε αυτό το μοτίβο @ARGV → <>.

Τροποποίηση @ARGV πριν τη <>

Μπορείτε να προ-επεξεργαστείτε τη @ARGV για να αλλάξετε τι διαβάζει η <>:

```
# 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($_);
}
```

Η μορφή cmd | μετατρέπει το όρισμα σε open-από-σωλήνα-εντολής· αυτό είναι ένα παλιό ιδίωμα Perl για διαφανή αποσυμπίεση σε σενάρια τύπου awk.

%ENV - το περιβάλλον διεργασίας

Η %ENV είναι το hash μεταβλητών περιβάλλοντος που κληρονομούνται από τη γονική διεργασία (τυπικά το κέλυφος). Η ανάγνωση είναι απλή:

```
my $home     = $ENV{HOME};
my $term     = $ENV{TERM} // 'dumb';
my $path     = $ENV{PATH};
```

Η εγγραφή αλλάζει το περιβάλλον που βλέπουν οι *διεργασίες-παιδιά* που αυτό το σενάριο Perl δημιουργεί στη συνέχεια:

```
$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
```

Η αλλαγή *δεν* διαδίδεται πίσω στον γονέα - το κέλυφος που εκκίνησε το σενάριο κρατά το αρχικό του περιβάλλον. Αν χρειάζεστε το γονικό κέλυφος να πάρει νέες τιμές, γράψτε τες σε αρχείο ή ζητήστε από το κέλυφος να κάνει `eval` την έξοδό σας (το τυπικό μοτίβο `dotenv`).

Μετατροπή τιμών σε συμβολοσειρά

Από την Perl 5.18, οι τιμές %ENV μετατρέπονται πάντα σε συμβολοσειρά κατά τον χρόνο ανάθεσης. Η αποθήκευση αναφοράς δεν κάνει πλέον πλήρη κύκλο:

```
my $arr = [1, 2, 3];
$ENV{DATA} = $arr;
print $ENV{DATA};                  # "ARRAY(0x...)" - stringified
```

Οι μεταβλητές περιβάλλοντος είναι μια διεπαφή τύπου συμβολοσειράς με το λειτουργικό σύστημα· η διατήρηση δομής μέσω ορίου `fork`/`exec` δεν θα λειτουργούσε ούτως ή άλλως. Σειριοποιήστε ό,τι μη τετριμμένο ως JSON ή παρόμοιο πριν την αποθήκευση.

Μεταβλητές περιβάλλοντος ειδικές της Perl

Ένα μικρό σύνολο μεταβλητών περιβάλλοντος τηρείται από την ίδια τη Perl, *όχι* από σενάρια χρηστών:

Μεταβλητή	Αποτέλεσμα
`PERL5LIB`	Διαδρομές χωρισμένες με άνω-κάτω τελεία που προστίθενται στην αρχή της `@INC` (υποκείμενες σε κανόνες taint)
`PERLLIB`	Ίδια, παλαιότερη μεταβλητή· χρησιμοποιείται μόνο αν η `PERL5LIB` δεν έχει τεθεί
`PERL5OPT`	Διακόπτες που εφαρμόζονται σαν να ήταν στη γραμμή εντολών (μόνο υποσύνολο)
`PERLDB_OPTS`	Επιλογές για τον αποσφαλματωτή
`PERL_UNICODE`	Προεπιλεγμένες ρυθμίσεις στρώματος UTF-8 (αντικατοπτρίζει τη `-C`)
`PERL_USE_UNSA`	Επαναπρόσθεση `.` στη `@INC` (αφαιρέθηκε εξ ορισμού από την 5.26 - αποθαρρύνεται έντονα)
`PERL_HASH_SEE`	Σπόρος τυχαιοποίησης hash ανά εκτέλεση
`PERL_SIGNALS`	`unsafe` για εξαίρεση από τον αναβαλλόμενο χειρισμό σημάτων
`HOME`	Διαβάζεται από την επέκταση glob `~`· δεν είναι ειδική της Perl αλλά σχετική
`PATH`	Αναζητείται από `system`/`exec` για εντολές χωρίς πλήρη διαδρομή

Σε tainted mode (`-T`), οι `PERL5LIB`, `PERL5OPT`, και `PERLLIB` αγνοούνται για αποτροπή κλιμάκωσης προνομίων. Δείτε τον *οδηγό διακοπτών γραμμής εντολών* για το πλήρες μοντέλο ασφαλείας.

`delete $ENV{KEY}` έναντι `$ENV{KEY} = undef`

Τα δύο *δεν* είναι ισοδύναμα:

```
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

- η `getenv("KEY")` επιστρέφει NULL. Μετά από ανάθεση `undef`, το παιδί βλέπει `KEY=""` (κενή συμβολοσειρά). Κώδικας που διακρίνει «μη τεθειμένο» από «τεθειμένο σε κενό» ενδιαφέρεται για αυτή τη διαφορά.

`@INC` - η διαδρομή αναζήτησης αρθρωμάτων

Όταν η `use` και η `require` αναζητούν ένα άρθρωμα, διασχίζουν τη `@INC` από αριστερά προς τα δεξιά και σταματούν στην πρώτη αντιστοιχία:

```
print "$_\n" for @INC;
# /usr/local/lib/perl5/site_perl/5.44.0/x86_64-linux
# /usr/local/lib/perl5/site_perl/5.44.0
# /usr/local/lib/perl5/5.44.0/x86_64-linux
# /usr/local/lib/perl5/5.44.0
# (no trailing "." since Perl 5.26)
```

Η `@INC` αρχικοποιείται κατά την εκκίνηση από συνδυασμό:

1. Ενσωματωμένες προεπιλογές (οι `installprivlib`, `installsitelib`, ... μεταγλώττισης).
2. Διακόπτες `-I /path` στη γραμμή εντολών, *προστίθενται στην αρχή* κατά σειρά.
3. Η μεταβλητή περιβάλλοντος `PERL5LIB` (ή `PERLLIB`), επίσης στην αρχή.

Μετά την εκκίνηση είναι απλώς ένας κανονικός πίνακας. *Η σειρά μετράει*: μια αντιστοιχία σε πρώιμο στοιχείο υπερισχύει.

`use lib` έναντι `unshift @INC` έναντι `push @INC`

Τρεις τρόποι προσθήκης διαδρομής. Δεν είναι ισοδύναμοι:

```
use lib '/my/lib';                    # unshift at compile-time
unshift @INC, '/my/lib';              # unshift at runtime
push    @INC, '/my/lib';              # push    at runtime
```

Οι διαφορές:

- Η **`use lib`** εκτελείται *κατά τη μεταγλώττιση*. Συμβαίνει πριν από κάθε δήλωση `use` που την ακολουθεί στη σελίδα. Αυτό είναι αυτό που σχεδόν πάντα θέλετε όταν λέτε «αυτό το σενάριο χρειάζεται προσαρμοσμένο κατάλογο βιβλιοθήκης»:

```
use lib '/opt/myapp/lib';
use MyApp::Module;                    # found because lib was␣
↪unshifted first
```

Εσωτερικά η `use lib '/path'` κάνει `BEGIN { unshift @INC, '/path' }`, οπότε η νέα διαδρομή αναζητείται *πρώτη* - ένα αντίγραφο αρθρώματος στο `/opt/myapp/lib` επισκιάζει ένα στη `@INC` συστήματος.

- Η **`unshift @INC, '/path'`** στο ανώτατο επίπεδο έχει σημασιολογία *χρόνου εκτέλεσης*. Λειτουργεί για κλήσεις `require` που συμβαίνουν μετά τη `unshift`, αλλά μια δήλωση `use` σε μεταγενέστερη γραμμή του ίδιου αρχείου *δεν θα τη δει* (επειδή η `use` είναι η ίδια κατά τη μεταγλώττιση):

```
unshift @INC, '/my/lib';
use Some::Module;                     # NOT found in /my/lib -␣
↪too early
```

Αυτή είναι μια συχνή παγίδα. Η `BEGIN { unshift @INC, '/my/lib' }` τη διορθώνει· η `use lib` είναι η καθαρότερη γραφή ακριβώς αυτού.

- Η **`push @INC, '/path'`** προσαρτά. Η νέα διαδρομή αναζητείται *τελευταία*. Αυτή είναι η σωστή επιλογή όταν θέλετε η βιβλιοθήκη σας να είναι εφεδρική - χρησιμοποιείται μόνο αν καμία άλλη τοποθεσία δεν έχει το άρθρωμα - για παράδειγμα, παρέχοντας μια ενσωματωμένη έκδοση ενός αρθρώματος CPAN που ο χρήστης μπορεί να έχει εγκαταστήσει στο σύστημα.
  - for example, providing a built-in version of a CPAN module that the user might have installed on the system.

Ο πρακτικός κανόνας: χρησιμοποιείτε *πάντα* `use lib` για «θέλω η διαδρομή αρθρωμάτων του κώδικά μου να ισχύει κατά τη μεταγλώττιση», κάτι που ισχύει σχεδόν σε κάθε περίπτωση.

Hooks `@INC`

Η `@INC` μπορεί να περιέχει όχι μόνο διαδρομές αλλά και αναφορές κώδικα και blessed αντικείμενα. Όταν η `require` συναντά ένα από αυτά, το καλεί ως hook - περνώντας τη ζητούμενη διαδρομή αρθρώματος - και το hook επιστρέφει είτε ανοιχτό filehandle είτε λίστα πηγαίου κώδικα, γεννήτριες, κ.λπ. Έτσι μηχανισμοί όπως `Module::Pluggable`, PAR, και `Test::MockModule` υποκλέπτουν τη φόρτωση αρθρωμάτων.

Το πλήρες πρωτόκολλο hook τεκμηριώνεται στη `require`. Οι περισσότεροι χρήστες δεν γράφουν ποτέ ένα· τα διαβάζετε όταν αποσφαλματώνετε προβλήματα «γιατί αυτό το άρθρωμα δεν φορτώνεται από εκεί που περιμένω».

`%INC` - η cache φορτωμένων αρθρωμάτων

Μετά την επιτυχή φόρτωση ενός αρθρώματος, η `%INC` το καταγράφει. Το κλειδί είναι η διαδρομή που ζητήθηκε να βρει η Perl (π.χ. `Foo/Bar.pm`)· η τιμή είναι η απόλυτη διαδρομή που ικανοποίησε το αίτημα:

```
use Data::Dumper;
print "Data::Dumper loaded from $INC{'Data/Dumper.pm'}\n";
# /usr/local/lib/perl5/site_perl/5.44.0/Data/Dumper.pm

# Show every loaded module:
for my $key (sort keys %INC) {
    print "$key  →  $INC{$key}\n";
}
```

Η `require` ελέγχει τη `%INC` πριν διασχίσει τη `@INC` - μια δεύτερη `require Foo;` είναι no-op επειδή η `Foo.pm` βρίσκεται ήδη στη cache. Για εξαναγκασμό εκ νέου require:

```
delete $INC{'Foo.pm'};
require Foo;                    # actually re-runs Foo.pm
```

Αυτό είναι το τυπικό ιδίωμα για δοκιμή συμπεριφοράς επαναφόρτωσης. Λάβετε υπόψη ότι η εκ νέου εκτέλεση αρχείου αρθρώματος **δεν** αναιρεί ό,τι δημιούργησε η πρώτη εκτέλεση - πακέτα, subs, και καθολικές μεταβλητές παραμένουν.

`$INC` - ο δείκτης μέσα σε hook `@INC`

Διαθέσιμο από την Perl 5.37.7. Όταν καλείται ένα hook `@INC`, η `$INC` τίθεται στον δείκτη του hook στη `@INC`. Μετά την επιστροφή του hook, ο επαναλήπτης προχωρά βάσει `$INC + 1` - οπότε το hook μπορεί να ξαναγράψει τη `@INC` και να κατευθύνει την αναζήτηση σε συγκεκριμένη θέση στη συνέχεια. Αυτό είναι προχωρημένο χαρακτηριστικό· ο καθημερινός κώδικας δεν το αγγίζει ποτέ.

Δείτε επίσης

- `use`, `require` - οι καταναλωτές της `@INC`/`%INC`.
- `use lib` - η κανονική γραφή μεταγλώττισης για προσθήκη στην αρχή της `@INC`.
- `Getopt::Long` - ο τυπικός αναλυτής για τη `@ARGV`.
- `readline` - η μορφή συνάρτησης του τελεστή «διαμάντι» `<>` που κινεί το φιλτράρισμα `@ARGV → STDIN`.
- `-i`, `-n`, `-p`, `-a` - διακόπτες γραμμής εντολών χτισμένοι πάνω στη `@ARGV`.
- *Μεταβλητές διεργασίας* - η `$0` είναι το όνομα σεναρίου· η `$$` είναι το PID· ζευγαρώστε τα με τη `@ARGV` για αυτο-επανεκκίνηση και καταγραφή.
- *Μονόγραμμα γραμμής εντολών* - ο οδηγός που δείχνει κάθε συνδυασμό `@ARGV`/`<>`/διακόπτη σε πλαίσιο.

Χειρισμός σημάτων - `%SIG`

Η `%SIG` είναι το hash όπου η Perl εκθέτει τους χειριστές σημάτων. Η ρύθμιση `$SIG{NAME}` εγκαθιστά χειριστή για το σήμα `NAME`· η διαγραφή του επαναφέρει την προεπιλεγμένη ενέργεια. Δύο ψευδο-σήματα - `__WARN__` και `__DIE__` - δεν είναι καθόλου πραγματικά σήματα OS αλλά hooks του διερμηνέα στις `warn` και `die`.

Ο πίνακας σημάτων

Τα πραγματικά σήματα OS προέρχονται από τις λίστες POSIX και `signal.h`. Τα κοινά - κλειδιά που θα χρησιμοποιήσετε πραγματικά:

Κλειδί	Σήμα	Προεπιλεγμένη ενέργεια	Πότε το χειρίζεστε
INT	SIGINT	τερματισμός	Ctrl-C: ομαλός τερματισμός
TERM	SIGTERM	τερματισμός	`kill <pid>`: ομαλός τερματισμός
HUP	SIGHUP	τερματισμός	αποσύνδεση τερματικού· οι δαίμονες επαναφορτώνουν ρύθμιση
QUIT	SIGQUIT	τερματισμός + core	Ctrl-\: όπως INT αλλά με core dump
PIPE	SIGPIPE	τερματισμός	εγγραφή σε κλειστό σωλήνα· συνήθως αγνοείται
CHLD	SIGCHLD	αγνόηση	πέθανε μια διεργασία-παιδί (`waitpid` εδώ)
USR1	SIGUSR1	τερματισμός	γεγονός ορισμένο από εφαρμογή
USR2	SIGUSR2	τερματισμός	γεγονός ορισμένο από εφαρμογή
ALRM	SIGALRM	τερματισμός	ο χρονομετρητής `alarm()` έληξε
WINCH	SIGWINCH	αγνόηση	αλλαγή μεγέθους παραθύρου τερματικού
IO	SIGIO	τερματισμός	ασύγχρονο I/O έτοιμο

Η πλήρης λίστα - συμπεριλαμβανομένων σημάτων που δεν μπορείτε να πιάσετε (KILL, STOP) και σημάτων ειδικών πλατφόρμας - προέρχεται από το άρθρωμα POSIX, που εξάγει τις αριθμητικές σταθερές (SIGINT, SIGTERM, ...) και τις μακροεντολές `wait` της οικογένειας WIFEXITED.

Εγκατάσταση χειριστή

Γίνονται αποδεκτές τρεις μορφές τιμής χειριστή:

```
$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
```

Μη χρησιμοποιείτε *bareword* (`$SIG{INT} = 'handler'`) - αυτό αναφέρεται σε *όνομα* στο πακέτο `main::`, που η Perl αναζητά κάθε φορά που φτάνει το σήμα. Αν μετονομάσετε τη handler, η μορφή bareword σιωπηρά συνεχίζει να στοχεύει το παλιό όνομα.

Ο χειριστής λαμβάνει ένα όρισμα - το όνομα σήματος ως συμβολοσειρά:

```
sub handler {
    my ($signame) = @_;
    print "received SIG$signame\n";
}
$SIG{INT}  = \&handler;
$SIG{TERM} = \&handler;
```

Ασφαλή σήματα - ο κανόνας αναβαλλόμενης παράδοσης

Από την Perl 5.8, η παράδοση σημάτων είναι **αναβαλλόμενη**. Όταν φτάσει ένα σήμα, το OS θέτει μια σημαία στον διερμηνέα· ο πραγματικός χειριστής Perl εκτελείται μόνο στο επόμενο *ασφαλές σημείο* - τυπικά μεταξύ εκτελέσεων εντολών. Αυτό σημαίνει:

- Ένας χειριστής **δεν** μπορεί να διακόψει μια `read`, `open`, `connect`, ή οποιαδήποτε άλλη κλήση συστήματος στη μέση. Οι αργές κλήσεις συστήματος επιστρέφουν πρόωρα με `EINTR` στη `$!` και ο χειριστής εκτελείται μετά.
- Ένας χειριστής **δεν** μπορεί να διακόψει μια αντιστοίχιση regex σε εξέλιξη. Ένα μακροχρόνιο regex σε κακόβουλη είσοδο δεν θα κοπεί σύντομα από SIGINT.
- Ο χειριστής εκτελείται σε κανονικό περιβάλλον Perl. Μπορεί να καλέσει οποιονδήποτε κώδικα Perl, να δεσμεύσει μνήμη, να εκτοξεύσει εξαιρέσεις - κανένας κίνδυνος επανεισόδου χειριστών σημάτων C.

Αυτή είναι η σωστή ισορροπία για σχεδόν κάθε πρόγραμμα. Αν πραγματικά χρειάζεστε χειρισμό σημάτων *μη ασφαλή* προ-5.8 - χειριστές που εκτελούνται σύγχρονα στο περιβάλλον σήματος, με όλους τους κινδύνους επανεισόδου - θέστε τη μεταβλητή περιβάλλοντος `PERL_SIGNALS=unsafe` πριν ξεκινήσετε την Perl. Μην το κάνετε αυτό αβασάνιστα.

Η PetaPerl υλοποιεί χειρισμό σημάτων αναβαλλόμενης παράδοσης.

Το μοτίβο ομαλού τερματισμού

Η τυπική μορφή: μια σημαία που τίθεται από τον χειριστή, ελέγχεται από τον κύριο βρόχο:

```
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;
}
```

Ο κύριος βρόχος ελέγχει τη σημαία σε σημείο όπου είναι ασφαλές να ξετυλιχθεί καθαρά. Ο χειριστής κάνει *μόνο* «θέτω τη σημαία» - χωρίς καταγραφή, χωρίς I/O - ελαχιστοποιώντας τον χρόνο εκτέλεσής του.

Για εφαρμογές που μπλοκάρουν σε μεγάλη κλήση συστήματος (`accept`, `read`, `select`), ο βρόχος πρέπει να δομηθεί ώστε να επαναλαμβάνει σε `EINTR`:

```
my $shutting_down = 0;
$SIG{INT}  = sub { $shutting_down = 1 };
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
while (!$shutting_down) {
    my $client = accept($srv, $sock);
    unless (defined $client) {
        next if $!{EINTR};        # signal woke us up - re-check
↪flag
        die "accept: $!";
    }
    handle($client);
}
```

Η `$!{EINTR}` (δείτε *μεταβλητές σφάλματος*) σας λέει ότι η κλήση συστήματος εξήλθε επειδή έφτασε ένα σήμα. Η σημαία τέθηκε από τον χειριστή· ο κύριος βρόχος βλέπει την αφύπνιση, ξαναελέγχει τη `$shutting_down`, και εξέρχεται καθαρά.

Συγκομιδή διεργασιών-παιδιών - SIGCHLD

Όταν μια διεργασία-παιδί εξέρχεται, ο γονέας λαμβάνει SIGCHLD. Ο χειριστής πρέπει να καλέσει τη `waitpid` για να *συγκομίσει* το zombie:

```
$SIG{CHLD} = sub {
    while ((my $kid = waitpid(-1, WNOHANG)) > 0) {
        my $status = $?;
        log_child_death($kid, $status);
    }
};
```

Ο βρόχος `while` με `WNOHANG` (από τη `POSIX :sys_wait_h`) χειρίζεται την περίπτωση όπου δύο παιδιά εξέρχονται κοντά μεταξύ τους - παραδίδεται μόνο ένα SIGCHLD (δεν μπαίνει σε ουρά), οπότε ο χειριστής πρέπει να εκκενώσει *όλες* τις εκκρεμείς εξόδους κάθε φορά που εκτελείται.

Η `$SIG{CHLD} = 'IGNORE'` είναι ειδική περίπτωση: δεν αγνοεί απλώς το σήμα, λέει στον πυρήνα να συγκομίζει αυτόματα τα παιδιά. Χρησιμοποιήστε το όταν το σενάριό σας κάνει fork παιδιά των οποίων η κατάσταση εξόδου δεν σας ενδιαφέρει.

SIGPIPE - η περίπτωση κλειστού σωλήνα

Όταν κάνετε `print` σε σωλήνα του οποίου ο αναγνώστης έχει φύγει, ο πυρήνας στέλνει SIGPIPE. Η προεπιλεγμένη ενέργεια είναι τερματισμός της διεργασίας - κάτι που σπάνια θέλετε. Τα περισσότερα προγράμματα φίλτρου πρέπει να αγνοούν SIGPIPE και να ελέγχουν αντ" αυτού την τιμή επιστροφής `print`:

```
local $SIG{PIPE} = 'IGNORE';

print {$child_pipe} $data
    or die "write to pipe: $!";  # $! will be EPIPE if the reader
↪closed
```

Με αγνοημένο τον χειριστή, η κλήση συστήματος επιστρέφει αποτυχία με `EPIPE` αντί να σκοτώσει το πρόγραμμα.

Χρονικά όρια μέσω SIGALRM

Σε συνδυασμό με τη `alarm` και ένα μπλοκ `eval`, ο SIGALRM παρέχει φορητό χρονικό όριο για κάθε λειτουργία που μπλοκάρει:

```
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> });
```

Το μοτίβο:

1. `local $SIG{ALRM}` - ο χειριστής έχει εμβέλεια μόνο για αυτή την κλήση.
2. `alarm $seconds` - προγραμματισμός του σήματος.
3. Εκτέλεση της εργασίας που μπλοκάρει.
4. `alarm 0` - ακύρωση του χρονομετρητή αν η εργασία πέτυχε.
5. Μέσα στον χειριστή, `die "TIMEOUT\n"` - η τελική `\n` καταστέλλει το επίθεμα «at line N» και κάνει την τιμή σφάλματος εύκολη στον έλεγχο.

`__WARN__` και `__DIE__` - hooks διερμηνέα

Αυτά τα δύο κλειδιά στη %SIG *δεν* είναι σήματα OS. Καλούνται όταν ο διερμηνέας πρόκειται να εκπέμψει μια προειδοποίηση (`warn`) ή να εκτοξεύσει μια μοιραία εξαίρεση (`die`).

`$SIG{__WARN__}` - υποκλοπή προειδοποιήσεων

```
my @warnings;
{
    local $SIG{__WARN__} = sub {
        my ($msg) = @_;
        push @warnings, $msg;
    };
    do_thing_that_might_warn();
}
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
# Now @warnings holds every warning that fired inside the block,
# without anything having printed to STDERR.
```

Ένας χειριστής __WARN__ που δεν κάνει τίποτα αποσιωπά εντελώς την προειδοποίηση. Ένας χειριστής που καλεί die αναβαθμίζει τις προειδοποιήσεις σε εξαιρέσεις:

```
local $SIG{__WARN__} = sub { die $_[0] };
eval { something_warny() };       # warns become catchable
```

Η λεξιλογική pragma use warnings FATAL => 'all' είναι πιο στοχευμένος τρόπος να γίνει αυτό - κάνει τις προειδοποιήσεις μοιραίες μόνο στη λεξιλογική εμβέλεια όπου η pragma είναι ενεργή. Χρησιμοποιήστε τη αντί για __WARN__ για απαιτήσεις «αντιμετώπισε τις προειδοποιήσεις ως σφάλματα».

$SIG{__DIE__} - υποκλοπή die

Ένας χειριστής __DIE__ εκτελείται όταν η die πρόκειται να εκτοξεύσει - τόσο για μη πιασμένες εξαιρέσεις όσο και μέσα σε eval. Μέσα σε eval, ο χειριστής εκτελείται *πριν* πιαστεί η εξαίρεση, κάτι που είναι η κλασική πηγή σύγχυσης: ένας καθολικός χειριστής __DIE__ που καταγράφει κάθε «θάνατο» θα ενεργοποιηθεί σε κάθε eval { ... } που καλεί die για κανονική ροή ελέγχου.

```
$SIG{__DIE__} = sub { print STDERR "dying: $_[0]" };
eval { die "expected error" };    # the handler still runs!
```

Αν θέλετε καθολική καταγραφή θανάτων, φιλτράρετε μέσω $^S:

```
$SIG{__DIE__} = sub {
    return if $^S;                # inside eval - let it be caught
    print STDERR "uncaught: $_[0]";
};
```

Ρεαλιστικά, **αποφύγετε τη $SIG{__DIE__} σε νέο κώδικα**. Η pragma use warnings και οι *μεταβλητές σφάλματος* καλύπτουν ό,τι χρειάζεστε χωρίς εκπλήξεις. Το σύγχρονο χαρακτηριστικό try/catch χειρίζεται εξαιρέσεις ρητά χωρίς τη δράση εξ αποστάσεως της $SIG{__DIE__}.

Εμφάνιση διαθέσιμων σημάτων

```
use Config;
my @signals = split / /, $Config{sig_name};
print "available signals: @signals\n";
# HUP INT QUIT ILL TRAP ABRT BUS FPE KILL ...
```

Ή, για την αντιστοίχιση αριθμός→όνομα:

```
use Config;
my @names = split / /, $Config{sig_name};
my @nums  = split / /, $Config{sig_num};
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
my %sig_num  = map { $names[$_] => $nums[$_] }  0..$#names;
my %sig_name = map { $nums[$_]  => $names[$_] } 0..$#names;
```

Αυτό κάνουν εσωτερικά τα περισσότερα εργαλεία εμφάνισης σημάτων.

Δείτε επίσης

- `POSIX` - η κανονική πηγή αριθμητικών σταθερών `SIG*` και μακροεντολών `WIFEXITED`/`WEXITSTATUS` για αποκωδικοποίηση κατάστασης παιδιού σε χειριστή SIGCHLD.
- `kill` - ο σύντροφος της `%SIG`: αποστολή σήματος σε διεργασία (τυπικά για δοκιμή του δικού σας χειριστή).
- `alarm` - προγραμματισμός SIGALRM· η βάση μοτίβων χρονικού ορίου.
- `waitpid` - συγκομιδή παιδιού μετά από SIGCHLD.
- `fork` - παράγει τα παιδιά των οποίων τους θανάτους αναφέρει η SIGCHLD.
- *Μεταβλητές σφάλματος* - η `$!` μετά από `EINTR`, η `$?` μετά από `waitpid`, η `$@` μετά από `die` μέσα από χειριστή.
- *Ενδοσκόπηση διερμηνέα* - η `$^S` για τον έλεγχο `__DIE__` «βρισκόμαστε μέσα σε `eval`».
- `try/catch` - η σύγχρονη εναλλακτική στον χειρισμό εξαιρέσεων βασισμένο σε `$SIG{__DIE__}`.
- *Προεπιλεγμένες μεταβλητές* - `$_`, `@_`. Ο σιωπηρός τελεστέος των περισσότερων ενσωματωμένων· ο πίνακας ορισμάτων κάθε sub.
- *Ταυτότητα διεργασίας* - `$0`, `$$`, `$<`, `$>`, `$(`, `$)`. Όνομα προγράμματος, PID, πραγματικά και effective UIDs/GIDs.
- *Κατάσταση σφάλματος* - `$!`, `$@`, `$?`, `$^E`. Οι τέσσερις μεταβλητές σφάλματος, κάθε μία αναφέρει από διαφορετικό στρώμα (βιβλιοθήκη C, eval/`die`, διεργασία-παιδί, επεκτεταμένο OS).
- *Κατάσταση filehandle και εκτύπωσης* - `$/`, `$\`, `$,`, `$"`, `$|`, `$.`, `$;`. Διαχωριστής εγγραφών εισόδου, διαχωριστές εξόδου, αυτόματη εκκένωση, μετρητής γραμμών, ενωτικό δεικτών.
- *Μεταβλητές αντιστοίχισης regex* - `$&`, `$\``, `$'`, `$1..$N`, `%+`, `%-`, `@+`, `@-`. Το αποτέλεσμα της τελευταίας επιτυχημένης αντιστοίχισης· η ιστορία απόδοσης πίσω από τη `$&` και τις σχετικές.
- *Ενδοσκόπηση διερμηνέα* - `$]`, `$^V`, `$^O`, `$^X`, `$^T`, `$^W`, `$^I`, `$^P`. Έκδοση, OS, διαδρομή εκτελέσιμου, χρόνος βάσης, προειδοποιήσεις, επιτόπια επεξεργασία, σημαίες αποσφαλματωτή.
- *Είσοδος προγράμματος* - `@ARGV`, `%ENV`, `@INC`, `%INC`, `$0`. Πώς το σενάριο λαμβάνει τα ορίσματά του, το περιβάλλον, και τη διαδρομή αναζήτησης αρθρωμάτων· η ιστορία `use lib` έναντι `unshift @INC`.

- *Χειρισμός σημάτων* - %SIG, $SIG{__WARN__}, $SIG{__DIE__}. Τα σήματα που μπορείτε να πιάσετε, ο κανόνας ασφαλών σημάτων, και η τυπική μορφή ομαλού τερματισμού.

Χώροι ονομάτων μεταβλητών

Κάθε ειδική μεταβλητή ζει σε **έναν** συγκεκριμένο χώρο ονομάτων. Οι περισσότερες βρίσκονται στο `main::` μερικές (@F, @ISA) είναι τοπικές πακέτου. Η τεκμηρίωση κάθε μεταβλητής αναφέρει ποιον.

Η σύμβαση ονομάτων είναι μια επιλογή σημείου στίξης και ένα sigil:

- $<digit> - μεταβλητές σύλληψης regex.
- $<symbol> - μεταβλητές ενός χαρακτήρα ($_, $!, $@, $/, ...).
- $^<letter> - μεταβλητές γράμματος ελέγχου ($^V, $^W, $^X). Γράφονται με κυριολεκτικό caret ακολουθούμενο από κεφαλαίο γράμμα.
- ${^NAME} - μεταβλητές εκτεταμένου ονόματος (${^GLOBAL_PHASE}, ${^CAPTURE}). Οι αγκύλες και το caret είναι απαραίτητα.
- $NAME μετά από use English - το ονομαστικό ψευδώνυμο για σχεδόν κάθε μεταβλητή στίξης. Η $ERRNO είναι η $!, η $EVAL_ERROR είναι η $@, κ.λπ.

Το άρθρωμα English

```
use English;            # all aliases, plus $PREMATCH/$MATCH/
 ↪$POSTMATCH
use English qw( -no_match_vars );    # skip $`, $&, $' aliases
```

Η English παρέχει αναγνώσιμα ψευδώνυμα για τις μεταβλητές στίξης: $EVAL_ERROR για τη $@, $INPUT_RECORD_SEPARATOR για τη $/, και ούτω καθεξής. Τα ψευδώνυμα είναι πραγματικές μεταβλητές Perl που μοιράζονται αποθήκευση με τα πρωτότυπα - η ανάθεση σε ένα γράφει στο άλλο.

Η εισαγωγή -no_match_vars συνιστάται σε κώδικα που δεν χρειάζεται ρητά τα $&, $`, ή $'. Δείτε *μεταβλητές αντιστοίχισης regex* για τον ιστορικό λόγο απόδοσης και γιατί δεν έχει πλέον σημασία σε σύγχρονες εκδόσεις perl.

Αυτή η σελίδα αναφέρει τις μορφές στίξης παντού επειδή ο περισσότερος κώδικας στην πράξη γράφεται έτσι. Τα ψευδώνυμα English σημειώνονται δίπλα στις σελίδες οικογένειας όπου είναι χρήσιμα.

Τοπικοποίηση ειδικών μεταβλητών

Σχεδόν κάθε ειδική μεταβλητή που ελέγχει τη συμπεριφορά μιας ενσωματωμένης ($/, $\, $,, $", $|, $_) είναι καθολική - θέστε την σε ένα σημείο και κάθε άλλο κομμάτι κώδικα στην ίδια διεργασία βλέπει την αλλαγή. Το τυπικό ιδίωμα για «χρειάζομαι διαφορετική τιμή μόνο γι" αυτό το μπλοκ» είναι η `local`:

```
sub slurp {
    my ($path) = @_;
    open my $fh, '<', $path or die "open $path: $!";
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    local $/;                               # slurp mode, only inside this␣
↪sub
    return <$fh>;
}
```

Χωρίς τη `local`, η `$/` του καλούντος θα αλλοιωνόταν μόνιμα. Η `local` αποθηκεύει την τρέχουσα τιμή, θέτει τη νέα, και επαναφέρει την παλιά τιμή όταν εξέλθει η περικλείουσα εμβέλεια - ακόμα και σε εξαίρεση. Αυτό ισχύει για κάθε ειδική μεταβλητή στις σελίδες οικογένειας, εκτός από τις μόνο-ανάγνωσης (μεταβλητές αντιστοίχισης regex, τυπικό όρισμα της `$0`, μεταβλητές έκδοσης).

Δείτε επίσης

- `perlop` - κάθε τελεστής που διαβάζει ή γράφει μια ειδική μεταβλητή ως μέρος του ορισμού του (σύνδεση regex, εκτύπωση, εύρος, σύγκριση).
- `perlfunc` - ενσωματωμένες συναρτήσεις, σχεδόν όλες οι οποίες διαβάζουν μια προεπιλογή από μία από αυτές τις μεταβλητές όταν κληθούν χωρίς όρισμα (`chomp`, `chop`, `print`, `lc`, `length`, `<>`, ...).
- *Regular expressions guide*
  - the regex language behind the match variables.
- *Λογική Boole για προγραμματιστές της Perl*
  - ο εννοιολογικός σύντροφος για τα ιδιώματα εξ ορισμού τιμών `||///` που επαναλαμβάνονται σε ολόκληρη αυτή την αναφορά.

5.1.2 Ενσωματωμένες συναρτήσεις

Αλφαβητικό ευρετήριο όλων των ενσωματωμένων της Perl 5. Το περιεχόμενο αντικατοπτρίζει το `perldoc perlfunc` με τις αποκλίσεις του pperl να επισημαίνονται μέσα σε κάθε σελίδα λεπτομερειών.

Για εναλλακτική περιήγηση ομαδοποιημένη ανά τομέα προβλήματος, δείτε τις *Ενσωματωμένες συναρτήσεις ανά κατηγορία*.

A

- **`abs`** - Επιστρέφει την απόλυτη τιμή ενός αριθμού.
- **`accept`** - Δέχεται μια εισερχόμενη σύνδεση σε υποδοχή που ακούει.
- **`alarm`** - Προγραμματίζει την παράδοση ενός `SIGALRM` στην τρέχουσα διεργασία μετά από ακέραιο αριθμό δευτερολέπτων πραγματικού χρόνου.
- **`all`** - Ελέγχει εάν το `BLOCK` επιστρέφει αληθές για **κάθε** στοιχείο της `LIST`.
- **`any`** - Επιστρέφει αληθές εάν το `BLOCK` δίνει αληθές για τουλάχιστον ένα στοιχείο της `LIST`.
- **`atan2`** - Τόξο εφαπτομένης του `Y/X` στο εύρος -π έως π.

B

- **bind** - Συνδέει μια τοπική διεύθυνση σε μια υποδοχή.
- **binmode** - Ορίζει τη στοίβα στρωμάτων I/O σε ένα filehandle - τυπικά για να παραδίδει ωμά bytes ή για να συνδέει μια κωδικοποίηση χαρακτήρων.
- **bless** - Σημαίνει αυτό που δείχνει μια αναφορά ως αντικείμενο ενός πακέτου.
- **break** - Έξοδος από ένα μπλοκ `given`.

C

- **caller** - Επιστρέφει πληροφορίες για την υπορουτίνα, την `eval` ή την `require` που κάλεσε τον τρέχοντα κώδικα.
- **catch** - Χειρίζεται μια εξαίρεση που έχει εγερθεί από προηγούμενο μπλοκ `try`.
- **chdir** - Αλλάζει τον τρέχοντα κατάλογο εργασίας της διεργασίας.
- **chmod** - Αλλάζει τα bits δικαιωμάτων μιας λίστας αρχείων.
- **chomp** - Αφαιρεί επιτόπου τον τερματικό διαχωριστή εγγραφών εισόδου από μια συμβολοσειρά.
- **chop** - Αφαιρεί τον τελευταίο χαρακτήρα μιας συμβολοσειράς και τον επιστρέφει.
- **chown** - Αλλάζει τον ιδιοκτήτη και την ομάδα μιας λίστας αρχείων.
- **chr** - Επιστρέφει τον χαρακτήρα του οποίου το σημείο κώδικα είναι ο δοσμένος αριθμός.
- **chroot** - Αλλάζει τον ριζικό κατάλογο της τρέχουσας διεργασίας και κάθε θυγατρικής που θα δημιουργήσει αργότερα.
- **class** - Δηλώνει έναν χώρο ονομάτων που συμπεριφέρεται ως εγγενής κλάση αντικειμένου.
- **close** - Κλείνει ένα filehandle, εκκενώνει τους ενταμιευτές του και απελευθερώνει τον υποκείμενο περιγραφέα αρχείου.
- **closedir** - Κλείνει ένα handle καταλόγου που είχε ανοιχτεί από την `opendir`.
- **connect** - Αρχίζει μια σύνδεση από μια υποδοχή προς απομακρυσμένη διεύθυνση.
- **continue** - Συνδέει σε έναν βρόχο ένα μπλοκ που εκτελείται μετά από κάθε επανάληψη, λίγο πριν επανελεγχθεί η συνθήκη.
- **cos** - Επιστρέφει το συνημίτονο ενός αριθμού δοσμένου σε ακτίνια.
- **crypt** - Μονόδρομος κατακερματισμός τύπου `passwd(5)` μιας συμβολοσειράς απλού κειμένου με salt.

D

- **dbmclose** - Διακόπτει τη σύνδεση ανάμεσα σε ένα αρχείο DBM και ένα hash που είχε προηγουμένως συνδεθεί με `dbmopen`.
- **dbmopen** - Συνδέει ένα αρχείο DBM στον δίσκο με ένα hash, ώστε οι αναγνώσεις και οι εγγραφές του hash να γίνονται αναζητήσεις και αποθηκεύσεις στη βάση δεδομένων.

- **defer** - Προγραμματίζει την εκτέλεση ενός μπλοκ όταν εξέρχεται η περικλείουσα εμβέλεια, για οποιοδήποτε λόγο.
- **defined** - Ελέγχει εάν μια τιμή, μεταβλητή ή υπορουτίνα είναι ορισμένη.
- **delete** - Αφαιρεί το ή τα ζεύγη κλειδιού-τιμής από ένα hash, ή στοιχείο/α από έναν πίνακα, και επιστρέφει αυτό που αφαιρέθηκε.
- **die** - Εγείρει μια εξαίρεση.
- **do** - Εκτελεί ένα μπλοκ κώδικα ή τρέχει ένα αρχείο πηγαίου κώδικα Perl σαν να ήταν μέρος του τρέχοντος προγράμματος.
- **dump** - Προκαλεί στο τρέχον πρόγραμμα την άμεση παραγωγή core dump.

E

- **each** - Διατρέχει ένα hash ή έναν πίνακα μία καταχώριση τη φορά.
- **endgrent** - Κλείνει τη βάση δεδομένων ομάδων μετά τη διάτρεξή της.
- **endhostent** - Κλείνει τη βάση δεδομένων υπολογιστών μετά τη διάτρεξη.
- **endnetent** - Κλείνει τη βάση δεδομένων δικτύων μετά τη διάτρεξη.
- **endprotoent** - Κλείνει τη βάση δεδομένων πρωτοκόλλων μετά τη διάτρεξη.
- **endpwent** - Κλείνει τον επαναλήπτη της βάσης passwd που έχει ανοιχτεί από τις `getpwent` ή `setpwent`.
- **endservent** - Κλείνει τη βάση δεδομένων υπηρεσιών μετά από διάτρεξη με την `getservent`.
- **eof** - Ελέγχει ένα filehandle για end-of-file.
- **eval** - Εκτελεί ένα κομμάτι κώδικα Perl παγιδεύοντας κάθε μοιραίο σφάλμα αντί να τερματιστεί το πρόγραμμα.
- **evalbytes** - Μεταγλωττίζει και εκτελεί μια συμβολοσειρά πηγαίου κώδικα Perl, εξαναγκάζοντας τον πηγαίο κώδικα να ερμηνευθεί ως **bytes** και όχι ως χαρακτήρες.
- **exec** - Εγκαταλείπει αυτό το πρόγραμμα και εκτελεί άλλο στην ίδια διεργασία.
- **exists** - Ελέγχει εάν ένα στοιχείο hash ή πίνακα, ή μια ονοματισμένη υπορουτίνα, υπάρχει - χωρίς να το δημιουργεί και χωρίς να ενδιαφέρεται τι περιέχει.
- **exit** - Τερματίζει το πρόγραμμα με μια τιμή κατάστασης.
- **exp** - Υψώνει το *e* σε δύναμη.

F

- **fc** - Επιστρέφει την casefolded μορφή Unicode μιας συμβολοσειράς για σύγκριση χωρίς διάκριση πεζών/κεφαλαίων.
- **fcntl** - Εκτελεί μια λειτουργία ελέγχου αρχείου `fcntl(2)` σε ένα filehandle.
- **field** - Δηλώνει μια μεταβλητή ανά στιγμιότυπο μέσα σε ένα μπλοκ `class`.
- **fileno** - Επιστρέφει τον αριθμό περιγραφέα αρχείου σε επίπεδο λειτουργικού συστήματος που βρίσκεται πίσω από ένα filehandle.

- **finally** - Εκτελεί κώδικα εκκαθάρισης κατά την έξοδο από `try`/`catch`, είτε το σώμα ολοκληρώθηκε επιτυχώς, είτε εγείρει εξαίρεση, είτε πραγματοποίησε άλμα προς τα έξω.
- **flock** - Τοποθετεί ένα συμβουλευτικό κλείδωμα σε ένα ανοιχτό αρχείο.
- **fork** - Δημιουργεί μια νέα διεργασία που εκτελεί το ίδιο πρόγραμμα στο ίδιο σημείο.
- **format** - Δηλώνει ένα πρότυπο αναφοράς βασισμένο σε εικόνα για χρήση από την `write`.
- **formline** - Μορφοποιεί μια λίστα τιμών σε συμβολοσειρά εικόνας και προσαρτά το αποτέλεσμα στον συσσωρευτή μορφοποίησης `$^A`.

G

- **getc** - Διαβάζει τον επόμενο μεμονωμένο χαρακτήρα από ένα filehandle.
- **getgrent** - Διαβάζει την επόμενη καταχώριση από τη βάση δεδομένων ομάδων του συστήματος.
- **getgrgid** - Αναζητά μια εγγραφή ομάδας με βάση τον αριθμητικό κωδικό ομάδας.
- **getgrnam** - Αναζητά μια ομάδα Unix με το όνομά της και επιστρέφει την εγγραφή `/etc/group` της.
- **gethostbyaddr** - Αναζητά μια εγγραφή υπολογιστή με βάση την πακεταρισμένη διεύθυνσή IP του.
- **gethostbyname** - Αναζητά μια εγγραφή υπολογιστή με βάση το όνομα DNS.
- **gethostent** - Ανακτά την επόμενη καταχώριση από τη βάση δεδομένων υπολογιστών.
- **getlogin** - Επιστρέφει το όνομα σύνδεσης του χρήστη που σχετίζεται με το ελέγχον τερματικό.
- **getnetbyaddr** - Αναζητά μια εγγραφή δικτύου με βάση την αριθμητική διεύθυνση δικτύου.
- **getnetbyname** - Αναζητά μια εγγραφή δικτύου με όνομα.
- **getnetent** - Διαβάζει την επόμενη καταχώριση από τη βάση δεδομένων δικτύων.
- **getpeername** - Επιστρέφει τη διεύθυνση του απομακρυσμένου άκρου μιας συνδεδεμένης υποδοχής.
- **getpgrp** - Επιστρέφει το POSIX αναγνωριστικό ομάδας διεργασιών στο οποίο ανήκει μια διεργασία.
- **getppid** - Επιστρέφει το αναγνωριστικό διεργασίας της γονικής της τρέχουσας διεργασίας.
- **getpriority** - Επιστρέφει την τρέχουσα τιμή nice χρονοπρογραμματισμού μιας διεργασίας, μιας ομάδας διεργασιών ή ενός χρήστη.
- **getprotobyname** - Αναζητά ένα πρωτόκολλο IP με το όνομά του σε κείμενο και επιστρέφει την καταχώρισή του στη βάση δεδομένων.
- **getprotobynumber** - Αναζητά καταχώριση πρωτοκόλλου δικτύου με βάση τον αριθμό πρωτοκόλλου που του έχει αντιστοιχιστεί.

- **getprotoent** - Ανακτά την επόμενη καταχώριση από τη βάση δεδομένων πρωτοκόλλων.
- **getpwent** - Επιστρέφει την επόμενη καταχώριση από τη βάση δεδομένων κωδικών του συστήματος.
- **getpwnam** - Αναζητά την εγγραφή passwd ενός χρήστη με βάση το όνομα σύνδεσης.
- **getpwuid** - Αναζητά την εγγραφή passwd ενός χρήστη με βάση το αριθμητικό UID.
- **getservbyname** - Αναζητά μια υπηρεσία δικτύου με βάση το όνομα κειμένου και το πρωτόκολλο.
- **getservbyport** - Αναζητά μια υπηρεσία δικτύου με βάση την αριθμητική θύρα και το πρωτόκολλο.
- **getservent** - Διαβάζει την επόμενη καταχώριση από τη βάση δεδομένων υπηρεσιών του συστήματος.
- **getsockname** - Επιστρέφει την τοπική διεύθυνση μιας συνδεδεμένης ή συνδεμένης υποδοχής.
- **getsockopt** - Διαβάζει μια επιλογή υποδοχής από τον πυρήνα ως αδιαφανή πακεταρισμένη συμβολοσειρά.
- **glob** - Επεκτείνει ένα μοτίβο ονόματος αρχείου τύπου shell στη λίστα των διαδρομών που αντιστοιχούν.
- **gmtime** - Μετατρέπει έναν χρόνο epoch σε αναλυμένο χρόνο UTC, είτε ως λίστα 9 στοιχείων είτε ως συμβολοσειρά τύπου `ctime(3)`.
- **goto** - Μεταφέρει την εκτέλεση αλλού στο πρόγραμμα χωρίς επιστροφή.
- **grep** - Φιλτράρει μια λίστα στα στοιχεία όπου το μπλοκ ή η έκφραση είναι αληθές.

H

- **hex** - Ερμηνεύει μια συμβολοσειρά ως δεκαεξαδικό αριθμό και επιστρέφει την αριθμητική τιμή της.

I

- **import** - Συμπληρώνει τον χώρο ονομάτων του καλούντος με ονόματα που το άρθρωμα επιλέγει να εξάγει.
- **index** - Βρίσκει τη θέση μιας υποσυμβολοσειράς μέσα σε μια συμβολοσειρά.
- **int** - Επιστρέφει το ακέραιο τμήμα ενός αριθμού, αποκόπτοντας προς το μηδέν.
- **ioctl** - Εκτελεί μια κλήση συστήματος ελέγχου συσκευής `ioctl(2)` σε ένα filehandle.
- **isa** - Ελέγχει εάν ένα αντικείμενο είναι στιγμιότυπο μιας κλάσης ή οποιασδήποτε υποκλάσης που προέρχεται από αυτήν.

J

- **join** - Συνενώνει μια λίστα συμβολοσειρών με έναν διαχωριστή ανάμεσα σε κάθε γειτονικό ζεύγος και επιστρέφει την ενιαία προκύπτουσα συμβολοσειρά.

K

- **keys** - Παραθέτει όλα τα κλειδιά ενός hash, ή όλους τους δείκτες ενός πίνακα.
- **kill** - Στέλνει ένα σήμα σε μια λίστα διεργασιών.

L

- **last** - Εξέρχεται άμεσα από έναν βρόχο, παραλείποντας το υπόλοιπο του σώματος και το μπλοκ `continue`.
- **lc** - Επιστρέφει αντίγραφο συμβολοσειράς με πεζά γράμματα.
- **lcfirst** - Επιστρέφει αντίγραφο συμβολοσειράς με τον πρώτο χαρακτήρα της σε πεζά.
- **length** - Επιστρέφει τον αριθμό των χαρακτήρων μιας συμβολοσειράς.
- **link** - Δημιουργεί έναν σκληρό σύνδεσμο από το `NEWFILE` προς το υπάρχον `OLDFILE`.
- **listen** - Σημαίνει μια υποδοχή ως παθητική, ώστε να μπορεί να δέχεται εισερχόμενες συνδέσεις.
- **local** - Αποθηκεύει την τρέχουσα τιμή μιας μεταβλητής πακέτου και την επαναφέρει όταν εξέρχεται η περικλείουσα εμβέλεια.
- **localtime** - Μετατρέπει έναν χρόνο epoch σε αναλυμένο ημερολογιακό χρόνο στην τοπική ζώνη ώρας.
- **lock** - Τοποθετεί ένα συμβουλευτικό κλείδωμα σε μια κοινόχρηστη μεταβλητή, πίνακα, hash ή υπορουτίνα μέχρι το κλείδωμα να βγει εκτός εμβέλειας.
- **log** - Επιστρέφει τον φυσικό λογάριθμο (βάση *e*) ενός αριθμού.
- **lstat** - Επιστρέφει τη λίστα κατάστασης 13 στοιχείων για μια διαδρομή **χωρίς** να ακολουθήσει συμβολικό σύνδεσμο.

M

- **m//** - Αναζητά ένα μοτίβο σε μια συμβολοσειρά και αναφέρει εάν - και τι - ταιριάζει.
- **map** - Εφαρμόζει ένα μπλοκ ή έκφραση σε κάθε στοιχείο μιας λίστας και επιστρέφει τα επίπεδα αποτελέσματα.
- **method** - Δηλώνει μια ονοματισμένη μέθοδο στιγμιοτύπου μέσα σε ένα μπλοκ `class`.
- **mkdir** - Δημιουργεί έναν μόνο κατάλογο στο σύστημα αρχείων.
- **msgctl** - Εκτελεί λειτουργία ελέγχου σε ουρά μηνυμάτων System V IPC.
- **msgget** - Δημιουργεί ή αναζητά μια ουρά μηνυμάτων System V IPC και επιστρέφει το id της.

- **msgrcv** - Λαμβάνει ένα μήνυμα από ουρά μηνυμάτων System V IPC.
- **msgsnd** - Στέλνει ένα μήνυμα σε ουρά μηνυμάτων System V IPC.
- **my** - Δηλώνει μία ή περισσότερες λεξιλογικά ορισμένες μεταβλητές.

N

- **next** - Ξεκινά αμέσως την επόμενη επανάληψη του περικλείοντος βρόχου.
- **no** - Η αντίστροφη του `use` κατά τη μεταγλώττιση - καλεί τη μέθοδο `unimport` ενός αρθρώματος για να απενεργοποιήσει ό,τι ενεργοποίησε η `use`.

O

- **oct** - Ερμηνεύει μια συμβολοσειρά ως αριθμό γραμμένο σε οκταδικό, δεκαεξαδικό ή δυαδικό σύστημα και επιστρέφει τον προκύπτοντα ακέραιο.
- **open** - Συσχετίζει ένα filehandle με ένα αρχείο, μια εντολή ή ένα βαθμωτό στη μνήμη.
- **opendir** - Ανοίγει έναν κατάλογο για ανάγνωση.
- **ord** - Επιστρέφει το σημείο κώδικα Unicode του πρώτου χαρακτήρα μιας συμβολοσειράς.
- **our** - Δηλώνει ένα λεξιλογικά ορισμένο ψευδώνυμο σε μια μεταβλητή πακέτου.

P

- **pack** - Μετατρέπει μια λίστα τιμών Perl σε δυαδική συμβολοσειρά σύμφωνα με ένα πρότυπο.
- **package** - Δηλώνει τον χώρο ονομάτων κατά τη μεταγλώττιση για τις δηλώσεις που ακολουθούν.
- **pipe** - Ανοίγει ένα ζεύγος συνδεδεμένων filehandles - ένα για ανάγνωση, ένα για εγγραφή.
- **pop** - Αφαιρεί και επιστρέφει το τελευταίο στοιχείο ενός πίνακα.
- **pos** - Αναφέρει ή ορίζει πού θα συνεχίσει η επόμενη αντιστοιχία regex με `/g` σε μια συμβολοσειρά.
- **print** - Γράφει μια λίστα τιμών σε ένα filehandle.
- **printf** - Γράφει μια μορφοποιημένη συμβολοσειρά σε ένα filehandle.
- **prototype** - Επιστρέφει τη συμβολοσειρά πρωτοτύπου μιας υπορουτίνας, ή `undef` εάν δεν έχει.
- **push** - Προσαρτά μία ή περισσότερες τιμές στο τέλος ενός πίνακα.

Q

- **q//** - Λεκτικό συμβολοσειράς μονών εισαγωγικών με επιλογή οριοθέτη.
- **qq//** - Κατασκευάζει συμβολοσειρά διπλών εισαγωγικών με παρεμβολή και οριοθέτη της επιλογής σας.

- **qr//** - Μεταγλωττίζει ένα μοτίβο μία φορά και επιστρέφει επαναχρησιμοποιήσιμο αντικείμενο regex.
- **quotemeta** - Επιστρέφει αντίγραφο συμβολοσειράς όπου κάθε χαρακτήρας σημαντικός για regex έχει διαφύγει με backslash, ώστε το αποτέλεσμα να μπορεί να παρεμβληθεί σε μοτίβο και να ταιριάζει με το κυριολεκτικό του περιεχόμενο.
- **qw//** - Κατασκευάζει λίστα από barewords διαχωρίζοντας μια οριοθετημένη συμβολοσειρά στα κενά, χωρίς εισαγωγικά ή κόμματα ανά λέξη.
- **qx//** - Εκτελεί μια εντολή shell και αιχμαλωτίζει την τυπική της έξοδο.

R

- **rand** - Επιστρέφει έναν ψευδοτυχαίο αριθμό κινητής υποδιαστολής στο ημίκλειστο εύρος `[0, EXPR)`.
- **read** - Διαβάζει σταθερή ποσότητα εισόδου με ενταμιευτή από ένα filehandle σε ένα βαθμωτό.
- **readdir** - Διαβάζει την επόμενη καταχώριση, ή όλες τις υπόλοιπες, από handle καταλόγου ανοιγμένο από την `opendir`.
- **readline** - Διαβάζει μία ή περισσότερες εγγραφές από ένα filehandle.
- **readlink** - Επιστρέφει τη διαδρομή στόχου που δείχνει ένας συμβολικός σύνδεσμος.
- **readpipe** - Εκτελεί μια εντολή shell και επιστρέφει την τυπική της έξοδο.
- **recv** - Διαβάζει ένα εισερχόμενο μήνυμα από μια υποδοχή σε ένα βαθμωτό.
- **redo** - Επανεκκινεί την τρέχουσα επανάληψη ενός βρόχου χωρίς να επανελέγξει τη συνθήκη και χωρίς να εκτελέσει το μπλοκ `continue`.
- **ref** - Επιστρέφει μια συμβολοσειρά που περιγράφει σε τι δείχνει μια αναφορά.
- **rename** - Αλλάζει το όνομα ενός αρχείου.
- **require** - Φορτώνει ένα αρχείο πηγαίου κώδικα Perl κατά τον χρόνο εκτέλεσης, ή απαιτεί ελάχιστη έκδοση Perl.
- **reset** - Καθαρίζει κάθε μεταβλητή πακέτου της οποίας το όνομα ξεκινά με ένα από ένα σύνολο γραμμάτων και επανοπλίζει αντιστοιχίσεις μιας χρήσης `m?pattern?`.
- **return** - Εξέρχεται από την τρέχουσα υπορουτίνα, `eval`, do `FILE`, μπλοκ `sort` ή μπλοκ regex eval, αποδίδοντας μια τιμή στον καλούντα.
- **reverse** - Αντιστρέφει μια λίστα - ή, σε βαθμωτό περιβάλλον, αντιστρέφει τους χαρακτήρες μιας συμβολοσειράς.
- **rewinddir** - Επαναφέρει ένα handle καταλόγου στην αρχή της λίστας του.
- **rindex** - Βρίσκει τη θέση της **τελευταίας** εμφάνισης μιας υποσυμβολοσειράς μέσα σε μια συμβολοσειρά.
- **rmdir** - Αφαιρεί έναν κενό κατάλογο.

S

- **s///** - Αναζητά ένα μοτίβο σε μια συμβολοσειρά και αντικαθιστά κάθε αντιστοιχία με την αντικατάσταση.
- **say** - Τυπώνει μια λίστα τιμών ακολουθούμενη από newline.
- **scalar** - Εξαναγκάζει την EXPR να αξιολογηθεί σε βαθμωτό περιβάλλον και επιστρέφει την τιμή της.
- **seek** - Επανατοποθετεί ένα filehandle για αναγνώσεις ή εγγραφές τυχαίας πρόσβασης.
- **seekdir** - Επαναφέρει ένα handle καταλόγου σε θέση που είχε προηγουμένως καταγραφεί από την telldir.
- **select** - Είτε ορίζει το προεπιλεγμένο filehandle εξόδου, είτε καλεί την κλήση συστήματος select(2) για πολυπλεξία I/O. Ίδιο όνομα, δύο άσχετες εργασίες - διακρίνονται από τον αριθμό ορισμάτων.
- **semctl** - Εκτελεί λειτουργία ελέγχου σε σύνολο σηματοφόρων System V.
- **semget** - Δημιουργεί ή αναζητά ένα σύνολο σηματοφόρων System V και επιστρέφει το αναγνωριστικό του.
- **semop** - Εκτελεί μία ή περισσότερες λειτουργίες σηματοφόρου System V ατομικά.
- **send** - Στέλνει ένα μήνυμα σε υποδοχή.
- **setgrent** - Επαναφέρει τον επαναλήπτη της βάσης δεδομένων ομάδων στην πρώτη καταχώριση.
- **sethostent** - Ανοίγει ή επαναφέρει τη βάση δεδομένων υπολογιστών για διάτρεξη.
- **setnetent** - Ανοίγει ή επαναφέρει τη βάση δεδομένων δικτύων για διάτρεξη.
- **setpgrp** - Ορίζει την ομάδα διεργασιών μιας διεργασίας.
- **setpriority** - Ορίζει την προτεραιότητα χρονοπρογραμματισμού (τιμή nice) μιας διεργασίας, ομάδας διεργασιών ή χρήστη.
- **setprotoent** - Ανοίγει τη βάση δεδομένων πρωτοκόλλων και την προετοιμάζει για διαδοχικές αναγνώσεις.
- **setpwent** - Επαναφέρει τον επαναλήπτη της βάσης κωδικών ώστε η επόμενη κλήση getpwent να επιστρέψει ξανά την πρώτη καταχώριση.
- **setservent** - Επαναφέρει τη βάση δεδομένων υπηρεσιών και προαιρετικά την κρατά ανοιχτή ανάμεσα στις αναζητήσεις.
- **setsockopt** - Ορίζει μια επιλογή σε επίπεδο πυρήνα σε μια ανοιχτή υποδοχή.
- **shift** - Αφαιρεί και επιστρέφει το πρώτο στοιχείο ενός πίνακα.
- **shmctl** - Ελέγχει ή ερωτά ένα τμήμα κοινόχρηστης μνήμης System V.
- **shmget** - Δημιουργεί ή αναζητά ένα τμήμα κοινόχρηστης μνήμης System V και επιστρέφει το αναγνωριστικό του.
- **shmread** - Αντιγράφει bytes από τμήμα κοινόχρηστης μνήμης System V σε ένα βαθμωτό της Perl.
- **shmwrite** - Αντιγράφει bytes σε τμήμα κοινόχρηστης μνήμης System V.

- **shutdown** - Τερματίζει μία κατεύθυνση σύνδεσης υποδοχής, ή και τις δύο.
- **sin** - Επιστρέφει το ημίτονο ενός αριθμού δοσμένου σε ακτίνια.
- **sleep** - Παύει τη διεργασία για ακέραιο αριθμό δευτερολέπτων.
- **socket** - Δημιουργεί ένα filehandle υποδοχής.
- **socketpair** - Δημιουργεί ένα ανώνυμο, συνδεδεμένο ζεύγος υποδοχών που επικοινωνούν μεταξύ τους.
- **sort** - Ταξινομεί μια λίστα και επιστρέφει την ταξινομημένη λίστα.
- **splice** - Αφαιρεί και/ή αντικαθιστά επιτόπου μια φέτα ενός πίνακα και επιστρέφει τα στοιχεία που αφαιρέθηκαν.
- **split** - Κόβει μια συμβολοσειρά σε λίστα πεδίων χρησιμοποιώντας έναν διαχωριστή regex.
- **sprintf** - Κατασκευάζει μια μορφοποιημένη συμβολοσειρά από πρότυπο μορφοποίησης και λίστα τιμών.
- **sqrt** - Επιστρέφει τη μη αρνητική τετραγωνική ρίζα της `EXPR`.
- **srand** - Σπείρει τη γεννήτρια ψευδοτυχαίων αριθμών.
- **stat** - Λαμβάνει τις πληροφορίες κατάστασης ενός αρχείου.
- **state** - Δηλώνει μια λεξιλογικά ορισμένη μεταβλητή της οποίας η τιμή διατηρείται μεταξύ κλήσεων της περικλείουσας υπορουτίνας.
- **study** - Μια κενή λειτουργία που διατηρείται για συμβατότητα πηγαίου κώδικα με παλαιότερο κώδικα Perl.
- **sub** - Δηλώνει ή ορίζει μια υπορουτίνα.
- **substr** - Εξάγει, αντικαθιστά ή ορίζει ψευδώνυμο σε μια συνεχή φέτα συμβολοσειράς.
- **symlink** - Δημιουργεί έναν συμβολικό σύνδεσμο στο `NEWFILE` που δείχνει στη συμβολοσειρά `OLDFILE`.
- **syscall** - Επικαλείται μια ωμή κλήση συστήματος με τον αριθμό πυρήνα της, περνώντας τα υπόλοιπα ορίσματα ως `int` ή ως δείκτη σε ενταμιευτή συμβολοσειράς.
- **sysopen** - Ανοίγει ένα αρχείο με τον χαμηλού επιπέδου τρόπο, περνώντας ένα ακέραιο bitmask `MODE` απευθείας στην υποκείμενη κλήση συστήματος `open(2)`.
- **sysread** - Διαβάζει ωμά bytes από ένα filehandle καλώντας την υποκείμενη κλήση συστήματος `read(2)`.
- **sysseek** - Επανατοποθετεί ένα filehandle σε επίπεδο συστήματος, παρακάμπτοντας τον ενταμιευτή PerlIO.
- **system** - Εκτελεί ένα ξεχωριστό πρόγραμμα και περιμένει να ολοκληρωθεί.
- **syswrite** - Γράφει bytes σε ένα filehandle με την ωμή κλήση συστήματος `write(2)`, παρακάμπτοντας το I/O με ενταμιευτή της Perl.

T

- **tell** - Επιστρέφει την τρέχουσα θέση σε bytes ενός filehandle.
- **telldir** - Επιστρέφει την τρέχουσα θέση ανάγνωσης ενός handle καταλόγου ως αδιαφανές διακριτικό.
- **tie** - Συνδέει μια μεταβλητή με μια κλάση, ώστε κάθε πρόσβαση στη μεταβλητή να αποστέλλεται μέσω των μεθόδων της κλάσης.
- **tied** - Επιστρέφει το αντικείμενο που υποστηρίζει μια tied μεταβλητή.
- **time** - Επιστρέφει τον τρέχοντα πραγματικό χρόνο ως ακέραιο αριθμό δευτερολέπτων από το epoch του συστήματος.
- **times** - Αναφέρει τον χρόνο CPU που έχει καταναλώσει αυτή η διεργασία και οι τερματισμένες θυγατρικές της.
- **tr///** - Αντικατάσταση χαρακτήρα-προς-χαρακτήρα. Η `tr` σαρώνει μια συμβολοσειρά και αντικαθιστά κάθε εμφάνιση χαρακτήρα της `SEARCHLIST` με τον θεσικά αντίστοιχο χαρακτήρα της `REPLACEMENTLIST`, επιστρέφοντας τον αριθμό των χαρακτήρων που έθιξε.
- **truncate** - Συντομεύει (ή επεκτείνει) ένα αρχείο σε ακριβές μήκος bytes.
- **try** - Εκτελεί ένα μπλοκ και εκτρέπει κάθε εξαίρεση που εγείρει σε ένα μπλοκ `catch`, με προαιρετικό μπλοκ `finally` που εκτελείται πάντα στην έξοδο.

U

- **uc** - Επιστρέφει αντίγραφο συμβολοσειράς με κεφαλαία γράμματα.
- **ucfirst** - Επιστρέφει αντίγραφο συμβολοσειράς με τον πρώτο χαρακτήρα της σε titlecase.
- **umask** - Ορίζει ή διαβάζει τη μάσκα κατάστασης δημιουργίας αρχείων της διεργασίας.
- **undef** - Η ακαθόριστη τιμή και ο τελεστής που την παράγει.
- **unlink** - Αφαιρεί μία ή περισσότερες καταχωρίσεις καταλόγου που ονομάζουν αρχεία.
- **unpack** - Εξάγει τυποποιημένες τιμές από δυαδική ή σταθερού πλάτους συμβολοσειρά σύμφωνα με ένα πρότυπο.
- **unshift** - Προσθέτει μία ή περισσότερες τιμές στην αρχή ενός πίνακα και επιστρέφει το νέο μήκος του πίνακα.
- **untie** - Διακόπτει τη σύνδεση ανάμεσα σε μια μεταβλητή και την tied κλάση της.
- **use** - Φορτώνει ένα άρθρωμα κατά τη μεταγλώττιση και εισάγει τα σύμβολά του στο τρέχον πακέτο.
- **utime** - Ορίζει χρόνους πρόσβασης και τροποποίησης σε μια λίστα αρχείων.

V

- **values** - Παραθέτει κάθε τιμή ενός hash (ή κάθε στοιχείο ενός πίνακα).
- **vec** - Διαβάζει ή γράφει μια θέση σταθερού πλάτους μέσα σε συμβολοσειρά που αντιμετωπίζεται ως πακεταρισμένο διάνυσμα bit.

W

- **wait** - Μπλοκάρει μέχρι να εξέλθει κάποια θυγατρική διεργασία και τη συγκομίζει.
- **waitpid** - Περιμένει να τερματιστεί μια συγκεκριμένη θυγατρική διεργασία και τη συγκομίζει.
- **wantarray** - Αναφέρει το περιβάλλον κλήσης της τρέχουσας υπορουτίνας.
- **warn** - Εκπέμπει μια προειδοποίηση στο STDERR.
- **write** - Αποτυπώνει μία εγγραφή σε ένα filehandle μέσω του συσχετισμένου του `format`.

Y

- **y///** - Μεταγραμματίζει χαρακτήρες σε μια συμβολοσειρά. Το y/// είναι συνώνυμο της `tr///` - οι δύο είναι ταυτόσημες από κάθε άποψη.

__ (διπλή υπογράμμιση)

- **__CLASS__** - Επιστρέφει το όνομα κλάσης του στιγμιοτύπου επί του οποίου ενεργείται.
- **__FILE__** - Το όνομα του αρχείου πηγαίου κώδικα στο οποίο μεταγλωττίζεται το διακριτικό.
- **__LINE__** - Διακριτικό μεταγλώττισης που αξιολογείται στον αριθμό γραμμής στον οποίο εμφανίζεται στον πηγαίο κώδικα.
- **__PACKAGE__** - Επιστρέφει το όνομα του πακέτου που ισχύει στο σημείο όπου εμφανίζεται το διακριτικό.
- **__SUB__** - Επιστρέφει μια αναφορά στην τρέχουσα υπορουτίνα.

Ενσωματωμένες συναρτήσεις ανά κατηγορία

Οι ίδιες ενσωματωμένες όπως στο *αλφαβητικό ευρετήριο*, ομαδοποιημένες εδώ ανά τομέα προβλήματος. Μια συνάρτηση μπορεί να εμφανίζεται σε περισσότερες από μία κατηγορίες: οι κατηγορίες είναι ένα σύνολο ετικετών, όχι δέντρο. Κάθε σύνδεσμος δείχνει στην ίδια σελίδα λεπτομερειών ανεξάρτητα από την κατηγορία μέσω της οποίας έφτασε ο αναγνώστης.

Βαθμωτά και συμβολοσειρές

- **chomp** - Αφαιρεί επιτόπου τον τερματικό διαχωριστή εγγραφών εισόδου από μια συμβολοσειρά.
- **chop** - Αφαιρεί τον τελευταίο χαρακτήρα μιας συμβολοσειράς και τον επιστρέφει.

- **chr** - Επιστρέφει τον χαρακτήρα του οποίου το σημείο κώδικα είναι ο δοσμένος αριθμός.
- **crypt** - Μονόδρομος κατακερματισμός τύπου `passwd(5)` μιας συμβολοσειράς απλού κειμένου με salt.
- **fc** - Επιστρέφει την casefolded μορφή Unicode μιας συμβολοσειράς για σύγκριση χωρίς διάκριση πεζών/κεφαλαίων.
- **hex** - Ερμηνεύει μια συμβολοσειρά ως δεκαεξαδικό αριθμό και επιστρέφει την αριθμητική τιμή της.
- **index** - Βρίσκει τη θέση μιας υποσυμβολοσειράς μέσα σε μια συμβολοσειρά.
- **lc** - Επιστρέφει αντίγραφο συμβολοσειράς με πεζά γράμματα.
- **lcfirst** - Επιστρέφει αντίγραφο συμβολοσειράς με τον πρώτο χαρακτήρα της σε πεζά.
- **length** - Επιστρέφει τον αριθμό των χαρακτήρων μιας συμβολοσειράς.
- **oct** - Ερμηνεύει μια συμβολοσειρά ως αριθμό γραμμένο σε οκταδικό, δεκαεξαδικό ή δυαδικό σύστημα και επιστρέφει τον προκύπτοντα ακέραιο.
- **ord** - Επιστρέφει το σημείο κώδικα Unicode του πρώτου χαρακτήρα μιας συμβολοσειράς.
- **pack** - Μετατρέπει μια λίστα τιμών Perl σε δυαδική συμβολοσειρά σύμφωνα με ένα πρότυπο.
- **q//** - Λεκτικό συμβολοσειράς μονών εισαγωγικών με επιλογή οριοθέτη.
- **qq//** - Κατασκευάζει συμβολοσειρά διπλών εισαγωγικών με παρεμβολή και οριοθέτη της επιλογής σας.
- **reverse** - Αντιστρέφει μια λίστα - ή, σε βαθμωτό περιβάλλον, αντιστρέφει τους χαρακτήρες μιας συμβολοσειράς.
- **rindex** - Βρίσκει τη θέση της **τελευταίας** εμφάνισης μιας υποσυμβολοσειράς μέσα σε μια συμβολοσειρά.
- **sprintf** - Κατασκευάζει μια μορφοποιημένη συμβολοσειρά από πρότυπο μορφοποίησης και λίστα τιμών.
- **substr** - Εξάγει, αντικαθιστά ή ορίζει ψευδώνυμο σε μια συνεχή φέτα συμβολοσειράς.
- **tr///** - Αντικατάσταση χαρακτήρα-προς-χαρακτήρα. Η `tr` σαρώνει μια συμβολοσειρά και αντικαθιστά κάθε εμφάνιση χαρακτήρα της `SEARCHLIST` με τον θεσικά αντίστοιχο χαρακτήρα της `REPLACEMENTLIST`, επιστρέφοντας τον αριθμό των χαρακτήρων που έθιξε.
- **uc** - Επιστρέφει αντίγραφο συμβολοσειράς με κεφαλαία γράμματα.
- **ucfirst** - Επιστρέφει αντίγραφο συμβολοσειράς με τον πρώτο χαρακτήρα της σε titlecase.
- **y///** - Μεταγραμματίζει χαρακτήρες σε μια συμβολοσειρά. Το `y///` είναι συνώνυμο της `tr///` - οι δύο είναι ταυτόσημες από κάθε άποψη.

Κανονικές εκφράσεις και αντιστοίχιση μοτίβων

- **`m//`** - Αναζητά ένα μοτίβο σε μια συμβολοσειρά και αναφέρει εάν - και τι - ταιριάζει.
- **`pos`** - Αναφέρει ή ορίζει πού θα συνεχίσει η επόμενη αντιστοιχία regex με `/g` σε μια συμβολοσειρά.
- **`qr//`** - Μεταγλωττίζει ένα μοτίβο μία φορά και επιστρέφει επαναχρησιμοποιήσιμο αντικείμενο regex.
- **`quotemeta`** - Επιστρέφει αντίγραφο συμβολοσειράς όπου κάθε χαρακτήρας σημαντικός για regex έχει διαφύγει με backslash, ώστε το αποτέλεσμα να μπορεί να παρεμβληθεί σε μοτίβο και να ταιριάζει με το κυριολεκτικό του περιεχόμενο.
- **`s///`** - Αναζητά ένα μοτίβο σε μια συμβολοσειρά και αντικαθιστά κάθε αντιστοιχία με την αντικατάσταση.
- **`split`** - Κόβει μια συμβολοσειρά σε λίστα πεδίων χρησιμοποιώντας έναν διαχωριστή regex.
- **`study`** - Μια κενή λειτουργία που διατηρείται για συμβατότητα πηγαίου κώδικα με παλαιότερο κώδικα Perl.

Αριθμητικές συναρτήσεις

- **`abs`** - Επιστρέφει την απόλυτη τιμή ενός αριθμού.
- **`atan2`** - Τόξο εφαπτομένης του `Y/X` στο εύρος -π έως π.
- **`cos`** - Επιστρέφει το συνημίτονο ενός αριθμού δοσμένου σε ακτίνια.
- **`exp`** - Υψώνει το *e* σε δύναμη.
- **`hex`** - Ερμηνεύει μια συμβολοσειρά ως δεκαεξαδικό αριθμό και επιστρέφει την αριθμητική τιμή της.
- **`int`** - Επιστρέφει το ακέραιο τμήμα ενός αριθμού, αποκόπτοντας προς το μηδέν.
- **`log`** - Επιστρέφει τον φυσικό λογάριθμο (βάση *e*) ενός αριθμού.
- **`oct`** - Ερμηνεύει μια συμβολοσειρά ως αριθμό γραμμένο σε οκταδικό, δεκαεξαδικό ή δυαδικό σύστημα και επιστρέφει τον προκύπτοντα ακέραιο.
- **`rand`** - Επιστρέφει έναν ψευδοτυχαίο αριθμό κινητής υποδιαστολής στο ημίκλειστο εύρος `[0, EXPR)`.
- **`sin`** - Επιστρέφει το ημίτονο ενός αριθμού δοσμένου σε ακτίνια.
- **`sqrt`** - Επιστρέφει τη μη αρνητική τετραγωνική ρίζα της `EXPR`.
- **`srand`** - Σπείρει τη γεννήτρια ψευδοτυχαίων αριθμών.

Πίνακες

- **`each`** - Διατρέχει ένα hash ή έναν πίνακα μία καταχώριση τη φορά.
- **`keys`** - Παραθέτει όλα τα κλειδιά ενός hash, ή όλους τους δείκτες ενός πίνακα.
- **`pop`** - Αφαιρεί και επιστρέφει το τελευταίο στοιχείο ενός πίνακα.
- **`push`** - Προσαρτά μία ή περισσότερες τιμές στο τέλος ενός πίνακα.

- **shift** - Αφαιρεί και επιστρέφει το πρώτο στοιχείο ενός πίνακα.
- **splice** - Αφαιρεί και/ή αντικαθιστά επιτόπου μια φέτα ενός πίνακα και επιστρέφει τα στοιχεία που αφαιρέθηκαν.
- **unshift** - Προσθέτει μία ή περισσότερες τιμές στην αρχή ενός πίνακα και επιστρέφει το νέο μήκος του πίνακα.
- **values** - Παραθέτει κάθε τιμή ενός hash (ή κάθε στοιχείο ενός πίνακα).

Λίστες

- **all** - Ελέγχει εάν το BLOCK επιστρέφει αληθές για **κάθε** στοιχείο της LIST.
- **any** - Επιστρέφει αληθές εάν το BLOCK δίνει αληθές για τουλάχιστον ένα στοιχείο της LIST.
- **grep** - Φιλτράρει μια λίστα στα στοιχεία όπου το μπλοκ ή η έκφραση είναι αληθές.
- **join** - Συνενώνει μια λίστα συμβολοσειρών με έναν διαχωριστή ανάμεσα σε κάθε γειτονικό ζεύγος και επιστρέφει την ενιαία προκύπτουσα συμβολοσειρά.
- **map** - Εφαρμόζει ένα μπλοκ ή έκφραση σε κάθε στοιχείο μιας λίστας και επιστρέφει τα επίπεδα αποτελέσματα.
- **qw//** - Κατασκευάζει λίστα από barewords διαχωρίζοντας μια οριοθετημένη συμβολοσειρά στα κενά, χωρίς εισαγωγικά ή κόμματα ανά λέξη.
- **reverse** - Αντιστρέφει μια λίστα - ή, σε βαθμωτό περιβάλλον, αντιστρέφει τους χαρακτήρες μιας συμβολοσειράς.
- **sort** - Ταξινομεί μια λίστα και επιστρέφει την ταξινομημένη λίστα.
- **unpack** - Εξάγει τυποποιημένες τιμές από δυαδική ή σταθερού πλάτους συμβολοσειρά σύμφωνα με ένα πρότυπο.

Hashes

- **delete** - Αφαιρεί το ή τα ζεύγη κλειδιού-τιμής από ένα hash, ή στοιχείο/α από έναν πίνακα, και επιστρέφει αυτό που αφαιρέθηκε.
- **each** - Διατρέχει ένα hash ή έναν πίνακα μία καταχώριση τη φορά.
- **exists** - Ελέγχει εάν ένα στοιχείο hash ή πίνακα, ή μια ονοματισμένη υπορουτίνα, υπάρχει - χωρίς να το δημιουργεί και χωρίς να ενδιαφέρεται τι περιέχει.
- **keys** - Παραθέτει όλα τα κλειδιά ενός hash, ή όλους τους δείκτες ενός πίνακα.
- **values** - Παραθέτει κάθε τιμή ενός hash (ή κάθε στοιχείο ενός πίνακα).

I/O

- **binmode** - Ορίζει τη στοίβα στρωμάτων I/O σε ένα filehandle - τυπικά για να παραδίδει ωμά bytes ή για να συνδέει μια κωδικοποίηση χαρακτήρων.
- **close** - Κλείνει ένα filehandle, εκκενώνει τους ενταμιευτές του και απελευθερώνει τον υποκείμενο περιγραφέα αρχείου.
- **closedir** - Κλείνει ένα handle καταλόγου που είχε ανοιχτεί από την opendir.

- **dbmclose** - Διακόπτει τη σύνδεση ανάμεσα σε ένα αρχείο DBM και ένα hash που είχε προηγουμένως συνδεθεί με dbmopen.
- **dbmopen** - Συνδέει ένα αρχείο DBM στον δίσκο με ένα hash, ώστε οι αναγνώσεις και οι εγγραφές του hash να γίνονται αναζητήσεις και αποθηκεύσεις στη βάση δεδομένων.
- **die** - Εγείρει μια εξαίρεση.
- **eof** - Ελέγχει ένα filehandle για end-of-file.
- **fileno** - Επιστρέφει τον αριθμό περιγραφέα αρχείου σε επίπεδο λειτουργικού συστήματος που βρίσκεται πίσω από ένα filehandle.
- **flock** - Τοποθετεί ένα συμβουλευτικό κλείδωμα σε ένα ανοιχτό αρχείο.
- **format** - Δηλώνει ένα πρότυπο αναφοράς βασισμένο σε εικόνα για χρήση από την write.
- **getc** - Διαβάζει τον επόμενο μεμονωμένο χαρακτήρα από ένα filehandle.
- **print** - Γράφει μια λίστα τιμών σε ένα filehandle.
- **printf** - Γράφει μια μορφοποιημένη συμβολοσειρά σε ένα filehandle.
- **read** - Διαβάζει σταθερή ποσότητα εισόδου με ενταμιευτή από ένα filehandle σε ένα βαθμωτό.
- **readdir** - Διαβάζει την επόμενη καταχώριση, ή όλες τις υπόλοιπες, από handle καταλόγου ανοιγμένο από την opendir.
- **readline** - Διαβάζει μία ή περισσότερες εγγραφές από ένα filehandle.
- **rewinddir** - Επαναφέρει ένα handle καταλόγου στην αρχή της λίστας του.
- **say** - Τυπώνει μια λίστα τιμών ακολουθούμενη από newline.
- **seek** - Επανατοποθετεί ένα filehandle για αναγνώσεις ή εγγραφές τυχαίας πρόσβασης.
- **seekdir** - Επαναφέρει ένα handle καταλόγου σε θέση που είχε προηγουμένως καταγραφεί από την telldir.
- **select** - Είτε ορίζει το προεπιλεγμένο filehandle εξόδου, είτε καλεί την κλήση συστήματος `select(2)` για πολυπλεξία I/O. Ίδιο όνομα, δύο άσχετες εργασίες - διακρίνονται από τον αριθμό ορισμάτων.
- **syscall** - Επικαλείται μια ωμή κλήση συστήματος με τον αριθμό πυρήνα της, περνώντας τα υπόλοιπα ορίσματα ως `int` ή ως δείκτη σε ενταμιευτή συμβολοσειράς.
- **sysread** - Διαβάζει ωμά bytes από ένα filehandle καλώντας την υποκείμενη κλήση συστήματος `read(2)`.
- **sysseek** - Επανατοποθετεί ένα filehandle σε επίπεδο συστήματος, παρακάμπτοντας τον ενταμιευτή PerlIO.
- **syswrite** - Γράφει bytes σε ένα filehandle με την ωμή κλήση συστήματος `write(2)`, παρακάμπτοντας το I/O με ενταμιευτή της Perl.
- **tell** - Επιστρέφει την τρέχουσα θέση σε bytes ενός filehandle.

- **telldir** - Επιστρέφει την τρέχουσα θέση ανάγνωσης ενός handle καταλόγου ως αδιαφανές διακριτικό.
- **truncate** - Συντομεύει (ή επεκτείνει) ένα αρχείο σε ακριβές μήκος bytes.
- **warn** - Εκπέμπει μια προειδοποίηση στο STDERR.
- **write** - Αποτυπώνει μία εγγραφή σε ένα filehandle μέσω του συσχετισμένου του `format`.

Δεδομένα σταθερού μήκους

- **pack** - Μετατρέπει μια λίστα τιμών Perl σε δυαδική συμβολοσειρά σύμφωνα με ένα πρότυπο.
- **read** - Διαβάζει σταθερή ποσότητα εισόδου με ενταμιευτή από ένα filehandle σε ένα βαθμωτό.
- **syscall** - Επικαλείται μια ωμή κλήση συστήματος με τον αριθμό πυρήνα της, περνώντας τα υπόλοιπα ορίσματα ως `int` ή ως δείκτη σε ενταμιευτή συμβολοσειράς.
- **sysread** - Διαβάζει ωμά bytes από ένα filehandle καλώντας την υποκείμενη κλήση συστήματος `read(2)`.
- **sysseek** - Επανατοποθετεί ένα filehandle σε επίπεδο συστήματος, παρακάμπτοντας τον ενταμιευτή PerlIO.
- **syswrite** - Γράφει bytes σε ένα filehandle με την ωμή κλήση συστήματος `write(2)`, παρακάμπτοντας το I/O με ενταμιευτή της Perl.
- **unpack** - Εξάγει τυποποιημένες τιμές από δυαδική ή σταθερού πλάτους συμβολοσειρά σύμφωνα με ένα πρότυπο.
- **vec** - Διαβάζει ή γράφει μια θέση σταθερού πλάτους μέσα σε συμβολοσειρά που αντιμετωπίζεται ως πακεταρισμένο διάνυσμα bit.

Filehandles, αρχεία, κατάλογοι

- **chdir** - Αλλάζει τον τρέχοντα κατάλογο εργασίας της διεργασίας.
- **chmod** - Αλλάζει τα bits δικαιωμάτων μιας λίστας αρχείων.
- **chown** - Αλλάζει τον ιδιοκτήτη και την ομάδα μιας λίστας αρχείων.
- **chroot** - Αλλάζει τον ριζικό κατάλογο της τρέχουσας διεργασίας και κάθε θυγατρικής που θα δημιουργήσει αργότερα.
- **fcntl** - Εκτελεί μια λειτουργία ελέγχου αρχείου `fcntl(2)` σε ένα filehandle.
- **glob** - Επεκτείνει ένα μοτίβο ονόματος αρχείου τύπου shell στη λίστα των διαδρομών που αντιστοιχούν.
- **ioctl** - Εκτελεί μια κλήση συστήματος ελέγχου συσκευής `ioctl(2)` σε ένα filehandle.
- **link** - Δημιουργεί έναν σκληρό σύνδεσμο από το `NEWFILE` προς το υπάρχον `OLDFILE`.

- **lstat** - Επιστρέφει τη λίστα κατάστασης 13 στοιχείων για μια διαδρομή **χωρίς** να ακολουθήσει συμβολικό σύνδεσμο.
- **mkdir** - Δημιουργεί έναν μόνο κατάλογο στο σύστημα αρχείων.
- **open** - Συσχετίζει ένα filehandle με ένα αρχείο, μια εντολή ή ένα βαθμωτό στη μνήμη.
- **opendir** - Ανοίγει έναν κατάλογο για ανάγνωση.
- **readlink** - Επιστρέφει τη διαδρομή στόχου που δείχνει ένας συμβολικός σύνδεσμος.
- **rename** - Αλλάζει το όνομα ενός αρχείου.
- **rmdir** - Αφαιρεί έναν κενό κατάλογο.
- **select** - Είτε ορίζει το προεπιλεγμένο filehandle εξόδου, είτε καλεί την κλήση συστήματος `select(2)` για πολυπλεξία I/O. Ίδιο όνομα, δύο άσχετες εργασίες - διακρίνονται από τον αριθμό ορισμάτων.
- **stat** - Λαμβάνει τις πληροφορίες κατάστασης ενός αρχείου.
- **symlink** - Δημιουργεί έναν συμβολικό σύνδεσμο στο `NEWFILE` που δείχνει στη συμβολοσειρά `OLDFILE`.
- **sysopen** - Ανοίγει ένα αρχείο με τον χαμηλού επιπέδου τρόπο, περνώντας ένα ακέραιο bitmask `MODE` απευθείας στην υποκείμενη κλήση συστήματος `open(2)`.
- **umask** - Ορίζει ή διαβάζει τη μάσκα κατάστασης δημιουργίας αρχείων της διεργασίας.
- **unlink** - Αφαιρεί μία ή περισσότερες καταχωρίσεις καταλόγου που ονομάζουν αρχεία.
- **utime** - Ορίζει χρόνους πρόσβασης και τροποποίησης σε μια λίστα αρχείων.

Ροή ελέγχου

- **__FILE__** - Το όνομα του αρχείου πηγαίου κώδικα στο οποίο μεταγλωττίζεται το διακριτικό.
- **__LINE__** - Διακριτικό μεταγλώττισης που αξιολογείται στον αριθμό γραμμής στον οποίο εμφανίζεται στον πηγαίο κώδικα.
- **__PACKAGE__** - Επιστρέφει το όνομα του πακέτου που ισχύει στο σημείο όπου εμφανίζεται το διακριτικό.
- **__SUB__** - Επιστρέφει μια αναφορά στην τρέχουσα υπορουτίνα.
- **break** - Έξοδος από ένα μπλοκ `given`.
- **caller** - Επιστρέφει πληροφορίες για την υπορουτίνα, την `eval` ή την `require` που κάλεσε τον τρέχοντα κώδικα.
- **catch** - Χειρίζεται μια εξαίρεση που έχει εγερθεί από προηγούμενο μπλοκ `try`.
- **continue** - Συνδέει σε έναν βρόχο ένα μπλοκ που εκτελείται μετά από κάθε επανάληψη, λίγο πριν επανελεγχθεί η συνθήκη.
- **defer** - Προγραμματίζει την εκτέλεση ενός μπλοκ όταν εξέρχεται η περικλείουσα εμβέλεια, για οποιοδήποτε λόγο.

- **die** - Εγείρει μια εξαίρεση.
- **do** - Εκτελεί ένα μπλοκ κώδικα ή τρέχει ένα αρχείο πηγαίου κώδικα Perl σαν να ήταν μέρος του τρέχοντος προγράμματος.
- **dump** - Προκαλεί στο τρέχον πρόγραμμα την άμεση παραγωγή core dump.
- **eval** - Εκτελεί ένα κομμάτι κώδικα Perl παγιδεύοντας κάθε μοιραίο σφάλμα αντί να τερματιστεί το πρόγραμμα.
- **evalbytes** - Μεταγλωττίζει και εκτελεί μια συμβολοσειρά πηγαίου κώδικα Perl, εξαναγκάζοντας τον πηγαίο κώδικα να ερμηνευθεί ως **bytes** και όχι ως χαρακτήρες.
- **exit** - Τερματίζει το πρόγραμμα με μια τιμή κατάστασης.
- **finally** - Εκτελεί κώδικα εκκαθάρισης κατά την έξοδο από `try`/`catch`, είτε το σώμα ολοκληρώθηκε επιτυχώς, είτε εγείρει εξαίρεση, είτε πραγματοποίησε άλμα προς τα έξω.
- **goto** - Μεταφέρει την εκτέλεση αλλού στο πρόγραμμα χωρίς επιστροφή.
- **last** - Εξέρχεται άμεσα από έναν βρόχο, παραλείποντας το υπόλοιπο του σώματος και το μπλοκ `continue`.
- **method** - Δηλώνει μια ονοματισμένη μέθοδο στιγμιοτύπου μέσα σε ένα μπλοκ `class`.
- **next** - Ξεκινά αμέσως την επόμενη επανάληψη του περικλείοντος βρόχου.
- **redo** - Επανεκκινεί την τρέχουσα επανάληψη ενός βρόχου χωρίς να επανελέγξει τη συνθήκη και χωρίς να εκτελέσει το μπλοκ `continue`.
- **return** - Εξέρχεται από την τρέχουσα υπορουτίνα, `eval`, do `FILE`, μπλοκ `sort` ή μπλοκ regex eval, αποδίδοντας μια τιμή στον καλούντα.
- **sub** - Δηλώνει ή ορίζει μια υπορουτίνα.
- **try** - Εκτελεί ένα μπλοκ και εκτρέπει κάθε εξαίρεση που εγείρει σε ένα μπλοκ `catch`, με προαιρετικό μπλοκ `finally` που εκτελείται πάντα στην έξοδο.
- **wantarray** - Αναφέρει το περιβάλλον κλήσης της τρέχουσας υπορουτίνας.

Εμβέλεια

- **caller** - Επιστρέφει πληροφορίες για την υπορουτίνα, την `eval` ή την `require` που κάλεσε τον τρέχοντα κώδικα.
- **class** - Δηλώνει έναν χώρο ονομάτων που συμπεριφέρεται ως εγγενής κλάση αντικειμένου.
- **field** - Δηλώνει μια μεταβλητή ανά στιγμιότυπο μέσα σε ένα μπλοκ `class`.
- **import** - Συμπληρώνει τον χώρο ονομάτων του καλούντος με ονόματα που το άρθρωμα επιλέγει να εξάγει.
- **local** - Αποθηκεύει την τρέχουσα τιμή μιας μεταβλητής πακέτου και την επαναφέρει όταν εξέρχεται η περικλείουσα εμβέλεια.
- **my** - Δηλώνει μία ή περισσότερες λεξιλογικά ορισμένες μεταβλητές.
- **our** - Δηλώνει ένα λεξιλογικά ορισμένο ψευδώνυμο σε μια μεταβλητή πακέτου.

- **package** - Δηλώνει τον χώρο ονομάτων κατά τη μεταγλώττιση για τις δηλώσεις που ακολουθούν.
- **state** - Δηλώνει μια λεξιλογικά ορισμένη μεταβλητή της οποίας η τιμή διατηρείται μεταξύ κλήσεων της περικλείουσας υπορουτίνας.
- **use** - Φορτώνει ένα άρθρωμα κατά τη μεταγλώττιση και εισάγει τα σύμβολά του στο τρέχον πακέτο.

Διάφορα

- **defined** - Ελέγχει εάν μια τιμή, μεταβλητή ή υπορουτίνα είναι ορισμένη.
- **formline** - Μορφοποιεί μια λίστα τιμών σε συμβολοσειρά εικόνας και προσαρτά το αποτέλεσμα στον συσσωρευτή μορφοποίησης `$^A`.
- **lock** - Τοποθετεί ένα συμβουλευτικό κλείδωμα σε μια κοινόχρηστη μεταβλητή, πίνακα, hash ή υπορουτίνα μέχρι το κλείδωμα να βγει εκτός εμβέλειας.
- **prototype** - Επιστρέφει τη συμβολοσειρά πρωτοτύπου μιας υπορουτίνας, ή `undef` εάν δεν έχει.
- **reset** - Καθαρίζει κάθε μεταβλητή πακέτου της οποίας το όνομα ξεκινά με ένα από ένα σύνολο γραμμάτων και επανοπλίζει αντιστοιχίσεις μιας χρήσης `m?pattern?`.
- **scalar** - Εξαναγκάζει την `EXPR` να αξιολογηθεί σε βαθμωτό περιβάλλον και επιστρέφει την τιμή της.
- **undef** - Η ακαθόριστη τιμή και ο τελεστής που την παράγει.

Διεργασίες

- **alarm** - Προγραμματίζει την παράδοση ενός `SIGALRM` στην τρέχουσα διεργασία μετά από ακέραιο αριθμό δευτερολέπτων πραγματικού χρόνου.
- **exec** - Εγκαταλείπει αυτό το πρόγραμμα και εκτελεί άλλο στην ίδια διεργασία.
- **fork** - Δημιουργεί μια νέα διεργασία που εκτελεί το ίδιο πρόγραμμα στο ίδιο σημείο.
- **getpgrp** - Επιστρέφει το POSIX αναγνωριστικό ομάδας διεργασιών στο οποίο ανήκει μια διεργασία.
- **getppid** - Επιστρέφει το αναγνωριστικό διεργασίας της γονικής της τρέχουσας διεργασίας.
- **getpriority** - Επιστρέφει την τρέχουσα τιμή nice χρονοπρογραμματισμού μιας διεργασίας, μιας ομάδας διεργασιών ή ενός χρήστη.
- **kill** - Στέλνει ένα σήμα σε μια λίστα διεργασιών.
- **pipe** - Ανοίγει ένα ζεύγος συνδεδεμένων filehandles - ένα για ανάγνωση, ένα για εγγραφή.
- **qx//** - Εκτελεί μια εντολή shell και αιχμαλωτίζει την τυπική της έξοδο.
- **readpipe** - Εκτελεί μια εντολή shell και επιστρέφει την τυπική της έξοδο.
- **setpgrp** - Ορίζει την ομάδα διεργασιών μιας διεργασίας.

- **setpriority** - Ορίζει την προτεραιότητα χρονοπρογραμματισμού (τιμή nice) μιας διεργασίας, ομάδας διεργασιών ή χρήστη.
- **sleep** - Παύει τη διεργασία για ακέραιο αριθμό δευτερολέπτων.
- **system** - Εκτελεί ένα ξεχωριστό πρόγραμμα και περιμένει να ολοκληρωθεί.
- **times** - Αναφέρει τον χρόνο CPU που έχει καταναλώσει αυτή η διεργασία και οι τερματισμένες θυγατρικές της.
- **wait** - Μπλοκάρει μέχρι να εξέλθει κάποια θυγατρική διεργασία και τη συγκομίζει.
- **waitpid** - Περιμένει να τερματιστεί μια συγκεκριμένη θυγατρική διεργασία και τη συγκομίζει.

Αρθρώματα

- **do** - Εκτελεί ένα μπλοκ κώδικα ή τρέχει ένα αρχείο πηγαίου κώδικα Perl σαν να ήταν μέρος του τρέχοντος προγράμματος.
- **import** - Συμπληρώνει τον χώρο ονομάτων του καλούντος με ονόματα που το άρθρωμα επιλέγει να εξάγει.
- **no** - Η αντίστροφη του `use` κατά τη μεταγλώττιση - καλεί τη μέθοδο `unimport` ενός αρθρώματος για να απενεργοποιήσει ό,τι ενεργοποίησε η `use`.
- **package** - Δηλώνει τον χώρο ονομάτων κατά τη μεταγλώττιση για τις δηλώσεις που ακολουθούν.
- **require** - Φορτώνει ένα αρχείο πηγαίου κώδικα Perl κατά τον χρόνο εκτέλεσης, ή απαιτεί ελάχιστη έκδοση Perl.
- **use** - Φορτώνει ένα άρθρωμα κατά τη μεταγλώττιση και εισάγει τα σύμβολά του στο τρέχον πακέτο.

Κλάσεις και αντικειμενοστρέφεια

- **__CLASS__** - Επιστρέφει το όνομα κλάσης του στιγμιοτύπου επί του οποίου ενεργείται.
- **bless** - Σημαίνει αυτό που δείχνει μια αναφορά ως αντικείμενο ενός πακέτου.
- **class** - Δηλώνει έναν χώρο ονομάτων που συμπεριφέρεται ως εγγενής κλάση αντικειμένου.
- **dbmclose** - Διακόπτει τη σύνδεση ανάμεσα σε ένα αρχείο DBM και ένα hash που είχε προηγουμένως συνδεθεί με `dbmopen`.
- **dbmopen** - Συνδέει ένα αρχείο DBM στον δίσκο με ένα hash, ώστε οι αναγνώσεις και οι εγγραφές του hash να γίνονται αναζητήσεις και αποθηκεύσεις στη βάση δεδομένων.
- **field** - Δηλώνει μια μεταβλητή ανά στιγμιότυπο μέσα σε ένα μπλοκ `class`.
- **isa** - Ελέγχει εάν ένα αντικείμενο είναι στιγμιότυπο μιας κλάσης ή οποιασδήποτε υποκλάσης που προέρχεται από αυτήν.
- **method** - Δηλώνει μια ονοματισμένη μέθοδο στιγμιοτύπου μέσα σε ένα μπλοκ `class`.

- **package** - Δηλώνει τον χώρο ονομάτων κατά τη μεταγλώττιση για τις δηλώσεις που ακολουθούν.
- **ref** - Επιστρέφει μια συμβολοσειρά που περιγράφει σε τι δείχνει μια αναφορά.
- **tie** - Συνδέει μια μεταβλητή με μια κλάση, ώστε κάθε πρόσβαση στη μεταβλητή να αποστέλλεται μέσω των μεθόδων της κλάσης.
- **tied** - Επιστρέφει το αντικείμενο που υποστηρίζει μια tied μεταβλητή.
- **untie** - Διακόπτει τη σύνδεση ανάμεσα σε μια μεταβλητή και την tied κλάση της.
- **use** - Φορτώνει ένα άρθρωμα κατά τη μεταγλώττιση και εισάγει τα σύμβολά του στο τρέχον πακέτο.

Υποδοχές (sockets)

- **accept** - Δέχεται μια εισερχόμενη σύνδεση σε υποδοχή που ακούει.
- **bind** - Συνδέει μια τοπική διεύθυνση σε μια υποδοχή.
- **connect** - Αρχίζει μια σύνδεση από μια υποδοχή προς απομακρυσμένη διεύθυνση.
- **getpeername** - Επιστρέφει τη διεύθυνση του απομακρυσμένου άκρου μιας συνδεδεμένης υποδοχής.
- **getsockname** - Επιστρέφει την τοπική διεύθυνση μιας συνδεδεμένης ή συνδεμένης υποδοχής.
- **getsockopt** - Διαβάζει μια επιλογή υποδοχής από τον πυρήνα ως αδιαφανή πακεταρισμένη συμβολοσειρά.
- **listen** - Σημαίνει μια υποδοχή ως παθητική, ώστε να μπορεί να δέχεται εισερχόμενες συνδέσεις.
- **recv** - Διαβάζει ένα εισερχόμενο μήνυμα από μια υποδοχή σε ένα βαθμωτό.
- **send** - Στέλνει ένα μήνυμα σε υποδοχή.
- **setsockopt** - Ορίζει μια επιλογή σε επίπεδο πυρήνα σε μια ανοιχτή υποδοχή.
- **shutdown** - Τερματίζει μία κατεύθυνση σύνδεσης υποδοχής, ή και τις δύο.
- **socket** - Δημιουργεί ένα filehandle υποδοχής.
- **socketpair** - Δημιουργεί ένα ανώνυμο, συνδεδεμένο ζεύγος υποδοχών που επικοινωνούν μεταξύ τους.

SysV IPC

- **msgctl** - Εκτελεί λειτουργία ελέγχου σε ουρά μηνυμάτων System V IPC.
- **msgget** - Δημιουργεί ή αναζητά μια ουρά μηνυμάτων System V IPC και επιστρέφει το id της.
- **msgrcv** - Λαμβάνει ένα μήνυμα από ουρά μηνυμάτων System V IPC.
- **msgsnd** - Στέλνει ένα μήνυμα σε ουρά μηνυμάτων System V IPC.
- **semctl** - Εκτελεί λειτουργία ελέγχου σε σύνολο σηματοφόρων System V.

- **semget** - Δημιουργεί ή αναζητά ένα σύνολο σηματοφόρων System V και επιστρέφει το αναγνωριστικό του.
- **semop** - Εκτελεί μία ή περισσότερες λειτουργίες σηματοφόρου System V ατομικά.
- **shmctl** - Ελέγχει ή ερωτά ένα τμήμα κοινόχρηστης μνήμης System V.
- **shmget** - Δημιουργεί ή αναζητά ένα τμήμα κοινόχρηστης μνήμης System V και επιστρέφει το αναγνωριστικό του.
- **shmread** - Αντιγράφει bytes από τμήμα κοινόχρηστης μνήμης System V σε ένα βαθμωτό της Perl.
- **shmwrite** - Αντιγράφει bytes σε τμήμα κοινόχρηστης μνήμης System V.

Πληροφορίες χρηστών και ομάδων

- **endgrent** - Κλείνει τη βάση δεδομένων ομάδων μετά τη διάτρεξή της.
- **endpwent** - Κλείνει τον επαναλήπτη της βάσης passwd που έχει ανοιχτεί από τις `getpwent` ή `setpwent`.
- **getgrent** - Διαβάζει την επόμενη καταχώριση από τη βάση δεδομένων ομάδων του συστήματος.
- **getgrgid** - Αναζητά μια εγγραφή ομάδας με βάση τον αριθμητικό κωδικό ομάδας.
- **getgrnam** - Αναζητά μια ομάδα Unix με το όνομά της και επιστρέφει την εγγραφή `/etc/group` της.
- **getlogin** - Επιστρέφει το όνομα σύνδεσης του χρήστη που σχετίζεται με το ελέγχον τερματικό.
- **getpwent** - Επιστρέφει την επόμενη καταχώριση από τη βάση δεδομένων κωδικών του συστήματος.
- **getpwnam** - Αναζητά την εγγραφή passwd ενός χρήστη με βάση το όνομα σύνδεσης.
- **getpwuid** - Αναζητά την εγγραφή passwd ενός χρήστη με βάση το αριθμητικό UID.
- **setgrent** - Επαναφέρει τον επαναλήπτη της βάσης δεδομένων ομάδων στην πρώτη καταχώριση.
- **setpwent** - Επαναφέρει τον επαναλήπτη της βάσης κωδικών ώστε η επόμενη κλήση `getpwent` να επιστρέψει ξανά την πρώτη καταχώριση.

Πληροφορίες δικτύου

- **endhostent** - Κλείνει τη βάση δεδομένων υπολογιστών μετά τη διάτρεξη.
- **endnetent** - Κλείνει τη βάση δεδομένων δικτύων μετά τη διάτρεξη.
- **endprotoent** - Κλείνει τη βάση δεδομένων πρωτοκόλλων μετά τη διάτρεξη.
- **endservent** - Κλείνει τη βάση δεδομένων υπηρεσιών μετά από διάτρεξη με την `getservent`.
- **gethostbyaddr** - Αναζητά μια εγγραφή υπολογιστή με βάση την πακεταρισμένη διεύθυνσή IP του.
- **gethostbyname** - Αναζητά μια εγγραφή υπολογιστή με βάση το όνομα DNS.

- **gethostent** - Ανακτά την επόμενη καταχώριση από τη βάση δεδομένων υπολογιστών.
- **getnetbyaddr** - Αναζητά μια εγγραφή δικτύου με βάση την αριθμητική διεύθυνση δικτύου.
- **getnetbyname** - Αναζητά μια εγγραφή δικτύου με όνομα.
- **getnetent** - Διαβάζει την επόμενη καταχώριση από τη βάση δεδομένων δικτύων.
- **getprotobyname** - Αναζητά ένα πρωτόκολλο IP με το όνομά του σε κείμενο και επιστρέφει την καταχώρισή του στη βάση δεδομένων.
- **getprotobynumber** - Αναζητά καταχώριση πρωτοκόλλου δικτύου με βάση τον αριθμό πρωτοκόλλου που του έχει αντιστοιχιστεί.
- **getprotoent** - Ανακτά την επόμενη καταχώριση από τη βάση δεδομένων πρωτοκόλλων.
- **getservbyname** - Αναζητά μια υπηρεσία δικτύου με βάση το όνομα κειμένου και το πρωτόκολλο.
- **getservbyport** - Αναζητά μια υπηρεσία δικτύου με βάση την αριθμητική θύρα και το πρωτόκολλο.
- **getservent** - Διαβάζει την επόμενη καταχώριση από τη βάση δεδομένων υπηρεσιών του συστήματος.
- **sethostent** - Ανοίγει ή επαναφέρει τη βάση δεδομένων υπολογιστών για διάτρεξη.
- **setnetent** - Ανοίγει ή επαναφέρει τη βάση δεδομένων δικτύων για διάτρεξη.
- **setprotoent** - Ανοίγει τη βάση δεδομένων πρωτοκόλλων και την προετοιμάζει για διαδοχικές αναγνώσεις.
- **setservent** - Επαναφέρει τη βάση δεδομένων υπηρεσιών και προαιρετικά την κρατά ανοιχτή ανάμεσα στις αναζητήσεις.

Χρόνος

- **gmtime** - Μετατρέπει έναν χρόνο epoch σε αναλυμένο χρόνο UTC, είτε ως λίστα 9 στοιχείων είτε ως συμβολοσειρά τύπου `ctime(3)`.
- **localtime** - Μετατρέπει έναν χρόνο epoch σε αναλυμένο ημερολογιακό χρόνο στην τοπική ζώνη ώρας.
- **time** - Επιστρέφει τον τρέχοντα πραγματικό χρόνο ως ακέραιο αριθμό δευτερολέπτων από το epoch του συστήματος.
- **times** - Αναφέρει τον χρόνο CPU που έχει καταναλώσει αυτή η διεργασία και οι τερματισμένες θυγατρικές της.

Κλάσεις και αντικειμενοστρέφεια

CLASS

Επιστρέφει το όνομα της κλάσης του στιγμιότυπου που χειρίζεται τη δεδομένη στιγμή.

Το `__CLASS__` είναι ένα διακριτικό που αναγνωρίζεται κατά τη μεταγλώττιση και αποτιμάται σε χρόνο εκτέλεσης στο όνομα της κλάσης του στιγμιότυπου που το

επικαλείται. Μέσα σε σώμα `method` ή σε έκφραση αρχικοποιητή `field` (που εισάγεται με `use feature 'class'`), δίνει την **πραγματική** κλάση του αντικειμένου που κατασκευάζεται ή χρησιμοποιείται - η οποία μπορεί να είναι υποκλάση του πακέτου στο οποίο βρίσκεται κειμενικά ο κώδικας. Εκτός κώδικα που χρησιμοποιεί το χαρακτηριστικό class, είναι κατ" ουσίαν ισοδύναμο με το `ref($self)`, όμως σε αντίθεση με το `ref` μπορεί να χρησιμοποιηθεί σε αρχικοποιητή πεδίου, όπου το `$self` δεν έχει ακόμη δεσμευτεί.

Σύνοψη

```
use feature 'class';

field $x = __CLASS__->DEFAULT_X;        # in a field initializer
method m { return __CLASS__; }          # in a method body
```

Τι επιστρέφεται

Μια απλή συμβολοσειρά: το όνομα της κλάσης του στιγμιότυπου που την επικαλείται. Η τιμή είναι το ίδιο είδος συμβολοσειράς που επιστρέφει η `ref` για μια blessed αναφορά. **Δεν** είναι αναφορά· καλέστε μεθόδους πάνω της όπως θα κάνατε για οποιοδήποτε όνομα κλάσης.

```
my $name = __CLASS__;                   # e.g. "DifferentCustomField
 ↪"
my $obj  = __CLASS__->new(...);         # class-method dispatch
```

Πότε διαφέρει από το __PACKAGE__

Το `__PACKAGE__` επιλύεται κατά τη μεταγλώττιση και ονοματίζει το πακέτο στο οποίο εμφανίζεται κειμενικά το διακριτικό. Το `__CLASS__` επιλύεται σε χρόνο εκτέλεσης και ονοματίζει την κλάση του *στιγμιότυπου* - η οποία, για μέθοδο ή αρχικοποιητή πεδίου που κληρονομείται από υποκλάση, είναι η υποκλάση.

```
use feature 'class';

class Base {
    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"
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
# $f was initialised to 20, not 10, because __CLASS__ dispatched
# through the Derived package.
```

Σε μη κληρονομικό περιβάλλον (χωρίς υποκλάσεις), τα __CLASS__ και __PACKAGE__ αποδίδουν την ίδια συμβολοσειρά.

Παραδείγματα

Μέθοδος κλάσης που καλείται από αρχικοποιητή πεδίου - η κανονική περίπτωση χρήσης. Χωρίς το __CLASS__, μια υποκλάση δεν μπορεί να υπερκαλύψει την προεπιλογή:

```
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
```

Μέσα σε σώμα μεθόδου, ισοδύναμο με `ref $self`:

```
class Logger {
    method log ($msg) {
        print STDERR "[", __CLASS__, "] $msg\n";
    }
}
```

Κατασκευή νέου στιγμιότυπου της ίδιας κλάσης με το αντικείμενο που το επικαλείται - χρήσιμο για εργοστασιακές μεθόδους τύπου clone που πρέπει να σέβονται τις υποκλάσεις:

```
class Node {
    field $label :param;
    method spawn ($child_label) {
        return __CLASS__->new(label => $child_label);
    }
}
```

Σύγκριση με το __PACKAGE__ για διαγνωστικά που θέλουν να καρφώσουν τον ακριβή στόχο αποστολής:

```
class Thing {
    method describe {
        return sprintf "defined in %s, dispatched as %s",
            __PACKAGE__, __CLASS__;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    }
}
```

Οριακές περιπτώσεις

- **Έγκυρο μόνο υπό `use feature 'class'`** μέσα σε σώμα `method`, σε αρχικοποιητή `field`, ή σε άλλη κατασκευή του χαρακτηριστικού class (για παράδειγμα μπλοκ `ADJUST`). Η χρήση του εκτός τέτοιου περιβάλλοντος είναι σφάλμα κατά τη μεταγλώττιση.
- **Χωρίς παρενθέσεις, χωρίς ορίσματα.** Το `__CLASS__` είναι διακριτικό τύπου bareword, όχι κλήση συνάρτησης. Το `__CLASS__()` είναι συντακτικό σφάλμα.
- **Δεν είναι το ίδιο με `ref $self`** κατά την αρχικοποίηση των πεδίων - το `$self` δεν έχει ακόμη δεσμευτεί όσο τα πεδία ορίζονται, άρα το `ref $self` δεν είναι διαθέσιμο. Το `__CLASS__` είναι ο μόνος τρόπος να φτάσετε στην κλάση του στιγμιότυπου μέσα από αρχικοποιητή πεδίου.
- **Δεν είναι `ref` για κλήση μεθόδου κλάσης.** Όταν καλείται ως μέθοδος κλάσης (χωρίς στιγμιότυπο στο χέρι), το `__CLASS__` αποδίδει την κλάση πάνω στην οποία κλήθηκε η μέθοδος, όχι το πακέτο στο οποίο ορίστηκε η μέθοδος. Αυτό αντικατοπτρίζει την κανονική αποστολή μεθόδου.
- **Συμβολοσειρά, όχι αναφορά.** Οι κλήσεις μεθόδων μέσω `__CLASS__->...` είναι αποστολές μεθόδων κλάσης, όχι αποστολές μεθόδων στιγμιότυπου. Η μέθοδος δεν θα δει blessed αναφορά ως invocant.
- **The `class` feature is marked experimental** in Perl 5.44; it emits a `experimental::class` warning unless silenced with `no warnings 'experimental::class'`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `__PACKAGE__` - όνομα πακέτου κατά τη μεταγλώττιση του περιβάλλοντος κώδικα· χρησιμοποιήστε το όταν θέλετε το κειμενικό πακέτο, όχι την κλάση χρόνου εκτέλεσης
- `ref` - όνομα κλάσης χρόνου εκτέλεσης μιας blessed αναφοράς· χρησιμοποιήστε το όταν κρατάτε ήδη στιγμιότυπο και δεν βρίσκεστε μέσα σε αρχικοποιητή πεδίου
- `bless` - συσχετίζει μια αναφορά με όνομα κλάσης· η χαμηλού επιπέδου πρωτογενής λειτουργία στην οποία τελικά επιλύεται η αποστολή τύπου `__CLASS__`
- `isa` - ελέγχει αν η τρέχουσα κλάση είναι υποκλάση ενός δοθέντος ονόματος· συνδυάζεται φυσικά με το `__CLASS__` για ελέγχους χρόνου εκτέλεσης
- `sub` - δηλώνει υπορουτίνες, συμπεριλαμβανομένης της μορφής `method` που κάνει το `__CLASS__` να έχει νόημα

Ροή ελέγχου

FILE

Το όνομα του αρχείου πηγαίου κώδικα στο οποίο μεταγλωττίζεται το διακριτικό.

Το `__FILE__` είναι διακριτικό μεταγλώττισης, όχι συνάρτηση χρόνου εκτέλεσης. Στο σημείο όπου το διαβάζει ο μεταγλωττιστής, αντικαθίσταται από μια συμβολοσειρά κυριολεκτικής τιμής που περιέχει το όνομα αρχείου από το οποίο διαβάζει εκείνη τη στιγμή ο μεταγλωττιστής. Δεν δέχεται ορίσματα, δεν καλείται ποτέ με παρενθέσεις, και δεν έχει κόστος κατά τον χρόνο εκτέλεσης - μέχρι να εκτελεστεί το πρόγραμμά σας, το `__FILE__` έχει ήδη γίνει μια απλή σταθερά συμβολοσειράς.

Η μοναδική του δουλειά είναι να επιτρέπει στον κώδικα να αναγνωρίζει το αρχείο στο οποίο βρίσκεται, για μηνύματα σφαλμάτων, γραμμές καταγραφής, διαγνωστικά δοκιμών, και παρόμοιες αναφορές.

Σύνοψη

```
__FILE__
```

Τι επιστρέφεται

Μια συμβολοσειρά: το όνομα αρχείου όπως το γνώριζε ο μεταγλωττιστής όταν διάβασε αυτό το διακριτικό. Για ένα σενάριο που καλείται ως `perl /tmp/run.pl` η τιμή είναι `/tmp/run.pl`· για ένα άρθρωμα που φορτώνεται μέσω `require` ή `use` είναι η διαδρομή που επέλυσε το `require` έναντι του `@INC` (συνήθως απόλυτη διαδρομή). Για κώδικα που τροφοδοτείται μέσω της τυπικής εισόδου ή μέσω `-e` είναι `-` ή `-e` αντίστοιχα.

Επειδή η αντικατάσταση γίνεται κατά τη μεταγλώττιση, η τιμή είναι σταθερή για κάθε κειμενική εμφάνιση του διακριτικού. Δύο διακριτικά `__FILE__` σε διαφορετικά αρχεία παράγουν διαφορετικές συμβολοσειρές· δύο στο ίδιο αρχείο παράγουν την ίδια συμβολοσειρά.

Καθολική κατάσταση που επηρεάζει

Καμία κατά τον χρόνο εκτέλεσης. Κατά τη μεταγλώττιση η τιμή λαμβάνεται από το τρέχον όνομα αρχείου του μεταγλωττιστή, που είναι η ίδια τιμή την οποία αναφέρει η `caller` σε περιβάλλον λίστας ως στοιχείο `[1]` και την οποία χρησιμοποιούν οι `die` / `warn` για να δομήσουν το προεπιλεγμένο τους επίθεμα `"at FILE line LINE."`.

Αυτό το όνομα αρχείου μπορεί να **επανεγγραφεί** από οδηγία `#line` ενσωματωμένη στον πηγαίο κώδικα - δείτε *Οριακές περιπτώσεις* παρακάτω. Μόλις επανεγγραφεί, τα διακριτικά `__FILE__` που μεταγλωττίζονται μετά την οδηγία αναφέρουν το νέο όνομα.

Παραδείγματα

Αναγνώριση του σεναρίου που εκτελείται σε μια γραμμή καταγραφής:

```
warn "started from ", __FILE__, "\n";
# started from /tmp/run.pl
```

Επισύναψη της θέσης στον πηγαίο κώδικα σε ένα προσαρμοσμένο σφάλμα χωρίς το προεπιλεγμένο επίθεμα της `die`:

```
sub fail {
    my ($msg) = @_;
    die sprintf("%s at %s line %d\n", $msg, __FILE__, __LINE__);
}
```

Δόμηση διαδρομής σχετικής με το τρέχον άρθρωμα, ώστε τα αρχεία πόρων να διανέμονται μαζί με τον κώδικα που τα φορτώνει:

```
use File::Basename qw(dirname);
use File::Spec;
my $data = File::Spec->catfile(dirname(__FILE__), "data.txt");
```

Χρησιμοποιήστε το `__FILE__` μέσα σε μια συμβολοσειρά που περνά στην `eval` ώστε οι αναφορές σφαλμάτων να δείχνουν στον αρχικό πηγαίο κώδικα και όχι στον ανώνυμο ενταμιευτή (`eval N`):

```
my $code = qq[\n#line ] . __LINE__ . qq[ "] . __FILE__ . qq["\n]
         . q[die "boom"];
eval $code;
# boom at <current file> line <current line>.
```

Διάκριση μεταξύ «εκτελείται ως σενάριο» και «φορτώνεται ως άρθρωμα» - ο συνηθισμένος έλεγχος είναι αν το αρχείο κλήθηκε απευθείας:

```
__PACKAGE__->run(@ARGV) if __FILE__ eq $0;
```

Οριακές περιπτώσεις

- **Κατά τη μεταγλώττιση, όχι κατά τον χρόνο εκτέλεσης.** Το `__FILE__` είναι διακριτικό που ο αναλυτής αντικαθιστά με κυριολεκτική συμβολοσειρά. Δεν είναι συνάρτηση από την οποία μπορείτε να πάρετε αναφορά· το `\&__FILE__` είναι συντακτικό σφάλμα. Μετατρέψτε το σε συμβολοσειρά, αποθηκεύστε το, ή περάστε το - αλλά δεν μπορείτε να αναβάλετε την αποτίμησή του.
- **Σταθερό ανά εμφάνιση.** Η κλήση της ίδιας υπορουτίνας από δύο διαφορετικά αρχεία δεν αλλάζει το `__FILE__` που βλέπει το σώμα της υπορουτίνας· αυτή η τιμή έχει εμπεδωθεί όταν μεταγλωττίστηκε η υπορουτίνα. Αν χρειάζεστε το όνομα αρχείου του καλούντος, χρησιμοποιήστε την `caller`.
- **Οι οδηγίες `#line` επανεγγράφουν την τιμή.** Σχόλιο της μορφής `# line N "NAME"` στη στήλη μηδέν επαναφέρει τόσο τον αριθμό γραμμής όσο και (προαιρετικά) το όνομα αρχείου που αναφέρει ο μεταγλωττιστής. Κάθε διακριτικό `__FILE__` που μεταγλωττίζεται μετά από μια τέτοια οδηγία υιοθετεί το νέο όνομα. Αυτός είναι ο μηχανισμός που χρησιμοποιούν οι γεννήτριες κώδικα και οι μηχανές προτύπων ώστε τα σφάλματα να δείχνουν στον αρχικό πηγαίο κώδικα `.tt` / `.pod` / `.xs` αντί στον παραγόμενο `.pl`.

  ```
  # line 1 "config.tt"
  my $x = __FILE__;       # "config.tt"
  ```

  Η πλήρης σύνταξη της οδηγίας προδιαγράφεται στο `perlsyn` υπό το *Plain Old Comments (Not!)*.

- **Αρθρώματα.** Μέσα σε ένα άρθρωμα, το `__FILE__` είναι η διαδρομή που χρησιμοποίησε η `require` για να φορτώσει το αρχείο - συνήθως απόλυτη και κάτω από κάποια ρίζα του `@INC`. Μην το αναλύετε ως `Some/Module.pm`· χρησιμοποιήστε `dirname(__FILE__)` αν χρειάζεστε τον κατάλογο, όχι χειρουργική σε συμβολοσειρά.
- **Ειδικές πηγές γραμμής εντολών.** Ο κώδικας από `-e` αναφέρει `-e`· ο κώδικας από `-E` αναφέρει επίσης `-e`· κώδικας που διοχετεύεται μέσω stdin αναφέρει `-`. Η string `eval` χωρίς οδηγία `#line` αναφέρει `(eval N)` όπου το `N` είναι ο μετρητής eval.
- **Η `open` ενός ορίσματος και ο σωσίας της `__FILE__`.** Δεν υπάρχει handle `__FILE__`· το διακριτικό δεν έχει καμία σχέση με filehandles. Η `open(__FILE__)` δεν ανοίγει το τρέχον αρχείο πηγαίου κώδικα - προσπαθεί να ανοίξει αρχείο με όνομα κυριολεκτικά το αναπτυγμένο string του διακριτικού, που σπάνια είναι αυτό που θέλετε.
- **Μη τροποποιήσιμο κατά τον χρόνο εκτέλεσης.** Η ανάθεση στο `__FILE__` είναι συντακτικό σφάλμα. Ο μηχανισμός για την αλλαγή του τι αναφέρει το `__FILE__` είναι η οδηγία `#line` κατά τη μεταγλώττιση· δεν υπάρχει αντίστοιχο κατά τον χρόνο εκτέλεσης.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `__LINE__` - συνοδευτικό διακριτικό μεταγλώττισης για τον τρέχοντα αριθμό γραμμής· τα δύο χρησιμοποιούνται σχεδόν πάντα μαζί
- `__PACKAGE__` - διακριτικό μεταγλώττισης για το όνομα του τρέχοντος πακέτου, ίδια οικογένεια ειδικών διακριτικών
- `caller` - αναζήτηση κατά τον χρόνο εκτέλεσης του ονόματος αρχείου, της γραμμής και του πακέτου του καλούντος· χρησιμοποιήστε την όταν χρειάζεστε τη θέση στον πηγαίο κώδικα του καλούντος αντί της δικής σας
- `die` - προσαρτά αυτόματα το `at FILE line LINE.`· καταφύγετε ρητά στο `__FILE__` μόνο όταν καταστέλλετε αυτό το επίθεμα με τελικό newline
- `perlsyn` - η ενότητα *Plain Old Comments (Not!)* προδιαγράφει την οδηγία `#line` που επανεγγράφει το `__FILE__`

Ροή ελέγχου

LINE

Ένα διακριτικό μεταγλώττισης που αποτιμάται στον αριθμό γραμμής στην οποία εμφανίζεται μέσα στον πηγαίο κώδικα.

Το `__LINE__` δεν είναι συνάρτηση. Είναι ένα ειδικό διακριτικό που ο αναλυτής αντικαθιστά, κατά τη μεταγλώττιση, με έναν κυριολεκτικό ακέραιο - τον αριθμό γραμμής της θέσης στο αρχείο όπου βρίσκεται το διακριτικό. Είναι ο τυπικός τρόπος για να σφραγίσετε ένα διαγνωστικό, μια γραμμή καταγραφής ή μια εκτύπωση

αποσφαλμάτωσης με την ακριβή θέση που την παρήγαγε, χωρίς το κόστος μιας αναζήτησης `caller` κατά τον χρόνο εκτέλεσης.

Σύνοψη

```
__LINE__
```

Τι επιστρέφεται

Ένας ακέραιος: ο αριθμός γραμμής του διακριτικού, με αρίθμηση από το 1, μέσα στην τρέχουσα μονάδα μεταγλώττισης. Η τιμή είναι ένας απλός αριθμός, χρησιμοποιήσιμος οπουδήποτε δέχεται αριθμό - σε αριθμητικές πράξεις, σε παρεμβολή σε συμβολοσειρά, ως κλειδί πίνακα κατακερματισμού, ως όρισμα σε `die` ή `warn`.

```
print __LINE__, "\n";        # prints the line number of this line
my $where = "line " . __LINE__;
```

Επειδή η αντικατάσταση γίνεται κατά την ανάλυση, κάθε εμφάνιση του `__LINE__` είναι ουσιαστικά μια διαφορετική σταθερά. Δύο αντίγραφα σε δύο διαφορετικές γραμμές παράγουν δύο διαφορετικούς αριθμούς· ένα μεμονωμένο `__LINE__` μέσα σε βρόχο παράγει τον ίδιο αριθμό σε κάθε επανάληψη.

Κατά τη μεταγλώττιση, όχι κατά τον χρόνο εκτέλεσης

Η τιμή παγώνει όταν μεταγλωττίζεται ο πηγαίος κώδικας. Μια συνηθισμένη παρανόηση:

```
sub where { return __LINE__ }     # returns the line where `__LINE__
↪`
                                  # was written, i.e. the `sub` body
↪'s
                                  # line - NOT the caller's line
```

Για να αποτυπώσετε τον αριθμό γραμμής *του καλούντος*, χρησιμοποιήστε την `caller`:

```
sub where { my (undef, undef, $line) = caller; return $line }
```

Μέσα σε `eval` συμβολοσειράς, το `__LINE__` αριθμεί τις γραμμές της συμβολοσειράς που μεταγλωττίζεται (ξεκινώντας από 1 για την πρώτη γραμμή της συμβολοσειράς), όχι τις γραμμές του περιβάλλοντος αρχείου.

Αλλαγή του αναφερόμενου αριθμού γραμμής

Ο αριθμός γραμμής που αναφέρει το `__LINE__` είναι ο αριθμός γραμμής στον οποίο ο λεκτικός αναλυτής πιστεύει ότι βρίσκεται. Μια οδηγία `#line` επαναφέρει αυτή την πεποίθηση:

```
#line 1000 "virtual.pl"
print __LINE__, "\n";        # prints 1001
```

Το `#line NNN` από μόνο του ρυθμίζει μόνο τον αριθμό γραμμής· το `#line NNN "NAME"` αλλάζει επίσης ό,τι αναφέρει το `__FILE__`. Γεννήτριες κώδικα, μηχανές προτύπων και

προεπεξεργαστές το χρησιμοποιούν για να κάνουν τα διαγνωστικά να δείχνουν πίσω στον αρχικό πηγαίο κώδικα αντί στο παραγόμενο αρχείο. Δείτε το *Plain Old Comments (Not!)* στην ενότητα perlsyn της αναφοράς για την ακριβή σύνταξη και τους κανόνες εμβέλειας.

Παραδείγματα

Σφραγίστε ένα διαγνωστικό με την προέλευσή του:

```
die "bad input at ", __FILE__, " line ", __LINE__, "\n"
    unless defined $x;
```

Η die με μήνυμα που **δεν** τελειώνει σε newline προσαρτά ήδη " at FILE line LINE.\n". Χρησιμοποιήστε την παραπάνω ρητή μορφή μόνο όταν θέλετε να ελέγξετε τη μορφοποίηση, ή όταν έχετε καταστείλει το αυτόματο επίθημα με ένα τελικό newline.

Καταγραφή με πρόθεμα αρχείου και γραμμής:

```
sub logf {
    my ($file, $line) = (__FILE__, __LINE__);
    warn "[$file:$line] @_\n";
}
```

Αυτό καταγράφει τη γραμμή όπου *γράφτηκε* το __LINE__ - μέσα στην logf - κάτι που σπάνια είναι αυτό που θέλετε. Το σωστό μοτίβο χρησιμοποιεί την caller:

```
sub logf {
    my (undef, $file, $line) = caller;
    warn "[$file:$line] @_\n";
}
```

Χρησιμοποιήστε το __LINE__ απευθείας όταν σημειώνετε *αυτό* το κομμάτι κώδικα, όχι τον καλούντα:

```
print STDERR "checkpoint at line ", __LINE__, "\n";
```

Μετατοπίστε τα διαγνωστικά με #line:

```
#line 42 "user-input.pl"
die "syntax error\n";       # dies reporting user-input.pl line 42
```

Οριακές περιπτώσεις

- **Δεν είναι συνάρτηση, δεν καλείται.** Το __LINE__() είναι συντακτικό σφάλμα. Το διακριτικό δεν δέχεται ορίσματα και δεν έχει μορφή με παρενθέσεις.
- **Δεν παρεμβάλλεται μέσα σε συμβολοσειρές σε διπλά εισαγωγικά.** Το "see line __LINE__" εκτυπώνει την κυριολεκτική συμβολοσειρά see line __LINE__. Συνενώστε αντ' αυτού: "see line " . __LINE__.
- **Δεν είναι έγκυρο bareword κλειδιού πίνακα κατακερματισμού στη συνηθισμένη σύντομη μορφή.** Μέσα σε περιβάλλον κλειδιού πίνακα κατακερματισμού { ... }, το $h{__LINE__} λειτουργεί επειδή το διακριτικό

αναλύεται ως έκφραση, αλλά και το `$h{ __LINE__ }` λειτουργεί και είναι σαφέστερο. Το `$h{__LINE__ => 1}` αναλύει το `__LINE__` ως bareword παχιού κόμματος (δηλαδή τη συμβολοσειρά `"__LINE__"`), κάτι που σχεδόν ποτέ δεν είναι αυτό που θέλετε - γράψτε `(__LINE__, 1)` ή `(__LINE__() => 1)` αν θέλετε την κυριολεκτική τιμή.

- **Μέσα σε `eval` συμβολοσειράς**, το `__LINE__` μετράει από τη γραμμή 1 της αποτιμημένης συμβολοσειράς. Προθέστε τη δική σας οδηγία `#line` στη συμβολοσειρά για να κάνετε τα διαγνωστικά να δείχνουν στο περικλείον αρχείο:

```
my $code = qq{#line @{[__LINE__+1]} "@{[__FILE__]}"\n} . $user_
↪code;
eval $code;
```

- **Μέσα στο σώμα ενός here-doc**, το `__LINE__` αποτιμάται στη γραμμή του *ανοίγματος* `<<` στη γραμμή όπου αναφέρεται το here-doc, όχι σε γραμμές μέσα στο σώμα - τα σώματα here-doc είναι δεδομένα, όχι κώδικας.
- **Heredoc και οδηγίες γραμμής.** Το `#line` μέσα σε σώμα here-doc δεν έχει καμία επίδραση· είναι απλώς κείμενο. Το `#line` λειτουργεί μόνο ως οδηγία στην αρχή γραμμής, σε περιβάλλον κώδικα.
- **Αρνητικοί ή μηδενικοί αριθμοί γραμμής.** Το `#line 0` είναι νόμιμο και κάνει την επόμενη φυσική γραμμή να αναφέρεται ως γραμμή 1 (η ίδια η οδηγία είναι γραμμή 0, η γραμμή μετά είναι γραμμή 1). Οι αρνητικοί αριθμοί γίνονται δεκτοί αλλά παράγουν εκπληκτικά διαγνωστικά· μην το κάνετε.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `__FILE__` - το συνοδευτικό διακριτικό για το όνομα του τρέχοντος αρχείου πηγαίου κώδικα· τα δύο σχεδόν πάντα χρησιμοποιούνται μαζί
- `__PACKAGE__` - διακριτικό μεταγλώττισης για το όνομα του τρέχοντος πακέτου, ίδια οικογένεια ειδικών διακριτικών
- `caller` - το ισοδύναμο χρόνου εκτέλεσης όταν χρειάζεστε τον αριθμό γραμμής *του καλούντος*, όχι τη γραμμή του ίδιου του σημείου κλήσης
- `die` - προσαρτά αυτόματα `at FILE line LINE` όταν το μήνυμα δεν έχει τελικό newline, καταργώντας την ανάγκη για ρητό `__LINE__`
- `warn` - ίδια αυτόματη συμπεριφορά εντοπισμού θέσης με την `die` για μη μοιραία διαγνωστικά

Ροή ελέγχου

PACKAGE

Επιστρέφει το όνομα του πακέτου που είναι σε ισχύ στο σημείο όπου εμφανίζεται το token.

Η __PACKAGE__ είναι κυριολεκτικό κατά τη μεταγλώττιση: ο αναλυτής την αντικαθιστά με μια σταθερή συμβολοσειρά που κατονομάζει όποιο πακέτο έθεσε σε ισχύ η πιο πρόσφατη δήλωση `package` για αυτή τη λεξιλογική εμβέλεια. Δεν αναζητεί τίποτα κατά τον χρόνο εκτέλεσης, δεν ακολουθεί την κληρονομικότητα και δεν την αφορά ποια κλάση χρησιμοποιεί ο καλών κώδικας - είναι απλώς το όνομα του πακέτου στο οποίο ανήκει το πηγαίο κείμενο που την περιέχει.

Σύνοψη

```
__PACKAGE__
__PACKAGE__->some_method(@args)
$__PACKAGE__::VERSION          # does NOT work - see Edge cases
```

Τι επιστρέφεται

Μια απλή συμβολοσειρά που περιέχει το πλήρως κατονομασμένο όνομα πακέτου (π.χ. `"My::Thing"`, `"main"`), εσωτερικευμένη κατά τη μεταγλώττιση. Η τιμή είναι σταθερά για όλη τη διάρκεια της περικλείουσας εμβέλειας - δεν μπορεί να αλλάξει κατά τον χρόνο εκτέλεσης και δεν εξαρτάται από τον τρόπο κλήσης του κώδικα.

Εκτός οποιασδήποτε ρητής δήλωσης `package`, η τιμή είναι `"main"`.

Καθολική κατάσταση που επηρεάζει

Καμία. Η __PACKAGE__ επιλύεται κατά τη μεταγλώττιση από το λεξιλογικό πακέτο που είναι σε ισχύ στην πηγαία της θέση· δεν διαβάζει καμία κατάσταση του διερμηνέα όταν εκτελείται και δεν έχει παρενέργειες.

Γιατί να χρησιμοποιήσετε ένα token αντί να γράψετε το όνομα

Τρεις λόγοι, με τη σειρά που τους συναντάτε στην πράξη:

1. **Το αρχείο ή το πακέτο μετονομάζεται και τίποτε άλλο δεν χρειάζεται να αλλάξει.** Κάθε __PACKAGE__ ενημερώνεται αυτόματα· κάθε χειρωνακτικά γραμμένο `"My::Thing"` πρέπει να εντοπιστεί και να διορθωθεί.
2. **Είναι ο μόνος φορητός τρόπος για να γράψετε «αυτή την κλάση» μέσα σε μορφή μπλοκ package**, όπου το πακέτο τελειώνει στην κλείνουσα αγκύλη και ένας εσωτερικός βοηθός δεν μπορεί να υποθέσει ότι το όνομα αρχείου ταιριάζει.
3. **Συνδυάζεται καθαρά με μεθόδους κλάσης**: το __PACKAGE__->new(...) σε έναν βοηθό κατασκευαστή καλεί τη σωστή κλάση είτε το αρχείο φορτώνεται απευθείας είτε αντιγράφεται σε μια υποκλάση που δεν έχει ακόμα επανορίσει τον βοηθό.

Για το αντίστοιχο κατά τον χρόνο εκτέλεσης - «σε ποια κλάση είναι πραγματικά blessed το $self;» - χρησιμοποιήστε το `ref` ή, σε κώδικα με `use feature 'class'`, το `__CLASS__`. Δεν είναι το ίδιο πράγμα· δείτε *Οριακές περιπτώσεις*.

Παραδείγματα

Ανάγνωση του τρέχοντος πακέτου μέσα σε ένα άρθρωμα:

```
package My::Thing;
say __PACKAGE__;                    # My::Thing
```

Χρησιμοποιήστε το __PACKAGE__ για να προσπελάσετε μια μεταβλητή πακέτου χωρίς να κωδικοποιήσετε σταθερά το όνομα - ανθεκτικό σε μετονομασίες:

```
package My::Thing;
our $VERSION = '1.23';

sub version_string {
    no strict 'refs';
    return ${ __PACKAGE__ . '::VERSION' };
}
```

Βοηθός κατασκευαστή που παραμένει σωστός όταν αντιγραφεί σε υποκλάση ή όταν το πακέτο εναλλάσσεται με τη μορφή μπλοκ:

```
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`.
```

Ίχνος εκσφαλμάτωσης που τυπώνει από πού προήλθε μια γραμμή καταγραφής, χωρίς να χρειάζεται να επαναλάβετε το όνομα του πακέτου σε κάθε μήνυμα:

```
package My::Worker;
use v5.36;

sub log_debug {
    my ($msg) = @_;
    warn __PACKAGE__ . ": $msg\n";
}
```

Επίλυση μεταξύ πακέτων με τη μορφή μπλοκ - κάθε __PACKAGE__ βλέπει όποιο package την περιβάλλει:

```
package Outer;
say __PACKAGE__;                    # Outer

package Inner {
    say __PACKAGE__;                # Inner
}

say __PACKAGE__;                    # Outer again
```

Οριακές περιπτώσεις

- **Σταθερά κατά τη μεταγλώττιση, όχι μεταβλητή.** Η `__PACKAGE__` δεν είναι βαθμωτό· οι `$__PACKAGE__` και `${__PACKAGE__}` δεν υπάρχουν. Παρεμβάλλεται σε συμβολοσειρές με διπλά εισαγωγικά ως η συμβολοσειρά στην οποία επιλύεται:

```
package Foo;
say "in @{[__PACKAGE__]}";      # in Foo
say "in $__PACKAGE__";          # prints "in " - $__PACKAGE__ is␣
↪undef
```

  Το ιδίωμα `@{[ ... ]}` είναι ο συντομότερος τρόπος για να την παρεμβάλετε.

- **Η συμβολική αναφορά σε μεταβλητή πακέτου** απαιτεί συνένωση συμβολοσειρών, όχι παρεμβολή sigil:

```
no strict 'refs';
my $v = ${ __PACKAGE__ . '::VERSION' };   # works
my $v = ${"__PACKAGE__::VERSION"};        # WRONG - literal␣
↪package
```

- **Δεν είναι το ίδιο με το `ref $self`.** Η `__PACKAGE__` είναι το σημείο όπου γράφτηκε ο πηγαίος κώδικας· το `ref $self` είναι αυτό στο οποίο πραγματικά έγινε bless το αντικείμενο. Μια μέθοδος που κληρονομείται από μια βασική κλάση επιστρέφει τη βασική κλάση από την `__PACKAGE__`, αλλά την υποκλάση από το `ref $self`:

```
package Base;
sub who { (__PACKAGE__, ref $_[0]) }

package Child;
our @ISA = ('Base');

say join " / ", Child->who;     # Base / Child
```

  Όταν θέλετε την κλάση στην οποία επικλήθηκε η μέθοδος, χρησιμοποιήστε το `ref` στον επικαλούμενο (invocant), το `__CLASS__` μέσα σε ένα μπλοκ `class`/`method`, ή πάρτε το `$_[0]` / `shift` ως το όνομα της κλάσης για μια μέθοδο κλάσης.

- **Εμβέλεια μορφής μπλοκ.** Μέσα σε `package NAME { ... }`, η `__PACKAGE__` είναι `NAME`· μετά την κλείνουσα αγκύλη η προηγούμενη τιμή αποκαθίσταται. Δείτε το εμφωλευμένο παράδειγμα παραπάνω.

- **Το `eval STRING` συλλαμβάνει το πακέτο μεταγλώττισής του.** Η συμβολοσειρά μεταγλωττίζεται στο πακέτο του καλούντος, οπότε η `__PACKAGE__` μέσα σε `eval "..."` είναι το πακέτο που κάλεσε το `eval`:

```
package Foo;
eval 'say __PACKAGE__';          # Foo
```

- **Πριν από οποιαδήποτε δήλωση package** (αρχή σεναρίου, αρχή μονόγραμμου), η `__PACKAGE__` είναι `"main"`.

- **Η `__PACKAGE__` δεν σημαίνει «τρέχουσα κλάση».** Υπό `use feature 'class'`, το όνομα κλάσης και το όνομα πακέτου συμφωνούν στη γραμμή δήλωσης, αλλά

μπορούν να διαφέρουν σε ένα σημείο κλήσης που γίνεται σε υποκλάση. Μέσα σε `method`, προτιμήστε το `__CLASS__` όταν θέλετε την κλάση του παραλήπτη.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `package` - η δήλωση που ορίζει τι επιστρέφει η `__PACKAGE__`· κάθε τιμή που δεν είναι `main` προέρχεται από μία τέτοια
- `__CLASS__` - το αντίστοιχο κατά τον χρόνο εκτέλεσης για κώδικα μέσα σε μπλοκ `class`: επιστρέφει την κλάση του επικαλούμενου στιγμιότυπου, που μπορεί να είναι υποκλάση
- `ref` - ρωτήστε ένα αντικείμενο σε ποια κλάση έγινε bless· το σωστό εργαλείο όταν έχετε το `$self` και θέλετε την πραγματική του κλάση
- `caller` - το πακέτο που καλεί κατά τον χρόνο εκτέλεσης· χρησιμοποιήστε το όταν χρειάζεστε το πακέτο του καλούντος, όχι το δικό σας
- `bless` - η πρωτογενής λειτουργία που μετατρέπει μια αναφορά σε αντικείμενο ενός δεδομένου πακέτου· τυπικά συνοδεύεται από κατασκευαστές τύπου `__PACKAGE__->new(...)`
- `our` - δηλώνει ένα λεξιλογικό ψευδώνυμο προς μια μεταβλητή πακέτου στο τρέχον πακέτο μεταγλώττισης (το ίδιο πακέτο που κατονομάζει η `__PACKAGE__`)

Ροή ελέγχου

SUB

Επιστρέφει μια αναφορά στην υπορουτίνα που εκτελείται τη δεδομένη στιγμή.

Το `__SUB__` είναι το σύμβολο αυτο-αναφοράς: μέσα σε μια sub, αποτιμάται σε μια αναφορά κώδικα που δείχνει σε αυτήν την ίδια sub, χωρίς η sub να χρειάζεται να γνωρίζει το όνομά της. Εκτός κάθε sub, είναι `undef`. Ο συνηθισμένος λόγος για την προσφυγή σε αυτό είναι η ανώνυμη αναδρομή - μια sub αποθηκευμένη σε λεξιλογική μεταβλητή ή που περνά ως callback χρειάζεται έναν τρόπο να καλέσει τον εαυτό της που δεν εξαρτάται από όνομα κατονομασμένο πλήρως με πακέτο, και το `__SUB__` είναι αυτός ο τρόπος.

Σύνοψη

```
__SUB__
__SUB__->(@args)
goto __SUB__
```

Δεν δέχεται ορίσματα και δεν έχει μορφή με παρενθέσεις - είναι σύμβολο κατά τη μεταγλώττιση, όχι κλήση συνάρτησης.

Τι επιστρέφεται

Μια αναφορά κώδικα (το ίδιο πράγμα που θα σας έδινε το `\&name` για μια κατονομασμένη sub), ή `undef` όταν αποτιμάται εκτός οποιασδήποτε υπορουτίνας. Η αναφορά είναι προς την sub σε **χρόνο εκτέλεσης**, συμπεριλαμβανομένης κάθε closure πάνω σε λεξιλογικές μεταβλητές που αποτυπώθηκαν όταν δημιουργήθηκε η sub, οπότε η αναδρομική κλήση της βλέπει το ίδιο αποτυπωμένο περιβάλλον.

```
my $fact = sub {
    my $n = shift;
    $n < 2 ? 1 : $n * __SUB__->($n - 1);
};

print $fact->(5);               # 120
```

Η μορφή κλήσης είναι μια συνηθισμένη επίκληση coderef: `__SUB__->(@args)` ή `&{__SUB__}(@args)`. Κανένα όρισμα δεν περνά σιωπηρά· η αναδρομή πρέπει να τα προωθεί ρητά.

Γιατί να μη χρησιμοποιήσουμε απλώς το όνομα της sub

Οι κατονομασμένες subs μπορούν ήδη να εκτελούν αναδρομή με το όνομά τους - το `factorial(...)` μέσα στη `sub factorial` δουλεύει. Το `__SUB__` δικαιολογεί τη θέση του σε τρεις περιπτώσεις όπου ένα όνομα δεν είναι διαθέσιμο, αξιόπιστο ή ακριβό:

- **Ανώνυμες subs**. Μια ανώνυμη sub που εκχωρείται σε λεξιλογική μεταβλητή δεν έχει σταθερό όνομα κατονομασμένο με πακέτο. Ένας συνηθισμένος τρόπος παράκαμψης πριν την 5.16 ήταν η αποτύπωση μιας αναφοράς προς `\&name` ή η χρήση κόλπων Y-combinator· το `__SUB__` αντικαθιστά και τα δύο.
- **Subs που περνούν ως callbacks**. Μια sub που παραδίδεται σε ένα άλλο API (συγκριτής ταξινόμησης, χειριστής συμβάντων, βήμα επαναλήπτη) ενδέχεται να κληθεί πίσω προτού αποκτήσει όνομα ορατό στο σημείο της αναδρομής.
- **Αναδρομή με κλήση ουράς μέσω `goto`**. Το `goto __SUB__` επαναχρησιμοποιεί το τρέχον πλαίσιο κλήσης και μεταπηδά στην κορυφή της ίδιας sub με όποια τιμή του `@_` έχετε ρυθμίσει, κάτι που έχει σημασία για βαθιά αναδρομή που διαφορετικά θα ανατίναζε τη στοίβα.

Καθολική κατάσταση που επηρεάζει

Το `__SUB__` διαβάζει το πλαίσιο κλήσης που εκτελείται τη δεδομένη στιγμή - το ίδιο πλαίσιο που επιθεωρεί ο `caller`. Δεν διαβάζει ούτε τροποποιεί ειδικές μεταβλητές.

Παραδείγματα

Ανώνυμο παραγοντικό, η σχολική περίπτωση:

```
my $fact = sub {
    my $n = shift;
    $n < 2 ? 1 : $n * __SUB__->($n - 1);
};
print $fact->(6);               # 720
```

Άθροισμα με αναδρομή ουράς μέσω goto __SUB__ - χωρίς αύξηση στοίβας ανά επανάληψη, επειδή το goto επαναχρησιμοποιεί το τρέχον πλαίσιο:

```
my $sum = sub {
    my ($acc, @rest) = @_;
    return $acc unless @rest;
    @_ = ($acc + shift @rest, @rest);
    goto __SUB__;
};
print $sum->(0, 1..1_000_000);  # 500000500000
```

Διάσχιση εμφωλευμένης δομής δεδομένων χωρίς να κατονομαστεί ο διασχιστής:

```
my $count_leaves = sub {
    my $node = shift;
    return 1 unless ref $node eq 'ARRAY';
    my $n = 0;
    $n += __SUB__->($_) for @$node;
    return $n;
};
print $count_leaves->([1, [2, 3], [[4], 5]]);   # 5
```

Εκτός οποιασδήποτε sub, το __SUB__ είναι undef:

```
print defined(__SUB__) ? "yes" : "no";          # no
```

Αποτυπωμένο, όχι αναλυόμενο κατά την κλήση. Το __SUB__ αποτιμάται κάθε φορά που εκτελείται, στο πλαίσιο εντός του οποίου εκτελείται - έτσι σε μια εμφωλευμένη sub αναφέρεται στην **εσωτερική** sub:

```
sub outer {
    my $inner = sub { __SUB__ };    # inner's own coderef
    return $inner->();
}
my $ref = outer();
print $ref == $ref;                 # 1 - stable coderef
```

Οριακές περιπτώσεις

- **Απαιτεί το χαρακτηριστικό current_sub**. Το __SUB__ βρίσκεται πίσω από use feature 'current_sub', ενεργοποιημένο εξ ορισμού υπό use v5.16 ή οποιοδήποτε μεταγενέστερο πακέτο εκδόσεων. Κώδικας που στοχεύει παλαιότερα επίπεδα σύνταξης πρέπει να το ενεργοποιήσει ρητά, ή να προθέσει στο σύμβολο το CORE:: ως CORE::__SUB__.
- **Χωρίς μορφή με παρενθέσεις**. Το __SUB__() είναι συντακτικό σφάλμα - το σύμβολο δεν είναι συνάρτηση, είναι δεσμευμένη λέξη που αποτιμάται σε coderef. Καλέστε το αποτέλεσμα με __SUB__->() ή &{__SUB__}().
- **Εκτός οποιασδήποτε sub**, συμπεριλαμβανομένης της εμβέλειας αρχείου και του εσωτερικού των μπλοκ BEGIN, UNITCHECK, CHECK, INIT και END, το __SUB__ επιστρέφει undef. Η κλήση του (__SUB__->()) σε αυτές τις θέσεις πεθαίνει με Can't use an undefined value as a subroutine reference.

- **Μέσα σε μπλοκ `eval`**, το `__SUB__` αναφέρεται στην **περικλείουσα sub**, όχι στην `eval` - ένα `eval { ... }` δεν δημιουργεί νέο πλαίσιο sub για αυτόν τον σκοπό. Μέσα σε `eval EXPR` ισχύει ο ίδιος κανόνας: το σύμβολο αναλύεται στην εξωτερική sub.
- **Μέσα σε μπλοκ κώδικα regex** (`/(?{...})/` ή `/(??{...})/`), η συμπεριφορά του `__SUB__` υπόκειται σε αλλαγές - το upstream το τεκμηριώνει ως ασταθές. Μη βασίζεστε σε αυτό που επιστρέφει εκεί.
- **Ταυτότητα μεταξύ κλήσεων**. Δύο αποτιμήσεις του `__SUB__` μέσα στην ίδια sub επιστρέφουν το ίδιο coderef (η `==` συγκρίνει ως αληθής). Μεταξύ διαφορετικών closures που φτιάχνονται από το ίδιο `sub { ... }`, κάθε closure βλέπει το δικό του coderef - ένα ανά αποτυπωμένο περιβάλλον.
- **`goto __SUB__` έναντι `__SUB__->()`**. Η μορφή `goto` επαναχρησιμοποιεί το τρέχον πλαίσιο κλήσης και αντικαθιστά το `@_` με ό,τι αναθέσατε πριν το `goto`· δεν μεγαλώνει τη στοίβα κλήσεων. Η μορφή κλήσης coderef δημιουργεί νέο πλαίσιο κάθε φορά, οπότε χτυπά το όριο βάθους αναδρομής σε βαθιά αναδρομικές εισόδους.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `sub` - ορίζει την υπορουτίνα της οποίας την αναφορά επιστρέφει το `__SUB__`· το πρώτο μέρος που πρέπει να κοιτάξετε όταν η ανώνυμη αναδρομή είναι το εργαλείο για τη δουλειά
- `goto` - συνεργάζεται με το `__SUB__` για αναδρομή με κλήση ουράς μέσω `goto __SUB__`, το ιδίωμα που αποφεύγει την αύξηση της στοίβας
- `caller` - επιθεώρηση του ίδιου πλαισίου κλήσης από έξω (πακέτο, αρχείο, γραμμή) αντί για λήψη αναφοράς σε αυτό
- `ref` - κατηγοριοποίηση αυτού που επέστρεψε το `__SUB__`· πάντα `"CODE"` μέσα σε sub, κενή συμβολοσειρά εκτός
- `return` - ο συνηθισμένος τρόπος εξόδου από τη sub της οποίας η αναφορά είναι το `__SUB__`· σχετικός όταν επιλέγει κανείς μεταξύ αναδρομής και κλήσης ουράς μέσω `goto __SUB__`

Αριθμητικές συναρτήσεις

abs

Επιστρέφει την απόλυτη τιμή ενός αριθμού.

Η `abs` παίρνει ένα μοναδικό βαθμωτό, το εξαναγκάζει σε αριθμητικό περιβάλλον και επιστρέφει το μέτρο του - την τιμή με αφαιρεμένο το πρόσημο. Οι μη αριθμητικές συμβολοσειρές μετατρέπονται με τους ίδιους κανόνες όπως στην αριθμητική (αρχικά κενά και προαιρετικό πρόσημο, ακολουθούμενα από ψηφία· τα υπόλοιπα απορρίπτονται, με προειδοποίηση υπό `use warnings`). Αν το `VALUE` παραλειφθεί, η `abs` λειτουργεί επί του `$_`.

Σύνοψη

```
abs VALUE
abs
abs($x)
```

Τι επιστρέφεται

Ένας μη αρνητικός αριθμός. Το αποτέλεσμα είναι ακέραιος αν η είσοδος ήταν ακέραιος εντός του εύρους ακεραίων της πλατφόρμας, διαφορετικά αριθμός κινητής υποδιαστολής. Η `abs` δεν αλλάζει ποτέ το όρισμά της - το αποτέλεσμα είναι ένα νέο βαθμωτό, οπότε η `abs $x` αφήνει το `$x` ανέπαφο.

Καθολική κατάσταση που επηρεάζει

Χωρίς όρισμα, η `abs` διαβάζει το `$_`. Δεν γράφει ούτε διαβάζει καμία άλλη ειδική μεταβλητή. Υπό `use warnings`, ένα μη αριθμητικό όρισμα προκαλεί την τυπική προειδοποίηση `Argument "…" isn't numeric` μέσω της κανονικής διαδρομής αριθμητικής μετατροπής.

Παραδείγματα

Αφαίρεση του προσήμου από έναν ακέραιο:

```
print abs(-7);              # 7
print abs(7);               # 7
```

Μέτρο κινητής υποδιαστολής - το αποτέλεσμα διατηρεί την πλήρη ακρίβεια της εισόδου:

```
print abs(-3.14159);        # 3.14159
print abs(-1e-10);          # 1e-10
```

Χρήση της μορφής με προεπιλεγμένο όρισμα μέσα σε βρόχο επί του `$_`:

```
for (-3, -1, 0, 2, 4) {
    print abs, "\n";        # 3, 1, 0, 2, 4 on separate lines
}
```

Απόσταση μεταξύ δύο σημείων σε μια αριθμητική γραμμή:

```
sub distance { abs($_[0] - $_[1]) }
print distance(10, 3);      # 7
print distance(3, 10);      # 7
```

Συμβολοσειρά που μοιάζει με αρνητικό αριθμό - η `abs` αναλύει το αρχικό αριθμητικό πρόθεμα και αφαιρεί το πρόσημο:

```
print abs("-42");           # 42
print abs("  -3.5xyz");     # 3.5  (warning under use warnings)
```

Οριακές περιπτώσεις

- Η **`abs(undef)`** επιστρέφει `0`. Υπό `use warnings` εκπέμπει `Use of uninitialized value in abs`.
- **Μη αριθμητική συμβολοσειρά**: η `abs("hello")` επιστρέφει `0` και υπό `use warnings` προειδοποιεί ότι το όρισμα δεν είναι αριθμητικό.
- **Όριο υπερχείλισης ακεραίου**: η `abs` του πιο αρνητικού ακεραίου της πλατφόρμας (`PERL_INT_MIN`, π.χ. `-2**63` σε 64-bit συστήματα) προάγεται σε αποτέλεσμα κινητής υποδιαστολής, επειδή το θετικό αντίστοιχο δεν χωράει σε προσημασμένο ακέραιο:

```
use Config;
my $min = -(2 ** ($Config{ivsize} * 8 - 1));
print abs($min);            # 9.22337203685478e+18 on 64-bit
```

- **Ειδικές τιμές IEEE 754**: η `abs(-0.0)` επιστρέφει `0` (θετικό μηδέν). Η `abs("-Inf")` επιστρέφει `Inf`. Η `abs("NaN")` επιστρέφει `NaN` - το `NaN` δεν έχει πρόσημο που η `abs` να μπορεί να αφαιρέσει με νόημα.
- **Μιγαδικοί αριθμοί και μεγάλοι ακέραιοι** δεν υποστηρίζονται από την ενσωματωμένη συνάρτηση. Η `use Math::Complex` υπερφορτώνει την `abs` ώστε να υπολογίζει το μέτρο· τα και παρέχουν τη δική τους αφαίρεση προσήμου σε αντικείμενα blessed μέσω υπερφόρτωσης τελεστών.
- **Δεν υπάρχει επιτόπια μορφή**: η `abs` επιστρέφει πάντα νέο βαθμωτό. Για να αντικαταστήσετε την τιμή μιας μεταβλητής με το μέτρο της, εκχωρήστε ξανά:

```
$x = abs $x;
```

- **Σημείωση συντακτικού αναλυτή**: η μοναδιαία μορφή `abs $x` συνδέεται πιο σφιχτά από τον `,` αλλά πιο χαλαρά από τους περισσότερους αριθμητικούς τελεστές, οπότε το `abs $x + 1` αναλύεται ως `abs($x) + 1`, όχι ως `abs($x + 1)`. Βάλτε παρενθέσεις όταν αμφιβάλλετε:

```
print abs $x + 1;           # abs($x) + 1
print abs($x + 1);          # abs of the sum
```

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `sqrt` - τετραγωνική ρίζα· όπως και η `abs` είναι καθαρή αριθμητική μοναδιαία συνάρτηση με μορφή προεπιλογής στο `$_`
- `int` - αποκοπή προς το μηδέν· συνδυάζεται με την `abs` όταν θέλετε το ακέραιο μέτρο ενός αριθμού κινητής υποδιαστολής
- `sprintf` - μορφοποιημένη αριθμητική έξοδος· χρησιμοποιήστε `%d` ή `%g` σε συνδυασμό με την `abs` για να αποδώσετε μη προσημασμένα μέτρα
- `$_` - το προεπιλεγμένο όρισμα όταν η `abs` καλείται χωρίς όρισμα

Υποδοχές (sockets)

accept

Αποδοχή εισερχόμενης σύνδεσης σε μια υποδοχή που ακούει.

Η `accept` αποτελεί τη διακομιστική πλευρά της χειραψίας σύνδεσης. Περιμένει στο `GENERICSOCKET` - μια υποδοχή που έχει ήδη δημιουργηθεί με `socket`, συνδεθεί με `bind` και τεθεί σε κατάσταση ακρόασης με `listen` - μέχρι να συνδεθεί ένας πελάτης, οπότε εγκαθιστά μια ολοκαίνουρια συνδεδεμένη υποδοχή στο `NEWSOCKET` και επιστρέφει την πακεταρισμένη διεύθυνση του πελάτη. Το `GENERICSOCKET` παραμένει σε κατάσταση ακρόασης, έτοιμο για την επόμενη κλήση. Η συμπεριφορά αντικατοπτρίζει την κλήση συστήματος POSIX `accept(2)`.

Σύνοψη

```
accept NEWSOCKET, GENERICSOCKET
```

Τι επιστρέφεται

Η διεύθυνση του πελάτη στην πακεταρισμένη μορφή που χρησιμοποιείται από την οικογένεια διευθύνσεων του `GENERICSOCKET` - η ίδια μορφή που θα περνούσατε στη `connect` ή θα αποπακετάρατε με `Socket::unpack_sockaddr_in` για IPv4, ή με `Socket::unpack_sockaddr_in6` / `unpack_sockaddr_un` για IPv6 και υποδοχές πεδίου Unix αντίστοιχα. Σε αποτυχία επιστρέφει ψευδή τιμή (την κενή συμβολοσειρά) και θέτει την `$!` στο υποκείμενο `errno`.

Το `NEWSOCKET` συμπληρώνεται ως παρενέργεια - είναι ένα απλό όρισμα filehandle, όχι τιμή επιστροφής. Μια bareword όπως το `CLIENT` αυτο-δημιουργεί ένα typeglob· ένα λεκτικό filehandle όπως το `my $client` συμπληρώνεται επιτόπου:

```
my $client;
accept($client, $server) or die "accept: $!";
# $client is now a readable/writable handle to the connected peer
```

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται σε αποτυχία στο υποκείμενο `errno` (`EINTR`, `EAGAIN`/`EWOULDBLOCK` σε μη μπλοκαριστικές υποδοχές ακρόασης, `ECONNABORTED`, `EMFILE`, `ENFILE`).
- `$^F` - ο μέγιστος περιγραφέας αρχείου του συστήματος. Σε συστήματα που υποστηρίζουν τη σημαία close-on-exec, η `accept` θέτει `FD_CLOEXEC` στον νέο περιγραφέα όταν ο αριθμός του είναι μεγαλύτερος από το `$^F` (προεπιλογή 2, καλύπτοντας τα τυπικά ρεύματα). Αυξήστε το `$^F` πριν την `accept` αν θέλετε η υποδοχή που έγινε αποδεκτή να επιβιώσει της `exec`.

Παραδείγματα

Ελάχιστος βρόχος αποδοχής τύπου TCP echo:

```
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
```

Αποπακετάρισμα της επιστρεφόμενης διεύθυνσης για καταγραφή:

```
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;
```

Μη μπλοκαριστική accept. Θέστε πρώτα το $server σε μη μπλοκαριστική κατάσταση· μια κλήση χωρίς εκκρεμή σύνδεση επιστρέφει τότε ψευδή τιμή με την $! σε EAGAIN ή EWOULDBLOCK:

```
use Errno qw(EAGAIN EWOULDBLOCK);
use Fcntl qw(F_GETFL F_SETFL O_NONBLOCK);

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: $!";
    }
}
```

Διατηρήστε την αποδεκτή υποδοχή κατά τη διάρκεια της exec αυξάνοντας το $^F ώστε ο χρόνος εκτέλεσης να μη θέσει FD_CLOEXEC:

```
local $^F = 10_000;
accept(my $client, $server) or die "accept: $!";
exec "/usr/libexec/handler", fileno($client);
```

Οριακές περιπτώσεις

- **Το GENERICSOCKET δεν ακούει** - η `accept` αποτυγχάνει με την `$!` σε `EINVAL`. Καλέστε πρώτα την `listen` στην υποδοχή του διακομιστή.
- **Το NEWSOCKET είναι ήδη ανοιχτό** - η υπάρχουσα λαβή κλείνει σιωπηλά πριν εγκατασταθεί ο νέος περιγραφέας, ακριβώς όπως στις `open` και `socket`. Αν η παλιά λαβή ήταν η μόνη αναφορά σε ένα ρεύμα με ενταμιευτή, τυχόν μη εκκενωμένα δεδομένα χάνονται. Κλείστε τη ρητά εκ των προτέρων όταν αυτό έχει σημασία.
- **Σήμα κατά την κλήση** - ένα σήμα που παραδίδεται ενώ η `accept` είναι μπλοκαρισμένη κάνει την υποκείμενη κλήση συστήματος να επιστρέψει `EINTR`. Το pperl δεν επανεκκινεί αυτόματα: η `accept` επιστρέφει ψευδή τιμή και αφήνει την `$!` σε `EINTR`. Τυλίξτε σε βρόχο επανάληψης αν οι χειριστές σημάτων σας δεν τερματίζουν τη διεργασία.
- **Εξάντληση περιγραφέων** - το `EMFILE` (όριο διεργασίας) ή το `ENFILE` (όριο συστήματος) σε έναν φορτωμένο διακομιστή είναι ο συνηθισμένος λόγος που η `accept` αρχίζει να αποτυγχάνει υπό φορτίο. Αντιμετωπίστε τα ως παροδικά· κλείστε αδρανείς πελάτες και ξαναδοκιμάστε.
- **Αναντιστοιχία οικογένειας διευθύνσεων στην πακεταρισμένη επιστροφή** - ο πακεταρισμένος ενταμιευτής ακολουθεί την οικογένεια του `GENERICSOCKET`. Το αποπακετάρισμα μιας διεύθυνσης πεδίου Unix με `Socket::unpack_sockaddr_in` παράγει σκουπίδια. Παρακολουθήστε εσείς την οικογένεια ή χρησιμοποιήστε την `Socket::sockaddr_family` για να γίνει η αποστολή.
- **Οι παρενθέσεις είναι προαιρετικές, τα κόμματα όχι** - τα `accept $c, $s` και `accept($c, $s)` λειτουργούν και τα δύο· το `accept $c $s` είναι συντακτικό σφάλμα.
- **Περιβάλλον λίστας έναντι βαθμωτού** - η πακεταρισμένη διεύθυνση επιστροφής είναι συμβολοσειρά bytes· το περιβάλλον λίστας **δεν** την επεκτείνει σε `(port, addr)`. Καλέστε την `sockaddr_in` / αντίστοιχη για αποπακετάρισμα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `socket` - δημιουργεί την υποδοχή διακομιστή την οποία περιμένει η `accept`
- `bind` - προσαρτά την υποδοχή διακομιστή σε τοπική διεύθυνση πριν τις `listen` και `accept`
- `listen` - υποχρεωτικό βήμα μεταξύ της `bind` και της `accept`· καθορίζει το βάθος της ουράς εκκρεμοτήτων
- `connect` - το αντίστοιχο της πλευράς πελάτη· η `accept` επιστρέφει τη διεύθυνση που έδωσε μια κλήση `connect`
- `getpeername` - ανακτά την ίδια πακεταρισμένη διεύθυνση αργότερα από το ίδιο το `NEWSOCKET`

- `Socket` - οι σταθερές (`PF_INET`, `SOCK_STREAM`, `INADDR_ANY`) και οι packers (`sockaddr_in`, `unpack_sockaddr_in`) που μετατρέπουν την πακεταρισμένη τιμή επιστροφής σε κάτι αξιοποιήσιμο
- `$^F` - κατώφλι που ελέγχει το αν η `accept` θέτει `FD_CLOEXEC` στον νέο περιγραφέα

Διεργασίες

alarm

Προγραμματίζει την παράδοση ενός `SIGALRM` στην τρέχουσα διεργασία μετά από έναν ακέραιο αριθμό δευτερολέπτων πραγματικού χρόνου.

Η `alarm` οπλίζει έναν μοναδικό χρονομετρητή αντίστροφης μέτρησης ανά διεργασία. Όταν λήξει ο χρονομετρητής, ο πυρήνας παραδίδει το `SIGALRM`· τι θα συμβεί στη συνέχεια εξαρτάται εξ ολοκλήρου από τον εγκατεστημένο χειριστή σήματος στο `%SIG`. Ο χρονομετρητής τρέχει σε πραγματικό χρόνο, όχι σε χρόνο CPU, οπότε μια διεργασία που κοιμάται ή είναι μπλοκαρισμένη λαμβάνει το σήμα στην ώρα της (με επιφύλαξη μικρών αποκλίσεων χρονοπρογραμματισμού).

Σύνοψη

```
alarm SECONDS
alarm
```

Χωρίς όρισμα, χρησιμοποιείται η τιμή του `$_`.

Τι επιστρέφεται

Ο αριθμός των δευτερολέπτων που απέμεναν στον **προηγούμενο** χρονομετρητή, ή `0` αν δεν εκκρεμούσε χρονομετρητής. Αυτό επιτρέπει να αποθηκεύσετε και να επαναφέρετε το alarm ενός καλούντος γύρω από μια εμφωλευμένη χρονομετρημένη λειτουργία:

```
my $prev = alarm 10;
# ... do something that must finish in 10s ...
alarm $prev;                        # restore caller's timer
```

Η `alarm 0` ακυρώνει κάθε εκκρεμή χρονομετρητή και επιστρέφει τα δευτερόλεπτα που είχαν απομείνει σε αυτόν.

Καθολική κατάσταση που επηρεάζει

- `%SIG` - η θυρίδα `ALRM` ονοματίζει τον χειριστή που καλείται όταν ενεργοποιείται ο χρονομετρητής. Χωρίς χειριστή, η προεπιλεγμένη συμπεριφορά του `SIGALRM` είναι ο τερματισμός της διεργασίας.
- `$_` - διαβάζεται όταν η `alarm` καλείται χωρίς όρισμα.
- `$@` - μεταφέρει το μήνυμα `die` έξω από το κανονικό ιδίωμα `eval { alarm ...; ...; alarm 0 }`.
- `$!` - μια μπλοκαριστική κλήση συστήματος που διακόπτεται από `SIGALRM` θέτει το `$!` σε `EINTR`, **αλλά** η Perl επανεκκινεί αυτόματα πολλές κλήσεις συστήματος

σε ορισμένα συστήματα, οπότε δεν μπορείτε να βασιστείτε ότι θα δείτε EINTR. Χρησιμοποιήστε `eval` / `die` αντ" αυτού (δείτε παρακάτω).

Το κανονικό ιδίωμα χρονικού ορίου

Χρησιμοποιήστε την `alarm` με `eval` και `die` για να βάλετε ανώτατο όριο πραγματικού χρόνου σε οποιοδήποτε μπλοκ κώδικα, περιλαμβανομένης μπλοκαριστικής 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 ...
}
```

Τρεις λεπτομέρειες είναι κρίσιμες:

- Η `local` περιορίζει το εμβέλειο του χειριστή στο μπλοκ `eval` ώστε να μη διαρρέει στον καλούντα.
- Το `\n` στο `"timeout\n"` καταστέλλει το επίθεμα αρχείου/γραμμής που η `die` θα πρόσθετε διαφορετικά, καθιστώντας αξιόπιστο τον έλεγχο ακριβούς αντιστοιχίας στο `$@`.
- Η `alarm 0` **μέσα** στο `eval` ακυρώνει τον χρονομετρητή στη διαδρομή επιτυχίας. Χωρίς αυτή, ένας χρονομετρητής που έμεινε οπλισμένος μπορεί να ενεργοποιηθεί κατά τη διάρκεια άσχετου κώδικα αφού επιστρέψει το `eval`.

Παραδείγματα

Απλή προθεσμία πραγματικού χρόνου:

```
$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";
```

Εμφωλευμένοι χρονομετρητές - αποθήκευση και επαναφορά:

```
my $saved = alarm 30;               # remember caller's timer
do_something();
alarm $saved;                       # put it back
```

Ακύρωση εκκρεμούς χρονομετρητή:

```
my $left = alarm 0;                     # 0 if none was pending
```

Περιοδικός παλμός χωρίς `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
```

Καθυστέρηση μικρότερη του δευτερολέπτου - χρησιμοποιήστε `Time::HiRes`:

```
use Time::HiRes qw(alarm);
alarm 0.25;                             # 250 ms
```

Οριακές περιπτώσεις

- **Μόνο ακέραια δευτερόλεπτα.** Η πυρηνική `alarm` αποκόπτει κλασματικά ορίσματα σε ακέραια δευτερόλεπτα. Για λεπτότερη ακρίβεια, φορτώστε το `Time::HiRes`, που εξάγει μια αντικαταστάτη `alarm` η οποία δέχεται αριθμούς κινητής υποδιαστολής.
- **Απόκλιση χρονοπρογραμματισμού.** Το σήμα μπορεί να φτάσει έως και περίπου ένα δευτερόλεπτο νωρίτερα ή αργότερα λόγω του τρόπου με τον οποίο ο πυρήνας μετράει τα δευτερόλεπτα και προγραμματίζει τη διεργασία σας. Αντιμετωπίστε την `alarm N` ως ένα όριο *όχι-πριν-από-N-μείον-1*, όχι ως ακριβή προθεσμία.
- **Ένας χρονομετρητής ανά διεργασία.** Κάθε κλήση `alarm` αντικαθιστά τον προηγούμενο χρονομετρητή· δεν υπάρχει ουρά. Η τιμή επιστροφής είναι η μόνη καταγραφή σας για ό,τι αντικαταστάθηκε.
- **Η προεπιλεγμένη συμπεριφορά είναι μοιραία.** Χωρίς εγκατεστημένο χειριστή για το `ALRM`, η λήξη του χρονομετρητή τερματίζει τη διεργασία με `Alarm clock`. Εγκαταστήστε έναν χειριστή *πριν* οπλίσετε τον χρονομετρητή.
- **Μην το συνδυάζετε με `sleep`.** Σε πολλά συστήματα η `sleep` υλοποιείται πάνω στην `alarm`, οπότε το να οπλίσετε τη μία ακυρώνει την άλλη. Διαλέξτε μία ανά μπλοκ.
- **Το `EINTR` δεν είναι αξιόπιστο.** Η Perl εγκαθιστά χειριστές σημάτων με `SA_RESTART` σε πολλές πλατφόρμες, οπότε ένα `SIGALRM` που διακόπτει μια μπλοκαριστική κλήση συστήματος επανεκκινεί διαφανώς την κλήση αντί να επιστρέψει με την `$!` σε `EINTR`. Ο φορητός τρόπος για την επιβολή προθεσμίας είναι `eval` / `die` από τον χειριστή, όπως φαίνεται παραπάνω.
- **Τα σήματα παραδίδονται μεταξύ των τελεστών Perl.** Μέσα σε σφιχτό βρόχο καθαρής Perl ο χειριστής τρέχει στο επόμενο ασφαλές σημείο, όχι ακαριαία· μια κλήση XS που δεσμεύει την CPU μπορεί να καθυστερήσει περαιτέρω την παράδοση.
- **Η `fork` εκκαθαρίζει τον χρονομετρητή.** Οι θυγατρικές διεργασίες ξεκινούν χωρίς εκκρεμές alarm ανεξάρτητα από το τι είχε οπλίσει η γονική.

- **Η exec εκκαθαρίζει τον χρονομετρητή** στο Linux· η εικόνα της διεργασίας αντικατάστασης ξεκινά χωρίς εκκρεμές SIGALRM.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `sleep` - καθυστέρηση πραγματικού χρόνου· συχνά υλοποιείται πάνω στην `alarm`, οπότε οι δύο δεν συνδυάζονται
- `die` - εγείρει από τον χειριστή ALRM την εξαίρεση που πιάνει η `eval` για την υλοποίηση χρονικού ορίου
- `eval` - πιάνει το `die` από τον χειριστή και επιτρέπει στο πρόγραμμα να ανακάμψει από χρονικό όριο
- `%SIG` - όπου εγκαθίσταται ο χειριστής ALRM· ο χειριστής αποφασίζει τι *σημαίνει* ο χρονομετρητής
- `select` - η μορφή τεσσάρων ορισμάτων δέχεται χρονικό όριο κινητής υποδιαστολής και αποτελεί εναλλακτική της `alarm` για προθεσμίες I/O
- `Time::HiRes` - απευθείας αντικαταστάτης της `alarm` που δέχεται αριθμούς κινητής υποδιαστολής μικρότερους του δευτερολέπτου μέσω `setitimer(2)`

Λίστες

all

Ελέγχει αν το BLOCK επιστρέφει αληθές για **κάθε** στοιχείο της LIST.

Η `all` διατρέχει τη LIST από αριστερά προς τα δεξιά, ψευδωνυμώντας το `$_` σε κάθε στοιχείο διαδοχικά και αξιολογώντας το BLOCK με εκείνο το στοιχείο. Επιστρέφει αληθή τιμή αν κάθε αξιολόγηση δίνει αλήθεια, και ψευδή τιμή μόλις κάποια αξιολόγηση δώσει ψεύδος. Μόλις ένα στοιχείο κάνει το μπλοκ ψευδές, η `all` σταματά και επιστρέφει - δεν αξιολογεί το μπλοκ για τα υπόλοιπα στοιχεία.

Αυτή είναι η `all` σε επίπεδο γλώσσας ως δεσμευμένη λέξη, ενεργοποιούμενη από το χαρακτηριστικό `keyword_all`. Είναι διακριτή από την `List::Util::all` - δείτε *Οριακές περιπτώσεις* παρακάτω.

Σύνοψη

```
use feature 'keyword_all';
no warnings 'experimental::keyword_all';

all { /\w+/ } @words;
all { defined $_ } @list;        # canonical "all defined" check
all { $_ > 0 } @numbers;
```

Τι επιστρέφεται

Μια λογική τιμή. Αληθής αν κάθε στοιχείο περάσει, ψευδής διαφορετικά. Οι ακριβείς τιμές αληθές/ψευδές είναι απροσδιόριστες - αντιμετωπίστε το αποτέλεσμα ως συνθήκη, όχι ως δεδομένο προς διάδοση.

Η κενή λίστα επιστρέφει αληθές. Αυτή είναι η *κενή αλήθεια*: η δήλωση «κάθε στοιχείο ικανοποιεί το BLOCK» είναι τετριμμένα αληθής όταν δεν υπάρχουν στοιχεία προς έλεγχο. Είναι η πλέον συνηθισμένη πηγή έκπληξης με την `all`:

```
all { $_ > 0 } ();                # TRUE - no elements, nothing to
↪fail
```

Προστατευτείτε με ρητό έλεγχο μεγέθους όταν μια κενή λίστα πρέπει να αντιμετωπίζεται ως αποτυχία:

```
if (@numbers && all { $_ > 0 } @numbers) { ... }
```

Καθολική κατάσταση που επηρεάζει

- `$_` - ψευδωνυμίζεται σε κάθε στοιχείο της `LIST` διαδοχικά για τη διάρκεια του μπλοκ. Επειδή πρόκειται για ψευδώνυμο και όχι αντίγραφο, η εκχώρηση στο `$_` μέσα στο μπλοκ μεταβάλλει το υποκείμενο στοιχείο της λίστας. Η προηγούμενη τιμή του `$_` επαναφέρεται όταν επιστρέφει η `all`.

Παραδείγματα

Έλεγχος ότι κάθε τιμή σε μια λίστα είναι ορισμένη - η κανονική χρήση:

```
my @row = get_record();
die "missing field" unless all { defined $_ } @row;
```

Έλεγχος ότι κάθε αριθμός είναι θετικός:

```
my @prices = (1.99, 4.50, 0.99);
say "ok" if all { $_ > 0 } @prices;          # ok
```

Επικύρωση εγγραφής όπου κάθε πεδίο πρέπει να ταιριάζει στο δικό του κατηγόρημα. Συνδυάστε πολλαπλούς ελέγχους μέσα στο μπλοκ αντί να αλυσιδώνετε κλήσεις της `all`:

```
my @fields = ($name, $age, $email);
my $ok = all {
    defined $_ && length $_ > 0
} @fields;
```

Συνδυάστε την `all` με την `any` για έναν αποκλειστικό έλεγχο - τουλάχιστον μία αντιστοιχία, αλλά όχι κάθε στοιχείο ίδιο:

```
if (any { $_ eq 'admin' } @roles
    && !all { $_ eq 'admin' } @roles) {
    say "mixed roles, including admin";
}
```

Συμπεριφορά βραχυκυκλώματος: το μπλοκ τρέχει μόνο μέχρι την πρώτη αποτυχία. Χρήσιμο για κατηγορήματα με παρενέργειες ή κόστος:

```
all { expensive_check($_) } @candidates;    # stops at first failure
```

Οριακές περιπτώσεις

- **Η κενή λίστα επιστρέφει αληθές.** Κενή αλήθεια. Προστατευτείτε με `@list && all { ... } @list` όταν το άδειασμα πρέπει να αποτυγχάνει.
- **Βραχυκυκλώνει στο πρώτο ψευδές.** Το μπλοκ δεν καλείται για κανένα στοιχείο μετά την πρώτη αποτυχία. Μη βασίζεστε σε παρενέργειες του μπλοκ που να εκτελούνται για κάθε στοιχείο - χρησιμοποιήστε βρόχο `for` αν τις χρειάζεστε.
- **Το `$_` είναι ψευδώνυμο, όχι αντίγραφο.** Η τροποποίηση του `$_` μέσα στο μπλοκ τροποποιεί το στοιχείο της λίστας:

```
my @xs = (1, 2, 3);
all { $_++; $_ > 0 } @xs;
# @xs is now (2, 3, 4)
```

- **Experimental under 5.44.** The feature emits a compile-time warning in the `experimental::keyword_all` category unless that category is silenced:

```
no warnings 'experimental::keyword_all';
```

- **Δεν είναι το ίδιο με την `List::Util::all`.** Η ενσωματωμένη είναι δεσμευμένη λέξη της γλώσσας ενεργοποιούμενη από το χαρακτηριστικό `keyword_all` και αναλύει το `BLOCK` ως πραγματικό μπλοκ (χωρίς αρχική `sub`, χωρίς τελευταίο κόμμα). Η `List::Util::all` είναι κανονική υπορουτίνα εισαγμένη από άρθρωμα και δέχεται όρισμα αναφοράς κώδικα / πρωτοτύπου μπλοκ. Για τους περισσότερους πρακτικούς σκοπούς συμπεριφέρονται το ίδιο σε μη κενές λίστες, αλλά:
  - Η μορφή ως δεσμευμένη λέξη είναι κατασκευή σε επίπεδο συντακτικού αναλυτή και μπορεί να βελτιστοποιηθεί πιο επιθετικά.
  - Οι δύο δεν μοιράζονται ούτε υλοποίηση ούτε κατηγορία προειδοποίησης.
  - Code that needs to run on pre-5.44 perls should use `List::Util::all`.
- **Ισοδύναμη μορφή με `grep`.** Το `all { COND } @xs` είναι σημασιολογικά ισοδύναμο με `@xs == grep { COND } @xs` - αλλά η `grep` αξιολογεί πάντα το μπλοκ για κάθε στοιχείο, οπότε στερείται τη συμπεριφορά βραχυκυκλώματος της `all`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- any - αληθές αν το μπλοκ επιστρέφει αληθές για **τουλάχιστον ένα** στοιχείο· ο κατοπτρικός τελεστής της all
- grep - αξιολογεί το μπλοκ για κάθε στοιχείο και επιστρέφει τις αντιστοιχίες· το scalar @list == grep { … } @list είναι η all χωρίς το βραχυκύκλωμα
- defined - το κατηγόρημα μέσα στον κανονικό έλεγχο all { defined $_ } @list
- List::Util::all - module-level equivalent available on perls older than 5.44; also provides none and notall which have no language-level counterpart
- List::Util::any - αντίστοιχο σε επίπεδο αρθρώματος της δεσμευμένης λέξης any, μαζί με τις none και notall για τις αντίθετες πολικότητες

Λίστες

any

Επιστρέφει αληθές αν το BLOCK δίνει αληθές για τουλάχιστον ένα στοιχείο της LIST.

Η any εκτελεί το BLOCK μία φορά ανά στοιχείο της LIST, ψευδωνυμώντας το $_ στο τρέχον στοιχείο, και επιστρέφει αληθή τιμή τη στιγμή που το μπλοκ δίνει αληθές. Αν κανένα στοιχείο δεν κάνει το μπλοκ αληθές - συμπεριλαμβανομένης της περίπτωσης όπου η LIST είναι κενή - η any επιστρέφει ψευδές. Δεν εκτελεί ποτέ το μπλοκ περισσότερες φορές από όσες χρειάζεται: το πρώτο αληθές αποτέλεσμα αποφασίζει την απάντηση και τα υπόλοιπα στοιχεία δεν εξετάζονται.

any is not unconditionally available. It is a feature-gated core keyword introduced in Perl 5.44; 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 5.44.

Σύνοψη

```
use feature 'keyword_any';
any BLOCK LIST
```

Τι επιστρέφεται

Λογική τιμή: αληθή αν κάποια επανάληψη του BLOCK έδωσε αληθές, ψευδή διαφορετικά. Η ακριβής μορφή της τιμής αληθές/ψευδές είναι απροσδιόριστη - αντιμετωπίστε την επιστροφή ως λογική, μη βασίζεστε ότι θα λάβετε ειδικά 1 και "".

Μια κενή LIST επιστρέφει ψευδές. Αυτή είναι η μαθηματικά σωστή απάντηση (ο υπαρξιακός ποσοδείκτης πάνω σε κενό πεδίο είναι ψευδής) και ταιριάζει με την List::Util::any.

Καθολική κατάσταση που επηρεάζει

- Το `$_` ψευδωνυμίζεται σε κάθε στοιχείο της `LIST` διαδοχικά ενώ τρέχει το `BLOCK`. Η ψευδωνυμοποίηση είναι του ίδιου είδους που χρησιμοποιούν οι `map` και `grep`: η τροποποίηση του `$_` μέσα στο μπλοκ τροποποιεί το ίδιο το στοιχείο της λίστας. Χρησιμοποιήστε μια ονομασμένη λεξιλογική μεταβλητή μέσω `my $x = $_` στην πρώτη γραμμή του μπλοκ αν χρειάζεστε ασφαλές αντίγραφο.

Παραδείγματα

Έλεγχος αν κάποια γραμμή σε μια λίστα ταιριάζει σε ένα μοτίβο:

```
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";
}
```

Έλεγχος αριθμητικού κατωφλίου:

```
use feature 'keyword_any';
my @nums = (1, 4, 7, 12, 3);
print "over limit\n" if any { $_ > 10 } @nums;   # over limit
```

Ανίχνευση σημαίας γραμμής εντολών, χωρίς εισαγωγή αρθρώματος:

```
use feature 'keyword_any';
if (any { $_ eq '--verbose' } @ARGV) {
    $verbose = 1;
}
```

Η any σε ζεύγος με την `all` ως έλεγχος ορθότητας - κάθε εγγραφή έχει `id`, και τουλάχιστον μία είναι σημαδεμένη ως ενεργή:

```
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 5.44:

```
use List::Util qw(any);
print "found\n" if any { $_ eq 'needle' } @haystack;
```

Οριακές περιπτώσεις

- **Βραχυκυκλώνει στο πρώτο αληθές αποτέλεσμα.** Οι παρενέργειες στο BLOCK τρέχουν μόνο για τα στοιχεία που εξετάστηκαν πριν (και συμπεριλαμβανομένου) το πρώτο αληθές στοιχείο. Μη χρησιμοποιείτε την any για να οδηγήσετε βρόχο που πρέπει να αγγίξει κάθε στοιχείο - χρησιμοποιήστε for ή map γι" αυτό.
- **Κενή LIST επιστρέφει ψευδές.** Το any { anything } () είναι ψευδές χωρίς να εκτελεστεί ποτέ το μπλοκ.
- **Η ψευδωνυμοποίηση του $_ είναι πανομοιότυπη με τις map και grep.** Η εγγραφή στο $_ μεταβάλλει το στοιχείο της πηγαίας λίστας. Το ιδίωμα any { my $x = $_; ... } καθιστά ρητό το αντίγραφο.
- **Η return μέσα στο BLOCK επιστρέφει από την περικλείουσα υπορουτίνα**, όχι από την any. Για να τερματίσετε τη σάρωση νωρίς με συγκεκριμένη τιμή αλήθειας, αφήστε το μπλοκ να αξιολογηθεί σε εκείνη την τιμή - η any ήδη βραχυκυκλώνει.
- **Βασική δεσμευμένη λέξη έναντι List::Util::any.** Και οι δύο υλοποιούν την ίδια σημασιολογία. Η βασική δεσμευμένη λέξη αποφεύγει ένα πλαίσιο κλήσης σε επίπεδο Perl ανά επίκληση της ίδιας της any, κάτι που εμφανίζεται σε σφιχτό προφιλάρισμα· το ίδιο το μπλοκ τρέχει ανά στοιχείο και στις δύο περιπτώσεις. Η ανάμιξη και των δύο στο ίδιο αρχείο είναι νόμιμη αλλά συγχέει· διαλέξτε μία ανά αρχείο.
- **Προειδοποίηση experimental::keyword_any.** Επειδή η δεσμευμένη λέξη εξακολουθεί να σημειώνεται ως πειραματική, η ενεργοποίησή της εκπέμπει προειδοποίηση στην κατηγορία experimental::keyword_any κατά την πρώτη χρήση. Αποσιωπήστε την με no warnings 'experimental::keyword_any'; αφότου αποφασίσετε να εξαρτηθείτε από τη δεσμευμένη λέξη.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- all - το αντίστοιχο του καθολικού ποσοδείκτη· αληθές αν και μόνο αν το BLOCK δίνει αληθές για **κάθε** στοιχείο
- grep - συλλέγει κάθε στοιχείο που ταιριάζει· το scalar grep { ... } @list έχει την ίδια τιμή αλήθειας με το any { ... } @list αλλά εκτελεί το μπλοκ για κάθε στοιχείο
- List::Util::any - pre-5.44 subroutine form with identical semantics
- List::Util::first - επιστρέφει το πρώτο στοιχείο για το οποίο το μπλοκ είναι αληθές, αντί για λογική τιμή· χρησιμοποιήστε την όταν χρειάζεστε το ίδιο το στοιχείο
- map - ίδιο συμβόλαιο ψευδωνυμοποίησης $_, αλλά επισκέπτεται πάντα κάθε στοιχείο και συλλέγει αποτελέσματα

Αριθμητικές συναρτήσεις

atan2

Τόξο εφαπτομένης του Y/X στο εύρος -π έως π.

Η atan2 είναι το τόξο εφαπτομένης δύο ορισμάτων - η αντίστροφη της εφαπτομένης που διατηρεί αρκετή πληροφορία για τις εισόδους της ώστε να επιστρέψει το σωστό τεταρτημόριο. Σε αντίθεση με την atan ενός ορίσματος, που συμπτύσσει τα δύο πρόσημα των Y και X σε έναν λόγο και δεν μπορεί να ξεχωρίσει το άνω ημιεπίπεδο από το κάτω, η atan2 εξετάζει και τα δύο ορίσματα και επιστρέφει γωνία που καλύπτει ολόκληρο τον κύκλο: (-π, π] ακτίνια. Είναι ο κανονικός τρόπος στην Perl να μετατρέψετε καρτεσιανές συντεταγμένες (X, Y) στην πολική γωνία θ.

Σύνοψη

```
atan2 Y, X
```

Τι επιστρέφεται

Αριθμός κινητής υποδιαστολής, η γωνία σε ακτίνια μεταξύ του θετικού άξονα x και του σημείου (X, Y), μετρούμενη αντίστροφα από τους δείκτες του ρολογιού. Εύρος: -π < result ≤ π. Το πρόσημο ακολουθεί το Y: θετικό Y δίνει θετική γωνία (άνω ημιεπίπεδο), αρνητικό Y δίνει αρνητική γωνία (κάτω ημιεπίπεδο). Για να μετατρέψετε σε μοίρες πολλαπλασιάστε επί 180 / π:

```
my $deg = atan2($y, $x) * 180 / 3.14159265358979;
```

Παραδείγματα

Υπολογισμός του π στην ακρίβεια της μηχανής. Η γωνία από την αρχή έως το (0, 1) είναι ακριβώς π/2, οπότε το atan2(1, 0) * 2 δίνει π:

```
my $pi = atan2(1, 0) * 2;
printf "%.15f\n", $pi;          # 3.141592653589793
```

Διαχείριση τεταρτημορίων. Οι τέσσερις βασικές κατευθύνσεις πέφτουν σε ακριβή πολλαπλάσια του π/2, με το (−1, 0) στο σύνορο +π, όχι -π:

```
print atan2( 1,  0), "\n";      #  1.5707963267949   (+π/2, up)
print atan2( 0,  1), "\n";      #  0                 (right)
print atan2(-1,  0), "\n";      # -1.5707963267949   (-π/2, down)
print atan2( 0, -1), "\n";      #  3.14159265358979  (+π, left)
```

Πολική μετατροπή. Δοθέντος ενός σημείου (X, Y), ανακτήστε ακτίνα και γωνία:

```
my ($x, $y) = (3, 4);
my $r     = sqrt($x*$x + $y*$y);    # 5
my $theta = atan2($y, $x);          # 0.927295218001612 rad ≈ 53.13°
```

Εφαπτομένη μέσω της τυπικής ταυτότητας. Η Perl δεν έχει ενσωματωμένη tan, αλλά οι sin και cos τη δίνουν άμεσα - και η atan2 αντιστρέφει το αποτέλεσμα χωρίς ασάφεια:

```
sub tan { sin($_[0]) / cos($_[0]) }
my $angle = 0.7;
print atan2(tan($angle), 1), "\n";  # 0.7  (recovers the input)
```

Σάρωση πλήρους κύκλου. Επαναλαμβάνοντας το Y γύρω από τον μοναδιαίο κύκλο φαίνεται ότι το εύρος καλύπτει το `(-π, π]`:

```
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));
}
```

Οριακές περιπτώσεις

- **Και τα δύο ορίσματα μηδέν**: η `atan2(0, 0)` είναι μαθηματικά απροσδιόριστη. Η τιμή που επιστρέφει η Perl είναι ό,τι επιστρέφει η C `atan2(3)` της πλατφόρμας για αυτή την περίπτωση - σε glibc και musl αυτή είναι `0`, και το POSIX επιτρέπει οποιαδήποτε τιμή στο `[-π, π]`. Μη βασίζεστε σε συγκεκριμένο αποτέλεσμα· ελέγξτε για `($x == 0 && $y == 0)` πριν την κλήση αν το μηδέν είναι πραγματική πιθανότητα στις εισόδους σας.
- **Προσημασμένα μηδενικά**: σε πλατφόρμες IEEE-754 η `atan2(+0, -0)` επιστρέφει `+π` και η `atan2(-0, -0)` επιστρέφει `-π`. Αυτό έχει σημασία μόνο σε κώδικα που σκόπιμα παρακολουθεί τα πρόσημα του μηδενός· ο περισσότερος κώδικας Perl δεν τα διακρίνει ποτέ.
- **Άπειρα**: `atan2(Inf, Inf) == π/4`, `atan2(Inf, -Inf) == 3π/4`, `atan2(-Inf, Inf) == -π/4`, `atan2(-Inf, -Inf) == -3π/4`. Η κατεύθυνση «προς `(±Inf, ±Inf)`» παραμένει μια καλά ορισμένη γωνία που διχοτομεί το αντίστοιχο τεταρτημόριο.
- **Είσοδοι NaN**: αν είτε το Y είτε το X είναι `NaN`, το αποτέλεσμα είναι `NaN`. Διαδίδεται μέσα από κάθε περαιτέρω αριθμητική πράξη - προστατευτείτε με έναν έλεγχο `Scalar::Util::looks_like_number` σε εισόδους από μη έμπιστες πηγές.
- **Μη αριθμητικά ορίσματα**: οι συμβολοσειρές εξαναγκάζονται σε αριθμούς με τους συνήθεις κανόνες της Perl. Η `atan2("3", "4")` λειτουργεί· η `atan2("abc", 1)` είναι ισοδύναμη με `atan2(0, 1)` και εκπέμπει προειδοποίηση `Argument "abc" isn't numeric` υπό `use warnings`.
- **Η σειρά ορισμάτων είναι `Y, X`, όχι `X, Y`**: αυτό ταιριάζει με την `atan2(3)` της C, την `ATAN2` της FORTRAN και του τόξου εφαπτομένης δύο ορισμάτων κάθε άλλης γλώσσας. Η αντιστροφή των ορισμάτων δίνει τη συμπληρωματική γωνία - ένα διακριτικό σφάλμα που οι έλεγχοι σε ένα μόνο τεταρτημόριο θα παραβλέψουν.
- **Η atan2 δεν είναι η atan**: δεν υπάρχει μορφή ενός ορίσματος. Το `atan2($ratio)` είναι συντακτικό σφάλμα στο σημείο κλήσης. Για το τόξο εφαπτομένης ενός ορίσματος, χρησιμοποιήστε `Math::Trig::atan` ή `POSIX::atan`, ή γράψτε το ως `atan2($ratio, 1)`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `sin` - ημίτονο· σε ζεύγος με τη `cos` για ανακατασκευή των συντεταγμένων μιας γωνίας αφότου την έχει εξαγάγει η `atan2`
- `cos` - συνημίτονο· η συνιστώσα X κατά τη μετατροπή ακτίνας και γωνίας ξανά σε καρτεσιανές συντεταγμένες
- `sqrt` - τετραγωνική ρίζα· χρησιμοποιείται με την `atan2` στη μετατροπή Καρτεσιανών → πολικών `($r, $theta) = (sqrt($x**2+$y**2), atan2($y, $x))`
- - παρέχει `tan`, `atan`, `asin`, `acos` και την υπερβολική οικογένεια· εξάγει επίσης το `pi` ως σταθερά ώστε να μη χρειάζεται να γράψετε `atan2(1,0)*2`

Υποδοχές (sockets)

bind

Σύνδεση τοπικής διεύθυνσης σε μια υποδοχή.

Η `bind` είναι το περιτύλιγμα της Perl για την κλήση συστήματος POSIX `bind(2)`. Συσχετίζει το ήδη ανοιγμένο filehandle `SOCKET` με την τοπική διεύθυνση `NAME`, ώστε οι επόμενες κλήσεις `listen` / `accept` (για διακομιστή) ή `connect` / `send` (για πελάτη με σταθερή τοπική διεύθυνση) να χρησιμοποιούν αυτή τη διεύθυνση ως πηγή τους. Το `NAME` είναι μια **πακεταρισμένη** δομή διεύθυνσης στη μορφή που αναμένει η οικογένεια διευθύνσεων της υποδοχής - η εργασία του πακεταρίσματος είναι δική σας, τυπικά μέσω βοηθητικών συναρτήσεων του `Socket` όπως `` `sockaddr_in` `` ή `` `sockaddr_un` ``.

Σύνοψη

```
bind SOCKET, NAME
```

Τι επιστρέφεται

Αληθής τιμή σε επιτυχία, ψευδής σε αποτυχία (με την `$!` να τίθεται στο υποκείμενο errno: `EADDRINUSE`, `EACCES`, `EADDRNOTAVAIL`, κ.λπ.). Ελέγχετε πάντα την τιμή επιστροφής - σε αντίθεση με την `print`, μια αποτυχημένη `bind` είναι σφάλμα κατά τη φάση ρύθμισης που πρέπει να ματαιώσει το πρόγραμμα:

```
bind $sock, $addr
    or die "bind failed: $!";
```

Καθολική κατάσταση που επηρεάζει

Καμία άμεσα. Η `bind` θέτει την `$!` σε αποτυχία· αυτή είναι η μόνη καθολική του διερμηνέα που εμπλέκεται. Η ίδια η υποδοχή πρέπει να είναι ήδη ανοιχτή - τυπικά

παράγεται από την `socket` ή την `socketpair` - και η `bind` μεταβάλλει την κατάστασή της στην πλευρά του πυρήνα, όχι κάποια μεταβλητή σε επίπεδο Perl.

Παραδείγματα

Δέσμευση μιας υποδοχής διακομιστή TCP στη θύρα 9000 σε όλες τις τοπικές διεπαφές και έναρξη ακρόασης. Το `INADDR_ANY` επιλέγει «οποιαδήποτε τοπική διεύθυνση»· η πακεταρισμένη μορφή προέρχεται από το `` `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: $!";
```

Δέσμευση μιας υποδοχής πελάτη σε συγκεκριμένη τοπική διεύθυνση πριν τη σύνδεση - χρησιμοποιείται όταν ο απομακρυσμένος ομότιμος φιλτράρει με βάση τη διεύθυνση IP πηγής, ή όταν ένας υπολογιστής διαθέτει πολλές διεπαφές και χρειάζεστε την εξερχόμενη κίνηση σε συγκεκριμένη:

```
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: $!";
```

Δέσμευση ενός ακροατή τομέα UNIX σε μια διαδρομή του συστήματος αρχείων. Η διαδρομή δεν πρέπει να υπάρχει ήδη - η `bind` δημιουργεί τον κόμβο υποδοχής και τον αφήνει πίσω όταν η διεργασία τερματίσει, οπότε καθαρίστε τον μόνοι σας:

```
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: $!";
```

Δέσμευση μιας υποδοχής UDP σε σταθερή τοπική θύρα, ώστε οι απαντήσεις από τον ομότιμο να επιστρέφουν σε προβλέψιμη διεύθυνση:

```
use Socket;

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
socket(my $udp, PF_INET, SOCK_DGRAM, getprotobyname("udp"))
    or die "socket: $!";
bind($udp, sockaddr_in(5300, INADDR_ANY))
    or die "bind: $!";
```

Η εκ νέου δέσμευση μιας προηγουμένως απελευθερωμένης θύρας σε σύντομο χρονικό διάστημα απαιτεί το SO_REUSEADDR· αλλιώς η bind αποτυγχάνει με EADDRINUSE κατά τη διάρκεια του παραθύρου TIME_WAIT:

```
use Socket;

setsockopt($srv, SOL_SOCKET, SO_REUSEADDR, pack("l", 1))
    or die "setsockopt: $!";
bind($srv, sockaddr_in(9000, INADDR_ANY))
    or die "bind: $!";
```

Οριακές περιπτώσεις

- **Το SOCKET πρέπει να είναι ήδη ανοιχτό.** Η bind δεν δημιουργεί την υποδοχή - καλέστε πρώτα την socket. Ένα κλειστό ή ποτέ-ανοιγμένο filehandle αποτυγχάνει με EBADF.
- **Το NAME είναι μια πακεταρισμένη συμβολοσειρά bytes, όχι αναγνώσιμη από άνθρωπο διεύθυνση.** Περάστε την έξοδο των `sockaddr_in`, `sockaddr_in6` ή `sockaddr_un` από το Socket. Το πέρασμα μιας απλής συμβολοσειράς όπως "127.0.0.1:9000" παράγει EINVAL ή χειρότερα - ο πυρήνας διαβάζει τα bytes ως struct sockaddr.
- **Η θύρα 0 σημαίνει «αφήστε τον πυρήνα να επιλέξει».** Μετά από μια επιτυχημένη bind με θύρα 0, ανακτήστε την πραγματική θύρα με την getsockname και το `sockaddr_in` σε λειτουργία αποπακεταρίσματος.
- **Οι θύρες κάτω από 1024** απαιτούν αυξημένα προνόμια σε Unix - η bind αποτυγχάνει με EACCES για μια διεργασία χωρίς προνόμια.
- Το **EADDRINUSE** είναι η πιο κοινή αποτυχία. Πυροδοτείται όταν μια άλλη διεργασία κατέχει τη διεύθυνση, όταν μια προγενέστερη υποδοχή στην ίδια διεύθυνση βρίσκεται ακόμα σε TIME_WAIT, ή όταν καλείτε την bind δύο φορές στην ίδια υποδοχή. Το SO_REUSEADDR (βλέπε setsockopt) αντιμετωπίζει την περίπτωση TIME_WAIT· οι υπόλοιπες είναι γνήσιες συγκρούσεις.
- **Οι υποδοχές τομέα UNIX αφήνουν πίσω εγγραφή στο σύστημα αρχείων** που επιβιώνει της διεργασίας. Αφαιρέστε την πριν από μια νέα bind (unlink $path) και κατά τον καθαρό τερματισμό.
- **Η δέσμευση μετά από connect είναι σφάλμα.** Όταν μια υποδοχή έχει συνδεθεί, η τοπική της διεύθυνση είναι καθορισμένη. Καλέστε πρώτα την bind αν χρειάζεστε συγκεκριμένη διεύθυνση πηγής.
- **Οι υποδοχές IPv6** χρησιμοποιούν `sockaddr_in6` και PF_INET6· η ανάμειξη οικογενειών μεταξύ της υποδοχής και της πακεταρισμένης διεύθυνσης αποτυγχάνει με EAFNOSUPPORT.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `socket` - δημιουργία του filehandle υποδοχής στο οποίο συνδέεται η `bind`· πρέπει να καλείται πρώτη
- `connect` - το αντίστοιχο στην πλευρά του πελάτη· θέτει την απομακρυσμένη διεύθυνση αντί της τοπικής
- `listen` - επισήμανση μιας δεσμευμένης υποδοχής ως πρόθυμης να δεχτεί εισερχόμενες συνδέσεις
- `accept` - άντληση της επόμενης εκκρεμούς σύνδεσης από μια δεσμευμένη υποδοχή που ακούει
- `getsockname` - ανάκτηση της πραγματικής διεύθυνσης που επέλεξε η `bind`, ειδικά μετά από δέσμευση στη θύρα 0
- `setsockopt` - ορισμός του `SO_REUSEADDR` και άλλων *πριν* την `bind` για έλεγχο της επαναχρησιμοποίησης διεύθυνσης
- `Socket` - πακετάρει και αποπακετάρει το όρισμα `NAME`· πηγή των `` `sockaddr_in` ``, `` `sockaddr_un` ``, `INADDR_ANY`, `PF_INET` και σχετικών σταθερών

I/O

binmode

Ορίζει τη στοίβα στρωμάτων I/O σε ένα filehandle - τυπικά για να παραδίδει ωμά bytes ή για να επισυναφθεί μια κωδικοποίηση χαρακτήρων.

Η `binmode` αναδιαμορφώνει το `FILEHANDLE` έτσι ώστε οι επόμενες αναγνώσεις και εγγραφές να περνούν από μια συγκεκριμένη στοίβα στρωμάτων PerlIO. Στη μορφή με ένα όρισμα, θέτει το handle σε ωμή δυαδική κατάσταση (χωρίς μετατροπή CRLF, χωρίς αποκωδικοποίηση χαρακτήρων). Στη μορφή με δύο ορίσματα, ωθεί (ή, για μια χούφτα ψευδο-στρωμάτων, αντικαθιστά ή απομακρύνει) τα κατονομαζόμενα στρώματα I/O στη στοίβα του handle. Καλέστε την μετά την `open` και πριν από οποιοδήποτε I/O στο handle, εξαιρώντας το `:encoding`, που μπορεί να εφαρμοστεί στη μέση της ροής.

Σύνοψη

```
binmode FILEHANDLE
binmode FILEHANDLE, LAYER
```

Τι επιστρέφεται

1 σε επιτυχία, `undef` σε αποτυχία, με την `$!` τεθειμένη. Η αποτυχία στην πράξη είναι σπάνια - η συνηθισμένη αιτία είναι η διαβίβαση κακοσχηματισμένης συμβολοσειράς στρώματος ή άγνωστου ονόματος στρώματος. Η τιμή επιστροφής αξίζει να ελέγχεται όταν το όρισμα του στρώματος παρέχεται από τον χρήστη:

```
binmode $fh, ":encoding($enc)"
    or die "bad encoding '$enc': $!";
```

Αν το FILEHANDLE είναι μια έκφραση αντί για bareword ή απλό βαθμωτό, η τιμή του λαμβάνεται ως το όνομα του filehandle, σύμφωνα με τους κανόνες ανάλυσης filehandle των open και print.

Καθολική κατάσταση που επηρεάζει

Η ίδια η binmode δεν διαβάζει ειδικές μεταβλητές, αλλά τα στρώματα που εγκαθιστά αλλάζουν τον τρόπο με τον οποίο το μετέπειτα I/O στο handle αλληλεπιδρά με αυτές. Η στοίβα στρωμάτων καθορίζει πώς οι $/ (διαχωριστής εγγραφής εισόδου) και $\ (διαχωριστής εγγραφής εξόδου) αντιστοιχίζονται και εκπέμπονται έναντι της εξωτερικής ροής bytes - σε πλατφόρμες όπου η κατάσταση κειμένου μεταφράζει το \n σε ακολουθία πολλαπλών bytes, η μετατροπή συμβαίνει μέσα στο στρώμα, όχι στη μεταβλητή.

Στρώματα PerlIO

Ένα στρώμα είναι μια συμβολοσειρά που ξεκινά με : και κατονομάζει ένα φίλτρο στη στοίβα PerlIO - δείτε το PerlIO για τον πλήρη κατάλογο. Μπορούν να δοθούν πολλαπλά στρώματα σε μία κλήση με συνένωσή τους:

```
binmode $fh, ":raw:utf8";
```

Η σειρά είναι από κάτω προς τα πάνω: το :raw εφαρμόζεται πρώτο (πλησιέστερα στον περιγραφέα αρχείου του OS), στη συνέχεια το :utf8 τοποθετείται πάνω από αυτό. Οι αναγνώσεις διασχίζουν τη στοίβα από κάτω προς τα πάνω· οι εγγραφές τη διασχίζουν από πάνω προς τα κάτω.

Τα συνήθως χρησιμοποιούμενα στρώματα:

- :raw - γυμνώνει το handle σε byte-προς-byte I/O. Απενεργοποιεί τη μετατροπή CRLF, αφαιρεί τυχόν στρώματα κωδικοποίησης χαρακτήρων και σημαδεύει το handle ως bytes. Παρά τους περιστασιακούς ισχυρισμούς περί του αντιθέτου, το :raw **δεν** είναι απλά το αντίστροφο του :crlf· απενεργοποιεί επίσης οποιοδήποτε άλλο στρώμα που θα αλλοίωνε τη δυαδική φύση της ροής. Αυτό εγκαθιστά η μορφή με ένα όρισμα.
- :crlf - μεταφράζει ακολουθίες \r\n σε \n στην είσοδο και \n σε \r\n στην έξοδο. Αυτή είναι η προεπιλογή σε συστήματα της οικογένειας Windows / DOS για handles σε κατάσταση κειμένου· η ρητή εφαρμογή του εξαναγκάζει τη μετατροπή CRLF σε οποιαδήποτε πλατφόρμα.
- :utf8 - σημαδεύει τα δεδομένα του handle ως UTF-8. Χωρίς επικύρωση στην είσοδο: τα bytes γίνονται δεκτά και σημαδεύονται ως χαρακτήρες. Στην έξοδο, οι χαρακτήρες κωδικοποιούνται σε bytes UTF-8. Γρήγορο, αλλά εμπιστεύεται την πηγή.
- :encoding(NAME) - αποκωδικοποιεί bytes σε χαρακτήρες στην είσοδο και κωδικοποιεί χαρακτήρες σε bytes στην έξοδο, χρησιμοποιώντας την κατονομαζόμενη κωδικοποίηση (UTF-8, iso-8859-1, shiftjis, κ.λπ.). Η μη έγκυρη για την κωδικοποίηση είσοδος προκαλεί προειδοποίηση και

αντικατάσταση. Το `:encoding` ωθεί σιωπηρά το `:utf8` πάνω από τον εαυτό του, διότι η Perl εργάζεται εσωτερικά σε UTF-8. Δείτε το `PerlIO::encoding` για λεπτομέρειες και επιλογές ρύθμισης.

- `:bytes` - το αντίστροφο του `:utf8`· σημαδεύει τα δεδομένα του handle ως bytes, απενεργοποιώντας οποιαδήποτε σημαία χαρακτηριστικής σημασιολογίας έχει θέσει ένα υψηλότερο στρώμα.

Δύο ψευδο-στρώματα ελέγχουν το σχήμα της στοίβας αντί να φιλτράρουν:

- `:pop` - αφαιρεί το πιο πάνω στρώμα από το handle.
- `:push` - χρησιμοποιείται με το `:via(...)` για τη στοίβαξη ενός προσαρμοσμένου αρθρώματος στρώματος.

Το `:raw` στη μορφή με δύο ορίσματα λειτουργεί ως επαναφορά: αφαιρεί στρώματα μέχρι το handle να βρίσκεται στην πιο ελάχιστη κατάστασή του, αντί να ωθεί ένα νέο στρώμα από πάνω.

Παραδείγματα

Η μορφή με ένα όρισμα, για φορητή ανάγνωση δυαδικού αρχείου:

```
open my $img, "<", "photo.jpg" or die $!;
binmode $img;
my $bytes = do { local $/; <$img> };
```

Ανάγνωση αρχείου κειμένου UTF-8 με επικύρωση:

```
open my $fh, "<", "notes.txt" or die $!;
binmode $fh, ":encoding(UTF-8)";
while (my $line = <$fh>) {
    # $line contains decoded characters, not bytes
}
```

Εγγραφή τερματισμών γραμμών τύπου Windows σε οποιαδήποτε πλατφόρμα:

```
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
```

Αφαίρεση προηγουμένως εφαρμοσμένου στρώματος κωδικοποίησης και επιστροφή σε bytes:

```
binmode $fh, ":pop";               # remove topmost layer
binmode $fh, ":raw";               # or: flatten to bare bytes
```

Επικύρωση κωδικοποίησης παρεχόμενης από τον χρήστη πριν τη δέσμευση σε αυτή:

```
my $enc = $ENV{INPUT_ENCODING} // "UTF-8";
binmode $fh, ":encoding($enc)"
    or die "unsupported encoding '$enc': $!";
```

Οριακές περιπτώσεις

- **Καλέστε την μετά την open, πριν από οποιοδήποτε I/O.** Η `binmode` σε ένα handle από το οποίο έχει ήδη γίνει ανάγνωση ή εγγραφή παράγει ορισμένα αλλά εξαρτώμενα από την υλοποίηση αποτελέσματα· τα περισσότερα στρώματα εκκενώνουν τους εκκρεμείς ενταμιευτές κατά την εγκατάσταση, πράγμα που μπορεί να χάσει δεδομένα που πέρασαν τα όρια του στρώματος. Το `:encoding` αποτελεί την τεκμηριωμένη εξαίρεση - μπορεί να ωθηθεί στη μέση της ροής χωρίς εκκένωση.
- **Η μορφή με ένα όρισμα είναι `:raw`, όχι «δεν κάνει τίποτα».** Σε Unix το παρατηρήσιμο αποτέλεσμα είναι συχνά μηδαμινό, διότι τα handles βρίσκονται ήδη σε κατάσταση προσανατολισμένη σε bytes, αλλά σε Windows απενεργοποιεί τη μετατροπή CRLF. Γράφετε φορητό κώδικα σαν η `binmode $fh` να είναι πάντα σημασιακά ενεργή.
- **CRLF σε Windows / DOS.** Χωρίς `binmode`, το runtime της C σε αυτές τις πλατφόρμες μετατρέπει `\r\n` σε `\n` στην είσοδο και το αντίστροφο στην έξοδο. Για δυαδικές μορφές (εικόνες, συμπιεσμένα δεδομένα, σειριοποιημένες δομές) αυτό αλλοιώνει σιωπηλά τη ροή. Η `binmode` είναι εκεί υποχρεωτική.
- **Ctrl-Z ως τέλος αρχείου.** Σε συστήματα της οικογένειας Microsoft, τα handles σε κατάσταση κειμένου αντιμετωπίζουν το `\cZ` (byte 0x1A) ως τέλος αρχείου. Δυαδικά δεδομένα που τυχαίνει να περιέχουν αυτό το byte θα εμφανιστούν αποκομμένα εκτός αν χρησιμοποιηθεί η `binmode`.
- **Το `:utf8` βασίζεται μόνο στην εμπιστοσύνη.** Ένα handle εισόδου με `:utf8` αλλά χωρίς `:encoding(UTF-8)` θα σας παραδώσει ευχαρίστως μη έγκυρο UTF-8 σημαδεμένο ως χαρακτήρες. Χρησιμοποιήστε `:encoding(UTF-8)` όταν η πηγή δεν είναι έμπιστη.
- **Το `:encoding` παρασύρει το `:utf8`.** Μετά από `binmode $fh, ":encoding(shiftjis)"` το handle έχει στη στοίβα του τόσο το `:encoding(shiftjis)` όσο και το `:utf8` - η εσωτερική μορφή της perl είναι UTF-8 ανεξάρτητα από την κωδικοποίηση επί της γραμμής.
- **Εκφράσεις filehandle.** Το `binmode $handles[0], ":utf8"` λειτουργεί· κάθε έκφραση που αποδίδει τιμή filehandle γίνεται δεκτή ως πρώτο όρισμα και αποτιμάται για να κατονομάσει το handle.
- **Επηρεάζει κάθε πρωτογενή λειτουργία I/O στο handle.** Η στοίβα στρωμάτων διέπει εξίσου τις `read`, `sysread`, `syswrite`, `seek`, `tell`, `print` και `readline`. Συγκεκριμένα, η `sysread` και η `syswrite` σε ένα handle σημαδεμένο με `:utf8` συνεχίζουν να περνούν από το στρώμα και ενδέχεται να επιστρέψουν μερικούς χαρακτήρες.
- **Αλληλεπίδραση με την pragma open.** Το `use open ":encoding(UTF-8)"` εγκαθιστά ένα προεπιλεγμένο σύνολο στρωμάτων για τα handles που ανοίγουν μετέπειτα. Ρητή κλήση της `binmode` υπερισχύει της προεπιλογής αυτής για το συγκεκριμένο handle.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `open` - δημιουργεί εξαρχής το filehandle· δέχεται στρώματα απευθείας στη συμβολοσειρά κατάστασης (`<:encoding(UTF-8)`) ώστε να παρακαμφθεί μια επακόλουθη κλήση `binmode`
- `Encode` - κωδικοποιεί και αποκωδικοποιεί συμβολοσειρές στη μνήμη, ανεξάρτητα από οποιοδήποτε filehandle
- `PerlIO` - πλήρης κατάλογος στρωμάτων, συμπεριλαμβανομένων λιγότερο συνηθισμένων στρωμάτων όπως `:scalar`, `:via` και `:mmap`
- `read` / `sysread` - πρωτογενείς λειτουργίες εισόδου των οποίων η σημασιολογία byte / χαρακτήρα εξαρτάται από τη στοίβα στρωμάτων
- `print` / `syswrite` - πρωτογενείς λειτουργίες εξόδου των οποίων η κωδικοποίηση και η συμπεριφορά τερματισμού γραμμής εξαρτώνται από τη στοίβα στρωμάτων
- `$/` - διαχωριστής εγγραφής εισόδου, αντιστοιχιζόμενος έναντι της ροής bytes μετά τα στρώματα
- `$\` - διαχωριστής εγγραφής εξόδου, εγγραφόμενος μέσω της στοίβας στρωμάτων

Κλάσεις και αντικειμενοστρέφεια

bless

Σημαδεύει αυτό στο οποίο δείχνει μια αναφορά ως αντικείμενο σε ένα πακέτο.

Η `bless` συνδέει τον αναφερόμενο της `REF` με το πακέτο που κατονομάζει το `CLASSNAME`, ώστε οι κλήσεις μεθόδων μέσω της `REF` να αποστέλλονται στις υπορουτίνες αυτού του πακέτου (και στους προγόνους του στο `@ISA`). Η ίδια η αναφορά δεν αλλάζει· αυτό που αλλάζει είναι η ετικέτα κλάσης του αναφερόμενου. Η `bless` επιστρέφει την `REF`, γι" αυτό είναι σχεδόν πάντα η τελευταία εντολή σε έναν κατασκευαστή.

Σύνοψη

```
bless REF, CLASSNAME
bless REF
```

Τι επιστρέφεται

Την ίδια αναφορά που πέρασε ως πρώτο όρισμα, η οποία τώρα φέρει την ετικέτα κλάσης `CLASSNAME`. Η επιστροφή της αναφοράς σας επιτρέπει να καλέσετε την `bless` ως τελική κλήση από έναν κατασκευαστή:

```
sub new {
    my ($class, %args) = @_;
    return bless { %args }, $class;
}
```

Από εκείνο το σημείο και μετά, η `ref` επί της επιστρεφόμενης αναφοράς αποδίδει το όνομα κλάσης αντί για τον ακατέργαστο τύπο αναφοράς (HASH, ARRAY, CODE, ...).

Μορφές

Δύο μορφές κλήσης, η μία ισχυρά προτιμώμενη:

- **Δύο ορισμάτων - χρησιμοποιήστε πάντα αυτή.**

  ```
  bless $ref, $class;
  ```

  Ο αναφερόμενος γίνεται αντικείμενο στο πακέτο που κατονομάζει η `$class`. Η χρήση του πρώτου ορίσματος του κατασκευαστή (`$_[0]`, συμβατικά `$class`) διατηρεί τον κατασκευαστή κληρονομήσιμο: μια υποκλάση που καλεί `Parent::new('Child', ...)` λαμβάνει πίσω ένα αντικείμενο blessed στο `Child`, όχι στο `Parent`.

- **Δύο ορισμάτων με κενή συμβολοσειρά - bless στο `main`.**

  ```
  bless $ref, "";
  ```

  Ένα κενό CLASSNAME αντιμετωπίζεται ως το πακέτο `main`. Σπάνια χρήσιμο εκτός από σκόπιμα τεχνάσματα.

- **Ενός ορίσματος - αποθαρρύνεται.**

  ```
  bless $ref;
  ```

  Ο αναφερόμενος γίνεται bless στο *τρέχον* πακέτο (το πακέτο στο οποίο μεταγλωττίζεται η κλήση `bless`). Αυτό σπάει την κληρονομικότητα: μια υποκλάση που κληρονομεί έναν κατασκευαστή `bless` ενός ορίσματος καταλήγει εξακολουθητικά με αντικείμενα στο πακέτο του γονέα, όχι του καλούντος. Χρησιμοποιήστε τη μορφή δύο ορισμάτων εκτός εάν έχετε συγκεκριμένο λόγο να μην το κάνετε, και τεκμηριώστε τον λόγο.

Παραδείγματα

Μια ελάχιστη κλάση βασισμένη σε κατακερματισμό:

```
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);
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
print ref $p, "\n";       # Point
print $p->x, ",", $p->y;  # 3,4
```

Κληρονομήσιμος κατασκευαστής - η ίδια new λειτουργεί για υποκλάση χωρίς τροποποίηση:

```
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
```

Τα αντικείμενα βασισμένα σε πίνακα λειτουργούν με τον ίδιο τρόπο - οποιοσδήποτε τύπος αναφοράς μπορεί να γίνει bless:

```
package Pair;
sub new {
    my ($class, $a, $b) = @_;
    return bless [$a, $b], $class;
}
sub first  { $_[0]->[0] }
sub second { $_[0]->[1] }

my $p = Pair->new("left", "right");
print $p->first, "/", $p->second;   # left/right
```

Η εκ νέου bless ενός υπάρχοντος αντικειμένου αλλάζει την ετικέτα κλάσης του επιτόπου:

```
my $obj = bless {}, 'OldClass';
bless $obj, 'NewClass';
print ref $obj;            # NewClass
```

Η `bless` επιστρέφει την αναφορά, γι” αυτό λειτουργεί η αλυσίδωση σε κλήσεις μεθόδων στην ίδια γραμμή:

```
Logger->new(file => $path)->info("started");
#                          ^ called on the blessed ref
```

Οριακές περιπτώσεις

- **Αυτό που λαμβάνει την ετικέτα είναι η αναφορά, όχι η μεταβλητή.** Τα αντίγραφα της αναφοράς μοιράζονται την ετικέτα κλάσης επειδή όλα δείχνουν στον ίδιο αναφερόμενο:

```
my $a = bless {}, 'Foo';
my $b = $a;
print ref $b;            # Foo
```

- **Η εκ νέου bless επιτρέπεται και είναι άμεση.** Μια αναφορά μπορεί να γίνει bless όσες φορές θέλετε· κάθε κλήση γράφει πάνω από την προηγούμενη ετικέτα κλάσης. Δεν υπάρχει ενσωματωμένη «unbless» - κάντε rebless σε ένα ουδέτερο πακέτο αν χρειάζεται να αφαιρέσετε την αντικειμενική φύση από έναν αναφερόμενο στον οποίο εξακολουθείτε να διατηρείτε αναφορές.
- **`CLASSNAME` ίσο με `""` σημαίνει `main`.** Ρητό και σπάνια επιθυμητό· συνήθως ένδειξη ότι ένα υπολογισμένο όνομα κλάσης βγήκε κενό κατά λάθος.
- **Αποφύγετε το `bless $ref, "0"`.** Το όνομα κλάσης `"0"` είναι ψευδής τιμή, οπότε κώδικας που ελέγχει `if (ref $obj) { ... }` θα διαβάσει εσφαλμένα το αντικείμενο ως «μη αναφορά». Επιλέξτε οποιοδήποτε όνομα κλάσης που δεν είναι ψευδές.
- **Σύμβαση πεζών-κεφαλαίων για ονόματα κλάσεων.** Χρησιμοποιήστε ανάμεικτη γραφή (`My::Package`). Τα ονόματα με ΟΛΟΚΕΦΑΛΑΙΑ είναι δεσμευμένα για ενσωματωμένους τύπους της Perl (`SCALAR`, `ARRAY`, `HASH`, `CODE`, ...) και τα ονόματα μόνο με πεζά είναι δεσμευμένα για pragmas (`strict`, `warnings`, `utf8`). Το bless σε οποιονδήποτε από αυτούς τους χώρους ονομάτων προκαλεί σύγχυση με τον μηχανισμό σε επίπεδο γλώσσας.
- **Πρώτο όρισμα που δεν είναι αναφορά αποτελεί μοιραίο σφάλμα.** Το `bless 42, 'Foo'` πεθαίνει με `Can't bless non-reference value`. Το πρώτο όρισμα πρέπει να είναι αναφορά.
- **Η `bless` ενός ορίσματος επιλέγει το πακέτο μεταγλώττισης.** Χρησιμοποιεί το πακέτο μέσα στο οποίο βρίσκεται *λεξιλογικά* η κλήση `bless` - όχι το πακέτο του καλούντος, όχι το πακέτο της `$class`. Αυτός ακριβώς είναι ο λόγος που σπάει την κληρονομικότητα και που υπάρχει η μορφή δύο ορισμάτων.
- **Οι ασθενείς αναφορές και το bless αλληλεπιδρούν όπως αναμένεται.** Η `Scalar::Util::weaken` σε μια blessed αναφορά δεν την αποκαθιστά· η εξασθενημένη θυρίδα εξακολουθεί να βλέπει την ετικέτα κλάσης, και η `ref` λειτουργεί μέχρι να καταστραφεί ο αναφερόμενος.
- **Η `DESTROY` εκπυρσοκροτεί στον blessed αναφερόμενο.** Όταν η τελευταία αναφορά σε έναν blessed αναφερόμενο εξαφανιστεί, η Perl καλεί την `DESTROY` στην κλάση της αναφοράς (ακολουθώντας το `@ISA`). Η αλλαγή της κλάσης με μια καθυστερημένη `bless` αλλάζει ποια `DESTROY` εκτελείται.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `ref` - διαβάστε πίσω την ετικέτα κλάσης από μια blessed αναφορά, ή τον τύπο αναφοράς (HASH, ARRAY, ...) για μια μη-blessed
- `package` - δηλώστε το πακέτο στο οποίο μεταγλωττίζεται μια κλήση `bless` ή ένας ορισμός υπορουτίνας· ο στόχος της `bless` ενός ορίσματος
- `class` - η πειραματική ενσωματωμένη σύνταξη κλάσης που αντικαθιστά το χειρωνακτικά γραμμένο μοτίβο κατασκευαστή `package` + `bless` με δηλωτική σύνταξη
- `tie` - μια διαφορετική μορφή μαγείας ανά μεταβλητή· άσχετη με την `bless` παρά την επιφανειακή διατύπωση «επισύναψη συμπεριφοράς σε μια τιμή»
- `@ISA` - η αλυσίδα κληρονομικότητας που συμβουλεύεται κατά την αποστολή μιας κλήσης μεθόδου σε blessed αναφορά
- `DESTROY` - η μέθοδος καταστροφέα που η Perl καλεί στον blessed αναφερόμενο όταν εξαφανιστεί η τελευταία του αναφορά· δηλώνεται ως υπορουτίνα μέσα στο πακέτο της κλάσης

Ροή ελέγχου

break

Έξοδος από ένα μπλοκ `given`.

Η `break` τερματίζει αμέσως το περικλείον μπλοκ `given` και μεταφέρει τον έλεγχο στην επόμενη δήλωση - το ανάλογο της `last` για βρόχους, εδώ για `given`. Δεν δέχεται ορίσματα, δεν επιστρέφει τιμή, και εφαρμόζεται μόνο στο εσώτατο `given`· δεν μπορεί να στοχεύσει εξωτερικό μπλοκ με ετικέτα.

Σύνοψη

```
break
```

Διαθεσιμότητα - διαβάστε πρώτα αυτό

`break` is a keyword of the `switch` feature, together with `given`, `when`, `default`, and (inside a `when`) `continue`. In Perl 5.44 the switch feature is **disabled by default** and is not part of any current feature bundle: `use v5.36` through `use v5.44` 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';
```

Χωρίς αυτή την ενεργοποίηση, η bareword `break` αναλύεται ως απλή κλήση υπορουτίνας και αποτυγχάνει κατά τη μεταγλώττιση με `Bareword "break" not allowed while "strict subs" in use`, ή κατά τον χρόνο εκτέλεσης με `Undefined`

subroutine &main::break called. Η πλήρως κατονομασμένη μορφή CORE::break λειτουργεί από οποιαδήποτε εμβέλεια χωρίς ενεργοποίηση του χαρακτηριστικού και είναι ο τρόπος να την καλέσετε σε κώδικα που κατά τα άλλα διατηρεί το switch ανενεργό.

Τα given/when βασίζονται στο smartmatch (~~), το οποίο είναι το ίδιο πειραματικό και προειδοποιεί σε κάθε χρήση εκτός αν κατασταλούν οι προειδοποιήσεις experimental::smartmatch. Νέος κώδικας **δεν** πρέπει να καταφεύγει στα given/when/break· δείτε *Replacements* παρακάτω.

Συμπεριφορά

Μέσα σε μπλοκ given, η break ξετυλίγει μέχρι τη δήλωση αμέσως μετά το μπλοκ. Είναι ισοδύναμη με αυτό που πέφτει σιωπηρά από το κάτω μέρος ενός κλάδου when - οι ρήτρες when εκτελούν break στο τέλος του σώματός τους εκτός αν το σώμα τελειώνει σε continue, οπότε ρητή break χρησιμοποιείται κυρίως για πρόωρη έξοδο από μακρύ σώμα when:

```
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" }
}
```

Εκτός μπλοκ given, η break εγείρει Can't "break" outside a given block κατά τον χρόνο εκτέλεσης.

Αντικαταστάσεις

Για νέο κώδικα, προτιμήστε μία από αυτές τις μορφές. Είναι απλή Perl, δεν χρειάζονται ενεργοποίηση χαρακτηριστικού, και δεν έχουν συννεφιά απαξίωσης από πάνω τους:

- **Αλυσίδα if / elsif / else** - άμεση αντικατάσταση για τις περισσότερες χρήσεις given/when:

  ```
  if    ($cmd eq 'quit') { cleanup(); log_exit() }
  elsif ($cmd eq 'help') { show_help() }
  else                   { warn "unknown: $cmd" }
  ```

- **Πίνακας αποστολής** - ένας hash από coderefs με κλειδί την είσοδο, με προεπιλεγμένο κλάδο. Κλιμακώνεται καλύτερα από μια μακριά σκάλα elsif και κάνει το σύνολο των χειριζόμενων περιπτώσεων δεδομένα αντί για ροή ελέγχου:

  ```
  my %dispatch = (
      quit => sub { cleanup(); log_exit() },
  ```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    help => \&show_help,
);
($dispatch{$cmd} // sub { warn "unknown: $cmd" })->();
```

- **Βρόχος με ετικέτα με `last`** - όταν θέλετε συγκεκριμένα τη μορφή «πρόωρη έξοδος από αυτό το μπλοκ» που παρείχε η `break`, τυλίξτε το σώμα σε σκέτο μπλοκ και κάντε `last` από αυτό:

```
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";
}
```

Παραδείγματα

Ρητή ενεργοποίηση χαρακτηριστικού, πρόωρη έξοδος από σώμα when:

```
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
```

Χρήση της `CORE::break` χωρίς καθολική ενεργοποίηση του χαρακτηριστικού:

```
sub classify {
    my ($n) = @_;
    given ($n) {
        when ($_ < 0) { return "neg"; CORE::break }
        when (0)      { return "zero" }
        default       { return "pos" }
    }
}
```

Οριακές περιπτώσεις

- **Καμία μορφή ορίσματος.** Η `break` δεν δέχεται τίποτα. Το `break LABEL` είναι συντακτικό σφάλμα - σε αντίθεση με τα `last`/`next`, η `break` δεν έχει τρόπο να στοχεύσει εξωτερικό μπλοκ.
- **Εκτός `given`.** Η κλήση της `break` (ή της `CORE::break`) όταν δεν υπάρχει μπλοκ `given` στη στοίβα κλήσεων εγείρει `Can't "break" outside a given block`.
- **Η `return` υπερισχύει.** Μέσα σε υπορουτίνα, η `return` ξετυλίγει ολόκληρη την sub

ανεξάρτητα από οποιοδήποτε περικλείον given. Η break ξετυλίγει μόνο μέχρι το τέλος του given· ο έλεγχος συνεχίζει στην περιβάλλουσα sub.

- **Σιωπηρή break στο τέλος της when.** Το σώμα μιας ρήτρας when τελειώνει με σιωπηρή break εκτός αν το σώμα είναι continue, οπότε ρητή break στο τέλος σώματος when είναι περιττή. Η continue καταστέλλει τη σιωπηρή break και αφήνει τον έλεγχο να περάσει στην επόμενη when.
- **Δεν είναι δεσμευμένη λέξη βρόχου.** Η break δεν αλληλεπιδρά με τα for, while, foreach, do, ή σκέτα μπλοκ. Για αυτά, χρησιμοποιήστε τη last.
- **Πειραματικές προειδοποιήσεις.** Η εκτέλεση κώδικα given/when χωρίς no warnings 'experimental::smartmatch' εκτυπώνει την προειδοποίηση μία φορά ανά σημείο smartmatch, που σε ένα πολυάσχολο μπλοκ given μπορεί να είναι κάθε επανάληψη.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- last - το αντίστοιχο για βρόχο· εξέρχεται από το εσώτατο περικλείον for / while / foreach / σκέτο μπλοκ, και σε αντίθεση με την break μπορεί να στοχεύσει εξωτερικό μπλοκ με ετικέτα
- next - ξεκινά την επόμενη επανάληψη βρόχου αντί να εξέρχεται από αυτόν· χρήσιμη όταν η break είναι λάθος μορφή επειδή θέλετε συνέχιση, όχι τερματισμό
- return - ξετυλίγει εντελώς από την περικλείουσα υπορουτίνα, όχι μόνο από το μπλοκ given· χρησιμοποιήστε την όταν η given είναι διανομέας μέσα σε sub και η απάντηση είναι έτοιμη

Ροή ελέγχου · Εμβέλεια

caller

Επιστρέφει πληροφορίες σχετικά με την υπορουτίνα, την eval ή την require που κάλεσε τον κώδικα που εκτελείται αυτή τη στιγμή.

Η caller διατρέχει τη στοίβα κλήσεων. Χωρίς όρισμα επιστρέφει μια σύντομη εγγραφή τριών στοιχείων που προσδιορίζει *ποιος μας κάλεσε*. Με ένα αριθμητικό όρισμα EXPR διατρέχει EXPR πλαίσια πιο πίσω και επιστρέφει μια εκτενέστερη εγγραφή έντεκα στοιχείων - τη μορφή που χρησιμοποιούν ο debugger, τα αρθρώματα ιχνηλάτησης στοίβας, καθώς και ο μηχανισμός των warn/die για να εμπλουτίζουν τα διαγνωστικά μηνύματα με πληροφορίες αρχείου και γραμμής. Τα πλαίσια XS είναι αόρατα στην caller· στη θέση τους εμφανίζεται το επόμενο πλαίσιο καθαρής Perl.

Σύνοψη

```
caller
caller EXPR
```

Τι επιστρέφεται

Σε βαθμωτό περιβάλλον, το όνομα του πακέτου του καλούντος, ή `undef` αν δεν υπάρχει καλών (δηλαδή βρισκόμαστε στο ανώτατο επίπεδο του αρχείου, εκτός από `sub`, `eval` ή `require`):

```
my $pkg = caller;
```

Σε περιβάλλον λίστας χωρίς όρισμα, τρία στοιχεία - το πακέτο, το αρχείο και ο αριθμός γραμμής του άμεσου καλούντος:

```
# 0         1           2
my ($package, $filename, $line) = caller;
```

Σε περιβάλλον λίστας με αριθμητικό όρισμα, έντεκα στοιχεία που περιγράφουν το πλαίσιο EXPR επίπεδα ψηλότερα. Το `0` είναι ο άμεσος καλών, το 1 είναι ο δικός του καλών, και ούτω καθεξής:

```
# 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);
```

Τα έντεκα πεδία, με τη σειρά:

- `$package`, `$filename`, `$line` - η τοποθεσία του *σημείου κλήσης* μέσα στον καλούντα εκείνου του πλαισίου. Οι ίδιες τρεις τιμές που επιστρέφει η σύντομη μορφή.
- `$subroutine` - το πλήρως προσδιορισμένο όνομα της υπορουτίνας στην οποία ανήκει το πλαίσιο, π.χ. `"Foo::bar"`. Για ένα πλαίσιο `eval` είναι η κυριολεκτική συμβολοσειρά `"(eval)"`. Αν η υπορουτίνα έχει αφαιρεθεί από τον πίνακα συμβόλων μετά την κλήση, είναι `"(unknown)"`.
- `$hasargs` - αληθές αν ένας φρέσκος `@_` εγκαταστάθηκε για το πλαίσιο. Ψευδές για πλαίσια `eval` και `require`.
- `$wantarray` - το περιβάλλον στο οποίο κλήθηκε το πλαίσιο, όπως θα το είχε αναφέρει η `wantarray`: αληθές για λίστα, ορισμένο-αλλά-ψευδές (`""`) για βαθμωτό, `undef` για κενό (void).
- `$evaltext` - για ένα πλαίσιο `eval EXPR`, το κείμενο πηγαίου κώδικα που μεταγλωττίστηκε. Undef για `eval BLOCK` και για μη-`eval` πλαίσια.
- `$is_require` - αληθές αν το πλαίσιο δημιουργήθηκε από `require` ή `use` (κάθε `use` δημιουργεί ένα πλαίσιο `require` εμφωλευμένο μέσα σε ένα πλαίσιο `eval EXPR`).
- `$hints` - η τιμή του `$^H` τη στιγμή που μεταγλωττίστηκε ο κώδικας του πλαισίου. Εσωτερικό· η διάταξη δεν είναι σταθερή μεταξύ εκδόσεων της Perl.
- `$bitmask` - η τιμή του `${^WARNING_BITS}` στο ίδιο σημείο. Επίσης εσωτερικό.
- `$hinthash` - μια αναφορά στο hash `%^H` που καταγράφηκε κατά τη μεταγλώττιση, ή `undef` αν ήταν κενό. **Μην** την τροποποιείτε - η αναφορά δείχνει στο ζωντανό optree.

Καθολική κατάσταση που επηρεάζει

- `@DB::args` - ορίζεται ως παρενέργεια όταν η `caller` καλείται με όρισμα *μέσα από το πακέτο* DB. Η λίστα ορισμάτων του καλούντος ψευδωνυμοποιείται σε αυτόν τον πίνακα για χρήση από τον debugger. Δείτε *Οριακές περιπτώσεις* παρακάτω - αυτή είναι μια προσπάθεια βέλτιστης πρόθεσης και έχει αιχμηρές γωνίες.
- `$^H`, `${^WARNING_BITS}`, `%^H` - διαβάζονται από τη μεταγλωττισμένη κατάσταση του πλαισίου-στόχου για να συμπληρώσουν τα `$hints`, `$bitmask`, `$hinthash`.

Παραδείγματα

Βρείτε το πακέτο του άμεσου καλούντος και αναφέρετε στο σωστό αρχείο καταγραφής:

```
sub log_for_caller {
    my $pkg = caller;
    warn "[$pkg] @_\n";
}
```

Παραγάγετε μια ετικέτα τοποθεσίας σε στιλ `warn`. Η `caller` επιστρέφει το όνομα αρχείου και τη γραμμή του σημείου κλήσης ένα πλαίσιο ψηλότερα, που είναι σχεδόν πάντα η πληροφορία που πραγματικά θέλετε σε ένα διαγνωστικό:

```
sub note {
    my (undef, $file, $line) = caller;
    print STDERR "note at $file line $line: @_\n";
}
```

Διατρέξτε ολόκληρη τη στοίβα. Επαναλάβετε την `caller($i)` μέχρι να επιστρέψει κενή λίστα:

```
sub stacktrace {
    my @frames;
    for (my $i = 0; my @f = caller($i); $i++) {
        push @frames, \@f;
    }
    return @frames;
}
```

Ανιχνεύστε αν η τρέχουσα υπορουτίνα κλήθηκε σε περιβάλλον λίστας, βαθμωτό ή κενό. Αυτό είναι η `wantarray` στο τρέχον επίπεδο, αλλά για το πλαίσιο κάποιου άλλου το προσεγγίζετε μέσω της `caller`:

```
sub describe_parent_context {
    my $wa = (caller(1))[5];
    return !defined $wa ? "void"
         :  $wa         ? "list"
         :               "scalar";
}
```

Διακρίνετε ένα πλαίσιο `require`/`use` από ένα κανονικό `eval`:

```
sub called_from_require {
    my @f = caller(1);
    return @f && $f[7];            # field 7 is $is_require
}
```

Αποτυπώστε ένα μεμονωμένο πλαίσιο όπως κάνει ο προεπιλεγμένος μορφοποιητής της die / του Carp - "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";
}
```

Οριακές περιπτώσεις

- **Βαθμωτό περιβάλλον, χωρίς καλούντα**: στο ανώτατο επίπεδο ενός σεναρίου, η `scalar caller` επιστρέφει undef, ενώ η `caller` σε περιβάλλον λίστας επιστρέφει την κενή λίστα. Ένα κοινό ιδίωμα για το «εκτελούμαι με require ως άρθρωμα ή ως σενάριο;» είναι το `caller() ? ... : ...`.
- **Τα πλαίσια XS παραλείπονται**. Η `caller` αναφέρει μόνο πλαίσια καθαρής Perl. Αν μια υπορουτίνα καθαρής Perl κλήθηκε από μια υπορουτίνα XS η οποία με τη σειρά της κλήθηκε από άλλη υπορουτίνα καθαρής Perl, η `caller(0)` από την εσωτερική υπορουτίνα αναφέρει την εξωτερική υπορουτίνα καθαρής Perl - το στρώμα XS είναι αόρατο. Τα βάθη στοίβας που μετρώνται με την `caller` δεν περιλαμβάνουν επομένως διανομείς XS.
- **Τα πλαίσια eval δεν έχουν ορίσματα**. Το `$hasargs` είναι ψευδές για πλαίσια eval και require, και το `$subroutine` είναι η κυριολεκτική τιμή `"(eval)"` - όχι το πακέτο του περιβάλλοντος κώδικα.
- **`eval BLOCK` έναντι `eval EXPR`**. Και τα δύο δίνουν `$subroutine eq "(eval)"`. Μόνο η `eval EXPR` συμπληρώνει το `$evaltext`· η `eval BLOCK` το αφήνει undef.
- **Η use δημιουργεί δύο πλαίσια**. Μια εντολή `use Foo` μεταγλωττίζεται ως `BEGIN { require Foo; Foo->import; }`, ενώ η ίδια η require υλοποιείται ως `eval EXPR`. Η διατρέξιμη της στοίβας μέσα από μια γραμμή use βλέπει επομένως ένα πλαίσιο eval σημασμένο ως require μέσα σε ένα κανονικό πλαίσιο eval. Ελέγξτε το `$is_require` για να τα διακρίνετε.
- **Υπορουτίνες `(unknown)`**. Αν η υπορουτίνα που κατονομάζεται σε ένα πλαίσιο έχει διαγραφεί από τον πίνακα συμβόλων (επεμβάσεις στον πίνακα συμβόλων, `delete $Foo::{bar}`) μεταξύ της κλήσης και της `caller`, το `$subroutine` διαβάζεται ως `"(unknown)"`.
- **Οι βρόχοι και τα μπλοκ `try` δεν αποτελούν πλαίσια**. Τα `while`, `for`, `foreach` και `try` δεν δημιουργούν πλαίσια κλήσης. Η `caller` βλέπει μόνο κλήσεις υπορουτινών και eval.
- **Πλαίσια που αφαιρεί ο βελτιστοποιητής**. Ο βελτιστοποιητής μπορεί να αφαιρέσει πλαίσια ουραίας κλήσης και παρόμοια προτού η `caller` μπορέσει να τα παρατηρήσει. Για `caller(N)` με `N > 1`, το πλαίσιο που λαμβάνετε δεν είναι

εγγυημένα το πλαίσιο που περιμένατε. Μην κωδικοποιείτε το βάθος στοίβας ως φέρουσα σύμβαση.

- **Το `@DB::args` λειτουργεί με βέλτιστη πρόθεση**. Συμπληρώνεται μόνο όταν η `caller` καλείται με όρισμα μέσα από το πακέτο DB. Οι τιμές είναι ψευδώνυμα προς τον `@_` του πλαισίου-στόχου, όχι αντίγραφα - οπότε:
  - Οι μεταβολές που έχει ήδη κάνει το πλαίσιο στον `@_` είναι ορατές· οι αρχικές τιμές κατά την κλήση δεν διατηρούνται.
  - Το `shift @_` μέσα στο πλαίσιο συνήθως αναιρείται στο `@DB::args`, όμως τα `pop @_`, οι splices ή η λήψη αναφοράς στον `@_` σπάνε αυτή τη συμπεριφορά.
  - Τα στοιχεία μπορεί να έχουν απελευθερωθεί και να έχουν επαναχρησιμοποιηθεί για άλλες τιμές μέχρι να τα διαβάσετε.
  - Παλιά κατάσταση από προηγούμενη κλήση της `caller` ενδέχεται να παραμένει στο `@DB::args` αν η τρέχουσα κλήση δεν προκάλεσε ανανέωση. Χρησιμοποιήστε το για backtraces και εμφάνιση στον debugger· μην βασίζεστε σε αυτό για τη λογική του προγράμματος.
- **Το `$hinthash` είναι ζωντανό**. Η αναφορά hash στο πεδίο 10 δείχνει στο πραγματικό στιγμιότυπο `%^H` που έχει ψηθεί στο optree. Η τροποποίηση των περιεχομένων του μεταβάλλει την κατάσταση του μεταγλωττιστή. Διαβάστε το· μην το γράφετε.
- **Το όνομα αρχείου και η γραμμή μπορούν να επανεγγραφούν**. Οι οδηγίες `#line` και οτιδήποτε άλλο αναγνωρίζει ο μηχανισμός «Plain Old Comments (Not!)» (δείτε `perlsyn`) αλλάζουν τις τιμές που αναφέρει η `caller`, ακριβώς όπως αλλάζουν τα `__FILE__` και `__LINE__`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `die` - εγείρει εξαίρεση· το προεπιλεγμένο μήνυμά της χρησιμοποιεί τις ίδιες πληροφορίες αρχείου-και-γραμμής που εκθέτει η `caller`
- `eval` - δημιουργεί πλαίσια που η `caller` σημαίνει με `$subroutine eq "(eval)"` και μπορεί να συμπληρώσει τα `$evaltext` / `$is_require`
- `wantarray` - αναφέρει το περιβάλλον κλήσης του τρέχοντος πλαισίου· η ίδια τιμή εμφανίζεται ως πεδίο 5 της `caller(N)` για το πλαίσιο `N` επίπεδα ψηλότερα
- `$^H` - bits υπόδειξης κατά τη μεταγλώττιση· εμφανίζονται ως πεδίο 8
- `%^H` - hash υπόδειξης κατά τη μεταγλώττιση· εμφανίζεται ως πεδίο 10
- `${^WARNING_BITS}` - bitmask προειδοποιήσεων· εμφανίζεται ως πεδίο 9

Ροή ελέγχου

catch

Χειρίζεται μια εξαίρεση που εκτοξεύθηκε από προηγούμενο μπλοκ `try`.

Η `catch` είναι το δεύτερο μισό της κατασκευής `try`/`catch`. Δεν είναι αυτόνομη εντολή: ακολουθεί ένα μπλοκ `try`, δηλώνει μια λεξιλογική VAR για να λάβει την τιμή της εξαίρεσης, και εκτελεί το BLOCK της μόνο όταν το μπλοκ `try` πέθανε. Αν το μπλοκ `try` ολοκληρώθηκε κανονικά, το μπλοκ `catch` παραλείπεται. Ο έλεγχος συνεχίζει τότε στην εντολή που ακολουθεί ολόκληρη την κατασκευή (ή στο μπλοκ `finally`, αν υπάρχει).

Αυτή η σύνταξη πρέπει να ενεργοποιηθεί με `use feature 'try'`. Οι `try` και `catch` δεν είναι πειραματικές από την Perl 5.40 και μετά.

Σύνοψη

```
try BLOCK catch (VAR) BLOCK
try BLOCK catch (VAR) BLOCK finally BLOCK
```

Τι επιστρέφεται

Η `catch` δεν είναι κλήση συνάρτησης και δεν έχει επιστροφή με την έννοια της έκφρασης. Όπως άλλες κατασκευές ροής ελέγχου, η τιμή ολόκληρης της `try`/`catch` είναι η τιμή του μπλοκ που εκτελέστηκε τελευταίο - χρήσιμο μέσα σε μπλοκ `do` ή ως ουραία έκφραση μιας υπορουτίνας:

```
my $value = do {
    try {
        fetch_thing(@args);
    }
    catch ($e) {
        warn "fetch failed: $e";
        $DEFAULT;
    }
};
```

Αν η `try` εκτελέστηκε μέχρι το τέλος, η κατασκευή αποδίδει την τελευταία τιμή του μπλοκ `try` και η VAR δεν δένεται ποτέ. Αν η `try` πέθανε, η κατασκευή αποδίδει την τελευταία τιμή του μπλοκ `catch`.

Η δήλωση VAR

Οι παρενθέσεις μετά την `catch` είναι υποχρεωτικές και πρέπει να περιέχουν ακριβώς μία δήλωση βαθμωτής μεταβλητής. Συμπεριφέρεται ως δήλωση `my`

- 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
```

Η VAR έχει εμβέλεια μόνο μέσα στο μπλοκ `catch`. Δεν είναι ορατή στο προηγούμενο μπλοκ `try`, σε οποιοδήποτε επόμενο μπλοκ `finally`, ούτε μετά την κατασκευή. Η τιμή της είναι ό,τι εκτόξευσε το μπλοκ `try` - μια συμβολοσειρά, μια λίστα που μετατράπηκε

σε συμβολοσειρά, ή μια αναφορά, τυπικά ένα blessed αντικείμενο εξαίρεσης. Δείτε την `die` για το πώς παράγεται η τιμή της εξαίρεσης.

Σε αντίθεση με την `eval`, η `catch` **δεν** διαβάζει ούτε καθαρίζει το `$@`· η εξαίρεση ταξιδεύει απευθείας μέσω της VAR. Το `$@` παραμένει ανέγγιχτο από την κατασκευή.

Καθολική κατάσταση που επηρεάζει

Η ίδια η `catch` δεν επηρεάζει καμία ειδική μεταβλητή. Η κατασκευή `try`/`catch` στο σύνολό της αφήνει το `$@` αμετάβλητο - σκόπιμη απόκλιση από την `eval` που έχει σκοπό να αποτρέψει τους καλούντες από το να συγχέουν τα δύο ιδιώματα. Αν χρειάζεστε επιθεώρηση τύπου `$@`, χρησιμοποιήστε την `eval` αντί γι" αυτό.

Παραδείγματα

Σύλληψη μιας απλής εξαίρεσης συμβολοσειράς και αναφορά της:

```
use feature 'try';

try {
    risky();
}
catch ($e) {
    warn "risky failed: $e";
}
```

Επιθεώρηση ενός δομημένου αντικειμένου εξαίρεσης:

```
try {
    process($record);
}
catch ($e) {
    if (ref($e) && $e->isa('My::NotFound')) {
        return;                         # missing is fine
    }
    die $e;                             # re-throw everything else
}
```

Επανεκπομπή αμετάβλητης με `die`. Επειδή η `catch` δεν αγγίζει το `$@`, επανεκπέμπετε τη μεταβλητή που δεσμεύσατε, όχι το `$@`:

```
try {
    step();
}
catch ($e) {
    log_error($e);
    die $e;                             # propagates to the next
 ↪handler
}
```

Παραγωγή τιμής από την κατασκευή χωρίς `return`. Η χρήση της `return` μέσα σε `try` ή `catch` επιστρέφει από την **περικλείουσα υπορουτίνα**, όχι από την κατασκευή - δείτε

Οριακές περιπτώσεις:

```
sub safe_parse {
    my ($text) = @_;
    try {
        parse($text);
    }
    catch ($e) {
        warn "parse error: $e";
        undef;                          # value of the catch block
    }
}
```

Ένα μπλοκ `catch` που το ίδιο πεθαίνει - η νέα εξαίρεση διαδίδεται προς τα έξω στον επόμενο περικλείοντα χειριστή (`try`/`catch`, `eval`, ή έξοδος προγράμματος):

```
try {
    open_db();
}
catch ($e) {
    cleanup();
    die "open_db failed: $e";          # thrown from inside catch
}
```

Οριακές περιπτώσεις

- **Δεν μπορεί να σταθεί μόνη της.** Το `catch (VAR) { ... }` χωρίς προηγούμενο μπλοκ `try` είναι συντακτικό σφάλμα. Η `catch` αναλύεται ως μέρος της κατασκευής `try`, όχι ως ανεξάρτητη δεσμευμένη λέξη.
- **Ακριβώς ένα βαθμωτό στις παρενθέσεις.** Οι παρενθέσεις είναι υποχρεωτικές, πρέπει να δηλώνουν ακριβώς ένα βαθμωτό, και η `my` υπονοείται. Το `catch { ... }` χωρίς παρενθέσεις είναι συντακτικό σφάλμα.
- **Η VAR έχει εμβέλεια μπλοκ.** Υπάρχει μόνο μέσα στο μπλοκ `catch`. Ένα μπλοκ `finally` που ακολουθεί **δεν** τη βλέπει.
- **Οι ψευδείς εξαιρέσεις εξακολουθούν να ενεργοποιούν τον χειριστή.** Η `try` διακρίνει το «πέθανε» από το «ολοκληρώθηκε κανονικά» με μια σημαία που τίθεται κατά το ξετύλιγμα, όχι με την αληθότητα της τιμής εξαίρεσης. Τα `die 0`, `die ""` και `die undef` εισέρχονται όλα στο μπλοκ `catch`· η VAR λαμβάνει την τιμή όπως εκτοξεύθηκε (με τον συνηθισμένο εξαναγκασμό της `die` του `undef` σε `"Died at ..."`).
- **Το `$@` δεν τίθεται.** Κώδικας που αναζητά το `$@` μέσα σε μπλοκ `catch` διαβάζει το `$@` του καλούντος, όχι την εξαίρεση που μόλις συνέλαβε. Χρησιμοποιήστε την VAR.
- **Ένα μπλοκ `catch` μπορεί να πεθάνει.** Οι εξαιρέσεις που εγείρονται μέσα στο `catch` **δεν** συλλαμβάνονται εκ νέου από την ίδια κατασκευή. Διαδίδονται στον επόμενο περικλείοντα χειριστή. Αυτός είναι ο μηχανισμός για επανεκπομπή (`die $e`) και για αναφορά αποτυχιών καθαρισμού από τον ίδιο τον χειριστή.
- **Η `return` μέσα στο `catch` επιστρέφει από την περικλείουσα υπορουτίνα**, όχι από το μπλοκ `catch`. Αυτό διαφέρει από το `eval` BLOCK, όπου η `return` θα έβγαινε

μόνο από την `eval`. Για να αποδώσετε τιμή *από* την κατασκευή, χρησιμοποιήστε την τελική έκφραση του μπλοκ αντί της `return`.

- **Τα στοιχεία ελέγχου βρόχου (`last`, `next`, `redo`) μέσα στο `catch`** στοχεύουν περικλείοντα βρόχο, όχι την κατασκευή. Η κατασκευή `try`/`catch` δεν αποτελεί η ίδια βρόχο.
- **Η `caller()` δεν βλέπει το πλαίσιο `catch`.** Επειδή η `catch` δεν παρεμποδίζει την `return`, είναι αόρατη στην `caller`, όπως είναι και τα `while` ή `foreach`.
- **Συνδυάζεται μόνο με την αμέσως προηγούμενη `try`.** Δεν μπορείτε να έχετε δύο μπλοκ `catch`, ούτε να τοποθετήσετε εντολές μεταξύ `try` και `catch`. Το προαιρετικό μπλοκ `finally`, αν υπάρχει, πρέπει να ακολουθεί την `catch`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `try` - το μπλοκ του οποίου την αποτυχία χειρίζεται αυτή η `catch`· πάντοτε συνδυάζεται με ακριβώς μία `catch`
- `finally` - προαιρετικό τρίτο μπλοκ που εκτελείται ανεξάρτητα από το αν πέθανε το μπλοκ `try`· δεν μπορεί να κάνει `return` ή να χρησιμοποιήσει στοιχεία ελέγχου βρόχου
- `die` - παράγει την τιμή εξαίρεσης που λαμβάνει η VAR· χρησιμοποιήστε την μέσα στο `catch` για επανεκπομπή
- `eval` - παλαιότερο ιδίωμα χειρισμού εξαιρέσεων που χρησιμοποιεί το `$@`· σε νέο κώδικα προτιμήστε την `try`/`catch` για σαφέστερη εμβέλεια και χωρίς παγίδες του `$@`
- `$@` - παραμένει ανέγγιχτο από την `try`/`catch`· επιθεωρήστε την VAR αντί γι” αυτό

Filehandles, αρχεία, κατάλογοι

chdir

Αλλάζει τον τρέχοντα κατάλογο εργασίας της διεργασίας.

Η `chdir` είναι το περιτύλιγμα της Perl γύρω από τις κλήσεις συστήματος `chdir(2)` και - όπου ο πυρήνας το υποστηρίζει - `fchdir(2)`. Αλλάζει τον κατάλογο ως προς τον οποίο επιλύονται οι σχετικές διαδρομές για την τρέχουσα διεργασία και, σε περίπτωση επιτυχίας, διατηρεί αυτή την αλλαγή για κάθε επόμενη λειτουργία αρχείου στο πρόγραμμα.

Σύνοψη

```
chdir EXPR
chdir FILEHANDLE
chdir DIRHANDLE
chdir
```

Τι επιστρέφεται

1 σε επιτυχία, `0` σε αποτυχία. Σε αποτυχία η `$!` τίθεται στο υποκείμενο `errno` ώστε να μπορείτε να ξεχωρίσετε το «no such directory» (ENOENT) από το «permission denied» (EACCES) από το «not a directory» (ENOTDIR):

```
chdir $path
    or die "chdir $path failed: $!";
```

Η επιστροφή είναι πάντα μια απλή λογική τιμή - δεν υπάρχει χωριστή τιμή «άλλαξε σε κανέναν κατάλογο επειδή δεν δόθηκε όρισμα». Η μορφή χωρίς όρισμα είτε πετυχαίνει αλλάζοντας στον αρχικό κατάλογο (home), είτε αποτυγχάνει όπως κάθε άλλη κλήση.

Καθολική κατάσταση που επηρεάζει

- `%ENV` - διαβάζεται όταν η `chdir` καλείται χωρίς όρισμα. Η σειρά αναζήτησης είναι πρώτα `$ENV{HOME}`, μετά `$ENV{LOGDIR}`. Αν κανένα δεν είναι ορισμένο, η `chdir` δεν κάνει τίποτα και επιστρέφει `0`.
- `$!` - τίθεται σε αποτυχία στο `errno` που αναφέρει ο πυρήνας.

Η `chdir` **δεν** ενημερώνει το `$ENV{PWD}`. Τα κελύφη διατηρούν το `PWD` τα ίδια· ένα πρόγραμμα Perl που νοιάζεται για το `PWD` στις θυγατρικές διεργασίες πρέπει να αναθέσει τιμή στο `$ENV{PWD}` μετά από επιτυχημένη `chdir`.

Μορφές ορίσματος

- **`chdir EXPR`** - η `EXPR` μετατρέπεται σε συμβολοσειρά και περνά στην `chdir(2)` ως διαδρομή. Οι σχετικές διαδρομές επιλύονται ως προς τον τρέχοντα κατάλογο εργασίας τη στιγμή της κλήσης.
- **`chdir FILEHANDLE`** και **`chdir DIRHANDLE`** - σε Linux (και σε κάθε σύστημα με `fchdir(2)`) ο πυρήνας αλλάζει κατάλογο σε αυτόν στον οποίο αναφέρεται το handle. Το `FILEHANDLE` εδώ σημαίνει filehandle ανοιγμένο σε κατάλογο, ή directory handle από την `opendir`. Η χρήση της μορφής handle αποφεύγει μια κούρσα TOCTOU μεταξύ της επίλυσης μιας διαδρομής και της αλλαγής εντός αυτής.
- **`chdir`** (χωρίς όρισμα) - αλλάζει στο `$ENV{HOME}`, με εφεδρεία το `$ENV{LOGDIR}` αν το `HOME` δεν είναι ορισμένο. Αν και τα δύο δεν είναι ορισμένα η κλήση αποτυγχάνει και η `$!` παραμένει αμετάβλητη· ελέγξτε την τιμή επιστροφής, όχι το `$!`.

Παραδείγματα

Μετάβαση σε κατάλογο και έξοδος αν δεν υπάρχει:

```
chdir "/var/log"
    or die "cannot enter /var/log: $!";
```

Εκτέλεση ενός μπλοκ εργασιών σε διαφορετικό κατάλογο και επιστροφή στο σημείο εκκίνησης:

```
use Cwd qw(getcwd);
my $origin = getcwd;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
chdir $work_dir or die "chdir $work_dir: $!";
# ... do work relative to $work_dir ...
chdir $origin  or die "chdir $origin: $!";
```

Αλλαγή μέσω directory handle (χωρίς κούρσα σε συστήματα με `fchdir(2)`):

```
opendir my $dh, $path or die "opendir $path: $!";
chdir $dh             or die "fchdir $path: $!";
```

Πήγαινε σπίτι:

```
chdir or die "no HOME/LOGDIR in environment";
```

Δοκιμή λίστας υποψηφίων, με στάση στον πρώτο που λειτουργεί:

```
for my $d ("/srv/app", "/opt/app", "/usr/local/app") {
    last if chdir $d;
}
```

Οριακές περιπτώσεις

- **Κενή συμβολοσειρά ή `undef`**: τόσο η `chdir ""` όσο και η `chdir undef` αποτυγχάνουν και θέτουν την `$!` σε `ENOENT`. **Δεν** καταφεύγουν στο `$HOME` - η μορφή χωρίς όρισμα ενεργοποιείται από την *απουσία* ορίσματος, όχι από ένα όρισμα που τυχαίνει να είναι κενό.
  - the no-argument form is triggered by *absence* of an argument, not by an argument that happens to be empty.
- **Σχετικές διαδρομές σε thread με δικό του cwd**: ο τρέχων κατάλογος εργασίας του πυρήνα είναι ανά διεργασία, όχι ανά thread, στο Linux. Δύο threads που καλούν `chdir` ταυτόχρονα μπαίνουν σε κούρσα.
- **Symlink σε κατάλογο**: η `chdir` ακολουθεί τα symlinks, καταλήγοντας στον στόχο. Χρησιμοποιήστε πρώτα την `readlink` ή την `Cwd::abs_path` αν χρειάζεται να ξέρετε τον πραγματικό προορισμό.
- **Διαγραμμένος κατάλογος**: αν ο κατάλογος αφαιρέθηκε αφού μπήκατε σε αυτόν, οι σχετικές λειτουργίες από μέσα αποτυγχάνουν με `ENOENT` και τα περισσότερα κελύφη εμφανίζουν τη διαδρομή ως `(deleted)`. Μπορείτε ακόμη να βγείτε με `chdir` δίνοντας απόλυτη διαδρομή.
- **Μορφή handle σε συστήματα χωρίς `fchdir(2)`**: εγείρει εξαίρεση. Το Linux την υποστηρίζει πάντα· ο περιορισμός είναι ζήτημα φορητότητας για κώδικα που τρέχει επίσης σε παλαιότερα συστήματα.
- **Μη έμπιστη είσοδος**: το να περνάτε στην `chdir` διαδρομή που παρέχεται από τον χρήστη χωρίς επικύρωση επιτρέπει στον καλούντα να ανακατευθύνει όλες τις επόμενες λειτουργίες σχετικής διαδρομής στο πρόγραμμα. Καταναγκαστικά κανονικοποιήστε με `Cwd::abs_path` και ελέγξτε έναντι λίστας επιτρεπτών πριν την κλήση αν η είσοδος διασχίζει όριο εμπιστοσύνης.
- **Μετά από `fork`**: το παιδί κληρονομεί το cwd του γονέα. Μια `chdir` στο παιδί δεν επηρεάζει τον γονέα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `mkdir` - δημιουργία καταλόγου· συνδυάστε την με `chdir` όταν φτιάχνετε χώρο εργασίας και εισέρχεστε σε αυτόν
- `rmdir` - αφαίρεση καταλόγου· μη κάνετε `rmdir` στο cwd, βγείτε πρώτα με `chdir`
- `opendir` - άνοιγμα directory handle χρησιμοποιήσιμου ως όρισμα στην `chdir` για αλλαγές καταλόγου χωρίς κούρσα
- `chroot` - ισχυρότερος περιορισμός από την `chdir`· αλλάζει τη ρίζα του συστήματος αρχείων για τη διεργασία, όχι μόνο το cwd
- `$ENV{HOME}` - συμβουλεύεται από τη μορφή χωρίς όρισμα της `chdir` πριν την εφεδρεία στο `$ENV{LOGDIR}`
- `$!` - μεταφέρει το `errno` όταν η `chdir` επιστρέφει `0`

Filehandles, αρχεία, κατάλογοι

chmod

Αλλάζει τα bits δικαιωμάτων μιας λίστας αρχείων.

Η `chmod` ορίζει σε κάθε αρχείο της `LIST` τα δικαιώματα `MODE`. Η `MODE` είναι **αριθμητική** τιμή - τα δώδεκα κατώτερα bits είναι η συνηθισμένη τριάδα δικαιωμάτων του Unix συν τα bits setuid, setgid και sticky, ακριβώς όπως τα δέχεται η `chmod(2)`. Η `LIST` δέχεται τόσο απλά ονόματα διαδρομής όσο και ανοιχτά filehandles. Επιστρέφει τον αριθμό των αρχείων για τα οποία πέτυχε η υποκείμενη κλήση συστήματος.

Σύνοψη

```
chmod MODE, LIST
```

Τι επιστρέφεται

Ο αριθμός των αρχείων που άλλαξαν με επιτυχία, ως ακέραιος. Το μηδέν σημαίνει ότι κάθε κλήση απέτυχε· τιμή μικρότερη του `scalar @files` σημαίνει μερική αποτυχία. Η `chmod` **δεν** εκτοξεύει εξαίρεση σε αποτυχία επιμέρους αρχείου - συνεχίζει και αθροίζει τις επιτυχίες.

Η `$!` αντικατοπτρίζει το σφάλμα μόνο από το **τελευταίο** αρχείο που απέτυχε. Σε μια εκτέλεση με ανάμικτη επιτυχία/αποτυχία δεν υπάρχει ενσωματωμένος τρόπος να μάθετε ποια αρχεία απέτυχαν· αν χρειάζεστε αναφορά σφαλμάτων ανά αρχείο, καλέστε την `chmod` μία φορά ανά αρχείο και ελέγξτε την τιμή επιστροφής κάθε φορά:

```
for my $f (@files) {
    chmod 0644, $f
        or warn "chmod $f: $!";
}
```

Σωστή σύνταξη της MODE

Η MODE είναι ακέραιος. Οι τρεις τρόποι να γραφτεί:

- **Οκταδικό κυριολεκτικό με αρχικό μηδέν**: `0644`, `0755`, `0600`. Αυτή είναι η συμβατική μορφή και αντιστοιχεί στα ψηφία που θα πληκτρολογούσατε στην `chmod(1)`.
- **Οκταδικό κυριολεκτικό με πρόθεμα `0o`**: `0o644`, `0o755`. Γίνεται δεκτό από την Perl 5.34 και μετά· ισοδύναμο με τη μορφή αρχικού μηδενός και πιο εύκολο να διαβαστεί ως «οκταδικό» με μια ματιά.
- **Συμβολικές σταθερές από το `Fcntl`**: η έκφραση `use Fcntl qw(:mode); S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH` αποτιμάται σε `0755`. Χρήσιμο όταν ένα σύνολο δικαιωμάτων χτίζεται από επιμέρους κομμάτια.

Ένα όρισμα συμβολοσειράς είναι σχεδόν πάντα σφάλμα:

```
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+)
```

Αν η mode έρχεται πραγματικά ως συμβολοσειρά (αρχείο διαμόρφωσης, όρισμα CLI), μετατρέψτε την πρώτα με την `oct`:

```
my $mode = "0644";       # string from config
chmod oct($mode), "foo"; # oct("0644") == 0644
```

Καθολική κατάσταση που επηρεάζει

Ορίζει την `$!` σε οποιαδήποτε αποτυχία ώστε να αντιστοιχεί στην τελευταία αποτυχημένη κλήση `chmod(2)` / `fchmod(2)`. Δεν διαβάζει καθολικές μεταβλητές του διερμηνέα.

Παραδείγματα

Αλλαγή ενός αρχείου, έλεγχος επιτυχίας:

```
chmod 0755, "install.sh"
    or die "chmod install.sh: $!";
```

Αλλαγή πολλών αρχείων σε μία κλήση, αποτύπωση του πλήθους:

```
my $cnt = chmod 0644, "a.txt", "b.txt", "c.txt";
print "updated $cnt of 3 files\n";
```

Με πίνακα ονομάτων διαδρομής:

```
my @executables = glob "bin/*";
chmod 0755, @executables;
```

Διατήρηση των υπαρχόντων bits, προσθήκη δικαιώματος εγγραφής για τον ιδιοκτήτη. Διαβάζει την mode με την `stat`, εφαρμόζει μάσκα για τα bits δικαιωμάτων, κάνει OR με το νέο bit και την γράφει πίσω:

```
open my $fh, "<", "foo" or die $!;
my $perm = (stat $fh)[2] & 07777;
chmod $perm | 0200, $fh;
```

Συμβολική μορφή μέσω του `Fcntl`:

```
use Fcntl qw(:mode);
chmod S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH, @executables;
# identical to: chmod 0755, @executables;
```

Αναφορά σφαλμάτων ανά αρχείο (χρειάζεται για να μάθετε *ποιο* αρχείο απέτυχε σε κλήση πολλαπλών αρχείων):

```
my @failed;
for my $f (@files) {
    push @failed, $f unless chmod 0644, $f;
}
warn "chmod failed on: @failed\n" if @failed;
```

Οριακές περιπτώσεις

- **Κενή λίστα**: η `chmod 0644` επιστρέφει `0`. Η mode αποτιμάται αλλά δεν γίνεται καμία κλήση συστήματος.
- **Modes ως συμβολοσειρά**: τόσο η `chmod "0644", $f` όσο και η `chmod "+x", $f` είναι λανθασμένες. Η `chmod` δεν αναλύει ποτέ συμβολικές συμβολοσειρές mode - σε αντίθεση με την `chmod(1)`, η ενσωματωμένη συνάρτηση παίρνει αριθμό, όχι προδιαγραφή.
- **Filehandles έναντι barewords**: ένα bareword στη `LIST` είναι όνομα αρχείου, όχι filehandle. Περάστε ένα filehandle ως glob ή ως αναφορά glob:

  ```
  chmod 0644, *FH;          # filehandle
  chmod 0644, \*FH;         # filehandle
  chmod 0644, $fh;          # filehandle (lexical)
  chmod 0644, "FH";         # filename "FH" - almost certainly a␣
  ↪bug
  ```

  Σε συστήματα χωρίς `fchmod(2)`, το πέρασμα filehandle εκπέμπει εξαίρεση. Το Linux υποστηρίζει την `fchmod(2)`, οπότε στις πλατφόρμες-στόχους της pperl αυτό λειτουργεί πάντα.
- **Bits setuid / setgid / sticky**: τα τρία ανώτερα bits της λέξης mode (`04000`, `02000`, `01000`) θέτουν αντίστοιχα setuid, setgid και το sticky bit. Το `0o4755` δημιουργεί setuid εκτελέσιμο· το `0o2775` έναν setgid κατάλογο.
- **Symlinks**: η `chmod` ακολουθεί τα symlinks. Η αλλαγή δικαιωμάτων στο ίδιο το symlink δεν είναι φορητή και δεν εκτίθεται από αυτή την ενσωματωμένη συνάρτηση

  - χρησιμοποιήστε την `lchmod` από κάποιο άρθρωμα αν η πλατφόρμα την υποστηρίζει.

- **Δικαίωμα αλλαγής δικαιωμάτων**: μόνο ο ιδιοκτήτης του αρχείου (και ο root) μπορούν να καλέσουν την `chmod(2)` σε ένα αρχείο. Η αποτυχία θέτει την `$!` σε `EPERM`.
- **Ανύπαρκτες διαδρομές**: η chmod σε ανύπαρκτη διαδρομή θέτει την `$!` σε `ENOENT` και συνεισφέρει `0` στο πλήθος επιτυχιών. Δεν προειδοποιεί ούτε τερματίζει.
- **Η umask δεν εφαρμόζεται**: η `umask` μασκάρει τη mode *νεοδημιουργημένων* αρχείων από τις `open`, `sysopen`, `mkdir` και παρόμοιες. Η chmod ορίζει τη mode κυριολεκτικά - χωρίς μασκάρισμα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `chown` - η αδελφή κλήση για αλλαγή ιδιοκτήτη/ομάδας· ίδια μορφή λίστας-και-επιστροφής-πλήθους
- `stat` - διαβάστε την τρέχουσα mode ώστε να την τροποποιήσετε αντί να την αντικαταστήσετε εξ ολοκλήρου
- `umask` - προεπιλεγμένη μάσκα δικαιωμάτων που εφαρμόζεται σε *νέα* αρχεία· άσχετη με την chmod, η οποία θέτει τα bits κυριολεκτικά
- `open` / `sysopen` - δημιουργούν αρχεία· το όρισμα mode της `sysopen` μασκάρεται από την `umask` κατά τη δημιουργία και κατόπιν μπορεί να γίνει πιο αυστηρό ή πιο χαλαρό με την chmod
- `oct` - αναλύει το `"0644"` στον ακέραιο `0644` όταν η mode έρχεται ως συμβολοσειρά
- `$!` - σφάλμα συστήματος από το τελευταίο αρχείο που απέτυχε· έχει νόημα μόνο για κλήσεις ενός αρχείου ή αμέσως μετά από αποτυχημένη κλήση ανά αρχείο μέσα σε βρόχο

Βαθμωτά και συμβολοσειρές

chomp

Αφαιρεί τον τελικό διαχωριστή εγγραφών εισόδου από μια συμβολοσειρά, επιτόπια.

Η chomp αφαιρεί κάθε τελική υποσυμβολοσειρά που ταιριάζει με την τρέχουσα τιμή της `$/` από το όρισμά της και επιστρέφει το σύνολο των χαρακτήρων που αφαιρέθηκαν από κάθε όρισμα που επηρέασε. Μεταβάλλει το όρισμά της· η τιμή επιστροφής είναι μετρητής, όχι η συμβολοσειρά μετά το chomp. Αν δεν δοθεί όρισμα, λειτουργεί επί της `$_`. Η chomp είναι ο ασφαλέστερος αντίστοιχος της `chop`: αφαιρεί τον διαχωριστή μόνο αν είναι πράγματι παρών, και σέβεται την `$/` ώστε να κάνει το σωστό σε αρχεία που τερματίζουν χωρίς τελικό newline ή χρησιμοποιούν τερματιστή εγγραφών που δεν είναι newline.

Σύνοψη

```
chomp VARIABLE
chomp( LIST )
chomp
```

Τι επιστρέφεται

Ο συνολικός αριθμός χαρακτήρων που αφαιρέθηκαν, ως ακέραιος. Συνήθως `0` (δεν υπήρχε διαχωριστής) ή 1 (αφαιρέθηκε ένα `"\n"` υπό την προεπιλογή της `$/`). Όταν η `$/` είναι συμβολοσειρά πολλών χαρακτήρων, ένα επιτυχημένο chomp επιστρέφει το μήκος της. Όταν γίνεται chomp σε λίστα, ο μετρητής αθροίζεται σε όλα τα στοιχεία.

Η τιμή μετά το chomp βρίσκεται στη μεταβλητή που περάσατε· η chomp καλείται για το αποτέλεσμα, όχι για την τιμή επιστροφής της:

```
my $line = "hello\n";
my $n = chomp $line;        # $line is "hello", $n is 1
```

Καθολική κατάσταση που επηρεάζει

- `$/` - **διαχωριστής εγγραφών εισόδου**. Η chomp διαβάζει την `$/` σε κάθε κλήση και αφαιρεί μια τελική εμφάνιση της τρέχουσας τιμής της. Η αλλαγή της `$/` αλλάζει αυτό που αφαιρεί η chomp· αυτό είναι συνήθως αυτό που θέλετε όταν θέτετε την `$/` για να διαβάσετε εγγραφές με προσαρμοσμένο διαχωρισμό.
- `$_` - καταναλώνεται ως το προεπιλεγμένο όρισμα όταν η chomp καλείται χωρίς όρισμα. Η `chomp;` μέσα σε `while (<>) { ... }` είναι το κανονικό ιδίωμα.
- Επαναλήπτης πίνακα κατακερματισμού - το chomp σε πίνακα κατακερματισμού επαναφέρει τον επαναλήπτη του `each` (δείτε *Οριακές περιπτώσεις*).

Παραδείγματα

Η καθημερινή χρήση - αφαιρέστε το τελικό newline από την είσοδο:

```
while (my $line = <$fh>) {
    chomp $line;
    # process $line without the trailing "\n"
}
```

Η μορφή χωρίς όρισμα λειτουργεί επί της `$_`:

```
while (<>) {
    chomp;                          # removes "\n" from $_
    my @fields = split /:/;
}
```

Chomp σε ανάθεση lvalue - οι παρενθέσεις είναι **απαραίτητες** ώστε το όρισμα στην chomp να είναι ολόκληρη η ανάθεση, όχι μόνο η μεταβλητή:

```
chomp(my $cwd    = `pwd`);          # strip "\n" from command output
chomp(my $answer = <STDIN>);        # strip "\n" from user input
```

Chomp σε λίστα - γίνεται chomp σε κάθε στοιχείο και επιστρέφεται ο συνολικός μετρητής:

```
my @lines = ("a\n", "b\n", "no-nl");
my $n = chomp @lines;               # @lines = ("a", "b", "no-nl"),
 ↪$n = 2
```

Προσαρμοσμένος διαχωριστής εγγραφών - η chomp ακολουθεί ό,τι έχει τεθεί στην `$/`:

```
local $/ = "\r\n";
my $line = "GET / HTTP/1.0\r\n";
chomp $line;                        # removes "\r\n", returns 2
```

Λειτουργία παραγράφου - το `$/ = ""` κάνει την chomp να αφαιρέσει **όλα** τα τελικά newlines από τη συμβολοσειρά:

```
local $/ = "";
my $para = "line one\nline two\n\n\n";
chomp $para;                        # $para is "line one\nline two"
```

Οριακές περιπτώσεις

- **Κανένας διαχωριστής παρών**: no-op. Επιστρέφει `0` και αφήνει το όρισμα αμετάβλητο. Αυτό είναι το όλο νόημα της chomp έναντι της chop
  - safe to call on strings that may or may not end in `$/`.
- **Όρισμα undef**: το chomp σε απροσδιόριστο βαθμωτό το αφήνει απροσδιόριστο και επιστρέφει `0`. Υπό `use warnings` αυτό εκπέμπει προειδοποίηση `uninitialized`.
- **Όρισμα πίνακα κατακερματισμού**: η chomp %h κάνει chomp σε κάθε τιμή (όχι στα κλειδιά) και **επαναφέρει τον επαναλήπτη each του πίνακα**. Ένας βρόχος `while (my ($k, $v) = each %h)` σε εξέλιξη πάνω στον %h θα διαταραχθεί.
- **Λειτουργία slurp (`$/ = undef`)** και **λειτουργία εγγραφών σταθερού μήκους** (`$/` τεθειμένη σε αναφορά σε ακέραιο): η chomp δεν αφαιρεί τίποτα και επιστρέφει `0`. Δεν υπάρχει έννοια τελικού διαχωριστή σε αυτές τις λειτουργίες.
- **Παγίδα με παρενθέσεις στην ανάθεση**: χωρίς παρενθέσεις, η chomp δένει ισχυρότερα από το =:

  ```
  chomp  $cwd = `pwd`;              # parsed as (chomp $cwd) =␣
   ↪`pwd`;
  chomp( $cwd = `pwd` );            # what you meant
  ```

  Ίδια παγίδα με λίστα κόμματος χωρίς παρενθέσεις:

  ```
  chomp  $x, $y;                    # parsed as chomp($x), $y;
  chomp( $x, $y );                  # chomps both
  ```

- **Λειτουργεί οποιοδήποτε lvalue**, όχι μόνο απλά βαθμωτά. Στοιχεία πίνακα και πίνακα κατακερματισμού, τομές, αποαναφορές και εκφράσεις ανάθεσης είναι όλα νόμιμοι στόχοι.
- **Σταθερές και τιμές μόνο για ανάγνωση**: το chomp σε συμβολοσειρά μόνο για ανάγνωση εκπέμπει σφάλμα `Modification of a read-only value attempted`. Οι κυριολεκτικές συμβολοσειρές είναι μόνο για ανάγνωση.
- **Πολυ-byte `$/`**: η chomp συγκρίνει bytes της συμβολοσειράς με bytes της `$/`. Αν η συμβολοσειρά και η `$/` διαφωνούν ως προς την κωδικοποίηση (η μία έχει τεθεί η σημαία UTF-8, η άλλη όχι), η σύγκριση γίνεται στην εσωτερική αναπαράσταση bytes και μπορεί να σας εκπλήξει. Κρατήστε και τις δύο πλευρές στην ίδια κωδικοποίηση.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `chop` - αφαιρεί χωρίς όρους τον **τελευταίο χαρακτήρα** και τον επιστρέφει· παλαιότερο, λιγότερο εκλεπτυσμένο εργαλείο που έχει αντικατασταθεί από την chomp για καθαρισμό εισόδου
- `split` - συχνά το επόμενο βήμα μετά την chomp, διασπά την πλέον καθαρισμένη εγγραφή σε πεδία
- `readline` / `<>` - διαβάζει εγγραφές που τερματίζονται από την `$/`· η chomp αναιρεί τον τερματιστή που άφησε η `readline`
- `each` - ο επαναλήπτης πίνακα κατακερματισμού που η chomp `%hash` επαναφέρει
- `$/` - διαχωριστής εγγραφών εισόδου· η chomp το διαβάζει σε κάθε κλήση
- `$_` - προεπιλεγμένο όρισμα όταν η chomp καλείται χωρίς ρητό στόχο

Βαθμωτά και συμβολοσειρές

chop

Αφαιρεί τον τελευταίο χαρακτήρα μιας συμβολοσειράς και τον επιστρέφει.

Η chop συντομεύει επιτόπου το όρισμά της κατά ακριβώς έναν χαρακτήρα και επιστρέφει τον χαρακτήρα που αφαιρέθηκε. Δεν σαρώνει τη συμβολοσειρά και δεν την αντιγράφει - απλώς μετακινεί το τέλος. Αν παραλειφθεί η VARIABLE, η chop λειτουργεί στο `$_`. Αν η VARIABLE είναι hash, η chop κάνει chop κάθε τιμή του hash (όχι τα κλειδιά) και επαναφέρει τον επαναλήπτη `each` του hash ως παρενέργεια. Κάθε lvalue είναι έγκυρος στόχος, συμπεριλαμβανομένης μιας έκφρασης ανάθεσης.

Σύνοψη

```
chop VARIABLE
chop( LIST )
chop
```

Τι επιστρέφεται

Ο χαρακτήρας που αφαιρέθηκε, ως συμβολοσειρά. Αν ο στόχος ήταν κενός ή απροσδιόριστος, επιστρέφεται η κενή συμβολοσειρά και ο στόχος μένει αμετάβλητος. Όταν η chop εφαρμόζεται σε λίστα, κάθε στοιχείο υφίσταται chop, αλλά επιστρέφεται μόνο ο χαρακτήρας που αφαιρέθηκε από το **τελευταίο** στοιχείο.

```
my $last = chop $line;     # one character, or "" if $line was empty
```

Καθολική κατάσταση που επηρεάζει

Διαβάζει το $_ όταν καλείται χωρίς όρισμα. Καμία άλλη ειδική μεταβλητή δεν συμβουλεύεται - ειδικότερα, η chop **δεν** επηρεάζεται από το $/ (τον διαχωριστή εγγραφών εισόδου), που είναι έγνοια της chomp.

Παραδείγματα

Chop ενός βαθμωτού:

```
my $s = "hello";
my $c = chop $s;              # $s is now "hell", $c is "o"
```

Chop στο $_ όταν δεν δίνεται όρισμα - ιδιωματικό μέσα σε βρόχο ανάγνωσης που θέλει τον τερματιστή να φύγει ανεξάρτητα από το τι είναι:

```
while (<$fh>) {
    chop;                     # drops whatever character ends the line
    push @rows, $_;
}
```

Chop μιας λίστας. Κάθε στοιχείο χάνει τον τελευταίο του χαρακτήρα· η τιμή επιστροφής είναι ο χαρακτήρας που αφαιρέθηκε μόνο από το **τελευταίο** στοιχείο:

```
my @lines = ("foo\n", "bar\n", "baz!");
my $last  = chop @lines;    # @lines is ("foo","bar","baz"), $last
↪is "!"
```

Chop ενός hash. Οι τιμές υφίστανται chop, τα κλειδιά όχι, και ο επαναλήπτης each επαναφέρεται:

```
my %h = (a => "one\n", b => "two\n");
chop %h;                      # %h is (a => "one", b => "two")
```

Chop ανάθεσης - η chop δουλεύει με κάθε lvalue, και η ανάθεση επιστρέφει lvalue:

```
chop(my $cwd = `pwd`);        # strip the trailing newline from pwd's
↪output
```

Διατήρηση όλων εκτός από τον τελευταίο χαρακτήρα χωρίς τροποποίηση του πρωτοτύπου - δεν είναι δουλειά για την chop, που μεταλλάσσει. Χρησιμοποιήστε αντί αυτής την substr με αρνητικό μήκος:

```
my $head = substr($s, 0, -1);   # $s is unchanged
```

Οριακές περιπτώσεις

- **Κενός ή απροσδιόριστος στόχος**: επιστρέφει την κενή συμβολοσειρά "" και αφήνει τον στόχο ως έχει. Η chop σε undef *δεν* εγείρει από μόνη της προειδοποίηση· απλώς παράγει "".
- **Στόχος μόνο για ανάγνωση**: εγείρει `Modification of a read-only value attempted`. Αυτό περιλαμβάνει κυριολεκτικές τιμές συμβολοσειρών και τιμές σημειωμένες ως μόνο για ανάγνωση μέσω της `Internals::SvREADONLY`.
- **Συμβολοσειρές UTF-8**: η chop αφαιρεί έναν **χαρακτήρα**, όχι ένα byte. Ένας χαρακτήρας πολλαπλών bytes στο τέλος της συμβολοσειράς αφαιρείται ολόκληρος και επιστρέφεται ως συμβολοσειρά UTF-8. Η σημαία UTF-8 του στόχου διατηρείται σε ό,τι απομένει.
- **Τιμή επιστροφής λίστας**: επιστρέφεται μόνο ο χαρακτήρας που αφαιρέθηκε από το τελευταίο στοιχείο· οι υπόλοιποι απορρίπτονται. Αν χρειάζεστε κάθε αφαιρούμενο χαρακτήρα, κάντε chop στα στοιχεία μεμονωμένα σε έναν βρόχο.
- **Ένας χαρακτήρας έναντι τερματισμού γραμμής**: η chop αφαιρεί άνευ όρων **έναν** χαρακτήρα, ό,τι κι αν είναι αυτός - συμπεριλαμβανομένων κενών, ψηφίων ή γραμμάτων. Για να αφαιρέσετε υπό συνθήκη έναν τερματιστή γραμμής (αυτό που λέει το `$/` ότι μοιάζει), χρησιμοποιήστε την chomp.
- **Παρενθεσιοποίηση του αναλυτή**: η `chop($x = $y)` κάνει chop στο αποτέλεσμα της ανάθεσης, που είναι το `$x`. Η `chop $x, $y` αναλύεται ως `chop($x), $y` - το κόμμα δεν επεκτείνει τη λίστα ορισμάτων της chop χωρίς παρενθέσεις. Γράψτε `chop($x, $y)` όταν εννοείτε τη μορφή λίστας.
- **Επαναφορά επανάληψης του hash**: επειδή η `chop %h` διασχίζει τις τιμές μέσω του επαναλήπτη του hash, οποιοσδήποτε εν εξελίξει βρόχος each πάνω από το %h ακυρώνεται.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- chomp - υπό συνθήκη ξαδέρφη: αφαιρεί τελικό `$/`, όχι έναν τελευταίο χαρακτήρα χωρίς όρους· επιστρέφει τον αριθμό των χαρακτήρων που αφαιρέθηκαν αντί για τον ίδιο τον χαρακτήρα
- substr - μη καταστροφική εναλλακτική· η `substr($s, 0, -1)` επιστρέφει όλα εκτός από τον τελευταίο χαρακτήρα χωρίς να μεταλλάσσει το `$s`
- `s///` - η `$s =~ s/.\z//s` είναι το αντίστοιχο σε regex και αυτή από την οποία η chop είναι ρητά ταχύτερη, επειδή η chop ούτε σαρώνει ούτε αντιγράφει
- lc, uc - άλλοι (κατά προσέγγιση) επιτόπιοι μετασχηματισμοί συμβολοσειρών που δέχονται κάθε lvalue
- `$_` - ο σιωπηρός στόχος όταν η chop καλείται χωρίς όρισμα

- `$/` - **δεν** συμβουλεύεται από την chop· τεκμηριώνεται εδώ μόνο για να επισημανθεί η αντίθεση με την `chomp`

Filehandles, αρχεία, κατάλογοι

chown

Αλλάζει τον ιδιοκτήτη και την ομάδα μιας λίστας αρχείων.

Η chown παίρνει ένα αριθμητικό id χρήστη, ένα αριθμητικό id ομάδας και μια λίστα αρχείων, και αλλάζει την ιδιοκτησία σε κάθε αρχείο στο οποίο η καλούσα διεργασία επιτρέπεται να επεμβαίνει. Τα δύο πρώτα στοιχεία της `LIST` είναι τα νέα `UID` και `GID`, με αυτή τη σειρά· κάθε επόμενο στοιχείο είναι όνομα αρχείου (ή, σε συστήματα που υποστηρίζουν την `fchown(2)`, ένα filehandle). Η τιμή `-1` σε οποιαδήποτε από τις θέσεις id σημαίνει «άφησε αυτό το id αμετάβλητο» - χρήσιμη όταν θέλετε να αλλάξετε μόνο την ομάδα ή μόνο τον ιδιοκτήτη.

Σύνοψη

```
chown UID, GID, LIST
chown $uid, $gid, $file
chown $uid, $gid, @files
chown -1, $gid, @files          # change group only
```

Τι επιστρέφεται

Ο αριθμός των αρχείων στα οποία η ιδιοκτησία άλλαξε επιτυχώς, ως απλός ακέραιος. Όχι λογική τιμή - μια κλήση πάνω σε δέκα αρχεία που πετυχαίνει σε επτά επιστρέφει 7. Συγκρίνετε με το μήκος της λίστας όταν χρειάζεστε σημασιολογία «όλα ή τίποτα»:

```
my @files = ('a', 'b', 'c');
my $ok = chown $uid, $gid, @files;
$ok == @files
    or die "chown failed on ", scalar(@files) - $ok, " files: $!";
```

Όταν κάποιο μεμονωμένο αρχείο αποτυγχάνει, η `$!` κρατάει το errno από την **τελευταία** κλήση που απέτυχε. Μια τιμή επιστροφής μικρότερη από `scalar @files` είναι το αξιόπιστο σήμα «κάτι πήγε στραβά»· η `$!` σας λέει ποια ήταν η τελευταία αποτυχία, όχι κάθε αποτυχία.

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται στο errno της τελευταίας υποκείμενης κλήσης `chown(2)` που απέτυχε. Αμετάβλητη όταν κάθε αρχείο πετυχαίνει.

Μόνο αριθμητικά id

Η chown **δεν** δέχεται ονόματα χρήστη ή ομάδας. Περάστε τη συμβολοσειρά `"root"` εκεί που αναμένεται uid και θα πάρετε αριθμητική μετατροπή σε `0` (με προειδοποίηση `Argument … isn't numeric υπό use warnings`) - σχεδόν ποτέ αυτό που θέλετε. Μεταφράστε ονόματα σε αριθμούς με τις `getpwnam` και `getgrnam`:

```
my $uid = getpwnam('alice')  // die "no such user";
my $gid = getgrnam('staff')  // die "no such group";
chown $uid, $gid, @files;
```

Η επιστροφή της `getpwnam` σε περιβάλλον λίστας δίνει την πλήρη εγγραφή passwd· σε βαθμωτό περιβάλλον δίνει μόνο το uid, που είναι συνήθως αυτό που θέλει η `chown`.

Κανόνες δικαιωμάτων

Σε Linux και σε κάθε άλλο σύστημα POSIX, μόνο ο υπερχρήστης μπορεί να δώσει ένα αρχείο σε διαφορετικό ιδιοκτήτη. Ένας μη προνομιούχος χρήστης μπορεί να αλλάξει την **ομάδα** ενός αρχείου που του ανήκει, αλλά μόνο σε μία από τις δικές του δευτερεύουσες ομάδες. Η προσπάθεια να γίνει κάτι περισσότερο επιστρέφει αποτυχία με την `$!` τεθειμένη σε `EPERM`.

```
chown 0, 0, 'passwd.new'
    or warn "need root to chown: $!";
```

Ο περιορισμός επιβάλλεται από τον πυρήνα, όχι από την Perl - δεν υπάρχει προέλεγχος που θα σας γλίτωνε από μια αποτυχημένη κλήση συστήματος. Βασιστείτε στην τιμή επιστροφής.

Παραδείγματα

Αλλαγή και του ιδιοκτήτη και της ομάδας σε ένα μόνο αρχείο:

```
chown $uid, $gid, '/var/log/app.log'
    or die "chown failed: $!";
```

Αλλαγή μόνο της ομάδας, χωρίς να θιγεί ο ιδιοκτήτης:

```
chown -1, $gid, @files;
```

Μετάφραση ονομάτων σε αριθμητικά id πριν την κλήση:

```
my ($login, $pass, $uid, $gid) = getpwnam('www-data')
    or die "www-data not in passwd file";
chown $uid, $gid, glob('/var/www/*');
```

Μετρήστε πόσα αρχεία ανατέθηκαν πράγματι εκ νέου:

```
my @targets = glob('/srv/data/*.dat');
my $changed = chown $uid, $gid, @targets;
warn "only $changed of ", scalar(@targets), " files changed: $!"
    if $changed != @targets;
```

Chown μέσω filehandle σε σύστημα που υποστηρίζει την `fchown(2)`. Το filehandle πρέπει να είναι glob ή αναφορά σε glob - ένα bareword αντιμετωπίζεται ως όνομα αρχείου:

```
open my $fh, '>', '/tmp/out' or die $!;
chown $uid, $gid, $fh;
```

Οριακές περιπτώσεις

- **Κενή λίστα αρχείων**: `chown $uid, $gid;` - κανένα αρχείο για ενέργεια, επιστρέφει `0`, δεν αγγίζει την `$!`.
- **`-1` σε θέση id**: τυπική φρουρά POSIX με σημασία «άφησε αυτό το id αμετάβλητο». Λειτουργεί σε κάθε υποστηριζόμενη πλατφόρμα.
- **Συμβολοσειρά ως id**: μια μη αριθμητική συμβολοσειρά γίνεται σιωπηρά `0` (root / wheel) μετά τη συνηθισμένη αριθμητική μετατροπή. Υπό `use warnings` παίρνετε προειδοποίηση `Argument isn't numeric`, αλλά δεν υπάρχει σκληρό σφάλμα. Πάντα να περνάτε ακέραιο.
- **Αρχείο που δεν υπάρχει**: μετράει ως αποτυχία, δεν διακόπτει την κλήση. Τα υπόλοιπα αρχεία εξακολουθούν να επεξεργάζονται.
- **Στόχος symlink**: η chown ακολουθεί τα symlinks και αλλάζει το αρχείο στο οποίο δείχνουν. Για να κάνετε chown στον ίδιο τον σύνδεσμο, χρησιμοποιήστε την `lchown(2)` μέσω της `syscall` ή ένα άρθρωμα που την εκθέτει - ο πυρήνας της Perl δεν την εκθέτει.
- **Μορφή filehandle**: σε συστήματα χωρίς `fchown(2)` (σπάνιο στην πράξη σε Linux), το πέρασμα filehandle εγείρει εξαίρεση. Τα barewords στη λίστα αρχείων αντιμετωπίζονται πάντα ως ονόματα αρχείων - χρησιμοποιήστε `\*FH` ή ένα `my $fh` για να εννοήσετε «αυτό το handle».
- **Περιορισμός παραχώρησης**: σε συστήματα POSIX με `_PC_CHOWN_RESTRICTED` ενεργό για τη διαδρομή-στόχο, μόνο ο root μπορεί να αλλάξει τον ιδιοκτήτη γενικά. Δοκιμάστε με:

```
use POSIX qw(pathconf _PC_CHOWN_RESTRICTED);
my $restricted = pathconf($path, _PC_CHOWN_RESTRICTED);
```

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `chmod` - αλλάζει τα bits δικαιωμάτων αρχείου· δέχεται το ίδιο σχήμα «mode, μετά λίστα αρχείων» και επιστρέφει τον ίδιο μετρητή αριθμού αρχείων που άλλαξαν
- `stat` - διαβάστε τον τρέχοντα ιδιοκτήτη, ομάδα και mode ενός αρχείου πριν αποφασίσετε τι θα αλλάξετε
- `umask` - θέτει την προεπιλεγμένη μάσκα δικαιωμάτων για αρχεία που δημιουργούνται από αυτή τη διεργασία· συμπληρώνει την chown για νεοδημιουργημένα αρχεία
- `getpwnam` - μεταφράζει ένα όνομα χρήστη στο αριθμητικό uid που απαιτεί η chown
- `getgrnam` - μεταφράζει ένα όνομα ομάδας στο αριθμητικό gid που απαιτεί η chown
- `$!` - κρατάει το errno της τελευταίας ανά-αρχείο κλήσης `chown(2)` που απέτυχε, όταν ο μετρητής επιστροφής είναι κάτω από το μήκος της λίστας

Βαθμωτά και συμβολοσειρές

chr

Επιστρέφει τον χαρακτήρα του οποίου το σημείο κώδικα είναι ο δοθείς αριθμός.

Η `chr` παίρνει έναν μη αρνητικό ακέραιο και επιστρέφει συμβολοσειρά ενός χαρακτήρα που περιέχει τον χαρακτήρα σε αυτό το σημείο κώδικα του τρέχοντος συνόλου χαρακτήρων. Η `chr(65)` δίνει `"A"` τόσο σε ASCII όσο και σε Unicode· η `chr(0x263a)` δίνει το χαμογελαστό πρόσωπο Unicode ☺. Αν παραλειφθεί το `NUMBER`, η `chr` λειτουργεί στο `$_`. Η αντίστροφη λειτουργία είναι η `ord`.

Σύνοψη

```
chr NUMBER
chr
chr($n)
```

Τι επιστρέφεται

Συμβολοσειρά μήκους ενός χαρακτήρα. Με σημασιολογία Unicode - προεπιλογή για οποιοδήποτε σημείο κώδικα πάνω από το `0x7f` - αυτός ο χαρακτήρας μπορεί να κωδικοποιείται εσωτερικά ως πολλά bytes UTF-8, οπότε η `length` αναφέρει 1 ενώ η `bytes::length` μπορεί να αναφέρει έως τέσσερα. Για σημεία κώδικα στο εύρος `0..127` το αποτέλεσμα είναι μονοβυτική συμβολοσειρά ASCII και τα δύο μήκη συμπίπτουν.

Καθολική κατάσταση που επηρεάζει

Χωρίς όρισμα, η `chr` διαβάζει το `$_`. Δεν γράφει τίποτα. Η pragma `bytes` αλλάζει το τι κάνει η `chr` με είσοδο εκτός εύρους ή αρνητική (δείτε *Οριακές περιπτώσεις*)· καμία άλλη pragma ή ειδική μεταβλητή δεν την επηρεάζει.

Παραδείγματα

Χαρακτήρας ASCII από το σημείο κώδικά του:

```
print chr(65);              # A
print chr(97), "\n";        # a
```

Δόμηση συμβολοσειράς από λίστα σημείων κώδικα με `map` και `join`:

```
my @code = (72, 101, 108, 108, 111);
print join("", map { chr } @code), "\n";   # Hello
```

Σημείο κώδικα Unicode πάνω από το εύρος ASCII - το αποτέλεσμα είναι ένας χαρακτήρας, τέσσερα bytes στην εσωτερική του μορφή UTF-8:

```
use utf8;
my $smiley = chr(0x263a);
print length($smiley);                      # 1
use bytes; print length($smiley);           # 3
```

Πλήρης κύκλος με την `ord` - για κάθε μονό χαρακτήρα `$c`, ισχύει η ταυτότητα `chr(ord($c)) eq $c`:

```
my $c = "Z";
print chr(ord($c)) eq $c ? "yes" : "no";     # yes
```

Μορφή προεπιλεγμένου ορίσματος μέσα σε βρόχο πάνω από το `$_`:

```
for (65, 66, 67) {
    print chr;                               # prints ABC
}
```

Η αρνητική είσοδος αποδίδει τον χαρακτήρα αντικατάστασης Unicode `U+FFFD`:

```
print chr(-1) eq "\x{fffd}" ? "repl" : "?";  # repl
```

Οριακές περιπτώσεις

- **Κλασματικό όρισμα**: αποκόπτεται προς το μηδέν πριν την αναζήτηση. Η `chr(65.9)` επιστρέφει `"A"`.
- **Μη αριθμητική συμβολοσειρά**: μετατρέπεται με τους κανονικούς κανόνες αριθμητικού εξαναγκασμού, με προειδοποίηση υπό `use warnings`. Η `chr("65abc")` επιστρέφει `"A"`.
- **Η `chr(undef)`** επιστρέφει `chr(0)`, τον χαρακτήρα NUL `"\0"`, και προειδοποιεί υπό `use warnings` για χρήση μη αρχικοποιημένης τιμής.
- **Αρνητικές τιμές**: εκτός της `pragma bytes`, η αρνητική είσοδος επιστρέφει τον χαρακτήρα αντικατάστασης Unicode `"\x{fffd}"`. Υπό `use bytes`, χρησιμοποιούνται τα χαμηλά οκτώ bits της ακέραιας τιμής: η `chr(-1)` αποδίδει το byte `"\xff"`.
- **Τιμές πάνω από `0x10ffff`** (το μέγιστο σημείο κώδικα Unicode) εξακολουθούν να επιστρέφουν συμβολοσειρά ενός χαρακτήρα υπό την προεπιλεγμένη σημασιολογία, αλλά εκπέμπουν προειδοποίηση `Unicode non-character is illegal for interchange` για τα εύρη μη-χαρακτήρων, και προειδοποίηση `Unicode surrogate U+...` για το εύρος surrogate `0xd800..0xdfff` όταν η συμβολοσειρά κωδικοποιηθεί αργότερα.
- **Το εύρος 128..255**: οι χαρακτήρες σε αυτό το εύρος εξ ορισμού **δεν** αποθηκεύονται εσωτερικά ως UTF-8, για συμβατότητα με συμβολοσειρές bytes. Συγκρίνονται ίσοι με τη μονοβυτική τους μορφή και με το ίδιο σημείο κώδικα Unicode· η εσωτερική αναπαράσταση έχει σημασία μόνο όταν τους εξετάζετε με `bytes::length` ή όταν αλληλεπιδράτε με στρώματα PerlIO.
- **Η `chr` δεν εγείρει ποτέ `croak`** για οποιαδήποτε αριθμητική είσοδο. Τα σφάλματα αναδύονται μόνο ως προειδοποιήσεις, ποτέ ως εξαιρέσεις.
- **Δεν είναι lvalue**: η `chr` επιστρέφει φρέσκο βαθμωτό. Η `chr(65) = "B"` είναι συντακτικό σφάλμα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `ord` - αντίστροφη της `chr`: σημείο κώδικα του πρώτου χαρακτήρα μιας συμβολοσειράς
- `sprintf` - η `sprintf "%c", $n` είναι ισοδύναμη με την `chr $n`· χρησιμοποιήστε την για να εμβολιάσετε έναν χαρακτήρα σε μια μεγαλύτερη μορφή με μία κλήση
- `pack` - δόμηση πολυχαρακτηρικής συμβολοσειράς από λίστα σημείων κώδικα με μία κλήση (`pack "U*", @code` για Unicode, `pack "C*", @code` για bytes)
- `hex` - αναλύει δεκαεξαδική συμβολοσειρά στον ακέραιο που στη συνέχεια περνάτε στην `chr`· ζευγαρώνει με την `chr` κατά την αποκωδικοποίηση συμβολισμού `U+XXXX`
- `use bytes` - pragma που μεταβάλλει την `chr` σε σημασιολογία προσανατολισμένη σε bytes για τιμές εκτός του `0..0x7f`
- `perlunicode` - υπόβαθρο για τον τρόπο με τον οποίο η Perl αποθηκεύει και συγκρίνει συμβολοσειρές Unicode

Filehandles, αρχεία, κατάλογοι

chroot

Αλλάζει τον ριζικό κατάλογο της τρέχουσας διεργασίας και κάθε παιδιού που αυτή δημιουργεί στη συνέχεια.

Μετά από επιτυχή `chroot`, κάθε όνομα διαδρομής που ξεκινά με `/` αναλύεται σχετικά ως προς τον κατάλογο που υποδεικνύεται από το `FILENAME` αντί για την πραγματική ρίζα του συστήματος αρχείων. Η διεργασία χάνει τη δυνατότητα να ονομάζει αρχεία εκτός αυτού του υποδέντρου μέσω απόλυτων διαδρομών. Ο πυρήνας το επιβάλλει αυτό για την τρέχουσα διεργασία και για κάθε απόγονο - δεν υπάρχει εμβέλεια ανά νήμα ούτε τρόπος αναίρεσης από το εσωτερικό του κελιού.

Το `chroot` **δεν** αλλάζει τον τρέχοντα κατάλογο εργασίας. Αν ο κατάλογος εργασίας ήταν εκτός της νέας ρίζας τη στιγμή που η κλήση πέτυχε, παραμένει εκτός, και οι σχετικές διαδρομές μπορούν ακόμη να ξεφύγουν. Ακολουθήστε πάντοτε μια επιτυχημένη `chroot` με `chdir` στο `/` (που πλέον σημαίνει εντός του κελιού) πριν κάνετε οτιδήποτε άλλο.

Η κλήση περιορίζεται στον υπερχρήστη. Μια ανεξουσιοδότητη διεργασία λαμβάνει `EPERM` και το `$!` τίθεται ανάλογα.

Σύνοψη

```
chroot FILENAME
chroot
```

Τι επιστρέφεται

1 σε επιτυχία, `0` σε αποτυχία. Σε αποτυχία το `$!` κρατά το `errno` από την υποκείμενη κλήση συστήματος `chroot(2)` - συνήθως `EPERM` (όχι root), `ENOENT` (η διαδρομή δεν υπάρχει), ή `ENOTDIR` (η διαδρομή δεν είναι κατάλογος).

Ελέγχετε πάντα την τιμή επιστροφής. Μια σιωπηλή αποτυχία αφήνει τη διεργασία να εκτελείται εκτός του προβλεπόμενου κελιού, κάτι που είναι συνήθως χειρότερο από τη διαδρομή κώδικα που πραγματοποίησε την κλήση εξαρχής.

```
chroot "/var/empty"
    or die "chroot: $!";
chdir "/"
    or die "chdir after chroot: $!";
```

Καθολική κατάσταση που επηρεάζει

- `$_` - όταν καλείται χωρίς όρισμα, το `chroot` χρησιμοποιεί την τιμή του `$_` ως διαδρομή. Μέσα σε βρόχο `while (<>)` αυτό σπάνια είναι αυτό που θέλετε· περάστε τη διαδρομή ρητά.
- `$!` - τίθεται στο υποκείμενο `errno` σε αποτυχία.

Παραδείγματα

Η κανονική ασφαλής ακολουθία - αλλαγή ρίζας, μετά αλλαγή καταλόγου εργασίας στη νέα ρίζα, και μετά αποβολή δικαιωμάτων:

```
chroot "/srv/jail"   or die "chroot: $!";
chdir  "/"           or die "chdir: $!";
$) = $(= 65534;                          # drop to 'nobody' gids
$> = $< = 65534;                         # drop to 'nobody' uids
```

Ανοίξτε αρχεία και φορτώστε κοινόχρηστες βιβλιοθήκες **πριν** από το `chroot`, όχι μετά. Η νέα ρίζα τυπικά δεν έχει ούτε `/etc/resolv.conf` ούτε τις διαδρομές αναζήτησης του δυναμικού συνδέτη:

```
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: $!";

# $log and $cfg are still usable inside the jail.
```

Έλεγχος των τρόπων αποτυχίας ξεχωριστά - η διάκριση μεταξύ «όχι root» και «λανθασμένη διαδρομή» έχει σημασία για τα διαγνωστικά:

```
use Errno qw(EPERM ENOENT ENOTDIR);

unless (chroot $path) {
    die "chroot: need root privileges\n"       if $! == EPERM;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    die "chroot: $path does not exist\n"       if $! == ENOENT;
    die "chroot: $path is not a directory\n"   if $! == ENOTDIR;
    die "chroot: $!";
}
```

Η μορφή με προεπιλεγμένο όρισμα διαβάζει τη διαδρομή από το `$_`:

```
$_ = "/srv/jail";
chroot or die "chroot: $!";
```

Οριακές περιπτώσεις

- **Όχι root**: αποτυγχάνει με EPERM. Δεν υπάρχει χαλάρωση βασισμένη σε δυνατότητες (capabilities) στο τυπικό Linux εκτός αν η διεργασία κατέχει το CAP_SYS_CHROOT.
- **Σχετική διαδρομή**: το `chroot "jail"` αναλύεται ως προς τον τρέχοντα κατάλογο εργασίας τη στιγμή της κλήσης. Προτιμήστε απόλυτες διαδρομές ώστε η συμπεριφορά να μην εξαρτάται από το cwd του καλούντος.
- **Symlinks στο `FILENAME`**: αναλύονται πριν τεθεί σε ισχύ η chroot. Symlinks *εντός* της νέας ρίζας που δείχνουν σε απόλυτες διαδρομές θα αναλυθούν σχετικά ως προς τη νέα ρίζα, όχι την πραγματική ρίζα - αυτή είναι η προβλεπόμενη συμπεριφορά εγκλεισμού, όχι σφάλμα.
- **Οι ανοιχτοί περιγραφείς αρχείων επιζούν**: κάθε handle που ανοίχθηκε πριν την κλήση παραμένει χρήσιμο μετά. Έτσι οι καλοδιατεθειμένοι δαίμονες διατηρούν αρχεία καταγραφής, διαμόρφωση και υποδοχές που ακούν κατά τη μετάβαση.
- **Καμία διαφυγή από μέσα**: μια διεργασία δεν μπορεί να «βγει» κάνοντας `chroot`. Ο μόνος τρόπος εξόδου από ένα chroot είναι να την συλλέξει και να την αντικαταστήσει μια προνομιακή διεργασία *εκτός* αυτού.
- **Κατάλογος εργασίας εκτός της νέας ρίζας**: ο πυρήνας επιτρέπει την κλήση, αλλά οι σχετικές διαδρομές και η διάσχιση `..` από εκεί μπορούν ακόμη να φτάσουν σε αρχεία εκτός του κελιού. Το υποχρεωτικό `chdir "/"` μετά το `chroot` κλείνει αυτό το κενό.
- **Threads**: το `chroot` στο Linux είναι ανά διεργασία (αυστηρά, ανά fs struct). Υπό `use threads` όλα τα νήματα διερμηνέα μοιράζονται την ίδια ρίζα μετά την κλήση - δεν υπάρχει κελί ανά νήμα.
- **Δεν επηρεάζει τις `$0`, `%ENV`, ούτε τις ανοιχτές υποδοχές**. Αλλάζει μόνο η ανάλυση διαδρομών.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `chdir` - η υποχρεωτική συνέχεια μετά από επιτυχή `chroot`· χωρίς αυτό ο κατάλογος εργασίας μπορεί να δείχνει ακόμη εκτός της νέας ρίζας
- `fork` - τα παιδιά κληρονομούν το chroot· συνδυάστε με `chroot` για να εκτελέσετε κώδικα χωρίς εμπιστοσύνη σε περιορισμένο υποδέντρο
- `exec` - εκτελεί το πρόγραμμα αντικατάστασης εντός της τρέχουσας ρίζας, οπότε το εκτελέσιμο-στόχος και οι δυναμικές του βιβλιοθήκες πρέπει να υπάρχουν εντός του κελιού
- `open` - το άνοιγμα αρχείων με απόλυτη διαδρομή μετά από `chroot` αναζητά ονόματα εντός του κελιού· ανοίξτε εκ των προτέρων ό,τι χρειάζεστε από έξω πριν την κλήση
- `$_` - η προεπιλεγμένη διαδρομή όταν το `chroot` καλείται χωρίς όρισμα
- `$!` - κρατά το `errno` από το `chroot(2)` σε αποτυχία

Εμβέλεια · Κλάσεις και αντικειμενοστρέφεια

class

Δηλώνει έναν χώρο ονομάτων που συμπεριφέρεται ως εγγενής κλάση αντικειμένων.

Η `class` εισάγει ένα νέο πακέτο όπως ακριβώς και η `package`, αλλά το χαρακτηρίζει ως *κλάση*: ο runtime προσαρτά έναν κατασκευαστή με το όνομα `new`, ενεργοποιεί τις δεσμευμένες λέξεις `field` και `method` μέσα στο σώμα της, και υποστηρίζει απλή κληρονομικότητα μέσω του χαρακτηριστικού `:isa`. Είναι η απάντηση του πυρήνα της γλώσσας στον χειροποίητο αντικειμενοστρεφή προγραμματισμό βασισμένο στο `bless` - το ίδιο πρόβλημα, χωρίς το επαναλαμβανόμενο πλαίσιο κώδικα.

Σημείωση

Experimental The `class` feature (codename *Corinna*) is experimental in Perl 5.44 and may still change in incompatible ways. Enable it with `use feature 'class'` and silence the notice with `no warnings 'experimental::class'`.

Σύνοψη

```
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
```

Ελάχιστη λειτουργική κλάση:

```
use v5.38;
use feature 'class';
no warnings 'experimental::class';
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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
```

Τι επιστρέφεται

Η `class` είναι δήλωση, όχι έκφραση - δεν επιστρέφει τιμή που μπορείτε να αναθέσετε. Αυτό που λαμβάνετε είναι:

- Ένα νέο πακέτο στον πίνακα συμβόλων, προσπελάσιμο όπως οποιοδήποτε άλλο πακέτο.
- Έναν παραγόμενο κατασκευαστή new. **Μην γράφετε δικό σας new.** Ο παραγόμενος τιμά τα χαρακτηριστικά πεδίων `:param` και εκτελεί κάθε μπλοκ `ADJUST` με τη σειρά δήλωσης.
- Το `$self` δεσμεύεται έμμεσα μέσα σε κάθε σώμα `method`.
- Το `__CLASS__` είναι χρησιμοποιήσιμο μέσα σε αρχικοποιητές πεδίων και μεθόδους για να ονομάσει την κλάση που κατασκευάζεται τη δεδομένη στιγμή (η οποία μπορεί να είναι υποκλάση).

Μορφές

Η `class` δέχεται τέσσερις συντακτικές μορφές. Ο διαχωρισμός μεταξύ των μορφών **μπλοκ** και **δήλωσης** αντικατοπτρίζει ακριβώς την `package`:

- **`class NAME BLOCK`** - ο ορισμός της κλάσης είναι το περιεχόμενο του μπλοκ. Η εμβέλεια τερματίζει με το άγκιστρο κλεισίματος.
- **`class NAME;`** - η υπόλοιπη τρέχουσα εμβέλεια (ή αρχείο) μέχρι την επόμενη δήλωση `class` ή `package` αποτελεί το σώμα της κλάσης. Χρήσιμο για αρχεία μίας κλάσης.
- **`class NAME VERSION ...`** - η έκδοση δηλώνεται όπως στο `package Foo 1.234`. Η έκδοση πρέπει να προηγείται οποιωνδήποτε χαρακτηριστικών.
- **`class NAME :ATTRIBUTES ...`** - ένα ή περισσότερα χαρακτηριστικά κλάσης, χωρισμένα με λευκούς χαρακτήρες, καθένα ξεκινώντας με `:`. Το μόνο χαρακτηριστικό κλάσης που ορίζεται προς το παρόν είναι το `:isa`.

```
class My::Thing 1.02 :isa(My::Base) {
    # body
}
```

Πεδία, μέθοδοι, ADJUST

Μέσα στο σώμα μιας κλάσης τρεις νέες δεσμευμένες λέξεις βρίσκονται εντός εμβέλειας:

- Η `field` δηλώνει αποθηκευτικό χώρο ανά στιγμιότυπο. Τα πεδία συμπεριφέρονται σαν λεξιλογικές μεταβλητές μέσα στις μεθόδους, αλλά κάθε στιγμιότυπο έχει τη δική του θέση. Τα χαρακτηριστικά `:param`, `:reader` και `:writer` παράγουν σύνδεση κατασκευαστή και προσπελαστές.
- Η `method` δηλώνει μια υπορουτίνα που δεσμεύει αυτόματα το `$self` και που μπορεί να διαβάζει και να γράφει τα περικλείοντα πεδία με το όνομά τους. Οι μέθοδοι λειτουργούν σαν να ίσχυε το `use feature 'signatures'`· το `$self` δεν εμφανίζεται στην υπογραφή.
- Το `ADJUST BLOCK` εκτελείται μετά τους αρχικοποιητές πεδίων, κατά την κατασκευή, με το `$self` ήδη δεσμευμένο. Χρησιμοποιήστε το για επικύρωση μετά την κατασκευή και για παράγωγα πεδία.

```
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 }
}
```

Τα πεδία είναι **πάντα λεξιλογικά ως προς την κλάση** - δεν κληρονομούνται. Οι υποκλάσεις που χρειάζονται πρόσβαση στην κατάσταση μιας γονικής κλάσης την αποκτούν μέσω μεθόδων, όχι μέσω πεδίων.

Κληρονομικότητα: `:isa`

Μια κλάση κληρονομεί από το πολύ μία άλλη κλάση:

```
class Shape { ... }

class Circle :isa(Shape) {
    field $r :param;
    method area { 3.14159265 * $r * $r }
}
```

Υποστηρίζεται μόνο απλή κληρονομικότητα. Ο στόχος του `:isa` φορτώνεται αυτόματα αν δεν βρίσκεται ήδη στη μνήμη - ισοδύναμο με `use MODULE ()` πριν από τη δήλωση της κλάσης. Μπορεί να απαιτηθεί ελάχιστη έκδοση με τον ίδιο τρόπο όπως το `use MODULE VERSION`:

```
class Circle :isa(Shape 2.000) { ... }
```

Το `:isa` κληρονομεί μόνο **μεθόδους**. Τα πεδία είναι ιδιωτικός αποθηκευτικός χώρος ανά κλάση· μια υποκλάση δεν έχει έμμεση όψη στα πεδία της γονικής κλάσης.

Καθολική κατάσταση που επηρεάζει

- **%INC** - το `:isa(Parent)` φορτώνει το `Parent` μέσω `require` κατά την πρώτη χρήση και το καταγράφει εδώ, ακριβώς όπως το `use`.
- **@ISA** - συμπληρώνεται για την κλάση από το χαρακτηριστικό `:isa`, ώστε η υπάρχουσα ενδοσκόπηση `isa` / `UNIVERSAL` / `SUPER::` να εξακολουθεί να λειτουργεί.
- **Ο πίνακας συμβόλων** - το όνομα της κλάσης γίνεται stash στο `%::` ακριβώς όπως θα δημιουργούσε η `package`.

Παραδείγματα

Απλή κλάση με μια υποχρεωτική παράμετρο και ένα παράγωγο πεδίο:

```
class User {
    field $name :param;
    field $greeting;

    ADJUST { $greeting = "Hello, $name" }

    method greet { say $greeting }
}

User->new(name => 'Ada')->greet;          # Hello, Ada
```

Προεπιλεγμένες τιμές παραμέτρων - το `=` εφαρμόζεται όταν ο καλών παρέλειψε την παράμετρο· το `//=` εφαρμόζεται επίσης όταν η τιμή ήταν `undef`· το `||=` εφαρμόζεται επίσης όταν η τιμή ήταν ψευδής:

```
class Window {
    field $title :param          = 'untitled';
    field $width :param          //= 640;
    field $shown :param          ||= 1;
}
```

Απλή κληρονομικότητα με υπερκάλυψη μεθόδου:

```
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
```

Ιδιωτική μέθοδος μέσω λεξιλογικού `my method`, η οποία καλείται μέσω `->&`:

```
class Safe {
    field $secret :param;

    my method check ($n) { $n == $secret }

    method unlock ($n) {
        return $self->&check($n) ? "open" : "denied";
    }
}
```

Χρήση του __CLASS__ ώστε ένας αρχικοποιητής πεδίου της βασικής κλάσης να εντοπίζει την υπερκάλυψη της υποκλάσης:

```
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
```

Μορφή δήλωσης για το υπόλοιπο του αρχείου:

```
use v5.38;
use feature 'class';
class Config;

field $path :param;
method path { $path }
# File ends here; whole remainder is Config's body.
```

Οριακές περιπτώσεις

- **Μην ορίζετε new**. Ο παραγόμενος κατασκευαστής είναι το μόνο υποστηριζόμενο σημείο εισόδου. Ένα `sub new { ... }` γραμμένο από τον χρήστη μέσα σε σώμα `class` αποτελεί σφάλμα κατά τη μεταγλώττιση.
- **Καμία πολλαπλή κληρονομικότητα**. Το `:isa(A, B)` είναι συντακτικό σφάλμα. Πολλαπλά χαρακτηριστικά `:isa` σε μία κλάση επίσης απορρίπτονται.
- **Καμία τροποποίηση του @ISA**. Αντιμετωπίστε το `@ISA` ως μόνο για ανάγνωση για κλάσεις που δηλώνονται με `class`. Η προσθήκη σε αυτό κατά τον χρόνο εκτέλεσης είναι απροσδιόριστη.
- **Η package δεν μπορεί να ξανανοίξει μια κλάση**. Μόλις ένας χώρος ονομάτων δηλωθεί με `class`, οι επόμενες δηλώσεις `package` που στοχεύουν το ίδιο όνομα αποτελούν σφάλμα κατά τη μεταγλώττιση, και αντιστρόφως.

- **Τα πεδία δεν είναι λεξιλογικές μεταβλητές** με τη συνηθισμένη έννοια. Δεν μπορείτε να αναφερθείτε σε ένα πεδίο από έξω από το σώμα της κλάσης, και δεν μπορείτε να εφαρμόσετε `our` ή `local` σε αυτό. Μόνο μέθοδοι και μπλοκ `ADJUST` βλέπουν τα πεδία.
- **Το `$self` είναι μόνο για ανάγνωση** μέσα σε μια μέθοδο. Η ανάθεση σε αυτό αποτελεί σφάλμα κατά τον χρόνο εκτέλεσης.
- **Οι αρχικοποιητές πεδίων εκτελούνται πριν υπάρξει το `$self`**. Μέσα σε `field $x = EXPR`, μπορείτε να χρησιμοποιήσετε `__CLASS__` και προηγουμένως δηλωμένα πεδία, αλλά όχι το `$self`. Οτιδήποτε χρειάζεται το πλήρως κατασκευασμένο στιγμιότυπο ανήκει σε μπλοκ `ADJUST`.
- **Καμία πρόσβαση σε πεδίο πριν τη δήλωση**. Μέσα στην έκφραση αρχικοποιητή ενός πεδίου, ορατά είναι μόνο τα πεδία που έχουν δηλωθεί *νωρίτερα* στο σώμα της κλάσης.
- **Επίλυση μεθόδων**. Η αποστολή εξακολουθεί να γίνεται μέσω της συνήθους MRO (`UNIVERSAL::isa`, `SUPER::`), οπότε το `$self->SUPER::foo(@args)` λειτουργεί όπως και για κλάσεις βασισμένες στο `bless`.
- **Εμβέλεια μορφής δήλωσης**. Η `class Foo;` καταναλώνει το υπόλοιπο του περικλείοντος μπλοκ - συνήθως το υπόλοιπο του αρχείου. Μια δεύτερη `class` ή `package` την τερματίζει· δεν υπάρχει τρόπος να «κλείσει» πρόωρα μια κλάση μορφής δήλωσης.
- **Πειραματική προειδοποίηση**. Κάθε δήλωση `class` εκπέμπει μια προειδοποίηση `class is experimental` υπό τις προεπιλεγμένες προειδοποιήσεις. Χρησιμοποιήστε `no warnings 'experimental::class'` μόλις το χαρακτηριστικό μπει σε πραγματική χρήση.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44. The feature is flagged experimental in upstream; pperl mirrors that status and emits the same warning category (`experimental::class`).

Δείτε επίσης

- `field` - δηλώνει αποθηκευτικό χώρο ανά στιγμιότυπο μέσα σε σώμα `class`· υποστηρίζει τα `:param`, `:reader`, `:writer`
- `method` - δηλώνει μια υπορουτίνα με έμμεσο `$self` και άμεση πρόσβαση στα πεδία της περικλείουσας κλάσης
- `package` - η δήλωση χώρου ονομάτων χωρίς κλάση· η `class` είναι το αντικειμενοστρεφές αδελφικό αντίστοιχο με τις ίδιες συντακτικές μορφές
- `bless` - ο πριν την 5.38 τρόπος σύνδεσης μιας κλάσης σε μια αναφορά· εξακολουθεί να υποστηρίζεται και διαλειτουργεί με αντικείμενα κατασκευασμένα με `class` για ελέγχους `isa`/`can`
- `isa` - τελεστής για έλεγχο μέλους κλάσης· λειτουργεί τόσο για αντικείμενα δηλωμένα με `class` όσο και για βασισμένα στο `bless`

- `use` - πώς το `:isa(Parent)` φορτώνει τη γονική κλάση από κάτω από τα παρασκήνια

I/O

close

Κλείνει ένα filehandle, εκκενώνει τους ενταμιευτές του και απελευθερώνει τον υποκείμενο περιγραφέα αρχείου.

Η `close` ολοκληρώνει όποιο I/O εκκρεμούσε στο `FILEHANDLE`, αποδομεί τη στοίβα στρωμάτων PerlIO και ζητά από τον πυρήνα να κλείσει τον περιγραφέα. Η τιμή επιστροφής διακρίνει το καθαρό κλείσιμο από μια περίπτωση όπου είτε μια ενταμιευμένη εγγραφή απέτυχε κατά την έξοδο, είτε ένα στρώμα PerlIO ανέφερε σφάλμα, είτε - για σωλήνες - η διεργασία-παιδί τερμάτισε με μη μηδενικό κωδικό. Χωρίς όρισμα, η `close` λειτουργεί στο τρέχον επιλεγμένο filehandle (βλ. `select`).

Σύνοψη

```
close FILEHANDLE
close EXPR
close
```

Τι επιστρέφεται

Αληθής τιμή (1) σε καθαρό κλείσιμο· ψευδής τιμή διαφορετικά. Η ψευδής επιστροφή σημαίνει ότι κάτι από τα παρακάτω πήγε στραβά:

- Μια εκκρεμής ενταμιευμένη εγγραφή δεν μπόρεσε να εκκενωθεί (πλήρης δίσκος, σπασμένος σωλήνας, κλειστή υποδοχή).
- Ένα στρώμα PerlIO (π.χ. `:encoding(...)`, `:gzip` από στρώμα του CPAN) ανέφερε σφάλμα.
- Είχε ήδη καταγραφεί προϋπάρχον επίμονο σφάλμα στο handle.
- Το handle προήλθε από `open` μέσω σωλήνα και το παιδί τερμάτισε με μη μηδενικό κωδικό (ή απέτυχε άλλη κλήση συστήματος εμπλεκόμενη στην αποδόμηση του σωλήνα).

Όταν η `close` επιστρέψει ψευδή τιμή, η `$!` κρατά το σφάλμα της αποτυχημένης λειτουργίας και διαβάζεται με ασφάλεια. Για handles σωλήνα, αν το μόνο πρόβλημα ήταν μη μηδενικός τερματισμός του παιδιού, η `$!` τίθεται σε `0` και η κατάσταση του παιδιού εμφανίζεται στην `$?` (βλ. παρακάτω).

Ελέγχετε πάντα την τιμή επιστροφής σε εγγράψιμα handles: το τελικό ενταμιευμένο μπλοκ εκκενώνεται μέσα στην `close`, οπότε η `print` που επέστρεψε αληθή τιμή νωρίτερα **δεν** εγγυάται ότι τα δεδομένα έφτασαν στον δίσκο.

```
close $fh
    or die "close $path failed: $!";
```

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται στο errno της αποτυχίας όταν η `close` επιστρέφει ψευδή τιμή, συμπεριλαμβανομένου του σφάλματος εκκένωσης κατά την τελική ενταμιευμένη εγγραφή. Καθαρίζεται σε `0` σε κλείσιμο σωλήνα όπου το μόνο πρόβλημα ήταν ο μη μηδενικός τερματισμός του παιδιού.
- `$?` - για handles από `open` μέσω σωλήνα, τίθεται στην κατάσταση αναμονής της διεργασίας-παιδιού: τα 8 κατώτερα bits είναι το σήμα, τα 8 ανώτερα bits είναι ο κωδικός εξόδου (ίδια κωδικοποίηση με την `system`).
- `${^CHILD_ERROR_NATIVE}` - για handles σωλήνα, τίθεται στην ωμή OS-native τιμή κατάστασης (πριν τη φορητή κωδικοποίηση της Perl).
- `$.` - μια **ρητή** `close` σε handle εισόδου μηδενίζει τον τρέχοντα αριθμό γραμμής. Ένα έμμεσο κλείσιμο μέσω επανανοίγματος του ίδιου handle όχι.
- Τρέχον επιλεγμένο filehandle (μέσω `select`) - η `close` χωρίς όρισμα το κλείνει.

Παραδείγματα

Κλείσιμο κανονικού αρχείου και έλεγχος για αποτυχία εκκένωσης:

```
open my $fh, ">", "out.txt" or die $!;
print {$fh} @records;
close $fh or die "flush to out.txt failed: $!";
```

Κλείσιμο σωλήνα ανοιγμένου για εγγραφή και επιθεώρηση της κατάστασης εξόδου του παιδιού. Η `close` περιμένει το παιδί και κατόπιν συμπληρώνει την `$?`:

```
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);
```

Κλείσιμο σωλήνα ανοιγμένου για ανάγνωση και κατόπιν συλλογή της κατάστασης εξόδου της εντολής. Ο κωδικός εξόδου είναι τα 8 ανώτερα bits της `$?`:

```
open(my $who, "-|", "who") or die $!;
my @lines = <$who>;
close $who;
my $exit = $? >> 8;
```

Η γυμνή `close` κλείνει όποιο handle έκανε πιο πρόσφατα τρέχον η `select`, και κατόπιν επαναφέρει το STDOUT ως το επιλεγμένο handle:

```
open my $log, ">>", "app.log" or die $!;
my $old = select $log;
print "entry\n";
close;                          # closes $log (the selected handle)
select $old;
```

Το πρόωρο κλείσιμο του άκρου ανάγνωσης ενός σωλήνα κάνει τον εγγραφέα να λάβει SIGPIPE. Αν ο εγγραφέας δεν έχει χειριστή, θα τερματιστεί - αποστραγγίστε πρώτα τον σωλήνα αν χρειάζεστε ο εγγραφέας να εξέλθει καθαρά:

```
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: $!";
```

Οριακές περιπτώσεις

- **Γυμνή `close`** - χωρίς όρισμα, κλείνει το τρέχον επιλεγμένο filehandle. Σε ένα μόλις ξεκινημένο πρόγραμμα αυτό είναι το STDOUT· μετά από `select($fh)` είναι το `$fh`. Σπάνια αυτό που θέλετε εκτός κώδικα με στενή εμβέλεια.
- **Το επανάνοιγμα κλείνει για εσάς** - η `open my $fh, ...` σε handle που είναι ήδη ανοιχτό θα κλείσει πρώτα τον παλιό περιγραφέα. Το έμμεσο κλείσιμο **δεν** μηδενίζει την `$.`· η ρητή `close` το κάνει. Χρησιμοποιήστε ρητή `close` όταν η κατάσταση του αριθμού γραμμής έχει σημασία.
- **Η έξοδος από την εμβέλεια κλείνει τα λεξιλογικά handles** - όταν η τελευταία αναφορά σε ένα λεξιλογικό filehandle βγει εκτός εμβέλειας, η Perl το κλείνει. Αν αυτό το έμμεσο κλείσιμο αποτύχει (σφάλμα εκκένωσης, μη μηδενική έξοδος του παιδιού), η Perl εκπέμπει προειδοποίηση `Warning: unable to close filehandle %s properly`. Καλέστε ρητά την `close` σε σημείο όπου μπορείτε να ελέγξετε την τιμή επιστροφής και να σιγάσετε αυτή την προειδοποίηση.
- **Έμμεσα handles** - το `FILEHANDLE` μπορεί να είναι οποιαδήποτε έκφραση που παράγει χρησιμοποιήσιμο filehandle: ένα bareword (`close STDERR`), ένα βαθμωτό (`close $fh`), μια αναφορά glob (`close \*FH`) ή ένα αυτο-δημιουργημένο handle από `open my $fh, ...`.
- **Το κλείσιμο σωλήνα περιμένει** - το κλείσιμο handle σωλήνα μπλοκάρει μέχρι να εξέλθει το παιδί, οπότε η `$?` έχει συμπληρωθεί όταν επιστρέφει η `close`. Υπό threads, αν το ίδιο handle σωλήνα παραμένει ανοιχτό σε άλλο thread, η `close` επιστρέφει αληθή τιμή αμέσως χωρίς να συλλέξει το παιδί.
- **Κλείσιμο σωλήνα με μη μηδενικό τερματισμό** - η `close` επιστρέφει ψευδή τιμή και θέτει την `$!` σε `0`, ώστε η έκφραση `$! ? "$!" : "exit $?"` να διακρίνει αποτυχία σε επίπεδο OS από κανονικό μη μηδενικό τερματισμό.
- **Κλειστό ή μη έγκυρο handle** - το κλείσιμο handle που δεν είναι ανοιχτό θέτει την `$!` σε `"Bad file descriptor"` και επιστρέφει ψευδή τιμή. Υπό `use warnings`, εκπέμπεται προειδοποίηση `close() on unopened filehandle`.
- **STDIN / STDOUT / STDERR** - μπορούν να κλείσουν όπως οποιοδήποτε άλλο handle. Ο περιγραφέας απελευθερώνεται και θα ξαναχρησιμοποιηθεί από την επόμενη open στο πρότυπο handle, στο οποίο βασίζεται η κλασική ανακατεύθυνση `open STDOUT, ">", "log"`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `open` - το αντίστροφο· μια `open` σε ήδη ανοιχτό handle εκτελεί πρώτα έμμεσο `close` (αλλά δεν μηδενίζει την `$.`)
- `print` - η τελική ενταμιευμένη `print` εκκενώνεται μέσα στην `close`, οπότε η πραγματική επιτυχία της εγγραφής γίνεται γνωστή μόνο μετά την `close`
- `fileno` - αριθμός περιγραφέα ενός handle· γίνεται μη έγκυρος μόλις πετύχει η `close`
- `binmode` - η στοίβα στρωμάτων που έχει οριστεί από την `binmode` αποδομείται κατά τη διάρκεια της `close`, και η ίδια η αποδόμηση μπορεί να αποτύχει
- `$?` - κατάσταση εξόδου του παιδιού που συμπληρώνεται από την `close` σε handle σωλήνα
- `$!` - σφάλμα από την αποτυχημένη λειτουργία κλεισίματος

I/O

closedir

Κλείνει ένα handle καταλόγου που άνοιξε η `opendir`.

Η `closedir` αποδεσμεύει τη ροή καταλόγου του λειτουργικού συστήματος που συνδέεται με το `DIRHANDLE` και ελευθερώνει τη θέση ώστε το handle να μπορεί να επαναχρησιμοποιηθεί. Είναι το αντίστοιχο της `opendir`, και ο μόνος υποστηριζόμενος τρόπος κλεισίματος ενός handle καταλόγου - η `close` λειτουργεί σε filehandles και **δεν** δέχεται handle καταλόγου, επειδή τα filehandles και τα handles καταλόγων ζουν σε χωριστούς χώρους ονομάτων.

Σύνοψη

```
closedir DIRHANDLE
closedir $dh
```

Τι επιστρέφεται

1 σε επιτυχία, `undef` σε αποτυχία (με το `$!` ρυθμισμένο στο σφάλμα από την υποκείμενη κλήση `closedir(3)`). Αξίζει να ελέγχεται η τιμή επιστροφής σε δικτυακά ή FUSE-υποστηριζόμενα συστήματα αρχείων όπου το κλείσιμο ενδέχεται να αναδείξει ένα ετεροχρονισμένο σφάλμα I/O.

```
closedir $dh
    or warn "closedir $path failed: $!";
```

Το κλείσιμο ενός handle καταλόγου που είναι ήδη κλειστό, ή ενός που δεν άνοιξε ποτέ επιτυχώς, επιστρέφει `undef` και - υπό `use warnings` - εκπέμπει μια προειδοποίηση `closedir() attempted on invalid dirhandle`.

Λεξιλογικά handles καταλόγων και αυτόματο κλείσιμο

Ένα handle καταλόγου που αποθηκεύεται σε λεξιλογικό βαθμωτό - η μορφή που παράγει το `opendir my $dh, $path` - κλείνει αυτόματα όταν το βαθμωτό βγει από την εμβέλεια (ή ανατεθεί ξανά, ή τερματιστεί το πρόγραμμα). Η ρητή `closedir` τότε είναι ζήτημα χρονισμού, όχι ορθότητας:

```
sub entries {
    my ($path) = @_;
    opendir my $dh, $path or die "opendir $path: $!";
    return readdir $dh;
    # $dh is closed here as it leaves scope
}
```

Καλέστε ρητά την `closedir` όταν χρειάζεται η αποδέσμευση να συμβεί σε γνωστό σημείο - πριν από fork, πριν τη μετονομασία του καταλόγου σε πλατφόρμες που τον κρατούν ανοιχτό, πριν τον έλεγχο της τιμής επιστροφής, ή απλώς όταν η περικλείουσα εμβέλεια είναι μεγάλη και κρατά πολλούς άλλους πόρους.

Τα bareword handles καταλόγων (καθολικά πακέτου, π.χ. `opendir DH, $path`) **δεν** κλείνουν αυτόματα. Διατηρούνται μέχρι να τερματιστεί το πρόγραμμα ή να κληθεί η `closedir DH`.

Καθολική κατάσταση που επηρεάζει

Η `closedir` δεν διαβάζει ούτε γράφει σε κάποια από τις τεκμηριωμένες ειδικές μεταβλητές της Perl. Θέτει το `$!` σε αποτυχία. **Δεν** επηρεάζει το `$_`, το επιλεγμένο filehandle, ούτε καμία από τις καθολικές μεταβλητές διαχωριστών εξόδου.

Παραδείγματα

Βασικός κύκλος ανοίγματος/ανάγνωσης/κλεισίματος:

```
opendir my $dh, "." or die "opendir: $!";
my @names = readdir $dh;
closedir $dh;
```

Έλεγχος της τιμής επιστροφής - χρήσιμο σε δικτυακά συστήματα αρχείων:

```
opendir my $dh, "/mnt/nfs/project" or die "opendir: $!";
my @entries = readdir $dh;
closedir $dh
    or die "closedir failed: $!";
```

Επαναφορά στην αρχή με `rewinddir` αντί για `closedir` + `opendir` όταν θέλετε να ξαναδιαβάσετε τον ίδιο κατάλογο:

```
opendir my $dh, $path or die $!;
my @first = readdir $dh;
rewinddir $dh;
my @again = readdir $dh;              # same set as @first
closedir $dh;
```

Bareword handle - πρέπει να κλείσει ρητά:

```
opendir DH, "/etc" or die $!;
while (my $name = readdir DH) {
    last if $name eq "hosts";
}
closedir DH;                            # not optional for barewords
```

Η χρήση της `close` σε handle καταλόγου **δεν** δουλεύει:

```
opendir my $dh, "." or die $!;
close $dh;                              # wrong - does not close the
 ↪dir
closedir $dh;                           # correct
```

Οριακές περιπτώσεις

- **`close` σε handle καταλόγου**: τα filehandles και τα handles καταλόγων είναι διακριτοί τύποι. Η `close` σε handle καταλόγου είναι είτε no-op είτε σφάλμα ανάλογα με τη μορφή του handle· χρησιμοποιείτε πάντα την `closedir` για να κλείσετε ό,τι άνοιξε η `opendir`.
- **Διπλό κλείσιμο**: η κλήση της `closedir` δύο φορές στο ίδιο handle επιστρέφει `undef` στη δεύτερη κλήση και θέτει το `$!`. Υπό `use warnings` εκπέμπεται προειδοποίηση `closedir() attempted on invalid dirhandle`.
- **Κλεισμένο ή ποτέ-ανοιχτό handle**: ίδια συμπεριφορά με το διπλό κλείσιμο - επιστροφή `undef` και προειδοποίηση υπό `use warnings`.
- **Εκκρεμής κατάσταση `readdir`**: η `closedir` απορρίπτει οποιεσδήποτε αδιάβαστες εγγραφές. Αν θέλετε να επανεκκινήσετε την επανάληψη, χρησιμοποιήστε `rewinddir` αντί για close + reopen.
- **Θέσεις `seekdir` / `telldir`**: οι θέσεις που λαμβάνονται από την `telldir` ακυρώνονται από την `closedir`. Έχουν νόημα μόνο για τη διάρκεια ζωής μίας ανοιχτής ροής καταλόγου.
- **Έξοδος από εμβέλεια κατά τη διάρκεια εξαίρεσης**: τα λεξιλογικά handles καταλόγων εξακολουθούν να κλείνουν όταν η εμβέλεια ξετυλίγεται μέσω `die`, οπότε η ρητή `closedir` σε κώδικα καθαρισμού είναι συνήθως περιττή.
- **Fork**: μετά από `fork`, γονέας και παιδί μοιράζονται την κατάσταση της ροής καταλόγου στον πυρήνα· καθένας θα πρέπει να καλέσει `closedir` στη διεργασία που δεν τη χρειάζεται πλέον.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `opendir` - άνοιγμα handle καταλόγου· η `closedir` είναι το αντίστοιχό της
- `readdir` - διαβάζει την επόμενη εγγραφή (ή όλες τις εγγραφές σε περιβάλλον λίστας) από ένα ανοιχτό handle καταλόγου
- `rewinddir` - επαναφέρει τη ροή καταλόγου στην αρχή χωρίς να την κλείσει
- `seekdir` - επανατοποθετεί τη ροή καταλόγου σε τιμή που επέστρεψε προηγουμένως η `telldir`
- `telldir` - επιστρέφει την τρέχουσα θέση εντός μιας ανοιχτής ροής καταλόγου
- `close` - κλείνει ένα **file** handle· δεν λειτουργεί σε handles καταλόγων

Υποδοχές (sockets)

connect

Ξεκινά μια σύνδεση από μια υποδοχή προς μια απομακρυσμένη διεύθυνση.

Η `connect` είναι το μισό από την πλευρά του πελάτη στην χειραψία της υποδοχής. Δοθέντος ενός `SOCKET` που δημιουργήθηκε με `socket` και μιας πακεταρισμένης διεύθυνσης `NAME` στη μορφή που αναμένει η οικογένεια διευθύνσεων της υποδοχής, ζητάει από τον πυρήνα να εγκαθιδρύσει σύνδεση με τον συγκεκριμένο ομότιμο. Είναι λεπτό περιτύλιγμα γύρω από την κλήση συστήματος `connect(2)` και κληρονομεί κάθε σημασιολογία μπλοκαρίσματος, μη μπλοκαρίσματος και σφάλματος αυτής της κλήσης.

Σύνοψη

```
connect SOCKET, NAME
```

Τι επιστρέφεται

Αληθές (1) σε επιτυχία, ψευδές σε αποτυχία με την `$!` τεθειμένη στο σφάλμα του συστήματος. Ελέγχετε πάντα την τιμή επιστροφής - δεν υπάρχει χρήσιμη προεπιλεγμένη συμπεριφορά όταν αποτυγχάνει μια απόπειρα σύνδεσης:

```
connect($sock, $peer)
    or die "connect to $host:$port failed: $!";
```

Καθολική κατάσταση που επηρεάζει

- Η `$!` τίθεται στο `errno` του πυρήνα σε αποτυχία. Ερμηνεύστε την έναντι των σταθερών της `Errno`, όχι μέσω της τιμής συμβολοσειράς - το μηνύμα μετά τη μετατροπή σε συμβολοσειρά εξαρτάται από το locale.
- Το ίδιο το handle της υποδοχής φέρει την κατάσταση της σύνδεσης· καμία άλλη ειδική μεταβλητή δεν εμπλέκεται.

Δόμηση του NAME: είναι ένα πακεταρισμένο struct, όχι συμβολοσειρά

Το δεύτερο όρισμα **δεν** είναι συμβολοσειρά `"host:port"`. Είναι ένα πακεταρισμένο sockaddr στην ακριβή διάταξη bytes που αναμένει ο πυρήνας για την οικογένεια διευθύνσεων της υποδοχής. Δομήστε το με τους βοηθούς από την `Socket`:

- `sockaddr_in($port, $inaddr)` για IPv4 - η `$inaddr` από την `inet_aton` ή την `inet_pton`.
- `sockaddr_in6($port, $in6addr)` για IPv6 - η `$in6addr` από την `inet_pton(AF_INET6, ...)`.
- `sockaddr_un($path)` για υποδοχές τομέα Unix.

Το χειροκίνητο πακετάρισμα με την `pack` λειτουργεί, αλλά δεν υπάρχει λόγος να το κάνετε όταν η `Socket` εξάγει ήδη τους κατάλληλους κατασκευαστές.

Παραδείγματα

Ένας απλός πελάτης TCP πάνω σε IPv4:

```
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";
```

Σύγχρονος πελάτης ανεξαρτήτως οικογένειας διευθύνσεων μέσω `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});
    close $sock;
    undef $sock;
}
defined $sock or die "could not connect to any address";
```

Ένας πελάτης τομέα Unix:

```
use Socket;

socket(my $sock, PF_UNIX, SOCK_STREAM, 0) or die "socket: $!";
connect($sock, sockaddr_un("/tmp/my.sock"))
    or die "connect: $!";
```

Connect χωρίς μπλοκάρισμα με timeout μέσω `select` - ο ιδιωματικός τρόπος να βάλετε φραγή σε μια απόπειρα connect σε καθαρή 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 $! = $err; "$!" };
}
```

Οριακές περιπτώσεις

- **Αναντιστοιχία οικογένειας διευθύνσεων.** Αν το NAME πακεταρίστηκε για διαφορετική οικογένεια από αυτή με την οποία δημιουργήθηκε η υποδοχή (π.χ. sockaddr_in έναντι υποδοχής PF_INET6), η connect αποτυγχάνει με EAFNOSUPPORT ή EINVAL. Τα πακεταρισμένα bytes φέρουν την οικογένεια· ο πυρήνας ελέγχει.
- **Ήδη συνδεδεμένη.** Η κλήση connect σε υποδοχή ροής που είναι ήδη συνδεδεμένη αποτυγχάνει με EISCONN.
- **Υποδοχές χωρίς σύνδεση.** Σε υποδοχές SOCK_DGRAM / SOCK_RAW, η connect δεν ανταλλάσσει πακέτα· καταγράφει έναν προεπιλεγμένο ομότιμο ώστε οι επόμενες κλήσεις send / recv να μπορούν να παραλείψουν τη διεύθυνση. Μπορεί να ξανακληθεί με νέο NAME, ή να εκκαθαριστεί καλώντας τη με sockaddr AF_UNSPEC.
- **Υποδοχές χωρίς μπλοκάρισμα.** Αν η υποδοχή είναι σε λειτουργία χωρίς μπλοκάρισμα, ένα connect TCP συνήθως επιστρέφει ψευδές με την $! τεθειμένη σε EINPROGRESS. Η σύνδεση **δεν έχει ακόμα εγκαθιδρυθεί**. Περιμένετε να γίνει η υποδοχή εγγράψιμη (μέσω select ή του βρόχου συμβάντων σας), και μετά διαβάστε το SO_ERROR με την getsockopt για να λάβετε την τελική κατάσταση. Ένα SO_ERROR ίσο με μηδέν σημαίνει ότι η σύνδεση είναι επάνω· οτιδήποτε άλλο είναι ο πραγματικός κωδικός αποτυχίας.

- **Σήματα που διακόπτουν την κλήση.** Μια μπλοκαριστική `connect` μπορεί να επιστρέψει ψευδές με την `$!` τεθειμένη σε `EINTR` αν εκτελέστηκε χειριστής σήματος. Η υποδοχή εξακολουθεί να συνδέεται στο παρασκήνιο· πρέπει να περιμένετε στην εγγραψιμότητα και να ελέγξετε το `SO_ERROR` όπως στην περίπτωση χωρίς μπλοκάρισμα - **μην** ξαναπροσπαθήσετε την κλήση `connect`, αυτό παράγει `EALREADY`.
- **Μη δεσμευμένη διεύθυνση προέλευσης.** Αν δεν έχει κληθεί πρώτα η `bind`, ο πυρήνας επιλέγει διεύθυνση προέλευσης και εφήμερη θύρα αυτόματα. Η συνηθισμένη περίπτωση είναι να παραλείπεται η `bind` στον πελάτη.
- **Κλεισμένη ή μη ανοιγμένη υποδοχή.** Αποτυγχάνει με `EBADF`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `socket` - δημιουργεί το άκρο επί του οποίου λειτουργεί η `connect`
- `bind` - καρφώνει την τοπική διεύθυνση πριν τη σύνδεση· σπάνια χρειάζεται σε πελάτη
- `accept` - ο αντίστοιχος της `connect` από την πλευρά του διακομιστή
- `getpeername` - ανακτά τη διεύθυνση του ομότιμου μιας συνδεδεμένης υποδοχής
- `shutdown` - κλείνει τη μία κατεύθυνση μιας συνδεδεμένης υποδοχής χωρίς να απελευθερώνει τον περιγραφέα
- `Socket` - packers (`sockaddr_in`, `sockaddr_un`, ...) και σταθερές (`PF_INET`, `SOCK_STREAM`, ...) που χρησιμοποιούνται για τη δόμηση ορισμάτων
- `Errno` - συμβολικές σταθερές για σύγκριση της `$!` έναντι `EINPROGRESS`, `EISCONN`, `ECONNREFUSED`, ...

Ροή ελέγχου

continue

Συνδέει σε έναν βρόχο ένα μπλοκ που εκτελείται μετά από κάθε επανάληψη, ακριβώς πριν επαναελεγχθεί η συνθήκη.

Η `continue` δεν είναι ενσωματωμένη συνάρτηση - είναι δεσμευμένη λέξη ελέγχου ροής. Στη μία της εναπομένουσα σημασία εισάγει το τελικό μπλοκ ενός βρόχου `while`, `until` ή `foreach`. Το μπλοκ εκτελείται στο τέλος κάθε επανάληψης **και** οποτεδήποτε η `next` μεταφέρει τον έλεγχο εκτός του κύριου σώματος, αλλά **όχι** όταν το κάνουν οι `last` ή `redo`. Παίζει τον ρόλο που παίζει η τρίτη πρόταση ενός βρόχου `for (init; cond; step)` στη C.

Σύνοψη

```
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 }
```

Τι επιστρέφεται

Η continue είναι δήλωση, όχι έκφραση. Δεν έχει τιμή επιστροφής και δεν μπορεί να εμφανιστεί σε θέση τιμής. Ο ίδιος ο βρόχος αποδίδει ό,τι απαιτεί το περιβάλλον του (τίποτα σε κενό περιβάλλον, την τελευταία αποτιμημένη έκφραση κάτω από κάποιες κατασκευές)· το μπλοκ continue δεν συμβάλλει σε αυτό.

Μοντέλο εκτέλεσης

Για βρόχο με συνημμένο μπλοκ continue, κάθε επανάληψη εκτελείται με αυτή τη σειρά:

1. Αποτίμηση της συνθήκης (ή λήψη του επόμενου στοιχείου της λίστας, για foreach). Αν ο βρόχος δεν πρέπει να ξανατρέξει, έξοδος.
2. Εκτέλεση του κύριου σώματος.
3. Εκτέλεση του μπλοκ continue.
4. Επιστροφή στο βήμα 1.

Οι τρεις δεσμευμένες λέξεις ελέγχου βρόχου αλληλεπιδρούν με το μπλοκ continue διαφορετικά:

- next - άλμα κατευθείαν στο βήμα 3. Το μπλοκ continue **εκτελείται**, και μετά επαναελέγχεται η συνθήκη. Αυτό είναι το νόημα του να υπάρχει μπλοκ continue: κατάσταση που πρέπει να προχωρά σε κάθε επανάληψη (μετρητής, αριθμός γραμμής, μια εφάπαξ επαναφορά αντιστοιχίας) εξακολουθεί να προχωρά όταν το σώμα βραχυκυκλώνει μέσω next.
- last - εγκαταλείπει εντελώς τον βρόχο. Το μπλοκ continue **δεν** εκτελείται.
- redo - επανεκκινεί το βήμα 2 χωρίς να επαναελέγξει τη συνθήκη και χωρίς να εκτελέσει το μπλοκ continue. Χρησιμοποιήστε την όταν το σώμα χρειάζεται άλλο ένα πέρασμα πάνω από τα δεδομένα της τρέχουσας επανάληψης (για παράδειγμα, μετά την παρεμβολή επιπλέον εισόδου).

Η παράλειψη του μπλοκ continue ισοδυναμεί με κενό.

Παραδείγματα

Μετρητής που προχωρά ακόμη και όταν το σώμα παρακάμπτει:

```
my $n = 0;
for my $line (@lines) {
    next if $line =~ /^\s*#/;    # skip comments
    process($line);
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
} continue {
    $n++;                       # still counts skipped lines
}
print "scanned $n lines\n";
```

Ισοδυναμία με βρόχο `for` τύπου C - οι δύο μορφές παρακάτω είναι ίδιες, επειδή η Perl ξαναγράφει την κεφαλίδα τύπου C ως `while` με μπλοκ `continue`:

```
for (my $i = 1; $i < 10; $i++) { say $i }

my $i = 1;
while ($i < 10) {
    say $i;
} continue {
    $i++;
}
```

Επαναφορά εφάπαξ αντιστοιχιών και μετρητών γραμμής κατά τη ροή αρχείων με <>:

```
while (<>) {
    m?(fred)?   && s//WILMA/;
    m?(barney)? && s//BETTY/;
} continue {
    print "$ARGV $.: $_";
    close ARGV if eof;          # reset $.
    reset      if eof;          # reset ?pat?
}
```

Η αλληλεπίδραση με την `redo` - σημειώστε ότι η `continue` εκτελείται στη διαδρομή `next` αλλά όχι στη διαδρομή `redo`:

```
while (<>) {
    chomp;
    if (s/\\$//) {
        $_ .= <>;
        redo unless eof;        # merge continuation, re-enter body
    }                           # continue block skipped on redo
    process($_);
} continue {
    $seen++;                    # counts completed records only
}
```

Οριακές περιπτώσεις

- **No trailing block** - bare `continue` (without a following `{ ... }`) is a parse error in Perl 5.44. 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 5.44; `continue` now only has meaning as a trailing loop block.
- **Η `last` παρακάμπτει το μπλοκ `continue`.** Αν το μπλοκ περιέχει εκκαθάριση που χρειάζεστε σε κάθε διαδρομή εξόδου, βάλτε την εκκαθάριση μετά τον βρόχο ή σε

φύλακα εμβέλειας, όχι στο `continue`.

- **Η `redo` παρακάμπτει το μπλοκ `continue`.** Αν το μπλοκ `continue` προωθεί έναν μετρητή, μια επανάληψη μέσω `redo` δεν θα τον αυξήσει - αυτό είναι συνήθως αυτό που θέλετε (η επανάληψη δεν έχει ακόμη ολοκληρωθεί) αλλά μπορεί να εκπλήξει κώδικα που στηρίζεται στο ότι ο μετρητής ταιριάζει με τον αριθμό εισόδων στο σώμα.
- **Το `do { ... } while` δεν** είναι πραγματικός βρόχος - οι `last`, `next` και `redo` δεν λειτουργούν σε αυτό, και ένα τελικό μπλοκ `continue` είναι συντακτικό σφάλμα.
- **Βρόχοι με ετικέτα** - το `LABEL:` προσαρτάται στη δεσμευμένη λέξη `while` / `for` / `foreach`, όχι στο `continue`. Οι `next LABEL` και `last LABEL` στοχεύουν τον βρόχο ως σύνολο· το μπλοκ `continue` αποτελεί μέρος αυτού του βρόχου.
- **Ένα μπλοκ από μόνο του** είναι εφάπαξ βρόχος, οπότε ένα σκέτο `{ ... } continue { ... }` αναλύεται και εκτελεί το μπλοκ `continue` μία φορά μετά το σώμα - σπάνια χρήσιμο, αλλά νόμιμο.
- **Foreach με ψευδωνυμοποίηση** - μέσα σε βρόχο `foreach` η μεταβλητή του βρόχου είναι ψευδώνυμο του τρέχοντος στοιχείου της λίστας. Το μπλοκ `continue` εξακολουθεί να εκτελείται με αυτό το ψευδώνυμο εντός εμβέλειας, ώστε μπορείτε να επιθεωρήσετε ή να μεταβάλετε το στοιχείο εκεί όπως και στο σώμα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `next` - άλμα στο μπλοκ `continue` (αν υπάρχει) και επανέλεγχος της συνθήκης· η μόνη δεσμευμένη λέξη ελέγχου βρόχου που **εκτελεί** το μπλοκ `continue`
- `last` - εγκατάλειψη του βρόχου εξ ολοκλήρου· **παρακάμπτει** το μπλοκ `continue`
- `redo` - επανεκκίνηση του σώματος χωρίς επανέλεγχο της συνθήκης· **παρακάμπτει** το μπλοκ `continue`
- `return` - εγκατάλειψη της περικλείουσας υπορουτίνας· επίσης παρακάμπτει το μπλοκ `continue` επειδή φεύγει από τον βρόχο
- `break` - companion keyword of the removed `switch` feature; like the bare `given` / `when` form of `continue`, no longer available in 5.44

Αριθμητικές συναρτήσεις

cos

Επιστρέφει το συνημίτονο ενός αριθμού δοσμένου σε ακτίνια.

Το `cos` είναι το τυπικό τριγωνομετρικό συνημίτονο: δέχεται μια γωνία μετρημένη σε **ακτίνια** και επιστρέφει τον λόγο της προσκείμενης πλευράς προς την υποτείνουσα του αντίστοιχου ορθογωνίου τριγώνου. Αν παραλειφθεί η EXPR, η `cos` ενεργεί επί της `$_`. Το αποτέλεσμα είναι πάντα αριθμός κινητής υποδιαστολής στο κλειστό διάστημα `[-1, 1]`.

Σύνοψη

```
cos EXPR
cos
```

Τι επιστρέφεται

Αριθμός κινητής υποδιαστολής διπλής ακρίβειας στο `[-1, 1]`. Η Perl προωθεί την κλήση στη ρουτίνα C `cos(3)` της πλατφόρμας, οπότε η ακρίβεια και η συμπεριφορά στρογγυλοποίησης ταιριάζουν με τη libm σας. Οι ειδικές τιμές ακολουθούν το IEEE 754: το `cos(0)` είναι ακριβώς 1, τα `cos("inf")` και `cos("nan")` είναι αμφότερα `NaN`.

Ακτίνια, όχι μοίρες

Το πιο συχνό σφάλμα με το `cos` είναι να του δίνετε μοίρες. Μια πλήρης περιστροφή είναι `2 * π` ακτίνια, όχι `360`. Μετατρέψτε μία φορά και κρατήστε τη μετατραμμένη τιμή:

```
use POSIX qw(acos);
my $pi = acos(-1);                  # π 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
```

Το εξάγει τις `deg2rad` και `rad2deg` αν προτιμάτε να μη γράψετε μόνοι σας τη μετατροπή.

Παραδείγματα

Διατρέξτε τον μοναδιαίο κύκλο σε οκτώ βήματα και τυπώστε `(cos θ, sin θ)`:

```
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);
}
```

Συγκρίνετε το `cos` με τη σειρά Taylor γύρω από το μηδέν. Για μικρά `$x` το ανάπτυγμα τεσσάρων όρων είναι ήδη ακριβές μέχρι δέκα δεκαδικά ψηφία - ένας χρήσιμος έλεγχος ορθότητας κατά τη μεταφορά αριθμητικού κώδικα:

```
sub cos_taylor {
    my $x = shift;
    my $x2 = $x * $x;
    1 - $x2/2 + $x2*$x2/24 - $x2*$x2*$x2/720;
}
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
my $x = 0.3;
printf "libm:   %.15f\n", cos($x);
printf "taylor: %.15f\n", cos_taylor($x);
```

Προεπιλεγμένο όρισμα. Μέσα σε έναν βρόχο `while (<>)` το `cos` χωρίς όρισμα διαβάζει την τρέχουσα γραμμή - σχεδόν πάντα σφάλμα, αλλά η γλώσσα το επιτρέπει:

```
$_ = 0;
print cos, "\n";                    # 1
```

Αντίστροφο συνημίτονο μέσω της ταυτότητας από το `perlfunc`:

```
sub acos { atan2( sqrt(1 - $_[0] * $_[0]), $_[0] ) }

print acos(0.5), "\n";              # 1.0471975511966 (= π/3)
```

Χρήση του για το πλήρες σύνολο των αντίστροφων και υπερβολικών συναρτήσεων αντί να γράψετε δικές σας:

```
use Math::Trig;
print acos(0.5), "\n";              # same result, no hand-written␣
↪formula
```

Οριακές περιπτώσεις

- **Τα μη αριθμητικά ορίσματα εξαναγκάζονται σε `0`.** Το `cos("hello")` επιστρέφει `1` διότι η συμβολοσειρά αριθμοποιείται πρώτα σε `0`. Υπό `use warnings` αυτό παράγει προειδοποίηση `Argument "hello" isn't numeric in cos`. Επικυρώστε την είσοδο με τη `looks_like_number` από το `Scalar::Util` αν η πηγή είναι μη έμπιστη.
- **Τα πολύ μεγάλα ορίσματα χάνουν ακρίβεια.** Το `cos` ανάγει πρώτα το όρισμά του modulo 2π, και για `|x|` της τάξης του `1e16` το πλησιέστερο αναπαραστάσιμο double διαφέρει από το x κατά περισσότερο από π. Το αποτέλεσμα είναι ακόμη αριθμός στο `[-1, 1]`, αλλά είναι ουσιαστικά τυχαίο. Ανάγετε μόνοι σας τη γωνία πριν την κλήση αν συσσωρεύετε μια φάση επί πολλές επαναλήψεις.
- **Το `undef` εξαναγκάζεται σε `0`** και επιστρέφει `1`, με τη συνήθη προειδοποίηση `uninitialized` υπό `use warnings`.
- **Ειδικές τιμές IEEE**: τα `cos("inf")`, `cos("-inf")` και `cos("nan")` επιστρέφουν όλα `NaN`. Συγκρίνετε με `$result != $result` (η μοναδική τιμή που δεν είναι ίση με τον εαυτό της) για να ανιχνεύσετε `NaN`, ή χρησιμοποιήστε `POSIX::isnan`.
- **Δεν υπάρχει υποστήριξη μιγαδικών αριθμών** στο ενσωματωμένο. Για μιγαδικά ορίσματα χρησιμοποιήστε το , το οποίο υπερφορτώνει το `cos` μέσω υπερφόρτωσης τελεστών:

```
use Math::Complex;
my $z = cplx(1, 2);
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
print cos($z), "\n";                # 2.03272300701967-3.
↪0518977991518i
```

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `sin` - ημίτονο μιας γωνίας σε ακτίνια· η συντροφική συνάρτηση που χρησιμοποιείται συχνότερα παράλληλα με το `cos`
- `atan2` - τόξο εφαπτομένης δύο ορισμάτων, το συνηθισμένο δομικό στοιχείο για την αντίστροφη τριγωνομετρία (`acos`, `asin`) σε καθαρή Perl
- `sqrt` - τετραγωνική ρίζα· εμφανίζεται στην κλασική ταυτότητα του `acos` `acos(x) = atan2(sqrt(1 - x*x), x)`
- - `acos`, `asin`, `atan`, υπερβολικές παραλλαγές, `deg2rad`/`rad2deg` και βοηθητικά μεγάλου κύκλου
- - υπερφορτώνει το `cos` (και το υπόλοιπο της τριγωνομετρικής οικογένειας) για μιγαδικά ορίσματα
- `POSIX` - εκθέτει την `acos` απευθείας, καθώς και `isnan`/`isinf` για την ανίχνευση των μη πεπερασμένων αποτελεσμάτων που συζητήθηκαν παραπάνω

Βαθμωτά και συμβολοσειρές

crypt

Μονόδρομος κατακερματισμός τύπου `passwd(5)` μιας συμβολοσειράς απλού κειμένου με salt.

Η `crypt` είναι λεπτό περιτύλιγμα της συνάρτησης `crypt(3)` της C library του συστήματος. Μετατρέπει το `PLAINTEXT` και το `SALT` σε μια σύντομη συμβολοσειρά - μια *σύνοψη* - και την επιστρέφει. Τα ίδια `PLAINTEXT` και `SALT` παράγουν πάντα την ίδια σύνοψη· μικρές αλλαγές σε οποιοδήποτε από τα δύο παράγουν εντελώς διαφορετική σύνοψη. Δεν υπάρχει αντίστροφη συνάρτηση: δεν μπορείτε να ανακτήσετε το `PLAINTEXT` από μια σύνοψη.

⚠ Προειδοποίηση

Η `crypt` **δεν είναι συνάρτηση κρυπτογράφησης γενικής χρήσης**, παρά το όνομα. Δεν μπορεί να αναιρεθεί. Δεν είναι επίσης **σύγχρονος κατακερματισμός κωδικών πρόσβασης** στις περισσότερες παραδοσιακές διαμορφώσεις - το ιστορικό σχήμα Unix εξετάζει μόνο τα πρώτα οκτώ bytes του `PLAINTEXT` και παράγει σύνοψη 13 bytes πάνω σε salt 12 bits, που είναι ακατάλληλο για την προστασία κωδικών πρόσβασης από σύγχρονους επιτιθέμενους. Αυτό για το οποίο είναι κατάλληλη η `crypt` είναι η επαλήθευση μιας συμβολοσειράς που παρείχε ο χρήστης έναντι μιας ήδη αποθηκευμένης σύνοψης που έχει παραγάγει η `crypt` του ίδιου συστήματος. Για οτιδήποτε άλλο - κρυπτογράφηση, παραγωγή κλειδιών, ή κατακερματισμό μεγάλων

δεδομένων - χρησιμοποιήστε μια αληθινή κρυπτογραφική βιβλιοθήκη. Δείτε *Δείτε επίσης* παρακάτω.

Σύνοψη

```
crypt PLAINTEXT, SALT
```

Και τα δύο ορίσματα είναι συμβολοσειρές. Η τιμή επιστροφής είναι μια σύνοψη συμβολοσειράς.

Τι επιστρέφεται

Μια συμβολοσειρά της οποίας η ακριβής μορφή εξαρτάται από την υλοποίηση της `crypt(3)` του υπολογιστή. Παραδοσιακά είναι 13 bytes: δύο bytes salt ακολουθούμενα από 11 bytes από το σύνολο `[./0-9A-Za-z]`. Οι σύγχρονες glibc, musl και BSD libcs δέχονται εκτεταμένα salts που επιλέγουν ισχυρότερα σχήματα (MD5, SHA-256, SHA-512, bcrypt, yescrypt) και επιστρέφουν μεγαλύτερες συνόψεις στη μορφή αρχείου `passwd` `$id$salt$hash`.

Αντιμετωπίστε τη σύνοψη ως αδιαφανή. Μην την αναλύετε, μην υποθέτετε μήκος, και μην υποθέτετε ποια bytes του `SALT` κατανάλωσε η υλοποίηση. Ο σωστός τρόπος για να επαληθεύσετε ένα απλό κείμενο έναντι μιας υπάρχουσας σύνοψης είναι να περάσετε την ίδια τη σύνοψη πίσω ως salt:

```
if (crypt($plain, $digest) eq $digest) { ... }
```

Το salt που χρησιμοποιήθηκε για τη δημιουργία της σύνοψης είναι ενσωματωμένο στη σύνοψη, οπότε αυτό επανατρέχει το ίδιο σχήμα με τις ίδιες παραμέτρους στο υποψήφιο απλό κείμενο. Κώδικας γραμμένος έτσι λειτουργεί με το κλασικό σχήμα DES και με κάθε σύγχρονη επέκταση που εκθέτει μια δοθείσα `crypt(3)`, χωρίς να χρειάζεται να γνωρίζετε ποιο σχήμα παρήγαγε την αποθηκευμένη σύνοψη.

Καθολική κατάσταση που επηρεάζει

Η `crypt` δεν διαβάζει ούτε γράφει καμία ειδική μεταβλητή της Perl. Σε είσοδο Unicode μπορεί να εκπέμψει μοιραίο σφάλμα `Wide character in crypt` (δείτε *Οριακές περιπτώσεις*).

Παραδείγματα

Επαλήθευση κωδικού πρόσβασης έναντι αποθηκευμένης σύνοψης τύπου `passwd`:

```
my $stored = (getpwuid($<))[1];        # encrypted field from passwd

print "Password: ";
chomp(my $word = <STDIN>);

if (crypt($word, $stored) eq $stored) {
    print "ok\n";
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
} else {
    die "Sorry...\n";
}
```

Δημιουργία φρέσκιας σύνοψης για νέο κωδικό πρόσβασης χρησιμοποιώντας τυχαίο salt δύο χαρακτήρων από το κλασικό αλφάβητο. Το σύνολο είναι μόνο σύσταση· οι ακριβείς χαρακτήρες που γίνονται δεκτοί σε ένα salt είναι ό,τι επιτρέπει η `crypt(3)` του υπολογιστή, και η Perl δεν επιβάλλει δικό της περιορισμό:

```
my @chars = ('.', '/', 0..9, 'A'..'Z', 'a'..'z');
my $salt  = $chars[rand 64] . $chars[rand 64];
my $hash  = crypt($plaintext, $salt);
```

Αίτημα για συγκεκριμένο σύγχρονο σχήμα με χρήση εκτεταμένου salt (glibc και συμβατές libcs). Το πρόθεμα `$id$` επιλέγει τον αλγόριθμο - το `$1$` είναι MD5-crypt, το `$5$` είναι SHA-256-crypt, το `$6$` είναι SHA-512-crypt, το `$2b$` είναι bcrypt όπου υποστηρίζεται:

```
my $hash = crypt($plaintext, '$6$rounds=100000$' . $random_salt);
```

Επαναεπαλήθευση με χρήση της αποθηκευμένης σύνοψης ως salt - ανεξάρτητη από το σχήμα:

```
my $candidate = crypt($plaintext, $stored_digest);
my $match     = $candidate eq $stored_digest;
```

Η σύγκριση σταθερού χρόνου των δύο συνόψεων εξακολουθεί να είναι ευθύνη του καλούντος· η απλή `eq` διαρρέει πληροφορία χρονισμού. Αν το μοντέλο απειλών το απαιτεί, συγκρίνετε bit-προς-bit με βρόχο που δεν βραχυκυκλώνει, ή χρησιμοποιήστε ρουτίνα σύγκρισης από κρυπτογραφικό άρθρωμα.

Οριακές περιπτώσεις

- **Αποκοπή πρώτων-8-bytes στο κλασικό DES.** Στην παραδοσιακή `crypt(3)` του Unix, μόνο τα πρώτα οκτώ bytes του `PLAINTEXT` επηρεάζουν τη σύνοψη: η `crypt("password123", $s)` και η `crypt("password456", $s)` συγκρούονται. Τα εκτεταμένα σχήματα (`$1$`, `$5$`, `$6$`, `$2b$`, ...) δεν έχουν αυτό το όριο, αλλά το κλασικό σχήμα είναι αυτό που λαμβάνετε από salt δύο bytes χωρίς πρόθεμα.
- **Οι υποθέσεις για το μήκος του salt είναι λάθος.** Μην υποθέτετε ότι η `substr($digest,0,2)` είναι το salt. Οι εκτεταμένες συνόψεις μεταφέρουν το salt σε διαφορετική θέση και με διαφορετικό μήκος. Επαναχρησιμοποιήστε την πλήρη σύνοψη ως όρισμα salt (`crypt($plain, $digest)`) και αφήστε την `crypt(3)` να την αναλύσει.
- **Είσοδος Unicode.** Αν το `PLAINTEXT` ή το `SALT` περιέχει χαρακτήρες πάνω από το σημείο κώδικα 255, η Perl επιχειρεί να υποβαθμίσει ένα αντίγραφο της συμβολοσειράς σε bytes πριν καλέσει την `crypt(3)`. Αν κάθε χαρακτήρας χωρά σε ένα byte, η υποβάθμιση πετυχαίνει σιωπηλά. Αν δεν χωρά κάποιος, η `crypt` πεθαίνει με `Wide character in crypt`. Κωδικοποιήστε ρητά (π.χ. `Encode::encode_utf8`) αν το απλό κείμενό σας μπορεί να περιέχει ευρείς χαρακτήρες.

- **Μη υποστηριζόμενο σχήμα επιστρέφει αποτυχία.** Το πέρασμα ενός προθέματος `$id$` που δεν κατανοεί η `crypt(3)` του υπολογιστή τυπικά επιστρέφει undef ή μια σύντομη συμβολοσειρά που ξεκινά με `*0` ή `*1`. Πάντα ελέγχετε τη μορφή επιστροφής πριν τη χρησιμοποιήσετε.
- **Η `crypt(3)` μπορεί να απουσιάζει ή να είναι περιορισμένη.** Σε μικρό αριθμό υπολογιστών η συνάρτηση της βιβλιοθήκης έχει αφαιρεθεί ή αντικατασταθεί με stub για λόγους ελέγχου εξαγωγών. Η `crypt` στην Perl είναι τόσο ικανή όσο η υποκείμενη κλήση libc· αν η κλήση libc είναι stub, τότε το ίδιο ισχύει και εδώ.
- **Ακατάλληλη για κατακερματισμό μεγάλων δεδομένων.** Η `crypt` αγνοεί το μεγαλύτερο μέρος της εισόδου της στο κλασικό σχήμα και είναι αργή ανά κλήση στα τεντωμένα σχήματα. Είναι το λάθος εργαλείο για άθροισμα ελέγχου αρχείων ή για δόμηση αποθήκευσης διευθυνσιοδοτούμενης μέσω περιεχομένου.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44. pperl 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.

Δείτε επίσης

- `Digest::MD5` - MD5-128 πάνω από αυθαίρετες συμβολοσειρές bytes· χρησιμοποιήστε για αποτύπωση δεδομένων, όχι για αποθήκευση κωδικών πρόσβασης
- `Digest::SHA` - συνόψεις οικογένειας SHA-1/2· χρησιμοποιήστε για ελέγχους ακεραιότητας και ως δομικό στοιχείο για HMAC και παραγωγή κλειδιών
- `Crypt::Bcrypt`, `Crypt::Argon2`, `Crypt::Passphrase` - αρθρώματα CPAN για σύγχρονο κατακερματισμό κωδικών πρόσβασης· προτιμήστε αυτά αντί της `crypt` για κάθε νέο κώδικα επαλήθευσης κωδικών
- `Crypt::Eksblowfish`, `Crypt::ScryptKDF` - αρθρώματα CPAN για παραγωγή κλειδιών βάσει bcrypt και scrypt
- getpwuid - ανακτά το πεδίο κρυπτογραφημένου κωδικού από την τοπική βάση δεδομένων `passwd`· το έβδομο πεδίο της είναι η σύνοψη που περνάτε ως `SALT` κατά την επαλήθευση
- chomp - αφαιρεί τον τελικό newline από έναν κωδικό πρόσβασης που διαβάστηκε με `<STDIN>` πριν τον περάσετε στην `crypt`

I/O · Κλάσεις και αντικειμενοστρέφεια

dbmclose

Διακόπτει τη δέσμευση μεταξύ ενός αρχείου DBM και ενός hash που είχε προηγουμένως προσαρτηθεί με dbmopen.

Η `dbmclose` είναι το αντίστοιχο αποδόμησης της dbmopen. Εκκενώνει τυχόν εκκρεμείς εγγραφές, απελευθερώνει τα handles της βιβλιοθήκης DBM και αφήνει το `HASH` ως ένα συνηθισμένο, μη tied hash που μπορείτε να συνεχίσετε να χρησιμοποιείτε ή να

απορρίψετε. Σε σύγχρονο κώδικα η dbmclose έχει σε μεγάλο βαθμό αντικατασταθεί από την untie - οι δύο επιτυγχάνουν το ίδιο πράγμα, αλλά η untie λειτουργεί για κάθε tied μεταβλητή, όχι μόνο για hashes δεσμευμένα σε DBM. Προτιμήστε την untie εκτός αν συντηρείτε υπάρχοντα ζεύγη dbmopen/dbmclose και θέλετε να διαβάζονται συμμετρικά.

Σύνοψη

```
dbmclose HASH
dbmclose %hash
```

Τι επιστρέφεται

Αληθές σε επιτυχία, ψευδές σε αποτυχία. Οι περιπτώσεις αποτυχίας είναι σπάνιες - η συνηθέστερη είναι το πέρασμα ενός hash που εξαρχής δεν ήταν tied σε αρχείο DBM, οπότε η dbmclose επιστρέφει ψευδές και θέτει το $!. Τα περισσότερα προγράμματα δεν ελέγχουν την τιμή επιστροφής· τα ενδιαφέροντα σφάλματα (γεμάτος δίσκος, υπέρβαση ορίου χώρου) εμφανίζονται στις εγγραφές που προηγήθηκαν του κλεισίματος, όχι στην ίδια την dbmclose. Ελέγξτε την όταν το αρχείο DBM βρίσκεται σε σύστημα αρχείων όπου η καθυστερημένη αποτυχία εγγραφής κατά το κλείσιμο είναι πιθανή:

```
dbmclose(%cache)
    or warn "dbmclose failed: $!";
```

Καθολική κατάσταση που επηρεάζει

Η dbmclose διαβάζει και τροποποιεί την κατάσταση tie του HASH. Δεν αγγίζει καμία από τις συνήθεις ειδικές μεταβλητές που σχετίζονται με την I/O ($,, $\, $|, το επιλεγμένο filehandle)· η πρόσβαση DBM δεν περνά μέσα από τα στρώματα PerlIO που χρησιμοποιεί η print και οι όμοιές της. Σε αποτυχία, το $! τίθεται από την υποκείμενη βιβλιοθήκη DBM.

Παραδείγματα

Ανοίξτε ένα αρχείο DBM, διασχίστε το επαναληπτικά, κλείστε το:

```
dbmopen(%HIST, '/usr/lib/news/history', 0666)
    or die "dbmopen: $!";
while (my ($key, $val) = each %HIST) {
    print $key, ' = ', unpack('L', $val), "\n";
}
dbmclose(%HIST);
```

Ισοδύναμο με χρήση του σύγχρονου ζεύγους tie/untie - έτσι θα έπρεπε να μοιάζει ο νέος κώδικας:

```
use AnyDBM_File;
use Fcntl;
tie(my %HIST, 'AnyDBM_File', '/usr/lib/news/history', O_RDWR, 0666)
    or die "tie: $!";
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
# ... use %HIST ...
untie %HIST;
```

Προστατευτείτε από το κλείσιμο ενός μη δεσμευμένου hash:

```
my %h;
dbmclose(%h)
    or warn "not tied to a DBM file: $!";   # warns
```

Συνδυάστε με ένα μπλοκ `eval` για να εντοπίσετε τοπικά τυχόν σφάλματα εκκαθάρισης χωρίς να ματαιωθεί ο περιβάλλων κώδικας:

```
eval {
    dbmclose(%cache) or die "dbmclose: $!";
    1;
} or warn $@;
```

Οριακές περιπτώσεις

- **Hash όχι tied σε αρχείο DBM**: επιστρέφει ψευδές, θέτει το `$!`. Δεν εγείρεται εξαίρεση. Το hash μένει αμετάβλητο.
- **Hash tied μέσω `tie`, όχι `dbmopen`**: η `dbmclose` εξακολουθεί να λειτουργεί και είναι ισοδύναμη με την `untie` για αυτό το hash, επειδή η ίδια η `dbmopen` υλοποιείται ως `tie` εσωτερικά. Η ανάμειξη των δύο API είναι νόμιμη αλλά μπερδεύει - διαλέξτε ένα ανά hash και μείνετε σε αυτό.
- **Αναφορές εξακολουθούν να κρατούνται στο tied αντικείμενο**: όπως η `untie`, η `dbmclose` διακόπτει τη δέσμευση μεταξύ του hash και του αρχείου DBM, αλλά αν άλλος κώδικας εξακολουθεί να κρατά αναφορά στο υποκείμενο tied αντικείμενο, η βιβλιοθήκη DBM ενδέχεται να μην απελευθερώσει στην πραγματικότητα τα handles της μέχρι να απορριφθεί αυτή η αναφορά. Το σύμπτωμα είναι ένα αρχείο DBM που μένει κλειδωμένο ή ανοιχτό πέρα από την κλήση `dbmclose`. Ένα πρόγραμμα με `use warnings` λαμβάνει σε αυτή την περίπτωση την προειδοποίηση `untie attempted while N inner references still exist`.
- **Κλήση της `dbmclose` δύο φορές στο ίδιο hash**: η πρώτη κλήση πετυχαίνει, η δεύτερη επιστρέφει ψευδές. Κώδικας με στυλ idempotent θα πρέπει είτε να παρακολουθεί αν το hash είναι αυτή τη στιγμή tied είτε να αγνοεί την αποτυχία.
- **Το hash γίνεται `local` ή φεύγει από την εμβέλειά του χωρίς `dbmclose`**: η δέσμευση διακόπτεται έμμεσα όταν το hash καταστρέφεται, και η βιβλιοθήκη DBM κλείνει το αρχείο ως μέρος αυτής της αποδόμησης. Το ρητό `dbmclose` είναι προτιμότερο, επειδή κάνει τις αποτυχίες εκκένωσης εγγραφής ορατές σε ένα γνωστό σημείο του προγράμματος.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `dbmopen` - το μισό άνοιγμα του ζεύγους· κάθε `dbmclose` αντιστοιχεί σε ένα προηγούμενο dbmopen στο ίδιο hash
- `untie` - η σύγχρονη, γενικού σκοπού αντικατάσταση· λειτουργεί για κάθε tied μεταβλητή, όχι μόνο για hashes δεσμευμένα σε DBM
- `tie` - το σύγχρονο αντίστοιχο της `dbmopen`, που χρησιμοποιείται με τα `AnyDBM_File`, `DB_File`, `GDBM_File`, `NDBM_File`, ή `SDBM_File` για τη δέσμευση ενός hash σε αρχείο DBM
- `close` - άσχετο, αλλά η ανάλογη λειτουργία για filehandles: χρειάζεται όταν σας ενδιαφέρει οι αποτυχίες εκκένωσης εγγραφής να εμφανίζονται σε ένα γνωστό σημείο

I/O · Κλάσεις και αντικειμενοστρέφεια

dbmopen

Συνδέει ένα αρχείο DBM στον δίσκο με έναν κατακερματισμό, ώστε οι αναγνώσεις και εγγραφές του κατακερματισμού να γίνονται αναζητήσεις και αποθηκεύσεις στη βάση δεδομένων.

Η dbmopen είναι ο παλιού τύπου τρόπος να δέσει κανείς (tie) έναν κατακερματισμό με ένα αρχείο dbm, ndbm, sdbm, gdbm ή Berkeley DB. Έχει σε μεγάλο βαθμό αντικατασταθεί από την `tie` με ρητό άρθρωμα backend DBM (`DB_File`, `GDBM_File`, `SDBM_File`, `NDBM_File`, `AnyDBM_File`), που σας δίνει έλεγχο επί του ποια υλοποίηση χρησιμοποιείται και επί των επιλογών ανά backend. Επιλέξτε την dbmopen όταν διαβάζετε παλαιότερο κώδικα ή όταν η συντομία του μονόγραμμου έχει μεγαλύτερη σημασία από την επιλογή backend.

Σε αντίθεση με την `open`, το πρώτο όρισμα **δεν** είναι filehandle - είναι ο κατακερματισμός που θα δεθεί. Το `HASH` γράφεται με το sigil `%`.

Σύνοψη

```
dbmopen %HASH, $DBNAME, $MASK
dbmopen(%HASH, $DBNAME, 0)       # open existing only; never create
dbmclose %HASH                   # or: untie %HASH
```

Τι επιστρέφεται

Αληθής τιμή σε επιτυχία, ψευδής τιμή σε αποτυχία. Σε αποτυχία, το `$!` τίθεται στο σφάλμα συστήματος που εμπόδισε τη σύνδεση. Οι συνήθεις περιπτώσεις αποτυχίας είναι «η βάση δεδομένων δεν υπάρχει και η `MASK` είναι `0`» και σφάλματα δικαιωμάτων στα υποκείμενα αρχεία.

```
dbmopen(%cache, $path, 0)
    or die "cache $path missing or unreadable: $!";
```

Μετά από επιτυχημένη κλήση, ο `%HASH` συμπεριφέρεται όπως οποιοσδήποτε άλλος κατακερματισμός: το `$HASH{key} = $value` γράφει στη βάση δεδομένων, το

`$HASH{key}` διαβάζει από αυτήν, η `delete` αφαιρεί μια εγγραφή, η `exists` ελέγχει συμμετοχή, και οι `each` / `keys` / `values` επαναλαμβάνουν. Τα δεδομένα βρίσκονται στον δίσκο· ο κατακερματισμός είναι μόνο μια προβολή.

Ορίσματα

- **HASH** - η μεταβλητή κατακερματισμού που θα δεθεί. Πρέπει να γραφτεί με %. Οποιαδήποτε υπάρχοντα περιεχόμενα απορρίπτονται. Ο κατακερματισμός παραμένει δεσμευμένος μέχρι να κληθεί επ” αυτού η `dbmclose` ή η `untie`, ή μέχρι να βγει εκτός εμβέλειας.
- **DBNAME** - η διαδρομή της βάσης δεδομένων **χωρίς** την επέκταση αρχείου του backend. Τα παραδοσιακά αποθηκευτικά μέσα dbm / sdbm χρησιμοποιούν δύο αρχεία με ονόματα `` `DBNAME.dir` `` και `` `DBNAME.pag` ``· τα gdbm και Berkeley DB χρησιμοποιούν ένα μόνο αρχείο. Το πέρασμα του `"/var/lib/app/cache"` ανοίγει τα `/var/lib/app/cache.dir` + `/var/lib/app/cache.pag` υπό sdbm, ή το `/var/lib/app/cache` υπό gdbm.
- **MASK** - λειτουργία δημιουργίας για όσα αρχεία πρέπει να δημιουργηθούν, στην ίδια μορφή με το τρίτο όρισμα της `chmod` (τυπικά οκταδικό κυριολεκτικό όπως `0666` ή `0644`). Η ενεργή `umask` εφαρμόζεται στη MASK, οπότε τα πραγματικά δικαιώματα στον δίσκο είναι `MASK & ~umask`. **Μια MASK ίση με `0` καταστέλλει τη δημιουργία**: αν η βάση δεδομένων δεν υπάρχει ήδη, η dbmopen επιστρέφει ψευδές και το `$!` τίθεται.

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται σε αποτυχία (βάση δεδομένων ανύπαρκτη, άρνηση δικαιωμάτων, αλλοιωμένο υποκείμενο μέσο).
- Η ενεργή `umask` - μασκάρει τη MASK όταν δημιουργούνται νέα υποκείμενα αρχεία.

Παραδείγματα

Ανοίξτε μια βάση δεδομένων ιστορικού, επαναλάβετε με `each` και έπειτα κλείστε την:

```
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;
```

Άνοιγμα μόνο για ανάγνωση, αρνούμενο τη δημιουργία της βάσης δεδομένων:

```
dbmopen(%cache, "$ENV{HOME}/.app/cache", 0)
    or die "no cache at $ENV{HOME}/.app/cache: $!";
my $hit = $cache{$key};
dbmclose %cache;
```

Επιλέξτε ένα συγκεκριμένο backend φορτώνοντας το άρθρωμά του πριν καλέσετε την dbmopen. Το AnyDBM_File συμβουλεύεται το `@AnyDBM_File::ISA` για να αποφασίσει ποια υλοποίηση χρησιμοποιεί η dbmopen:

```
use DB_File;
dbmopen(%NS_Hist, "$ENV{HOME}/.netscape/history.db", 0644)
    or die "open netscape history: $!";
```

Προστατευμένη από εγγραφή βάση δεδομένων - η εκχώρηση αποτυγχάνει σιωπηλά (ή πεθαίνει ανάλογα με το backend)· προφυλαχθείτε με `eval` όταν χρειάζεστε διερεύνηση:

```
dbmopen(%db, $path, 0) or die $!;
my $writable = eval { $db{__probe__} = 1; delete $db{__probe__}; 1 }
 ↪;
warn "read-only\n" unless $writable;
```

Μεγάλη βάση δεδομένων - αποφύγετε τις `keys` / `values`, που υλοποιούν ολόκληρη τη λίστα στη μνήμη. Χρησιμοποιήστε αντί γι” αυτές την `each`:

```
dbmopen(%big, $path, 0) or die $!;
while (my ($k, $v) = each %big) {
    process($k, $v);
}
dbmclose %big;
```

Οριακές περιπτώσεις

- **Το sigil κατακερματισμού είναι απαραίτητο.** `dbmopen %h, $name, 0666` - η γραφή `$h` ή `\%h` δεν λειτουργεί. Η πρώτη θέση είναι το όνομα του κατακερματισμού ως κατακερματισμός, όχι ως αναφορά.
- **Το `MASK = 0` είναι το ιδίωμα «άνοιγμα μόνο υπάρχουσας».** Ένα φαινομενικά εύλογο `0666` δημιουργεί σιωπηλά μια νέα κενή βάση δεδομένων αν το αρχείο λείπει, πράγμα που σχεδόν ποτέ δεν θέλει ένας αναγνώστης σε διαδρομή αναζήτησης σε cache.
- **Οι επεκτάσεις διαχειρίζονται από το backend.** Μην περάσετε `"cache.dir"` ή `"cache.pag"` - περάστε `"cache"`. Το `sdbm` προσθέτει `.dir` / `.pag`· το `gdbm` γράφει στο γυμνό όνομα.
- **Ένα άνοιγμα ανά διεργασία για παλαιότερο DBM.** Αν το backend είναι η παραδοσιακή βιβλιοθήκη `dbm(3)` (σπάνια σήμερα), μια διεργασία μπορεί να έχει μόνο ένα DBM ανοιχτό τη φορά. Τα `sdbm` και `gdbm` δεν έχουν αυτόν τον περιορισμό.
- **Όρια μεγέθους τιμών.** Το `sdbm` περιορίζει τα κλειδιά + τιμές σε περίπου 1008 bytes ανά εγγραφή· υπερβολικές εγγραφές αποκόπτονται ή απορρίπτονται. Τα `gdbm` και Berkeley DB έχουν πολύ υψηλότερα όρια. Σε περίπτωση αμφιβολίας, χρησιμοποιήστε την `tie` με `DB_File` και ελέγξτε τα όρια του ίδιου του backend.
- **Οι αριθμητικές τιμές είναι bytes στον δίσκο.** Η αποθήκευση `$h{k} = 42` γράφει τη συμβολοσειρά `"42"`, όχι ακέραιο. Πακετάρετε ρητά με `pack` / `unpack` όταν το πεδίο χρειάζεται σταθερή δυαδική διάταξη (όπως στο παράδειγμα του ιστορικού news παραπάνω).
- **Το κλείσιμο είναι προαιρετικό αλλά συνιστάται.** Η σύνδεση απελευθερώνεται όταν ο κατακερματισμός βγει εκτός εμβέλειας, αλλά μια ρητή `dbmclose` ή `untie` εκκενώνει τις εκκρεμείς εγγραφές και αναδεικνύει σφάλματα κατά το κλείσιμο.

- **Εκ νέου σύνδεση ενός ήδη συνδεδεμένου κατακερματισμού.** Η κλήση της dbmopen σε κατακερματισμό που είναι ήδη συνδεδεμένος κλείνει πρώτα την προηγούμενη σύνδεση.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `dbmclose` - απελευθερώστε μια σύνδεση dbmopen και εκκενώστε τις εκκρεμείς εγγραφές
- `tie` - η σύγχρονη αντικαταστάτρια· συνδυάστε με `DB_File`, `GDBM_File`, `SDBM_File` ή `AnyDBM_File` για να επιλέξετε backend ρητά
- `untie` - απελευθερώστε μια σύνδεση που έγινε είτε με dbmopen είτε με `tie`
- `each` - ροή εγγραφών χωρίς υλοποίηση ολόκληρης της βάσης δεδομένων στη μνήμη
- `umask` - μασκάρει το όρισμα `MASK` όταν δημιουργούνται υποκείμενα αρχεία
- `open` - η συνηθισμένη ενσωματωμένη συνάρτηση ανοίγματος αρχείου, που δέχεται filehandle και όχι κατακερματισμό

Ροή ελέγχου

defer

Προγραμματισμός εκτέλεσης ενός μπλοκ όταν εξέρχεται η περικλείουσα εμβέλεια, για οποιονδήποτε λόγο.

Η `defer` είναι δεσμευμένη λέξη ελέγχου ροής, όχι συνάρτηση. Γράφοντας `defer { ... }` μέσα σε μια εμβέλεια καταγράφει το μπλοκ για μεταγενέστερη εκτέλεση· όταν ο έλεγχος εγκαταλείπει το περικλείον μπλοκ - με πτώση από το τέλος, με `return`, με `die`, με goto, ή με μια δήλωση ελέγχου βρόχου (`next`, `last`, `redo`) - το αποθηκευμένο μπλοκ εκτελείται κατά την έξοδο. Είναι το τυπικό ιδίωμα για τη συσχέτιση απόκτησης με απελευθέρωση (open/close, lock/unlock, push/pop) χωρίς εξάρτηση από καταστροφείς αντικειμένων ή τύλιγμα κάθε διαδρομής σε `eval`.

Το χαρακτηριστικό είναι **πειραματικό** και πρέπει να ενεργοποιηθεί. Είτε δηλώστε ρητή συμμετοχή είτε χρησιμοποιήστε ένα bundle:

```
use feature 'defer';    # explicit
use v5.36;              # bundle - enables defer among others
```

Υπό `use warnings 'experimental::defer'` (ή απλό `use warnings`) η πρώτη χρήση εκπέμπει πειραματική προειδοποίηση. Σιγάστε την με `no warnings 'experimental::defer'` αφότου αποφασίσετε να βασιστείτε στο χαρακτηριστικό.

Σύνοψη

```
defer BLOCK
```

Η `defer` δέχεται ένα μπλοκ οριοθετημένο από αγκύλες και τίποτε άλλο. Δεν υπάρχει μορφή έκφρασης, μορφή με filehandle, ούτε τιμή επιστροφής - η δήλωση υπάρχει για την παρενέργειά της κατά την έξοδο από την εμβέλεια.

Τι επιστρέφεται

Η `defer` είναι δήλωση, όχι έκφραση. Δεν συνεισφέρει καμία τιμή στο περιβάλλον της. Η ίδια η τιμή επιστροφής του μπλοκ απορρίπτεται όταν εκτελείται κατά την έξοδο από την εμβέλεια.

Πότε εκτελείται το αναβληθέν μπλοκ

Το αναβληθέν μπλοκ εκτελείται **μία φορά**, όταν ο έλεγχος εγκαταλείπει το άμεσα περικλείον μπλοκ, ανεξάρτητα από το πώς εξέρχεται ο έλεγχος:

- κανονική πτώση από το τέλος του μπλοκ,
- ρητή `return` από την περιέχουσα υπορουτίνα,
- εξαίρεση που εκτοξεύτηκε από `die` ή διαδόθηκε από κληθείσα υπορουτίνα,
- `goto` εκτός του μπλοκ,
- έλεγχος βρόχου μέσω `next`, `last` ή `redo`.

Αν η ροή εκτέλεσης δεν φτάσει ποτέ στην ίδια τη δήλωση `defer`, το μπλοκ **δεν** εισάγεται στην ουρά και δεν εκτελείται. Αυτό είναι το ακριβές αντίθετο ενός φασικού `END`, τον οποίο ο μεταγλωττιστής εισάγει στην ουρά άνευ όρων:

```
use feature 'defer';

{
    defer { say "This will run"; }
    return;
    defer { say "This will not"; }   # never reached, never␣
↪scheduled
}
```

Σειρά LIFO

Πολλαπλά μπλοκ `defer` στην ίδια εμβέλεια εκτελούνται με σειρά **last-in, first-out** - το πιο πρόσφατα προγραμματισμένο μπλοκ εκτελείται πρώτο. Αυτό ταιριάζει στη φυσική εμφώλευση της απόκτησης πόρων: το τελευταίο που αποκτήθηκε είναι το πρώτο που απελευθερώνεται.

```
use feature 'defer';

{
    defer { say "1"; }
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    defer { say "2"; }
    defer { say "3"; }
}
# prints 3, then 2, then 1
```

Καθολική κατάσταση που επηρεάζει

Καμία άμεσα. Το αναβληθέν μπλοκ βλέπει τις καθολικές του διερμηνέα ($_, $!, $@, το επιλεγμένο filehandle, κ.λπ.) με όποιες τιμές κρατούν **τη στιγμή που εκτελείται το μπλοκ**, όχι τη στιγμή που προγραμματίστηκε. Μια `defer` που θέλει να διατηρήσει την `$!` ή την `$@` κατά τον καθαρισμό πρέπει να τις αποθηκεύσει και να τις αποκαταστήσει η ίδια - αλλιώς μια κλήση συστήματος μέσα στο μπλοκ θα τις καταστρέψει και ο καλών της περικλείουσας υπορουτίνας θα δει τις αντικατεστημένες τιμές.

```
defer {
    local ($!, $@);
    close $fh;          # clobbers $! on failure - local'd copy␣
↪absorbs it
}
```

Παραδείγματα

Συσχετισμός μιας open με την αντίστοιχη `close`, με εγγύηση εκτέλεσης ακόμα και σε `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.
}
```

Ξεκλείδωμα ενός κοινόχρηστου πόρου σε κάθε διαδρομή εξόδου:

```
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
}
```

Καθαρισμός LIFO εμφωλευμένων πόρων - ο εξωτερικός πόρος απελευθερώνεται τελευταίος, κάτι που συνήθως είναι το επιθυμητό:

```
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.
}
```

Προσωρινή προσαρμογή της κατάστασης του διερμηνέα και αποκατάστασή της κατά την έξοδο χωρίς χρήση του `local`:

```
use feature 'defer';

sub with_separator {
    my $saved = $,;
    $, = " | ";
    defer { $, = $saved; }

    print @_, "\n";
}
```

Οριακές περιπτώσεις

- **Υπό συνθήκη προγραμματισμός**. Μια δήλωση `defer` προγραμματίζει το μπλοκ της μόνο όταν ο έλεγχος πραγματικά φτάσει σε αυτήν. Μια πρόωρη `return` ή `die` που συμβαίνει πριν τη γραμμή `defer` δεν αφήνει τίποτα προς εκτέλεση. Τοποθετήστε την `defer` αμέσως μετά τη συσχετισμένη απόκτηση, ώστε να μην μπορούν να διαχωριστούν από μια ενδιάμεση αποτυχία.
- **Αφού προγραμματιστεί, εκτελείται πάντα**. Δεν υπάρχει μηχανισμός ακύρωσης του προγραμματισμού ενός μπλοκ `defer`. Αν η απόκτηση που απελευθερώνει το μπλοκ έχει αποτύχει, είτε μην προγραμματίζετε την `defer` (κρατήστε την μετά τον έλεγχο επιτυχίας) είτε κάντε το μπλοκ ταυτοδυναμικό ώστε μια περιττή απελευθέρωση να είναι αβλαβής.
- **Καμία διαφυγή ροής ελέγχου από το μπλοκ**. Ένα μπλοκ `defer` μπορεί να εκτοξεύσει εξαίρεση, αλλά **δεν** μπορεί να κάνει `return` από την περικλείουσα υπορουτίνα, να κάνει `goto` σε ετικέτα εκτός του εαυτού του, ή να εκτελέσει `next`, `last` ή `redo` για βρόχο που περικλείει την `defer`. Αυτές οι κατασκευές είναι αποδεκτές *μέσα* στην `defer` όταν δρουν σε βρόχους **που περιέχονται εξ ολοκλήρου εντός** του ίδιου του σώματος του μπλοκ:

```
defer {
    foreach (1 .. 5) { last if $_ == 3; }   # permitted
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
}

foreach (6 .. 10) {
    defer { last if $_ == 8; }              # forbidden
}
```

- **Εξαίρεση κατά τη διάρκεια ξετυλίγματος εξαίρεσης**. Αν το μπλοκ `defer` εκτελείται επειδή διαδίδεται μια εξαίρεση και το ίδιο το μπλοκ εκτοξεύει, η γλώσσα δεν καθορίζει την προκύπτουσα εξαίρεση - μόνο ότι ο καλών θα δει κάποια. Μη βασίζεστε σε ποια εξαίρεση θα επικρατήσει. Τυλίξτε τον επικίνδυνο καθαρισμό σε `eval` αν χρειάζεστε να διατηρηθούν και τα δύο σήματα.
- **Οι εξαιρέσεις διαδίδονται κανονικά**. Μια εξαίρεση που εκτοξεύεται από ένα μπλοκ `defer` το οποίο εκτελείται για οποιονδήποτε άλλο λόγο (κανονική έξοδος, return, έλεγχος βρόχου) διαδίδεται στον καλούντα ακριβώς όπως κάθε άλλη εξαίρεση. Μια αποτυχημένη `close` στον καθαρισμό μπορεί έτσι να μετατρέψει μια φαινομενικά επιτυχημένη συνάρτηση σε ορατή στον καλούντα `die`.
- **Οι $@ και $! μέσα στο μπλοκ**. Στη διαδρομή εξόδου που οδηγείται από εξαίρεση, η `$@` κρατά την εν εξελίξει εξαίρεση όσο εκτελείται το μπλοκ `defer`. Η ανάγνωσή της είναι αποδεκτή· η αντικατάστασή της αλλάζει αυτό που βλέπει ο καλών μετά την ολοκλήρωση του ξετυλίγματος.
- **Λεπτότητα εμβέλειας**. Η `defer` συνδέεται με το **άμεσα** περικλείον μπλοκ, όχι με την περικλείουσα υπορουτίνα. Μια `defer` μέσα σε `if (...) { defer { ... } }` εκτελείται όταν εξέρχεται το μπλοκ `if`, όχι όταν επιστρέφει η υπορουτίνα. Τοποθετήστε την `defer` σε εμβέλεια υπορουτίνας αν θέλετε σημασιολογία εξόδου υπορουτίνας.
- **Πειραματική προειδοποίηση**. Υπό `use warnings` η πρώτη εμφάνιση προειδοποιεί. Δηλώστε σκόπιμη συμμετοχή με `no warnings 'experimental::defer'` αφού δεσμευτείτε στο χαρακτηριστικό.
- **Μη διαθέσιμη χωρίς το guard του χαρακτηριστικού**. Χωρίς `use feature 'defer'` (ή ένα bundle που το ενεργοποιεί), το `defer { ... }` αναλύεται ως κλήση μιας υπορουτίνας ονόματι `defer` ακολουθούμενη από ένα μπλοκ - σχεδόν σίγουρα όχι ό,τι εννοούσατε, και συνήθως σφάλμα κατά τη μεταγλώττιση.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44. 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 5.44.

Δείτε επίσης

- `eval` - το καθιερωμένο εργαλείο για τη μετατροπή εξαιρέσεων σε τιμές επιστροφής· η `defer` το συμπληρώνει εγγυώμενη τον καθαρισμό χωρίς ρητό περιτύλιγμα `eval` σε κάθε διαδρομή
- `die` - μία από τις διαδρομές εξόδου που πυροδοτεί αναβληθέντα μπλοκ· διαβάστε την μαζί με την `defer` όταν σχεδιάζετε καθαρισμό

- `return` - άλλη μια διαδρομή εξόδου· μια `defer` που προγραμματίστηκε πριν από ένα `return` εκτελείται πριν συνεχίσει ο καλών
- `local` - το άλλο ιδίωμα για αποκατάσταση κατάστασης με εμβέλεια· επιλέξτε `local` για δυναμική εμβέλεια μιας μόνο μεταβλητής, `defer` για αυθαίρετες ενέργειες καθαρισμού
- `$@` - η τρέχουσα εξαίρεση· ορατή μέσα σε ένα μπλοκ `defer` που εκτελείται στη διαδρομή ξετυλίγματος εξαίρεσης
- `$!` - το errno συστήματος· εύκολα καταστρέφεται από κλήσεις συστήματος καθαρισμού, εφαρμόστε `local` μέσα στο αναβληθέν μπλοκ αν ο καλών χρειάζεται να διατηρηθεί

Διάφορα

defined

Έλεγχος εάν μια τιμή, μεταβλητή ή υπορουτίνα είναι ορισμένη.

Η `defined` επιστρέφει αληθές όταν η `EXPR` κρατά οποιαδήποτε τιμή εκτός της απροσδιόριστης τιμής `undef`, και ψευδές όταν κρατά `undef`. Είναι ο μοναδικός τρόπος να διακρίνεις την `undef` από τις άλλες ψευδείς τιμές (`0`, `""`, `"0"`), κάτι που ένας απλός λογικός έλεγχος δεν μπορεί.

Σύνοψη

```
defined EXPR
defined
defined &SUBROUTINE
```

Τι επιστρέφεται

Μια λογική τιμή: αληθές αν η `EXPR` είναι ορισμένη, ψευδές αν είναι `undef`.

Χωρίς όρισμα, η `defined` λειτουργεί στην `$_`.

Το αποτέλεσμα είναι ευαίσθητο στο περιβάλλον μόνο με τη συνήθη λογική έννοια· η `defined` δεν διαδίδει περιβάλλον λίστας στην `EXPR`.

Γιατί υπάρχει, και γιατί ένας απλός λογικός έλεγχος δεν αρκεί

Πολλές λειτουργίες επιστρέφουν `undef` για να σηματοδοτήσουν αποτυχία, τέλος αρχείου, σφάλμα συστήματος, μη αρχικοποιημένη μεταβλητή ή κάποια άλλη εξαιρετική συνθήκη. Η `defined` σας επιτρέπει να διακρίνετε αυτή τη φρουρά από ένα νόμιμο αποτέλεσμα. Ένας απλός λογικός έλεγχος αντιμετωπίζει τα `undef`, `0`, `""` και `"0"` όμοια - και τα τέσσερα είναι ψευδή - οπότε δεν μπορείτε να χρησιμοποιήσετε την αληθοτιμία για να διακρίνετε «τίποτα δεν επιστράφηκε» από «επιστράφηκε μηδέν»:

```
my $n = some_count();
if ($n)         { ... }   # false for 0 AND for undef
if (defined $n) { ... }   # false only for undef
```

Σημειώστε ότι η undef είναι η ίδια έγκυρη βαθμωτή τιμή. Η παρουσία της δεν σημαίνει *απαραίτητα* ότι κάτι πήγε στραβά: η pop @empty επιστρέφει undef, αλλά το ίδιο κάνει και η pop @array όταν το στοιχείο που απομακρύνθηκε ήταν το ίδιο undef. Η defined σας λέει ότι η τιμή είναι undef· δεν σας λέει γιατί.

Προεπιλεγμένο όρισμα: $_

Χωρίς όρισμα, η defined ελέγχει την $_:

```
while (<$fh>) {
    next unless defined;          # same as: defined $_
    chomp;
    ...
}
```

defined &subroutine

Η defined &func ρωτά αν η υπορουτίνα func έχει οριστεί ποτέ. Μια προδήλωση (sub func;) **δεν** την κάνει ορισμένη· ένα πραγματικό σώμα την κάνει:

```
sub greet { "hello" }
print defined &greet   ? "yes\n" : "no\n";   # yes
print defined &missing ? "yes\n" : "no\n";   # no
```

Μια υπορουτίνα που δεν είναι ορισμένη μπορεί παρ” όλα αυτά να είναι *καλέσιμη*: η μέθοδος AUTOLOAD του πακέτου μπορεί να την δημιουργήσει στην πρώτη κλήση. Η defined &func αντικατοπτρίζει την τρέχουσα κατάσταση του πίνακα συμβόλων, όχι το τι μπορεί να παραγάγει ένα μελλοντικό AUTOLOAD. Δείτε το perlsub.

Η τιμή επιστροφής δεν επηρεάζεται από προδηλώσεις.

Στοιχεία πίνακα κατακερματισμού: defined έναντι exists

Σε ένα στοιχείο πίνακα κατακερματισμού, η defined ρωτά για την **τιμή**, όχι το κλειδί:

```
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
```

Χρησιμοποιήστε την exists για να ρωτήσετε «βρίσκεται αυτό το κλειδί στον πίνακα κατακερματισμού;» και την defined για να ρωτήσετε «κρατά αυτή η θέση μια χρησιμοποιήσιμη τιμή;». Οι δύο διαφέρουν όποτε μια τιμή έχει εκχωρηθεί ρητά ως undef ή έχει αφεθεί undef εκ κατασκευής (π.χ. $h{b} = undef, ή @h{qw(a b c)} = (1)).

Η ίδια διάκριση ισχύει για τα στοιχεία πίνακα: η `exists $a[7]` αναφέρει αν ο πίνακας είναι αρκετά μεγάλος και η θέση έχει εκχωρηθεί, ενώ η `defined $a[7]` αναφέρει αν αυτή η θέση κρατά αυτή τη στιγμή μη-undef τιμή.

Παραδείγματα

Το κοινό ιδίωμα pop-μέχρι-άδειασμα - σταματά μόνο όταν ο πίνακας εξαντληθεί, όχι σε νόμιμο ψευδές στοιχείο:

```
while (defined(my $val = pop @ary)) {
    process($val);
}
```

Έλεγχος κλήσης συστήματος για αποτυχία:

```
my $target = readlink $sym;
die "can't readlink $sym: $!" unless defined $target;
```

Κλήση ενός coderef μόνο αν παρασχέθηκε ένα:

```
sub dispatch {
    my ($cb, @args) = @_;
    return defined &$cb ? $cb->(@args) : die "no callback";
}
```

Παροχή προεπιλογής χωρίς να καταστρέφεται νόμιμο `0` ή `""`:

```
$debugging = 0 unless defined $debugging;
```

Η σύγχρονη μορφή είναι ο τελεστής defined-or `//`, ο οποίος εκτελεί ακριβώς αυτόν τον έλεγχο ενσωματωμένα:

```
$debugging //= 0;
my $name = $user // "anonymous";
```

Σύλληψη regex: το ότι η $1 είναι ορισμένη σας λέει ότι η ομάδα σύλληψης συμμετείχε στην αντιστοιχία, ακόμη και αν ταίριαξε με την κενή συμβολοσειρά:

```
"ab" =~ /a(.*)b/;
print defined $1 ? "matched '$1'\n" : "no match\n";   # matched ''
```

Συγκεντρώσεις: τα `defined @array`, `defined %hash` δεν υποστηρίζονται πλέον

Παλαιότερα τα `defined @array` και `defined %hash` ανέφεραν αν είχε ποτέ δεσμευτεί μνήμη για τη συγκέντρωση - μια λεπτομέρεια υλοποίησης που σπάνια ήταν αυτό που οι καλούντες πραγματικά ήθελαν. Η σύγχρονη Perl απορρίπτει αυτές τις μορφές. Ελέγξτε το μέγεθος της συγκέντρωσης αντί αυτού:

```
if (@array) { print "array has elements\n" }
if (%hash)  { print "hash has members\n"   }
```

Σε λογικό περιβάλλον ένας πίνακας αποτιμάται στο πλήθος των στοιχείων του και ένας πίνακας κατακερματισμού αποτιμάται σε αληθή τιμή αν και μόνο αν έχει κλειδιά. Αυτός είναι ο έλεγχος που σχεδόν πάντα θέλατε.

Ο έλεγχος *ενός στοιχείου* μιας συγκέντρωσης παραμένει αποδεκτός και είναι η συνήθης χρήση: `defined $array[$i]`, `defined $hash{$key}`.

Υπερβολική χρήση

Η `defined` χρησιμοποιείται συχνά υπερβολικά. Το μηδέν και η κενή συμβολοσειρά είναι ορισμένες τιμές. Μια αντιστοιχία regex που συλλαμβάνει μηδέν χαρακτήρες θέτει παρ" όλα αυτά την `$1` σε ορισμένη κενή συμβολοσειρά - η αντιστοιχία δεν απέτυχε· ταίριαξε κάτι που έτυχε να έχει μήκος μηδέν. Όταν μια συνάρτηση επιστρέφει `undef`, παραδέχεται ότι δεν μπορεί να σας δώσει ουσιαστική απάντηση. Καταφεύγετε στην `defined` όταν αμφισβητείτε την ακεραιότητα μιας τιμής. Για «είναι μηδέν;» ή «είναι κενό;» μια απλή σύγκριση με `0` ή `""` είναι αυτό που χρειάζεστε.

Οριακές περιπτώσεις

- **Σκέτη `defined`** χωρίς όρισμα ελέγχει την `$_`. Μέσα σε `map`, `grep` ή βρόχο `while (<$fh>)` αυτή είναι η ιδιωματική μορφή.
- **`defined(&func)` έναντι `defined(&{$ref})`**: και τα δύο λειτουργούν. Το δεύτερο αποαναφέρει μια αναφορά κώδικα και ρωτά αν η θέση που αναφέρεται κρατά αυτή τη στιγμή ορισμένο σώμα υπορουτίνας.
- **Η `defined $hash{$key}` δεν αυτο-δημιουργεί το κλειδί** - η `defined` είναι περιβάλλον rvalue και αφήνει τον πίνακα κατακερματισμού αμετάβλητο. Αντιπαραβάλετε με το `$hash{$key}{inner} = 1`, το οποίο αυτο-δημιουργεί.
- **Αλυσιδωτή πρόσβαση πίνακα κατακερματισμού/πίνακα**: η `defined $h{a}{b}` αυτο-δημιουργεί την `$h{a}` σε μια κενή αναφορά πίνακα κατακερματισμού για να εκτελέσει την εσωτερική αναζήτηση, ακόμη και αν η εξωτερική `defined` διαβάζει το αποτέλεσμα. Αυτή είναι συνέπεια του πώς αποτιμάται η αλυσίδα αποαναφοράς, όχι της ίδιας της `defined`. Για να εξετάσετε εμφωλευμένη δομή χωρίς αυτο-δημιουργία, ελέγξτε κάθε επίπεδο πρώτα με την `exists`.
- **`defined $&`, `defined $1`**: μετά από επιτυχή αντιστοιχία αυτές είναι ορισμένες· μετά από αποτυχημένη αντιστοιχία διατηρούν όποια τιμή είχαν από την προηγούμενη επιτυχή αντιστοιχία στην ίδια εμβέλεια. Η `defined` από μόνη της δεν είναι ασφαλής έλεγχος για «πέτυχε η αντιστοιχία;» - ελέγξτε την τιμή επιστροφής του τελεστή αντιστοίχισης αντί αυτού.
- **Η `defined` δεν είναι lvalue**. Το `defined $x = $y` είναι συντακτικό σφάλμα· βάλτε την εκχώρηση σε παρενθέσεις: `defined($x = $y)`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `undef` - η τιμή έναντι της οποίας ελέγχει η `defined`, και ο τελεστής που την παράγει ή την καθαρίζει
- `exists` - ρωτά αν ένα κλειδί πίνακα κατακερματισμού ή δείκτης πίνακα είναι παρών, ανεξάρτητα από το αν η τιμή του είναι `undef`
- `ref` - όταν το ερώτημα δεν είναι «είναι ορισμένο» αλλά «τι είδους αναφορά είναι αυτή»
- `// και //=` - τελεστής defined-or· η σύγχρονη ενσωματωμένη μορφή του `defined($x) ? $x : $default`
- `$_` - το προεπιλεγμένο όρισμα όταν η `defined` καλείται χωρίς έκφραση

Hashes

delete

Αφαιρεί τα κατονομασμένα ζεύγη κλειδιού-τιμής από έναν πίνακα κατακερματισμού, ή στοιχεία από έναν πίνακα, και επιστρέφει αυτό που αφαιρέθηκε.

Η `delete` φτάνει μέσα σε ένα σύνθετο και αφαιρεί τις καθορισμένες εγγραφές οριστικά - όχι απλώς θέτοντάς τες σε `undef`. Μετά την `delete`, η `exists` στο ίδιο κλειδί ή ευρετήριο επιστρέφει ψευδές, η `keys` δεν το παραθέτει πλέον, και η επανάληψη στον πίνακα κατακερματισμού με `each` το παρακάμπτει. Το όρισμα πρέπει να είναι στοιχείο ή τομή ενός πίνακα κατακερματισμού ή πίνακα· η τελική του λειτουργία επιλέγει τι θα αφαιρεθεί, και οτιδήποτε προηγείται αυτής της λειτουργίας (αποαναφορές, κλήσεις μεθόδων) είναι απλώς πλοήγηση.

Σύνοψη

```
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
```

Τι επιστρέφεται

Σε περιβάλλον λίστας, η `delete` επιστρέφει την τιμή ή τις τιμές που αφαιρέθηκαν, με τη σειρά των ορισμάτων. Όρισμα τομής του οποίου το κλειδί ή ευρετήριο δεν υπήρχε αποδίδει `undef` στην αντίστοιχη θέση, οπότε το μήκος της επιστρεφόμενης λίστας ταιριάζει με το μήκος της λίστας κλειδιών/ευρετηρίων.

Σε βαθμωτό περιβάλλον, η `delete` επιστρέφει την τιμή του τελευταίου στοιχείου που αφαιρέθηκε, ή `undef` αν το κλειδί/ευρετήριο δεν ήταν παρόν.

Η μορφή τομής κλειδιών/τιμών της 5.20+ `delete %hash{LIST}` επιστρέφει λίστα από εναλλασσόμενα ζεύγη κλειδιού/τιμής - δύο στοιχεία για κάθε διαγραμμένη εγγραφή - γεγονός που την καθιστά τον ιδιωματικό τρόπο για να αποσπάσετε ένα κατονομασμένο σύνολο εγγραφών και να κρατήσετε και τις δύο πλευρές.

```
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)
```

Καθολική κατάσταση που επηρεάζει

- **%ENV** - η `delete $ENV{VAR}` καθαρίζει τη μεταβλητή περιβάλλοντος για την τρέχουσα διεργασία και τα μελλοντικά της παιδιά. Η επίδραση είναι πραγματική, όχι μόνο στον πίνακα κατακερματισμού από την πλευρά της Perl· οι κλήσεις συστήματος βλέπουν την αλλαγή.
- **tied σύνθετα** - η `delete` δρομολογείται στη μέθοδο `DELETE` του πακέτου που κάνει tie. Η τιμή επιστροφής είναι ό,τι επιλέξει να επιστρέψει η υλοποίηση· κάποιοι tied πίνακες κατακερματισμού επιστρέφουν τη διαγραμμένη τιμή, κάποιοι όχι.
- **$_** - δεν επηρεάζεται. Η `delete` δεν έχει προεπιλογή· η `EXPR` είναι υποχρεωτική.

Παραδείγματα

Αφαίρεση ενός κλειδιού, επιθεώρηση του τι υπήρχε:

```
my %cfg = (host => "db1", port => 5432, user => "alice");
my $old_host = delete $cfg{host};
# $old_host is "db1"; exists $cfg{host} is false
```

Απελευθέρωση μνήμης με αφαίρεση πολλών εγγραφών σε μία κλήση (τομή πίνακα κατακερματισμού):

```
delete @cache{@stale_keys};
```

Απόσπαση ενός συνόλου κατονομασμένων εγγραφών διατηρώντας τα ζεύγη (τομή κλειδιών/τιμών 5.20+):

```
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)
```

Διαγραφή μέσα από εμφωλευμένη δομή. Η αρχική πλοήγηση μπορεί να είναι αυθαίρετη - μόνο η τελική λειτουργία αποφασίζει τι αφαιρείται:

```
my $tree = { users => { alice => { roles => [qw(admin ro)] } } };
delete $tree->{users}{alice}{roles}[0];
# roles is now (undef, "ro")
delete $tree->{users}{alice};
# the whole user record is gone
```

Αποφύγετε αυτο-δημιουργία στην πλευρά ανάγνωσης ενώ συνεχίζετε να αφαιρείτε στην πλευρά εγγραφής. Η `delete` δεν αυτο-δημιουργεί ποτέ τους ενδιάμεσους πίνακες κατακερματισμού από τους οποίους περνά:

```
delete $h{outer}{inner};    # does not create $h{outer} if absent
```

Συγκρίνετε με την `splice` όταν θέλετε τα ευρετήρια να ανανεωθούν σε αρίθμηση. Η `delete` σε στοιχείο πίνακα αφήνει τρύπα· η `splice` κλείνει την τρύπα:

```
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
```

Οριακές περιπτώσεις

- **`delete` έναντι ανάθεσης `undef`**: η `$h{k} = undef` αφήνει το k στον πίνακα κατακερματισμού με απροσδιόριστη τιμή· η `exists $h{k}` είναι ακόμη αληθής. Η `delete $h{k}` αφαιρεί το κλειδί εξ ολοκλήρου.
- **Απόν κλειδί σε βαθμωτό περιβάλλον**: η `delete $h{nope}` επιστρέφει `undef`. Αυτό είναι αδιάκριτο από τη διαγραφή κλειδιού του οποίου η τιμή ήταν `undef`· χρησιμοποιήστε πρώτα την `exists` αν η διάκριση έχει σημασία.
- **`each` + `delete` μέσα στον βρόχο**: ασφαλές όταν διαγράφετε μόνο το τρέχον κλειδί (το κλειδί που μόλις επέστρεψε η `each`). Η διαγραφή *άλλων* κλειδιών κατά την επανάληψη μπορεί να παραλείψει εγγραφές ή να τις ξαναεπισκεφθεί - ο επαναλήπτης πίνακα κατακερματισμού της Perl δεν είναι σταθερός υπό δομική αλλαγή. Η ασφαλής αναδιατύπωση είναι να συλλέξετε πρώτα τα κλειδιά:

  ```
  for my $k (grep { $h{$_} < 0 } keys %h) {
      delete $h{$k};
  }
  ```

- **Οι διαγραφές στο τέλος πίνακα συρρικνώνουν τον πίνακα**: αν τα διαγραμμένα στοιχεία βρίσκονται στο πάνω άκρο, το `$#array` πέφτει στο υψηλότερο ευρετήριο που ακόμη ελέγχεται ως αληθές υπό `exists`. Οι τρύπες στη μέση παραμένουν τρύπες.
- **Η `delete` σε πίνακα αποθαρρύνεται**: η έννοια του «απόντος ευρετηρίου» δεν είναι συνεκτική για πίνακες - τα ευρετήρια είναι θεσιακά, όχι κατονομασμένα. Αφήστε τη συρρίκνωση πίνακα στις `shift`, `pop` ή `splice`. Η `WARNING` στο upstream POD επαναλαμβάνεται: καταφύγετε στη `delete` σε πίνακα μόνο όταν ξέρετε ότι θέλετε τρύπα.
- **`delete local $h{k}`**: αφαιρεί την εγγραφή για το περικλείον μπλοκ και την επαναφέρει στην έξοδο από την εμβέλεια. Χρήσιμη για προσωρινή απόκρυψη κλειδιών διαμόρφωσης από κώδικα που καλείται μέσα στο μπλοκ χωρίς να χαθεί η αρχική τιμή:

  ```
  {
      local $ENV{LANG};         # still present, but undef
      delete local $ENV{PATH};  # gone for this block only
  ```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    run_thing();
}   # both restored to original values here
```

- **Tied πίνακας κατακερματισμού, tied πίνακας**: η `delete` καλεί την `DELETE`· η τιμή επιστροφής είναι ό,τι παρέχει η υλοποίηση tie. Ένας tied πίνακας κατακερματισμού DBM διαγράφει την εγγραφή στον δίσκο.
- **Στόχος μόνο για ανάγνωση**: η `delete` σε στοιχείο μόνο για ανάγνωση εκπέμπει σφάλμα `Modification of a read-only value attempted`.
- **Η έκφραση πρέπει να τελειώνει σε στοιχείο ή τομή**: η `delete $h{k} + 1` είναι σφάλμα μεταγλώττισης· η `delete` έχει χαμηλότερη προτεραιότητα από την αριθμητική και αναμένει αναζήτηση σύνθετου τύπου lvalue, όχι υπολογισμένη τιμή.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `exists` - ο αντίστοιχος ερώτησης· χρησιμοποιήστε την πριν την `delete` όταν χρειάζεται να ξεχωρίσετε «δεν υπήρξε ποτέ εκεί» από «ήταν εκεί με τιμή `undef`»
- `each` - επαναλήπτης πίνακα κατακερματισμού· ασφαλής συνδυασμός με τη `delete` μόνο όταν διαγράφετε το τρέχον κλειδί
- `keys` - στιγμιότυπο των κλειδιών του πίνακα κατακερματισμού· ο συνήθης τρόπος δόμησης λίστας διαγραφής που δεν θα αλληλεπιδράσει με την κατάσταση του επαναλήπτη
- `values` - συνοδευτική της `keys` όταν θέλετε το σύνολο των διαγραμμένων τιμών χωρίς τα κλειδιά
- `splice` - το σωστό εργαλείο όταν θέλετε να αφαιρέσετε στοιχεία πίνακα *και* να ξαναριθμήσετε τα εναπομείναντα ευρετήρια
- `undef` - εκκαθαρίζει ολόκληρο σύνθετο (`undef %h`, `undef @a`) ή ένα μεμονωμένο βαθμωτό· ο καθιερωμένος τρόπος για να αδειάσετε πίνακα κατακερματισμού ή πίνακα συνολικά αντί να διαγράφετε κάθε κλειδί

I/O · Ροή ελέγχου

die

Εγείρει μια εξαίρεση.

Η `die` εκτοξεύει μια εξαίρεση φτιαγμένη από τη `LIST`. Αν η εξαίρεση συμβεί μέσα σε `eval`, η `eval` τερματίζει με `undef` και η `$@` κρατά την τιμή της εξαίρεσης. Αν δεν την πιάσει κανείς, η Perl τυπώνει το μήνυμα στο `STDERR` και ο διερμηνέας εξέρχεται με μη μηδενική κατάσταση. Η εξαίρεση μπορεί να είναι απλή συμβολοσειρά, λίστα που μετατρέπεται σε συμβολοσειρά, ή αναφορά - τυπικά ένα blessed αντικείμενο που κουβαλά δομημένη κατάσταση σφάλματος.

Σύνοψη

```
die LIST
die $message
die $exception_object
die                          # re-raise / propagate current $@
```

Τι επιστρέφεται

Η die δεν επιστρέφει. Ο έλεγχος μεταφέρεται στην πλησιέστερη δυναμικά περικλείουσα eval (η οποία τότε επιστρέφει undef σε βαθμωτό περιβάλλον ή την κενή λίστα σε περιβάλλον λίστας), ή - αν δεν υπάρχει περικλείουσα eval - ο διερμηνέας ξετυλίγεται, εκτελεί τους χειριστές END και DESTROY και εξέρχεται.

Η ίδια η τιμή της εξαίρεσης προσγειώνεται στην $@:

- **Συμβολοσειριακές εξαιρέσεις** - το μετατραπέν σε συμβολοσειρά μήνυμα, συμπεριλαμβανομένου τυχόν προσαρτημένου at FILE line N.\n (βλ. παρακάτω).
- **Εξαιρέσεις με τιμή αναφοράς** - η ίδια η αναφορά, αμετάβλητη. Αυτός είναι ο ιδιωματικός τρόπος εκτόξευσης ενός blessed αντικειμένου εξαίρεσης.

Σε μη πιασμένη έξοδο, ο κωδικός εξόδου παράγεται από τις $! και $?:

```
exit $! if $!;               # errno
exit $? >> 8 if $? >> 8;     # child exit status
exit 255;                    # last resort
```

Βασιστείτε μόνο στο ότι ο κωδικός εξόδου θα είναι μη μηδενικός· η ακριβής τιμή είναι μια κατά το δυνατόν βέλτιστη κωδικοποίηση όποιας καθολικής κατάστασης σφάλματος ήταν ενεργή τη στιγμή της εκτόξευσης.

Καθολική κατάσταση που επηρεάζει

- $@ - εγγράφεται με την εξαίρεση κατά το ξετύλιγμα σε μια eval, διαβάζεται από τη μορφή της die χωρίς ορίσματα για τη διάδοση υπάρχουσας εξαίρεσης, και διαβάζεται από τον εκτυπωτή μη πιασμένων εξαιρέσεων.
- $! - διαβάζεται κατά την παραγωγή κωδικού εξόδου για μη πιασμένη εξαίρεση, και συνηθίζεται να παρεμβάλλεται μέσα στο ίδιο το μήνυμα (die "open $f: $!").
- $? - διαβάζεται κατά την παραγωγή κωδικού εξόδου αν η $! είναι μηδέν. Οι χειριστές END και DESTROY μπορούν ακόμη να τη μεταβάλουν πριν η διεργασία πραγματικά εξέλθει.
- $. - ο τρέχων αριθμός γραμμής εισόδου, προσαρτάται σε συμβολοσειριακές εξαιρέσεις που στερούνται τελικού newline.
- $SIG{__DIE__} - αν τεθεί, ο χειριστής εκτελείται *πριν* διαδοθεί η εξαίρεση, με την εξαίρεση ως όρισμά του. Ο χειριστής μπορεί να αντικαταστήσει την εξαίρεση καλώντας ξανά την die με διαφορετική τιμή.
- $^S - αναγνώσιμη από τους χειριστές $SIG{__DIE__} ώστε να διακρίνουν αν η die συμβαίνει μέσα σε eval.

Ο κανόνας του τελικού newline

Το αν η `die` προσαρτά πληροφορία τοποθεσίας εξαρτάται από έναν χαρακτήρα στο τέλος του μηνύματος:

- **Μήνυμα που τελειώνει σε `"\n"`** - χρησιμοποιείται αυτούσιο. Δεν προσαρτάται τίποτα. Χρησιμοποιήστε αυτή τη μορφή όταν το μήνυμα είναι ήδη ένα πλήρες, προοριζόμενο για τον χρήστη διαγνωστικό.
- **Μήνυμα που δεν τελειώνει σε `"\n"`** - η Perl προσαρτά `" at FILE line N"`, και αν διαβάζεται κάποιο αρχείο, `", <HANDLE> line M"`, και στη συνέχεια τελική `"."` και newline. Χρησιμοποιήστε αυτή τη μορφή κατά την ανάπτυξη, ή όποτε ο εντοπισμός της κλήσης της `die` στον πηγαίο κώδικα είναι χρησιμότερος από ένα τακτοποιημένο μήνυμα.

```
die "bad config\n";                 # "bad config\n"
die "bad config";                   # "bad config at script.pl line␣
 ↪42.\n"
```

Ένα συνηθισμένο ιδίωμα είναι να προσαρτάται `", stopped"` ώστε το επίθεμα τοποθεσίας να διαβάζεται φυσικά:

```
die "/etc/games is no good, stopped";
# /etc/games is no good, stopped at canasta line 123.
```

Στις εξαιρέσεις-αναφορές δεν προσαρτάται ποτέ πληροφορία τοποθεσίας - η μετατροπή ενός αντικειμένου σε συμβολοσειρά για επιθεώρηση θα ακύρωνε το νόημα της εκτόξευσης αντικειμένου. Αν θέλετε αρχείο και γραμμή μέσα σε μια δομημένη εξαίρεση, αποτυπώστε τα κατά τη στιγμή της κατασκευής (`__FILE__`, `__LINE__`, ή `caller`).

Εξαιρέσεις με τιμή αναφοράς

Η διαβίβαση μιας αναφοράς - τυπικά ενός blessed αντικειμένου - επιτρέπει στην `$@` να μεταφέρει δομημένη κατάσταση αντί για μια συμβολοσειρά που θα πρέπει να αναλύσετε εκ των υστέρων:

```
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
    }
}
```

Υπερφορτώστε τη μετατροπή σε συμβολοσειρά στις κλάσεις εξαιρέσεων ώστε μια μη πιασμένη εξαίρεση να εξακολουθεί να τυπώνει χρήσιμο μήνυμα:

```
use overload '""' => sub { "[MyApp::Error ".$_[0]->code."]\n" };
```

Η μετατραπείσα σε συμβολοσειρά μορφή πρέπει να τελειώνει σε newline και να μην είναι κενή, σε αντιστοιχία με τη μεταχείριση των συμβολοσειριακών εξαιρέσεων.

Επειδή η `$@` είναι καθολική, αντιγράψτε την σε μια λεκτική μεταβλητή πριν την επιθεωρήσετε - οποιαδήποτε κλήση μεθόδου, `ref`, ή `isa` εκτελέσετε κατά τον χειρισμό μπορεί η ίδια εσωτερικά να καλέσει `eval` και να αλλοιώσει την `$@`:

```
if (my $err = $@) { ... }       # safe
if ($@) { $@->method; ... }     # unsafe: method may reset $@
```

Διάδοση: die χωρίς όρισμα (ή με κενή συμβολοσειρά)

Όταν η `LIST` είναι κενή ή μετατρέπεται σε συμβολοσειρά `""`, η `die` δεν εκτοξεύει νέα εξαίρεση - επανεγείρει ό,τι βρίσκεται στην `$@`:

- Αν η `$@` κρατά **συμβολοσειρά**, η `die` προσαρτά `"\t...propagated at FILE line N.\n"` και επανεκτοξεύει. Αυτός είναι ο καθιερωμένος τρόπος για να αφεθεί μια εξαίρεση να συνεχίσει στη στοίβα κλήσεων μετά από έναν έλεγχο μερικής αντιστοιχίας:

  ```
  eval { risky() };
  die unless $@ =~ /Expected exception/;
  ```

- Αν η `$@` κρατά **αναφορά αντικειμένου με μέθοδο PROPAGATE**, η μέθοδος αυτή καλείται ως `$@ = eval { $@->PROPAGATE(__FILE__, __LINE__) }`, και η τιμή επιστροφής αντικαθιστά την `$@` πριν την επανεκτόξευση. Υλοποιήστε την `PROPAGATE` σε μια κλάση εξαίρεσης όταν θέλετε η εξαίρεση να καταγράφει την αλυσίδα των επανεγέρσεων.
- Αν η `$@` κρατά αναφορά αντικειμένου **χωρίς** `PROPAGATE`, η αναφορά επανεκτοξεύεται αμετάβλητη.
- Αν η `$@` είναι επίσης κενή, χρησιμοποιείται η συμβολοσειρά `"Died"`.

Ο γάντζος $SIG{__DIE__}

Θέτοντας την `$SIG{__DIE__}` εγκαθίσταται χειριστής που εκτελείται όταν πυροδοτείται η `die`, με την εξαίρεση ως όρισμά του. Ο χειριστής μπορεί να επιθεωρήσει ή να αντικαταστήσει την εξαίρεση καλώντας ξανά την `die` με διαφορετική τιμή. Εκτελείται **ακόμη και μέσα σε `eval`**, που είναι ο συνηθισμένος λόγος για τον οποίο τον θέλει κανείς εκτός εμποδίου:

```
local $SIG{__DIE__} = sub {
    die @_ if $^S;              # inside eval: do nothing special
    log_fatal(@_);              # outside eval: log before we exit
};
```

Η $^S είναι undef κατά τη μεταγλώττιση, αληθής μέσα σε eval, και ψευδής στο ανώτατο επίπεδο - χρησιμοποιήστε την για να φιλτράρετε εργασία του χειριστή που έχει νόημα μόνο για μη πιασμένες εξαιρέσεις.

Παραδείγματα

Εγκατάλειψη με καθαρό μήνυμα (παρατηρήστε ότι το τελικό newline καταστέλλει την προσάρτηση αρχείου/γραμμής):

```
chdir '/var/spool' or die "can't cd to spool: $!\n";
```

Καλέστε την die χωρίς newline κατά την ανάπτυξη ώστε να προσαρτηθεί η τοποθεσία:

```
die "unexpected empty record";
# unexpected empty record at parse.pl line 87.
```

Πιάσιμο συμβολοσειριακής εξαίρεσης:

```
eval {
    parse($input);
};
if ($@) {
    warn "parse failed: $@";    # $@ already ends in "\n" if the
                                # die'd message did
}
```

Εκτόξευση και πιάσιμο blessed αντικειμένου εξαίρεσης:

```
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
    }
}
```

Υπό συνθήκη διάδοση - πιάστε μια κλάση, αφήστε τις υπόλοιπες να περάσουν:

```
eval { work() };
die unless $@ =~ /^recoverable:/;
recover();
```

Οριακές περιπτώσεις

- **Λίστα με δύο ή περισσότερα στοιχεία** μετατρέπεται σε συμβολοσειρές και συνενώνεται για να σχηματίσει το μήνυμα: `die "read failed on ", $file, ": $!"`. Ο κανόνας του newline ισχύει για το συνενωμένο αποτέλεσμα.

- **Κενή λίστα** (`die;` ή `die "";`) πυροδοτεί τη διαδρομή διάδοσης που περιγράφηκε παραπάνω, όχι νέα εξαίρεση.
- **Αναφορά με τελικό newline μέσα στη μετατραπείσα σε συμβολοσειρά μορφή της** είναι αδιάφορη: στην αναφορά δεν προσαρτάται ποτέ τοποθεσία, ανεξάρτητα από το τι παράγει η υπερφόρτωση `""` της.
- **`die` μέσα σε `DESTROY` κατά την καθολική καταστροφή** μπορεί ακόμη να πυροδοτήσει την `$SIG{__DIE__}` αλλά ενδέχεται να μη διαδοθεί όπως αναμένεται - ο διερμηνέας ήδη ξετυλίγεται. Κρατήστε τους καταστροφείς αμυντικούς.
- **Μη πιασμένη `die` κατά τη διάρκεια μπλοκ `END`** εκτελεί τα υπόλοιπα μπλοκ `END` αλλά θέτει την κατάσταση εξόδου σύμφωνα με τον κανόνα `$!`/`$?` παραπάνω.
- **Εμφωλευμένη `eval` και `$SIG{__DIE__}`**: ο χειριστής εκτελείται για *κάθε* die, συμπεριλαμβανομένων αυτών που πιάνονται από εσωτερική `eval`. Προστατευθείτε με τη `$^S` αν ο χειριστής έχει παρενέργειες.
- **Επανεκτόξευση της τρέχουσας `$@`**: προτιμήστε γυμνό `die;` έναντι `die $@;`. Η γυμνή μορφή πυροδοτεί τη μηχανική διάδοσης (επίθεμα `...propagated`, γάντζος `PROPAGATE`)· το `die $@` όχι - μεταχειρίζεται την `$@` ως νέα τιμή εξαίρεσης.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `eval` - η μόνη κατασκευή που πιάνει μια `die`· η μορφή της με μπλοκ παγιδεύει εξαιρέσεις στην `$@`
- `warn` - ίδιοι κανόνες μορφοποίησης (τελικό newline καταστέλλει την τοποθεσία), αλλά γράφει στο `STDERR` και δεν ξετυλίγει
- `caller` - αποτύπωση αρχείου και γραμμής τη στιγμή της εκτόξευσης κατά την κατασκευή δομημένου αντικειμένου εξαίρεσης
- `exit` - χρησιμοποιήστε την όταν θέλετε συγκεκριμένο κωδικό εξόδου χωρίς τη μηχανική χειρισμού εξαιρέσεων
- `$@` - όπου προσγειώνεται η εξαίρεση αφού την πιάσει μια `eval`, και η πηγή από την οποία η `die` χωρίς ορίσματα επανεγείρει
- `$SIG{__DIE__}` - γάντζος πριν την εκτόξευση που μπορεί να επιθεωρήσει ή να ξαναγράψει την εξαίρεση
- `$^S` - λέει σε έναν χειριστή `$SIG{__DIE__}` αν εκτελείται μέσα σε `eval`

Ροή ελέγχου · Αρθρώματα

do

Εκτελεί ένα μπλοκ κώδικα ή τρέχει ένα αρχείο πηγαίου κώδικα Perl σαν να ήταν τμήμα του τρέχοντος προγράμματος.

Η `do` φοράει δύο άσχετα καπέλα που μοιράζονται μόνο μια δεσμευμένη λέξη. Το `do BLOCK` ομαδοποιεί δηλώσεις σε μία ενιαία έκφραση και επιστρέφει την τιμή της

τελευταίας - συντακτικό εργαλείο, όχι συνάρτηση. Το do EXPR μεταχειρίζεται τη συμβολοσειρά ως όνομα αρχείου και μεταγλωττίζει-και-εκτελεί το αρχείο σε χρόνο εκτέλεσης. Διαβάστε όποια ενότητα ταιριάζει στη μορφή που βλέπετε· η ανάμειξή τους είναι η συνηθέστερη πηγή σύγχυσης γύρω από αυτή τη δεσμευμένη λέξη.

Σύνοψη

```
do BLOCK                          # group statements, yield last value
do EXPR                           # load and run a Perl file
do { ... } while CONDITION;       # post-test loop idiom
```

Τι επιστρέφεται

- do BLOCK - η τιμή της τελευταίας έκφρασης που αποτιμήθηκε στο μπλοκ, σε όποιο περιβάλλον επιβάλλει η περιβάλλουσα έκφραση.
- do EXPR - η τιμή της τελευταίας έκφρασης που αποτιμήθηκε στο αρχείο σε επιτυχία· undef σε αποτυχία, με την $@ τεθειμένη στο σφάλμα μεταγλώττισης ή την $! τεθειμένη στο σφάλμα I/O.

do BLOCK - ομαδοποίηση δηλώσεων σε μία έκφραση

Το do BLOCK δεν είναι συνάρτηση. Είναι συντακτική κατασκευή που μετατρέπει μια ακολουθία δηλώσεων οριοθετημένη από αγκύλες σε έναν ενιαίο όρο, του οποίου η τιμή είναι η τιμή της τελευταίας δήλωσης. Καταφύγετε σε αυτή όταν η γλώσσα ζητά έκφραση αλλά έχετε αρκετές δηλώσεις να εκτελέσετε:

```
my $x = do {
    my $tmp = fetch_raw();
    $tmp =~ s/\s+\z//;
    lc $tmp;
};                                # $x = cleaned, lower-cased value
```

Όταν το do BLOCK τροποποιείται από επόμενο while ή until, το μπλοκ εκτελείται **μία φορά πριν** ελεγχθεί η συνθήκη - αυτό είναι το καθιερωμένο ιδίωμα βρόχου με μετα-έλεγχο:

```
do {
    $line = <$fh>;
} while defined $line && $line !~ /^END\b/;
```

Σε οποιαδήποτε άλλη δήλωση, οι while και until ως τροποποιητές δήλωσης ελέγχουν τη συνθήκη πρώτα. Η do αποτελεί την εξαίρεση.

Οι next, last, redo δεν λειτουργούν μέσα σε do BLOCK

Ένα do BLOCK **δεν** είναι βρόχος, ακόμη και όταν ακολουθείται από while ή until. Οι δεσμευμένες λέξεις ελέγχου βρόχου δεν το βλέπουν:

```
do {
    last if $done;              # WRONG - not a loop
} while $more;
```

Αν χρειάζεστε έλεγχο βρόχου, τυλίξτε το σώμα σε γυμνό μπλοκ, που *είναι* εμβέλεια προσβάσιμη σε βρόχο:

```
{
    last if $done;
    redo if $retry;
}
```

Ή χρησιμοποιήστε ρητό βρόχο `while`.

`return` μέσα σε γυμνό do BLOCK

Η `return` μέσα σε do BLOCK επιστρέφει από την **περικλείουσα υπορουτίνα**, όχι από το ίδιο το μπλοκ. Αυτό είναι ενίοτε αυτό που θέλετε και ενίοτε σφάλμα - όπως και να έχει, να ξέρετε ποιο από τα δύο γράφετε:

```
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)";
}
```

Για να αποδώσετε μια τιμή από το μπλοκ χωρίς να επιστρέψετε από την υπορουτίνα, αφήστε απλώς την τελευταία έκφραση να είναι η τιμή - αυτό είναι όλο το νόημα του do BLOCK.

do EXPR - φόρτωση και εκτέλεση αρχείου πηγαίου κώδικα Perl

Το do EXPR αποτιμά την EXPR, μεταχειρίζεται το αποτέλεσμα ως όνομα αρχείου, διαβάζει το αρχείο, το μεταγλωττίζει και το εκτελεί στο τρέχον πακέτο. Είναι παρόμοιο, αλλά όχι ακριβώς ίδιο, με

```
eval `cat stat.pl`;
```

Η μορφή do δεν τρέχει εξωτερική διεργασία, διατηρεί το αρχικό όνομα αρχείου για τα μηνύματα σφαλμάτων, και **δεν μπορεί να δει λεκτικές μεταβλητές της περικλείουσας εμβέλειας** - κάτι που το `eval STRING` μπορεί. Όπως η `eval STRING`, το αρχείο επαναναλύεται σε κάθε κλήση, οπότε do FILE μέσα σε βρόχο είναι σχεδόν πάντα λάθος.

Επίλυση διαδρομής και @INC

Το όνομα αρχείου ερμηνεύεται ως εξής:

- Απόλυτες διαδρομές (`/foo/stat.pl`), διαδρομές που ξεκινούν με `./`, και διαδρομές που ξεκινούν με `../` χρησιμοποιούνται ως έχουν.
- Κάθε άλλη σχετική διαδρομή αναζητείται κατά μήκος του `@INC`, και σε επιτυχία η διαδρομή που βρέθηκε καταγράφεται στο `%INC` υπό το κλειδί που διαβιβάσατε.

```
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
```

Από την Perl 5.26 και μετά, το `.` δεν βρίσκεται προεπιλεγμένα στο `@INC`, άρα ένα γυμνό do `'stat.pl'` δεν επαναφέρεται πλέον στον τρέχοντα κατάλογο. Αν το αρχείο δεν βρεθεί, η Perl εκπέμπει την υπόδειξη:

```
do "stat.pl" failed, '.' is no longer in @INC;
did you mean do "./stat.pl"?
```

Διαβιβάστε `./stat.pl` όταν εννοείτε τον τρέχοντα κατάλογο.

Τιμή επιστροφής και αναφορά σφαλμάτων

Το do EXPR έχει τρία αποτελέσματα που πρέπει να διακρίνετε:

Αποτέλεσμα	Επιστροφή	$@	$!
Αρχείο μεταγλωττίστηκε και εκτελέστηκε	τιμή της τελευταίας έκφρασης	""	αμετάβλητη
Αρχείο διαβάστηκε αλλά δεν μεταγλωττίστηκε	undef	σφάλμα μεταγλώττισης	ενδέχεται επίσης να τεθεί
Αρχείο δεν μπόρεσε να διαβαστεί	undef	""	σφάλμα I/O

Επειδή η αποτυχία μεταγλώττισης μπορεί παρεμπιπτόντως να θέσει και την `$!`, **ελέγχετε πάντα πρώτα την `$@`**:

```
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;
    }
}
```

Οι τρεις `warn` αντιστοιχούν στα τρία σχήματα αποτυχίας παραπάνω συν την τέταρτη περίπτωση - το αρχείο εκτελέστηκε αλλά η τελευταία του έκφραση ήταν ψευδής.

Γιατί να μη χρησιμοποιείτε do FILE για αρθρώματα

Για τη φόρτωση κώδικα βιβλιοθήκης, χρησιμοποιήστε `require` ή `use` αντί για do EXPR. Η `require` αποθηκεύει στην προσωρινή μνήμη μέσω του %INC (ώστε ένα αρχείο να φορτώνεται το πολύ μία φορά), εγείρει εξαίρεση σε αποτυχία αντί να επιστρέφει `undef`, και ολοκληρώνεται με τους γάντζους του @INC. Η `use` προσθέτει επιπλέον φόρτωση κατά τη μεταγλώττιση και χειρισμό εισαγωγής. Το do EXPR υπάρχει για αρχεία ρυθμίσεων και ad-hoc ενσωμάτωση σεναρίων - όχι για βιβλιοθήκες.

Παραδείγματα

Ομαδοποίηση δηλώσεων σε έκφραση:

```
my $config = do {
    open my $fh, "<", $path or die $!;
    local $/;
    <$fh>;
};                              # $config = whole file as one string
```

Βρόχος μετα-ελέγχου με do … while:

```
my $n;
do {
    print "enter a positive number: ";
    chomp($n = <STDIN>);
} until defined $n && $n =~ /\A\d+\z/ && $n > 0;
```

Ανάγνωση αρχείου ρυθμίσεων σε σύνταξη Perl, με πλήρη τριμερή χειρισμό σφαλμάτων:

```
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;
```

Υπό συνθήκη επιλογή τιμής όπου κάθε διακλάδωση αποτελείται από αρκετές δηλώσεις:

```
my $greeting = do {
    if ($user) {
        my $name = $user->display_name;
        "hello, $name";
    }
    else {
        "hello, stranger";
    }
};
```

Προσωρινή τοπικοποίηση που πρέπει να καλύπτει αρκετές δηλώσεις και να αποδίδει και τιμή:

```
my $dump = do {
    local $Data::Dumper::Sortkeys = 1;
    local $Data::Dumper::Indent   = 1;
    Data::Dumper::Dumper(\%state);
};
```

Καθολική κατάσταση που επηρεάζει

- `@INC` - αναζητείται όταν η do EXPR καλείται με σχετική διαδρομή διαφορετική από `./...` ή `../...`.
- `%INC` - ενημερώνεται με τη διαδρομή που επιλύθηκε όταν η do EXPR πετυχαίνει σε αναζήτηση στο `@INC`.
- `$@` - τίθεται από την do EXPR σε αποτυχία μεταγλώττισης· καθαρίζεται σε επιτυχία.
- `$!` - τίθεται από την do EXPR σε αποτυχία I/O (το αρχείο δεν μπόρεσε να διαβαστεί).
- `$0`, `__FILE__`, `__LINE__` - το τρέχον υπό μεταγλώττιση όνομα αρχείου τίθεται στο αρχείο που φορτώνεται όσο η do EXPR το αναλύει, ώστε οι προειδοποιήσεις και τα μηνύματα `die` να δείχνουν στο σωστό αρχείο.

Το do BLOCK δεν επηρεάζει καθολικές μεταβλητές· είναι καθαρά συντακτική κατασκευή.

Οριακές περιπτώσεις

- **Το do BLOCK δεν είναι βρόχος.** Οι `next`, `last` και `redo` μέσα σε do ... while δεν αφορούν το μπλοκ. Χρησιμοποιήστε γυμνό μπλοκ ή πραγματικό βρόχο `while`.
- **Η `return` μέσα σε do BLOCK επιστρέφει από την περικλείουσα υπορουτίνα.** Όχι από το μπλοκ. Δεν υπάρχει τρόπος να επιστραφεί τιμή από do BLOCK πέρα από το να γίνει η τελευταία αποτιμούμενη έκφραση.
- **Οι τροποποιητές δήλωσης ελέγχουν εκ των προτέρων παντού αλλού.** Το `STATEMENT while COND` ελέγχει πρώτα· το `do BLOCK while COND` ελέγχει μετά την πρώτη εκτέλεση. Αυτή είναι σκόπιμη εξαίρεση της γλώσσας.
- **Σχετικές διαδρομές χωρίς `./` αναζητούν στο `@INC`.** Από την 5.26, το `.` δεν είναι στο `@INC`, οπότε `do 'stat.pl'` δεν σημαίνει πια «αρχείο τρέχοντος καταλόγου»· διαβιβάστε `./stat.pl` αν αυτή ήταν η πρόθεση.
- **Το do EXPR δεν μπορεί να δει περικλείουσες λεκτικές μεταβλητές.** Το αρχείο αναλύεται σαν να βρίσκεται στο ανώτατο επίπεδο. Αν χρειάζεστε πρόσβαση σε λεκτικές μεταβλητές, χρησιμοποιήστε `eval STRING` - με τις συνήθεις επιφυλάξεις περί taint και injection.
- **Το αρχείο επαναναλύεται σε κάθε κλήση.** Η do EXPR δεν αποθηκεύεται προσωρινά μέσω του `%INC` με τον τρόπο που είναι η `require`. Κάλεσμα μέσα σε βρόχο επαναδιαβάζει και επαναμεταγλωττίζει το αρχείο σε κάθε επανάληψη.
- **Ελέγχετε την `$@` πριν την `$!`.** Μια αποτυχία μεταγλώττισης μπορεί επίσης να θέσει την `$!` ως παρενέργεια· το σφάλμα μεταγλώττισης στην `$@` είναι αυτό που πραγματικά θέλετε να αναφέρετε.
- **Ένα αρχείο που μεταγλωττίζεται αλλά επιστρέφει ψευδή τιμή είναι αμφίσημο.** Το `unless ($ret = do $file) { ... }` πυροδοτείται τόσο για «το αρχείο

απέτυχε» όσο και για «το αρχείο εκτελέστηκε και επέστρεψε `0`». Αν επιτρέπεται το αρχείο ρυθμίσεών σας να καταλήγει σε ψευδή τιμή, χρησιμοποιήστε ρητά την `defined` και την `$@` αντί να ελέγχετε την αλήθεια της επιστροφής.

- **do χωρίς όρισμα είναι συντακτικό σφάλμα.** Και οι δύο μορφές απαιτούν έναν όρο: ένα μπλοκ ή μια έκφραση.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `eval` - η `eval BLOCK` παγιδεύει εξαιρέσεις χωρίς να διαβάζει αρχείο· η `eval STRING` αναλύει αυθαίρετο κώδικα και **μπορεί** να δει περικλείουσες λεκτικές μεταβλητές, σε αντίθεση με την `do EXPR`
- `require` - φορτώνει αρχείο βιβλιοθήκης με προσωρινή αποθήκευση μέσω `%INC` και αυτόματη εξαίρεση σε αποτυχία· το σωστό εργαλείο για αρθρώματα
- `return` - σημειώστε ότι η `return` μέσα σε `do BLOCK` επιστρέφει από την περικλείουσα υπορουτίνα, όχι από το μπλοκ
- `@INC` - διαδρομή αναζήτησης αρθρωμάτων που συμβουλεύεται η `do EXPR` σε σχετικά ονόματα αρχείων
- `%INC` - καταγράφει τη διαδρομή που επιλύθηκε μετά από μια επιτυχημένη `do EXPR` με αναζήτηση στο `@INC`
- `$@` - σφάλμα κατά τη μεταγλώττιση από `do EXPR`
- `$!` - σφάλμα I/O από `do EXPR` όταν το αρχείο δεν μπορεί να διαβαστεί

Ροή ελέγχου

dump

Προκαλεί στο εκτελούμενο πρόγραμμα να παράγει αμέσως core dump.

Η `dump` είναι προϊστορικός τελεστής της Perl του οποίου ο αρχικός σκοπός ήταν να παγώσει έναν εκτελούμενο διερμηνέα σε αρχείο core ώστε το (παρεχόμενο ξεχωριστά) εργαλείο `undump` να μπορέσει να μετατρέψει αυτό το αρχείο core πίσω σε εκτελέσιμο. Το προκύπτον δυαδικό, όταν εκτελούνταν, θα επανεκκινούσε το πρόγραμμα κάνοντας άλμα στο `LABEL` μέσω `goto`, παρακάμπτοντας το κόστος εκκίνησης της επανανάγνωσης και επαναμεταγλώττισης του πηγαίου κώδικα. Αυτή η ροή εργασίας είναι νεκρή για δεκαετίες: το `undump` δεν διανεμήθηκε ποτέ μαζί με την Perl, τα σύγχρονα λειτουργικά συστήματα καθιστούν τη μετατροπή core-σε-εκτελέσιμο ουσιαστικά αδύνατη, και ολόκληρος ο τελεστής είναι πλέον μια ιστορική περιέργεια. Από την Perl 5.30 και μετά πρέπει να γράφεται `CORE::dump()` ώστε να αποθαρρύνεται η κατά λάθος χρήση.

Δεν υπάρχει σχεδόν καμία κατάσταση στη σύγχρονη Perl όπου η `dump` είναι η σωστή απάντηση. Αν την επιλέξατε επειδή θέλετε ίχνος στοίβας, χρησιμοποιήστε `Carp::confess`. Αν θέλετε αφύσικη έξοδο, χρησιμοποιήστε `die` ή `exit`. Αν θέλετε στιγμιότυπο της κατάστασης του προγράμματος, χρησιμοποιήστε εργαλείο checkpointing σε επίπεδο λειτουργικού.

Σύνοψη

```
CORE::dump();
CORE::dump LABEL;
CORE::dump EXPR;
```

Τι επιστρέφεται

Τίποτα. Η dump δεν επιστρέφει: ο έλεγχος εγκαταλείπει τον διερμηνέα της Perl με τον τρόπο που η `abort(3)` εγκαταλείπει ένα πρόγραμμα C. Η διεργασία τερματίζεται από τον χειρισμό core dump του πυρήνα (τύπου SIGABRT), με τις συνήθεις παρενέργειες: μια κατάσταση εξόδου που αντικατοπτρίζει το σήμα, ένα πιθανό αρχείο `core` στον τρέχοντα κατάλογο αν το `RLIMIT_CORE` του χρήστη και το `core_pattern` του πυρήνα το επιτρέπουν, και κανένα ομαλό τερματισμό του runtime της Perl. Τα μπλοκ END **δεν** εκτελούνται. Οι μέθοδοι DESTROY **δεν** εκτελούνται. Η ενταμιευμένη έξοδος σε οποιοδήποτε filehandle χάνεται.

Καθολική κατάσταση που επηρεάζει

Καμία άμεσα. Η dump παρακάμπτει εντελώς τον κανονικό μηχανισμό τέλους προγράμματος - δεν διαβάζει ούτε γράφει ειδικές μεταβλητές, δεν εκκενώνει ενταμιευτές, και δεν εκτελεί καταστροφείς. Όποια κι αν είναι η κατάσταση του διερμηνέα τη στιγμή της κλήσης, αυτή προσγειώνεται στο αρχείο core.

Παραδείγματα

Πυροδότηση core dump από εκτελούμενο πρόγραμμα (απαιτεί `CORE::` από την 5.30 και μετά):

```
CORE::dump();
```

Με ετικέτα - αρχικά, το δυαδικό που παρήγαγε το undump θα συνέχιζε την εκτέλεση εδώ. Με το undump εξαφανισμένο αυτό είναι καθαρά διακοσμητικό· η διεργασία απλώς τερματίζεται όπως πριν:

```
RESTART:
    # ... initialisation ...
CORE::dump RESTART;
```

Όνομα ετικέτας υπολογιζόμενο κατά τον χρόνο εκτέλεσης (η μορφή dump EXPR, διαθέσιμη από την Perl 5.18):

```
my $label = pick_entry_point();
CORE::dump $label;
```

Αυτό που σχεδόν σίγουρα θέλετε αντί αυτής. Για μεταθανάτια ανάλυση με ίχνος στοίβας:

```
use Carp;
Carp::confess("unreachable state: $state");
```

Για καθαρή αφύσικη έξοδο με κωδικό κατάστασης:

```
die "fatal: $reason\n";      # exit via $@, runs END / DESTROY
exit 1;                      # plain exit, runs END / DESTROY
```

Οριακές περιπτώσεις

- **Σύνταξη από την 5.30**: σκέτο dump ή dump LABEL σε επίπεδο δήλωσης είναι σφάλμα μεταγλώττισης εκτός αν γραφεί ως CORE::dump ή CORE::dump(...). Αυτό είναι σκόπιμο - η ομάδα του core έκανε τον τελεστή δύσκολο να προσεγγιστεί κατά λάθος.
- **Το LABEL δεν επιλύεται κατά την κλήση**: ιστορικά η ετικέτα αναζητούνταν μόνο αφού το undump μετενσάρκωνε το πρόγραμμα, οπότε ένα τυπογραφικό λάθος δεν θα εντοπιζόταν κατά τη στιγμή της dump. Σήμερα το θέμα είναι ασήμαντο - το πρόγραμμα δεν συνεχίζει ποτέ, οπότε η ετικέτα είναι καθαρή τεκμηρίωση.
- **Η dump EXPR**: η έκφραση αποτιμάται σε βαθμωτό περιβάλλον και πρέπει να αποδίδει όνομα ετικέτας. Δεν έχει επίδραση στο αποτέλεσμα, για τον ίδιο λόγο.
- **Ιδιορρυθμίες ανάλυσης**: η dump έχει την ίδια προτεραιότητα με την ανάθεση και εξαιρείται από τον κανόνα «μοιάζει-με-συνάρτηση». Η dump ("foo") . "bar" αναλύεται με το "bar" ως μέρος του ορίσματος, όχι ως μεταγενέστερη συνένωση. Στην πράξη αυτό έχει σημασία μόνο αν γράφετε dump καθόλου, κάτι που δεν θα έπρεπε.
- **Τα ανοιχτά αρχεία χάνονται**: κάθε filehandle που είναι ανοιχτό τη στιγμή της dump εξαφανίζεται όταν (στην ιστορική ροή εργασίας) ξεκινά το μετενσαρκωμένο πρόγραμμα. Η προειδοποίηση στο POD για «πιθανή σύγχυση που προκαλείται στην Perl» χρονολογείται από την εποχή που κάποιος μπορεί όντως να το δοκίμαζε· σήμερα η προειδοποίηση είναι ακαδημαϊκή επειδή τίποτα δεν συνεχίζει.
- **Τα αρχεία core μπορεί να μην παραχθούν**: το αν ένα αρχείο core προσγειώνεται όντως στον δίσκο εξαρτάται από το ulimit -c, το kernel.core_pattern του πυρήνα, τα δικαιώματα του συστήματος αρχείων, και το αν η διεργασία τρέχει υπό supervisor που συλλαμβάνει ή απορρίπτει αρχεία core. Το να επιτύχει η dump στον τερματισμό της διεργασίας δεν είναι το ίδιο με το να είναι διαθέσιμο μετά κάποιο αρχείο core.
- **Φορητότητα**: σε συστήματα χωρίς ουσιαστική υποστήριξη core dump ο τελεστής συρρικνώνεται σε «τερμάτισε τη διεργασία»· δείτε το perlport για ιστορικές σημειώσεις.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- exit - τερματίζει κανονικά το πρόγραμμα, εκτελώντας τα μπλοκ END και τους καταστροφείς· αυτό είναι που σχεδόν σίγουρα θέλετε αντί της dump
- die - εγείρει εξαίρεση· χωρίς χειριστή τερματίζει το πρόγραμμα αλλά εξακολουθεί να εκτελεί τα μπλοκ END και να εκκενώνει την έξοδο

- `goto` - ο τελεστής μεταφοράς ελέγχου του οποίου η μορφή `LABEL` η `dump` ιστορικά υποτίθεται ότι συνεργαζόταν μαζί της μετά τη μετενσάρκωση
- `Carp::confess` - ίχνος στοίβας κατά την έξοδο, η σύγχρονη απάντηση στο «θέλω να δω την κατάσταση του προγράμματος στο σημείο της αποτυχίας»
- `warn` - εκπέμπει διαγνωστικό χωρίς τερματισμό· συνδυάστε με άρθρωμα ίχνους στοίβας όταν θέλετε πληροφορία χωρίς να τερματίσετε
- `perlrun` - τεκμηριώνει τον ιστορικό διακόπτη γραμμής εντολών `-u` που εκτελεί `dump` αμέσως μετά τη μεταγλώττιση

Πίνακες · Hashes

each

Διάσχιση ενός πίνακα κατακερματισμού ή πίνακα μία εγγραφή τη φορά.

Η `each` προωθεί έναν επαναλήπτη ανά container και επιστρέφει την επόμενη εγγραφή. Για έναν πίνακα κατακερματισμού αυτή η εγγραφή είναι ένα ζεύγος `(key, value)`· για έναν πίνακα είναι ένα ζεύγος `(index, value)`. Όταν ο επαναλήπτης έχει διασχίσει κάθε εγγραφή, η επόμενη κλήση επιστρέφει την κενή λίστα (ή `undef` σε βαθμωτό περιβάλλον) και έπειτα επανέρχεται στην αρχή, ώστε ένας νέος βρόχος `while (each ...)` να ξεκινήσει από την αρχή. Η σειρά του πίνακα κατακερματισμού είναι τυχαία ανά διεργασία και ανά πίνακα κατακερματισμού· η σειρά του πίνακα είναι αυστηρά από τον χαμηλότερο δείκτη προς τα πάνω.

Σύνοψη

```
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
```

Τι επιστρέφεται

- **Περιβάλλον λίστας**: μια λίστα δύο στοιχείων `($key, $value)` για έναν πίνακα κατακερματισμού, `($index, $value)` για έναν πίνακα. Σε εξάντληση, η κενή λίστα.
- **Βαθμωτό περιβάλλον**: το κλειδί (για πίνακα κατακερματισμού) ή ο δείκτης (για πίνακα). Σε εξάντληση, `undef`.
- **Σκέτη `each %h` σε συνθήκη `while`/`for`**: ελέγχει την *οριστικότητα* του επιστρεφόμενου κλειδιού/δείκτη, όχι την αληθοτιμή, και εκχωρεί το κλειδί στην `$_`. Αυτό κάνει το `while (each %h) { print "$_\n" }` να λειτουργεί ακόμη και όταν ένα κλειδί είναι η κενή συμβολοσειρά ή `0`.

Μετά την εξάντληση ο επαναλήπτης επαναφέρεται σιωπηρά, οπότε η κλήση *μετά* από εκείνη που επέστρεψε κενή ξαναξεκινά την επανάληψη από την αρχή.

Καθολική κατάσταση που επηρεάζει

Κάθε πίνακας κατακερματισμού και κάθε πίνακας φέρει **τον δικό του εσωτερικό επαναλήπτη**. Οι each, keys και values προωθούν και μοιράζονται όλες αυτόν τον έναν επαναλήπτη - είναι τρεις όψεις του ίδιου δρομέα, όχι τρεις ανεξάρτητοι βρόχοι. Αυτό είναι το πιο σημαντικό μεμονωμένο γεγονός σχετικά με την each:

- Η κλήση `keys %h` ή `values %h` (σε οποιοδήποτε περιβάλλον, ακόμη και ως εφάπαξ μέτρηση όπως `scalar keys %h`) **επαναφέρει** τον επαναλήπτη του πίνακα κατακερματισμού. Ένας βρόχος `while (each %h)` που διακόπτεται από `scalar keys %h` στο σώμα του θα ξαναξεκινήσει από την αρχή στην επόμενη επανάληψη - ή χειρότερα, θα εκτελείται για πάντα.
- **Η αναφορά ενός πίνακα κατακερματισμού σε περιβάλλον λίστας επαναφέρει επίσης τον επαναλήπτη του** (π.χ. `my @copy = %h`, ή πέρασμα της %h σε υπορουτίνα). Οι επαναλήπτες πινάκων *δεν* επαναφέρονται από αναφορά σε περιβάλλον λίστας.
- Δύο βρόχοι που επαναλαμβάνουν το ίδιο %h μοιράζονται έναν δρομέα. Η εμφώλευση `while (each %h) { ... while (each %h) { ... } ... }` επισκέπτεται κάθε εγγραφή το πολύ μία φορά συνολικά, όχι `n * m` φορές.
- Ένας ανώνυμος πίνακας κατακερματισμού/πίνακας στη συνθήκη `(each %{ +{ a => 1 } })` κατασκευάζει έναν *νέο* container σε κάθε επανάληψη, οπότε ο επαναλήπτης είναι πάντα φρέσκος - ο βρόχος δεν τερματίζει ποτέ.
- Η σκέτη μορφή της each εκχωρεί στην $_· αν βασίζεστε στην $_ αλλού στο σώμα του βρόχου, αποθηκεύστε την με local.

Κάθε είσοδος σε ένα βρόχο `while (each ...)` λαμβάνει όποια θέση επαναλήπτη βρίσκεται ο container εκείνη τη στιγμή. Δεν υπάρχει τρόπος να απομονώσετε τον επαναλήπτη ενός βρόχου από άλλο κώδικα που αγγίζει τον ίδιο container. Αν το σώμα του βρόχου καλεί συνάρτηση που μπορεί να διαβάσει τον ίδιο πίνακα κατακερματισμού, το ασφαλές μοτίβο είναι αντί αυτού ένας βρόχος foreach πάνω σε keys.

Παραδείγματα

Βασική επανάληψη πίνακα κατακερματισμού:

```
my %colour = (red => "#f00", green => "#0f0", blue => "#00f");
while (my ($name, $hex) = each %colour) {
    print "$name = $hex\n";
}
# order is unspecified
```

Επανάληψη πίνακα (Perl 5.12+):

```
my @letters = ('a' .. 'e');
while (my ($i, $c) = each @letters) {
    print "[$i] $c\n";
}
# [0] a
# [1] b
# [2] c
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
# [3] d
# [4] e
```

Εκτύπωση πίνακα κατακερματισμού σε ταξινομημένη σειρά - η each **δεν** ταξινομεί, οπότε καταφύγετε αντί αυτού σε keys συν sort:

```
for my $k (sort keys %colour) {
    print "$k = $colour{$k}\n";
}
```

Σκέτη each σε συνθήκη while, που θέτει την $_ στο κλειδί σε κάθε επανάληψη (Perl 5.18+):

```
while (each %ENV) {
    print "$_=$ENV{$_}\n";
}
```

Ασφαλής διαγραφή κατά την επανάληψη - μπορείτε να διαγράψετε το κλειδί που *μόλις επιστράφηκε* από την each χωρίς να ενοχλήσετε τον δρομέα:

```
while (my ($key, $value) = each %hash) {
    delete $hash{$key} if $value < 0;   # safe
}
```

Επανάληψη πίνακα κατακερματισμού με βρόχο foreach όταν οτιδήποτε στο σώμα μπορεί να αγγίξει το %hash (το εύρωστο μοτίβο - λάβετε πρώτα στιγμιότυπο των κλειδιών):

```
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.
}
```

foreach πολλαπλών τιμών πάνω σε πίνακα κατακερματισμού (Perl 5.36+), που καθιστά την each περιττή για τους περισσότερους βρόχους και αποφεύγει την παγίδα του κοινόχρηστου επαναλήπτη:

```
foreach my ($key, $value) (%hash) {
    print "$key => $value\n";
}
```

Οριακές περιπτώσεις

- **Ο επαναλήπτης δεν επαναφέρεται μεταξύ επανεισόδων**: ένας βρόχος while (each %h) που εξέρχεται πρόωρα μέσω last, return ή die αφήνει τον επαναλήπτη στη μέση της διάσχισης. Ο επόμενος while (each %h) στον ίδιο πίνακα κατακερματισμού συνεχίζει από εκεί που σταμάτησε ο πρώτος, όχι από την αρχή. Καλέστε keys %h σε περιβάλλον κενού - keys %h; - για να εξαναγκάσετε επαναφορά πριν τον νέο βρόχο.

- **Οι keys/values επαναφέρουν τον επαναλήπτη**: οποιαδήποτε κλήση στις `keys` ή `values` στον ίδιο πίνακα κατακερματισμού - συμπεριλαμβανομένης της `scalar keys %h` μέσα στο σώμα του βρόχου - επαναφέρει τον δρομέα.
- **Η αναφορά σε περιβάλλον λίστας επαναφέρει τον επαναλήπτη πίνακα κατακερματισμού, όχι πίνακα**: το `my @flat = %h` επαναφέρει τον δρομέα του `%h`· το ισοδύναμο `my @copy = @a` δεν αγγίζει τον επαναλήπτη του `@a`.
- **Η διαγραφή κατά την επανάληψη δεν είναι καθορισμένη** εκτός από την εγγραφή που επιστράφηκε πιο πρόσφατα. Η διαγραφή οποιουδήποτε άλλου κλειδιού μπορεί να οδηγήσει σε εγγραφές που επισκέπτονται δύο φορές ή παραλείπονται εντελώς.
- **Η προσθήκη κατά την επανάληψη δεν είναι καθορισμένη** - η εισαγωγή νέου κλειδιού στη μέση του βρόχου μπορεί να επισκεφθεί ή να μην επισκεφθεί, και μπορεί να διαταράξει τη σειρά των εγγραφών που δεν έχουν ακόμη ιδωθεί.
- **Ανώνυμος container στη συνθήκη**: η `each %{ +{ a => 1 } }` ή η `each @{ [1,2,3] }` κατασκευάζει έναν φρέσκο container κάθε φορά που αποτιμάται η συνθήκη, οπότε ο επαναλήπτης είναι πάντα νέος και ο βρόχος δεν τερματίζει ποτέ.
- **Δεσμευμένοι πίνακες κατακερματισμού και πίνακες** εκχωρούν στις `FIRSTKEY`/`NEXTKEY` (πίνακες κατακερματισμού) ή `FETCHSIZE`/`FETCH` (πίνακες), οπότε η σειρά επανάληψης είναι ό,τι επιλέγει η δεσμευμένη κλάση και καμία από τις παραπάνω εγγυήσεις δεν ισχύει απαραίτητα.
- **Οριστικότητα έναντι αληθοτιμίας στη συνθήκη βρόχου**: τα `while (each %h)` και `while (my $k = each %h)` ελέγχουν αμφότερα την *οριστικότητα* της έκφρασης, οπότε ένα κλειδί `"0"` ή `""` δεν τερματίζει τον βρόχο. Αυτό αποτελεί ειδική περίπτωση συγκεκριμένα για την `each`.
- **Βαθμωτές εκφράσεις**: η `each $href` σε βαθμωτό που κρατά αναφορά ήταν πειραματικό χαρακτηριστικό στην 5.14 και **αφαιρέθηκε στην 5.24**. Αποαναφέρετε ρητά: `each %$href`, `each @$aref`.
- **Κοινόχρηστη κατάσταση μεταξύ υπορουτινών**: το πέρασμα του `%h` σε υπορουτίνα που καλεί `each %h`, `keys %h` ή `values %h` πάνω του διαταράσσει τον επαναλήπτη του καλούντος. Αν δημοσιεύετε έναν πίνακα κατακερματισμού, υποθέστε ότι οι καλούντες μπορεί να σπάσουν τον βρόχο σας· προτιμήστε `foreach (keys ...)` για ευρωστία.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `keys` - στιγμιότυπο όλων των κλειδιών (πίνακας κατακερματισμού) ή δεικτών (πίνακας)· **επαναφέρει τον επαναλήπτη** που χρησιμοποιεί η `each`
- `values` - στιγμιότυπο όλων των τιμών· **επαναφέρει τον επαναλήπτη** που χρησιμοποιεί η `each`
- `delete` - ασφαλής για την εγγραφή που μόλις επιστράφηκε από την `each`, μη ασφαλής για οποιαδήποτε άλλη κατά την επανάληψη
- `exists` - έλεγχος ύπαρξης κλειδιού/δείκτη χωρίς διατάραξη του επαναλήπτη

- `for` / `foreach` - ο πιο ασφαλής βρόχος όταν το σώμα μπορεί να αγγίξει τον container· συνδυάστε με `keys` για να λάβετε πρώτα στιγμιότυπο
- `sort` - συνδυάστε με `keys` όταν χρειάζεστε καθορισμένη σειρά επανάληψης· η `each` δεν ταξινομεί ποτέ

Πληροφορίες χρηστών και ομάδων

endgrent

Κλείνει τη βάση δεδομένων ομάδων μετά την επανάληψη πάνω της.

Η `endgrent` είναι το μισό του καθαρισμού της τριάδας επανάληψης βάσης ομάδων `setgrent` / `getgrent` / `endgrent`. Λέει στην C library ότι έχετε τελειώσει τη διάσχιση του `/etc/group` (ή της πηγής NSS πίσω από αυτό), απελευθερώνοντας όποιους περιγραφείς αρχείων ή ενταμιευμένη κατάσταση άνοιξε η βιβλιοθήκη για λογαριασμό σας. Μετά την `endgrent`, η επόμενη κλήση `getgrent` ξεκινά φρέσκια επανάληψη από την κορυφή.

Σύνοψη

```
endgrent;
endgrent();
```

Χωρίς ορίσματα, χωρίς ουσιαστική τιμή επιστροφής.

Τι επιστρέφεται

Η τιμή επιστροφής είναι απροσδιόριστη και δεν πρέπει να βασίζεστε σε αυτήν. Η `endgrent` καλείται για την παρενέργειά της - το κλείσιμο της βάσης ομάδων - όχι για αποτέλεσμα. Θεωρήστε την ως κλήση σε κενό περιβάλλον.

```
endgrent;                               # correct
my $ok = endgrent;                      # wrong: $ok is not a success␣
↪flag
```

Αν η πλατφόρμα σας δεν υλοποιεί την `endgrent`, η κλήση εγείρει croak με `The endgrent function is unimplemented` κατά τον χρόνο εκτέλεσης. Αυτός είναι ουσιαστικά έλεγχος που γίνεται μία φορά - αν η συνάρτηση δουλεύει στο σύστημά σας έστω και μία φορά, θα συνεχίσει να δουλεύει.

Καθολική κατάσταση που επηρεάζει

Η `endgrent` λειτουργεί αμιγώς στην εσωτερική κατάσταση επανάληψης ομάδων της libc που μοιράζονται οι `setgrent` και `getgrent`. Δεν διαβάζει ούτε γράφει καμία ειδική μεταβλητή της Perl. Δεν αλλάζει το `$!`: μια διαδρομή αποτυχίας που θέτει το `errno` στην υποκείμενη κλήση C δεν αναδύεται στην Perl.

Η κατάσταση επανάληψης είναι **ανά διεργασία**, όχι ανά thread. Σε πρόγραμμα με threads, η `endgrent` ενός thread κλείνει την επανάληψη για κάθε thread - δείτε *Οριακές περιπτώσεις*.

- Οριακές περιπτώσεις

Παραδείγματα

Το κανονικό μοτίβο επανάληψης-και-κλεισίματος. Πάντα ζευγαρώνετε έναν βρόχο `setgrent` / `getgrent` με `endgrent` ώστε προγράμματα που τρέχουν για πολύ ώρα να μη διαρρέουν τον περιγραφέα αρχείου που κρατά ανοιχτό η C library:

```
setgrent;
while (my @g = getgrent) {
    my ($name, $passwd, $gid, $members) = @g;
    print "$name ($gid): $members\n";
}
endgrent;
```

Το να μπείτε σε `endgrent` στη μέση της επανάληψης είναι νόμιμο - απλώς εγκαταλείπει όποια θέση δρομέα κρατούσε η βιβλιοθήκη:

```
setgrent;
while (my @g = getgrent) {
    last if $g[0] eq 'wheel';       # found what we wanted
}
endgrent;                           # release the handle anyway
```

Επανεκκίνηση της επανάληψης. Η `endgrent` ακολουθούμενη από `setgrent` είναι ο φορητός τρόπος για επαναφορά στην αρχή:

```
setgrent;
my @first_pass  = collect_groups();
endgrent;

setgrent;
my @second_pass = collect_groups();  # starts over from the top
endgrent;
```

Αμυντικός καθαρισμός σε μπλοκ `END` για προγράμματα που μπορεί να πεθάνουν στη μέση της διάσχισης:

```
setgrent;
END { endgrent }
while (my @g = getgrent) { ... }    # any die still triggers END
```

Οριακές περιπτώσεις

- **Η κλήση της `endgrent` χωρίς προηγούμενη `setgrent` ή `getgrent`** είναι αβλαβής. Η C library σιωπηρά δεν κάνει τίποτα αν δεν είναι σε εξέλιξη επανάληψη.
- **Η κλήση της δύο φορές διαδοχικά** είναι επίσης αβλαβής - η δεύτερη κλήση είναι no-op.
- **Αμφισημία του αναλυτή.** Η `endgrent` δεν δέχεται όρισμα και έχει την προτεραιότητα ονομασμένου μοναδιαίου τελεστή. Μια bareword έκφραση αμέσως στα δεξιά της θα αναλυθεί ως όρισμα και θα απορριφθεί:

```
endgrent or die;                    # WRONG: parses as␣
↪endgrent(or die)
endgrent() or die;                  # right: empty parens␣
↪disambiguate
```

Στην πράξη δεν υπάρχει τίποτα με το οποίο να ελέγξετε την τιμή επιστροφής, οπότε αυτή η παγίδα σπάνια δαγκώνει - αλλά ο ίδιος κανόνας ισχύει για κάθε αλυσιδωτή έκφραση.

- **Τα threads μοιράζονται την επανάληψη.** Η `endgrent` από ένα thread τερματίζει την επανάληψη για όλα τα threads της ίδιας διεργασίας, επειδή η υποκείμενη κατάσταση είναι καθολική σε επίπεδο διεργασίας. Μην διαπλέκετε κλήσεις `getgrent` μεταξύ threads περιμένοντας ότι κάθε ένα θα δει κάθε εγγραφή.
- **NSS και caching.** Σε συστήματα όπου η βάση δεδομένων ομάδων εξυπηρετείται μέσω NSS (nss_ldap, nss_sss, κ.λπ.), η `endgrent` απελευθερώνει caches ανά module που μπορεί να κρατά το backend NSS. Μακρόβιοι δαίμονες που διασχίζουν περιοδικά τη βάση δεδομένων ομάδων θα πρέπει να καλούν `endgrent` μετά από κάθε διάσχιση ώστε να αποφεύγεται η συσσώρευση κατάστασης backend.
- **Δεν επηρεάζει τις κλήσεις αναζήτησης βάσει ονόματος.** Οι `getgrnam` και `getgrgid` είναι ανεξάρτητες του δρομέα επανάληψης. Η `endgrent` δεν ακυρώνει τα αποτελέσματά τους ούτε τα ενταμιευμένα δεδομένα τους, και δεν χρειάζεται να καλείτε `endgrent` πριν ή μετά από αυτές.
- **Μη υλοποιημένη σε ορισμένες πλατφόρμες.** Η Perl εγείρει croak `The endgrent function is unimplemented` αν η libc του υπολογιστή δεν έχει `endgrent(3)`. Όλα τα σύγχρονα συστήματα Linux, BSD και macOS την παρέχουν.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `setgrent` - ανοίγει ή επαναφέρει τη βάση δεδομένων ομάδων πριν τη διάσχισή της· πάντα ο πρόλογος επανάληψης που τερματίζει με `endgrent`
- `getgrent` - ανακτά την επόμενη εγγραφή από την επανάληψη που κλείνει η `endgrent`
- `getgrnam` - αναζήτηση μίας ομάδας με βάση το όνομα, ανεξάρτητη από τον δρομέα επανάληψης· δεν χρειάζεται `endgrent`
- `getgrgid` - ίδια με την `getgrnam` αλλά με κλειδί αριθμητικό gid
- `endpwent` - η παράλληλη κλήση καθαρισμού για τη βάση δεδομένων passwd· οι δύο επαναλήψεις είναι ανεξάρτητες και και οι δύο χρειάζονται κλείσιμο αν ανοίξατε και τις δύο

Πληροφορίες δικτύου

endhostent

Κλείνει τη βάση δεδομένων υπολογιστών μετά την επανάληψη.

Η `endhostent` απελευθερώνει οποιουσδήποτε πόρους κρατά η βάση δεδομένων υπολογιστών μετά από διάσχιση που οδηγείται από την `gethostent`. Ζευγαρώνει με την `sethostent`, η οποία ανοίγει ή επαναφέρει τη βάση δεδομένων, και είναι άμεσο περιτύλιγμα της ομώνυμης κλήσης της C library. Η κλήση της όταν δεν είναι σε εξέλιξη επανάληψη είναι αβλαβής.

Σύνοψη

```
endhostent;
endhostent();
```

Η `endhostent` δεν δέχεται ορίσματα. Οι παρενθέσεις είναι προαιρετικές.

Τι επιστρέφεται

Καμία χρήσιμη τιμή επιστροφής. Η κλήση γίνεται για την παρενέργειά της - την αποδόμηση όποιας κατάστασης ανά διεργασία διατηρεί η `gethostent(3)` της πλατφόρμας μεταξύ επαναλήψεων (περιγραφείς αρχείων στην παραδοσιακή επανάληψη `/etc/hosts`, υποδοχές resolver σε συστήματα με NSS).

Καθολική κατάσταση που επηρεάζει

- Τον δρομέα επανάληψης της βάσης δεδομένων υπολογιστών που μοιράζονται οι `gethostent`, `sethostent`, `gethostbyname`, και `gethostbyaddr`. Μετά την `endhostent`, η επόμενη κλήση `gethostent` ξεκινά εκ νέου από την αρχή (συνήθως μετά από σιωπηρή `sethostent`).
- Το `$?` μεταφέρει την τιμή `h_errno` της C library σε συστήματα όπου οι αναζητήσεις υπολογιστών την αναδύουν. Οι περισσότερες υλοποιήσεις της `endhostent` η ίδια δεν αγγίζουν το `h_errno`, αλλά μια επόμενη `gethostent` μπορεί να το κάνει.

Αυτή η κατάσταση είναι ανά διεργασία, όχι ανά thread. Δύο threads που επαναλαμβάνουν υπολογιστές ταυτόχρονα θα αλληλομπλέξουν εγγραφές και κανένα δεν θα δει την πλήρη λίστα

- pair `sethostent` / `gethostent` / `endhostent` within one thread.

Παραδείγματα

Κλείσιμο της βάσης δεδομένων μετά από πλήρη διάσχιση:

```
sethostent(1);                          # 1 = keep connection open
while (my @h = gethostent()) {
    print "$h[0]\n";                    # canonical hostname
}
endhostent();
```

Πάντα ζευγαρώνετε `endhostent` με `sethostent` μέσα σε υπορουτίνα ώστε η κατάσταση επανάληψης να μη διαρρέει στον καλούντα:

```
sub list_hosts {
    sethostent(0);
    my @names;
    while (my ($name) = gethostent()) {
        push @names, $name;
    }
    endhostent();
    return @names;
}
```

Αμυντικός καθαρισμός - η endhostent είναι ασφαλής ακόμα κι όταν δεν είναι σε εξέλιξη επανάληψη, οπότε είναι εντάξει να την καλείτε χωρίς όρους σε διαδρομή σφάλματος:

```
eval {
    sethostent(1);
    process_hosts();
    1;
} or do {
    my $err = $@;
    endhostent();
    die $err;
};
endhostent();
```

Η χρήση της αντικειμενοστραφούς διεπαφής accessor από το Net::hostent δεν αλλάζει τον κανόνα - ο υποκείμενος επαναλήπτης είναι ο ίδιος που κλείνει η endhostent:

```
use Net::hostent;
sethostent(1);
while (my $h = gethostent()) {
    print $h->name, "\n";
}
endhostent();
```

Οριακές περιπτώσεις

- **No-op όταν δεν είναι τίποτα ανοιχτό**: η κλήση της endhostent χωρίς προηγούμενη sethostent ή gethostent είναι ορισμένη και δεν κάνει τίποτα παρατηρήσιμο.
- **Διαπλοκή με αναζητήσεις βάσει ονόματος / διεύθυνσης**: μια κλήση gethostbyname ή gethostbyaddr στη μέση επανάληψης μπορεί να επαναφέρει τον δρομέα σε ορισμένες πλατφόρμες (κάθε υλοποίηση που εσωτερικά καλεί sethostent(0) για αναζητήσεις μίας εκτέλεσης). Μην αναμιγνύετε τα δύο στυλ έναντι της ίδιας βάσης δεδομένων χωρίς εκ νέου κλήση της sethostent μετά.
- **Πλατφόρμες χωρίς επαναλήπτη υπολογιστών**: σε συστήματα των οποίων ο resolver δεν υποστηρίζει αρίθμηση (οι περισσότερες σύγχρονες διαμορφώσεις NSS που ερωτούν μόνο DNS), η gethostent επιστρέφει αμέσως κενή λίστα. Η endhostent εξακολουθεί να είναι καλέσιμη και εξακολουθεί να είναι no-op.
- **Ασφάλεια threads**: ο επαναλήπτης υπολογιστών είναι καθολικός σε επίπεδο

διεργασίας. Σε πρόγραμμα πολλών threads, προστατέψτε την τριάδα `sethostent` / `gethostent` / `endhostent` με δικό σας mutex.

- **Πρωτότυπο** / **ανάλυση**: η `endhostent` δεν έχει ορίσματα, οπότε δεν υπάρχει παγίδα αμφισημίας του αναλυτή. Η γραφή `endhostent LIST` είναι συντακτικό σφάλμα αντί να ερμηνεύεται σιωπηλά ως κλήση με ορίσματα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `sethostent` - ανοίγει ή επαναφέρει τη βάση δεδομένων υπολογιστών· το ανοιγόμενο μισό του ζεύγους επανάληψης
- `gethostent` - διαβάζει μία εγγραφή από την ανοιχτή βάση δεδομένων· επιστρέφει την κενή λίστα όταν εξαντληθεί η επανάληψη
- `gethostbyname` - αναζήτηση μίας εκτέλεσης βάσει ονόματος· δεν απαιτεί `sethostent` / `endhostent`
- `gethostbyaddr` - αναζήτηση μίας εκτέλεσης βάσει διεύθυνσης
- `endnetent` - ίδιο μοτίβο για τη βάση δεδομένων δικτύων
- `endservent` - ίδιο μοτίβο για τη βάση δεδομένων υπηρεσιών
- `$?` - λαμβάνει το `h_errno` από αποτυχημένες αναζητήσεις υπολογιστών σε πλατφόρμες όπου η C library το εκθέτει

Πληροφορίες δικτύου

endnetent

Κλείνει τη βάση δικτύων μετά την επανάληψη.

Η `endnetent` απελευθερώνει τους πόρους που κατέχει η βάση δικτύων - συνήθως το αρχείο `/etc/networks` ή το αντίστοιχο NSS-υποστηριζόμενο - που ανοίχτηκαν σιωπηρά από την πρώτη κλήση στις `getnetent`, `getnetbyname` ή `getnetbyaddr`, ή ρητά από την `setnetent`. Μετά την `endnetent` ο δρομέας επανάληψης χάνεται· μια επόμενη `getnetent` ξεκινά μια νέα σάρωση από την κορυφή της βάσης. Είναι ένα λεπτό περιτύλιγμα της `endnetent(3)` της βιβλιοθήκης C του συστήματος.

Σύνοψη

```
endnetent;
endnetent();
```

Τι επιστρέφεται

Μια αληθής τιμή ορισμένη από το σύστημα. Η συνάρτηση C επιστρέφει `void`· η Perl εκθέτει έναν εκπρόσωπο ώστε η κλήση να μπορεί να σταθεί ως εντολή. Μη συγκρίνετε την επιστρεφόμενη τιμή με τίποτα - δεν υπάρχει τίποτε χρήσιμο μέσα της.

Καθολική κατάσταση που αγγίζει

Η endnetent ενεργεί επί του δρομέα της βάσης δικτύων που είναι καθολικός σε επίπεδο διεργασίας και ανήκει στη libc. Ο δρομέας αυτός μοιράζεται με τις getnetent, getnetbyname, getnetbyaddr και setnetent· μια κλήση εδώ ακυρώνει την κατάσταση επανάληψης από την οποία εξαρτώνται οι συναρτήσεις αυτές. Καμία ειδική μεταβλητή της Perl δεν αγγίζεται.

Παραδείγματα

Σε συνδυασμό με τις setnetent και getnetent για μια πλήρη σάρωση της βάσης δικτύων:

```
setnetent(1);                           # stayopen = 1
while (my @net = getnetent()) {
    my ($name, $aliases, $addrtype, $net) = @net;
    print "$name\t$net\n";
}
endnetent();                            # release the cursor
```

Κλείστε μετά τις λίγες αναζητήσεις που πραγματικά χρειαστήκατε, ώστε το αρχείο της βάσης να μη μείνει ανοιχτό για το υπόλοιπο του προγράμματος:

```
setnetent(0);
my @loopback = getnetbyname("loopback");
my @link     = getnetbyname("link-local");
endnetent();
```

Ασφαλής κλήση όταν η βάση δεν ανοίχτηκε ποτέ - η libc τη μεταχειρίζεται ως no-op:

```
endnetent();                            # harmless on a fresh process
```

Οριακές περιπτώσεις

- **Χωρίς ορίσματα, προτεραιότητα τελεστή λίστας**. Η endnetent δεν παίρνει ορίσματα αλλά αναλύεται ως τελεστής λίστας, οπότε ένας αδέσποτος όρος μετά από αυτήν καταπίνεται: το endnetent $x είναι endnetent($x) και παράγει σφάλμα μεταγλώττισης «Too many arguments». Γράψτε endnetent; ή endnetent();.
- **Επανειλημμένες κλήσεις**. Η κλήση της endnetent δύο φορές στη σειρά είναι no-op στη δεύτερη κλήση σε κάθε libc που στοχεύουμε. Μη βασίζεστε στο ότι θα προκαλέσει σφάλμα.
- **Αλληλοδιαπλοκή με άλλους επαναλήπτες ΒΔ**. Ο δρομέας δικτύων είναι ανεξάρτητος από τους δρομείς που χρησιμοποιούν οι getpwent, getgrent, gethostent, getprotoent και getservent. Η endnetent δεν τους κλείνει - καλέστε την αντίστοιχη end*ent για κάθε επαναλήπτη που ανοίξατε.
- **Διχάλωση μεταξύ setnetent και endnetent**. Ο δρομέας ζει στην κατάσταση libc του γονέα· ένα παιδί τον κληρονομεί κατά τη στιγμή του fork αλλά περαιτέρω κλήσεις της getnetent σε οποιαδήποτε διεργασία θα αποσυγχρονιστούν. Κλείστε και ανοίξτε ξανά στο παιδί.

- **Υποστρώματα NSS**. Στη glibc η βάση δικτύων δρομολογείται μέσω NSS, οπότε η σημασία του «close» εξαρτάται από τη ρυθμισμένη υπηρεσία (`files`, `nis`, `ldap`, …). Η συμπεριφορά που βλέπει ο χρήστης - η επανάληψη ξεκινά εκ νέου στην επόμενη `getnetent` - είναι η ίδια· το κόστος της `endnetent` δεν είναι.
- **Ασφάλεια νημάτων**. Ο δρομέας δικτύων της libc είναι ένας μοναδικός καθολικός πόρος σε επίπεδο διεργασίας. Το pperl δεν σειριοποιεί την πρόσβαση σε αυτόν· ένα πρόγραμμα που επαναλαμβάνει τη βάση δικτύων από πολλά νήματα πρέπει να κάνει το δικό του κλείδωμα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `setnetent` - ανοίγει τη βάση δικτύων και επιλέγει εάν θα παραμένει ανοιχτή κατά τη διάρκεια μεμονωμένων αναζητήσεων
- `getnetent` - διαβάζει την επόμενη εγγραφή από τον δρομέα που κλείνει η `endnetent`
- `getnetbyname` - μεμονωμένη αναζήτηση με όνομα δικτύου· ανοίγει σιωπηρά τη βάση αλλά δεν την κλείνει
- `getnetbyaddr` - μεμονωμένη αναζήτηση με αριθμητική διεύθυνση, ο ίδιος κανόνας σιωπηρού ανοίγματος, χωρίς σιωπηρό κλείσιμο
- `endhostent` - το αντίστοιχο για τη βάση κεντρικών υπολογιστών· ίδιο μοτίβο, ανεξάρτητος δρομέας
- `endservent` - το αντίστοιχο για τη βάση υπηρεσιών, για πληρότητα της οικογένειας `end*ent`

Πληροφορίες δικτύου

endprotoent

Κλείνει τη βάση πρωτοκόλλων μετά την επανάληψη.

Η `endprotoent` είναι ο τερματιστής της τριάδας επανάληψης `getprotoent` / `setprotoent` / `endprotoent` που διασχίζει τη βάση πρωτοκόλλων του συστήματος (συνήθως `/etc/protocols`). Απελευθερώνει όποιο handle ή κατάσταση κρατούσε ανοιχτή η C library του συστήματος για την επανάληψη. Την καλείτε αφότου ολοκληρωθεί η ακολουθιακή διάσχιση· οι `getprotobyname` και `getprotobynumber` δεν τη χρειάζονται - ανοίγουν και κλείνουν τη βάση σε κάθε κλήση.

Δεν δέχεται ορίσματα, δεν επιστρέφει τίποτα χρήσιμο. Πρόκειται για έναν λεπτό περιτυλιγμό πάνω από την κλήση του συστήματος `endprotoent(3)`.

Σύνοψη

```
endprotoent
```

Τι επιστρέφεται

Καμία ουσιαστική τιμή επιστροφής. Η κλήση της σε περιβάλλον λίστας ή βαθμωτό περιβάλλον αποδίδει την κενή λίστα / αντίστοιχο του undef· τίποτα που να θέλατε να ελέγξετε. Η αξία της είναι η παρενέργεια του κλεισίματος της βάσης πρωτοκόλλων.

Παραδείγματα

Διασχίστε κάθε πρωτόκολλο, μετά κλείστε τη βάση:

```
setprotoent(1);                         # 1 = keep the file open
 ↪between calls
while (my @p = getprotoent()) {
    my ($name, $aliases, $proto) = @p;
    print "$proto\t$name\n";
}
endprotoent();
```

Βραχυκύκλωμα κατά τη μέση της διάσχισης - η `endprotoent` εξακολουθεί να κλείνει καθαρά:

```
setprotoent(0);
while (my @p = getprotoent()) {
    last if $p[0] eq 'tcp';
}
endprotoent();
```

Συνδυασμένη με μια απευθείας αναζήτηση, η `endprotoent` είναι περιττή - οι αναζητήσεις κατά όνομα και αριθμό διαχειρίζονται μόνες τους την κατάστασή τους:

```
my @tcp = getprotobyname('tcp');        # no endprotoent needed
```

Οριακές περιπτώσεις

- **Κλήση χωρίς προηγούμενο `setprotoent` / `getprotoent`**: ακίνδυνο. Η κλήση συστήματος είναι idempotent· το κλείσιμο μιας ήδη κλειστής βάσης είναι no-op σε κάθε υποστηριζόμενη πλατφόρμα.
- **Δεν δέχεται ορίσματα**: το `endprotoent $x` είναι συντακτικό σφάλμα - η ενσωματωμένη συνάρτηση έχει μηδενική πληθικότητα.
- **Μη επανεισόδιμη**: η βάση πρωτοκόλλων είναι ένας μοναδικός καθολικός πόρος μέσα στη C library. Δύο ταυτόχρονες επαναλήψεις στην ίδια διεργασία πατούν η μία την κατάσταση της άλλης. Χρησιμοποιήστε διασχίσεις ανά νήμα ή ανά υποδιεργασία αν χρειάζεστε παραλληλισμό.
- **Ιδιορρυθμίες πλατφόρμας**: η συμπεριφορά είναι ό,τι κάνει η `endprotoent(3)` του συστήματος που φιλοξενεί. Δείτε το `perlport` για τη λίστα πλατφορμών όπου

οι ρουτίνες της βάσης δικτύου αποκλίνουν ή απουσιάζουν εντελώς.

- **Στελεχωμένη με stub σε πλατφόρμες χωρίς `/etc/protocols`**: σε συστήματα όπου η C library δεν έχει βάση πρωτοκόλλων, η κλήση είναι no-op αντί για σφάλμα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `setprotoent` - ανοίγει τη βάση πρωτοκόλλων και προαιρετικά την κρατά ανοιχτή κατά τις κλήσεις `getprotoent`· συνδυάστε με την `endprotoent` για να οριοθετήσετε μια διάσχιση
- `getprotoent` - διαβάζει την επόμενη εγγραφή κατά την επανάληψη· οδηγεί τον βρόχο που τερματίζει η `endprotoent`
- `getprotobyname` - αναζήτηση μίας εκτέλεσης κατά όνομα πρωτοκόλλου· αυτοτελής, δεν χρειάζεται `endprotoent`
- `getprotobynumber` - αναζήτηση μίας εκτέλεσης κατά αριθμό πρωτοκόλλου· ίδια ανεξαρτησία από την τριάδα επανάληψης
- `endservent` - ο ανάλογος τερματιστής για τη βάση δεδομένων υπηρεσιών· ίδιο ιδίωμα, διαφορετικός πίνακας
- `endhostent` - ο ανάλογος τερματιστής για τη βάση δεδομένων υπολογιστών

Πληροφορίες χρηστών και ομάδων

endpwent

Κλείνει τον επαναλήπτη της βάσης passwd που άνοιξε από τις `getpwent` ή `setpwent`.

Η `endpwent` είναι η συντροφική κλήση κατάλυσης για ένα πέρασμα του αρχείου passwd. Μετά την επανάληψη της βάσης κωδικών με επανειλημμένες κλήσεις `getpwent`, η `endpwent` απελευθερώνει όποια κατάσταση ανά διεργασία είχε δεσμεύσει η βιβλιοθήκη C για να συγκρατήσει τον δρομέα επανάληψης (έναν ανοιχτό περιγραφέα αρχείου στο `/etc/passwd`, ένα ταμιευμένο buffer για NSS, μια σύνδεση σε `nscd` / `sssd`, μια συνεδρία LDAP, αναλόγως της ρύθμισης υπηρεσίας ονομάτων του υπολογιστή). Δεν παίρνει ορίσματα και περιτυλίγει απευθείας τη συστημική κλήση `endpwent(3)`.

Σύνοψη

```
endpwent
```

Τι επιστρέφεται

Μια μοναδική λογική τιμή. Η επιστροφή σπάνια ελέγχεται - σε κάθε πλατφόρμα που υποστηρίζει το PetaPerl, η `endpwent` δεν μπορεί να αποτύχει με ουσιαστικό τρόπο, και ο πραγματικός κώδικας τη μεταχειρίζεται ως κλήση void.

Καθολική κατάσταση που αγγίζει

Η endpwent δεν μεταλλάσσει καμία μεταβλητή ορατή στην Perl. Η κατάσταση που εκκαθαρίζει είναι εξ ολοκλήρου μέσα στη βιβλιοθήκη C: ο επαναλήπτης passwd ανά διεργασία, που μοιράζεται με τις getpwent και setpwent (και, σε ορισμένα συστήματα, με τις getpwnam / getpwuid όταν τυχαίνει να επαναχρησιμοποιούν το ίδιο υποκείμενο ρεύμα). Η κλήση της endpwent σε ένα νήμα επηρεάζει κάθε άλλο νήμα στην ίδια διεργασία - ο επαναλήπτης είναι ανά διεργασία, όχι ανά νήμα.

Παραδείγματα

Τυπικός ιδιωματισμός επανάληψης-και-κλεισίματος:

```
setpwent;
while (my @ent = getpwent) {
    my ($name, undef, $uid) = @ent;
    print "$uid\t$name\n";
}
endpwent;
```

Εξασφαλίστε ότι ο επαναλήπτης κλείνει ακόμη και αν ο βρόχος βγει νωρίς:

```
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
```

Συνδυάστε με μια φύλαξη local ώστε ο εμφωλευμένος κώδικας να μην μπορεί να αφήσει τον επαναλήπτη μισο-διανυσμένο:

```
sub each_user {
    my $cb = shift;
    setpwent;
    my $guard = Guard::guard { endpwent };   # or a manual eval-and-
↪rethrow
    while (my @ent = getpwent) { $cb->(@ent) }
}
```

Η endpwent χωρίς προηγούμενη επανάληψη είναι αβλαβής:

```
endpwent;                                    # no-op on a fresh␣
↪process
```

Οριακές περιπτώσεις

- **Χωρίς ορίσματα, χωρίς ανάγκη παρενθέσεων.** Η endpwent είναι επώνυμος μοναδιαίος τελεστής με μηδέν ορίσματα· γράψτε την σκέτη. Η endpwent() γίνεται δεκτή αλλά δεν προσθέτει τίποτα.
- **Ταυτοδύναμη.** Η κλήση της endpwent δύο φορές στη σειρά, ή η κλήση της πριν από οποιαδήποτε getpwent / setpwent, ορίζεται ως επιτυχημένο no-op σε κάθε υποστηριζόμενη πλατφόρμα.
- **Δεν ακυρώνει εγγραφές που έχετε ήδη ανακτήσει.** Πίνακες και πίνακες κατακερματισμού που επέστρεψαν προηγούμενες κλήσεις getpwent / getpwnam / getpwuid είναι σκέτα δεδομένα της Perl και παραμένουν έγκυρα μετά την endpwent - μόνο ο δρομέας επανάληψης κλείνει.
- **Ασφάλεια νημάτων.** Ο επαναλήπτης είναι καθολικός σε επίπεδο διεργασίας. Αν δύο νήματα διατρέχουν τη βάση passwd ταυτόχρονα, η endpwent κλείνει τον κοινό δρομέα και για τα δύο - κανένα νήμα δεν θα λάβει το πλήρες σύνολο εγγραφών. Χρησιμοποιήστε κλείδωμα γύρω από ολόκληρο το μπλοκ setpwent / βρόχος / endpwent όταν επαναλαμβάνετε από νήματα.
- **Υποστρώματα υπηρεσίας ονομάτων.** Σε υπολογιστές που χρησιμοποιούν LDAP, NIS, sssd ή οποιοδήποτε άλλο μη-flat-file υπόστρωμα, η endpwent μπορεί να απελευθερώσει έναν δικτυακό χειριστή. Ένας μακρόβιος δαίμονας που επαναλαμβάνει τη βάση passwd περιοδικά πρέπει να καλεί την endpwent στο τέλος κάθε περάσματος αντί να διαρρέει έναν χειριστή ανά πέρασμα.
- **Δεν είναι αναγκαστική εκκένωση κρυφής μνήμης.** Η endpwent κλείνει τον επαναλήπτη· **δεν** εκκενώνει τις κρυφές μνήμες getpwnam / getpwuid της βιβλιοθήκης C. Μια επόμενη getpwnam μπορεί να επιστρέψει ακόμη εγγραφή από την κρυφή μνήμη.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- setpwent - επανατυλίγει ή ανοίγει τον επαναλήπτη της βάσης passwd πριν από ένα πέρασμα· συνεργάζεται με την endpwent ως αγκύλη ανοίγματος/κλεισίματος
- getpwent - ανακτά την επόμενη εγγραφή από τον επαναλήπτη που κλείνει η endpwent
- getpwnam - απευθείας αναζήτηση με όνομα σύνδεσης, ανεξάρτητη από τον επαναλήπτη στις περισσότερες υλοποιήσεις
- getpwuid - απευθείας αναζήτηση με αριθμητικό UID, η ίδια σημείωση ανεξαρτησίας με την getpwnam
- endgrent - αναλογική κατάλυση για τον επαναλήπτη της βάσης ομάδων· οι δύο σχεδόν πάντα καλούνται μαζί

Πληροφορίες δικτύου

endservent

Κλείνει τη βάση δεδομένων υπηρεσιών μετά από διάσχιση με την `getservent`.

Η `endservent` είναι το ήμισυ αποδόμησης της τριάδας επανάληψης της βάσης δεδομένων υπηρεσιών. Αφού η `setservent` ανοίξει τη βάση (τυπικά `/etc/services` σε Linux) και μια ακολουθία κλήσεων `getservent` τη διασχίσει εγγραφή προς εγγραφή, η `endservent` απελευθερώνει το υποκείμενο handle και οποιαδήποτε ενταμιευμένη κατάσταση κρατάει η βιβλιοθήκη C. Τυλίγει την POSIX κλήση `endservent(3)` ένα-προς-ένα.

Σύνοψη

```
endservent;
endservent();
```

Τι επιστρέφεται

Τίποτα χρήσιμο. Η συνάρτηση καλείται για την παρενέργειά της· δεν επιστρέφει ουσιαστική τιμή. Μην ελέγχετε την τιμή επιστροφής.

Καθολική κατάσταση που επηρεάζει

- Τον επαναλήπτη της βάσης δεδομένων υπηρεσιών σε επίπεδο συστήματος που ανοίγει η `setservent` ή έμμεσα η πρώτη κλήση `getservent` / `getservbyname` / `getservbyport`. Η `endservent` κλείνει αυτόν τον επαναλήπτη.
- Η `$!` **δεν** τίθεται ουσιαστικά από αυτήν την κλήση· το συμβόλαιο POSIX για την `endservent(3)` δεν ορίζει επιστροφή σφάλματος.

Παραδείγματα

Σε συνδυασμό με τις `setservent` και `getservent` για απαρίθμηση κάθε εγγραφής στο `/etc/services`:

```
setservent 0;
while (my @e = getservent) {
    last unless @e;
    my ($name, $aliases, $port, $proto) = @e;
    print "$name/$proto = $port\n";
}
endservent;
```

Κλείστε μια διάσχιση πρόωρα. Η βάση δεδομένων άνοιξε έμμεσα από την πρώτη `getservent`· η `endservent` είναι παρ” όλα αυτά ο σωστός τρόπος να την απελευθερώσετε:

```
while (my @e = getservent) {
    last if $e[0] eq 'http';
}
endservent;
```

Αμυντικός καθαρισμός γύρω από ένα μπλοκ που μπορεί να αφήσει τον επαναλήπτη ανοιχτό. Ακίνδυνος όταν δεν βρίσκεται σε εξέλιξη καμία επανάληψη:

```
eval {
    setservent 1;
    scan_services();
    1;
} or do {
    endservent;          # always release the handle
    die $@;
};
endservent;
```

Οριακές περιπτώσεις

- **Η κλήση χωρίς προηγούμενη `setservent` ή `getservent`** είναι ακίνδυνη. Η υποκείμενη κλήση C ανέχεται ένα no-op κλείσιμο.
- **Διπλή κλήση** είναι ακίνδυνη για τον ίδιο λόγο. Επανειλημμένη `endservent` δεν αναφέρει σφάλμα.
- **Δεν επηρεάζει τις συναρτήσεις αναζήτησης ονόματος και θύρας** - οι `getservbyname` και `getservbyport` ανοίγουν και κλείνουν τα δικά τους βραχύβια handles και δεν επηρεάζονται από την κατάσταση επαναλήπτη που διαχειρίζεται η `endservent`.
- **Κανένα πρωτότυπο, κανένα όρισμα**. Το `endservent $x` είναι σφάλμα κατά τη μεταγλώττιση. Χρησιμοποιήστε `endservent;` ή `endservent();`.
- **Μη φορητό πέρα από Unix-like συστήματα**. Σε πλατφόρμες όπου η βιβλιοθήκη C δεν διαθέτει την `endservent(3)`, η κλήση εκπέμπει `endservent not implemented`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `setservent` - ανοίξτε τη βάση δεδομένων υπηρεσιών και, με μη μηδενικό `STAYOPEN`, κρατήστε το handle σε όλες τις επόμενες αναζητήσεις
- `getservent` - διαβάστε την επόμενη εγγραφή από την ανοιχτή βάση δεδομένων υπηρεσιών· συνδυάστε κάθε βρόχο επανάληψης με `endservent`
- `getservbyname` - αναζήτηση μίας εκτέλεσης με όνομα υπηρεσίας· δεν αλληλεπιδρά με τον επαναλήπτη που κλείνει η `endservent`
- `getservbyport` - αναζήτηση μίας εκτέλεσης με αριθμό θύρας· ίδια ανεξαρτησία από τον επαναλήπτη
- `endprotoent` - το ανάλογο για τη βάση δεδομένων πρωτοκόλλων, ίδια μορφή και ίδιοι κανόνες
- `endhostent` - το ανάλογο για τη βάση δεδομένων υπολογιστών

I/O

eof

Ελέγχει ένα filehandle για τέλος αρχείου.

Η `eof` αναφέρει αν η επόμενη ανάγνωση σε ένα filehandle θα επιστρέψει τέλος αρχείου. Οι τρεις μορφές κλήσης μοιάζουν, αλλά σημαίνουν διαφορετικά πράγματα: η `eof FILEHANDLE` ρωτά για ένα συγκεκριμένο handle, η σκέτη `eof` ρωτά για το handle από το οποίο έγινε η πιο πρόσφατη ανάγνωση, και η `eof()` με κενές παρενθέσεις ρωτά για την εικονική ροή ARGV που τροφοδοτείται από τον <>. Οι μορφές δεν είναι εναλλάξιμες - η επιλογή της λάθος είναι το κλασικό σφάλμα με αυτή την ενσωματωμένη συνάρτηση.

Σύνοψη

```
eof FILEHANDLE
eof()
eof
```

Τι επιστρέφεται

1 αν η επόμενη ανάγνωση στο handle θα επέστρεφε τέλος αρχείου, ή αν το handle δεν είναι ανοιχτό. `0` διαφορετικά. Η `eof` ποτέ δεν μπλοκάρει περιμένοντας περισσότερη είσοδο σε ένα κανονικό αρχείο, αλλά σε σωλήνα ή τερματικό ενδέχεται να χρειαστεί να διαβάσει και να επιστρέψει έναν χαρακτήρα για να αποφασίσει - δείτε *Οριακές περιπτώσεις*.

Καθολική κατάσταση που επηρεάζει

- ARGV - το μαγικό filehandle που υποστηρίζει τον τελεστή <>. Η `eof()` το εξετάζει και ενδέχεται να το προχωρήσει· η `eof` μέσα σε βρόχο `while (<>)` το ρωτά έμμεσα.
- `@ARGV` - η λίστα των ονομάτων αρχείων που διατρέχει ο <>. Η `eof()` που καλείται πριν χρησιμοποιηθεί ο <> πυροδοτεί επιθεώρηση του `@ARGV` και, αν είναι κενός, ρυθμίζει την ανάγνωση από το `STDIN`.
- `$.` - ο τρέχων αριθμός γραμμής εισόδου, συνδεδεμένος με το handle που ρωτά η `eof`. Δεν γράφεται από την ίδια την `eof`, αλλά είναι σχετικός στα ιδιώματα παρακάτω όπου η `close ARGV` τον επαναφέρει στα όρια αρχείων.

Οι τρεις μορφές, αναλυτικά

`eof FILEHANDLE` - ρωτά για αυτό το ένα handle. Το `FILEHANDLE` μπορεί να είναι bareword, βαθμωτό που κρατά handle, ή έκφραση που αποτιμάται σε handle. Επιστρέφει αληθές για ένα handle κλειστό ή που δεν άνοιξε ποτέ.

`eof` (χωρίς παρενθέσεις, χωρίς όρισμα) - ρωτά για το *τελευταίο αρχείο που διαβάστηκε*. Μέσα σε βρόχο `while (<>) { ... }` αυτό σημαίνει «το τρέχον αρχείο ARGV» - το φυσικό αρχείο που αυτή τη στιγμή ρέει, όχι ολόκληρη η ακολουθία του <>. Η `eof` γίνεται αληθής σε κάθε όριο αρχείου, μία φορά ανά αρχείο εισόδου.

`eof()` (κενές παρενθέσεις) - ρωτά για τη ροή του <> ως σύνολο. Γίνεται αληθής μόνο στο τέλος του *τελευταίου* αρχείου στο `@ARGV`. Αν ο <> δεν έχει χρησιμοποιηθεί ακόμη, η `eof()` θα εξετάσει το `@ARGV`, θα καταφύγει στο `STDIN` όταν το `@ARGV` είναι κενό, και ενδέχεται όντως να ανοίξει το πρώτο αρχείο για να απαντήσει στην ερώτηση.

Η σκέτη `eof` και η `eof()` φαίνονται σχεδόν πανομοιότυπες και κάνουν εντελώς διαφορετικά πράγματα. Οι παρενθέσεις έχουν σημασία.

Παραδείγματα

Έλεγχος ενός συγκεκριμένου handle πριν την επόμενη ανάγνωση:

```
open my $fh, "<", "data.txt" or die $!;
while (not eof $fh) {
    my $line = <$fh>;
    # process $line
}
close $fh;
```

Το κλασικό ιδίωμα `while (<>)`, με επαναφορά του `$.` σε κάθε όριο αρχείου. Η σκέτη `eof` (χωρίς παρενθέσεις) είναι αυτό που θέλετε εδώ - πυροδοτείται μία φορά ανά αρχείο εισόδου:

```
while (<>) {
    next if /^\s*#/;
    print "$.\t$_";
} continue {
    close ARGV if eof;        # not eof() - resets $. per file
}
```

Εισαγωγή ενός δείκτη πριν την τελευταία γραμμή του τελευταίου αρχείου. Η `eof()` με παρενθέσεις πιάνει το τέλος *ολόκληρης* της ροής <>, όχι κάθε αρχείου:

```
while (<>) {
    if (eof()) {
        print "--------------\n";
    }
    print;
    last if eof();            # needed when reading from a terminal
}
```

Προστασία έναντι κλειστού handle. Η `eof` σε κλειστό ή ποτέ-ανοιχτό handle επιστρέφει αληθές χωρίς προειδοποίηση:

```
my $fh;                       # never opened
if (eof $fh) {
    warn "handle not ready";
}
```

Οριακές περιπτώσεις

- **Η `eof` πριν από οποιαδήποτε ανάγνωση επιστρέφει ψευδές, όχι αληθές.** Η σκέτη `eof` χωρίς προηγούμενη ανάγνωση δεν έχει «τελευταίο αρχείο που διαβάστηκε» στο οποίο να αναφερθεί και η Perl την αντιμετωπίζει ως ψευδή. Το `if (eof) { ... }` στην αρχή ενός προγράμματος δεν σημαίνει «αν δεν υπάρχει είσοδος» - χρησιμοποιήστε `eof()` για αυτό.
- **Η `eof()` πριν χρησιμοποιηθεί ο `<>` έχει παρενέργειες.** Εξετάζει το `@ARGV`, και αν το `@ARGV` είναι κενό ρυθμίζει τον `<>` να διαβάζει από το `STDIN`. Μια απρόσεκτη `eof()` στην αρχή του προγράμματος μπορεί επομένως να συνδέσει σιωπηλά το πρόγραμμα στο `STDIN`.
- **Η `eof()` μετά την εξάντληση της εισόδου του `<>`** υποθέτει ότι ξεκινάτε ένα ακόμη πέρασμα του `@ARGV`. Αν το `@ARGV` είναι κενό σε εκείνο το σημείο, η επόμενη χρήση του `<>` διαβάζει το `STDIN`. Αυτό συνήθως δεν είναι αυτό που θέλετε· ορίστε το `@ARGV` ρητά ή ελέγξτε τη σκέτη μορφή `eof`.
- **Η `eof` σε τερματικό καταναλώνει έναν χαρακτήρα.** Η υλοποίηση διαβάζει ένα byte και του εφαρμόζει `ungetc` για να διαπιστώσει αν υπάρχει περισσότερη είσοδος. Σε τερματικά η επιστροφή ενδέχεται να μην επιβιώσει μετά την επίτευξη τέλους αρχείου - μην καλείτε `eof FH` σε τερματικό αφού έχετε ήδη χτυπήσει EOF εκεί, διαφορετικά το handle μπορεί να χάσει την κατάσταση τέλους αρχείου του.
- **Κλεισμένο ή ποτέ-ανοιχτό handle** - η `eof FILEHANDLE` επιστρέφει αληθές. Δεν υπάρχει τρόπος διάκρισης μεταξύ «κλειστό» και «στο EOF» χρησιμοποιώντας μόνο την `eof`· ελέγξτε την `fileno` ή παρακολουθήστε την κατάσταση ανοίγματος μόνοι σας αν αυτό έχει σημασία.
- **Η `eof(ARGV)` ισούται με τη σκέτη `eof` μέσα σε `while (<>)`.** Και οι δύο ρωτούν για το τρέχον αρχείο `ARGV`. Προτιμήστε τη σκέτη `eof` για συντομία και για να γίνει η μορφή `eof()` με παρενθέσεις οπτικά διακριτή.
- **Σπάνια χρειάζεστε καθόλου την `eof`.** Οι τελεστές εισόδου επιστρέφουν `undef` στο τέλος αρχείου, οπότε τα `while (my $line = <$fh>) { ... }` και `while (<>) { ... }` χειρίζονται τον τερματισμό χωρίς κανέναν έλεγχο `eof`. Καταφύγετε στην `eof` μόνο όταν χρειάζεται να ενεργήσετε *στο* όριο - τελευταία γραμμή, επαναφορά ανά αρχείο, δείκτης τελικού αρχείου.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `readline` - διαβάζει μια γραμμή από handle· επιστρέφει `undef` στο τέλος αρχείου, κάτι που αντικαθιστά τις περισσότερες χρήσεις της `eof`
- `read` - μπλοκαριστική ανάγνωση σταθερού πλήθους bytes· επιστρέφει `0` στο τέλος αρχείου
- `getc` - διαβάζει έναν μόνο χαρακτήρα· επιστρέφει `undef` στο τέλος αρχείου
- `close` - κλείσιμο handle· ο συνηθισμένος συνεργάτης της `eof` στο ιδίωμα επαναφοράς ανά αρχείο `close ARGV if eof`

- ARGV - το μαγικό handle που τροφοδοτείται από τον <> και που ρωτούν η eof και η eof()

Ροή ελέγχου

eval

Εκτελεί ένα κομμάτι κώδικα Perl με τυχόν μοιραίο σφάλμα να παγιδεύεται αντί να σκοτώνει το πρόγραμμα.

Η eval είναι ταυτόχρονα ο μηχανισμός πιασίματος εξαιρέσεων της Perl και ο φορτωτής κώδικα κατά τον χρόνο εκτέλεσης. Έρχεται σε δύο πραγματικές μορφές: **eval μπλοκ** (eval { ... }), που πιάνει εξαιρέσεις από ήδη μεταγλωττισμένο κώδικα, και **eval συμβολοσειράς** (eval EXPR), που μεταγλωττίζει και εκτελεί μια συμβολοσειρά πηγαίου κώδικα Perl σε χρόνο εκτέλεσης. Όπως και να έχει, μια die που διαφορετικά θα τερμάτιζε το πρόγραμμα γίνεται ανάθεση στην $@ και κανονική επιστροφή.

Σύνοψη

```
eval BLOCK                    # catch exceptions in compiled code
eval EXPR                     # compile and run source at runtime
eval                          # short for: eval $_
```

Τι επιστρέφεται

Η τιμή της τελευταίας έκφρασης που αποτιμήθηκε μέσα στο μίνι-πρόγραμμα, ή η τιμή που παραδόθηκε σε ρητή return. Η αποτίμηση γίνεται στο ίδιο περιβάλλον με την περιβάλλουσα eval - κενό, βαθμωτό ή λίστας - οπότε η wantarray μέσα σε eval αναφέρει το περιβάλλον του καλούντος, όχι «κάποιο περιβάλλον eval».

Αν κάτι μέσα εγείρει εξαίρεση (die, διαίρεση δια του μηδενός, συντακτικό σφάλμα σε eval συμβολοσειράς, οποιοδήποτε άλλο παγιδεύσιμο μοιραίο), η eval επιστρέφει undef σε βαθμωτό περιβάλλον και την κενή λίστα σε περιβάλλον λίστας, και η εξαίρεση προσγειώνεται στην $@:

```
my $n = eval { risky() };   # undef on failure, $@ holds the error
if ($@) { ... }
```

Σε επιτυχία, η $@ τίθεται στην κενή συμβολοσειρά. Κάθε ολοκληρωμένη eval αγγίζει την $@ - δεν υπάρχει επιλογή «αφήστε τη ήσυχη». Ένας τελεστής ροής ελέγχου που παρακάμπτει την κανονική έξοδο (last, goto έξω από το μπλοκ) μπορεί να αφήσει την $@ αμετάβλητη, που είναι οριακή περίπτωση, όχι χαρακτηριστικό στο οποίο να βασίζεστε.

Καθολική κατάσταση που επηρεάζει

- $@ - εγγράφεται σε κάθε κανονική έξοδο της eval: η εκτοξευμένη εξαίρεση σε αποτυχία, η κενή συμβολοσειρά σε επιτυχία. Διαβάζεται για να καθοριστεί αν η eval έπιασε κάτι.
- $^S - αντικατοπτρίζει αν η Perl βρίσκεται αυτή τη στιγμή μέσα σε eval (ή try). undef κατά την ανάλυση, αληθής κατά την εκτέλεση μέσα σε μπλοκ eval/try,

ψευδής στο ανώτατο επίπεδο. Διαβάζεται κυρίως από χειριστές `$SIG{__DIE__}` που θέλουν να ενεργήσουν διαφορετικά για πιασμένες έναντι μη πιασμένων εξαιρέσεων.

- `$SIG{__DIE__}` - πυροδοτείται σε **κάθε** `die`, συμπεριλαμβανομένων εκείνων που μια περικλείουσα `eval` πρόκειται να πιάσει. Χρησιμοποιήστε `local $SIG{__DIE__}` μέσα σε `eval` για να καταστείλετε έναν καθολικά εγκαταστημένο γάντζο για τη διάρκεια της παγίδας.
- `$SIG{__WARN__}` - δεν επηρεάζεται από την `eval`. Οι προειδοποιήσεις που εκπέμπονται μέσα στο μπλοκ εξακολουθούν να ακολουθούν την κανονική διαδρομή προειδοποίησης· η `eval` δεν τις ανακατευθύνει στην `$@`.
- `$_` - η πηγή για τη μορφή χωρίς όρισμα (`eval` χωρίς τίποτα μετά από αυτή είναι ακριβώς `eval $_`).

eval μπλοκ έναντι eval συμβολοσειράς

Οι δύο μορφές μοιάζουν παρόμοιες αλλά κάνουν πολύ διαφορετικά πράγματα.

- **eval μπλοκ** (`eval { ... }`) αναλύεται και μεταγλωττίζεται μαζί με το περιβάλλον πρόγραμμα. Σε χρόνο εκτέλεσης είναι απλώς ένα σήμα που λέει «πιάσε όποια `die` εκτοξευθεί από μέσα από αυτό το μπλοκ». Επειδή ο κώδικας είναι ήδη μεταγλωττισμένος, η eval μπλοκ είναι φθηνή και ασφαλής - ο μεταγλωττιστής τον έχει ήδη ελέγξει για συντακτικά σφάλματα. Αυτή είναι η μορφή στην οποία πρέπει να καταφεύγετε όποτε ο στόχος είναι ο χειρισμός εξαιρέσεων.
- **eval συμβολοσειράς** (`eval EXPR`) αποτιμά την `EXPR` σε βαθμωτό περιβάλλον, στη συνέχεια αναλύει και εκτελεί την προκύπτουσα συμβολοσειρά σαν να ήταν μπλοκ στη λεκτική εμβέλεια του περιβάλλοντος κώδικα. Η ανάλυση γίνεται κάθε φορά που προσεγγίζεται η `eval`. Ένα συντακτικό σφάλμα στη συμβολοσειρά γίνεται `$@` σε χρόνο εκτέλεσης, όχι αποτυχία κατά τη μεταγλώττιση του περιβάλλοντος προγράμματος. Αυτή είναι η μορφή για τη φόρτωση κώδικα που αποφασίζεται σε χρόνο εκτέλεσης - προαιρετικά χαρακτηριστικά, εκφράσεις παρεχόμενες από τον χρήστη, δυναμικά κατασκευασμένες υπορουτίνες - και κουβαλά τα κόστη και τους κινδύνους που αυτό συνεπάγεται.

Και οι δύο μορφές εκτελούνται στη λεκτική εμβέλεια του περιβάλλοντος κώδικα: οι εξωτερικές μεταβλητές `my` είναι ορατές, και αναθέσεις σε μεταβλητές πακέτου, ορισμοί υπορουτινών και ορισμοί φορμάτ που γίνονται μέσα στην `eval` παραμένουν αφού αυτή επιστρέψει.

Το συμβόλαιο της $@

Μετά την κανονική ολοκλήρωση μιας `eval`, η `$@` είναι πάντα ένα από:

- Η κενή συμβολοσειρά - το μπλοκ έτρεξε μέχρι το τέλος χωρίς εξαίρεση.
- Συμβολοσειρά που τελειώνει σε newline - `die "message\n"` ή συμβολοσειριακή εξαίρεση που ο καλούμενος έχει ήδη τερματίσει.
- Συμβολοσειρά με προσαρτημένη τοποθεσία (`... at FILE line N.\n`) - `die "message"` χωρίς τελικό newline, ή σφάλμα χρόνου εκτέλεσης που εγείρει η ίδια η Perl.

- Μια αναφορά, τυπικά ένα blessed αντικείμενο εξαίρεσης - η `die` που κλήθηκε με αναφορά.

Ελέγξτε την `$@` αμέσως μετά την `eval` και πουθενά αλλού. Οποιαδήποτε κλήση μεθόδου, έλεγχος `ref`, έλεγχος `isa` ή ταίριασμα regex κατά τον χειρισμό μπορεί να καλέσει η ίδια εσωτερικά μια `eval` (για παράδειγμα κατά την autoload ή την επίλυση υπερφόρτωσης) και να γράψει πάνω στην `$@` πριν τελειώσετε την επιθεώρησή της. Αντιγράψτε πρώτα σε λεκτική μεταβλητή:

```
eval { work() };
if (my $err = $@) {          # safe snapshot
    if (ref $err && $err->isa('MyApp::Error')) { handle($err) }
    else                                        { die $err      }
}
```

Εμφωλευμένη eval και local $@

Η εξωτερική `eval` βλέπει ό,τι κρατά η `$@` όταν τελειώσει το εξωτερικό μπλοκ - όχι ό,τι κρατούσε στο ενδιάμεσο. Αν ο εσωτερικός κώδικας έχει δική του `eval`, αυτή η εσωτερική `eval` θέτει την `$@` (στην κενή συμβολοσειρά σε επιτυχία), που μπορεί να αλλοιώσει ένα σφάλμα που ο εξωτερικός κώδικας ήθελε να διαφυλάξει. Η προστασία είναι `local $@` γύρω από την εσωτερική εργασία:

```
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 5.44 inherits) the order is fixed and the temporary is unnecessary.

Ασφάλεια της eval συμβολοσειράς

Η eval συμβολοσειράς εκτελεί ό,τι βρίσκεται στην έκφραση. Αν οποιοδήποτε μέρος αυτής της συμβολοσειράς προήλθε από μη έμπιστη είσοδο - αρχείο, δικτυακό άκρο, φόρμα χρήστη - η `eval` παραδίδει στον επιτιθέμενο τον πλήρη διερμηνέα Perl. Πρακτικοί κανόνες:

- Μη διαβιβάζετε ποτέ είσοδο χρήστη μέσω `eval EXPR` ατροποποίητη. «Είναι μόνο αριθμός» δεν αποτελεί δικλείδα - και το `"1; system 'rm -rf /'"` είναι μόνο μια συμβολοσειρά.
- Αν χρειάζεται πραγματικά κώδικας επιλεγμένος σε χρόνο εκτέλεσης, περιορίστε την είσοδο έναντι μιας αυστηρής λίστας επιτρεπτών *πριν* την παρεμβολή, και προτιμήστε καθοδηγούμενη από δεδομένα διανομή (hash από coderefs) έναντι παραγωγής πηγαίου κώδικα.

- Τιμές κινητής υποδιαστολής που παρεμβάλλονται σε πηγαίο κώδικα είναι εύθραυστες. Υπό `use locale` ο διαχωριστής δεκαδικών μπορεί να μην είναι τελεία, και τιμές όπως `"NaN"` ή `"Infinity"` μετατρέπονται σε συμβολοσειρές με γράμματα που ο αναλυτής διαβάζει ως barewords, όχι αριθμούς.
- Φίλτρα πηγαίου κώδικα που ενεργοποιούνται μέσα σε eval συμβολοσειράς διαρρέουν στην περιβάλλουσα εμβέλεια αρχείου (εκτός αν χρησιμοποιηθεί η `evalbytes`). Αυτή είναι μακροχρόνια ιδιοτροπία, όχι χαρακτηριστικό.

Η eval μπλοκ δεν έχει κανένα από αυτά τα προβλήματα - ο κώδικας μεταγλωττίστηκε μαζί με το υπόλοιπο πρόγραμμα.

`BEGIN` μέσα σε eval συμβολοσειράς

Μπλοκ `BEGIN { ... }` ενσωματωμένα σε eval συμβολοσειράς εκτελούνται αμέσως, πριν τρέξει ο υπόλοιπος κώδικας της `eval`, ταιριάζοντας τη συμπεριφορά τους σε μια κανονική μονάδα μεταγλώττισης. Θέστε την `${^MAX_NESTED_EVAL_BEGIN_BLOCKS}` σε `0` ώστε όποιο ενσωματωμένο `BEGIN` να εκτοξεύει εξαίρεση αντί να εκτελείται:

```
local ${^MAX_NESTED_EVAL_BEGIN_BLOCKS} = 0;
eval $untrusted;            # any BEGIN inside $untrusted dies
```

Αυτό δεν κάνει ασφαλή την eval συμβολοσειράς - απλώς αφαιρεί μία συγκεκριμένη επίθεση στις παρενέργειες κατά τη μεταγλώττιση.

`try/catch`

Under `use feature 'try'` (stable since 5.40, available in 5.44), the `try/catch/finally` construct provides dedicated exception-handling syntax:

```
use feature 'try';

try {
    risky();
}
catch ($e) {
    warn "caught: $e";
}
finally {
    cleanup();
}
```

Διαφορές από το `eval { ... }; if ($@) { ... }`:

- Η πιασμένη τιμή δένεται σε νέα λεκτική `$e` (ή σε όποιο όνομα δοθεί), όχι στην `$@`. Δεν χρειάζεται ο χορός στιγμιότυπου `my $err = $@`.
- Η `$@` δεν τίθεται ως παρενέργεια - το μπλοκ `try` δεν έχει καθολική παρενέργεια σε αυτή.
- Η `try` δεν είναι από μόνη της βρόχος, αλλά οι τελεστές ροής ελέγχου συμπεριφέρονται ανάλογα με την `eval BLOCK`: η `return` μέσα σε `try` επιστρέφει από την περικλείουσα υπορουτίνα, και οι `next`/`last`/`redo` στοχεύουν εξωτερικό βρόχο.

- `finally` runs on every exit path and cannot itself use `return`/`next`/`last`/`redo`. It is marked experimental in 5.44 and emits a warning under `experimental::try`.

Καταφύγετε στην try/catch σε νέο κώδικα. Το ιδίωμα eval-συν-$@ παραμένει σωστό και εξακολουθεί να είναι το σωστό εργαλείο όποτε ο περιβάλλων κώδικας πρέπει να τρέχει σε παλαιότερες Perls ή να μην απαιτεί την προστασία feature.

Παραδείγματα

Παγίδευση σφάλματος χρόνου εκτέλεσης και συνέχιση:

```
my $q = eval { $x / $y };
if ($@) { warn "division failed: $@"; $q = 0 }
```

Ανίχνευση αν ένα χαρακτηριστικό είναι διαθέσιμο χωρίς τερματισμό κατά την εκκίνηση:

```
my $have_socket = eval { socket(my $s, PF_INET, SOCK_STREAM, 0); 1 }
 ↪;
die "no sockets on this build" unless $have_socket;
```

Φόρτωση αρθρώματος του οποίου η απουσία θα πρέπει να είναι ανεκτή:

```
my $json = eval { require JSON::XS; JSON::XS->new } //
           do { require JSON::PP; JSON::PP->new };
```

Πιάσιμο blessed εξαίρεσης, επανεκτόξευση οποιουδήποτε άλλου:

```
eval {
    parse($input);
};
if (my $err = $@) {
    if (ref($err) && $err->isa('MyApp::ParseError')) {
        report($err);
    }
    else {
        die $err;
    }
}
```

Εμφωλευμένη eval με `local $@` ώστε ένα εσωτερικό προαιρετικό βήμα να μη σβήσει ένα εξωτερικό διαγνωστικό:

```
eval {
    do_main_work();
    { local $@; eval { maybe_cleanup() } }   # ignore cleanup errors
    die "main still failed\n" if $something_else_wrong;
};
warn $@ if $@;
```

Καταστολή ενός καθολικά εγκαταστημένου γάντζου $SIG{__DIE__} για τη διάρκεια ιδιωτικής παγίδας:

```
eval {
    local $SIG{__DIE__};            # don't trigger hooks while we
↪probe
    $answer = $x / $y;
};
warn $@ if $@;
```

Ένας γάντζος $SIG{__DIE__} που καταγράφει μόνο μη πιασμένες εξαιρέσεις, χρησιμοποιώντας τη $^S για να διακρίνει τις δύο περιπτώσεις:

```
$SIG{__DIE__} = sub {
    return if $^S;              # inside eval/try - let it propagate
    log_fatal(@_);              # top level - record before exit
};
```

Δυναμικά κατασκευασμένη διανομή (eval συμβολοσειράς που χρησιμοποιείται σκόπιμα, η είσοδος δεν προέρχεται από χρήστη):

```
for my $op (qw(add sub mul div)) {
    eval "sub ${op}_of { \$_[0] "
       . { add=>'+', sub=>'-', mul=>'*', div=>'/' }->{$op}
       . " \$_[1] }";
    die $@ if $@;
}
```

Οριακές περιπτώσεις

- **Χωρίς όρισμα** - η `eval;` είναι `eval $_`. Σπάνια αυτό που θέλετε· γράψτε ρητά `eval { ... }` ή `eval EXPR`.
- **Τελικό ερωτηματικό** - μπορεί να παραλειφθεί από το τέλος του EXPR ή του BLOCK. Τα `eval "1+2"` και `eval "1+2;"` είναι ισοδύναμα.
- **Η `eval BLOCK` δεν είναι βρόχος** - οι `next`, `last` και `redo` δεν μπορούν να αφήσουν ή να επανεκκινήσουν το μπλοκ. Ψάχνουν περικλείοντα βρόχο και κράζουν αν δεν υπάρχει.
- **Η `return` μέσα σε `eval BLOCK`** επιστρέφει από την *περικλείουσα υπορουτίνα*, όχι από την `eval` - το μπλοκ `eval` δεν είναι σώμα συνάρτησης. Για να παράγετε τιμή από το μπλοκ, βάλτε την τιμή ως την τελική έκφραση.
- **Οι προειδοποιήσεις δεν παγιδεύονται.** Η `warn` μέσα σε `eval` εξακολουθεί να γράφει στο STDERR. Δρομολογήστε τις στην `$@` μόνο αν θέσετε εσείς οι ίδιοι την `$SIG{__WARN__}` να το κάνει.
- **Ροή ελέγχου έξω από την `eval`** - μια `goto`, `last` ή `next` που πηδά έξω από το μπλοκ παρακάμπτει την κανονική έξοδο της `eval` και συνεπώς ενδέχεται να αφήσει την `$@` ανέγγιχτη. Μην κατασκευάζετε λογική που εξαρτάται από αυτό.
- **Εισαγωγικά στην `EXPR` - `eval $x` έναντι `eval "$x"`** συμπεριφέρονται πανομοιότυπα: και τα δύο εκτελούν τον κώδικα που περιέχει η $x. Τα διπλά εισαγωγικά προσθέτουν οπτικό θόρυβο, όχι σημασιολογία.

- **`eval '$x'` έναντι `eval { $x }`** - και τα δύο επιστρέφουν την τιμή της `$x`. Η μορφή μπλοκ μεταγλωττίζεται μαζί με το περιβάλλον πρόγραμμα· προτιμήστε την εκτός αν ο κώδικας χρειάζεται πράγματι να κατασκευαστεί σε χρόνο εκτέλεσης.
- **`eval ''` μέσα στο πακέτο DB** - δεν βλέπει τη συνηθισμένη περιβάλλουσα λεκτική εμβέλεια αλλά την εμβέλεια του πρώτου μη-DB καλούντος. Αυτό το αντιμετωπίζουν μόνο όσοι γράφουν debugger.
- **Αποτυχίες φόρτωσης XS** - κάποια προβλήματα δυαδικής διεπαφής κατά την `require` ενός αρθρώματος XS είναι μοιραία ακόμη και μέσα σε `eval`, εκτός αν τεθεί η `$ENV{PERL_DL_NONLAZY}`. Η συνηθισμένη περίπτωση είναι ασυμβατότητα εκδόσεων μεταξύ του τρέχοντος διερμηνέα και ενός προ-κατασκευασμένου `.so`. Το pperl δεν φορτώνει XS· η επιφύλαξη ισχύει όταν κώδικας μέσα σε `eval` αναθέτει τη φόρτωση σε ένα κλασικό perl5.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `die` - το αντίστοιχο: εκτοξεύει την εξαίρεση που πιάνει η `eval`. Ο κανόνας του τελικού newline στο μήνυμα διαμορφώνει το τι προσγειώνεται στην `$@`.
- `do` - `do FILE` διαβάζει και αποτιμά ένα αρχείο όπως η eval συμβολοσειράς διαβάζει και αποτιμά μια συμβολοσειρά· χρήσιμο για αρχεία ρυθμίσεων γραμμένα σε Perl.
- `return` - μέσα σε `eval BLOCK` επιστρέφει από την περικλείουσα υπορουτίνα, όχι από το μπλοκ· χρησιμοποιήστε μια τελική έκφραση για να επιστρέψετε τιμή από την ίδια την `eval`.
- `require` - συχνά τυλίγεται σε `eval` για ανίχνευση προαιρετικών αρθρωμάτων χωρίς τερματισμό κατά τη μεταγλώττιση.
- `evalbytes` - eval συμβολοσειράς που μεταχειρίζεται το όρισμά της ως συμβολοσειρά bytes και επιτρέπει στα φίλτρα πηγαίου κώδικα να λειτουργούν κανονικά· χρησιμοποιήστε την όταν έχει σημασία η σημασιολογία επιπέδου byte.
- `$@` - εκεί όπου προσγειώνεται μια πιασμένη εξαίρεση, καθαρίζεται στην κενή συμβολοσειρά σε επιτυχημένη `eval`. Η σελίδα της οικογένειας error καλύπτει επίσης τον κλασικό κίνδυνο αλλοίωσης της `$@`.
- `$^S` - λέει στους χειριστές `$SIG{__DIE__}` και `$SIG{__WARN__}` αν είναι αυτή τη στιγμή ενεργή κάποια `eval` ή `try`.
- `$SIG{__DIE__}` - γάντζος πριν την εκτόξευση· πυροδοτείται ακόμη και για εξαιρέσεις που μια περικλείουσα `eval` πρόκειται να πιάσει, που είναι ο συνηθισμένος λόγος για να την `local`-οποιήσετε για τη διάρκεια μιας παγίδας.

Ροή ελέγχου

evalbytes

Μεταγλωττίζει και εκτελεί μια συμβολοσειρά πηγαίου κώδικα Perl, αναγκάζοντας την πηγή να ερμηνευτεί ως **bytes** αντί ως χαρακτήρες.

Η `evalbytes` είναι το byte-προσανατολισμένο δίδυμο της string `eval`. Δέχεται μια `EXPR` (ή την `$_` όταν η `EXPR` παραλείπεται), αντιμετωπίζει τη συμβολοσειρά ως ωμή ροή bytes και την αναλύει ως πηγαίο κώδικα Perl. Η ανάλυση είναι μονωμένη από την pragma `use utf8` του καλούντος: όποιους κανόνες κωδικοποίησης δηλώνει η ίδια η συμβολοσειρά, αυτοί υπερισχύουν. Χρησιμοποιήστε την όταν ο κώδικας που πρόκειται να μεταγλωττίσετε διαβάστηκε από δίσκο, από το δίκτυο ή από κάποιο βήμα build και χρειάζεστε ο αναλυτής να δει ακριβώς τα bytes που του δώσατε.

Σύνοψη

```
evalbytes EXPR
evalbytes
CORE::evalbytes EXPR    # when the feature is not enabled
```

Τι επιστρέφεται

Σε βαθμωτό περιβάλλον, η τιμή της τελευταίας έκφρασης που αποτιμήθηκε· σε περιβάλλον λίστας, μια λίστα. Σε σφάλμα μεταγλώττισης ή χρόνου εκτέλεσης, `undef` (ή κενή λίστα) και η `$@` τίθεται στο μήνυμα σφάλματος· η `$@` είναι η κενή συμβολοσειρά σε επιτυχία. Η μορφή επιστροφής ταιριάζει ακριβώς με την string `eval` - η μόνη διαφορά είναι ο τρόπος αποκωδικοποίησης του ίδιου του κειμένου της πηγής.

```
my $result = evalbytes $source;
die $@ if $@;
```

Πώς διαφέρει από την string `eval`

Δύο συγκεκριμένες συμπεριφορές διακρίνουν την `evalbytes` από την `eval EXPR`:

- **Η πηγή αναλύεται ως bytes.** Κάθε οκτάδα στη συμβολοσειρά είναι ξεχωριστός χαρακτήρας εισόδου για τον λεκτικό αναλυτή, ανεξάρτητα από το αν η συμβολοσειρά είναι εσωτερικά σημειωμένη ως `utf8`. Μια ακολουθία UTF-8 πολλαπλών bytes στην πηγή εμφανίζεται στον αναλυτή ως τα επιμέρους bytes, όχι ως το αποκωδικοποιημένο σημείο κώδικα.
- **Σημεία κώδικα πάνω από το 255 είναι σκληρό σφάλμα.** Αν η συμβολοσειρά περιέχει οποιονδήποτε χαρακτήρα του οποίου ο τακτικός αριθμός υπερβαίνει το `0xFF`, δεν μπορεί να είναι ροή bytes και η μεταγλώττιση αποτυγχάνει με `Wide character in eval` αποθηκευμένο στην `$@`. Υποβαθμίστε πρώτα με την `utf8::encode` αν η πηγή κρατά αυτή τη στιγμή αποκωδικοποιημένο κείμενο.

Οι `use utf8` και `no utf8` **μέσα** στον υπό αξιολόγηση κώδικα έχουν τη συνηθισμένη τους επίδραση: η pragma ενεργοποιείται μόλις την φτάσει ο αναλυτής και επηρεάζει το υπόλοιπο εκείνης της μεταγλώττισης. Είναι το εξωτερικό πλαίσιο της πηγής που είναι καθηλωμένο στα bytes, όχι αυτό που δηλώνει η ίδια η πηγή.

Ενεργοποίηση του ονόματος χωρίς CORE::

Η evalbytes είναι feature-gated. Για να την καλέσετε ως bareword:

```
use feature 'evalbytes';
```

Μια δήλωση use v5.16 (ή υψηλότερη) ενεργοποιεί αυτόματα το χαρακτηριστικό evalbytes, μαζί με το υπόλοιπο bundle χαρακτηριστικών :5.16. Χωρίς κάποιο από τα δύο, το μη πλήρως κατονομασμένο όνομα δεν αναγνωρίζεται και πρέπει να γράψετε ρητά CORE::evalbytes - χρήσιμο μέσα σε παραγόμενο κώδικα ή σε μονόγραμμα που δεν φέρουν δήλωση έκδοσης.

```
use v5.16;
my $n = evalbytes '1 + 2';        # 3
```

```
# no feature, no version bundle:
my $n = CORE::evalbytes '1 + 2';  # 3
```

Καθολική κατάσταση που επηρεάζει

- $@ - καθαρίζεται πριν τη μεταγλώττιση, κατόπιν τίθεται στο μήνυμα σφάλματος σε αποτυχία ή στην κενή συμβολοσειρά σε επιτυχία. Ίδιο συμβόλαιο με την eval.
- $_ - διαβάζεται ως το κείμενο πηγής όταν η EXPR παραλείπεται.

Source filters ενεργοποιημένα *μέσα* στον υπό αξιολόγηση κώδικα εφαρμόζονται στον ίδιο τον κώδικα, ακριβώς όπως θα έκαναν μέσα σε μια string eval. Το filter βλέπει τα ίδια bytes που βλέπει ο αναλυτής.

Παραδείγματα

Αποτίμηση μιας κυριολεκτικής συμβολοσειράς bytes - καμία έκπληξη, καμία επιλογή κωδικοποίησης:

```
use feature 'evalbytes';
my $n = evalbytes 'my $x = 40; $x + 2';   # 42
```

Αποτίμηση πηγής που διαβάστηκε από αρχείο που είναι γνωστό ότι είναι κωδικοποιημένο σε UTF-8 αλλά για το οποίο η δηλωμένη pragma μέσα στην πηγή είναι αυτό που πρέπει να διέπει την ανάλυση:

```
use feature 'evalbytes';
open my $fh, '<:raw', 'script.pl' or die $!;
my $src = do { local $/; <$fh> };
my @values = evalbytes $src;
die "compile failed: $@" if $@;
```

Η ανάγνωση με :raw διατηρεί τη $src ως συμβολοσειρά bytes. Η evalbytes τότε τροφοδοτεί αυτά τα bytes στον αναλυτή αυτούσια· μια γραμμή use utf8; στην κορυφή του σεναρίου θα κάνει τη δική της δουλειά μόλις τη φτάσει ο αναλυτής.

Συγκρίνετε με τη string eval όταν ο καλών έχει 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";
```

Υπό `use utf8` το ίδιο το κυριολεκτικό `"length('caf\xc3\xa9')"` του καλούντος είναι συμβολοσειρά **χαρακτήρων** για την `eval`, οπότε η `length` μέσα στον υπό αξιολόγηση κώδικα βλέπει έναν χαρακτήρα é. Η `evalbytes` αναγκάζει την ίδια πηγή να αναλυθεί ως η ακολουθία bytes `c a f 0xC3 0xA9`, οπότε η `length` αναφέρει το πλήθος bytes.

Πιάσιμο ενός wide-character σφάλματος:

```
use feature 'evalbytes';
my $src = "print \x{2603}";          # contains U+2603 SNOWMAN
evalbytes $src;
warn "rejected: $@" if $@;           # "Wide character in eval ..."
```

Παράλειψη της EXPR - η πηγή προέρχεται από την `$_`:

```
use feature 'evalbytes';
for ('1+1', '2*3', '10-7') {
    my $r = evalbytes;
    print "$_ = $r\n";
}
```

Οριακές περιπτώσεις

- **Wide character στην πηγή**: οποιοδήποτε σημείο κώδικα πάνω από το `0xFF` ματαιώνει τη μεταγλώττιση και αφήνει `Wide character in eval` στην `$@`. Τρέξτε `utf8::encode` στην πηγή (ή διαβάστε το αρχείο με `:raw`) πριν την κλήση.
- **Η `use utf8` είναι τοπική στον υπό αξιολόγηση κώδικα.** Επηρεάζει κυριολεκτικά συμβολοσειρών και αναγνωριστικά μέσα στο αποτιμώμενο κείμενο μόλις ο αναλυτής φτάσει στη γραμμή της pragma. **Δεν** αλλάζει τον τρόπο που η `evalbytes` πλαισιώνει την εξωτερική πηγή - εκείνο το πλαίσιο είναι πάντα «bytes».
- **Το χαρακτηριστικό δεν είναι ενεργοποιημένο**: χωρίς `use feature 'evalbytes'` και χωρίς bundle `use v5.16+`, το bareword δεν αναγνωρίζεται. Γράψτε `CORE::evalbytes` για να παρακάμψετε την πύλη χαρακτηριστικού.
- **`evalbytes` χωρίς όρισμα και με `$_` undef**: μεταγλωττίζει κενή συμβολοσειρά, πράγμα που πετυχαίνει και επιστρέφει `undef` σε βαθμωτό περιβάλλον, την κενή λίστα σε περιβάλλον λίστας, με την `$@` κενή.
- **Τα source filters** μέσα στον υπό αξιολόγηση κώδικα βλέπουν τη ροή bytes. Ένα filter που περιμένει αποκωδικοποιημένους χαρακτήρες θα δυσλειτουργήσει· γράψτε filters για την `evalbytes` σε όρους bytes.

- **Λεξιλογικά και κατάσταση πακέτου**: ίδιοι κανόνες με τη string `eval`. Ο υπό αξιολόγηση κώδικας μοιράζεται το τρέχον πακέτο του καλούντος και βλέπει τα εντός εμβέλειας λεξιλογικά του καλούντος· οι δικές του μεταβλητές `my` εξαφανίζονται όταν ολοκληρωθεί η eval.
- **`die` μέσα σε `evalbytes`**: πιάνεται με τον ίδιο τρόπο όπως για την `eval`. Η εκτοξευμένη τιμή προσγειώνεται στην `$@`· η εκτέλεση συνεχίζεται μετά την κλήση `evalbytes`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `eval` - το αντίστοιχο σε character-mode· αναλύει την EXPR σύμφωνα με το Unicode πλαίσιο του καλούντος, οπότε η `use utf8` στον καλούντα αλλάζει τον τρόπο ερμηνείας της πηγής
- `die` - εκτοξεύει εξαίρεση μέσα από υπό αξιολόγηση κώδικα· θέτει την `$@` όπως κάθε άλλο σφάλμα χρόνου εκτέλεσης
- `do` - μεταγλωττίζει και εκτελεί ένα αρχείο· παίρνει την πηγή του από τον δίσκο αντί από συμβολοσειρά και δεν επηρεάζεται από το `utf8` πλαίσιο του καλούντος
- `require` - φορτώνει και μεταγλωττίζει ένα αρχείο μία φορά· το πρότυπο εργαλείο όταν η πηγή βρίσκεται στον δίσκο και πρέπει να εκτελεστεί μόνο κατά την πρώτη αναφορά
- `utf8` - η pragma που ελέγχει τη σημασιολογία χαρακτήρων έναντι bytes στα κυριολεκτικά συμβολοσειρών· η `evalbytes` υπάρχει ακριβώς για να καθηλώσει το εξωτερικό πλαίσιο στα bytes ανεξάρτητα από αυτή την pragma
- `$@` - η θέση σφάλματος που συμπληρώνεται σε αποτυχία της `evalbytes`, καθαρίζεται σε επιτυχία

Διεργασίες

exec

Εγκατάλειψη αυτού του προγράμματος και εκτέλεση άλλου στην ίδια διεργασία.

Η `exec` αντικαθιστά την τρέχουσα εικόνα διεργασίας με ένα άλλο πρόγραμμα. **Δεν** επιστρέφει σε επιτυχία - η εκτέλεση συνεχίζεται στο νέο πρόγραμμα, με το ίδιο id διεργασίας, ανοιχτά filehandles υποκείμενα στη σημαία close-on-exec, και όποιο περιβάλλον ίσχυε. Επιστρέφει ψευδές (και θέτει την `$!`) μόνο όταν ο στόχος δεν μπόρεσε να εκτοξευθεί *και* εκτελέστηκε άμεσα παρά μέσω του κελύφους. Καταφύγετε στην `system` όταν θέλετε το παιδί να εκτελεστεί και ο έλεγχος να επιστρέψει.

Σύνοψη

```
exec LIST
exec PROGRAM LIST
exec { PROGRAM } LIST
```

Τι επιστρέφεται

Τίποτα, σε επιτυχία - η διεργασία έχει αντικατασταθεί και αυτός ο διερμηνέας Perl έχει χαθεί. Σε αποτυχία η exec επιστρέφει ψευδές και αφήνει την αιτία στην $!. Η αποτυχία είναι ανιχνεύσιμη μόνο για τη διαδρομή άμεσου exec· όταν το όρισμα παραδίδεται στο κέλυφος, ένα πρόγραμμα που λείπει αναφέρεται από το ίδιο το κέλυφος και η exec εξακολουθεί να θεωρείται ότι «πέτυχε» στην εκτόξευση του /bin/sh.

```
exec '/usr/bin/vi', $file
    or die "couldn't exec vi: $!";
```

Επειδή η exec κανονικά δεν επιστρέφει, η Perl εκπέμπει προειδοποίηση υπό use warnings αν εμφανίζεται σε περιβάλλον κενού ακολουθούμενη από άλλη δήλωση που δεν είναι die, warn ή exit. Είτε χειριστείτε την αποτυχία ενσωματωμένα με or die, είτε τυλίξτε την κλήση σε μπλοκ για να σιωπήσετε την προειδοποίηση:

```
exec('foo') or die "couldn't exec foo: $!";
{ exec('foo') }; die "couldn't exec foo: $!";
```

Κέλυφος ή όχι κέλυφος - ο κανόνας της μονής συμβολοσειράς

Η exec επιλέγει μεταξύ δύο μηχανισμών εκτόξευσης με βάση τη μορφή της LIST:

- **Δύο ή περισσότερα ορίσματα** → η exec καλεί την execvp(3) άμεσα με τη λίστα. Δεν εμπλέκεται κέλυφος, καμία ερμηνεία μετα-χαρακτήρων, κανένας διαχωρισμός σε λέξεις.
- **Ακριβώς ένα όρισμα** → η exec σαρώνει αυτή τη μοναδική συμβολοσειρά για μετα-χαρακτήρες κελύφους. Αν βρεθούν, ολόκληρη η συμβολοσειρά περνά στο /bin/sh -c για ανάλυση. Αν δεν βρεθούν, η συμβολοσειρά διαχωρίζεται σε λευκούς χαρακτήρες και παραδίδεται απευθείας στην execvp.

Ο κανόνας έχει σημασία τόσο για ορθότητα όσο και για ασφάλεια. Μια συμβολοσειρά ενός ορίσματος που περιέχει κενό αλλά καθόλου μετα-χαρακτήρες διαχωρίζεται σε λέξεις και εκτελείται άμεσα - κάτι που συχνά είναι αυτό που θέλετε. Μια συμβολοσειρά ενός ορίσματος που περιέχει |, >, $, ;, χαρακτήρες glob ή εισαγωγικά περνά μέσω του κελύφους, και το κέλυφος θα επεκτείνει ευχαρίστως μπαλαντέρ, θα ερμηνεύσει μεταβλητές και θα διαχωρίσει με βάση το $IFS.

```
exec '/bin/echo', 'Your arguments are:', @ARGV;  # direct execvp
exec "sort $outfile | uniq";                     # pipeline → shell
exec "ls /tmp";                                  # no metachars →␣
↪split + execvp
```

Για να **εξαναγκάσετε** τη διαδρομή χωρίς κέλυφος ακόμη και με μία λογική εντολή, χρησιμοποιήστε τη μορφή έμμεσου αντικειμένου που περιγράφεται στη συνέχεια.

Μορφή έμμεσου αντικειμένου - ορισμός του argv[0], παράκαμψη του κελύφους

Η τοποθέτηση ενός μπλοκ ή βαθμωτού πριν την LIST (χωρίς κόμμα) λέει στην exec δύο πράγματα ταυτόχρονα: εκτέλεσε *αυτό* το πρόγραμμα, και πέρασε *εκείνα* τα ορίσματα ως argv. Το πρώτο στοιχείο της LIST γίνεται το argv[0] - το όνομα που βλέπει το παιδί για τον εαυτό του - το οποίο δεν χρειάζεται να ταιριάζει με τη διαδρομή του προγράμματος.

```
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
```

Η μορφή έμμεσου αντικειμένου αντιμετωπίζει πάντα την LIST ως λίστα πολλαπλών τιμών, ακόμη και αν έχει ένα μόνο στοιχείο. Αυτό απενεργοποιεί εντελώς τη σάρωση μετα-χαρακτήρων:

```
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]
```

Στη δεύτερη μορφή δεν υπάρχει πρόγραμμα στον δίσκο κυριολεκτικά ονομαζόμενο echo surprise, οπότε η execvp αποτυγχάνει, η exec επιστρέφει ψευδές, και η $! κρατά την αιτία. Αυτή είναι η ασφαλής συμπεριφορά: ο καλών μπορεί να αποφασίσει τι θα κάνει, αντί να εκτελεί το κέλυφος κάτι απροσδόκητο.

Καθολική κατάσταση που επηρεάζει

- $! - τίθεται σε αποτυχία στο errno από την υποκείμενη κλήση execvp(3). Έχει νόημα μόνο όταν η exec πράγματι επιστρέφει.
- $? - μένει αμετάβλητη από την ίδια την exec, αλλά μια αποτυχημένη προσπάθεια μέσω κελύφους μπορεί να αφήσει την κατάσταση εξόδου του κελύφους ορατή σε μια περικλείουσα system.
- Τα ανοιχτά filehandles της διεργασίας, το περιβάλλον (%ENV), ο τρέχων κατάλογος εργασίας, η umask, οι διαθέσεις σημάτων και τα όρια πόρων μεταφέρονται όλα στο πρόγραμμα αντικατάστασης - η exec αντικαθιστά το πρόγραμμα, όχι τη διεργασία.

Παραδείγματα

Εκτέλεση προγράμματος με ρητή λίστα ορισμάτων, άμεση execvp:

```
exec '/usr/bin/git', 'status', '--short'
    or die "exec git: $!";
```

Σκόπιμη παράδοση γραμμής εντολής στο κέλυφος, με χρήση μετα-χαρακτήρων:

```
exec "find . -name '*.tmp' -print0 | xargs -0 rm"
    or die "exec pipeline: $!";
```

Ορισμός διακριτού argv[0] ώστε η γραμμή ps του παιδιού να αναγνωρίζει τον ρόλο του:

```
exec { '/usr/local/bin/worker' } 'worker[queue=email]', @flags
    or die "exec worker: $!";
```

Fork-and-exec - το κλασικό μοτίβο για εκτόξευση παιδιού χωρίς αντικατάσταση του εαυτού σας:

```
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    exec '/usr/bin/gzip', '-9', $file
        or die "exec gzip: $!";
}
waitpid $pid, 0;
```

Εγγύηση της διαδρομής χωρίς κέλυφος όταν η λίστα ορισμάτων συναρμολογήθηκε από δεδομένα:

```
my @cmd = build_command();            # may contain one element or
↪many
exec { $cmd[0] } @cmd
    or die "exec $cmd[0]: $!";
```

Οριακές περιπτώσεις

- **Δεν επιστρέφει ποτέ σε επιτυχία.** Οποιοσδήποτε κώδικας μετά από σκέτη `exec` είναι προσβάσιμος μόνο σε αποτυχία. Υπό `use warnings`, μια επόμενη δήλωση εκτός των `die`, `warn` ή `exit` πυροδοτεί προειδοποίηση - υπαινιγμός της Perl ότι πιθανώς εννοούσατε `system`.
- **Παγίδα μονού ορίσματος.** Η `exec $cmd` όπου το `$cmd` προήλθε από είσοδο χρήστη αποτελεί διάνυσμα έγχυσης κελύφους αν το `$cmd` περιέχει μετα-χαρακτήρες. Χρησιμοποιήστε τη μορφή έμμεσου αντικειμένου ή περάστε μια προ-διαχωρισμένη λίστα.
- **Ενταμίευση εξόδου.** Η Perl προσπαθεί να εκκενώσει handles ανοιγμένα για έξοδο πριν παραδώσει τον έλεγχο στο νέο πρόγραμμα, αλλά αυτό δεν εξασφαλίζεται σε κάθε πλατφόρμα. Αν έχετε εκκρεμή έξοδο σε μη εκκενωμένο handle, θέστε την `$|` στο επιλεγμένο handle - ή καλέστε τη μέθοδο `autoflush` σε κάθε `IO::Handle` - πριν την `exec`.
- **Μπλοκ `END` και `DESTROY`.** Η `exec` **δεν** εκτελεί μπλοκ `END`, και **δεν** καλεί `DESTROY` στα αντικείμενά σας. Η εικόνα διεργασίας αντικαθίσταται· καθαρισμός σε επίπεδο Perl δεν συμβαίνει ποτέ. Οτιδήποτε χρειάζεται καθαρό τερματισμό (προσωρινά αρχεία, αφαίρεση lockfile, εκκένωση καταγραφών) πρέπει να συμβεί πριν την `exec`.
- **Close-on-exec.** Τα filehandles με τη σημαία `FD_CLOEXEC` ορισμένη κλείνουν από τον πυρήνα κατά τη διάρκεια της `execvp`· τα υπόλοιπα μένουν ανοιχτά. Ελέγξτε το με `fcntl` ή με το κατώφλι filehandle συστήματος `$^F`.
- **Η αποτυχία μέσω κελύφους είναι αόρατη.** Όταν η `exec` δρομολογείται μέσω `/bin/sh -c`, ένα στοχοπρόγραμμα που λείπει αναφέρεται από το κέλυφος και η ίδια η `exec` έχει ήδη αντικατασταθεί από το `/bin/sh`. Ο κώδικας Perl σας δεν θα δει επιστροφή αποτυχίας.
- **Λίστα ενός ορίσματος χωρίς μετα-χαρακτήρες** *εξακολουθεί* να διαχωρίζεται σε λευκούς χαρακτήρες και να περνά στην `execvp` - οπότε η `exec "ls /tmp"` είναι άμεσο exec του `/bin/ls` με ένα όρισμα, όχι κλήση κελύφους.

- **Κενή `LIST`** είναι σφάλμα κατά την εκτέλεση· δεν υπάρχει πρόγραμμα προς εκτέλεση.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `system` - εκτέλεση προγράμματος και αναμονή του· σχεδόν σίγουρα αυτό που θέλατε αν η `exec` βρίσκεται στην κορυφή του αρχείου
- `fork` - συνδυασμός με την `exec` στο παιδί για εκτόξευση προγράμματος χωρίς αντικατάσταση του γονέα
- `wait` - συλλογή ενός παιδιού που παράγεται από `fork` + `exec`
- `die` - τερματισμός του τρέχοντος προγράμματος με σφάλμα· ο ιδιωματικός σύντροφος του `exec ... or die`
- `$!` - λόγος σφάλματος που τίθεται όταν αποτυγχάνει το άμεσο exec
- `$|` - σημαία αυτόματης εκκένωσης· θέστε πριν την `exec` ώστε καμία ενταμιευμένη έξοδος να μη χαθεί όταν αντικαθίσταται η εικόνα διεργασίας

Hashes

exists

Έλεγχος εάν ένα στοιχείο πίνακα κατακερματισμού ή πίνακα, ή μια ονομασμένη υπορουτίνα, είναι παρόν - χωρίς να το δημιουργεί και χωρίς να ενδιαφέρεται για το τι κρατά.

Η `exists` θέτει ένα δομικό ερώτημα: *υπάρχει εγγραφή σε αυτή τη θέση;* Δεν ρωτά ποια είναι η τιμή της εγγραφής. Μια εγγραφή της οποίας η τιμή είναι `undef` εξακολουθεί να υπάρχει. Μια εγγραφή που δεν έχει εκχωρηθεί ποτέ δεν υπάρχει - η αναζήτησή της με απλή ανάγνωση θα επέστρεφε `undef`, αλλά η `exists` διακρίνει τις δύο περιπτώσεις.

Σύνοψη

```
exists $hash{KEY}
exists $array[INDEX]
exists &subroutine
exists $ref->{A}{B}          # autovivifies $ref->{A}!
```

Τι επιστρέφεται

Μια λογική τιμή: 1 (αληθές βαθμωτό) αν το στοιχείο ή η υπορουτίνα είναι παρόν, η κενή συμβολοσειρά `""` (ψευδές βαθμωτό) αν όχι. Το αποτέλεσμα είναι πάντα ένα απλό βαθμωτό - δεν υπάρχει παραλλαγή σε περιβάλλον λίστας.

```
my %h = (a => undef);
exists $h{a};                 # 1   (key exists)
defined $h{a};                # ""  (value is undef)
exists $h{b};                 # ""  (key does not exist)
```

Η exists δεν είναι η defined

Οι δύο απαντούν σε διαφορετικά ερωτήματα και διαφωνούν ακριβώς όταν μια υπάρχουσα εγγραφή κρατά undef:

Κατάσταση	exists	defined	Απλή ανάγνωση
Δεν εκχωρήθηκε ποτέ	ψευδές	ψευδές	undef
Εκχωρημένο σε undef	**αληθές**	ψευδές	undef
Εκχωρημένο σε οποιαδήποτε άλλη τιμή	αληθές	αληθές	αυτή η τιμή

Χρησιμοποιήστε την exists όταν χρειάζεται να γνωρίζετε αν ένα κλειδί έχει τοποθετηθεί στη δομή - π.χ. έναν πίνακα κατακερματισμού επιλογών όπου ένα σκέτο κλειδί σημαίνει «ο καλών δήλωσε συμμετοχή, η τιμή δεν έχει σημασία». Χρησιμοποιήστε την defined όταν χρειάζεται να γνωρίζετε αν μια τιμή είναι χρησιμοποιήσιμη.

Παραδείγματα

Έλεγχος κλειδιού πίνακα κατακερματισμού. Κλασικό ιδίωμα πίνακα κατακερματισμού επιλογών - παρουσία, όχι αληθοτιμή:

```
sub configure {
    my %opt = @_;
    my $verbose = exists $opt{verbose};     # caller passed verbose
 ↪=> anything
    my $color   = $opt{color} // "auto";    # caller-supplied or
 ↪default
    ...
}
```

Διάκριση μεταξύ «λείπει» και «παρόν αλλά undef»:

```
my %cache = (user_42 => undef);             # cached: "we looked,
 ↪nothing there"
if (exists $cache{user_42}) {
    # hit - even though the value is undef, we already did the
 ↪lookup
} else {
    # miss - have to query the database
}
```

Έλεγχος αραιού πίνακα. Η διαγραφή ενός στοιχείου πίνακα αφήνει μια τρύπα που η exists αναφέρει ως ψευδή:

```
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)
```

Ύπαρξη υπορουτίνας. Αληθές μόλις η υπορουτίνα δηλωθεί, ακόμη και αν το σώμα είναι μια προδήλωση χωρίς ορισμό ακόμα:

```
sub later;                    # forward declaration
exists &later;                # 1
defined &later;               # ""   (no body yet)

sub later { 42 }
defined &later;               # 1
```

Εμφωλευμένη δομή - ο **ασφαλής** τρόπος. Ελέγξτε κάθε επίπεδο πριν την κάθοδο, επειδή η μορφή με αλυσίδα βελών έχει παρενέργεια (βλέπε Οριακές περιπτώσεις):

```
if (exists $tree{users}
    && exists $tree{users}{$id}
    && exists $tree{users}{$id}{email}) {
    send_mail($tree{users}{$id}{email});
}
```

Οριακές περιπτώσεις

- **Αυτο-δημιουργία ενδιάμεσων επιπέδων.** Αυτή είναι η πιο σημαντική παγίδα. Η `exists $ref->{A}{B}{C}` ελέγχει το πιο εσωτερικό κλειδί, αλλά η αποτίμηση της έκφρασης αυτο-δημιουργεί κάθε **ενδιάμεσο** πίνακα κατακερματισμού - οι `$ref->{A}` και `$ref->{A}{B}` εμφανίζονται ως κενοί πίνακες κατακερματισμού ακόμη και όταν ο έλεγχος `exists` επιστρέφει ψευδές:

  ```
  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
  ```

  Το πιο εσωτερικό στοιχείο είναι το μόνο που δεν αυτο-δημιουργείται. Το ίδιο συμβαίνει μέσω αναφορών: η `exists $ref->{A}` σε ένα undef `$ref` αυτο-δημιουργεί την `$ref` σε αναφορά πίνακα κατακερματισμού. Αν χρειάζεται να εξετάσετε μια βαθιά δομή χωρίς να την μεταλλάσσετε, διασχίστε την ένα επίπεδο τη φορά με `exists` σε κάθε βήμα, ή χρησιμοποιήστε ειδικό βοηθητικό όπως τη σημασιολογία `[Data::Diver]` (μια αλυσίδα `eval` / εμβέλεια pragma no `autovivification`).

- **Η `exists` σε κλειδί με τιμή `undef` είναι αληθής.** Αυτή είναι η καθοριστική διαφορά από την `defined`. Η `delete` αφαιρεί τόσο το κλειδί όσο και την τιμή· η εκχώρηση `undef` διατηρεί το κλειδί.

- **Όρια πίνακα.** Η `exists $a[$i]` είναι ψευδής όταν το `$i` είναι πέραν του τέλους του πίνακα **ή** όταν η θέση έχει διαγραφεί με `delete` στη μέση του πίνακα. Οι αρνητικοί δείκτες μετρούν από το τέλος: η `exists $a[-1]` ελέγχει το τελευταίο στοιχείο αν ο πίνακας είναι μη κενός.
- **Η `exists` σε πίνακα αποθαρρύνεται upstream.** Η έννοια της διαγραφής ενός στοιχείου πίνακα δεν είναι εννοιολογικά καθαρή - η `delete $a[1]` αφήνει τρύπα αντί να μετατοπίζει. Ο περισσότερος κώδικας που καταφεύγει στην `exists $a[$i]` εξυπηρετείται καλύτερα από έναν έλεγχο ορίων (`$i <= $#a`) ή με τη χρήση ενός πίνακα κατακερματισμού.
- **Η `exists &sub()` είναι συντακτικό σφάλμα.** Το όρισμα πρέπει να ονοματίζει μια υπορουτίνα, όχι να καλεί μία. Η `exists &sub` ρωτά «είναι δηλωμένη η `sub`;»· η `exists &sub()` θα σήμαινε «υπάρχει η τιμή επιστροφής της κλήσης `sub`;», κάτι που είναι άνευ νοήματος.
- **Η `AUTOLOAD` δεν μετρά.** Η `exists &sub` είναι ψευδής για μια υπορουτίνα που θα δημιουργούνταν μέσω `AUTOLOAD` στην πρώτη κλήση. Μια αποτυχημένη `exists` συνεπώς δεν εγγυάται ότι μια κλήση θα αποτύχει.
- **Δεσμευμένες μεταβλητές.** Για δεσμευμένους πίνακες κατακερματισμού και πίνακες, η `exists` καλεί τη μέθοδο `EXISTS` στο δεσμευμένο αντικείμενο. Η υλοποίηση tie αποφασίζει τι σημαίνει «ύπαρξη» - μια tie `DBM` ελέγχει την εγγραφή στον δίσκο, μια οκνηρή tie μπορεί να υλοποιήσει την εγγραφή. Δείτε το `perltie`.
- **Το όρισμα πρέπει να είναι έκφραση lvalue της οποίας η τελική λειτουργία είναι αναζήτηση κλειδιού ή δείκτη, ή όνομα υπορουτίνας.** Τα `exists func()`, `exists $scalar`, `exists @array`, `exists %hash` είναι όλα σφάλματα. Γίνονται δεκτά μόνο η πρόσβαση σε στοιχείο και το `&sub`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `defined` - ρωτά αν η τιμή είναι χρησιμοποιήσιμη, όχι αν το κλειδί είναι παρόν· διαφέρει από την `exists` μόνο όταν η αποθηκευμένη τιμή είναι `undef`
- `delete` - αφαιρεί μια εγγραφή ώστε η `exists` σε αυτή να γίνει ψευδής· το αντίστοιχο της `exists`
- `each` - επαναλαμβάνει ζεύγη κλειδιού/τιμής ενός πίνακα κατακερματισμού· επισκέπτεται μόνο κλειδιά για τα οποία η `exists` είναι αληθής
- `keys` - επιστρέφει τη λίστα των υπαρχόντων κλειδιών· η `exists $h{$k}` ισοδυναμεί με το ερώτημα αν το `$k` είναι μεταξύ των `keys %h`, αλλά O(1) αντί για O(n)
- `ref` - κατηγοριοποιεί σε τι δείχνει μια αναφορά· χρήσιμη πριν την `exists` σε `$ref->{...}` αν δεν είστε σίγουροι ότι η `$ref` είναι αναφορά πίνακα κατακερματισμού
- `scalar` - εξαναγκασμός βαθμωτού περιβάλλοντος· σημειώστε ότι η `exists` είναι ήδη μόνο βαθμωτή και δεν τη χρειάζεται

Ροή ελέγχου

exit

Τερματισμός του προγράμματος με τιμή κατάστασης.

Η `exit` αποτιμά την `EXPR`, εκτελεί τον καθαρισμό τέλους προγράμματος (μπλοκ `END` και καταστροφείς αντικειμένων), και έπειτα εξέρχεται από τη διεργασία με αυτή την τιμή ως κωδικό κατάστασης. Χωρίς όρισμα εξέρχεται με `0` - επιτυχία.

Σύνοψη

```
exit EXPR
exit
```

Τι επιστρέφεται

Τίποτα. Η `exit` δεν επιστρέφει στον καλούντα της. Είναι δήλωση με την έννοια της ροής ελέγχου παρότι η γραμματική επιτρέπει να γραφεί μέσα σε έκφραση. Κώδικας μετά από `exit` στο ίδιο μπλοκ είναι απρόσιτος:

```
exit 1;
print "never\n";           # dead code
```

Τιμές κατάστασης και η αποκοπή 8 bit

Το λειτουργικό σύστημα αποθηκεύει την κατάσταση εξόδου σε **8 bit**. Όποιον ακέραιο περάσετε στην `exit` λαμβάνεται modulo 256 από τον πυρήνα πριν τον δει η γονική διεργασία. Η `exit 256` φαίνεται σαν `exit 0` στο κέλυφος. Η `exit -1` γίνεται 255. Περνάτε μόνο τιμές στο διάστημα `0..255` και αποφεύγετε την έκπληξη.

Οι μόνες παγκοσμίως φορητές τιμές είναι το `0` για επιτυχία και το 1 για γενικό σφάλμα. Άλλοι αριθμοί έχουν συμβάσεις - το εύρος `sysexits.h` `64..78` σε Unix, συγκεκριμένοι κωδικοί που αναμένονται από τα φίλτρα `sendmail`, και ούτω καθεξής - αλλά αυτά είναι συμφωνίες μεταξύ του προγράμματός σας και συγκεκριμένου καλούντος, όχι εγγυήσεις της γλώσσας. Η έξοδος με 69 (`EX_UNAVAILABLE`) λέει σε ένα φίλτρο εισερχομένων αλληλογραφίας `sendmail` να αναβάλει το μήνυμα· το ίδιο 69 από ένα σενάριο που τρέχει το `cron` δεν σημαίνει κάτι συγκεκριμένο.

```
exit 0;                     # success
exit 1;                     # generic failure
exit 2;                     # conventional "misuse" (grep, diff)
exit 77;                    # conventional "skip" (Automake test)
```

Καθαρισμός: μπλοκ END και καταστροφείς

Η `exit` **δεν** είναι άμεση κλήση συστήματος. Πριν φύγει η διεργασία, η Perl:

1. Εκτελεί κάθε μπλοκ `END` με **αντίστροφη σειρά μεταγλώττισης** (LIFO).

2. Καλεί την DESTROY σε κάθε αντικείμενο του οποίου το refcount πέφτει στο μηδέν καθώς η καθολική καταστροφή διαλύει τα pads, τον πίνακα συμβόλων και τις καθολικές πακέτου.

Και τα δύο είδη καθαρισμού μπορούν να παρατηρήσουν **και να αλλάξουν** την `$?`. Η τιμή που περάσατε στην `exit` τοποθετείται στην `$?` κατά την είσοδο στη φάση END/DESTROY· όποια τιμή κρατά η `$?` όταν επιστρέψει ο τελευταίος καθαρισμός είναι η κατάσταση με την οποία εξέρχεται η διεργασία.

```
END { $? = 0 if $? == 2; }  # rewrite "misuse" to success

exit 2;                     # process actually exits 0
```

Αυτό περιστασιακά είναι χρήσιμο και τακτικά εκπλήσσει. Αν ένα άρθρωμα που δεν γράψατε εσείς εγκαθιστά μπλοκ END που αγγίζει την `$?`, ο προσεκτικά επιλεγμένος κωδικός κατάστασής σας μπορεί να φτάσει στο γονικό κέλυφος αλλοιωμένος.

Όταν χρειάζεστε σκληρή, χωρίς όρους έξοδο που παρακάμπτει εντελώς τα END και DESTROY, καλέστε την `POSIX::_exit`:

```
use POSIX ();
POSIX::_exit(3);            # no END, no DESTROY, status = 3
```

Καταφύγετε στην `_exit` σε θυγατρικές διεργασίες μετά από `fork` όπου ο καθαρισμός ανήκει στον γονέα, και σε χειριστές σήματος όπου η εκτέλεση αυθαίρετου κώδικα Perl δεν είναι ασφαλής.

Η exit δεν είναι η die

Η `exit` είναι το λάθος εργαλείο για την αναφορά σφάλματος από μια βιβλιοθήκη ή υπορουτίνα. Ένας καλών ψηλότερα στη στοίβα μπορεί να θέλει να πιάσει την αποτυχία, να την καταγράψει και να συνεχίσει - ένα μπλοκ `eval` ή ένα περιτύλιγμα `Try::Tiny` δεν μπορεί να ανακάμψει από `exit`. Η διεργασία έχει χαθεί.

```
sub load_config {
    my ($path) = @_;
    open my $fh, "<", $path
        or die "open $path: $!";   # correct - trappable
    ...
}

# NOT:
# open my $fh, "<", $path or exit 1;
```

Χρησιμοποιήστε την `die` για συνθήκες σφάλματος. Κρατήστε την `exit` για το ανώτατο επίπεδο του προγράμματος όπου έχετε αποφασίσει, ως εφαρμογή, ότι η εκτέλεση τελείωσε.

Η exit μέσα σε eval

Η `exit` **δεν** πιάνεται από την `eval`. Σε αντίθεση με την `die`, η οποία ξετυλίγει στην πλησιέστερη `eval` και θέτει την `$@`, η `exit` περνά κατευθείαν πέρα από κάθε πλαίσιο `eval` στη στοίβα, εκτελεί τα END/DESTROY, και τερματίζει.

```
eval { exit 5 };            # process exits 5; the eval block
print "after eval\n";       # never runs
```

Αν γράφετε φιλοξενία πρόσθετων ή λουρί δοκιμών που πρέπει να επιβιώνει από κακή συμπεριφορά κώδικα χρήστη που καλεί `exit`, δεν υπάρχει σύλληψη σε επίπεδο γλώσσας. Η συμβατική λύση είναι να κάνετε fork, να αφήσετε το παιδί να καλέσει `exit`, και να ελέγξετε την κατάσταση του παιδιού στον γονέα:

```
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    run_user_code();        # may call exit
    exit 0;
}
waitpid $pid, 0;
my $status = $? >> 8;
```

Παραδείγματα

Πρόωρη επιστροφή από το πρόγραμμα μετά από προτροπή:

```
my $ans = <STDIN>;
exit 0 if $ans =~ /^[Xx]/;
```

Έξοδος με κατάσταση που προέρχεται από θυγατρική διεργασία - εξάγετε το υψηλό byte της `$?` ώστε το κέλυφος να δει τον ίδιο κωδικό που παρήγαγε το παιδί:

```
system @cmd;
exit $? >> 8 if $?;
```

Επανεγγραφή κατάστασης σε μπλοκ END - όποια τιμή κρατά η `$?` στο τέλος του καθαρισμού είναι η κατάσταση εξόδου που παραδίδεται στον γονέα:

```
END {
    $? = 0 if $? == 2 && $ENV{IGNORE_MISUSE};
}
```

Παράκαμψη του καθαρισμού σε ένα forked παιδί ώστε οι καταστροφείς του γονέα να μην εκτελεστούν δύο φορές:

```
use POSIX ();
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    exec @cmd;
    POSIX::_exit(127);      # exec failed; do NOT run parent's END
}
```

Οριακές περιπτώσεις

- **Η `exit` χωρίς όρισμα** είναι `exit 0`. Πολλά σενάρια βασίζονται στην πτώση από το τέλος του αρχείου για να πάρουν έμμεσο `0`· γράφοντας `exit;` κάνει την πρόθεση ρητή.
- **Μη ακέραιο όρισμα**: η EXPR αποτιμάται σε βαθμωτό περιβάλλον και μετατρέπεται σε ακέραιο. Η `exit "3x"` εξέρχεται με 3· η `exit "hello"` εξέρχεται με `0` και προειδοποίηση υπό `use warnings`.
- **Οι αρνητικές τιμές** λαμβάνονται modulo 256 από τον πυρήνα. Η `exit -1` εμφανίζεται στο κέλυφος ως 255. Περάστε μόνο `0..255` για να αποφύγετε τον εξαναγκασμό.
- **Οι τιμές πάνω από 255** χάνουν τα υψηλά τους bits. Η `exit 256` φαίνεται σαν επιτυχία· η `exit 300` φαίνεται σαν `exit 44`. Αν χρειάζεται να μεταφέρετε μεγαλύτερη τιμή, γράψτε την σε αρχείο ή στο STDERR και εξέλθετε με μικρό κωδικό κατάστασης.
- **Τα μπλοκ `END` μπορούν να ακυρώσουν την κατάστασή σας αλλά όχι την ίδια την έξοδο**. Ένα μπλοκ END δεν μπορεί να εμποδίσει τη διεργασία να τερματιστεί - μπορεί μόνο να αλλάξει την `$?`, και συνεπώς την κατάσταση που βλέπει ο γονέας.
- **Η σειρά καθολικής καταστροφής δεν είναι η αντίστροφη της κατασκευής**. Κατά την τελική σάρωση η Perl διασχίζει τις καθολικές πακέτου με σειρά καθορισμένη από την υλοποίηση. Μη βασίζεστε στο ότι ένα DESTROY θα εκτελεστεί πριν από άλλο· αν η σειρά έχει σημασία, διαλύστε ρητά τα αντικείμενα πριν καλέσετε `exit`.
- **Μέσα σε μπλοκ `BEGIN`**, η `exit` τερματίζει τη φάση μεταγλώττισης και εξέρχεται από τη διεργασία. Τα μπλοκ END που έχουν ήδη καταχωρηθεί εκτελούνται· μεταγενέστερος κώδικας BEGIN/END στο αρχείο όχι.
- **Threads**: σε πρόγραμμα με πολλαπλά threads διερμηνέα Perl, η `exit` τερματίζει ολόκληρη τη διεργασία, όχι μόνο το thread που καλεί. Χρησιμοποιήστε `threads->exit` για έξοδο ανά thread.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `die` - εκτόξευση παγιδεύσιμης εξαίρεσης· η σωστή επιλογή για αναφορά σφαλμάτων από υπορουτίνα
- `warn` - εκπομπή διαγνωστικού χωρίς τερματισμό· συνδυάζεται με `die` όπως η `print` συνδυάζεται με την `exit`
- Μπλοκ END - κώδικας καθαρισμού που καταχωρείται κατά τη μεταγλώττιση και εκτελείται σε σειρά LIFO σε κάθε έξοδο, συμπεριλαμβανομένης της `exit EXPR`
- Μέθοδοι DESTROY - καθαρισμός ανά αντικείμενο που καλείται κατά την καθολική καταστροφή αφού έχουν εκτελεστεί τα μπλοκ END
- `$?` - κατάσταση θυγατρικής διεργασίας / τελική κατάσταση εξόδου· ορατή και μεταβλητή από μπλοκ END και καταστροφείς

- `POSIX::_exit` - άμεσος τερματισμός που παρακάμπτει τα END και DESTROY· χρησιμοποιήστε σε forked παιδιά και χειριστές σήματος

Αριθμητικές συναρτήσεις

exp

Υψώνει το *e* σε δύναμη.

Η exp επιστρέφει το *e* (τη βάση των φυσικών λογαρίθμων, περίπου 2.718281828459045) υψωμένο στη δύναμη της EXPR. Είναι η αντίστροφη της `log`: για κάθε πεπερασμένο θετικό $x, η `exp(log($x))` αναπαράγει το $x εντός της στρογγυλοποίησης κινητής υποδιαστολής. Χωρίς όρισμα, η exp ενεργεί επί της `$_`.

Σύνοψη

```
exp EXPR
exp
```

Τι επιστρέφεται

Αριθμός κινητής υποδιαστολής διπλής ακρίβειας. Το αποτέλεσμα είναι πάντα μη αρνητικό: `exp(0) == 1`, η exp οποιασδήποτε θετικής τιμής είναι μεγαλύτερη του 1, και η exp οποιασδήποτε αρνητικής τιμής βρίσκεται στο ανοιχτό διάστημα `(0, 1)`.

Σε κάθε στόχο Linux που υποστηρίζει το pperl, ο υποκείμενος τύπος είναι IEEE 754 binary64, οπότε το εύρος των πεπερασμένων αποτελεσμάτων είναι περίπου από `5e-324` έως `1.7976931348623157e+308`. Ορίσματα πάνω από περίπου `709.78` υπερχειλίζουν σε `Inf`· ορίσματα κάτω από περίπου `-745` υποχειλίζουν σε `0.0`.

Παραδείγματα

Λάβετε το ίδιο το *e* και επαληθεύστε τον γύρο μετάβασης-επιστροφής έναντι της `log`:

```
my $e = exp(1);                     # 2.718281828459045
printf "%.15f\n", exp(log(42));     # 42.000000000000000
```

Η μορφή χωρίς όρισμα διαβάζει την `$_`:

```
for (1, 2, 3) {
    printf "exp(%d) = %.6f\n", $_, exp;
}
# exp(1) = 2.718282
# exp(2) = 7.389056
# exp(3) = 20.085537
```

Συνθέστε με άλλες αριθμητικές ενσωματωμένες. Το υπερβολικό ημίτονο, για παράδειγμα, δεν έχει αποκλειστικό τελεστή αλλά προκύπτει από την exp:

```
sub sinh { (exp($_[0]) - exp(-$_[0])) / 2 }
printf "%.6f\n", sinh(1);           # 1.175201
```

Υπερχείλιση στην κορυφή του εύρους IEEE 754:

```
print exp(709), "\n";                   # 8.21840746949e+307
print exp(710), "\n";                   # Inf
```

Υποχείλιση στον πυθμένα:

```
print exp(-745), "\n";                  # 5e-324  (smallest subnormal)
print exp(-746), "\n";                  # 0
```

Οριακές περιπτώσεις

- **Πολύ μεγάλο θετικό όρισμα**: Οποιοδήποτε όρισμα μεγαλύτερο από περίπου `709.78` παράγει `Inf`. Το `Inf` διαδίδεται μέσω επόμενων αριθμητικών πράξεων (`Inf + 1 == Inf`, `Inf - Inf` είναι `NaN`) και μετατρέπεται σε συμβολοσειρά ως `"Inf"`.
- **Πολύ αρνητικό όρισμα**: Ορίσματα κάτω από περίπου `-745` υποχειλίζουν σε `0.0`. Οι τιμές ενδιάμεσα αποδίδονται ως μη κανονικοί αριθμοί κινητής υποδιαστολής με προοδευτικά μειούμενη ακρίβεια.
- **Είσοδος `NaN`**: η `exp(NaN)` είναι `NaN`. Το `NaN` συγκρίνεται ως άνισο με τα πάντα συμπεριλαμβανομένου του εαυτού του - ελέγξτε με `Scalar::Util::looks_like_number` ή `$x != $x` αντί για `==`.
- **Η `exp(0)`** είναι ακριβώς 1, όχι στρογγυλοποιημένη προσέγγιση.
- **Μη αριθμητικό όρισμα**: Η Perl εξαναγκάζει το όρισμα μέσω των τυπικών κανόνων μετατροπής συμβολοσειράς σε αριθμό πριν καλέσει τη ρουτίνα C `exp(3)`. Τα `exp("")` και `exp("abc")` είναι επομένως `exp(0) == 1`, με προειδοποίηση `Argument "…" isn't numeric` υπό `use warnings`. Μια συμβολοσειρά με αριθμητικό πρόθεμα όπως η `"3.5xyz"` γίνεται `3.5`, και πάλι με προειδοποίηση.
- **Όρισμα `undef`**: Εξαναγκάζεται σε `0`, αποδίδοντας 1, με προειδοποίηση `uninitialized` υπό `use warnings`.
- **Περιβάλλον λίστας**: Η `exp` είναι μοναδιαία αριθμητική συνάρτηση. Αγνοεί το περιβάλλον λίστας και επιστρέφει πάντα ένα μοναδικό βαθμωτό. Η `my @r = exp(2)` βάζει ένα στοιχείο μέσα στον `@r`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `log` - φυσικός λογάριθμος, η αντίστροφη της `exp`· `exp(log($x)) == $x` για θετικό `$x`
- `sqrt` - τετραγωνική ρίζα· `sqrt($x) == exp(log($x)/2)` για θετικό `$x`
- `sin` - ημίτονο, η άλλη μοναδιαία υπερβατική ενσωματωμένη συνάρτηση
- `cos` - συνημίτονο· μαζί με τις `sin` και `exp` καλύπτει τις τυπικές υπερβατικές συναρτήσεις που είναι διαθέσιμες χωρίς άρθρωμα

- `abs` - απόλυτη τιμή, χρήσιμη όταν τροφοδοτείτε μια προσημασμένη ποσότητα στη `log` μετά από χειρισμό βασισμένο σε `exp`
- `$_` - προεπιλεγμένο όρισμα όταν η `exp` καλείται χωρίς ρητό τελεστέο

Βαθμωτά και συμβολοσειρές

fc

Επιστρέφει τη casefold μορφή μιας συμβολοσειράς κατά Unicode για σύγκριση χωρίς διάκριση πεζών-κεφαλαίων.

Η `fc` είναι η λειτουργία συμβολοσειράς που τροφοδοτεί την ισότητα συμβολοσειρών χωρίς διάκριση πεζών-κεφαλαίων. Παράγει μια μορφή της συμβολοσειράς όπου οι διακρίσεις πεζών-κεφαλαίων έχουν διαγραφεί, οπότε δύο συμβολοσειρές θεωρούνται ίσες χωρίς διάκριση πεζών-κεφαλαίων όταν - και μόνο όταν - τα αποτελέσματα της `fc` είναι byte προς byte πανομοιότυπα. Είναι επίσης η συνάρτηση πίσω από την ακολουθία διαφυγής `\F` σε συμβολοσειρές με διπλά εισαγωγικά.

Σύνοψη

```
use feature 'fc';           # or: use v5.16;

fc EXPR
fc                          # operates on $_
"\F...\E"                   # same casefold, inside a double-quoted
 ↪string
```

Η `fc` βρίσκεται πίσω από την πύλη του χαρακτηριστικού `fc`. Ενεργοποιήστε την ρητά με `use feature 'fc'`, ενσωματώστε την μέσω bundle έκδοσης (`use v5.16` ή νεότερο, `use feature ':5.16'`), ή καλέστε την πλήρως κατονομασμένη ως `CORE::fc` χωρίς καμία pragma.

Τι επιστρέφεται

Μια συμβολοσειρά που περιέχει τη casefold μορφή της `EXPR`. Το αποτέλεσμα είναι μια καινούργια συμβολοσειρά· το όρισμα δεν τροποποιείται ποτέ. Το μήκος μπορεί να αλλάξει - η casefold του `"\x{1E9E}"` (LATIN CAPITAL LETTER SHARP S) κανονικά επεκτείνεται σε `"ss"`, δύο χαρακτήρες από έναν.

Αντιμετωπίστε την τιμή επιστροφής ως αδιαφανή: είναι κλειδί κατάλληλο για σύγκριση ισότητας, όχι για εμφάνιση. Η `fc("Hello")` είναι `"hello"` για ASCII, αλλά στη γενική περίπτωση η έξοδος δεν είναι κάτι που θα δείχνατε σε χρήστη.

Γιατί όχι `lc` ή `uc`;

Η μετατροπή σε πεζά και η μετατροπή σε κεφαλαία **δεν** είναι αξιόπιστες για σύγκριση χωρίς διάκριση πεζών-κεφαλαίων. Και οι δύο παρακάτω είναι λάθος:

```
lc($a) eq lc($b)            # Wrong
uc($a) eq uc($b)            # Also wrong
```

Αποτυγχάνουν σε χαρακτήρες των οποίων η αντιστοίχιση πεζών/κεφαλαίων δεν είναι συμμετρική - με πιο γνωστό παράδειγμα το γερμανικό sharp S. Η `lc("\x{1E9E}")` είναι `"\x{1E9E}"` (δεν υπάρχει μορφή πεζών), αλλά η `uc("ß")` είναι `"SS"`. Η casefold παρακάμπτει αυτό αντιστοιχίζοντας και τις δύο πλευρές σε μια αφιερωμένη μορφή ισότητας:

```
fc($a) eq fc($b)            # Right
```

Το βασισμένο σε regex ισοδύναμο που ήταν σωστό πριν υπάρξει η `fc`:

```
$a =~ /^\Q$b\E\z/i
```

Η `fc` είναι ο άμεσος, χωρίς-regex τρόπος να πάρετε την ίδια απάντηση.

Καθολική κατάσταση που επηρεάζει

- `$_` - χρησιμοποιείται ως όρισμα όταν η EXPR παραλείπεται.
- `use locale` - μέσα σε εμβέλεια `use locale`, η casefold χαρακτήρων που διασχίζουν το όριο 255/256 απενεργοποιείται (δείτε τις *Οριακές περιπτώσεις* παρακάτω για τον κανόνα του U+1E9E).
- `use feature 'unicode_strings'` - επηρεάζει την `fc` με τον ίδιο τρόπο όπως επηρεάζει την `lc`: επιβάλλει πλήρη σημασιολογία Unicode σε συμβολοσειρές bytes που διαφορετικά θα αντιμετωπίζονταν υπό τους παλιούς 8-bit κανόνες.

Παραδείγματα

Ισότητα χωρίς διάκριση πεζών-κεφαλαίων, η κανονική χρήση:

```
use feature 'fc';
fc("Hello") eq fc("HELLO");             # true
fc("café")  eq fc("CAFÉ");              # true
```

Το γερμανικό sharp S - η σχολικού βιβλίου περίπτωση όπου και η `lc` και η `uc` αποτυγχάνουν αλλά η `fc` πετυχαίνει:

```
use feature 'fc';
my $a = "straße";
my $b = "STRASSE";
fc($a) eq fc($b);                       # true
lc($a) eq lc($b);                       # false - "straße" ne
 ↪"strasse"
```

Χρήση της `fc` ως κλειδί κατακερματισμού για αναζήτηση χωρίς διάκριση πεζών-κεφαλαίων:

```
use feature 'fc';
my %seen;
for my $word (@words) {
    $seen{ fc $word }++;                # groups "Foo", "FOO", "foo"
}
```

Χωρίς όρισμα - λειτουργεί στην $_:

```
use feature 'fc';
for (@lines) {
    next unless fc eq "quit";           # matches "QUIT", "Quit", ..
 ↪.
    last;
}
```

Μέσα σε συμβολοσειρά με διπλά εισαγωγικά μέσω της \F (ίδια λειτουργία, ενσωματωμένη):

```
use feature 'fc';
my $name = "Alice";
my $key  = "\F$name\E";                 # "alice" - same as fc(
 ↪$name)
```

Κλήση χωρίς την feature pragma, πλήρως κατονομασμένη:

```
my $folded = CORE::fc($input);          # works in any scope
```

Οριακές περιπτώσεις

- **Χωρίς όρισμα**: η `fc` χωρίς όρισμα κάνει fold την $_.
- **Όρισμα `undef`**: μετατρέπεται σε συμβολοσειρά ως κενή συμβολοσειρά, που με τη σειρά της κάνει fold στην κενή συμβολοσειρά. Εκπέμπει προειδοποίηση `uninitialized` υπό `use warnings`.
- **Το χαρακτηριστικό δεν είναι ενεργοποιημένο**: η `fc EXPR` χωρίς `use feature 'fc'` (ή bundle `use v5.16+`) είναι σφάλμα κατά τη μεταγλώττιση - ο αναλυτής δεν αναγνωρίζει την `fc` ως δεσμευμένη λέξη. Η `CORE::fc(EXPR)` λειτουργεί πάντα.
- **Το μήκος μπορεί να αλλάξει**: οι πλήρεις casefolds μπορούν να επεκτείνουν έναν χαρακτήρα σε πολλούς. Η `fc("\x{1E9E}")` είναι `"ss"`. Μη βασιστείτε ποτέ στο `length(fc $s) == length($s)`.
- **Δεν είναι πλήρης κύκλος**: η `fc` είναι μονόδρομη. Δεν υπάρχει λειτουργία «uncasefold»· η αρχική κατάσταση πεζών-κεφαλαίων χάθηκε.
- **Το U+1E9E υπό `use locale`**: η `fc` του *LATIN CAPITAL LETTER SHARP S* (U+1E9E) κανονικά κάνει fold σε `"ss"`. Υπό `use locale` αυτή η αντιστοίχιση καταστέλλεται, επειδή διασχίζει το όριο σημείων κώδικα 255/256, που οι κανόνες locale δεν χειρίζονται καθαρά. Αντί αυτού η `fc` επιστρέφει `"\x{17F}\x{17F}"` (δύο *LATIN SMALL LETTER LONG S*). Επειδή κάθε *long s* κάνει το ίδιο fold σε `"s"`, δύο από αυτά συγκρίνονται ίσα με ένα μεμονωμένο U+1E9E που έκανε fold εκτός της εμβέλειας locale - οπότε η σημασιολογία ισότητας διατηρείται παρότι η μορφή σε bytes διαφέρει.
- **Δεν παρέχονται Turkic και «simple» folds**: η Perl υλοποιεί μόνο την *πλήρη, μη-Turkic* μορφή casefold. Για την απλή μορφή ή τη Turkic παραλλαγή, χρησιμοποιήστε την `Unicode::UCD::casefold` ή το CPAN άρθρωμα `Unicode::Casing`.

- **Δεν είναι το ίδιο με NFKC ή NFC**: η `fc` διαγράφει τις διακρίσεις πεζών-κεφαλαίων, όχι διαφορές συμβατότητας. Η `fc("ffi")` (*LATIN SMALL LIGATURE FFI*) είναι `"ffi"` μετά το fold, όχι `"ffi"`. Συνδυάστε με την `Unicode::Normalize` αν χρειάζεστε και τα δύο.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `lc` - μετατρέπει συμβολοσειρά σε πεζά· χρησιμοποιήστε για εμφάνιση, όχι για σύγκριση χωρίς διάκριση πεζών-κεφαλαίων
- `uc` - μετατρέπει συμβολοσειρά σε κεφαλαία· ίδια προειδοποίηση με την `lc`
- `lcfirst` - μετατρέπει σε πεζά μόνο τον πρώτο χαρακτήρα
- `ucfirst` - μετατρέπει σε κεφαλαία μόνο τον πρώτο χαρακτήρα
- `index` - αναζήτηση υποσυμβολοσειράς· συνδυάστε με `fc` και στα δύο τελούμενα για μια παραλλαγή χωρίς διάκριση πεζών-κεφαλαίων
- `$_` - το προεπιλεγμένο όρισμα όταν η `EXPR` παραλείπεται

Filehandles, αρχεία, κατάλογοι

fcntl

Εκτελεί μια λειτουργία ελέγχου αρχείου `fcntl(2)` σε ένα filehandle.

Η `fcntl` είναι η άμεση σύνδεση της Perl με την κλήση συστήματος POSIX `fcntl(2)`. Διαβάζει ή αλλάζει κατάσταση ανά περιγραφέα - μη μπλοκαριστική λειτουργία, close-on-exec, συμβουλευτικά κλειδώματα εγγραφών, σημαίες περιγραφέα - επικαλούμενη τη `FUNCTION` επί του `FILEHANDLE`, με το `SCALAR` να παίζει ρόλο είτε ως όρισμα εισόδου, είτε ως ενταμιευτής εξόδου, είτε και τα δύο, ανάλογα με τη `FUNCTION` που περνάτε.

Οι κωδικοί `FUNCTION` δεν είναι κυριολεκτικά σε επίπεδο Perl· είναι μακροεντολές του προεπεξεργαστή C που εκτίθενται από το άρθρωμα `Fcntl`. Φορτώστε το πρώτα, αλλιώς η `FUNCTION` θα είναι ένα αόριστο bareword:

```
use Fcntl;
```

Σύνοψη

```
use Fcntl;
fcntl FILEHANDLE, FUNCTION, SCALAR
my $flags = fcntl($fh, F_GETFL, 0);
fcntl($fh, F_SETFL, $flags | O_NONBLOCK);
```

Τι επιστρέφεται

Σε επιτυχία, τον ακέραιο που επέστρεψε ο πυρήνας. Όταν αυτός ο ακέραιος είναι `0`, η Perl τον αντικαθιστά με τη συμβολοσειρά διπλής τιμής `"0 but true"` - αληθής σε λογικό περιβάλλον, αριθμητικό `0` σε αριθμητικό περιβάλλον, και εξαιρείται από την προειδοποίηση `Argument "..." isn't numeric`. Αυτό σας επιτρέπει να ελέγξετε το αποτέλεσμα με `or die` ακόμα και για λειτουργίες των οποίων η τιμή επιτυχίας είναι κυριολεκτικά μηδέν:

```
fcntl($fh, F_SETFD, FD_CLOEXEC)
    or die "F_SETFD: $!";
```

Σε αποτυχία, η `fcntl` επιστρέφει `undef` και θέτει το `$!` στο σχετικό `errno`. **Δεν** χρειάζεται να περιτυλίξετε την επιστροφή σε `defined` - η σύμβαση `"0 but true"` καθιστά επαρκή έναν απλό έλεγχο αλήθειας.

Για κωδικούς `FUNCTION` που γράφουν πίσω μέσω του `SCALAR` (όπως ο `F_GETLK`), τα ενημερωμένα bytes εμφανίζονται στο βαθμωτό που περάσατε· η επιστρεφόμενη τιμή ακολουθεί παρ” όλα αυτά τον κανόνα παραπάνω.

Πώς χρησιμοποιείται το SCALAR

Το `SCALAR` παίζει τρεις διαφορετικούς ρόλους ανάλογα με τη `FUNCTION`:

- **Ακέραιος εισόδου.** Για κλήσεις ρύθμισης σημαιών όπως οι `F_SETFD` ή `F_SETFL`, το `SCALAR` είναι η ακέραια μάσκα bits που θα εγκατασταθεί.
- **Παραβλεπόμενος placeholder.** Για κλήσεις ανάγνωσης σημαιών όπως οι `F_GETFD` ή `F_GETFL`, το `SCALAR` δεν χρησιμοποιείται από τον πυρήνα· περάστε `0` συμβατικά.
- **Ενταμιευτής δομής εισόδου/εξόδου.** Για κλήσεις κλειδώματος εγγραφών όπως οι `F_GETLK`, `F_SETLK` και `F_SETLKW`, το `SCALAR` πρέπει να περιέχει μια πακεταρισμένη `struct flock` - δημιουργήστε την με `pack` πριν την κλήση, και για την `F_GETLK` αποκωδικοποιήστε τα επιστρεφόμενα bytes με `unpack` στη συνέχεια.

Όταν το `SCALAR` χρησιμοποιείται ως ενταμιευτής εξόδου, προδιαστασιολογήστε το τουλάχιστον στο μήκος της δομής που θα γράψει ο πυρήνας· ένα υπομέγεθες βαθμωτό μεγαλώνει αυτόματα, αλλά η προδιαστασιολόγηση κάνει την πρόθεση εμφανή.

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται σε αποτυχία στο `errno` από την `fcntl(2)`.
- Οι σημαίες σε επίπεδο περιγραφέα επί του `FILEHANDLE` (π.χ. `O_NONBLOCK`, `FD_CLOEXEC`) διατηρούνται για όλη τη διάρκεια ζωής του περιγραφέα, επιβιώνουν της dup και της `fork` σύμφωνα με τους κανόνες POSIX, και **δεν** είναι ορατές από την Perl μέσω κάποιας άλλης μεταβλητής. Ο μόνος τρόπος να τις διαβάσετε πίσω είναι μια άλλη κλήση `fcntl` με `F_GETFL` / `F_GETFD`.

Παραδείγματα

Κάντε μια υποδοχή μη μπλοκαριστική - η κανονική χρήση:

```
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: $!";
```

Θέστε close-on-exec ώστε ο περιγραφέας να μην κληρονομείται από διεργασίες-παιδιά μέσω `exec`:

```
use Fcntl qw(F_SETFD FD_CLOEXEC);

fcntl($fh, F_SETFD, FD_CLOEXEC)
    or die "F_SETFD: $!";
```

Διαβάστε τις τρέχουσες σημαίες περιγραφέα και ελέγξτε ένα 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
}
```

Τοποθετήστε ένα συμβουλευτικό κλείδωμα εγγραφής σε ολόκληρο το αρχείο. Η διάταξη της `struct flock` είναι ειδική της πλατφόρμας· σε Linux το `s s l l i` καλύπτει τα `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: $!";
```

Απενεργοποίηση μιας μεμονωμένης σημαίας - μασκάρετε εκτός το `O_NONBLOCK` για να επαναφέρετε ένα handle σε μπλοκαριστική λειτουργία:

```
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: $!";
```

Οριακές περιπτώσεις

- **Ξεχάστηκε το `use Fcntl`**: τα `F_GETFL` και συναφή αναλύονται ως barewords και αποτιμώνται σε ενόχληση επιπέδου προειδοποίησης υπό `use strict` / `use warnings`, ή στη συμβολοσειρά `"F_GETFL"` διαφορετικά - και στις δύο περιπτώσεις η κλήση αποτυγχάνει με `EINVAL`. Φορτώνετε πάντα πρώτα το `Fcntl`.
- **Επιστροφή `"0 but true"`**: μη συγκρίνετε την επιστροφή με `== 0` ως δοκιμή αποτυχίας. Χρησιμοποιήστε απλό λογικό (`or die`) ή `defined` - το πρώτο αποτελεί το ιδίωμα.
- **`F_SETLK` έναντι `F_SETLKW`**: η `F_SETLK` επιστρέφει `undef` και θέτει το `$!` σε `EAGAIN` / `EACCES` όταν άλλη διεργασία κρατά ένα κλείδωμα που συγκρούεται. Η `F_SETLKW` μπλοκάρει μέχρι να εκχωρηθεί το κλείδωμα. Επιλέξτε σκόπιμα· η σύγχυσή τους είναι η συνηθισμένη πηγή κρεμασμάτων.
- **Έξοδος της `F_GETLK`**: σε επιτυχία τα bytes της `struct flock` στο `SCALAR` γράφονται από πάνω με πληροφορίες για το συγκρουόμενο κλείδωμα (ή `l_type == F_UNLCK` αν η περιοχή είναι ελεύθερη). Κάντε `unpack` με το ίδιο πρότυπο με το οποίο πακετάρατε.
- **Δεν υλοποιεί κάθε σύστημα κάθε FUNCTION**: η `fcntl` εγείρει εξαίρεση σε μηχάνημα που δεν υλοποιεί καθόλου την `fcntl(2)`. Σε συστήματα που την υλοποιούν, μη υποστηριζόμενοι κωδικοί `FUNCTION` αποτυγχάνουν με `EINVAL` μέσω του `$!`. Το άρθρωμα `Fcntl` εξάγει μόνο τις σταθερές που η τρέχουσα πλατφόρμα πράγματι ορίζει.
- **`fcntl` έναντι `ioctl`**: μοιράζονται τη σύμβαση επεξεργασίας ορισμάτων και τιμής επιστροφής, αλλά στοχεύουν διαφορετικές διεπαφές πυρήνα. Χρησιμοποιήστε την `fcntl` για σημαίες περιγραφέα και συμβουλευτικά κλειδώματα· χρησιμοποιήστε την `ioctl` για κωδικούς ελέγχου ειδικούς ανά συσκευή.
- **`fcntl` έναντι `flock`**: η `flock` χρησιμοποιεί κλειδώματα ολόκληρου αρχείου τύπου BSD `flock(2)`, τα οποία σε Linux συνεργάζονται με τα POSIX κλειδώματα της `fcntl` μόνο με περιορισμένους τρόπους. Αν χρειάζεστε κλείδωμα εύρους bytes ή σημασιολογία POSIX (κλείδωμα που απελευθερώνεται σε *οποιοδήποτε* `close` του αρχείου από τη διεργασία κατόχου), χρησιμοποιήστε την `fcntl` με `F_SETLK`· διαφορετικά η `flock` είναι απλούστερη.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `Fcntl` - το άρθρωμα που εξάγει τις σταθερές `F_*` / `O_*` / `FD_*` / `SEEK_*` που αναμένει η `fcntl`· κάντε πάντα `use` πρώτα
- `ioctl` - ίδια σύμβαση κλήσης και τιμής επιστροφής με την `fcntl`, αλλά για κωδικούς ελέγχου συσκευών αντί για σημαίες ελέγχου αρχείου
- `flock` - απλούστερα συμβουλευτικά κλειδώματα ολόκληρου αρχείου· επιλέξτε την όταν δεν χρειάζεστε εύρος bytes ή σημασιολογία POSIX
- `open` - ανοίγει εξαρχής το filehandle· ορισμένες σημαίες που ρυθμίζει η `fcntl` (π.χ. `O_NONBLOCK`, `O_APPEND`) μπορούν επίσης να τεθούν κατά το άνοιγμα

- `sysopen` - χαμηλού επιπέδου άνοιγμα που δέχεται τις ίδιες σημαίες `O_*` από το `Fcntl`· χρησιμοποιήστε το όταν θέλετε οι σημαίες να εφαρμοστούν ατομικά αντί μέσω μιας επακόλουθης `fcntl(F_SETFL)`
- `pack` / `unpack` - απαιτούνται για την κατασκευή και αποκωδικοποίηση του ενταμιευτή `struct flock` που χρησιμοποιείται από τις `F_GETLK` / `F_SETLK`

Εμβέλεια · Κλάσεις και αντικειμενοστρέφεια

field

Δηλώνει μια μεταβλητή ανά στιγμιότυπο μέσα σε ένα μπλοκ `class`.

Η `field` εισάγει μια νέα μεταβλητή της οποίας η αποθήκευση είναι ιδιωτική σε κάθε αντικείμενο της περικλείουσας κλάσης. Κάθε μέθοδος και κάθε μπλοκ `ADJUST` αυτής της κλάσης βλέπει το πεδίο σαν να ήταν μια λεξιλογική μεταβλητή σε εμβέλεια σε εκείνο το σημείο - αλλά η τιμή που βλέπει η καθεμιά είναι η τιμή που ανήκει στο *τρέχον* στιγμιότυπο. Τα πεδία αντικαθιστούν, στο πλαίσιο του χαρακτηριστικού class, την παρακολούθηση θέσεων hash σε μια blessed αναφορά.

The feature is still marked **experimental** in Perl 5.44 (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.

Σύνοψη

```
field $scalar;
field $scalar = EXPR;
field @array  : ATTRIBUTES;
field %hash   : ATTRIBUTES = EXPR;
```

Τι επιστρέφεται

Η `field` είναι δήλωση, όχι έκφραση. Δεν παράγει τιμή και δεν μπορεί να εμφανιστεί στη δεξιά πλευρά οτιδήποτε. Κάθε εντολή `field` δεσμεύει μία θέση ανά στιγμιότυπο στη διάταξη της κλάσης· η θέση είναι η ίδια η μεταβλητή του πεδίου, συνδεδεμένη με μια θέση αποθήκευσης ανά αντικείμενο κατά την είσοδο σε οποιαδήποτε μέθοδο ή σε μπλοκ `ADJUST`.

Επιτρέπονται βαθμωτά, πίνακες και hashes:

```
class Thing {
    field $scalar = 42;
    field @array  = qw(this is just an array);
    field %hash   = (species => 'Martian', planet => 'Mars');
}
```

Πού επιτρέπονται οι δηλώσεις πεδίων

Μόνο απευθείας μέσα σε ένα μπλοκ `class`. Η `field` δεν είναι δηλωτικός γενικής χρήσης - δεν επιτρέπεται μέσα σε σώμα `method`, μπλοκ `ADJUST`, εμφωλευμένη `sub` ή σε απλό κώδικα `package`. Ο μεταγλωττιστής την απορρίπτει οπουδήποτε αλλού:

```
class C {
    field $ok;                  # OK
    method m {
        field $nope;            # compile error
    }
}
```

Ένα πεδίο πρέπει να δηλωθεί πριν μπορέσει να αναφερθεί από οποιαδήποτε μέθοδο ή αρχικοποιητή στο ίδιο σώμα κλάσης.

Αρχικοποιητές πεδίων

Αν υπάρχει `= EXPR`, η έκφραση εκτελείται μία φορά ανά κλήση κατασκευαστή, αφού όλα τα πεδία που δηλώθηκαν νωρίτερα στο σώμα της κλάσης έχουν αρχικοποιηθεί. Αυτό έχει σημασία όταν η προεπιλεγμένη τιμή ενός μεταγενέστερου πεδίου εξαρτάται από ένα προηγούμενο:

```
class WithACounter {
    my $next_count = 1;
    field $count = $next_count++;
}
```

Μέσα σε έναν αρχικοποιητή, το `$self` **δεν** υπάρχει - το αντικείμενο βρίσκεται ακόμη υπό κατασκευή. Χρησιμοποιήστε το `__CLASS__` όταν ο αρχικοποιητής χρειάζεται το όνομα της κλάσης (για παράδειγμα, για να καλέσει μια μέθοδο κλάσης που οι υποκλάσεις μπορούν να παρακάμψουν):

```
class WithCustomField {
    use constant DEFAULT_X => 10;
    field $x = __CLASS__->DEFAULT_X;
}

class DifferentCustomField :isa(WithCustomField) {
    sub DEFAULT_X { rand > 0.5 ? 20 : 30 }
}
```

Ένα στιγμιότυπο της `DifferentCustomField` θα δει το `__CLASS__` να επιλύεται στο δικό του όνομα, οπότε ο αρχικοποιητής επιλέγει την παρακαμμένη `DEFAULT_X`.

Γνωρίσματα πεδίων

Τα γνωρίσματα μετά από άνω-κάτω τελεία ελέγχουν πώς το πεδίο συμμετέχει στην κατασκευή και αν παράγονται μέθοδοι προσπέλασης για αυτό.

`:param`

Λαμβάνει την τιμή του πεδίου από ένα ονομαστικό όρισμα προς τον κατασκευαστή:

```
field $x :param;
field $y :param(the_y_value);
```

Από προεπιλογή, το όνομα της παραμέτρου είναι το όνομα του πεδίου χωρίς το sigil ($x → x). Ένα ρητό όνομα σε παρενθέσεις το παρακάμπτει. Χωρίς προεπιλεγμένη έκφραση η παράμετρος είναι **υποχρεωτική**

- omitting it from `->new( . . . )` throws. With a defaulting expression the parameter is optional, and three default operators are available:
- `=` - η προεπιλογή εφαρμόζεται μόνο όταν ο καλών παρέλειψε την παράμετρο.
- `//=` - η προεπιλογή εφαρμόζεται επίσης όταν ο καλών πέρασε `undef`.
- `||=` - η προεπιλογή εφαρμόζεται επίσης όταν ο καλών πέρασε μια ψευδή τιμή.

```
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

Παράγει μια μέθοδο προσπέλασης με μηδέν ορίσματα που επιστρέφει το πεδίο. Χωρίς ρητό όνομα, η μέθοδος ονομάζεται από το πεδίο (χωρίς το sigil):

```
field $s :reader;

# equivalent to:
field $s;
method s () { return $s }
```

Επιτρέπεται ρητό όνομα:

```
field $x :reader(get_x);        # method get_x () { return $x }
```

Οι αναγνώστες (readers) μπορούν να εφαρμοστούν και σε πεδία πινάκων και hash· σε περιβάλλον λίστας η μέθοδος δίνει τα περιεχόμενα, ενώ σε βαθμωτό περιβάλλον τον αριθμό των στοιχείων - η συνήθης συμπεριφορά συμφραζομένων της υποκείμενης μεταβλητής:

```
field @users :reader;
...
scalar $instance->users;        # count of users
```

:writer

Παράγει μια μέθοδο setter ενός ορίσματος που εκχωρεί το όρισμά της στο πεδίο και επιστρέφει τον επικαλούμενο (invocant) (για υποστήριξη αλυσίδωσης). Το προεπιλεγμένο όνομα μεθόδου είναι το όνομα του πεδίου (χωρίς το sigil) με πρόθεμα `set_`:

```
field $s :writer;

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
# equivalent to:
field $s;
method set_s ($new) { $s = $new; return $self }
```

Επιτρέπεται ρητό όνομα:

```
field $x :writer(write_x);      # method write_x ($new) { ... }
```

Το `:writer` λειτουργεί προς το παρόν μόνο σε βαθμωτά πεδία· η εφαρμογή του σε πεδίο πίνακα ή hash αποτελεί μοιραίο σφάλμα κατά τη μεταγλώττιση. Φτιάξτε writers για πίνακες ή hash με το χέρι.

Καθολική κατάσταση που επηρεάζει

Καμία απευθείας. Η αποθήκευση των πεδίων ζει στο στιγμιότυπο, ενώ οι εκφράσεις αρχικοποιητών εκτελούνται στην εμβέλεια του κατασκευαστή - μπορούν να διαβάσουν οποιαδήποτε μεταβλητή ορατή στο σημείο της δήλωσης, αλλά η ίδια η `field` δεν επηρεάζει καμία τεκμηριωμένη ειδική μεταβλητή.

Παραδείγματα

Ελάχιστη κλάση με ένα πεδίο, αρχικοποιημένο στο 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
```

Παράμετροι κατασκευαστή μέσω `:param`, με συνδυασμό υποχρεωτικών και προεπιλεγμένων πεδίων:

```
class Point {
    field $x :param;             # required
    field $y :param = 0;         # optional, defaults to 0

    method as_string { "($x, $y)" }
}
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
Point->new(x => 3)->as_string;          # (3, 0)
Point->new(x => 3, y => 4)->as_string;  # (3, 4)
```

Οι αυτόματα παραγόμενες μέθοδοι προσπέλασης καλύπτουν τον συνηθισμένο πάγιο κώδικα:

```
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
```

Πεδία πινάκων και hash με προεπιλογές:

```
class Bag {
    field @items = ();
    field %counts;

    method add ($item) {
        push @items, $item;
        $counts{$item}++;
        return $self;
    }

    method unique { keys %counts }
}
```

Αρχικοποιητής που εξαρτάται από προηγούμενο πεδίο:

```
class Rect {
    field $w :param;
    field $h :param;
    field $area = $w * $h;       # evaluated after $w, $h are set
}
```

Οριακές περιπτώσεις

- **Η λεξιλογική ορατότητα έχει εμβέλεια μεθόδου, όχι αρχείου.** Ένα πεδίο είναι ορατό από οποιαδήποτε `method` ή μπλοκ `ADJUST` της ίδιας κλάσης, ανεξάρτητα από τη σειρά στο πηγαίο αρχείο. *Δεν* είναι ορατό από απλές δηλώσεις `sub` μέσα στο μπλοκ της κλάσης - μόνο από `method`.
- **Ανά στιγμιότυπο, όχι ανά κλάση.** Δύο στιγμιότυπα της ίδιας κλάσης έχουν ανεξάρτητη αποθήκευση για κάθε πεδίο. Η εκχώρηση σε ένα στιγμιότυπο δεν επηρεάζει κάποιο άλλο. Αν θέλετε κατάσταση σε επίπεδο κλάσης, χρησιμοποιήστε μια μεταβλητή `my` σε εμβέλεια αρχείου εκτός της λίστας πεδίων.

- **Το sigil καθορίζει τον τύπο.** Το `field $x` είναι βαθμωτή θέση, το `field @a` θέση πίνακα, το `field %h` θέση hash. Σε αντίθεση με τα αντικείμενα βάσει hash, δεν υπάρχει τρόπος να αλλάξετε τον τύπο ενός πεδίου σε χρόνο εκτέλεσης· ο τύπος υποδοχέα της θέσης καθορίζεται από το sigil κατά τη δήλωση.
- **Το `$self` είναι μη διαθέσιμο σε αρχικοποιητές.** Το αντικείμενο δεν έχει κατασκευαστεί ακόμη όταν εκτελείται ο αρχικοποιητής. Χρησιμοποιήστε το `__CLASS__` για το όνομα κλάσης, και αναβάλετε στο `ADJUST` ό,τι απαιτεί το πλήρες αντικείμενο.
- **Η σειρά των αρχικοποιητών είναι η σειρά δήλωσης.** Οι μεταγενέστεροι αρχικοποιητές μπορούν να αναφερθούν σε προηγούμενα πεδία· το αντίστροφο είναι αναφορά σε μη αρχικοποιημένη μεταβλητή.
- **`:writer` on non-scalar fields is fatal at compile time.** The attribute is defined only for scalars in 5.44; applying it to `@arr` or `%h` aborts compilation of the class.
- **Το `:param` χωρίς προεπιλογή είναι υποχρεωτικό.** Η `Class->new()` με την παράμετρο να λείπει εγείρει εξαίρεση. Προσθέστε `= EXPR`, `//= EXPR` ή `||= EXPR` για να την κάνετε προαιρετική.
- **Καμία απευθείας εξωτερική πρόσβαση.** Χωρίς `:reader` / `:writer` (ή χειρόγραφη μέθοδο) ένα πεδίο είναι μη προσπελάσιμο από έξω από την κλάση - αυτή είναι η ενθυλάκωση που παρέχει το χαρακτηριστικό class.
- **Πειραματική προειδοποίηση.** Η συντακτική ανάλυση μιας δήλωσης `field` εκπέμπει προειδοποίηση `experimental::class`, εκτός αν αυτή η κατηγορία έχει κατασταλεί· δείτε *Διαφορές από το upstream*.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44. 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.

Δείτε επίσης

- `class` - δηλώνει την κλάση το σώμα της οποίας είναι το μόνο μέρος όπου η `field` επιτρέπεται
- `method` - η μόνη μορφή υπορουτίνας που βλέπει μεταβλητές πεδίων με το όνομά τους
- `my` - συνήθης λεξιλογική δήλωση· χρησιμοποιήστε την για κατάσταση σε επίπεδο κλάσης που *δεν* πρέπει να είναι ανά στιγμιότυπο
- `our` - δήλωση καθολική σε πακέτο· χρησιμοποιήστε την για γνήσια καθολική κατάσταση που μια κλάση θέλει να εκθέσει
- `bless` - ο τρόπος δημιουργίας αντικειμένων πριν την 5.38· η `field` αντικαθιστά την παρακολούθηση θέσεων hash που χρειάζονταν οι κλάσεις βασισμένες στο `bless`
- `ref` - ελέγξτε την κλάση ενός αντικειμένου σε χρόνο εκτέλεσης· εξακολουθεί να λειτουργεί σε στιγμιότυπα που παράγονται από κατασκευαστή `class`

I/O

fileno

Επιστρέφει τον αριθμό περιγραφέα αρχείου σε επίπεδο OS που βρίσκεται πίσω από ένα filehandle.

Η `fileno` βλέπει πέρα από την αφαίρεση του ενταμιευμένου filehandle της Perl και σας δίνει τον ακατέργαστο ακέραιο περιγραφέα που χρησιμοποιεί ο πυρήνας. Είναι η γέφυρα μεταξύ του Perl I/O και των κλήσεων συστήματος που μιλούν απευθείας σε περιγραφείς - `select`, `POSIX::dup2`, `fcntl`, ή οτιδήποτε σε `IO::Poll` / `IO::Select`. Σε handle καταλόγου, υποβάλλει το ίδιο ερώτημα στο `dirfd(3)`, το οποίο δεν παρέχουν όλες οι πλατφόρμες.

Σύνοψη

```
fileno FILEHANDLE
fileno DIRHANDLE
fileno EXPR             # EXPR evaluated as an indirect handle
```

Τι επιστρέφεται

- Έναν ακέραιο `>= 0` - τον περιγραφέα αρχείου του OS - για ένα κανονικό ανοιχτό handle που υποστηρίζεται από πραγματικό αντικείμενο πυρήνα.
- `-1` όταν το handle είναι ανοιχτό αλλά **δεν** έχει πραγματικό περιγραφέα. Αυτό λαμβάνετε για handles στη μνήμη που δημιουργήθηκαν από την `open` με αναφορά σε βαθμωτό ως τρίτο όρισμα (`open my $fh, ">", \$buf`).
- `undef` όταν το handle είναι κλειστό, ποτέ ανοιχτό, ή - σε handle καταλόγου - όταν η πλατφόρμα δεν διαθέτει `dirfd(3)`. Στην περίπτωση του handle καταλόγου τίθεται το `$!`.

Οι τρεις κατηγορίες επιστροφής έχουν σημασία. Κώδικας που ελέγχει το `fileno($fh)` για αλήθεια συγχέει το «κανένας τέτοιος fd» με το «fd αριθμός 0» (STDIN). Συγκρίνετε πάντα ρητά:

```
my $fd = fileno $fh;
die "handle not open"      unless defined $fd;
die "handle has no real fd" if $fd == -1;
```

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται όταν η `fileno` σε handle καταλόγου αποτυγχάνει επειδή η πλατφόρμα δεν έχει αντίστοιχο του `dirfd(3)`.

Η `fileno` δεν διαβάζει ούτε γράφει σε καμία άλλη καθολική μεταβλητή I/O. Δεν εκκενώνει, δεν κάνει seek, δεν αλλάζει το επιλεγμένο τη δεδομένη στιγμή handle.

Παραδείγματα

Η συνηθισμένη περίπτωση - μετατροπή ενός filehandle της Perl στον αριθμό fd που θέλει μια κλήση συστήματος:

```
open my $fh, "<", "/etc/hostname" or die $!;
print fileno $fh, "\n";             # e.g. "3"
```

Τα τυπικά handles έχουν πάντα σταθερούς περιγραφείς 0, 1, 2:

```
print fileno STDIN,  "\n";          # 0
print fileno STDOUT, "\n";          # 1
print fileno STDERR, "\n";          # 2
```

Έλεγχος αν δύο handles είναι αντίγραφα του ίδιου υποκείμενου περιγραφέα (το ιδίωμα από το upstream POD, προσαρμοσμένο για την περίπτωση -1):

```
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";
}
```

Τα handles στη μνήμη δεν έχουν αντικείμενο πυρήνα, οπότε η `fileno` επιστρέφει -1:

```
my $buf = "";
open my $mh, ">", \$buf or die $!;
print fileno $mh, "\n";             # -1
```

Δημιουργία bitmap περιγραφέων για `select` - η σχολική χρήση της `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);
```

Handles καταλόγων, όπου η πλατφόρμα υποστηρίζει `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: $!";
}
```

Οριακές περιπτώσεις

- **Κλειστό handle**: επιστρέφει undef. Καμία προειδοποίηση υπό `use warnings` - η `fileno` είναι σκόπιμα σιωπηλή ώστε να μπορεί να χρησιμοποιηθεί ως ανιχνευτής.
- **Ποτέ-ανοιχτό bareword**: το `fileno NOSUCH` επιστρέφει undef· η strict mode θα αρνηθεί το bareword κατά τη μεταγλώττιση αν δεν έχει δηλωθεί.
- **Handles βαθμωτών στη μνήμη**: επιστρέφουν `-1`, όχι undef. Η αντιμετώπιση του `-1` ως «ανοιχτό αλλά μη χρησιμοποιήσιμο για κλήσεις συστήματος» είναι το προβλεπόμενο συμβόλαιο - το ίδιο το handle εξακολουθεί να λειτουργεί με print, readline κτλ.
- **Παγίδα `fileno(-1)`**: ένα ρητό `if (fileno $fh) { ... }` είναι σφάλμα έτοιμο να συμβεί: ο `fd 0` είναι το STDIN. Χρησιμοποιήστε `defined` και συγκρίνετε ρητά με το `-1`.
- **Έμμεσο filehandle μέσω έκφρασης**: αν το `FILEHANDLE` είναι οποιαδήποτε έκφραση εκτός από bareword ή απλό βαθμωτό, η τιμή του λαμβάνεται ως **έμμεσο όνομα handle** (συμβολοσειρά), όχι ως αντικείμενο handle. Για να περάσετε ένα handle αποθηκευμένο σε δομή δεδομένων, αποαναφέρετε πρώτα:

```
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
```

- **Handle καταλόγου σε πλατφόρμες χωρίς `dirfd(3)`**: επιστρέφει undef και θέτει το `$!`. Το Linux διαθέτει `dirfd(3)`, οπότε στους υποστηριζόμενους στόχους του pperl αυτή η διαδρομή δεν ασκείται.
- **Tied handles**: η `fileno` καλεί τη μέθοδο `FILENO` της κλάσης tie αν είναι ορισμένη· διαφορετικά επιστρέφει undef. Η tied κλάση είναι υπεύθυνη να επιστρέψει μια λογική τιμή (`-1` για «κανένας fd» είναι η σύμβαση).
- **Dup-αρισμένα handles**: δύο handles που άνοιξαν με open από τον ίδιο υποκείμενο fd (π.χ. `open my $copy, ">&", $orig`) επιστρέφουν τον **νέο** περιγραφέα, όχι τον αρχικό - η dup δημιούργησε φρέσκο fd.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- open - δημιουργεί τα filehandles που επιθεωρεί η `fileno`· οι λειτουργίες `>&` και `<&=` δέχονται αριθμούς περιγραφέων που κάνουν πλήρη κύκλο μέσω της `fileno`
- opendir - πηγή handles καταλόγων για τη μορφή dirhandle της `fileno`
- close - μετά από την οποία η `fileno` στο ίδιο handle επιστρέφει undef
- select - η μορφή τεσσάρων ορισμάτων καταναλώνει bitmaps κατασκευασμένα από τιμές `fileno`

- `binmode` - λειτουργεί στο στρώμα PerlIO πάνω από το fd· η `fileno` φτάνει από κάτω του

Ροή ελέγχου

finally

Εκτελεί κώδικα εκκαθάρισης κατά την έξοδο από ένα `try`/`catch`, είτε το σώμα ολοκληρώθηκε επιτυχώς, είτε εξεγέρθηκε εξαίρεση, είτε υπήρξε άλμα προς τα έξω.

Η `finally` εισάγει ένα τρίτο μπλοκ που μπορεί να ακολουθήσει ένα ζεύγος `try`/`catch`. Το σώμα του εκτελείται ακριβώς μία φορά, αφού ολοκληρωθεί το σώμα του `try` - με όποιον τρόπο και αν ολοκληρώθηκε. Φυσιολογική ολοκλήρωση, εξαίρεση που πιάστηκε από το `catch`, εξαίρεση που διαφεύγει επειδή δεν υπάρχει αντίστοιχο `catch`, `return` από την περικλείουσα υπορουτίνα, `goto`, `last`/`next`/`redo` που εγκαταλείπει τον περικλείοντα βρόχο: όλες αυτές οι διαδρομές περνούν μέσα από το μπλοκ `finally` πριν συνεχίσει ο έλεγχος. Το μοτίβο είναι ισοδύναμο με τον προγραμματισμό ενός μπλοκ `defer` στην αρχή του `try`, με μια σύνταξη που κρατά την εκκαθάριση οπτικά συνδεδεμένη με το try/catch στο οποίο ανήκει.

Σύνοψη

```
use feature 'try';
no warnings 'experimental::try';

try    { BODY }
catch ($e) { HANDLE }
finally { CLEANUP }
```

Τι επιστρέφεται

Η `finally` δεν έχει τιμή επιστροφής. Η τελική έκφραση του μπλοκ αποτιμάται για τις παρενέργειές της και κατόπιν απορρίπτεται - **δεν** συνεισφέρει στην τιμή της περικλείουσας έκφρασης `try`/`catch`, ακόμη και αν ολόκληρη η κατασκευή είναι η τελευταία εντολή μιας υπορουτίνας ή ενός μπλοκ `do` που χρησιμοποιείται για την παραγωγή τιμής.

```
my $v = do {
    try    { compute() }
    catch ($e) { $DEFAULT }
    finally { log("done"); 999 }   # 999 is discarded
};
# $v is either compute()'s value or $DEFAULT
```

Χρησιμοποιήστε την `finally` για εκκαθάριση που πρέπει οπωσδήποτε να συμβεί, όχι για τον υπολογισμό αποτελέσματος.

Κατάσταση χαρακτηριστικού

`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 5.44. Using it without silencing the warning produces:

```
finally is experimental at ... line N.
```

Ενεργοποιήστε το χαρακτηριστικό και σιγάστε την προειδοποίηση μαζί:

```
use feature 'try';
no warnings 'experimental::try';
```

Επειδή η κατηγορία προειδοποίησης είναι `experimental::try` - η ίδια που εξέπεμπαν τα `try`/`catch` - η σίγασή της ενεργοποιεί την `finally` χωρίς να επανενεργοποιεί προειδοποιήσεις στα πλέον σταθερά τμήματα του χαρακτηριστικού για τα οποία θέλετε διαγνωστικά.

Πότε εκτελείται το μπλοκ

Η σύμβαση είναι «κατά την έξοδο, με όποιον τρόπο και αν συμβεί». Συγκεκριμένα, η `finally` εκτελείται μετά από καθένα από τα ακόλουθα:

- **Φυσιολογική ολοκλήρωση του `try`**. Το σώμα του `try` έφτασε στο τέλος του ή επέστρεψε την τιμή της τελευταίας του έκφρασης· εκτελείται έπειτα η `finally`, και μετά η εντολή μετά την κατασκευή `try`.
- **Εξαίρεση που πιάστηκε από το `catch`**. Το σώμα του `try` εξήγειρε εξαίρεση, το `catch` ταίριαξε και τη χειρίστηκε· η `finally` εκτελείται μετά το σώμα του `catch`.
- **Εξαίρεση που δεν πιάστηκε**. Δεν υπάρχει `catch`, ή το ίδιο το `catch` την επανεξήγειρε με `die`· η `finally` εκτελείται ούτως ή άλλως, και κατόπιν η εξαίρεση συνεχίζει να διαδίδεται προς την επόμενη δυναμική `eval`/`try` προς τα έξω.
- **`return` μέσα σε `try` ή `catch`**. Η περικλείουσα υπορουτίνα πρόκειται να επιστρέψει· η `finally` εκτελείται, και μετά τίθεται σε ισχύ η επιστροφή.
- **Έλεγχος βρόχου (`last`, `next`, `redo`) μέσα σε `try` ή `catch`**. Η `finally` εκτελείται, και μετά ο έλεγχος βρόχου τίθεται σε ισχύ στον περικλείοντα βρόχο.
- **`goto &sub` ή `goto LABEL` που εγκαταλείπει το `try`**. Η `finally` εκτελείται πριν ολοκληρωθεί το άλμα.

Η σειρά έχει σημασία: αν το ίδιο το `catch` εγείρει νέα εξαίρεση, η `finally` εκτελείται ούτως ή άλλως, και η νέα εξαίρεση είναι αυτή που διαδίδεται (η αρχική αντικαθίσταται, ακριβώς όπως συμβαίνει με κάθε `die` μέσα σε χειριστή).

Περιορισμοί στο σώμα του μπλοκ

Ένα μπλοκ `finally` είναι σημείο εκκαθάρισης, όχι ανακατεύθυνσης ροής ελέγχου. Η Perl απορρίπτει οποιαδήποτε εντολή ροής ελέγχου που θα προσπαθούσε να παρακάμψει τον τρόπο με τον οποίο η κατασκευή ξετυλίγεται ήδη:

- Το **`return`** μέσα σε `finally` αποτελεί συντακτικό σφάλμα / σφάλμα μεταγλώττισης.
- Το **`goto`** μέσα σε `finally` αποτελεί συντακτικό σφάλμα / σφάλμα μεταγλώττισης.

- **Οι έλεγχοι βρόχου** (`last`, `next`, `redo`) που στοχεύουν βρόχο *εκτός* του μπλοκ `finally` απορρίπτονται.

Μέσα σε έναν απλό βρόχο που *περιέχεται* στο σώμα της `finally`, οι έλεγχοι βρόχου εφαρμόζονται σε εκείνον τον εσωτερικό βρόχο κανονικά - ο περιορισμός αφορά την έξοδο από το μπλοκ `finally`, όχι την επανάληψη στο εσωτερικό του.

Η `die` επιτρέπεται. Ένα μπλοκ `finally` που εγείρει εξαίρεση αντικαθιστά οποιαδήποτε εν εξελίξει εξαίρεση από το `try`/`catch` με τη νέα. Αυτό είναι συνήθως σφάλμα (χάνετε την αρχική αποτυχία), οπότε προστατεύστε κώδικα εκκαθάρισης που ίσως αποτύχει και ο ίδιος:

```
finally {
    eval { $fh->close };        # swallow close errors during
↪cleanup
}
```

Σχέση με την `defer`

Το `finally BLOCK` είναι «`defer BLOCK` τοποθετημένο στην αρχή του `try`, που όμως είναι προσαρτημένο στο ζεύγος try/catch». Και τα δύο εκτελούνται σε κάθε διαδρομή εξόδου από την περιέχουσα εμβέλειά τους· και τα δύο απαγορεύουν τις ίδιες εντολές ροής ελέγχου στο σώμα τους· και τα δύο απορρίπτουν την τιμή επιστροφής τους.

Η διαφορά βρίσκεται στην προσάρτηση εμβέλειας:

- Η `defer` προσαρτάται στο *εσώτερο περικλείον μπλοκ* και ενεργοποιείται κατά την έξοδο εκείνου του μπλοκ. Αποτελεί το πρωτογενές στοιχείο εκκαθάρισης γενικής χρήσης και λειτουργεί σε οποιοδήποτε μπλοκ.
- Η `finally` προσαρτάται στην *κατασκευή* `try`/`catch` και ενεργοποιείται όταν η κατασκευή ολοκληρώνεται. Είναι συντακτική ζάχαρη για τη συνηθισμένη περίπτωση όπου η εκκαθάριση αφορά συγκεκριμένα αυτό που έκανε το `try`.

```
# These two are behaviourally equivalent:

try {
    my $fh = acquire();
    defer { release($fh) }
    work($fh);
}
catch ($e) { warn $e }

try {
    my $fh = acquire();
    work($fh);
}
catch ($e) { warn $e }
finally { release_last() }      # if $fh were available here
```

Σημειώστε μια πρακτική συνέπεια: οι μεταβλητές που δηλώνονται μέσα στο σώμα του `try` εξέρχονται από την εμβέλεια προτού εκτελεστεί η `finally`. Αν η εκκαθάριση χρειάζεται έναν πόρο που έχει δεσμευτεί μέσα στο `try`, είτε δηλώστε τον σε εξωτερική εμβέλεια, είτε χρησιμοποιήστε `defer` μέσα στο `try` εκεί όπου η μεταβλητή είναι εν ζωή.

Παραδείγματα

Πάντα κλείστε ένα handle, ακόμη και σε σφάλμα:

```
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;
}
```

Απελευθερώστε ένα κλείδωμα ανεξαρτήτως αποτελέσματος:

```
my $lock = $mutex->lock;
try {
    critical_section();
}
finally {
    $lock->release;
}
```

Παρατηρησιμότητα: καταγράψτε κάθε διαδρομή εξόδου:

```
try {
    run($job);
}
catch ($e) {
    $job->mark_failed($e);
    die $e;                     # re-throw; finally still runs
}
finally {
    metrics->tick('job.exit', $job->id);
}
```

try/catch που παράγει τιμή, με την finally αποκλειστικά για εκκαθάριση:

```
my $result = do {
    try    { fetch($url) }
    catch ($e) { $CACHED }
    finally { close_connection() }
};
# $result is fetch() or $CACHED; finally's value is never used
```

Οριακές περιπτώσεις

- **Το `catch` μπορεί να παραλειφθεί.** Το `try { ... } finally { ... }` είναι νόμιμο και εκτελεί την `finally` τόσο σε φυσιολογική όσο και σε εξαιρετική έξοδο· η εξαίρεση στη συνέχεια διαδίδεται σαν να μην υπήρχε το `try`.
- **Εμφωλευμένα `try`/`finally`**: τα μπλοκ `finally` ενεργοποιούνται από το εσώτερο προς το εξώτερο καθώς ξετυλίγονται οι εμβέλειες, σύμφωνα με τη σημασιολογία της `defer`.
- **Εξαίρεση μέσα στην `finally`**: αντικαθιστά οποιαδήποτε εν εξελίξει εξαίρεση. Η αρχική χάνεται, εκτός αν την έχετε συλλάβει νωρίτερα στο `catch`.
- **Η `$@` κατά τη διάρκεια της `finally`**: δεν είναι αξιόπιστη για το «τι μόλις συνέβη». Αν η `finally` χρειάζεται να γνωρίζει αν εκτελείται μετά από επιτυχία ή αποτυχία, ορίστε μια σημαία στο μπλοκ `catch` και ελέγξτε την:

```
my $failed;
try   { risky() }
catch ($e) { $failed = $e; die $e }
finally {
    if ($failed) { rollback() } else { commit() }
}
```

- **Καθολική καταστροφή**: τα μπλοκ `finally` που έχουν καταχωριστεί από ένα `try` το οποίο είναι ακόμη ενεργό όταν ο διερμηνέας αρχίζει να αποδομείται εκτελούνται, αλλά ισχύουν οι ίδιες επιφυλάξεις με την `die` μέσα σε `DESTROY` - κρατήστε το σώμα αμυντικό.
- **Η συντακτική ανάλυση απαιτεί το χαρακτηριστικό**: χωρίς `use feature 'try'`, η λέξη `finally` δεν είναι δεσμευμένη λέξη και αναλύεται ως κλήση συνάρτησης τύπου bareword, που συνήθως παράγει ένα μπερδεμένο, άσχετο σφάλμα. Ενεργοποιείτε πάντα το χαρακτηριστικό πριν χρησιμοποιήσετε τη σύνταξη.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `try` - εισάγει το μπλοκ στην έξοδο του οποίου εγκαθίσταται η `finally`· δείτε *Try Catch Exception Handling* στο perlsyn
- `catch` - το προαιρετικό ενδιάμεσο μπλοκ που χειρίζεται τις εξαιρέσεις που εγείρονται από το `try`· η `finally` εκτελείται μετά από αυτό σε κάθε περίπτωση
- `defer` - το γενικής χρήσης μπλοκ «εκτέλεση κατά την έξοδο εμβέλειας»· η `finally` είναι `defer` προσαρτημένο σε ζεύγος `try`/`catch`
- `die` - ο μόνος τρόπος για να ενεργοποιηθεί η διακλάδωση `catch`· ένα `die` μέσα σε `finally` αντικαθιστά οποιαδήποτε εν εξελίξει εξαίρεση
- `eval` - ο παλαιότερος μηχανισμός σύλληψης εξαιρέσεων· τα `try`/`catch`/`finally` αντικαθιστούν το κοινό ιδίωμα `eval { ... }; if ($@)` με καθαρότερη εμβέλεια

και σωστή σημασιολογία ροής ελέγχου

I/O

flock

Τοποθετεί συμβουλευτικό κλείδωμα σε ένα ανοιχτό αρχείο.

Η `flock` είναι η φορητή πρωτογενής λειτουργία κλειδώματος αρχείων της Perl. Καλεί τη υποκείμενη `flock(2)`, `fcntl(2)` ή `lockf(3)` - όποια παρέχει η πλατφόρμα - στο `FILEHANDLE` και μπλοκάρει εξ ορισμού μέχρι να μπορέσει να χορηγηθεί το ζητούμενο κλείδωμα. Τα κλειδώματα καλύπτουν **ολόκληρο το αρχείο**, ποτέ ένα εύρος bytes, και είναι **συμβουλευτικά**: συντονίζουν μόνο μεταξύ διεργασιών που επίσης καλούν `flock` στο ίδιο αρχείο. Κώδικας που αγνοεί το κλείδωμα μπορεί ακόμα να διαβάσει ή να τροποποιήσει το περιεχόμενο.

Σύνοψη

```
flock FILEHANDLE, OPERATION
```

Τι επιστρέφεται

Αληθής τιμή σε επιτυχία, ψευδής τιμή σε αποτυχία (με την `$!` ορισμένη). Ελέγχετε πάντα την τιμή επιστροφής - με `LOCK_NB` η αποτυχία είναι το κανονικό σήμα «κάποιος άλλος το κρατά», όχι σφάλμα:

```
flock($fh, LOCK_EX | LOCK_NB)
    or die "already locked: $!";
```

Σε πλατφόρμα που δεν υλοποιεί κανένα από τα `flock(2)`, κλείδωμα `fcntl(2)` ή `lockf(3)`, η `flock` εγείρει μοιραίο σφάλμα αντί να επιστρέψει ψευδή τιμή. Στο Linux (τη μόνη υποστηριζόμενη πλατφόρμα της pperl) αυτό δεν συμβαίνει ποτέ.

Τιμές της OPERATION

Η `OPERATION` είναι ένας από τους αμοιβαία αποκλειόμενους τύπους αιτήματος, προαιρετικά συνδυασμένος με δυαδικό OR με τη σημαία μη μπλοκαρίσματος. Οι παραδοσιακές αριθμητικές τιμές είναι 1, 2, 4, 8 αλλά ο φορητός κώδικας εισάγει τα συμβολικά ονόματα από το `Fcntl`, είτε ξεχωριστά είτε ως ομάδα μέσω της ετικέτας `:flock`:

```
use Fcntl qw(:flock);
```

- `LOCK_SH` - κοινόχρηστο κλείδωμα. Πολλαπλές διεργασίες μπορούν να κρατούν κοινόχρηστο κλείδωμα στο ίδιο αρχείο ταυτόχρονα. Χρησιμοποιείται για αναγνώστες.
- `LOCK_EX` - αποκλειστικό κλείδωμα. Μόνο μία διεργασία μπορεί να το κρατά· καμία άλλη διεργασία δεν μπορεί να κρατά κανένα κλείδωμα στο αρχείο ταυτόχρονα. Χρησιμοποιείται για εγγραφείς.
- `LOCK_UN` - απελευθερώνει όποιο κλείδωμα κρατά αυτή τη στιγμή αυτό το filehandle.

- LOCK_NB - δυαδικό OR με LOCK_SH ή LOCK_EX για να γίνει η κλήση μη μπλοκαριστική· η flock επιστρέφει ψευδή τιμή αμέσως αν το κλείδωμα δεν μπορεί να χορηγηθεί, αντί να περιμένει.

Μια κλήση flock με νέα κατάσταση αντικαθιστά ατομικά υπάρχον κλείδωμα στο ίδιο filehandle (για παράδειγμα, αναβάθμιση από LOCK_SH σε LOCK_EX).

Καθολική κατάσταση που επηρεάζει

- Ορίζει την $! σε αποτυχία.
- Εκκενώνει τον ενταμιευτή εξόδου του FILEHANDLE πριν το κλείδωμα ή το ξεκλείδωμα, ώστε τα δεδομένα της κλειδωμένης περιοχής να φτάσουν στο OS πριν μια άλλη διεργασία δει την αλλαγή κατάστασης κλειδώματος.

Παραδείγματα

Αποκλειστικό κλείδωμα για εγγραφέα, με μπλοκάρισμα μέχρι να χορηγηθεί:

```
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;
```

Μη μπλοκαριστική προσπάθεια - παράλειψη της εργασίας αν τρέχει ήδη άλλη διεργασία:

```
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
```

Κλασικός appender mailbox. Παρατηρήστε την seek στο SEEK_END μετά το κλείδωμα - σε πολύ παλιά συστήματα η κατάσταση προσθήκης >> δεν είναι ατομική, οπότε η θέση εγγραφής πρέπει να επανακαθιερωθεί μόλις κρατηθεί το κλείδωμα:

```
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;
```

Κοινόχρηστο κλείδωμα ανάγνωσης κατά τη φόρτωση αρχείου ρυθμίσεων, επιτρέποντας σε άλλους αναγνώστες να προχωρούν ταυτόχρονα:

```
use Fcntl qw(:flock);
open my $fh, "<", "config.dat" or die $!;
flock($fh, LOCK_SH)             or die "lock: $!";
my @lines = <$fh>;
close $fh;                      # releases the lock
```

Αναβάθμιση κοινόχρηστου κλειδώματος σε αποκλειστικό - η δεύτερη `flock` εναλλάσσει ατομικά τις καταστάσεις:

```
flock($fh, LOCK_SH) or die $!;
# ... inspect contents ...
flock($fh, LOCK_EX) or die $!;  # now we intend to write
```

Οριακές περιπτώσεις

- **Τα κλειδώματα είναι ανά filehandle, όχι ανά αρχείο.** Δύο ανεξάρτητα ανοίγματα της ίδιας διαδρομής έχουν το καθένα τη δική του θέση κλειδώματος. Γι" αυτό το κλείδωμα του ίδιου του αρχείου σας δύο φορές από την ίδια διεργασία μέσω διαφορετικών handles μπορεί να οδηγήσει σε αδιέξοδο ή να συμπεριφερθεί απρόσμενα σε συστήματα βασισμένα σε `fcntl(2)`.
- **Το κλείσιμο του filehandle απελευθερώνει το κλείδωμα.** Δεν απαιτείται ξεχωριστό βήμα «ξεκλειδώματος» κατά την έξοδο του προγράμματος· η `LOCK_UN` είναι χρήσιμη μόνο όταν θέλετε να κρατήσετε το handle ανοιχτό μετά.
- **Η αποτυχία `LOCK_NB` δεν είναι croak.** Επιστρέφει ψευδή τιμή και θέτει την `$!` σε `EWOULDBLOCK`. Αντιμετωπίστε το ως αναμενόμενο αποτέλεσμα, όχι ως σφάλμα.
- **Μόνο συμβουλευτικά.** Μια διεργασία που δεν καλεί ποτέ `flock` δεν βλέπει κανένα κλείδωμα. Χρησιμοποιήστε την `flock` ως πρωτόκολλο συνεργασίας μεταξύ των δικών σας προγραμμάτων, όχι ως μηχανισμό ασφαλείας.
- **Μόνο ολόκληρα αρχεία.** Δεν υπάρχει κλείδωμα εύρους bytes μέσω της `flock`. Για κλείδωμα σε επίπεδο εγγραφής χρησιμοποιήστε `fcntl` με `F_SETLK` / `F_SETLKW`.
- **NFS και άλλα δικτυακά συστήματα αρχείων.** Ορισμένες παλιότερες υλοποιήσεις NFS δεν μπορούν να τιμήσουν την `flock` ανάμεσα σε υπολογιστές· η κλήση μπορεί σιωπηρά να γίνει no-op ή να επιστρέψει επιτυχία χωρίς να παρέχει αποκλεισμό. Χρησιμοποιήστε κλείδωμα μέσω `fcntl` αν ο δικτυακός συντονισμός έχει σημασία.
- **Κληρονομικότητα στο fork.** Όταν η πλατφόρμα χρησιμοποιεί πραγματική `flock(2)` (όπως το Linux), τα κλειδώματα κληρονομούνται μέσω του `fork`· το παιδί τα μοιράζεται με τον γονέα. Σε πλατφόρμες που εξομοιώνουν μέσω `fcntl(2)` τα κλειδώματα δεν επιβιώνουν του fork, πράγμα που κάνει τη συγγραφή διακομιστών κλειδώματος αρκετά πιο δύσκολη.
- **Εκκένωση ενταμιευτή.** Η Perl εκκενώνει τον ενταμιευτή εξόδου του `FILEHANDLE` σε κάθε κλήση `flock`. Κώδικας που συνδυάζει ενταμιευμένη `print` με κλείδωμα δεν χρειάζεται ρητή εκκένωση πριν το όριο του κλειδώματος.
- **Αλληλεπίδραση με ανοίγματα τύπου dup.** Ένα άνοιγμα της μορφής `open(my $A, ">>&=", $B)` μοιράζεται τον υποκείμενο περιγραφέα αρχείου με το `$B`, οπότε οι `flock($A)` και `flock($B)` ενεργούν στην ίδια θέση κλειδώματος. Ένα dup `>>&` δίνει σε κάθε handle τον δικό του περιγραφέα και τη δική του θέση κλειδώματος.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44. 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.

Δείτε επίσης

- `fcntl` - κλείδωμα εύρους bytes και συμβουλευτικό κλείδωμα με λεπτότερο έλεγχο· χρησιμοποιήστε την όταν η κοκκιδοποίηση σε επίπεδο ολόκληρου αρχείου είναι υπερβολικά αδρή ή όταν το απαιτούν δικτυακά συστήματα αρχείων
- `open` - παράγει το filehandle στο οποίο λειτουργεί η `flock`· να θυμάστε ότι η απαιτούμενη κατάσταση ανοίγματος (< έναντι >) έχει σημασία για τις `LOCK_SH` / `LOCK_EX` που εξομοιώνονται μέσω `fcntl`
- `close` - απελευθερώνει οποιοδήποτε `flock` εξακολουθεί να κρατείται στο handle
- `seek` - συνδυάστε με `flock` όταν προσθέτετε σε αρχεία που άλλες διεργασίες ενδέχεται να επέκτειναν ενώ περιμένατε για το κλείδωμα
- `fork` - η κληρονομικότητα κλειδωμάτων σε διεργασίες-παιδιά εξαρτάται από την υποκείμενη πρωτογενή λειτουργία κλειδώματος
- `Fcntl` - πηγή των σταθερών `LOCK_SH`, `LOCK_EX`, `LOCK_UN`, `LOCK_NB` και της ετικέτας εισαγωγής `:flock`

Διεργασίες

fork

Δημιουργεί μια νέα διεργασία που εκτελεί το ίδιο πρόγραμμα στο ίδιο σημείο.

Η `fork` εκδίδει την κλήση συστήματος `fork(2)`. Μετά την επιστροφή της, **δύο** διεργασίες εκτελούν το ίδιο πρόγραμμα Perl στην ίδια εντολή - η αρχική (γονέας) και ένα σχεδόν πανομοιότυπο αντίγραφο (παιδί). Οι δύο διακρίνονται μόνο από την τιμή επιστροφής της `fork`. Οι περιγραφείς αρχείων και, σε ορισμένα συστήματα, τα κλειδώματα που κρατούνται σε αυτούς μοιράζονται μεταξύ γονέα και παιδιού· καθετί άλλο αντιγράφεται. Στο Linux η αντιγραφή είναι φθηνή: οι σελίδες δεδομένων μοιράζονται copy-on-write, οπότε η πραγματική διπλασίαση συμβαίνει μόνο καθώς κάθε πλευρά γράφει στη μνήμη.

Σύνοψη

```
my $pid = fork;
```

Τι επιστρέφεται

Τρεις τιμές επιστροφής, μία ανά ρόλο:

- **Γονέας**: το PID του παιδιού (θετικός ακέραιος).
- **Παιδί**: `0`.

- **Αποτυχία**: `undef`, με το `$!` ορισμένο στο `errno` από την υποκείμενη `fork(2)` (συνήθως `EAGAIN` - όριο πόρων που έχει εξαντληθεί - ή `ENOMEM`).

Η κανονική αποστολή είναι μια τριμερής διακλάδωση. Ελέγξτε πρώτα για `undef`, μετά για `0`, ώστε η διαδρομή του γονέα να είναι η εξ ορισμού:

```
my $pid = fork // die "fork failed: $!";
if ($pid == 0) {
    # child
    exec $cmd, @args;
    die "exec failed: $!";
}
# parent continues here, with $pid = child's PID
```

Ο ιδιωματισμός δύο γραμμών «fork or die»:

```
defined(my $pid = fork) or die "fork: $!";
```

Η συγγραφή `fork or die` (χωρίς `defined`) αποτελεί σφάλμα: αντιμετωπίζει τη νόμιμη επιστροφή `0` του παιδιού ως αποτυχία και πεθαίνει σε κάθε παιδί.

Εκκενώστε πριν κάνετε fork

Η Perl επιχειρεί να εκκενώσει όλους τους χειριστές εξόδου πριν από την κλήση `fork(2)`, αλλά δεν θα πρέπει να βασίζεστε μόνο σε αυτό - οποιαδήποτε μη εκκενωμένα δεδομένα ενταμιευτή που τελικά διαφεύγουν διπλασιάζονται και στις δύο διεργασίες, οπότε τα ίδια bytes γράφονται δύο φορές. Η κανονική άμυνα είναι να απενεργοποιήσετε την ενταμίευση σε κάθε filehandle που σκοπεύετε να χρησιμοποιήσετε διασταυρωτά σε ένα 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: $!";
```

Η φόρτωση του `IO::Handle` δεν απαιτείται για το `autoflush` στη σύγχρονη Perl - η μέθοδος είναι διαθέσιμη σε όλα τα filehandles. Ο ορισμός του `$|` στο τρέχον επιλεγμένο handle έχει το ίδιο αποτέλεσμα για αυτό το ένα handle.

Συγκομιδή παιδιών - ή `$SIG{CHLD}`

Κάθε παιδί που τερματίζει πριν ο γονέας το συγκομίσει γίνεται **zombie**: μια εγγραφή στον πίνακα διεργασιών του πυρήνα που κρατείται ανοιχτή ώστε ο γονέας να μπορεί να διαβάσει την κατάσταση εξόδου. Αν τα αγνοήσετε, ο πίνακας διεργασιών γεμίζει.

Δύο τρόποι για να το κρατήσετε καθαρό:

- Συγκομίστε ενεργά με `wait` ή `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
```

- Πείτε στον πυρήνα ότι δεν σας ενδιαφέρει:

```
$SIG{CHLD} = 'IGNORE';            # children auto-reaped, no
↪status
```

  Χρήσιμο για εργάτες τύπου fire-and-forget. Χάνετε την κατάσταση εξόδου - οι `wait` και `waitpid` επίσης δεν θα βλέπουν πλέον τα παιδιά.

Για γονείς μακράς διάρκειας που ενδιαφέρονται για την κατάσταση εξόδου, εγκαταστήστε έναν χειριστή συγκομιδής που αποστραγγίζει όλα τα εκκρεμή παιδιά χωρίς να μπλοκάρει:

```
$SIG{CHLD} = sub {
    while ((my $pid = waitpid -1, POSIX::WNOHANG) > 0) {
        # optionally inspect $? here
    }
};
```

Δείτε το `perlipc` για πληρέστερα μοτίβα.

Τι κληρονομεί το παιδί

- **Ανοιχτοί περιγραφείς αρχείων**: μοιραζόμενοι. Ένα `print` σε οποιαδήποτε από τις δύο διεργασίες προωθεί την ίδια υποκείμενη μετατόπιση αρχείου. Κλείστε τους περιγραφείς που δεν χρειάζεται το παιδί, και ανοίξτε εκ νέου τα handles που το παιδί δεν θα έπρεπε να μοιράζεται με τον γονέα - ιδιαίτερα τα STDIN/STDOUT/STDERR όταν είναι συνδεδεμένα σε σωλήνα ή υποδοχή που οδηγεί τον καλούντα του γονέα. Ένα σενάριο CGI σε φόντο που κάνει fork και εξέρχεται χωρίς να κλείσει το κληρονομημένο του STDOUT θα αφήσει τον πελάτη HTTP να κρέμεται, επειδή το παιδί κρατά ακόμη την υποδοχή ανοιχτή:

```
open STDIN,  '<', '/dev/null' or die $!;
open STDOUT, '>', '/dev/null' or die $!;
open STDERR, '>', '/dev/null' or die $!;
```

- **Μνήμη**: λογικά αντιγραμμένη, φυσικά copy-on-write. Η τροποποίηση μιας μεγάλης δομής δεδομένων στη μία πλευρά υλοποιεί μόνο τις σελίδες που τροποποιήθηκαν.
- **Process ID**: αλλάζει. Το `$$` (επίσης `$PID` / `$PROCESS_ID` υπό το English) διαβάζεται εκ νέου από τον πυρήνα κατά την πρόσβαση μετά από fork, οπότε τόσο ο γονέας όσο και το παιδί βλέπουν το δικό τους PID.
- **Χειριστές σημάτων, %ENV, κατάλογος εργασίας, umask, ομάδα διεργασιών, ελεγκτικό τερματικό**: όλα αντιγράφονται.

- **Εκκρεμή alarms και χρονομετρητές**: δεν κληρονομούνται από το παιδί (κατά POSIX).
- **Κλειδώματα που κρατούνται μέσω `flock`**: μοιράζονται με τον γονέα - και οι δύο διεργασίες κρατούν το ίδιο κλείδωμα στην ίδια περιγραφή ανοιχτού αρχείου. Τα κλειδώματα που αποκτώνται μέσω κλειδωμάτων εγγραφής `fcntl` συμπεριφέρονται διαφορετικά· συμβουλευτείτε το `fcntl(2)`.

Παραδείγματα

Κλασικό fork/exec - εκκινήστε μια εξωτερική εντολή χωρίς να περάσετε από το κέλυφος:

```
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 $?;
```

Διασκορπίστε N εργάτες, μετά συγκομίστε τους όλους:

```
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;
```

Επικοινωνία γονέα/παιδιού μέσω σωλήνα. Ανοίξτε τον σωλήνα **πριν** το fork ώστε και οι δύο πλευρές να κληρονομήσουν τα δύο άκρα:

```
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";
```

Παρατηρήστε το `close` στο μη χρησιμοποιούμενο άκρο σε κάθε διεργασία - αφήστε και τα δύο άκρα ανοιχτά και στις δύο διεργασίες και ο αναγνώστης δεν θα δει ποτέ EOF.

Αποσπάστε ένα παιδί τύπου δαίμονα και αφήστε τον πυρήνα να το συγκομίσει:

```
$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
```

Οριακές περιπτώσεις

- **`fork` μέσα σε νήμα ή μετά από κώδικα ευαίσθητο στο `DESTROY`**: μόνο το καλούν νήμα επιβιώνει στο παιδί· οτιδήποτε περιμένει σε mutex ή συνενωμένο νήμα σε άλλο νήμα χάνεται. Κρατήστε τα σημεία fork πριν δημιουργήσετε νήματα.
- **Filehandles με ενταμιευτή στο παιδί**: οποιαδήποτε δεδομένα ακόμη ενταμιευμένα τη στιγμή του fork είναι παρόντα στους ενταμιευτές και των δύο διεργασιών. Όταν κάθε πλευρά τελικά εκκενώσει, τα bytes γράφονται δύο φορές. Κάντε autoflush πριν το fork.
- **Η `exit` στο παιδί εκτελεί τα μπλοκ `END` και τους `DESTROY`-ers** που γράφτηκαν από τον γονέα. Αυτό σπάνια είναι αυτό που θέλετε - προτιμήστε `POSIX::_exit` στο παιδί μετά από αποτυχία `exec`, ή εξασφαλίστε ότι τα μπλοκ `END` ελέγχουν το `$$` έναντι του PID που καταγράφηκε κατά την εκκίνηση του προγράμματος.
- **Σύγχυση σφάλματος και παιδιού**: η `fork` επιστρέφει `0` νόμιμα στο παιδί και `undef` σε αποτυχία. Χρησιμοποιήστε `//` και `defined`, ποτέ `||` / ελέγχους αλήθειας.
- **`EAGAIN`**: παροδικό - έχει συμπληρωθεί το όριο διεργασιών ανά χρήστη του πυρήνα. Δοκιμάστε ξανά με σύντομο sleep, ή μειώστε την ταυτοχρονικότητά σας.
- **Ο κατάλογος εργασίας και το `chdir` στο παιδί** δεν επηρεάζουν τον γονέα. Το ίδιο ισχύει για το `%ENV`, το `umask` και τους χειριστές σημάτων: το παιδί έχει το δικό του αντίγραφο από τη στιγμή του fork και μετά.
- **Οι αναζητήσεις σε κοινούς περιγραφείς μοιράζονται**: αν γονέας και παιδί γράφουν και οι δύο στο ίδιο κληρονομημένο `STDOUT`, η έξοδός τους εναλλάσσεται ανάλογα με ό,τι αποφασίζει ο χρονοπρογραμματιστής του ΛΣ. Συντονιστείτε με ρητό κλείδωμα ή δώστε σε κάθε παιδί το δικό του handle εξόδου.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Η pperl στοχεύει αποκλειστικά σε Linux, οπότε η `fork` είναι πάντα πραγματική `fork(2)`. Σε αντίθεση με την παραδοσιακή perl5 στα Windows - όπου η `fork` εξομοιώνεται με *ψευδο-διεργασίες* σε επίπεδο διερμηνέα που μοιράζονται μία διεργασία ΛΣ, με τις ιδιαιτερότητες που τεκμηριώνονται στο `perlfork` - η `fork` της pperl δημιουργεί μια ανεξάρτητη διεργασία ΛΣ με το δικό της PID, χώρο διευθύνσεων και εγγραφή στον πίνακα διεργασιών. Κώδικας που υπέθετε σημασιολογία ψευδο-διεργασιών (κοινή καθολική κατάσταση μεταξύ των «forked» πλευρών) δεν θα μεταφερθεί στην pperl.

Δείτε επίσης

- `exec` - αντικαθιστά την τρέχουσα εικόνα διεργασίας· η τυπική επόμενη κλήση σε παιδί αμέσως μετά από `fork`
- `wait` - μπλοκάρει μέχρι να εξέλθει οποιοδήποτε παιδί και επιστρέφει το PID του και την κατάστασή του στο `$?`
- `waitpid` - συγκομίστε ένα συγκεκριμένο παιδί, προαιρετικά χωρίς μπλοκάρισμα μέσω `WNOHANG`
- `exit` - τερματίζει την τρέχουσα διεργασία· χρησιμοποιήστε στο παιδί για να το τερματίσετε χωρίς να πέσει στον κώδικα του γονέα
- `pipe` - δημιουργήστε ένα ζεύγος συνδεδεμένων filehandles πριν το fork για να δώσετε σε γονέα και παιδί ένα κανάλι επικοινωνίας
- `$$` - το τρέχον ID διεργασίας, διαβάζεται εκ νέου μετά από `fork` ώστε κάθε πλευρά να βλέπει το δικό της PID
- `%SIG` - ιδίως το `$SIG{CHLD}`, το οποίο ελέγχει αν τα παιδιά συγκομίζονται αυτόματα ή πρέπει να συλλεχθούν με `wait` / `waitpid`
- `perlipc` - πληρέστερη πραγμάτευση του forking, των σημάτων, των σωλήνων και της συγκομιδής ετοιμοθάνατων παιδιών

I/O

format

Δηλώνει ένα πρότυπο αναφοράς βασισμένο σε εικόνα προς χρήση από την `write`.

Η `format` είναι **δήλωση**, όχι εκτελέσιμη εντολή. Συνδέει μια ονοματισμένη εικόνα - ένα μπλοκ από γραμμές εξόδου σταθερών στηλών με ενσωματωμένους δείκτες πεδίων - στον πίνακα συμβόλων, όπου αργότερα η `write` την αναζητά με το όνομά της. Οι εικόνες δεν παρεμβάλλουν μεταβλητές· κάθε γραμμή εικόνας ακολουθείται από μια γραμμή ορισμάτων, οι εκφράσεις της οποίας γεμίζουν τα πεδία με τη σειρά. Η `format` είναι παλαιό χαρακτηριστικό της Perl· το upstream σημειώνει ότι η χρήση της είναι περιορισμένη και παραπέμπει στο `perlform` για την πλήρη γραμματική των πεδίων.

Σύνοψη

```
format NAME =
FORMLIST
.

format =           # declares the "STDOUT" format
FORMLIST
.
```

Μια `FORMLIST` είναι μια ακολουθία γραμμών, καθεμία από τρία είδη: σχόλιο (# στη στήλη 1), **γραμμή εικόνας**, ή **γραμμή ορισμάτων** που παρέχει τιμές για τη γραμμή εικόνας ακριβώς από πάνω της. Ένα μοναδικό `.` στη στήλη 1 τερματίζει τη δήλωση.

Τι επιστρέφεται

Η `format` είναι δήλωση και δεν έχει τιμή κατά τον χρόνο εκτέλεσης. Εγκαθιστά τη μεταγλωττισμένη εικόνα στον πίνακα μορφών (format table) του τρέχοντος πακέτου με το όνομα `NAME` (ή `STDOUT` αν παραλειφθεί το `NAME`). Δεν καλείται· ενεργοποιείται έμμεσα από την `write`.

Τα κομμάτια μιας εικόνας

Κάθε γραμμή εικόνας αναμειγνύει κυριολεκτικό κείμενο με **δείκτες πεδίων**. Ένα πεδίο αρχίζει με `@` (κανονικό πεδίο) ή `^` (ειδικό πεδίο, που χρησιμοποιείται για τη λειτουργία γεμίσματος και για αριθμητικά πεδία με δυνατότητα να μένουν κενά) και εκτείνεται μέσω χαρακτήρων γεμίσματος που ορίζουν τόσο το πλάτος όσο και τη στοίχιση:

- `<` - κείμενο αριστερής στοίχισης
- `>` - κείμενο δεξιάς στοίχισης
- `|` - κεντραρισμένο κείμενο
- `#` - αριθμητικό με δεξιά στοίχιση· ένα προπορευόμενο `0` γεμίζει με μηδενικά· το `.` τοποθετεί την υποδιαστολή
- `...` - στο τέλος πεδίου κειμένου, αποδίδει τρεις τελείες όταν η τιμή έχει αποκοπεί
- `@*` - πεδίο μεταβλητού πλάτους που εκπέμπει αυτούσιο ένα πολυγραμμικό βαθμωτό
- `^*` - πεδίο μεταβλητού πλάτους που εκπέμπει την επόμενη γραμμή ενός βαθμωτού και καταναλώνει αυτό το τμήμα της μεταβλητής πηγής

Δύο σύμβολα καταστολής μπορούν να εμφανιστούν οπουδήποτε σε μια γραμμή εικόνας και δεν αποτελούν μέρος κανενός πεδίου:

- `~` - αν κάθε πεδίο σε αυτήν τη γραμμή είναι κενό, η γραμμή καταστέλλεται εξ ολοκλήρου· διαφορετικά το ίδιο το `~` αποδίδεται ως κενό
- `~~` - όπως το `~`, και επιπλέον επαναλαμβάνει τη γραμμή μέχρι να εξαντληθεί κάθε πεδίο της (κάθε βαθμωτό πηγής να είναι κενό ή κάθε έκφραση τύπου `shift` να επιστρέφει undef)

Το `perlform.pod` δίνει την πλήρη γραμματική των γραμμών εικόνας συμπεριλαμβανομένης της λειτουργίας γεμίσματος, των πολυγραμμικών πεδίων, της υπερχείλισης (`####`) και των εξαρτημένων από locale υποδιαστολών.

Η γραμμή ορισμάτων

Η γραμμή αμέσως κάτω από μια γραμμή εικόνας είναι η **γραμμή ορισμάτων**: μια λίστα εκφράσεων χωρισμένων με κόμματα, μία ανά πεδίο, η οποία αξιολογείται σε περιβάλλον λίστας πριν αποδοθεί η εικόνα. Μια μοναδική έκφραση που επιστρέφει λίστα μπορεί να γεμίσει πολλαπλά πεδία. Η λίστα ορισμάτων μπορεί να διασπαστεί σε πολλές γραμμές αν και μόνο αν το πρώτο σύμβολο στην πρώτη γραμμή είναι άγκιστρο ανοίγματος:

```
format Report =
@<<<<<<<< @>>>>>>
{ $name,
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
  $score }
.
```

Καθολικές μεταβλητές που ελέγχουν την `write`

Η έξοδος μέσω της `format` οδηγείται εξ ολοκλήρου από καθολικές μεταβλητές ανά filehandle. Καθεμία είναι προσαρτημένη στο **τρέχον επιλεγμένο filehandle**· η αλλαγή τους για άλλο handle απαιτεί έναν χορό `select` (βλέπε παρακάτω):

- `$~` - όνομα της τρέχουσας μορφής που χρησιμοποιείται για κάθε εγγραφή (προεπιλογή είναι το ίδιο το όνομα του filehandle)
- `$^` - όνομα της τρέχουσας μορφής κορυφής σελίδας (προεπιλογή είναι το όνομα του filehandle με προσαρτημένο `_TOP`)
- `$=` - μήκος σελίδας σε γραμμές (προεπιλογή 60)
- `$-` - γραμμές που απομένουν στην τρέχουσα σελίδα· η `write` το μειώνει και ενεργοποιεί τη μορφή κορυφής σελίδας όταν φτάσει στο μηδέν
- `$%` - τρέχων αριθμός σελίδας, αυξάνεται αυτόματα
- `$:` - χαρακτήρες πάνω στους οποίους μπορεί να σπάει η λειτουργία γεμίσματος (^) (προεπιλογή `" \n-"`)
- `$^L` - συμβολοσειρά που εκπέμπεται πριν από κάθε κορυφή σελίδας εκτός της πρώτης (προεπιλογή `"\f"`)

Υπό το `use English` αυτές είναι οι `$FORMAT_NAME`, `$FORMAT_TOP_NAME`, `$FORMAT_LINES_PER_PAGE`, `$FORMAT_LINES_LEFT`, `$FORMAT_PAGE_NUMBER`, `$FORMAT_LINE_BREAK_CHARACTERS` και `$FORMAT_FORMFEED`.

Επειδή αυτές είναι ανά handle, ο ορισμός τους για handle διαφορετικό από το τρέχον επιλεγμένο χρειάζεται `select`:

```
my $ofh = select $fh;
$~ = 'Report';
$^ = 'Report_TOP';
select $ofh;
```

Κορυφή φόρμας

Όταν η `write` μειώσει το `$-` στο μηδέν, εκπέμπει το `$^L` (εκτός από την πρώτη σελίδα), επανατοποθετεί το `$-` σε `$=`, αυξάνει το `$%` και επικαλείται τη μορφή που ονομάζει το `$^`. Το προεπιλεγμένο όνομα μορφής κορυφής σελίδας είναι το όνομα του filehandle με προσαρτημένο `_TOP`, οπότε η δήλωση του `STDOUT_TOP` αρκεί για εγγραφές μέσω του `STDOUT`.

Παραδείγματα

Μια ελάχιστη αναφορά με κεφαλίδα:

```
format STDOUT_TOP =
        Inventory
Name            Qty   Price
---------------------------
.
format STDOUT =
@<<<<<<<<<<<<<  @>>  @##.##
$name,          $qty, $price
.

for my $row (@rows) {
    ($name, $qty, $price) = @$row;
    write;
}
```

Εναλλαγή μορφών για handle που δεν είναι επιλεγμένο:

```
open my $fh, '>', 'report.txt' or die $!;
my $ofh = select $fh;
$~ = 'Report';
$^ = 'Report_TOP';
select $ofh;
write $fh;
```

Λειτουργία γεμίσματος με ~~ για να αποστραγγίσετε ένα πολυγραμμικό βαθμωτό μία γραμμή τη φορά:

```
format STDOUT =
Note: ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
      $note
~~    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
      $note
.
```

Τα πεδία ^ καταναλώνουν το `$note` καθώς αποδίδονται· η γραμμή ~~ επαναλαμβάνεται μέχρι το `$note` να είναι κενό.

Οριακές περιπτώσεις

- **Λεξιλογική ορατότητα**: μια format βλέπει μια μεταβλητή `my` μόνο όταν δηλώνεται μέσα στην εμβέλεια αυτής της μεταβλητής. Μια format ανώτατου επιπέδου δεν μπορεί να δει μεταβλητές `my` δηλωμένες μέσα σε υπορουτίνα· χρησιμοποιήστε καθολικές μεταβλητές πακέτου ή δηλώστε την format μέσα στην υπορουτίνα.
- **Εικόνες σε εισαγωγικά**: αν κατασκευάσετε μια εικόνα με `eval` από συμβολοσειρά με διπλά εισαγωγικά, το `@name` φαίνεται ως παρεμβολή πίνακα. Διαφύγετέ το ή χρησιμοποιήστε μονά εισαγωγικά γύρω από το πρότυπο.
- **Μια μοναχική . είναι ο τερματιστής**: μια εικόνα της οποίας το κυριολεκτικό κείμενο θα ξεκινούσε με . στη στήλη 1 πρέπει να εσοχίζει αυτήν τη γραμμή. Αυτό επίσης τσιμπάει κώδικα format που στέλνεται μέσω αφελών συστημάτων μεταφοράς αλληλογραφίας που κόβουν στο . σε γραμμή μόνο του.

- **Αριθμητική υπερχείλιση**: ένα πεδίο # στενότερο από τον μορφοποιημένο αριθμό αποδίδεται ως #### αντί να αποκόπτεται σιωπηρά.
- **Υποδιαστολή locale**: αν το `use locale` ήταν σε ισχύ όταν **δηλώθηκε** η format, το `LC_NUMERIC` επιλέγει τον χαρακτήρα υποδιαστολής. Η εναλλαγή του `use locale` κατά τη στιγμή της `write` δεν αλλάζει την εικόνα.
- **Χαρακτήρες ελέγχου** μέσα σε πεδίο κειμένου σταθερού πλάτους αντικαθίστανται από κενά για να διατηρηθεί η στοίχιση στηλών.
- **Ανάμειξη `print` και `write`**: νόμιμη στο ίδιο handle, αλλά το `$-` παρακολουθεί μόνο τις γραμμές που εκπέμπει η `write`. Προσαρμόστε το `$-` χειροκίνητα αν εναλλάξιμη έξοδος `print` καταναλώνει γραμμές σελίδας για τις οποίες ενδιαφέρεστε.
- **Δεν υπάρχει μηχανισμός υποσέλιδου**: δεν υπάρχει `$FORMAT_BOTTOM_NAME`. Τα υποσέλιδα γίνονται χειροκίνητα - ελέγξτε το `$-` πριν από κάθε `write` και εκπέμψτε εσείς το υποσέλιδο όταν φτάσει σε κατώφλι.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `write` - αποδίδει μία εγγραφή μέσω της τρέχουσας μορφής στο επιλεγμένο filehandle· ο μόνος τρόπος με τον οποίο μια δηλωμένη format παράγει έξοδο
- `select` - αλλάζει ποιο filehandle είναι το τρέχον, απαιτείται όταν ορίζετε τα `$~`, `$^`, `$=`, `$-`, `$%`, `$:` ή `$^L` για handle διαφορετικό από το `STDOUT`
- `$~` - όνομα της τρέχουσας μορφής
- `$^` - όνομα της τρέχουσας μορφής κορυφής σελίδας
- `$=` - μήκος σελίδας
- `$-` - γραμμές που απομένουν στην τρέχουσα σελίδα
- `$%` - τρέχων αριθμός σελίδας
- `$:` - χαρακτήρες διακοπής σε λειτουργία γεμίσματος
- `perlform.pod` - πλήρης γραμματική γραμμών εικόνας, κανόνες λειτουργίας γεμίσματος, εσωτερικά `formline` / `$^A`, και επεξεργασμένα παραδείγματα αναφορών

Διάφορα

formline

Μορφοποιεί μια λίστα τιμών σε μια συμβολοσειρά εικόνας και προσαρτά το αποτέλεσμα στον συσσωρευτή μορφοποίησης `$^A`.

Η `formline` είναι η μηχανή χαμηλού επιπέδου πίσω από τον μηχανισμό αναφορών `format` / `write` της Perl. Δέχεται μια συμβολοσειρά `PICTURE` που περιγράφει πλάτη και στοίχιση πεδίων, παρεμβάλλει τις τιμές από τη `LIST` σε αυτά τα πεδία, και προσαρτά το προκύπτον κείμενο στον καθολικό συσσωρευτή `$^A` (γνωστό επίσης ως `$ACCUMULATOR` υπό `use English`). Κάθε κλήση στη `write` εκκενώνει το `$^A` στο σχετιζόμενο filehandle

και το καθαρίζει· εκτός της `write`, διαχειρίζεστε εσείς το `$^A` - διαβάστε το για το μορφοποιημένο κείμενο, και μετά επαναφέρετέ το σε `""` πριν την επόμενη κλήση.

Το `PICTURE` είναι η ίδια μίνι-γλώσσα που χρησιμοποιεί η `format`: `@<<<<` για πεδίο κειμένου με αριστερή στοίχιση, `@>>>>` για δεξιά στοίχιση, `@||||` για κεντραρισμένο, `@####.##` για αριθμητικό πεδίο με δεξιά στοίχιση και υποδιαστολή, `@*` για απεριόριστο πεδίο κειμένου πολλών γραμμών, και `^<<<` (και παρόμοια) για το πεδίο λειτουργίας γεμίσματος που καταναλώνει σταδιακά το βαθμωτό όρισμά του. Δείτε `perldoc perlform` για το πλήρες λεξιλόγιο της εικόνας.

Σύνοψη

```
formline PICTURE, LIST
formline $picture, @values
```

Τι επιστρέφεται

Η `formline` επιστρέφει πάντα 1. Η χρήσιμη έξοδος είναι η παρενέργεια: το μορφοποιημένο κείμενο **προσαρτάται** στο `$^A`. Αν χρειάζεστε το κείμενο ως τιμή, διαβάστε το `$^A` μετά την κλήση και έπειτα καθαρίστε το:

```
$^A = "";
formline '@<<<<<<<<<  @>>>>', $name, $score;
my $line = $^A;
$^A = "";
```

Επειδή το αποτέλεσμα πηγαίνει σε συσσωρευτή, διαδοχικές κλήσεις της `formline` συσσωρεύουν έξοδο πολλών γραμμών χωρίς ενδιάμεσο join.

Καθολική κατάσταση που επηρεάζει

- `$^A` - ο **συσσωρευτής μορφοποίησης**. Η `formline` προσαρτά σε αυτόν· δεν καθαρίζεται ποτέ αυτόματα. Γνωστός επίσης ως `$ACCUMULATOR` υπό `use English`.
- `$:` - το σύνολο των χαρακτήρων στους οποίους επιτρέπεται να σπάνε τα πεδία λειτουργίας γεμίσματος (`^*`, `^<<<`). Εξ ορισμού `" \n-"`. Γνωστή επίσης ως `$FORMAT_LINE_BREAK_CHARACTERS` υπό `use English`.
- Locale `LC_NUMERIC` - αν είναι ενεργό το `use locale` όταν μεταγλωττίζεται η εικόνα, ο χαρακτήρας υποδιαστολής στα αριθμητικά πεδία (`@###.##`) ακολουθεί το locale.
- Τα πεδία λειτουργίας γεμίσματος (`^...`) **μεταλλάσσουν επιτόπου τα βαθμωτά ορίσματά τους**, αποκόπτοντας το τμήμα που καταναλώθηκε ώστε η επόμενη κλήση `formline` να συνεχίσει από εκεί που σταμάτησε η προηγούμενη. Περάστε αντίγραφο αν χρειάζεται να διατηρηθεί το πρωτότυπο.

Παραδείγματα

Δημιουργία μίας μορφοποιημένης γραμμής και ανάγνωσή της:

```
$^A = "";
formline '@<<<<<<<<  @>>>>>', "widget", 42;
print $^A, "\n";            # "widget       42\n"
$^A = "";
```

Δημιουργία ενός μπλοκ πολλών γραμμών σε μία κλήση - η εικόνα μπορεί να περιέχει newlines, και κάθε γραμμή της εικόνας καταναλώνει τις δικές της τιμές από τη LIST:

```
$^A = "";
formline <<'END', "Alice", 30, "Bob", 25;
@<<<<<<<<<<<  @###
@<<<<<<<<<<<  @###
END
print $^A;                  # "Alice         30\nBob           25\n"
$^A = "";
```

Μια βοηθητική swrite - η formline είναι για τη write ό,τι η sprintf για την printf:

```
sub swrite {
    my $picture = shift;
    local $^A = "";
    formline $picture, @_;
    return $^A;
}

my $row = swrite '@<<<<<<<<  @>>>>', "total", 1234;
```

Το πεδίο λειτουργίας γεμίσματος καταναλώνει το βαθμωτό του σε επαναλαμβανόμενες κλήσεις - παρατηρήστε ότι το $text μικραίνει κάθε φορά:

```
my $text = "the quick brown fox jumps over the lazy dog";
$^A = "";
formline '^<<<<<<<<<<<<<<', $text;
formline "\n";
formline '^<<<<<<<<<<<<<<', $text;
print $^A;                  # two lines, word-broken on spaces
$^A = "";
```

Αριθμητικό πεδίο με δεκαδικά και γέμισμα υπερχείλισης - μια τιμή που δεν χωρά αποδίδεται ως χαρακτήρες #:

```
$^A = "";
formline '@##.##', 3.14159;
formline "\n";
formline '@##.##', 9999.9;  # overflow
print $^A;                  # "  3.14\n######\n"
$^A = "";
```

Οριακές περιπτώσεις

- **Παράθεση της εικόνας**: μέσα σε διπλά εισαγωγικά, το @ εισάγει παρεμβολή πίνακα. Γράψτε τις εικόνες ως συμβολοσειρές σε μονά εισαγωγικά ή ως heredocs με τερματιστή σε εισαγωγικά (<<'END') για να κρατήσετε το @ κυριολεκτικό. Η κλασική παγίδα: το `formline "@<<<<<<<", $name` παρεμβάλλει το @< προτού η formline δει καν τη συμβολοσειρά.
- **Newlines στην εικόνα**: η formline δεν ενδιαφέρεται για το πλήθος των newlines που περιέχει το PICTURE. Τα σύμβολα ~ (απόκρυψη-αν-όλα-κενά) και ~~ (επανάληψη-έως-εξαντλήσεως) αντιμετωπίζουν ολόκληρη την εικόνα ως μία λογική γραμμή, οπότε μια μορφή εγγραφής με ~~ τυπικά χρειάζεται μία formline ανά φυσική γραμμή. Μια δήλωση format χωρίζει μια πολυγραμμή φόρμα σε μία κλήση formline ανά γραμμή εξόδου· όταν καλείτε απευθείας τη formline, κάνετε αυτό το χωρισμό μόνοι σας.
- **Ο συσσωρευτής δεν καθαρίζεται**: το πιο συνηθισμένο λάθος είναι να ξεχάσει κανείς να επαναφέρει το $^A μεταξύ λογικών χρήσεων. Η προσάρτηση είναι η ορισμένη συμπεριφορά, όχι σφάλμα. Χρησιμοποιήστε `local $^A = "";` μέσα σε βοηθητικές συναρτήσεις.
- **Πλήθος πεδίων έναντι πλήθους τιμών**: οι περισσευούμενες τιμές στη LIST αγνοούνται σιωπηλά· οι ελλείπουσες τιμές αποδίδονται ως κενές (κανονικό πεδίο) ή κενώνονται (ειδικό αριθμητικό πεδίο ^).
- **Το undef σε κανονικό πεδίο** μετατρέπεται σε κενή συμβολοσειρά υπό use warnings και πυροδοτεί προειδοποίηση uninitialized. Ένα ειδικό πεδίο (^###) κενώνει ολόκληρο το πεδίο σε undef χωρίς προειδοποίηση.
- **Χρήση βαθμωτού μορφοποίησης - αποτύπωση της εξόδου της write**: το άνοιγμα ενός filehandle πάνω σε αναφορά βαθμωτού είναι συχνά ευκολότερο από την οδήγηση της formline με το χέρι:

```
open my $fh, ">", \my $out or die $!;
format FH =
@<<<<<<<<  @>>>>
$name,     $score
.
$~ = "FH";
select((select($fh), $~ = "FH")[0]);
write $fh;
```

  Καταφύγετε στη formline απευθείας όταν θέλετε τη διαδρομή εικόνα-προς-συμβολοσειρά χωρίς εμπλοκή filehandle ή κατονομασμένης δήλωσης format.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `write` - οδηγεί μία ή περισσότερες κλήσεις `formline` από μια κατονομασμένη `format` και εκκενώνει το `$^A` σε filehandle
- `format` - συντακτικό δήλωσης για τα ζεύγη εικόνας/τιμής που η `write` τροφοδοτεί στη `formline`
- `sprintf` - η άλλη πρωτογενής λειτουργία «μορφοποίηση σε συμβολοσειρά»· χρησιμοποιήστε `sprintf` για οδηγίες `%` τύπου `printf`, `formline` για στήλες αναφορών σταθερού πλάτους και μπλοκ κειμένου λειτουργίας γεμίσματος
- `$^A` - ο συσσωρευτής στον οποίο γράφει η `formline`· διαβάστε τον για το αποτέλεσμα, καθαρίστε τον πριν την επόμενη λογική χρήση
- `$:` - χαρακτήρες στους οποίους μπορούν να σπάσουν τα πεδία λειτουργίας γεμίσματος (^*, ^<<<)

I/O

getc

Διαβάζει τον επόμενο μονό χαρακτήρα από ένα filehandle.

Η `getc` επιστρέφει έναν χαρακτήρα εισόδου από το `FILEHANDLE`, προωθώντας τη θέση ανάγνωσης του handle κατά αυτόν τον χαρακτήρα. Αν παραλειφθεί το `FILEHANDLE`, η `getc` διαβάζει από το `STDIN`. Σε τέλος αρχείου ή σε σφάλμα ανάγνωσης, η `getc` επιστρέφει `undef` και - στην περίπτωση σφάλματος - θέτει το `$!`. Η μονάδα είναι **χαρακτήρας**, όχι byte: υπό στρώμα PerlIO `:utf8` ή `:encoding(...)` μία κλήση καταναλώνει όσα bytes συγκροτούν ένα αποκωδικοποιημένο σημείο κώδικα.

Σύνοψη

```
getc FILEHANDLE
getc
```

Τι επιστρέφεται

Συμβολοσειρά ενός χαρακτήρα σε περίπτωση επιτυχίας, ή `undef` σε τέλος αρχείου ή σφάλμα. Διακρίνετε τις δύο περιπτώσεις ελέγχοντας το `$!`:

```
my $ch = getc $fh;
if (!defined $ch) {
    die "read error: $!" if $!;
    # otherwise clean EOF
}
```

Επειδή το `undef` συμπίπτει με την κενή συμβολοσειρά `""` σε συγκρίσεις τύπου συμβολοσειράς, ελέγχετε πάντα με `defined`, ποτέ με `length` ή αντιστοίχιση συμβολοσειράς.

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται όταν η `getc` επιστρέφει `undef` λόγω σφάλματος ανάγνωσης (δεν τίθεται σε καθαρό τέλος αρχείου).
- Το τρέχον filehandle εισόδου - η μορφή χωρίς όρισμα διαβάζει από το `STDIN`, όχι από το handle που έχει επιλεχθεί με την `select`. Η `select` διέπει την προεπιλεγμένη έξοδο, όχι την προεπιλεγμένη είσοδο.
- Τη στοίβα στρωμάτων PerlIO του handle - καθορίζει αν το επιστρεφόμενο βαθμωτό είναι ένα byte ή ένας αποκωδικοποιημένος χαρακτήρας, και αν εφαρμόζεται μετατροπή CRLF στο πέρασμα προς τα μέσα.

Παραδείγματα

Ανάγνωση ενός χαρακτήρα από το `STDIN`:

```
my $ch = getc;
print "got: $ch\n" if defined $ch;
```

Ανάγνωση από ρητό filehandle και ανίχνευση τέλους αρχείου:

```
open my $fh, "<", "input.txt" or die "open: $!";
while (defined(my $ch = getc $fh)) {
    print $ch;
}
close $fh;
```

Ανάγνωση χαρακτήρων, όχι bytes, από αρχείο UTF-8. Κάθε κλήση επιστρέφει ένα αποκωδικοποιημένο σημείο κώδικα ακόμα κι όταν η υποκείμενη ροή bytes χρησιμοποιεί ακολουθίες πολλαπλών bytes:

```
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
```

Κατανάλωση προτροπής ναι/όχι χωρίς αναμονή για ολόκληρη γραμμή - απαιτεί πρώτα την απενεργοποίηση του line buffering του τερματικού (δείτε *Οριακές περιπτώσεις*):

```
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";
```

Οριακές περιπτώσεις

- **Line buffering τερματικού**: σε διαδραστικό `STDIN`, ο οδηγός τερματικού δεν παραδίδει τίποτα στην Perl μέχρι ο χρήστης να πατήσει Enter, οπότε η `getc` μπλοκάρει μέχρι να είναι διαθέσιμη ολόκληρη γραμμή και μετά επιστρέφει μόνο τον πρώτο της χαρακτήρα. Η ίδια η `getc` δεν μπορεί να το παρακάμψει. Για να διαβάσετε ένα πάτημα πλήκτρου, βάλτε πρώτα το τερματικό σε μη

κανονική κατάσταση («cbreak») - καλέστε εξωτερικά `stty`, χρησιμοποιήστε `POSIX::termios` μέσω του αρθρώματος `POSIX` για μια φορητή λύση μέσα στη διεργασία, ή χρησιμοποιήστε το άρθρωμα CPAN `Term::ReadKey` για φιλικότερη διεπαφή. Θυμηθείτε να επαναφέρετε την κατάσταση του τερματικού κατά την έξοδο, συμπεριλαμβανομένης της εξόδου από χειριστές σημάτων.

- **Bytes έναντι χαρακτήρων υπό στρώματα I/O**: η τιμή επιστροφής είναι ό,τι παράγει η στοίβα στρωμάτων του handle. Ένα handle σε λειτουργία bytes αποδίδει ένα byte ανά κλήση· ένα handle `:utf8` ή `:encoding(...)` αποδίδει έναν αποκωδικοποιημένο χαρακτήρα ανά κλήση, καταναλώνοντας από τη ροή όσα bytes παίρνει αυτός ο χαρακτήρας. Παραμορφωμένη είσοδος υπό στρώμα αποκωδικοποίησης πυροδοτεί τη συνήθη προειδοποίηση ή σφάλμα του στρώματος αντί να επιστρέφει σιωπηρά σκουπίδια.
- **Τέλος αρχείου έναντι σφάλματος**: και τα δύο επιστρέφουν `undef`. Το `$!` εκκαθαρίζεται πριν την είσοδο στην κλήση αλλά τίθεται μόνο όταν συνέβη πραγματικό σφάλμα ανάγνωσης, οπότε το ιδίωμα είναι `defined $ch or $!` and `die ...`.
- **Κλειστό ή μη ανοιγμένο filehandle**: επιστρέφει `undef` και θέτει το `$!` σε `Bad file descriptor`. Υπό `use warnings` εκπέμπεται προειδοποίηση `getc() on closed filehandle`.
- **Αποδοτικότητα**: η `getc` είναι έναν-χαρακτήρα-τη-φορά, που διασχίζει τη στοίβα στρωμάτων PerlIO ανά κλήση. Για μαζική σάρωση, οι `read`, `sysread`, ή `readline` είναι ουσιαστικά ταχύτερες· κρατήστε την `getc` για διαδραστικές προτροπές και αναλυτές πρωτοκόλλων που χρειάζονται γνήσια έλεγχο έναν χαρακτήρα τη φορά.
- **Το προεπιλεγμένο handle είναι το STDIN, όχι το ARGV**: σε αντίθεση με τον τελεστή «διαμάντι» `<>`, η `getc` χωρίς όρισμα δεν διασχίζει το `@ARGV` ούτε καταφεύγει στο μαγικό handle `ARGV` - διαβάζει απευθείας από το `STDIN`.
- **Ευρεία είσοδος τερματικού**: η ανάγνωση χαρακτήρα από τερματικό σε raw mode εξακολουθεί να επιστρέφει ένα σημείο κώδικα μόνο όταν το στρώμα εισόδου αποκωδικοποιεί UTF-8. Μια επικόλληση που περιέχει χαρακτήρα πολλαπλών bytes υπό handle σε λειτουργία bytes αναδύεται ως πολλές ξεχωριστές κλήσεις `getc`, μία ανά byte.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `read` - διαβάζει σταθερό αριθμό χαρακτήρων ή bytes σε μία κλήση, πολύ ταχύτερη από βρόχο `getc`
- `readline` - διαβάζει μία ολόκληρη εγγραφή (γραμμή) τη φορά, διεπόμενη από το `$/`
- `sysread` - ανάγνωση χωρίς ενταμιευτή απευθείας από το λειτουργικό, παρακάμπτοντας τη στοίβα στρωμάτων PerlIO
- `eof` - ελέγχει αν η επόμενη `getc` / `readline` θα επέστρεφε `undef` για τέλος αρχείου

- `$!` - το errno που τίθεται όταν η getc επιστρέφει `undef` λόγω σφάλματος ανάγνωσης

Πληροφορίες χρηστών και ομάδων

getgrent

Διαβάζει την επόμενη εγγραφή από τη βάση δεδομένων ομάδων του συστήματος.

Η `getgrent` διατρέχει τη βάση δεδομένων ομάδων του υπολογιστή μία εγγραφή τη φορά, όπως θα τη διάβαζε ένας απλός βρόχος πάνω από το `/etc/group` γραμμή προς γραμμή, και επιστρέφει μια άποψη κάθε εγγραφής σε μορφή Perl. Είναι η μορφή επαναλήπτη των αναζητήσεων ομάδων - χρησιμοποιήστε την όταν θέλετε κάθε ομάδα, ή κάθε ομάδα που ικανοποιεί κάποιο κατηγόρημα. Για αναζήτηση μίας εγγραφής κατά όνομα ή gid, χρησιμοποιήστε καλύτερα την `getgrnam` ή την `getgrgid`.

Η πρώτη κλήση `getgrent` ανοίγει έμμεσα τη βάση δεδομένων. Η επανάληψη συνεχίζεται μεταξύ επόμενων κλήσεων μέχρι να εξαντληθεί η βάση δεδομένων, οπότε η `getgrent` επιστρέφει την κενή λίστα (ή `undef` σε βαθμωτό περιβάλλον). Καλέστε την `setgrent` για να επιστρέψετε στην αρχή, ή την `endgrent` για να κλείσετε τη βάση δεδομένων και να απελευθερώσετε τους πόρους που κρατά η βιβλιοθήκη C.

Σύνοψη

```
my ($name, $passwd, $gid, $members) = getgrent;
my $name                            = getgrent;        # scalar␣
↪context
```

Τι επιστρέφεται

Σε **περιβάλλον λίστας**, μια λίστα τεσσάρων στοιχείων - μία εγγραφή ανά ομάδα:

Δείκτης	Πεδίο	Σημασία
0	`$name`	Όνομα ομάδας (π.χ. `"wheel"`)
1	`$passwd`	Κωδικός ομάδας, συνήθως `"x"` σε υπολογιστές με shadow passwords
2	`$gid`	Αριθμητικό gid ομάδας
3	`$members`	Συμβολοσειρά ονομάτων σύνδεσης μελών διαχωρισμένων με κενά

Όταν η επανάληψη φτάσει στο τέλος, η επιστροφή είναι η κενή λίστα. Αν γράψετε `if (my @gr = getgrent)` ο βρόχος τερματίζεται καθαρά στο τέλος της βάσης δεδομένων.

Σε **βαθμωτό περιβάλλον**, η `getgrent` επιστρέφει μόνο το `$name`. Όταν η βάση δεδομένων εξαντληθεί, επιστρέφει `undef` - το σήμα που ελέγχουν τα περισσότερα σενάρια για να τερματίσουν έναν βρόχο `while`.

Το `$members` είναι απλή συμβολοσειρά, όχι πίνακας. Διασπάστε το σε λευκούς χαρακτήρες όταν χρειάζεστε τη λίστα:

```
my @logins = split ' ', $members;
```

Καθολική κατάσταση που επηρεάζει

Η getgrent διατηρεί **κατάσταση επανάληψης ανά διεργασία** μέσα στη βιβλιοθήκη C (νοητικά, μια θέση αρχείου μέσα στη βάση δεδομένων ομάδων). Δύο κλήσεις στην ίδια διεργασία μοιράζονται αυτή την κατάσταση· ένας εμφωλευμένος βρόχος που καλεί getgrent μέσα σε άλλον βρόχο getgrent θα μπερδέψει τις εγγραφές μεταξύ τους και θα χάσει τις περισσότερες. Σειριοποιήστε τα περάσματα, ή αποτυπώστε όλες τις εγγραφές πρώτα σε πίνακα:

```
my @all;
while (my @gr = getgrent) { push @all, [ @gr ] }
endgrent;
```

Ο επαναλήπτης δεν μηδενίζεται όταν επιστρέψει μια υπορουτίνα. Αν κώδικας βιβλιοθήκης που δεν ελέγχετε καλεί getgrent, μετακινεί τη θέση σας. Προστατέψτε με setgrent όταν χρειάζεστε καθαρή εκκίνηση.

Παραδείγματα

Διατρέχει κάθε ομάδα και τυπώνει το όνομα και το πλήθος μελών:

```
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;
```

Συλλέγει όλες τις ομάδες στις οποίες ανήκει ένα δεδομένο όνομα σύνδεσης:

```
my $user = 'alice';
my @groups;
while (my ($name, undef, undef, $members) = getgrent) {
    push @groups, $name
        if grep { $_ eq $user } split ' ', $members;
}
endgrent;
```

Βαθμωτό περιβάλλον - απαριθμεί κάθε όνομα ομάδας, ένα ανά γραμμή:

```
while (defined(my $name = getgrent)) {
    print $name, "\n";
}
endgrent;
```

Επαναφορά και επανανάγνωση, για παράδειγμα για να χτίσετε δύο ευρετήρια σε ένα πέρασμα έκαστο:

```
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;
```

Μετατροπή αριθμητικού gid από την `stat` σε όνομα ομάδας χωρίς δεύτερη αναζήτηση μέσω cache της εξόδου του επαναλήπτη:

```
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";
```

Οριακές περιπτώσεις

- **Τέλος επανάληψης έναντι κανονικής εγγραφής**: σε περιβάλλον λίστας, μια εξαντλημένη βάση δεδομένων επιστρέφει την κενή λίστα - η οποία ανατίθεται ως μηδέν στοιχεία, οπότε το `my @gr = getgrent; @gr or last;` είναι ο ιδιωματικός έλεγχος τερματισμού. Σε βαθμωτό περιβάλλον, ελέγξτε με `defined` αντί για αλήθεια, αφού ένα νόμιμο όνομα ομάδας θα μπορούσε καταρχήν να είναι η συμβολοσειρά `"0"`.
- **Δεν υπάρχει πεδίο κωδικού στο σύγχρονο Linux**: το `$passwd` είναι σχεδόν πάντα `"x"` (οι πραγματικοί κατακερματισμοί βρίσκονται στο `/etc/gshadow`). Μην ερμηνεύσετε τιμή διαφορετική του `"x"` ως «έχει οριστεί κωδικός» χωρίς να συμβουλευτείτε τη shadow βάση δεδομένων.
- **Κενό `$members`**: ομάδες χωρίς ρητά μέλη (χρήστες που εμφανίζονται μόνο μέσω του πρωτεύοντος gid) έχουν το `$members` ως κενή συμβολοσειρά. Η `split ' ', ""` επιστρέφει κενή λίστα - ο ιδιωματικός τρόπος να πάρετε τα μέλη ως πίνακα χωρίς ειδική περίπτωση.
- **Η έμμεση συμμετοχή μέσω πρωτεύοντος gid δεν αναφέρεται**: το `$members` απαριθμεί μόνο τους χρήστες που κατονομάζονται ρητά στην εγγραφή `/etc/group` της ομάδας. Ένας χρήστης του οποίου το πρωτεύον gid ταιριάζει με αυτή την ομάδα αλλά δεν αναφέρεται επίσης στο πεδίο `members` **δεν** εμφανίζεται. Διασταυρώστε με την `getpwent` αν χρειάζεστε και τα δύο είδη συμμετοχής.
- **Η εμφωλευμένη επανάληψη αλλοιώνει τη θέση**: ένας δεύτερος βρόχος `getgrent` που ξεκινά πριν ολοκληρωθεί ο πρώτος μοιράζεται τον ίδιο δρομέα. Χρησιμοποιήστε `setgrent` ανάμεσα στα περάσματα, ή τραβήξτε στιγμιότυπο της βάσης δεδομένων στη μνήμη εκ των προτέρων.
- **Tainted κατάσταση**: το πεδίο `$passwd` σημειώνεται ως tainted υπό `-T`, όπως και για την `getpwent` - αντιμετωπίστε το ως μη έμπιστη είσοδο παρότι η συνήθης τιμή είναι `"x"`.

- **Καθολικό σε επίπεδο διεργασίας, όχι ανά thread**: σε συστήματα χωρίς thread-safe παραλλαγές η κατάσταση επανάληψης είναι ανά διεργασία, όχι ανά thread. Δύο threads που διατρέχουν `getgrent` ταυτόχρονα θα χάσουν εγγραφές.
- **Η endgrent είναι προαιρετική αλλά ευγενική**: το να αφήσετε τη βάση δεδομένων ανοιχτή ως την έξοδο του προγράμματος λειτουργεί, αλλά αν έχετε τελειώσει με την επανάληψη, καλέστε την `endgrent` ώστε η βιβλιοθήκη C να μπορεί να απορρίψει τους ενταμιευτές της.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `getgrnam` - μεμονωμένη αναζήτηση κατά όνομα ομάδας· χρησιμοποιήστε την όταν γνωρίζετε ήδη το όνομα και θέλετε μόνο μία εγγραφή
- `getgrgid` - μεμονωμένη αναζήτηση κατά αριθμητικό gid· ο συνήθης συνεργάτης της `stat` όταν χρειάζεστε όνομα ομάδας για ένα αρχείο
- `setgrent` - επαναφέρει τον επαναλήπτη στην αρχή της βάσης δεδομένων ομάδων πριν από άλλο πέρασμα πλήρους σάρωσης
- `endgrent` - κλείνει τη βάση δεδομένων και απελευθερώνει τους πόρους που κρατά ανοιχτούς η βιβλιοθήκη C
- `getpwent` - ο παράλληλος επαναλήπτης πάνω από τη βάση δεδομένων passwd· συνδυάστε με `getgrent` όταν επιλύετε πλήρεις σχέσεις χρήστη-προς-ομάδα
- `scalar` - επιβάλλει βαθμωτό περιβάλλον όταν χρειάζεστε μόνο το όνομα ομάδας και θέλετε να παρακάμψετε τα άλλα πεδία

Πληροφορίες χρηστών και ομάδων

getgrgid

Αναζητά μια εγγραφή ομάδας με βάση το αριθμητικό ID ομάδας.

Η `getgrgid` ρωτά τη βάση δεδομένων ομάδων του συστήματος για την εγγραφή της οποίας το `gid` είναι `GID` και επιστρέφει ό,τι βρει. Είναι το αριθμητικό αντίστοιχο της `getgrnam` και αντικατοπτρίζει την κλήση `getgrgid(3)` της βιβλιοθήκης C. Σε περιβάλλον λίστας λαμβάνετε τα αποπακεταρισμένα πεδία της εγγραφής· σε βαθμωτό περιβάλλον λαμβάνετε το όνομα της ομάδας.

Σύνοψη

```
my ($name, $passwd, $gid, $members) = getgrgid($gid);
my  $name                           = getgrgid($gid);
```

Τι επιστρέφεται

Σε **περιβάλλον λίστας**, τέσσερις τιμές αποπακεταρισμένες από τη `struct group`:

```
#    0        1         2      3
my ( $name,   $passwd,  $gid,  $members ) = getgrgid($gid);
```

- `$name` - το όνομα της ομάδας.
- `$passwd` - το πεδίο κωδικού πρόσβασης της ομάδας. Σε σύγχρονα συστήματα είναι τυπικά `"x"` ή κενό· τα πραγματικά hashes βρίσκονται σε μια shadow βάση δεδομένων.
- `$gid` - το αριθμητικό ID ομάδας (ο ίδιος αριθμός που περάσατε, επιστρεφόμενος από την εγγραφή).
- `$members` - μία **μοναδική συμβολοσειρά διαχωρισμένη με κενά** των ονομάτων σύνδεσης των μελών, όχι λίστα. Διαχωρίστε την εσείς όταν χρειάζεστε πίνακα:

  ```
  my @users = split / /, (getgrgid($gid))[3];
  ```

Σε **βαθμωτό περιβάλλον**, μόνο το όνομα της ομάδας:

```
my $name = getgrgid(0);    # "root" on most Linux systems
```

Αν δεν υπάρχει ομάδα με το δοθέν GID, η συνάρτηση επιστρέφει την κενή λίστα σε περιβάλλον λίστας και `undef` σε βαθμωτό περιβάλλον. Μια αποτυχία σε επίπεδο συστήματος (π.χ. μη προσπελάσιμο backend NSS) αναφέρεται με τον ίδιο τρόπο· ελέγξτε το `$!` αν χρειάζεται να διακρίνετε.

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται σε αποτυχία σε επίπεδο συστήματος της υποκείμενης κλήσης `getgrgid(3)`. Δεν τίθεται όταν το GID απλώς δεν έχει αντίστοιχη εγγραφή· αυτό αποτελεί κανονικό «δεν βρέθηκε», όχι σφάλμα.
- Ο δρομέας επανάληψης της βάσης δεδομένων ομάδων σε επίπεδο διεργασίας τον οποίο διατηρεί η libc. Η `getgrgid` **δεν** προωθεί ούτε επαναφέρει τον δρομέα που χρησιμοποιούν οι `getgrent`, `setgrent` και `endgrent`, αλλά μοιράζεται τον ίδιο υποκείμενο περιγραφέα αρχείου στις περισσότερες υλοποιήσεις - η ανάμειξη άμεσων αναζητήσεων με επανάληψη είναι φορητή θεωρητικά και εκπληκτική στην πράξη.

Παραδείγματα

Αναζητήστε το όνομα της κύριας ομάδας για την τρέχουσα διεργασία:

```
my $group = getgrgid($();           # $( is the real GID
print "running as group: $group\n";
```

Επιλύστε το ID ομάδας ενός αρχείου σε όνομα, με μια λογική εφεδρεία για ορφανά GIDs (αρχεία που ανήκουν σε ομάδα που δεν υπάρχει πλέον):

```
my $gid   = (stat $file)[5];
my $name  = getgrgid($gid) // "<unknown gid $gid>";
```

Αποπακετάρετε την πλήρη εγγραφή και απαριθμήστε τα μέλη:

```
if (my ($name, undef, $gid, $members) = getgrgid(100)) {
    my @users = split / /, $members;
    print "group $name ($gid) has ", scalar(@users), " members\n";
}
```

Προφυλαχθείτε ρητά από την περίπτωση «δεν υπάρχει τέτοια ομάδα» - η κενή λίστα σε λογικό περιβάλλον / περιβάλλον λίστας είναι ήδη ψευδής, οπότε ένα `if` αρκεί:

```
unless (my @gr = getgrgid($gid)) {
    die "no group with gid $gid\n";
}
```

Συνδυάστε με την `getpwuid` για να αποδώσετε τις στήλες ιδιοκτήτη και ομάδας μιας λίστας καταλόγου:

```
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;
```

Οριακές περιπτώσεις

- **Έχει σημασία βαθμωτό έναντι περιβάλλοντος λίστας.** Η `getgrgid($gid)` σε βαθμωτό περιβάλλον αποδίδει μόνο το όνομα της ομάδας· περιτυλίξτε σε παρενθέσεις ή εκχωρήστε σε λίστα για να λάβετε την πλήρη εγγραφή:

  ```
  my $name    = getgrgid(0);              # "root"
  my @record  = getgrgid(0);              # four-element list
  my $gid_out = (getgrgid(0))[2];         # the gid field
  ```

- **Το `$members` είναι συμβολοσειρά, όχι πίνακας.** Η μορφή με ένωση κενών είναι ιστορική και ταιριάζει με την έξοδο `pp_gpwent` / `pp_ggrent` του perl5. Ο διαχωρισμός με `/ /` (ένα κενό) είναι το κανονικό ιδίωμα· ο διαχωρισμός με `/\s+/` είναι ελαφρώς πιο επιεικής αν κάποιο backend χρησιμοποιεί στηλοθέτες.

- **Οι κενές ομάδες** επιστρέφουν το `$members` ως κενή συμβολοσειρά. Η `split / /, ""` αποδίδει την κενή λίστα, που είναι αυτό που συνήθως θέλουν οι καλούντες.

- **Αριθμητικός εξαναγκασμός.** Το `GID` περνά στην κλήση C ως ακέραιος. Ένα μη αριθμητικό βαθμωτό εξαναγκάζεται: το `"0root"` γίνεται `0`. Περάστε καθαρό ακέραιο.

- **Επικάλυψη χώρου ονομάτων GID.** Πολλαπλές εγγραφές ομάδων με το ίδιο `gid` επιτρέπονται στο `/etc/group` και στο NSS. Η `getgrgid` επιστρέφει **μία** από αυτές - ποια ορίζεται από το backend. Μη βασίζεστε στο ποιο όνομα επιστρέφει όταν τα GIDs συγκρούονται· χρησιμοποιήστε την `getgrnam` για την αντίστροφη κατεύθυνση όταν έχει σημασία το όνομα.

- **NSS / nsswitch.conf.** Η αναζήτηση ακολουθεί τον επιλογέα υπηρεσιών ονομάτων του συστήματος - `files`, `sss`, `ldap` κ.λπ. Μια επιτυχημένη αναζήτηση σε ένα πλαίσιο δοκιμών σε έναν host μπορεί να αποτύχει σε άλλον με διαφορετική διαμόρφωση NSS. Αυτή είναι ιδιότητα της βιβλιοθήκης C, όχι της Perl.
- **Μεγάλες ομάδες.** Πολύ μεγάλες λίστες μελών ομάδας (εκατοντάδες μέλη) μπορεί να φτάνουν αποκομμένες αν ο εσωτερικός ενταμιευτής του backend είναι μικρότερος από την αποδιδόμενη γραμμή. Αυτό είναι περιορισμός της libc / NSS· η Perl αναδεικνύει πιστά ό,τι επέστρεψε το σύστημα.
- **Builds με threads.** Σε συστήματα που παρέχουν την `getgrgid_r`, η perl καλεί διαφανώς την επανεισερχόμενη παραλλαγή. Οι καλούντες δεν βλέπουν διαφορά.
- **Taint.** Υπό `-T`, τα `$passwd` και `$members` επιστρέφουν μολυσμένα (tainted) - το πρώτο επειδή μπορεί να κωδικοποιεί διαπιστευτήρια, το δεύτερο επειδή οι διαχειριστές μπορούν να το αλλάξουν. Τα `$name` και `$gid` δεν είναι μολυσμένα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `getgrnam` - η αντίστροφη αναζήτηση: εγγραφή ομάδας με όνομα αντί για `gid`
- `getgrent` - επαναλάβετε κάθε εγγραφή ομάδας όταν χρειάζεστε σάρωση αντί για αναζήτηση
- `setgrent` και `endgrent` - επαναφέρετε και απελευθερώστε τον δρομέα επανάληψης που χρησιμοποιεί η `getgrent`
- `getpwuid` - το αντίστοιχο από την πλευρά passwd· συνδυάστε με την `getgrgid` για να αποδώσετε στήλες ιδιοκτήτη/ομάδας
- `User::grent` - αντικείμενο προσπέλασης ανά όνομα που υπερκαλύπτει την `getgrgid` όταν προτιμάτε το `$gr->name` αντί για αποπακετάρισμα κατά θέση

Πληροφορίες χρηστών και ομάδων

getgrnam

Αναζητά μια ομάδα Unix κατά όνομα και επιστρέφει την εγγραφή της από το `/etc/group`.

Η `getgrnam` τυλίγει την `getgrnam(3)` της βιβλιοθήκης C. Δοθέντος ενός ονόματος ομάδας όπως `"wheel"` ή `"staff"`, επιστρέφει την εγγραφή που ταιριάζει από τη βάση δεδομένων ομάδων του συστήματος - το όνομα, το placeholder κρυπτογραφημένου κωδικού, το αριθμητικό GID και τη λίστα ονομάτων σύνδεσης των μελών. Χρησιμοποιήστε την όταν έχετε στα χέρια σας ένα όνομα ομάδας και χρειάζεστε το GID ή τη λίστα μελών· χρησιμοποιήστε `getgrgid` για την αντίστροφη αναζήτηση.

Σύνοψη

```
my ($name, $passwd, $gid, $members) = getgrnam NAME
my $gid = getgrnam NAME
```

Τι επιστρέφεται

Σε **περιβάλλον λίστας** προκύπτει λίστα τεσσάρων στοιχείων:

```
my ($name, $passwd, $gid, $members) = getgrnam "wheel";
```

Δείκτης	Πεδίο	Σημασία
0	`$name`	κανονικοποιημένο όνομα ομάδας (ίδιο με το NAME σε επιτυχημένη αναζήτηση)
1	`$passwd`	πεδίο κωδικού ομάδας, συνήθως `"x"` ή κενό
2	`$gid`	αριθμητικό group ID
3	`$members`	συμβολοσειρά ονομάτων σύνδεσης μελών διαχωρισμένων με κενά

Το `$members` είναι ένα μεμονωμένο βαθμωτό, **όχι** πίνακας. Διασπάστε το οι ίδιοι όταν θέλετε λίστα:

```
my @logins = split /\s+/, (getgrnam("wheel"))[3];
```

Σε **βαθμωτό περιβάλλον** προκύπτει το αριθμητικό GID (το «άλλο», αφού η αναζήτηση έγινε κατά όνομα):

```
my $gid = getgrnam "wheel";
```

Δεν υπάρχει τέτοια ομάδα: το περιβάλλον λίστας επιστρέφει την κενή λίστα· το βαθμωτό περιβάλλον επιστρέφει `undef`. Ελέγξτε τις βαθμωτές επιστροφές για definedness αντί για αλήθεια - το GID `0` (`root`) είναι έγκυρη επιτυχημένη αναζήτηση που είναι ταυτόχρονα και ψευδής:

```
defined(my $gid = getgrnam $group) or die "no group '$group'";
```

Καθολική κατάσταση που επηρεάζει

Καμία άμεσα. Η `getgrnam` είναι αναζήτηση χωρίς κατάσταση - σε αντίθεση με την `getgrent`, δεν προωθεί τον επαναλήπτη της βάσης δεδομένων ομάδων και οι `setgrent` / `endgrent` δεν την επηρεάζουν. Μοιράζεται όμως τον στατικό ενταμιευτή αποτελεσμάτων της υποκείμενης βιβλιοθήκης C με άλλες κλήσεις `getgr*` σε πλατφόρμες χωρίς thread-safe παραλλαγές· αντιγράψτε τα πεδία που σας ενδιαφέρουν πριν καλέσετε άλλη συνάρτηση `getgr*`.

Παραδείγματα

Επίλυση ονόματος ομάδας σε GID:

```
my $gid = getgrnam "wheel";
defined $gid or die "no 'wheel' group on this system";
print "wheel GID = $gid\n";
```

Απαρίθμηση των μελών μιας ομάδας:

```
my @fields = getgrnam "staff"
    or die "no 'staff' group";
my @members = split /\s+/, $fields[3];
print "staff has ", scalar(@members), " members: @members\n";
```

Χρήση ονόματος ομάδας για τον ορισμό του effective GID (απαιτεί προνόμια):

```
my $gid = getgrnam "video"
    // die "no 'video' group";
$) = "$gid $gid";                   # effective + supplementary
```

Έλεγχος αν ένας χρήστης εμφανίζεται ως μέλος ομάδας. Σημειώστε ότι η πρωτεύουσα ομάδα (το GID του χρήστη στο `/etc/passwd`) **δεν** βρίσκεται στο `$members` - ελέγξτε και τα δύο:

```
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
}
```

Αντικειμενοστραφής μορφή μέσω του `User::grent`, που υπερκαλύπτει την ενσωματωμένη συνάρτηση με εγγραφή κατά όνομα:

```
use User::grent;
my $g = getgrnam "wheel";
printf "wheel: gid=%d members=%s\n", $g->gid, "@{$g->members}";
```

Οριακές περιπτώσεις

- **Ομάδα που λείπει**: το περιβάλλον λίστας επιστρέφει `()`, το βαθμωτό επιστρέφει `undef`. Ένα γυμνό `if (getgrnam $g)` είναι εντάξει σε περιβάλλον λίστας αλλά λάθος σε βαθμωτό περιβάλλον όταν το GID θα μπορούσε να είναι `0`:

  ```
  if (defined(my $gid = getgrnam $g)) { ... }   # correct
  if (my $gid = getgrnam $g)           { ... }  # wrong for GID 0
  ```

- **Το members είναι συμβολοσειρά, όχι λίστα**. Η λήθη αυτού και η συγγραφή `scalar @members = getgrnam $g` (ή η ευρετηρίαση `$members[0]`) δίνει σιωπηρά λάθος αποτελέσματα. Πάντα `split /\s+/` στο τέταρτο πεδίο.

- **Τα μέλη πρωτεύουσας ομάδας λείπουν από τη λίστα**. Ένας χρήστης του οποίου το GID στο `/etc/passwd` ταιριάζει με την ομάδα είναι μέλος στα μάτια του πυρήνα αλλά δεν θα εμφανιστεί στο `$members`. Οι αναζητήσεις που κατηγοριοποιούν χρήστες κατά ομάδα πρέπει να συμβουλεύονται και την `getpwnam`.
- **Το πεδίο κωδικού δεν είναι κωδικός**. Σε κάθε σύγχρονο Linux το δεύτερο πεδίο είναι `"x"` (ανακατεύθυνση προς shadow) ή κενό. Δεν είναι ποτέ ο απλός κωδικός ή κατακερματισμός κωδικού ομάδας. Μη βασίζεστε στη μορφή του.
- **Διάκριση πεζών-κεφαλαίων**: τα ονόματα ομάδων διακρίνουν πεζά-κεφαλαία στο Linux. Η `getgrnam "Wheel"` δεν βρίσκει την `wheel`.
- **Όνομα με ενσωματωμένο κενό ή άνω-κάτω τελεία**: η μορφή της βάσης δεδομένων ομάδων δεν επιτρέπει κανένα από τα δύο, οπότε τέτοια ονόματα δεν μπορούν να υπάρξουν· η αναζήτηση απλώς αποτυγχάνει.
- **NSS backends**: σε συστήματα διαμορφωμένα με LDAP, SSSD ή άλλα αρθρώματα name-service switch, η `getgrnam` συμβουλεύεται και αυτά εκτός από το `/etc/group`. Τα αποτελέσματα συνεπώς εξαρτώνται από το `/etc/nsswitch.conf` και τα διαμορφωμένα backends, όχι μόνο από το `/etc/group`.
- **Taint**: τα πεδία προέρχονται έξω από το πρόγραμμα, οπότε υπό `-T` είναι tainted και πρέπει να ξεπλυθούν πριν χρησιμοποιηθούν σε ευαίσθητες λειτουργίες.
- **Ασφάλεια threads**: η pperl αντικαθιστά αυτόματα με την thread-safe `getgrnam_r` όταν την παρέχει η πλατφόρμα. Σε πλατφόρμες χωρίς αυτήν, ταυτόχρονες κλήσεις `getgr*` σε διαφορετικά threads μπορούν να αλλοιώσουν τα αποτελέσματα μεταξύ τους.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `getgrgid` - αντίστροφη αναζήτηση κατά αριθμητικό GID· καταφύγετε σε αυτήν όταν έχετε το GID και θέλετε το όνομα ομάδας ή τη λίστα μελών
- `getgrent` - διατρέξτε κάθε ομάδα στη βάση δεδομένων όταν δεν γνωρίζετε εκ των προτέρων το όνομα ή το GID
- `setgrent` - επαναφέρει τον επαναλήπτη της `getgrent` στην αρχή της βάσης δεδομένων ομάδων
- `endgrent` - κλείνει τον περιγραφέα αρχείου της βάσης δεδομένων ομάδων που ανοίχτηκε από την επανάληψη
- `getpwnam` - η παράλληλη κλήση για τη βάση δεδομένων χρηστών· χρησιμοποιήστε την παράλληλα με την `getgrnam` όταν χρειάζεστε να επιλύσετε την πρωτεύουσα ομάδα ενός χρήστη
- `User::grent` - αντικειμενοστραφές περιτύλιγμα που υπερκαλύπτει την `getgrnam` ώστε να επιστρέφει αντικείμενο εγγραφής με κατονομασμένους accessors (`name`, `passwd`, `gid`, `members`) αντί για λίστα κατά θέση

Πληροφορίες δικτύου

gethostbyaddr

Αναζητά μια εγγραφή host με βάση την πακεταρισμένη διεύθυνση IP του.

Η `gethostbyaddr` εκτελεί αντίστροφη αναζήτηση DNS: δοθείσας μιας πακεταρισμένης δυαδικής διεύθυνσης (τέσσερα bytes για IPv4, δεκαέξι για IPv6) και της αντίστοιχης οικογένειας διευθύνσεων, καλεί τη ρουτίνα C `gethostbyaddr(3)` του συστήματος και επιστρέφει την εγγραφή host. Η τυπική περίπτωση χρήσης - *«από ποιανού μηχάνημα είναι αυτό το πακέτο;»* - θέλει τη μορφή βαθμωτού περιβάλλοντος: ένα μοναδικό όνομα host. Η μορφή περιβάλλοντος λίστας εκθέτει την πλήρη `struct hostent`: κανονικό όνομα, ψευδώνυμα, οικογένεια διευθύνσεων, μήκος διεύθυνσης και κάθε διεύθυνση που γνωρίζει ο resolver για αυτόν τον host.

Σύνοψη

```
my $name = gethostbyaddr $packed_addr, $addrtype;
my ($name, $aliases, $addrtype, $length, @addrs) = gethostbyaddr
 ↪$packed_addr, $addrtype;
```

Τι επιστρέφεται

Βαθμωτό περιβάλλον - το κανονικό όνομα host ως απλή συμβολοσειρά, ή `undef` αν αποτύχει η αντίστροφη αναζήτηση.

Περιβάλλον λίστας - λίστα πέντε στοιχείων που αντικατοπτρίζει τη `struct hostent` της C:

- `$name` - το κανονικό όνομα host στο οποίο αντιστοίχισε τη διεύθυνση ο resolver.
- `$aliases` - συμβολοσειρά εναλλακτικών ονομάτων διαχωρισμένη με κενά. Ένα βαθμωτό, όχι λίστα.
- `$addrtype` - η οικογένεια διευθύνσεων των επιστρεφόμενων διευθύνσεων, τυπικά `AF_INET` (2) ή `AF_INET6` (10).
- `$length` - το μήκος κάθε διεύθυνσης σε bytes (4 για IPv4, 16 για IPv6).
- `@addrs` - κάθε διεύθυνση που επιστρέφεται για τον host, καθεμία πακεταρισμένη δυαδική συμβολοσειρά μήκους `$length` bytes. Συχνά μόνο μία, αλλά πολυσυνδεδεμένοι hosts μπορούν να αποδώσουν περισσότερες.

Σε αποτυχία αναζήτησης σε περιβάλλον λίστας, η επιστροφή είναι η κενή λίστα.

Καθολική κατάσταση που επηρεάζει

- `$?` - τίθεται στην τιμή `h_errno` της C σε αποτυχία, όταν η πλατφόρμα την εκθέτει. Ελέγξτε με `$? != 0` για να διακρίνετε το `HOST_NOT_FOUND` από το `TRY_AGAIN` και παρόμοια παροδικά σφάλματα.
- `$!` - δεν τίθεται με ουσιαστικό τρόπο από αποτυχίες DNS. Οι αντίστροφες αναζητήσεις αποτυγχάνουν μέσω `h_errno`, όχι `errno`· μην ελέγχετε το `$!` μετά από αποτυχημένη κλήση.

- Ο δρομέας επανάληψης της βάσης δεδομένων host που μοιράζεται με τις `gethostent`, `sethostent` και `endhostent`. Μια κλήση της `gethostbyaddr` μπορεί να επαναφέρει έναν δρομέα που άφησε ανοιχτό η `gethostent`.

Παραδείγματα

Βαθμωτή μορφή - αντιστοίχιση μιας τετράδας με τελείες σε όνομα host:

```
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";
```

Πλήρης κύκλος - μπροστινή αναζήτηση, και μετά αντιστροφή της πρώτης διεύθυνσης:

```
use Socket;
my $packed = gethostbyname("www.perl.org");
my $name   = gethostbyaddr($packed, AF_INET);
print "$name\n";
```

Αναγνωρίστε τον ομότιμο σε αποδεκτή σύνδεση υποδοχής:

```
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;
```

Μορφή λίστας - κάθε όνομα και διεύθυνση που γνωρίζει ο resolver για τον host:

```
use Socket;
my $packed = inet_aton("8.8.8.8");
my ($name, $aliases, $type, $len, @addrs) = gethostbyaddr($packed,␣
↪AF_INET);
print "canonical: $name\n";
print "aliases:   $_\n" for split ' ', $aliases;
print "address:   ", inet_ntoa($_), "\n" for @addrs;
```

Διεπαφή ανά όνομα μέσω της υπερκάλυψης `Net::hostent` - μέθοδοι προσπέλασης αντί για αποπακετάρισμα κατά θέση:

```
use Net::hostent;
use Socket;
my $h = gethostbyaddr(inet_aton("127.0.0.1"), AF_INET);
print $h->name, "\n" if $h;
```

Οριακές περιπτώσεις

- **Η ADDR είναι πακεταρισμένη δυαδική συμβολοσειρά, όχι τετράδα με τελείες.** Το άμεσο πέρασμα `"127.0.0.1"` δεν λειτουργεί - η Perl θα καλέσει τον resolver με εννέα ακατέργαστα bytes και δεν θα λάβει τίποτα χρήσιμο. Περάστε πάντοτε πρώτα από τις `Socket::inet_aton`, `Socket::inet_pton`, ή ισοδύναμο packer.

- **Το ADDRTYPE πρέπει να ταιριάζει με το πλάτος της διεύθυνσης.** Η AF_INET θέλει ADDR τεσσάρων bytes· η AF_INET6 δεκαέξι. Μια αναντιστοιχία είτε επιστρέφει undef, είτε σε ορισμένες πλατφόρμες διαβάζει πέρα από τον ενταμιευτή. Χρησιμοποιήστε τη σταθερά από το Socket, όχι έναν χειρωνακτικά γραμμένο ακέραιο.
- **Ελέγξτε για undef στη βαθμωτή μορφή.** Μια εγγραφή PTR που λείπει επιστρέφει undef, όχι κενή συμβολοσειρά. Πολλές διευθύνσεις στο δημόσιο internet δεν έχουν καθόλου αντίστροφη αντιστοίχιση.
- **Ο @addrs μπορεί να μην περιλαμβάνει τη διεύθυνση που ρωτήσατε.** Ο resolver επιστρέφει την πλήρη εγγραφή για τον κανονικό host, που μπορεί να περιέχει επιπλέον διευθύνσεις που τυχαίνει να μοιράζονται το όνομα host.
- **Το βαθμωτό περιβάλλον εκτελεί ούτως ή άλλως την πλήρη κλήση resolver.** Η ρουτίνα C πάντα συμπληρώνει ολόκληρη την hostent· η Perl απλώς απορρίπτει τη λίστα. Δεν υπάρχει πλεονέκτημα απόδοσης στη βαθμωτή μορφή πέραν από τον σαφέστερο κώδικα.
- **Η αντιστροφή μιας μπροστινής αναζήτησης δεν εγγυάται πλήρη κύκλο.** Η gethostbyaddr(gethostbyname($h), AF_INET) μπορεί να επιστρέψει διαφορετικό όνομα από το $h - το αντίστροφο DNS αντιστοιχίζει στο κανονικό όνομα host, που συχνά διαφέρει από το ψευδώνυμο που χρησιμοποίησε η μπροστινή ερώτηση.
- **Η υποστήριξη IPv6 εξαρτάται από την πλατφόρμα.** Δεν χειρίζεται κάθε gethostbyaddr(3) της libc την AF_INET6 καθαρά. Για φορητή αντίστροφη επίλυση διπλής στοίβας χρησιμοποιήστε την Socket::getnameinfo.
- **Ασφάλεια threads.** Η υποκείμενη ρουτίνα C διατηρεί στατική κατάσταση ανά διεργασία για τον δρομέα του host. Ταυτόχρονες κλήσεις από πολλαπλά threads μπορεί να αναμείξουν αποτελέσματα. Σενάρια που χρειάζονται thread-safe αντίστροφη επίλυση πρέπει να χρησιμοποιούν την Socket::getnameinfo.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- gethostbyname - η μπροστινή αναζήτηση: όνομα host → πακεταρισμένη διεύθυνση
- gethostent - επαναλάβετε κάθε εγγραφή στη βάση δεδομένων host αντί να αναζητήσετε μία διεύθυνση
- sethostent - επαναφέρετε στην αρχή τον επαναλήπτη host και ελέγξτε αν το αρχείο παραμένει ανοιχτό μεταξύ αναζητήσεων
- Socket::inet_aton - δημιουργήστε το πακεταρισμένο όρισμα ADDR από συμβολοσειρά τετράδας με τελείες
- Socket::inet_ntoa - αποδώστε μια πακεταρισμένη διεύθυνση από τον @addrs πίσω σε μορφή τετράδας με τελείες

- `Socket::getnameinfo` - σύγχρονος, ανεξάρτητος πρωτοκόλλου αντίστροφος resolver που χειρίζεται IPv6 και είναι thread-safe

Πληροφορίες δικτύου

gethostbyname

Αναζητά μια εγγραφή host με βάση το όνομα DNS.

Η `gethostbyname` επιλύει ένα όνομα host στις διευθύνσεις IP του και στην κανονική εγγραφή καλώντας τη ρουτίνα C `gethostbyname(3)` του συστήματος. Η τυπική περίπτωση χρήσης - *«δώσε μου την πακεταρισμένη IP αυτού του host»* - θέλει τη μορφή βαθμωτού περιβάλλοντος και τίποτε άλλο. Η μορφή περιβάλλοντος λίστας εκθέτει την πλήρη εγγραφή host: κανονικό όνομα, ψευδώνυμα, οικογένεια διευθύνσεων, μήκος διεύθυνσης και κάθε καταχωρισμένη διεύθυνση.

Σύνοψη

```
my $packed_ip = gethostbyname $name;
my ($name, $aliases, $addrtype, $length, @addrs) = gethostbyname
 ↪$name;
```

Τι επιστρέφεται

Βαθμωτό περιβάλλον - την πρώτη διεύθυνση IP ως πακεταρισμένη δυαδική συμβολοσειρά (τέσσερα bytes για IPv4, δεκαέξι για IPv6), ή `undef` αν αποτύχει η αναζήτηση. Μετατρέψτε σε τετράδα με τελείες με `Socket::inet_ntoa`:

```
use Socket;
my $packed = gethostbyname("www.perl.org");
my $dotted = inet_ntoa($packed) if defined $packed;
```

Περιβάλλον λίστας - λίστα πέντε στοιχείων που αντικατοπτρίζει τη `struct hostent` της C:

- `$name` - το κανονικό όνομα host.
- `$aliases` - συμβολοσειρά εναλλακτικών ονομάτων διαχωρισμένη με κενά.
- `$addrtype` - η οικογένεια διευθύνσεων, τυπικά `AF_INET` (2) ή `AF_INET6` (10).
- `$length` - το μήκος κάθε διεύθυνσης σε bytes (4 για IPv4, 16 για IPv6).
- `@addrs` - κάθε διεύθυνση που επιστρέφεται για το όνομα, καθεμία πακεταρισμένη δυαδική συμβολοσειρά μήκους `$length` bytes.

Σε αποτυχία αναζήτησης σε περιβάλλον λίστας, η επιστροφή είναι η κενή λίστα.

Καθολική κατάσταση που επηρεάζει

- `$?` - τίθεται στην τιμή `h_errno` της C σε αποτυχία, όταν η πλατφόρμα την εκθέτει. Ελέγξτε με `$? != 0` για να διακρίνετε το «δεν υπάρχει τέτοιος host» (`HOST_NOT_FOUND`) από το «δοκιμάστε ξανά» (`TRY_AGAIN`) και παρόμοια.

- Ο δρομέας επανάληψης της βάσης δεδομένων host που μοιράζεται με τις `gethostent`, `sethostent` και `endhostent`. Μια κλήση της `gethostbyname` μπορεί να επαναφέρει έναν δρομέα που άφησε ανοιχτό η `gethostent`.

Παραδείγματα

Βαθμωτή μορφή - η συνηθισμένη περίπτωση:

```
use Socket;
my $packed = gethostbyname("localhost");
defined $packed or die "cannot resolve localhost: $!";
print inet_ntoa($packed), "\n";      # 127.0.0.1
```

Μορφή λίστας - απαριθμήστε κάθε διεύθυνση για έναν πολυσυνδεδεμένο host:

```
use Socket;
my ($name, $aliases, $type, $len, @addrs) = gethostbyname("www.
↪example.com");
for my $addr (@addrs) {
    print inet_ntoa($addr), "\n";
}
```

Αποπακετάρετε χειρωνακτικά μια διεύθυνση IPv4, χωρίς το `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
```

Μπροστινή και έπειτα αντίστροφη - επιλύστε ένα όνομα, και έπειτα ψάξτε πίσω τη διεύθυνση:

```
use Socket;
my $packed = gethostbyname("www.perl.org");
my $back   = gethostbyaddr($packed, AF_INET);
print "$back\n";
```

Διεπαφή ανά όνομα μέσω της υπερκάλυψης `Net::hostent` - μέθοδοι προσπέλασης αντί για αποπακετάρισμα κατά θέση:

```
use Net::hostent;
my $h = gethostbyname("www.perl.org");
print $h->name, " ", join(",", @{$h->aliases}), "\n";
```

Οριακές περιπτώσεις

- **Καλέστε πάντα σε βαθμωτό περιβάλλον όταν θέλετε «την IP».** Η μορφή περιβάλλοντος λίστας δεν είναι ποτέ αυτό που θέλει ένας γρήγορος resolver και εκχωρεί τέσσερις επιπλέον τιμές επιστροφής. Τα upstream έγγραφα το επισημαίνουν ρητά: *«Βεβαιωθείτε ότι η* `gethostbyname` *καλείται σε SCALAR περιβάλλον και ότι η τιμή επιστροφής της ελέγχεται για ορισμένο.»*

- **Ελέγξτε για `undef` στη βαθμωτή μορφή.** Μια αποτυχημένη αναζήτηση επιστρέφει `undef`, όχι κενή συμβολοσειρά. Αντιθέτως, μια κυριολεκτική διεύθυνση `0.0.0.0` είναι έγκυρη πακεταρισμένη συμβολοσειρά τεσσάρων bytes και μετατρέπεται σε συμβολοσειρά ως ακολουθία μηδενικών bytes, όχι σε `undef`.
- **Ο `@addrs` μπορεί να είναι κενός** σε περιβάλλον λίστας ακόμη και όταν οι πρώτες τέσσερις τιμές επιστροφής είναι ορισμένες - ορισμένοι resolvers παράγουν εγγραφή χωρίς διευθύνσεις. Επαναλάβετε πάντα, μην ευρετηριάζετε `[0]` τυφλά.
- **Τα ψευδώνυμα είναι ένα βαθμωτό, όχι λίστα.** Το `$aliases` είναι μία συμβολοσειρά διαχωρισμένη με κενά. Διαχωρίστε σε λευκούς χαρακτήρες αν χρειάζεστε λίστα.
- **Οι αριθμητικές συμβολοσειρές IP γίνονται δεκτές.** Το πέρασμα του `"127.0.0.1"` επιστρέφει την πακεταρισμένη μορφή αυτής της διεύθυνσης χωρίς πλήρη κύκλο DNS - ο resolver βραχυκυκλώνει την αριθμητική είσοδο.
- **IPv6.** Η `gethostbyname` ερωτά μόνο την οικογένεια διευθύνσεων IPv4 (`AF_INET`). Για επίλυση διπλής στοίβας χρησιμοποιήστε την `Socket::getaddrinfo`, που επιστρέφει εγγραφές για κάθε οικογένεια διευθύνσεων που προσφέρει ο host.
- **Ασφάλεια threads.** Η υποκείμενη ρουτίνα C διατηρεί στατική κατάσταση ανά διεργασία για τον δρομέα του host. Ταυτόχρονες κλήσεις από πολλαπλά threads μπορεί να αναμείξουν αποτελέσματα. Σενάρια που χρειάζονται thread-safe επίλυση πρέπει να χρησιμοποιούν την `Socket::getaddrinfo`.
- **`h_errno` έναντι `$!`.** Οι αποτυχίες DNS θέτουν το `h_errno` (που εκτίθεται μέσω του `$?`), όχι το `errno` / `$!`. Ο έλεγχος του `$!` μετά από αποτυχημένη `gethostbyname` είναι χωρίς νόημα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `gethostbyaddr` - η αντίστροφη αναζήτηση: πακεταρισμένη διεύθυνση → εγγραφή host
- `gethostent` - επαναλάβετε κάθε εγγραφή στη βάση δεδομένων host αντί να αναζητήσετε ένα όνομα
- `sethostent` - επαναφέρετε στην αρχή τον επαναλήπτη host και ελέγξτε αν το αρχείο παραμένει ανοιχτό μεταξύ αναζητήσεων
- `Socket::inet_ntoa` - μετατρέψτε το πακεταρισμένο αποτέλεσμα βαθμωτού περιβάλλοντος σε συμβολοσειρά τετράδας με τελείες
- `Socket::inet_aton` - ο καθαρός βοηθός ονόματος-προς-πακεταρισμένο όταν δεν χρειάζεστε την πλήρη εγγραφή host
- `unpack` - αποκωδικοποιήστε χειρωνακτικά την πακεταρισμένη διεύθυνση όταν δεν είναι διαθέσιμο το `Socket`

Πληροφορίες δικτύου

gethostent

Ανακτά την επόμενη εγγραφή από τη βάση δεδομένων υπολογιστών.

`gethostent` walks the host database - on traditional Unix systems the `/etc/hosts` file, on modern systems whatever NSS source is configured

- επιστρέφοντας μία εγγραφή ανά κλήση μέχρι να εξαντληθεί η βάση δεδομένων. Χρησιμοποιήστε την όταν θέλετε να απαριθμήσετε κάθε γνωστό υπολογιστή, όχι όταν θέλετε να επιλύσετε ένα μεμονωμένο όνομα: η `gethostbyname` είναι το σωστό εργαλείο για στοχευμένη αναζήτηση.

Ο δρομέας επανάληψης μοιράζεται με τις `sethostent` (που τον επαναφέρει) και `endhostent` (που τον κλείνει). Ανάμεσα σε αυτές τις δύο κλήσεις, η `gethostent` αποδίδει κάθε εγγραφή με τη σειρά και κατόπιν επιστρέφει την κενή λίστα (ή `undef` σε βαθμωτό περιβάλλον) μόλις φτάσει στο τέλος της βάσης δεδομένων.

Σύνοψη

```
my ($name, $aliases, $addrtype, $length, @addrs) = gethostent;
my $name = gethostent;
```

Τι επιστρέφεται

Σε **περιβάλλον λίστας** - μια λίστα πέντε στοιχείων που αντιστοιχεί στο C `struct hostent`:

- `$name` - το κανονικοποιημένο hostname για αυτή την εγγραφή.
- `$aliases` - συμβολοσειρά εναλλακτικών ονομάτων διαχωρισμένων με κενά.
- `$addrtype` - η οικογένεια διευθύνσεων, συνήθως `AF_INET` (2) ή `AF_INET6` (10).
- `$length` - το μήκος σε bytes κάθε διεύθυνσης (4 για IPv4, 16 για IPv6).
- `@addrs` - κάθε διεύθυνση που καταγράφεται για αυτόν τον υπολογιστή, καθεμία πακεταρισμένη δυαδική συμβολοσειρά `$length` bytes.

Στο τέλος της βάσης δεδομένων η επιστροφή είναι η κενή λίστα. Διάκριση που αξίζει να θυμάστε: η σημείωση του upstream perlfunc ότι μια εγγραφή που λείπει μπορεί να επιστρέψει *«a single meaningless true value»* αφορά τις αναζητήσεις κατά όνομα και κατά διεύθυνση, όχι την `gethostent` - η επανάληψη πέρα από το τέλος είναι μια καθαρή κενή λίστα.

Σε **βαθμωτό περιβάλλον** - μόνο το κανονικοποιημένο `$name`, ή `undef` στο τέλος της βάσης δεδομένων. Χρησιμοποιήστε περιβάλλον λίστας όποτε χρειάζεστε οτιδήποτε πέρα από το όνομα· η ανασύνθεση των υπόλοιπων πεδίων εκ των υστέρων δεν είναι δυνατή.

Μετατροπή πακεταρισμένης διεύθυνσης από τον `@addrs` με `Socket::inet_ntoa` για IPv4 ή απευθείας με `unpack`:

```
use Socket;
print inet_ntoa($addrs[0]), "\n";
```

Καθολική κατάσταση που επηρεάζει

- Τον **δρομέα της βάσης δεδομένων υπολογιστών**, που μοιράζεται με τις `sethostent`, `endhostent`, `gethostbyname` και `gethostbyaddr`. Μια αναζήτηση κατά όνομα ή κατά διεύθυνση μπορεί να επαναφέρει ή να καταστήσει άκυρο έναν δρομέα που έχει αφεθεί στο μέσο επανάληψης από την `gethostent`· μη διαπλέκετε τα δύο στυλ χωρίς νέα `sethostent` ανάμεσα.
- `$?` - τίθεται στην τιμή `h_errno` της C σε αποτυχία, όπου την εκθέτει η πλατφόρμα. Ένα απλό τέλος της βάσης δεδομένων δεν είναι αποτυχία και αφήνει την `$?` ανέπαφη.
- `$!` - η `gethostent` είναι επαναλήπτης βάσης δεδομένων, όχι περιτύλιγμα κλήσης συστήματος με την έννοια των `open`/`read`. Τα σφάλματα DNS και NSS εμφανίζονται μέσω του `h_errno` (μέσω της `$?`), όχι μέσω της `$!`.

Παραδείγματα

Απαρίθμηση κάθε υπολογιστή στη βάση δεδομένων:

```
use Socket;
sethostent(1);
while (my ($name, $aliases, $type, $len, @addrs) = gethostent) {
    my @dotted = map { inet_ntoa($_) } @addrs;
    print "$name  @dotted\n";
}
endhostent;
```

Βαθμωτό περιβάλλον - απαρίθμηση μόνο των κανονικοποιημένων ονομάτων:

```
sethostent(0);
while (defined(my $name = gethostent)) {
    print "$name\n";
}
endhostent;
```

Χτίσιμο ενός κατακερματισμού όνομα-προς-διεύθυνση σε ένα πέρασμα, και κατόπιν τοπική αναζήτηση εγγραφών χωρίς περαιτέρω κλήσεις DNS:

```
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";
```

Διεπαφή ανά εγγραφή μέσω της υπερκάλυψης `Net::hostent` - μέθοδοι προσπέλασης αντί για αποπακετάρισμα κατά θέση:

```
use Net::hostent;
while (my $h = gethostent) {
    printf "%-30s %s\n", $h->name, join(",", @{$h->aliases});
}
```

Οριακές περιπτώσεις

- **Δεν υπάρχει έμμεση επαναφορά.** Η `gethostent` δεν ανοίγει ούτε επαναφέρει η ίδια τη βάση δεδομένων. Η πρώτη κλήση μετά την έναρξη του προγράμματος λειτουργεί επειδή η βιβλιοθήκη C ανοίγει νωχελικά τη βάση δεδομένων, αλλά μόλις φτάσετε στο τέλος οι περαιτέρω κλήσεις συνεχίζουν να επιστρέφουν τέλος-βάσης μέχρι μια `sethostent` να επαναφέρει τον δρομέα. Πάντα συνδυάζετε την απαρίθμηση με ρητές `sethostent` / `endhostent`.
- **Όρισμα `STAYOPEN` προς την `sethostent`.** Το πέρασμα αληθούς τιμής ζητά από τη βιβλιοθήκη να κρατήσει το αρχείο της βάσης δεδομένων ανοιχτό μεταξύ αναζητήσεων, πράγμα που έχει σημασία για πηγές NSS που διαφορετικά επανασυνδέονται σε κάθε κλήση. Το πέρασμα `0` επιτρέπει στη βιβλιοθήκη να κλείσει και να ανοίξει εκ νέου μεταξύ κλήσεων.
- **Το τέλος της βάσης δεδομένων είναι η κενή λίστα, όχι το `undef`.** Σε περιβάλλον λίστας η συνθήκη βρόχου `while (my @r = gethostent)` λειτουργεί επειδή η κενή λίστα είναι ψευδής. Σε βαθμωτό περιβάλλον χρησιμοποιήστε `while (defined(my $name = gethostent))` - ένα hostname `"0"` είναι θεωρητικά έγκυρο και θα τερμάτιζε ψευδώς έναν βρόχο βασισμένο σε αλήθεια.
- **Ο `@addrs` μπορεί να είναι κενός** ακόμη και για επιστρεφόμενη εγγραφή σε ορισμένα NSS backends. Προστατέψτε το βήμα αποπακεταρίσματος διεύθυνσης:

  ```
  next unless @addrs;
  ```

- **Τα aliases είναι ένα βαθμωτό, όχι λίστα.** Το `$aliases` είναι μία μεμονωμένη συμβολοσειρά διαχωρισμένη με κενά. Διασπάστε σε λευκούς χαρακτήρες αν θέλετε λίστα.
- **Μείγμα πηγών NSS.** Σε συστήματα βασισμένα σε glibc, η βάση δεδομένων hosts μπορεί να αντλεί από το `/etc/hosts`, το DNS, το mDNS και άλλα αρθρώματα NSS ανάλογα με το `nsswitch.conf`. Τα αρθρώματα NSS με backend DNS συνήθως **δεν** απαριθμούν - η `gethostent` σε βάση δεδομένων hosts μόνο μέσω DNS επιστρέφει την κενή λίστα αμέσως. Στις περισσότερες πραγματικές αναπτύξεις μόνο η πηγή `files` συνεισφέρει εγγραφές στην επανάληψη.
- **Διαπλοκή με αναζητήσεις κατά όνομα / κατά διεύθυνση.** Μια κλήση `gethostbyname` ή `gethostbyaddr` μπορεί να επαναφέρει τον κοινόχρηστο δρομέα. Ολοκληρώστε πρώτα την επανάληψη, ή επανεκκινήστε την με `sethostent` μετά την αναζήτηση.
- **Ασφάλεια threads.** Η υποκείμενη ρουτίνα C διατηρεί στατική κατάσταση ανά διεργασία για τον δρομέα επανάληψης. Ταυτόχρονες κλήσεις από πολλαπλά threads διαπλέκουν τα αποτελέσματα απρόβλεπτα. Περιορίστε την απαρίθμηση σε ένα μόνο thread.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `sethostent` - επαναφέρει τον επαναλήπτη και ελέγχει αν η βάση δεδομένων παραμένει ανοιχτή μεταξύ κλήσεων
- `endhostent` - κλείνει τον επαναλήπτη και απελευθερώνει το handle της βάσης δεδομένων
- `gethostbyname` - στοχευμένη αναζήτηση κατά όνομα αντί για πλήρη απαρίθμηση
- `gethostbyaddr` - στοχευμένη αντίστροφη αναζήτηση κατά πακεταρισμένη διεύθυνση
- `getnetent` - ίδιο μοτίβο επανάληψης για τη βάση δεδομένων δικτύων
- `Socket::inet_ntoa` - μετατρέπει πακεταρισμένη διεύθυνση από τον `@addrs` σε συμβολοσειρά dotted-quad

Πληροφορίες χρηστών και ομάδων

getlogin

Επιστρέφει το όνομα σύνδεσης του χρήστη που συσχετίζεται με το ελέγχον τερματικό.

Η `getlogin` τυλίγει την ομώνυμη συνάρτηση της βιβλιοθήκης C. Σε Linux συμβουλεύεται την εγγραφή του πυρήνα για το ποιος κατέχει το τρέχον ελέγχον τερματικό - παραδοσιακά καταγεγραμμένη στο `/var/run/utmp` - και επιστρέφει αυτό το όνομα ως συμβολοσειρά. Απαντάει στην ερώτηση *«ποιος συνδέθηκε σε αυτό το tty;»*, που **δεν** είναι η ίδια ερώτηση με *«ποιος τρέχει αυτή τη διεργασία;»*. Μια διεργασία που έχει αλλάξει το real ή το effective UID της, ή που δεν έχει καθόλου ελέγχον τερματικό (δαίμονας, cron job, entrypoint σε container), θα δει συνήθως την `getlogin` να επιστρέφει την κενή συμβολοσειρά ή `undef`.

Σύνοψη

```
my $name = getlogin;
```

Τι επιστρέφεται

Μια συμβολοσειρά - το όνομα σύνδεσης που είναι καταγεγραμμένο για το ελέγχον τερματικό - ή την κενή συμβολοσειρά / `undef` όταν δεν υπάρχει διαθέσιμη τέτοια εγγραφή. Αντιμετωπίστε τόσο την κενή συμβολοσειρά όσο και την απροσδιόριστη τιμή ως «άγνωστο» και επιστρέψτε σε αναζήτηση βασισμένη σε UID:

```
my $login = getlogin || getpwuid($<) || "Kilroy";
```

Η χρήση του `||` εδώ είναι σκόπιμη: η κενή συμβολοσειρά και η `undef` σημαίνουν και οι δύο «καμία απάντηση» και και οι δύο πρέπει να ενεργοποιήσουν την εφεδρεία.

Καθολική κατάσταση που επηρεάζει

- Διαβάζει την `$<` έμμεσα μόνο μέσω της ιδιωματικής εφεδρείας `getpwuid` παραπάνω· η ίδια η `getlogin` δεν παίρνει ορίσματα και δεν συμβουλεύεται καμία μεταβλητή της Perl.
- Θέτει την `$!` στις διαδρομές αποτυχίας της υποκείμενης κλήσης συστήματος που εκθέτει η βιβλιοθήκη C.

Παραδείγματα

Το κανονικό μοτίβο ποιος-είναι-αυτός-ο-χρήστης, ανθεκτικό σε αποσπασμένες συνεδρίες και αλλαγές UID:

```
my $login = getlogin || getpwuid($<) || "Kilroy";
print "hello, $login\n";
```

Σκέτη κλήση σε σενάριο προσαρτημένο σε κέλυφος:

```
my $name = getlogin;
print defined $name && length $name
    ? "tty owner: $name\n"
    : "no controlling tty\n";
```

Συγκρίνετε με την `getpwuid`, που απαντάει σε διαφορετική ερώτηση - *«ποιος είναι ο effective χρήστης που τρέχει αυτόν τον κώδικα;»*:

- *«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".
```

Μέσα σε δαίμονα ή σε cron job δεν υπάρχει ελέγχον τερματικό, οπότε η `getlogin` δεν έχει τίποτα να αναφέρει:

```
# running from cron
my $name = getlogin;
# $name is "" or undef; always pair with a getpwuid fallback
```

Οριακές περιπτώσεις

- **Δεν υπάρχει ελέγχον τερματικό**: δαίμονες, υπηρεσίες systemd, cron jobs, entrypoints σε container, και οτιδήποτε ξεκινάει χωρίς tty βλέπουν την κενή συμβολοσειρά ή `undef`. Αυτή είναι η συνηθισμένη περίπτωση εκτός διαδραστικών κελυφών - παρέχετε πάντα εφεδρεία.
- **sudo και su**: η `getlogin` επιστρέφει το **αρχικό** όνομα σύνδεσης (τον ιδιοκτήτη του tty), όχι τον χρήστη-στόχο. Μετά από `sudo -u bob script.pl` παίρνετε το δικό σας login, ενώ η `getpwuid` με `$<` επιστρέφει bob. Διαλέξτε αυτό που ταιριάζει στην ερώτηση που πραγματικά κάνετε.

- **Όχι πρωτογενής λειτουργία ταυτοποίησης**: η εγγραφή που διαβάζει η `getlogin` μπορεί να είναι πλαστογραφημένη ή παλιά· μια διεργασία που κληρονομεί tty από άσχετη συνεδρία μπορεί να δει όνομα που δεν έχει καμία σχέση με τα δικά της διαπιστευτήρια. Ποτέ μη φιλτράρετε προνομιακή συμπεριφορά με βάση την τιμή επιστροφής. Χρησιμοποιήστε την `getpwuid` πάνω στη `$<` / `$>` αντί γι" αυτό.
- **Κενή συμβολοσειρά έναντι `undef`**: οι πλατφόρμες διαφέρουν στο ποιο επιστρέφουν όταν δεν υπάρχει εγγραφή. Η εφεδρεία `||` χειρίζεται και τα δύο· ένας έλεγχος μόνο με `defined` όχι.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `getpwuid` - επιλύει τη `$<` ή τη `$>` σε όνομα, και είναι το σωστό δομικό στοιχείο για το *«ποιος τρέχει αυτή τη διεργασία;»*
- `getpwnam` - η αντίστροφη αναζήτηση όταν έχετε ήδη όνομα σύνδεσης και χρειάζεστε την εγγραφή passwd
- `getpwent` - επαναλάβετε ολόκληρη τη βάση δεδομένων passwd όταν χρειάζεστε πάνω από μία εγγραφή
- `$<` - real UID· το φυσικό όρισμα προς την `getpwuid` στο ιδίωμα εφεδρείας της `getlogin`
- `$>` - effective UID· χρησιμοποιήστε αυτό αντί για `$<` όταν σας ενδιαφέρει η ταυτότητα υπό την οποία θα τρέξουν πραγματικά οι προνομιακές λειτουργίες

Πληροφορίες δικτύου

getnetbyaddr

Αναζητά μια εγγραφή δικτύου με βάση τη αριθμητική διεύθυνση δικτύου του.

Η `getnetbyaddr` μεταφράζει έναν αριθμητικό αριθμό δικτύου και μια οικογένεια διευθύνσεων σε εγγραφή της βάσης δεδομένων δικτύου καλώντας τη ρουτίνα C `getnetbyaddr(3)` του συστήματος. Είναι αντίστροφη της `getnetbyname` και συμβουλεύεται την ίδια βάση δεδομένων δικτύου του συστήματος (τυπικά `/etc/networks` ή ένα backend NSS). Σε βαθμωτό περιβάλλον επιστρέφει μόνο το κανονικό όνομα δικτύου· σε περιβάλλον λίστας επιστρέφει την πλήρη εγγραφή.

Σύνοψη

```
my $name = getnetbyaddr $addr, $addrtype;
my ($name, $aliases, $addrtype, $net) = getnetbyaddr $addr,
 ↪$addrtype;
```

Τι επιστρέφεται

Βαθμωτό περιβάλλον - το κανονικό όνομα του δικτύου, ή undef αν δεν ταιριάζει καμία εγγραφή.

Περιβάλλον λίστας - λίστα τεσσάρων στοιχείων που αντικατοπτρίζει τη struct netent της C:

- $name - το κανονικό όνομα δικτύου.
- $aliases - συμβολοσειρά εναλλακτικών ονομάτων διαχωρισμένη με κενά.
- $addrtype - η οικογένεια διευθύνσεων της εγγραφής, τυπικά AF_INET (2).
- $net - ο αριθμός δικτύου ως μη προσημασμένος ακέραιος, σε host byte order.

Σε αποτυχία αναζήτησης σε περιβάλλον λίστας, η επιστροφή είναι η κενή λίστα. Αν η εγγραφή υπάρχει αλλά είναι κενή, η επιστροφή είναι μία μοναδική χωρίς νόημα αληθής τιμή, σύμφωνα με την κοινή σύμβαση της οικογένειας get*.

Ορίσματα

- ADDR - ο αριθμός δικτύου ως **ακέραιος σε host byte order**, όχι ως πακεταρισμένη συμβολοσειρά. Αυτή είναι η εγγενής ακέραια μορφή που λαμβάνετε πίσω από προηγούμενη κλήση getnetbyname, ή που δημιουργείτε με Socket::inet_aton ακολουθούμενη από unpack("N", ...) για να μετατρέψετε από network σε host order.
- ADDRTYPE - η οικογένεια διευθύνσεων, σχεδόν πάντα AF_INET από το άρθρωμα Socket.

Καθολική κατάσταση που επηρεάζει

- Ο δρομέας επανάληψης της βάσης δεδομένων δικτύου που μοιράζεται με τις getnetent, setnetent και endnetent. Μια κλήση της getnetbyaddr μπορεί να επαναφέρει έναν δρομέα που άφησε ανοιχτό η getnetent.
- $! - η τιμή errno της C σε αποτυχία, όπου η πλατφόρμα την θέτει. Οι περισσότερες αναζητήσεις στο /etc/networks αποτυγχάνουν σιωπηλά με το errno ανέγγιχτο, οπότε αντιμετωπίστε το undef ως το πρωτεύον σήμα αποτυχίας.

Παραδείγματα

Αναζητήστε το δίκτυο loopback με βάση τον ακέραιο αριθμό του:

```
use Socket;
my $name = getnetbyaddr(127 << 24, AF_INET);
print "$name\n";                    # loopback
```

Περιβάλλον λίστας - η πλήρης εγγραφή:

```
use Socket;
my ($name, $aliases, $type, $net)
    = getnetbyaddr(127 << 24, AF_INET);
printf "%s (net=%08x)\n", $name, $net;
```

Πλήρης κύκλος από συμβολοσειρά με τελείες, μέσω του 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);
```

Διεπαφή ανά όνομα μέσω της υπερκάλυψης Net::netent - μέθοδοι προσπέλασης αντί για αποπακετάρισμα κατά θέση:

```
use Net::netent;
my $n = getnetbyaddr(127 << 24, AF_INET);
print $n->name, "\n" if defined $n;
```

Μπροστινή και έπειτα αντίστροφη - επιλύστε ένα όνομα, και έπειτα ψάξτε πίσω τη διεύθυνση:

```
use Socket;
my $net  = (getnetbyname("loopback"))[3];
my $back = getnetbyaddr($net, AF_INET);
print "$back\n";                          # loopback
```

Οριακές περιπτώσεις

- **Ακέραιος, όχι πακεταρισμένη συμβολοσειρά.** Το ADDR είναι απλός ακέραιος σε host byte order. Το πέρασμα εδώ της πακεταρισμένης μορφής τεσσάρων bytes που καταναλώνει η gethostbyaddr θα αναζητήσει σιωπηλά τον λάθος αριθμό. Αυτό είναι το πιο συνηθισμένο λάθος με αυτή τη συνάρτηση.
- **Μόνο κλασικά (classful), στα περισσότερα συστήματα.** Η υποκείμενη βάση δεδομένων /etc/networks προηγείται του CIDR. Οι εγγραφές είναι classful αριθμοί δικτύου (π.χ. 127, 10, 192.168), και η κλήση συστήματος δεν είναι χρήσιμη για αυθαίρετα υποδίκτυα CIDR. Για σύγχρονη συλλογιστική υποδικτύων χρησιμοποιήστε το Socket ή ένα άρθρωμα CPAN όπως το NetAddr::IP.
- **Κενό /etc/networks.** Πολλές διανομές Linux παρέχουν σχεδόν κενό /etc/networks· περιμένετε οι περισσότερες αναζητήσεις να επιστρέφουν undef σε τέτοιον host. Πρόκειται για γεγονός διαμόρφωσης, όχι για σφάλμα στην κλήση.
- **Κενή εγγραφή βάσης δεδομένων σε περιβάλλον λίστας.** Σύμφωνα με τον κοινό κανόνα της οικογένειας get*, μια εγγραφή που υπάρχει αλλά δεν έχει πεδία επιστρέφει μία μοναδική χωρίς νόημα αληθή τιμή, όχι τη λίστα πέντε στοιχείων. Στην πράξη καμία πλατφόρμα δεν παράγει τέτοια εγγραφή για δίκτυα· προφυλαχθείτε με defined και μέτρηση στοιχείων αν σας ενδιαφέρει.
- **ADDRTYPE άλλο από AF_INET.** Η βιβλιοθήκη C δέχεται το όρισμα αλλά δεν έχει καθορισμένη συμπεριφορά για μη-Internet οικογένειες εδώ. Περάστε AF_INET εκτός αν το σύστημα-στόχος τεκμηριώνει διαφορετικά.
- **Ασφάλεια threads.** Η υποκείμενη ρουτίνα C διατηρεί στατική κατάσταση ανά διεργασία για τον δρομέα του δικτύου. Ταυτόχρονες κλήσεις από πολλαπλά threads μπορεί να αναμείξουν αποτελέσματα. Όπου η πλατφόρμα προσφέρει getnetbyaddr_r, η Perl την αντικαθιστά αυτόματα.

- **Προτεραιότητα τελεστή λίστας.** Όπως η υπόλοιπη οικογένεια `get*`, η `getnetbyaddr` αναλύεται ως τελεστής λίστας. Το `getnetbyaddr $n, AF_INET or die` συνδέει το `or die` με ολόκληρη τη λίστα ορισμάτων, που συνήθως είναι αυτό που θέλετε· προσθέστε παρενθέσεις σε περίπτωση αμφιβολίας.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `getnetbyname` - η μπροστινή αναζήτηση: κανονικό όνομα ή ψευδώνυμο → εγγραφή δικτύου
- `getnetent` - επαναλάβετε κάθε εγγραφή στη βάση δεδομένων δικτύου αντί να αναζητήσετε μία διεύθυνση
- `setnetent` - επαναφέρετε στην αρχή τον επαναλήπτη δικτύου και ελέγξτε αν η βάση δεδομένων παραμένει ανοιχτή μεταξύ αναζητήσεων
- `endnetent` - κλείστε τη βάση δεδομένων δικτύου όταν τελειώσετε με την επανάληψη
- `gethostbyaddr` - το αντίστοιχο σε επίπεδο host· δέχεται πακεταρισμένη διεύθυνση, όχι ακέραιο
- `Net::netent` - wrapper ανά όνομα που επιστρέφει αντικείμενο με προσπελάσεις `name` / `aliases` / `addrtype` / `net`

Πληροφορίες δικτύου

getnetbyname

Αναζητεί μια εγγραφή δικτύου με βάση το όνομα.

Η `getnetbyname` επιλύει ένα συμβολικό όνομα δικτύου (όπως παρατίθεται στη βάση δεδομένων δικτύων του συστήματος, τυπικά `/etc/networks`) στην αριθμητική του διεύθυνση δικτύου καλώντας τη ρουτίνα C `getnetbyname(3)`. Η μορφή σε βαθμωτό περιβάλλον επιστρέφει τον αριθμό δικτύου· η μορφή σε περιβάλλον λίστας εκθέτει την πλήρη εγγραφή: κανονικό όνομα, ψευδώνυμα, οικογένεια διευθύνσεων και αριθμό δικτύου.

Σύνοψη

```
my $net = getnetbyname $name;
my ($name, $aliases, $addrtype, $net) = getnetbyname $name;
```

Τι επιστρέφεται

Βαθμωτό περιβάλλον - ο αριθμός δικτύου ως απλός ακέραιος, ή `undef` αν το όνομα δεν είναι στη βάση δεδομένων δικτύων. Πρόκειται για το τμήμα δικτύου σε σειρά bytes host μιας διεύθυνσης IPv4, όχι πακεταρισμένη συμβολοσειρά. Για το δίκτυο κλάσης C `192.168.0`, η επιστρεφόμενη τιμή είναι `12625920` (δηλαδή `0xC0A800`).

Περιβάλλον λίστας - λίστα τεσσάρων στοιχείων που αντικατοπτρίζει το C `struct netent`:

- `$name` - το κανονικό όνομα δικτύου.
- `$aliases` - συμβολοσειρά εναλλακτικών ονομάτων διαχωρισμένων με κενά.
- `$addrtype` - η οικογένεια διευθύνσεων, τυπικά `AF_INET` (2).
- `$net` - ο αριθμός δικτύου, ίδιος ακέραιος με τη μορφή σε βαθμωτό περιβάλλον.

Σε αποτυχία αναζήτησης σε περιβάλλον λίστας, η επιστροφή είναι η κενή λίστα.

Καθολική κατάσταση που επηρεάζει

- Ο δρομέας επανάληψης της βάσης δεδομένων δικτύων που μοιράζεται με τις `getnetent`, `setnetent` και `endnetent`. Μια κλήση `getnetbyname` μπορεί να επαναφέρει έναν δρομέα που είχε αφεθεί ανοιχτός από τη `getnetent`.

Παραδείγματα

Μορφή σε βαθμωτό περιβάλλον - αναζητήστε ένα κατονομασμένο δίκτυο και εκτυπώστε το ως τετράδα με τελείες:

```
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
```

Μορφή λίστας - κάθε πεδίο της εγγραφής:

```
my ($name, $aliases, $type, $net) = getnetbyname("loopback");
print "name=$name aliases=$aliases type=$type net=$net\n";
```

Πλήρης κύκλος έναντι της `getnetbyaddr` - όνομα σε αριθμό, και μετά αριθμός πίσω σε όνομα:

```
use Socket qw(AF_INET);
my $net  = getnetbyname("loopback");
my $back = getnetbyaddr($net, AF_INET);
print "$back\n";                          # loopback
```

Διεπαφή κατά όνομα μέσω της παράκαμψης `Net::netent` - μέθοδοι προσπέλασης αντί για θεσιακό αποπακετάρισμα:

```
use Net::netent;
my $n = getnetbyname("loopback");
print $n->name, " ", $n->net, "\n";
```

Προστασία έναντι απούσας εγγραφής - όνομα που δεν είναι στη βάση δεδομένων επιστρέφει `undef`, όχι μηδέν:

```
my $net = getnetbyname("no-such-network");
if (defined $net) {
    print "found: $net\n";
} else {
    print "not in /etc/networks\n";
}
```

Οριακές περιπτώσεις

- **Ελέγξτε για `undef`, όχι για ψευδότητα.** Αριθμός δικτύου `0` είναι νόμιμη (αν και ασυνήθιστη) εγγραφή· η αντιμετώπιση της τιμής επιστροφής ως λογικής τιμής θα διαβάσει εσφαλμένα αυτή την περίπτωση ως αποτυχία. Πάντα ελέγχετε με `defined`.
- **Ο επιστρεφόμενος αριθμός είναι σε σειρά bytes host, όχι πακεταρισμένη συμβολοσειρά.** Σε αντίθεση με την `gethostbyname`, που επιστρέφει πακεταρισμένη διεύθυνση τεσσάρων bytes, η `getnetbyname` επιστρέφει απλό ακέραιο. Μην τον περάσετε στην `Socket::inet_ntoa` - αυτή η συνάρτηση αναμένει πακεταρισμένη συμβολοσειρά και θα παράγει σκουπίδια από ακέραιο.
- **Τα ψευδώνυμα είναι ένα βαθμωτό, όχι λίστα.** Το `$aliases` στη μορφή λίστας είναι μία ενιαία συμβολοσειρά διαχωρισμένη με κενά. Διασπάστε σε λευκούς χαρακτήρες αν χρειάζεστε λίστα.
- **Networks are typically class-boundary numbers.** The entries in `/etc/networks` are traditionally stored as the network portion only
  - `127` για loopback, `10` για το ιδιωτικό κλάσης A, όχι πλήρεις διευθύνσεις τεσσάρων οκτάδων. Σε σύγχρονα δίκτυα που δρομολογούνται με CIDR η χρησιμότητα αυτής της βάσης δεδομένων είναι περιορισμένη· οι περισσότερες εγκαταστάσεις διατηρούν μόνο λίγες κληροδοτημένες εγγραφές.
- **Καμία εφεδρεία DNS.** Σε αντίθεση με την `gethostbyname`, αυτή η κλήση συμβουλεύεται μόνο την τοπική βάση δεδομένων δικτύων. Δεν υπάρχει ισοδύναμο DNS για ονόματα δικτύων.
- **Κενή λίστα έναντι undef.** Το βαθμωτό περιβάλλον επιστρέφει `undef` σε αποτυχία· το περιβάλλον λίστας επιστρέφει την κενή λίστα. Τα δύο δεν είναι εναλλάξιμα - κώδικας γραμμένος για το ένα περιβάλλον θα συμπεριφερθεί λανθασμένα στο άλλο.
- **Ασφάλεια threads.** Η υποκείμενη ρουτίνα C κρατάει ανά διεργασία στατική κατάσταση για τον δρομέα δικτύων. Ταυτόχρονες κλήσεις από πολλά threads μπορεί να αλληλομπλεχθούν στα αποτελέσματα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `getnetbyaddr` - η αντίστροφη αναζήτηση: αριθμός δικτύου → εγγραφή δικτύου
- `getnetent` - επανάληψη σε κάθε εγγραφή της βάσης δεδομένων δικτύων αντί για αναζήτηση ενός ονόματος
- `setnetent` - επαναφορά του επαναλήπτη δικτύων στην αρχή και έλεγχος του αν το αρχείο παραμένει ανοιχτό μεταξύ αναζητήσεων
- `endnetent` - κλείνει τη βάση δεδομένων δικτύων μετά από επανάληψη
- `gethostbyname` - ο αντίστοιχος σε επίπεδο υπολογιστή, για επίλυση ονόματος host σε διεύθυνση IP
- `Net::netent` - αντικειμενοστραφές περιτύλιγμα κατά όνομα που επιστρέφει μεθόδους προσπέλασης αντί για θεσιακή λίστα

Πληροφορίες δικτύου

getnetent

Διαβάζει την επόμενη εγγραφή από τη βάση δεδομένων δικτύων.

Η `getnetent` περιβάλλει την ομώνυμη ρουτίνα της βιβλιοθήκης C. Διατρέχει τη βάση δεδομένων δικτύων του συστήματος (συνήθως το `/etc/networks`, ενίοτε ενισχυμένη από NIS ή άλλα backend αναλυτών) μία εγγραφή τη φορά, επιστρέφοντας την επόμενη εγγραφή σε κάθε κλήση. Συνδυάστε την με την `setnetent` για επαναφορά ή έλεγχο της σύνδεσης, και με την `endnetent` για κλείσιμό της.

Σύνοψη

```
my ($name, $aliases, $addrtype, $net) = getnetent;
my $name                              = getnetent;   # scalar␣
↪context
```

Τι επιστρέφεται

Σε περιβάλλον λίστας, μια λίστα τεσσάρων στοιχείων:

- `$name` - κανονικό όνομα δικτύου (π.χ. `"loopback"`).
- `$aliases` - συμβολοσειρά εναλλακτικών ονομάτων διαχωρισμένων με κενά.
- `$addrtype` - οικογένεια διευθύνσεων, τυπικά `AF_INET` (2).
- `$net` - αριθμός δικτύου ως ακέραιος σε σειρά bytes του υπολογιστή. Μετατροπή σε μορφή τελειών με `Socket::inet_ntoa(pack("N", $net))`.

Σε βαθμωτό περιβάλλον, μόνο το `$name`.

Όταν ο επαναλήπτης εξαντληθεί (δεν υπάρχουν άλλες εγγραφές), το περιβάλλον λίστας επιστρέφει την κενή λίστα και το βαθμωτό περιβάλλον επιστρέφει `undef`. Μια ελλείπουσα εγγραφή κατά την άμεση αναζήτηση επιστρέφει μία χωρίς νόημα αληθή τιμή σύμφωνα με τη σύμβαση perlfunc, αλλά δεδομένου ότι η `getnetent` δεν δέχεται κλειδί, η μόνη τερματική συνθήκη είναι «τέλος της βάσης».

Καθολική κατάσταση που επηρεάζει

Η getnetent διαβάζει από έναν **επαναλήπτη καθολικό σε επίπεδο διεργασίας** που διατηρείται από τη βιβλιοθήκη C. Κάθε κλήση προχωρά αυτόν τον επαναλήπτη· δεν υπάρχει κατάσταση ανά εμβέλεια, και ταυτόχρονη επανάληψη από δύο σημεία της ίδιας διεργασίας θα διαπλέκει εγγραφές. Η setnetent τον επαναφέρει και προαιρετικά ζητά από τη βιβλιοθήκη να διατηρήσει ανοιχτή την υποκείμενη σύνδεση για όλη τη διάρκεια της επανάληψης· η endnetent τον κλείνει.

Παραδείγματα

Διάσχιση ολόκληρης της βάσης δεδομένων δικτύων:

```
setnetent(1);
while (my ($name, $aliases, $addrtype, $net) = getnetent) {
    printf "%-20s %s\n", $name, join(" ", split / /, $aliases);
}
endnetent();
```

Το βαθμωτό περιβάλλον δίνει μόνο το όνομα - χρήσιμο για γρήγορη απαρίθμηση:

```
setnetent(1);
while (defined(my $name = getnetent)) {
    print "$name\n";
}
endnetent();
```

Αποκωδικοποίηση του αριθμού δικτύου σε μορφή τελειών:

```
use Socket;
setnetent(1);
while (my ($name, undef, undef, $net) = getnetent) {
    printf "%-20s %s\n", $name, inet_ntoa(pack "N", $net);
}
endnetent();
```

Αντικειμενοστραφής πρόσβαση μέσω Net::netent - η διεπαφή ανά όνομα από την standard library αντικαθιστά τη λίστα επιστροφής κατά θέση με ένα blessed αντικείμενο:

```
use Net::netent;
while (my $n = getnetent()) {
    printf "%s => %d\n", $n->name, $n->net;
}
```

Οριακές περιπτώσεις

- **Εξάντληση επαναλήπτη**: το περιβάλλον λίστας επιστρέφει (), το βαθμωτό περιβάλλον επιστρέφει undef. Οι μορφές βρόχου παραπάνω τερματίζονται φυσιολογικά.

- **Χωρίς setnetent πριν τον βρόχο**: η πρώτη κλήση ανοίγει σιωπηρά τη βάση. Η κλήση της `setnetent(1)` εξακολουθεί να συνιστάται - χωρίς τη σημαία «παραμονή ανοιχτή», ορισμένες βιβλιοθήκες C ξανανοίγουν το αρχείο σε κάθε εγγραφή και η απόδοση καταρρέει.
- **Πολλαπλές ταυτόχρονες διασχίσεις**: ο επαναλήπτης είναι καθολικός σε επίπεδο διεργασίας. Δύο βρόχοι επί της `getnetent` στην ίδια διεργασία θα μοιραστούν τις εγγραφές μεταξύ τους. Συλλέξτε σε πίνακα πρώτα αν και οι δύο χρειάζονται το πλήρες σύνολο.
- **Μη υποστηριζόμενο σε ορισμένες πλατφόρμες**: τα Windows δεν έχουν αντίστοιχο `/etc/networks` και η υποκείμενη ρουτίνα C απουσιάζει. Η προσπάθεια κλήσης της `getnetent` εκεί εγείρει `The getnetent function is unimplemented`. Στο Linux, η glibc παρέχει τη συνάρτηση αλλά η βάση είναι συχνά κενή - τα περισσότερα συστήματα βασίζονται σε DNS αντ' αυτού, το οποίο αυτή η διεπαφή δεν συμβουλεύεται.
- **Ασφάλεια νημάτων**: ο επαναλήπτης της βιβλιοθήκης C δεν είναι ασφαλής για νήματα. Η Perl ενδέχεται να αντικαταστήσει διαφανώς με `getnetent_r` σε συστήματα που την παρέχουν, αλλά μην υποθέσετε ότι αυτό είναι καθολικό. Σε πρόγραμμα με νήματα, σειριοποιήστε την πρόσβαση μέσω ενός μόνο νήματος.
- **Tainting**: όταν είναι ενεργοποιημένο το tainting, και τα τέσσερα πεδία επιστροφής είναι tainted - προέρχονται από αρχείο (ή απομακρυσμένο αναλυτή) εκτός του ελέγχου του προγράμματος.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `setnetent` - επαναφέρει τον επαναλήπτη και προαιρετικά κρατά την υποκείμενη σύνδεση ανοιχτή σε όλες τις κλήσεις
- `endnetent` - κλείνει τον επαναλήπτη όταν τελειώσετε
- `getnetbyname` - αναζήτηση ενός δικτύου με όνομα αντί για διάσχιση ολόκληρης της βάσης
- `getnetbyaddr` - αναζήτηση ενός δικτύου με τη διεύθυνσή του
- `gethostent` - η ανάλογη διάσχιση της βάσης δεδομένων υπολογιστών, με το ίδιο συμπληρωματικό ζεύγος `set`/`end`
- `Net::netent` - αντικειμενική διεπαφή ανά όνομα που αντικαθιστά τη λίστα επιστροφής κατά θέση

Υποδοχές (sockets)

getpeername

Επιστρέφει τη διεύθυνση του απομακρυσμένου άκρου μιας συνδεδεμένης υποδοχής.

Η `getpeername` ρωτάει τον πυρήνα ποιος είναι στην άλλη πλευρά του `SOCKET` και επιστρέφει την απάντηση ως πακεταρισμένη συμβολοσειρά bytes `sockaddr` -

του ίδιου σχήματος που πήρε η `connect` και του ίδιου σχήματος που επέστρεψε η `accept`. Στη συνέχεια δίνετε αυτή τη συμβολοσειρά bytes σε έναν αποπακεταριστή από την `Socket` (ή την `Socket::unpack_sockaddr_in`, `Socket::unpack_sockaddr_in6`, `Socket::unpack_sockaddr_un`) για να εξαγάγετε τη θύρα, τη διεύθυνση ή τη διαδρομή.

Είναι η συνοδευτική της `getsockname`, η οποία επιστρέφει το **τοπικό** άκρο της ίδιας υποδοχής.

Σύνοψη

```
use Socket;
my $peer = getpeername SOCKET;
my ($port, $iaddr) = unpack_sockaddr_in($peer);
```

Τι επιστρέφεται

Μια πακεταρισμένη συμβολοσειρά bytes `sockaddr` σε επιτυχία, `undef` σε αποτυχία (με την `$!` τεθειμένη - τυπικά `ENOTCONN` όταν η υποδοχή δεν έχει ομότιμο, ή `EBADF` όταν το `SOCKET` δεν είναι ανοιχτός περιγραφέας αρχείου).

Η συμβολοσειρά bytes είναι αδιαφανής. Η διάταξή της εξαρτάται από την οικογένεια διευθύνσεων της υποδοχής· πάντα αποκωδικοποιήστε την με αποπακεταριστή που ταιριάζει με την οικογένεια που αναμένετε:

- `AF_INET` → η `Socket::unpack_sockaddr_in` επιστρέφει `($port, $iaddr)` όπου το `$iaddr` είναι η πακεταρισμένη διεύθυνση IPv4 4 bytes.
- `AF_INET6` → η `Socket::unpack_sockaddr_in6` επιστρέφει `($port, $iaddr, $scope_id, $flowinfo)`.
- `AF_UNIX` → η `Socket::unpack_sockaddr_un` επιστρέφει τη διαδρομή του ομότιμου στο σύστημα αρχείων.

Αν δεν γνωρίζετε την οικογένεια στο σημείο κλήσης, ρωτήστε πρώτα την `Socket::sockaddr_family`:

```
my $peer   = getpeername $sock;
my $family = sockaddr_family($peer);
```

Καθολική κατάσταση που επηρεάζει

Η `getpeername` διαβάζει την κατάσταση περιγραφέα αρχείου του `SOCKET`. Θέτει την `$!` σε αποτυχία και κατά τα άλλα δεν αγγίζει καθολικές μεταβλητές του διερμηνέα. Δεν αλληλεπιδρά με την `$_`, το επιλεγμένο filehandle ή τους διαχωριστές εξόδου.

Παραδείγματα

Αναζήτηση του απομακρυσμένου ομότιμου IPv4 μιας υποδοχής TCP από την πλευρά πελάτη και απόδοσή του σε μορφή τετράδας με τελείες:

```
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;
```

Από την πλευρά του διακομιστή: κάθε υποδοχή που επιστρέφει η accept είναι ήδη συνδεδεμένη, και η ίδια η accept επιστρέφει τη διεύθυνση του ομότιμου. Η getpeername είναι χρήσιμη αργότερα, όταν εκείνη η πακεταρισμένη διεύθυνση δεν έχει διατηρηθεί:

```
while (my $client = accept(my $conn, $listen)) {
    handle($conn);
}

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";
    # ...
}
```

Επίλυση του ομότιμου πίσω σε όνομα host - σημειώστε ότι αυτό εκδίδει μια μπλοκαριστική αναζήτηση DNS:

```
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";
```

Μεικτός κώδικας IPv4 / IPv6 - αποκωδικοποιήστε γενικά μέσω της οικογένειας διευθύνσεων:

```
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;
}
```

Δοκιμή για το αν μια υποδοχή είναι καν συνδεδεμένη - η διαδρομή ENOTCONN είναι ο καλά ορισμένος τρόπος να ρωτήσετε:

```
if (defined getpeername($sock)) {
    # connected
} else {
    # $! == ENOTCONN for an unconnected socket
}
```

Οριακές περιπτώσεις

- **Μη συνδεδεμένη υποδοχή**: μια υποδοχή που έχει περάσει από `socket` αλλά όχι ακόμη από `connect` ή `accept` δεν έχει ομότιμο. Η `getpeername` επιστρέφει `undef` και θέτει την `$!` σε `ENOTCONN`.
- **Filehandle που δεν είναι υποδοχή**: το πέρασμα ενός απλού αρχείου ή σωλήνα αποδίδει `undef` με την `$!` τεθειμένη σε `ENOTSOCK`.
- **Κλεισμένο ή άκυρο filehandle**: επιστρέφει `undef` με την `$!` τεθειμένη σε `EBADF`. Υπό `use warnings` εκπέμπεται προειδοποίηση `getpeername() on closed socket`.
- **Υποδοχές datagram**: η `getpeername` επιστρέφει διεύθυνση μόνο όταν η υποδοχή `SOCK_DGRAM` έχει περάσει από `connect` σε προεπιλεγμένο ομότιμο· διαφορετικά αποτυγχάνει με `ENOTCONN` παρότι η υποδοχή είναι χρησιμοποιήσιμη μέσω `recv` / `send`.
- **`AF_UNIX` με μη δεσμευμένο ομότιμο**: μια υποδοχή τομέα Unix της οποίας ο ομότιμος δεν κάλεσε ποτέ `bind` επιστρέφει πακεταρισμένο `sockaddr_un` με κενή διαδρομή. Η `Socket::unpack_sockaddr_un` επιστρέφει την κενή συμβολοσειρά.
- **Μετά από `shutdown`**: η `getpeername` εξακολουθεί να πετυχαίνει εφόσον ο περιγραφέας είναι ανοιχτός· το shutdown καταρρίπτει τη ροή δεδομένων, όχι την καταγραφή του ομότιμου από τον πυρήνα.
- **Το SOCKET είναι διακριτικό filehandle που αναλύεται συντακτικά**: όπως και άλλες ενσωματωμένες συναρτήσεις υποδοχών, το όρισμα αναλύεται ως filehandle, όχι ως γενική έκφραση. Η `getpeername $handles[0]` είναι συντακτικό σφάλμα· χρησιμοποιήστε ένα βαθμωτό που κρατάει το handle, ή αποαναφορά `*{...}`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `getsockname` - κατοπτρική κλήση που επιστρέφει το **τοπικό** άκρο της ίδιας υποδοχής· συνδυάζεται με την `getpeername` όταν καταγράφονται και τα δύο άκρα μιας σύνδεσης
- `accept` - σας επιστρέφει ήδη τη διεύθυνση του ομότιμου για μια φρέσκα αποδεκτή υποδοχή από την πλευρά διακομιστή· χρησιμοποιήστε την `getpeername` αργότερα όταν αυτή η διεύθυνση δεν έχει διατηρηθεί
- `connect` - η κλήση που εγκαθιδρύει εξαρχής τον ομότιμο· η διεύθυνση που περνάτε στην `connect` είναι αυτή που θα επιστρέψει η `getpeername` στη συνέχεια

- `Socket` - παρέχει τους αποπακεταριστές (`unpack_sockaddr_in`, `unpack_sockaddr_in6`, `unpack_sockaddr_un`) και την `sockaddr_family` που χρειάζονται για την ερμηνεία της επιστρεφόμενης συμβολοσειράς bytes
- `gethostbyaddr` - μετατρέπει την πακεταρισμένη διεύθυνση IPv4 από τον αποπακεταριστή σε όνομα host
- `$!` - μεταφέρει τον λόγο σε αποτυχία (`ENOTCONN`, `ENOTSOCK`, `EBADF`)

Διεργασίες

getpgrp

Επιστρέφει το POSIX ID ομάδας διεργασιών στο οποίο ανήκει μια διεργασία.

Κάθε διεργασία Unix είναι μέλος ακριβώς μίας ομάδας διεργασιών, η οποία με τη σειρά της ανήκει σε μια συνεδρία. Οι ομάδες διεργασιών είναι ο τρόπος με τον οποίο ο πυρήνας απευθύνεται σε σχετιζόμενες διεργασίες ως μονάδα - πιο εμφανώς, είναι η μονάδα που λαμβάνει ένα σήμα από το ελέγχον τερματικό (`SIGINT` σε Ctrl-C, `SIGTSTP` σε Ctrl-Z). Η `getpgrp` ρωτάει τον πυρήνα για το ID ομάδας μιας δεδομένης διεργασίας· χωρίς όρισμα ή με `PID` ίσο με `0` απαντάει για την τρέχουσα διεργασία.

Σύνοψη

```
getpgrp
getpgrp PID
getpgrp 0
```

Τι επιστρέφεται

Έναν μη αρνητικό ακέραιο - το ID ομάδας διεργασιών σε βαθμωτό περιβάλλον. Τα ID ομάδας διεργασιών μοιράζονται τον χώρο αρίθμησης PID, και το PID του αρχηγού ομάδας ισούται με το ID ομάδας, οπότε η τιμή που επιστρέφει η `getpgrp 0` είναι το PID του αρχηγού ομάδας της τρέχουσας διεργασίας.

Σε σύστημα που δεν υλοποιεί καθόλου την `getpgrp(2)`, η κλήση εγείρει μοιραία εξαίρεση. Αυτό είναι εξαιρετικά σπάνιο σε σύγχρονα Unix· τα Linux, BSD, macOS, και Solaris την υποστηρίζουν όλα.

Καθολική κατάσταση που επηρεάζει

Καμία. Η `getpgrp` διαβάζει κατάσταση πυρήνα για την κατονομασμένη διεργασία και θέτει την `$!` μόνο σε αποτυχία (που, σε σύστημα POSIX, σημαίνει «no such process» για μη μηδενικό `PID` που δεν υπάρχει).

Παραδείγματα

Λάβετε τη δική της ομάδα της τρέχουσας διεργασίας:

```
my $pgid = getpgrp;
print "my group is $pgid\n";
```

PID ίσο με `0` σημαίνει «εγώ ο ίδιος» και είναι η φορητή μορφή - ορισμένα πολύ παλιά συστήματα δεν δέχονται καμία άλλη τιμή:

```
my $pgid = getpgrp 0;                    # same as getpgrp with no args
```

Ανίχνευση του αν η τρέχουσα διεργασία είναι ο δικός της αρχηγός ομάδας - ο συνηθισμένος έλεγχος που τρέχει ένας δαίμονας πριν καλέσει `setpgrp` ή `POSIX::setsid` για να αποσυνδεθεί από το ελέγχον τερματικό του:

```
if (getpgrp(0) == $$) {
    print "already a group leader\n";
}
else {
    print "group leader is ", getpgrp(0), "\n";
}
```

Αναζητήστε την ομάδα μιας άλλης διεργασίας, π.χ. ενός παιδιού που μόλις επέστρεψε η `fork`:

```
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    # child
    exec @cmd;
}
print "child $pid is in group ", getpgrp($pid), "\n";
```

Στείλτε σήμα σε κάθε μέλος μιας ομάδας διεργασιών. Η `kill` με αρνητικό PID στοχεύει την ομάδα, και η `getpgrp` είναι ο τρόπος με τον οποίο ανακαλύπτετε το ID ομάδας προς αρνητικοποίηση:

```
my $grp = getpgrp($child);
kill 'TERM', -$grp;                      # SIGTERM to the whole group
```

Οριακές περιπτώσεις

- **Παραλειπόμενο όρισμα ισούται με PID ίσο με `0`**: τόσο η `getpgrp` όσο και η `getpgrp 0` αναφέρουν την ομάδα της τρέχουσας διεργασίας. Προτιμήστε τη ρητή μορφή `0` όταν έχει σημασία η φορητότητα σε ιστορικά συστήματα.
- **Ανύπαρκτο PID**: επιστρέφει `undef` και θέτει την `$!` σε `"No such process"` (`ESRCH`). Ελέγξτε την `$!`, όχι την αληθότητα της τιμής επιστροφής - ένα νόμιμο ID ομάδας ίσο με `0` είναι ψευδές στην Perl αλλά δεν είναι κατάσταση σφάλματος (είναι, σε Linux, η ομάδα idle/swapper).
- **Μη επιτρεπτές αναζητήσεις**: σε συστήματα που περιορίζουν τις αναζητήσεις μεταξύ συνεδριών, η ερώτηση ενός PID σε διαφορετική συνεδρία μπορεί να αποτύχει με την `$!` τεθειμένη σε `"Operation not permitted"` (`EPERM`). Αυτό είναι ασυνήθιστο σε Linux, όπου οποιαδήποτε διεργασία μπορεί να ρωτήσει την ομάδα οποιουδήποτε PID.
- **`getpgrp` έναντι `getpgrp PID`**: η BSD-στυλ μορφή ενός ορίσματος δεν ήταν ιστορικά φορητή σε όλα τα Unix· το POSIX τυποποίησε και τις δύο. Στις πλατφόρμες όπου τρέχει η pperl (Linux, όλοι οι στόχοι cranelift), και οι δύο μορφές λειτουργούν.

- **Προσημασμένος έναντι μη προσημασμένου**: ο τύπος PID του πυρήνα είναι προσημασμένος, οπότε η επιστρεφόμενη τιμή χωράει άνετα σε ακέραιο της Perl. Επιστροφή `-1` δεν χρησιμοποιείται για επιτυχία σε καμία πλατφόρμα-στόχο της pperl· αντιμετωπίστε το `-1` ως φύλακα σφάλματος μόνο αν μεταφέρετε κώδικα από C codebase που το έλεγχε έτσι.
- **Δεν είναι shell-ψευδώνυμο για έλεγχο εργασιών**: οι σημειογραφίες `jobs` και `%` του κελύφους λειτουργούν πάνω σε πίνακες εργασιών που κρατάει το κέλυφος, όχι απευθείας πάνω σε ομάδες διεργασιών του πυρήνα. Η `getpgrp` σας λέει την αλήθεια του πυρήνα, που μπορεί να αποκλίνει από την ιδέα ενός υποκελύφους για το δέντρο εργασιών.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `setpgrp` - τοποθετήστε μια διεργασία σε νέα ή υπάρχουσα ομάδα· το εγγραφικό αντίστοιχο της `getpgrp`
- `getppid` - ID γονικής διεργασίας· συχνά χρειάζεται μαζί με το ID ομάδας όταν διασχίζετε σχέσεις διεργασιών
- `getpriority` - διαβάστε την προτεραιότητα χρονοπρογραμματισμού, που δέχεται επίσης επιλογέα ομάδας διεργασιών (`PRIO_PGRP`)
- `fork` - το παιδί κληρονομεί την ομάδα διεργασιών του γονέα του, οπότε η `getpgrp` σε ένα φρέσκο παιδί αναφέρει την ομάδα του γονέα μέχρι μια κλήση `setpgrp` να την αλλάξει
- `kill` - το `kill SIGNAL, -$pgid` στέλνει σήμα σε κάθε μέλος της ομάδας· η `getpgrp` είναι ο τρόπος με τον οποίο βρίσκετε το `$pgid`
- `$$` - τρέχον PID· ίσο με `getpgrp(0)` όταν η διεργασία είναι ο δικός της αρχηγός ομάδας

Διεργασίες

getppid

Επιστρέφει το ID διεργασίας του γονέα της τρέχουσας διεργασίας.

Η `getppid` είναι ένας λεπτός περιτυλιγμός πάνω από την POSIX κλήση συστήματος `getppid(2)`. Δεν παίρνει ορίσματα και επιστρέφει έναν ακέραιο - το PID οποιασδήποτε διεργασίας έκανε `fork` (ή `exec` προς) στο εκτελούμενο πρόγραμμα. Σε συνδυασμό με την `$$` (το PID της ίδιας της τρέχουσας διεργασίας), είναι ο τυπικός τρόπος για ένα παιδί να αναγνωρίσει τη διεργασία που το δημιούργησε.

Σύνοψη

```
my $parent = getppid;
```

Τι επιστρέφεται

Ένας θετικός ακέραιος - το PID της γονικής διεργασίας. Σε Linux αυτό είναι 1 (init / systemd) μόλις ο αρχικός γονέας έχει εξέλθει και το παιδί έχει επανατεθεί σε νέο γονέα· δεν υπάρχει ειδική τιμή επιστροφής για «ορφανό», η αλλαγή γονέα είναι διαφανής. Δεν επιστρέφει ποτέ `undef` ή `0` σε υποστηριζόμενη πλατφόρμα· η υποκείμενη κλήση συστήματος προδιαγράφεται να πετυχαίνει πάντα.

Καθολική κατάσταση που επηρεάζει

Καμία. Η `getppid` διαβάζει τον πίνακα διεργασιών του πυρήνα για την τρέχουσα διεργασία· δεν αγγίζει καμία ειδική μεταβλητή της Perl. Σημειώστε ότι η `$$` κρατάει το PID *αυτής* της διεργασίας, όχι του γονέα - ανακτήστε τον γονέα με την `getppid` ρητά.

Παραδείγματα

Αναγνωρίστε τον γονέα που εκκίνησε αυτό το σενάριο:

```
printf "my pid is %d, my parent is %d\n", $$, getppid;
```

Ανίχνευση ορφάνεψης - συνηθισμένο ιδίωμα για δαίμονες που θέλουν να εξέλθουν όταν η ελέγχουσα διεργασία τους φύγει:

```
while (1) {
    last if getppid == 1;       # parent died, we've been reparented
 ↪to init
    do_work();
    sleep 1;
}
```

Διεργασία-παιδί αναφέρει πίσω στον γονέα της μετά από `fork`:

```
my $pid = fork // die "fork failed: $!";
if ($pid == 0) {
    warn "child $$ spawned by $>, parent is " . getppid . "\n";
    exit 0;
}
waitpid $pid, 0;
```

Συγκρίνετε πριν και μετά τη `fork` για να επιβεβαιώσετε τη σχέση:

```
my $before = $$;
my $pid = fork // die $!;
if ($pid == 0) {
    die "parent mismatch" unless getppid == $before;
    exit;
}
waitpid $pid, 0;
```

Οριακές περιπτώσεις

- **Κανένα όρισμα, καμία ανάγκη παρενθέσεων**: η `getppid` είναι μηδενικής πληθικότητας ενσωματωμένη. Γράψτε `getppid` ή `getppid()`· και τα δύο αναλύονται πανομοιότυπα. Μην προσπαθήσετε να περάσετε όρισμα PID - σε αντίθεση με την `getpgrp`, δεν υπάρχει μορφή «γονέας άλλης διεργασίας».
- **Η επαναγονεοποίηση ορφανών είναι αόρατη**: αν ο αρχικός γονέας εξέλθει, ο πυρήνας θέτει νέο γονέα στη διεργασία (την `init`/`systemd` σε Linux, ή τον πλησιέστερο πρόγονο με `PR_SET_CHILD_SUBREAPER` ενεργοποιημένο). Η `getppid` αρχίζει σιωπηλά να επιστρέφει το PID του νέου γονέα. Κώδικας που προσωρινά αποθηκεύει την τιμή επιστροφής κατά την εκκίνηση και τη συγκρίνει αργότερα είναι ο τυπικός τρόπος ανίχνευσης αυτού.
- **Race στην εκκίνηση**: η κλήση της `getppid` εξαιρετικά νωρίς, πριν ο πυρήνας ολοκληρώσει τη ρύθμιση της διεργασίας, δεν αποτελεί ανησυχία σε κώδικα χρήστη της Perl - μέχρι να τρέξει ο διερμηνέας, το γονικό PID είναι σταθερό.
- **Αναδίπλωση PID**: τα PID ανακυκλώνονται. Ένα αποθηκευμένο γονικό PID μπορεί, σε βάθος μακροχρόνιας λειτουργίας, να αναφέρεται σε *διαφορετική* διεργασία από τον αρχικό γονέα ακόμη και χωρίς αλλαγή γονέα. Χρησιμοποιήστε `getppid == 1` (ή το αντίστοιχο PID subreaper), όχι ισότητα με μια αποθηκευμένη τιμή, όταν η πρόθεση είναι «έχει πεθάνει ο γονέας μου;».
- **Threads**: από την Perl 5.16 ο workaround LinuxThreads για Linux έχει αφαιρεθεί. Η `getppid` τώρα επιστρέφει πάντα ό,τι αναφέρει ο πυρήνας για την τρέχουσα ομάδα thread.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `$$` - το PID της ίδιας της τρέχουσας διεργασίας· συνδυάστε με `getppid` για να καταγράψετε και τα δύο άκρα της σχέσης γονέα/παιδιού
- `fork` - δημιουργία διεργασίας-παιδιού· η `getppid` του παιδιού επιστρέφει το PID της διεργασίας που έκανε fork
- `waitpid` - συλλέγει συγκεκριμένο παιδί με PID· η κατοπτρική λειτουργία από την πλευρά του γονέα
- `wait` - συλλέγει οποιοδήποτε παιδί· ίδια λειτουργία από την πλευρά του γονέα, χωρίς προσδιορισμένο PID
- `getpgrp` - ID *ομάδας* διεργασιών αντί για γονικό PID· συγγενής αλλά διακριτή έννοια του πυρήνα
- `kill` - αποστέλλει σήμα σε PID, συμπεριλαμβανομένου ενός που λήφθηκε από `getppid` (π.χ. ειδοποίηση του γονέα)

Διεργασίες

getpriority

Επιστρέφει την τρέχουσα τιμή nice χρονοπρογραμματισμού μιας διεργασίας, ομάδας διεργασιών ή χρήστη.

Η `getpriority` είναι λεπτό περιτύλιγμα γύρω από την κλήση συστήματος `getpriority(2)`. Οι τιμές nice κυμαίνονται από `-20` (η πιο ευνοϊκή προς τη διεργασία) έως 19 (η λιγότερο ευνοϊκή)· μια φρέσκα δημιουργημένη διεργασία κληρονομεί την τιμή του γονέα της, τυπικά `0`. Μόνο ο υπερχρήστης μπορεί να *χαμηλώσει* μια τιμή nice· κάθε χρήστης μπορεί να αυξήσει τη δική του.

Σύνοψη

```
getpriority WHICH, WHO
```

Το `WHICH` επιλέγει τι σημαίνει το `WHO`:

- `PRIO_PROCESS` - το `WHO` είναι ID διεργασίας (`0` σημαίνει την τρέχουσα διεργασία).
- `PRIO_PGRP` - το `WHO` είναι ID ομάδας διεργασιών (`0` σημαίνει την τρέχουσα ομάδα).
- `PRIO_USER` - το `WHO` είναι αριθμητικό ID χρήστη (`0` σημαίνει το πραγματικό UID της καλούσας διεργασίας).

Οι τρεις σταθερές προέρχονται από την `POSIX`· δεν είναι ενσωματωμένες και δεν εξάγονται προεπιλεγμένα.

```
use POSIX qw(PRIO_PROCESS PRIO_PGRP PRIO_USER);
```

Τι επιστρέφεται

Η τρέχουσα τιμή nice ως ακέραιος στο εύρος `-20 .. 19`.

Δεν υπάρχει διακριτή επιστροφή σφάλματος: το `-1` είναι απολύτως έγκυρη τιμή nice. Για να εντοπίσετε αποτυχία, καθαρίστε την `$!` πριν την κλήση και επιθεωρήστε την μετά:

```
$! = 0;
my $nice = getpriority PRIO_PROCESS, $pid;
die "getpriority: $!" if $!;
```

Σε πυρήνα που υλοποιεί την `getpriority(2)` αλλά επιστρέφει σφάλμα (τυπικά `ESRCH` για ανύπαρκτο στόχο ή `EINVAL` για κίβδηλο `WHICH`), η συνάρτηση επιστρέφει `-1` με την `$!` τεθειμένη. Σε πλατφόρμα που δεν υλοποιεί καθόλου την κλήση, η `getpriority` εγείρει μοιραία εξαίρεση· πιάστε την με `eval` αν το σενάριο πρέπει να τρέχει σε τέτοιο σύστημα.

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται όταν η υποκείμενη κλήση `getpriority(2)` αποτυγχάνει. Πρέπει να καθαριστεί πριν την κλήση για να ξεχωρίσει νόμιμο αποτέλεσμα `-1` από σφάλμα.

Παραδείγματα

Διαβάστε τη δική σας τιμή nice για την τρέχουσα διεργασία:

```
use POSIX qw(PRIO_PROCESS);
my $nice = getpriority PRIO_PROCESS, 0;
print "nice = $nice\n";             # nice = 0
```

Ελέγξτε άλλη διεργασία μέσω PID:

```
use POSIX qw(PRIO_PROCESS);
my $nice = getpriority PRIO_PROCESS, $pid;
```

Διαβάστε την προτεραιότητα της ομάδας διεργασιών της καλούσας διεργασίας:

```
use POSIX qw(PRIO_PGRP);
my $nice = getpriority PRIO_PGRP, 0;
```

Διαβάστε το ανώτατο όριο προτεραιότητας που σχετίζεται με ID χρήστη. Η τιμή που επιστρέφεται είναι η *χαμηλότερη* (πιο ευνοϊκή) τιμή nice που έχει οποιαδήποτε από τις διεργασίες αυτού του χρήστη:

```
use POSIX qw(PRIO_USER);
my $nice = getpriority PRIO_USER, $uid;
```

Ασφαλής ανάγνωση που ξεχωρίζει το -1-ως-τιμή από το -1-ως-σφάλμα:

```
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";
}
```

Οριακές περιπτώσεις

- **Το `-1` είναι έγκυρη επιστροφή**, όχι δείκτης σφάλματος. Πάντα καθαρίζετε την `$!` πριν την κλήση και ελέγξτε την μετά.
- **Το `WHO == 0`** είναι η συντομογραφία «εαυτός» για κάθε `WHICH`: τρέχον PID, τρέχον PGID, ή πραγματικό UID. Το να περάσετε ρητά το δικό σας PID / PGID / UID δίνει το ίδιο αποτέλεσμα.
- **Η `PRIO_USER` με UID που κατέχει πολλές διεργασίες** επιστρέφει την αριθμητικά χαμηλότερη τιμή nice μεταξύ αυτών των διεργασιών, αντικατοπτρίζοντας τον πιο ευνοϊκό χρονοπρογραμματισμό που απολαμβάνει αυτή τη στιγμή ο χρήστης.
- **Απών στόχος.** Ένα PID, PGID ή UID χωρίς αντίστοιχη διεργασία αποδίδει `-1` και θέτει την `$!` σε `ESRCH`.
- **Λάθος `WHICH`.** Τιμή εκτός των `PRIO_PROCESS` / `PRIO_PGRP` / `PRIO_USER` αποδίδει `-1` και θέτει την `$!` σε `EINVAL`.

- **Χωρίς `getpriority(2)` στο σύστημα.** Η κλήση εγείρει μοιραία εξαίρεση αντί να επιστρέψει τιμή. Προστατευτείτε με `eval` σε φορητό κώδικα.
- **Χωρίς εισαγωγή `POSIX`, χωρίς σταθερές.** Η γραφή `getpriority 0, 0` λειτουργεί κατά τύχη επειδή το `PRIO_PROCESS` τυχαίνει να είναι `0` σε Linux, αλλά δεν είναι φορητό. Πάντα εισάγετε και χρησιμοποιείτε τις κατονομασμένες σταθερές.
- **Η αύξηση προτεραιότητας απαιτεί root.** Η ίδια η `getpriority` δεν χρειάζεται προνόμια· η αντίστοιχη κλήση `setpriority` αποτυγχάνει με `EACCES` όταν μη-root καλών προσπαθεί να *μειώσει* την τιμή nice.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `setpriority` - η πλευρά εγγραφής του ίδιου ζεύγους κλήσεων συστήματος· ίδιες σταθερές `WHICH`, ίδιες επιφυλάξεις για το `-1`
- `POSIX` - πηγή των `PRIO_PROCESS`, `PRIO_PGRP` και `PRIO_USER`· καμία από αυτές δεν είναι ενσωματωμένη
- `getpgrp` - το ID ομάδας διεργασιών που θα περνούσατε ως `WHO` υπό `PRIO_PGRP`
- `fork` - το παιδί κληρονομεί την τιμή nice του γονέα· διαβάστε την πίσω με `getpriority` μετά το fork αν χρειάζεστε επιβεβαίωση
- `$!` - το μοναδικό κανάλι για διάκριση μιας αληθινής προτεραιότητας `-1` από αποτυχημένη κλήση

Πληροφορίες δικτύου

getprotobyname

Αναζητά ένα πρωτόκολλο IP βάσει του ονόματος κειμένου του και επιστρέφει την εγγραφή της βάσης δεδομένων του.

Η `getprotobyname` είναι το περιτύλιγμα της Perl γύρω από την `getprotobyname(3)` της C library του συστήματος. Δοθέντος ενός ονόματος πρωτοκόλλου όπως `"tcp"`, `"udp"`, ή `"icmp"`, επιστρέφει την αντίστοιχη εγγραφή από τη βάση δεδομένων `/etc/protocols` του συστήματος (ή όποιο backend κατευθύνει το `nsswitch.conf`). Η τυπική χρήση είναι η μετατροπή ενός αναγνώσιμου από άνθρωπο ονόματος πρωτοκόλλου στην αριθμητική τιμή που θέλουν οι `socket` και `setsockopt`.

Σύνοψη

```
getprotobyname NAME
my ($name, $aliases, $proto) = getprotobyname NAME;
my $proto                    = getprotobyname NAME;   # scalar␣
↪context
```

Τι επιστρέφεται

Σε περιβάλλον λίστας, μια λίστα που μοιάζει με τετραστοιχειακή με τρία πεδία:

```
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
```

Σε βαθμωτό περιβάλλον, μόνο τον αριθμητικό αριθμό πρωτοκόλλου (`$proto`). Αυτή είναι η συνηθισμένη μορφή, επειδή ο αριθμός πρωτοκόλλου είναι ολόκληρος ο λόγος για τον οποίο έγινε η αναζήτηση:

```
my $tcp = getprotobyname("tcp");   # 6
```

Αν το πρωτόκολλο δεν βρεθεί, η επιστροφή είναι κενή λίστα σε περιβάλλον λίστας και `undef` σε βαθμωτό περιβάλλον. Πάντα ελέγχετε την οριστικότητα πριν χρησιμοποιήσετε το αποτέλεσμα - το `0` είναι έγκυρος αριθμός πρωτοκόλλου (`"ip"`), οπότε ο έλεγχος μόνο για αλήθεια είναι λάθος.

```
defined(my $tcp = getprotobyname("tcp"))
    or die "tcp protocol not in /etc/protocols";
```

Καθολική κατάσταση που επηρεάζει

- Μοιράζεται τον δρομέα επανάληψης της βάσης δεδομένων πρωτοκόλλων με τις `getprotobynumber`, `getprotoent`, `setprotoent`, και `endprotoent`. Μια κλήση στην `getprotobyname` μπορεί να επανατοποθετήσει ή να επαναφέρει αυτόν τον δρομέα ανάλογα με την C library του συστήματος· μη διαπλέκετε αναζητήσεις βάσει ονόματος με βρόχο επανάληψης `getprotoent`.
- Σε ορισμένες πλατφόρμες η υποκείμενη `getprotobyname(3)` θέτει το `$!` / `errno` σε περίπτωση αποτυχίας. Φορητός κώδικας δεν πρέπει να βασίζεται σε αυτό - αντιμετωπίστε μια απροσδιόριστη επιστροφή ως «δεν βρέθηκε» και προχωρήστε.

Παραδείγματα

Μετατροπή ονόματος σε αριθμό πρωτοκόλλου για την `socket`:

```
use Socket;
my $proto = getprotobyname("tcp")
    // die "tcp not configured";
socket(my $sock, PF_INET, SOCK_STREAM, $proto)
    or die "socket: $!";
```

Μορφή λίστας, για να δείτε το κανονικό όνομα και τα ψευδώνυμα που χρησιμοποιεί ένα σύστημα:

```
my ($name, $aliases, $proto) = getprotobyname("TCP");
print "name=$name aliases=$aliases number=$proto\n";
# name=tcp aliases=TCP number=6
```

Αμυντική αναζήτηση με εφεδρεία για τα δύο πρωτόκολλα που πραγματικά χρειάζεται κάθε σενάριο:

```
my %num_for = (tcp => 6, udp => 17, icmp => 1);
sub proto_num {
    my ($name) = @_;
    return getprotobyname($name) // $num_for{lc $name};
}
```

Το βαθμωτό περιβάλλον απαιτείται όταν η τιμή επιστροφής τροφοδοτεί έκφραση· το περιβάλλον λίστας θα ισοπεδωνόταν στην περιβάλλουσα λίστα:

```
my @args = (SOCK_STREAM, scalar getprotobyname("tcp"));
```

Οριακές περιπτώσεις

- **Η ευαισθησία στα πεζά/κεφαλαία εξαρτάται από το σύστημα.** Η `getprotobyname(3)` της glibc ταιριάζει το κανονικό όνομα και κάθε ψευδώνυμο χωρίς διάκριση πεζών-κεφαλαίων· άλλες C libraries μπορεί να μην το κάνουν. Αν έχει σημασία η φορητότητα, περάστε τη μορφή πεζών (`"tcp"`, όχι `"TCP"`).
- **Ο αριθμός πρωτοκόλλου `0` είναι έγκυρος** (`"ip"`). Χρησιμοποιήστε `defined(...)`, όχι λογική αλήθεια, όταν ελέγχετε για επιτυχία.
- **Δεν είναι το ίδιο με αριθμό θύρας.** Η `getprotobyname("http")` επιστρέφει `undef` - το `"http"` είναι όνομα *υπηρεσίας*, όχι πρωτοκόλλου. Χρησιμοποιήστε την `getservbyname` για αναζητήσεις υπηρεσιών.
- **Η κενή συμβολοσειρά ψευδωνύμου είναι κανονική.** Πολλές εγγραφές δεν έχουν ψευδώνυμα· η δεύτερη τιμή επιστροφής είναι τότε η κενή συμβολοσειρά, όχι `undef`.
- **Παρενέργειες του δρομέα επανάληψης.** Αν το πρόγραμμά σας είναι στη μέση ενός βρόχου `getprotoent`, η κλήση της `getprotobyname` μπορεί να επαναφέρει τον δρομέα και να παραλείψει ή να επαναλάβει εγγραφές. Ολοκληρώστε την επανάληψη, ή αποθηκεύστε την κατάσταση, πριν κάνετε αναζητήσεις βάσει ονόματος.
- **Ασφάλεια threads.** Η υποκείμενη κλήση C δεν είναι thread-safe σε κάθε πλατφόρμα - τυπικά χρησιμοποιεί στατικό ενταμιευτή ανά διεργασία. Η Perl μπορεί να χρησιμοποιήσει την επανεισόδιμη μορφή `getprotobyname_r` όταν είναι διαθέσιμη, αλλά μην το υποθέτετε.
- **Λείπει η εγγραφή `/etc/protocols`.** Σε minimal containers η βάση δεδομένων μπορεί να απουσιάζει, οπότε κάθε κλήση επιστρέφει `undef`. Δομήστε πίνακες πρωτοκόλλων μέσα στον κώδικά σας όταν το περιβάλλον στόχος δεν εγγυάται την παρουσία βάσης πρωτοκόλλων.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `getprotobynumber` - η αντίστροφη αναζήτηση· χρησιμοποιήστε την όταν έχετε το αριθμητικό πρωτόκολλο και θέλετε το όνομά του
- `getprotoent` - επανάληψη σε κάθε εγγραφή της βάσης δεδομένων πρωτοκόλλων μία εγγραφή τη φορά
- `setprotoent` / `endprotoent` - έλεγχος του δρομέα επανάληψης που μοιράζονται με αυτή τη συνάρτηση
- `getservbyname` - η ισοδύναμη αναζήτηση για υπηρεσίες (θύρες), όχι πρωτόκολλα· τα δύο συγχέονται συχνά
- `socket` - ο συνήθης καταναλωτής του επιστρεφόμενου αριθμού πρωτοκόλλου
- `Net::protoent` - αντικειμενοστραφές περιτύλιγμα που επιστρέφει blessed εγγραφή με ονομασμένους accessors αντί για λίστα κατά θέση

Πληροφορίες δικτύου

getprotobynumber

Αναζητά μια εγγραφή πρωτοκόλλου δικτύου με τον αριθμό πρωτοκόλλου που της έχει εκχωρηθεί.

Η `getprotobynumber` περιβάλλει την κλήση συστήματος `getprotobynumber(3)`. Δοθέντος ενός αριθμητικού αναγνωριστικού πρωτοκόλλου - των ίδιων τιμών που κατονομάζονται από σταθερές όπως `IPPROTO_TCP` (6), `IPPROTO_UDP` (17), `IPPROTO_ICMP` (1) - επιστρέφει την αντίστοιχη εγγραφή από τη βάση πρωτοκόλλων του συστήματος (τυπικά το `/etc/protocols`). Η συνηθισμένη χρήση είναι η μετατροπή ενός αριθμού πρωτοκόλλου που λαμβάνεται από το καλώδιο, από έναν πίνακα δρομολόγησης ή που επιστρέφει η `getsockopt` σε ευανάγνωστο όνομα.

Σύνοψη

```
getprotobynumber NUMBER
```

Τι επιστρέφεται

Σε περιβάλλον λίστας, μια λίστα τεσσάρων στοιχείων:

```
my ($name, $aliases, $proto) = getprotobynumber($number);
```

- `$name` - κανονικό όνομα πρωτοκόλλου (`"tcp"`, `"udp"`, `"icmp"`).
- `$aliases` - συμβολοσειρά εναλλακτικών ονομάτων για το πρωτόκολλο διαχωρισμένων με κενά, ή κενή συμβολοσειρά αν δεν υπάρχουν.
- `$proto` - ο αριθμός πρωτοκόλλου που περάσατε, που επιστρέφεται από την εγγραφή της βάσης.

Σε βαθμωτό περιβάλλον, λαμβάνετε μόνο το `$name`. Αν ο αριθμός δεν βρίσκεται στη βάση, η βαθμωτή επιστροφή είναι `undef` και η επιστροφή λίστας είναι κενή.

```
my $name = getprotobynumber(6);          # "tcp"
my @rec  = getprotobynumber(6);          # ("tcp", "", 6)
my $none = getprotobynumber(9999);       # undef
```

Καθολική κατάσταση που επηρεάζει

Μοιράζεται τον δρομέα της βάσης πρωτοκόλλων με τις `getprotobyname`, `getprotoent`, `setprotoent` και `endprotoent`. Η κλήση της `getprotobynumber` δεν ενοχλεί μια επανάληψη σε εξέλιξη στα περισσότερα συστήματα, αλλά φορητός κώδικας που διαπλέκει μεμονωμένες αναζητήσεις με επανάληψη μέσω `getprotoent` θα πρέπει να καλέσει στη συνέχεια την `setprotoent` για επαναφορά του δρομέα.

Σε αποτυχία, το `$!` τίθεται στο υποκείμενο `errno`, αν και οι περισσότερες υλοποιήσεις απλώς επιστρέφουν «δεν βρέθηκε» χωρίς να ρυθμίσουν `errno`.

Παραδείγματα

Ανάλυση του κανονικού ονόματος ενός αριθμού πρωτοκόλλου:

```
my $name = getprotobynumber(6);
print "$name\n";                         # "tcp"
```

Πλήρης εγγραφή με aliases:

```
my ($name, $aliases, $proto) = getprotobynumber(1);
print "$name ($proto): $aliases\n";      # e.g. "icmp (1): ICMP"
```

Αντίστροφη αναζήτηση πρωτοκόλλου που βγαίνει από επικεφαλίδα πακέτου:

```
sub protocol_label {
    my ($num) = @_;
    my $name = getprotobynumber($num);
    return defined $name ? $name : "proto-$num";
}
```

Πλήρης κύκλος έναντι της `getprotobyname` - χρήσιμο ως έλεγχος ορθότητας ότι η βάση πρωτοκόλλων του συστήματος γνωρίζει το όνομα που περιμένετε:

```
my $num  = getprotobyname("tcp");        # 6
my $back = getprotobynumber($num);       # "tcp"
```

Καλαίσθητη εκτύπωση ενός επιπέδου `getsockopt`. Το όρισμα `level` της `getsockopt` είναι αριθμός πρωτοκόλλου· η `getprotobynumber` το μετατρέπει πίσω σε ετικέτα για καταγραφή:

```
my ($level) = unpack("I", $packed_level);
printf "socket option at level %s\n", getprotobynumber($level) //
 ↪$level;
```

Οριακές περιπτώσεις

- **Παγίδα προτεραιότητας τελεστή λίστας.** Παρόλο που η `getprotobynumber` δέχεται ένα όρισμα, αναλύεται με προτεραιότητα τελεστή λίστας. Το όνομα της συνάρτησης καταπίνει ό,τι βρίσκεται δεξιά του ως μια μεγάλη έκφραση λίστας:

```
getprotobynumber $number eq 'icmp'    # WRONG - equivalent to:
getprotobynumber($number eq 'icmp')   # looks up protocol 0 or 1
getprotobynumber($number) eq 'icmp'   # what you meant
```

  Βάζετε πάντα παρενθέσεις στο όρισμα όταν συγκρίνετε το αποτέλεσμα, ή αναθέστε πρώτα την αναζήτηση σε μεταβλητή.

- **Άγνωστος αριθμός**: η βαθμωτή επιστροφή είναι `undef`· η επιστροφή λίστας είναι κενή λίστα. Προστατευτείτε με `defined`:

```
defined(my $name = getprotobynumber($n))
    or die "unknown protocol number $n";
```

- **Μη ακέραιο όρισμα**: μετατρέπεται σε ακέραιο μέσω του συνηθισμένου αριθμητικού εξαναγκασμού πριν την αναζήτηση. Μια συμβολοσειρά όπως `"tcp"` εξαναγκάζεται σε `0` και λαμβάνετε όποια εγγραφή (αν υπάρχει) είναι καταχωρημένη για το πρωτόκολλο `0`, που είναι συνήθως το `"ip"`. Για αναζήτηση με βάση το όνομα, χρησιμοποιήστε την `getprotobyname`.

- **Αρνητικοί ή εκτός εύρους αριθμοί**: καμία εγγραφή, επιστρέφεται κενό / `undef`. Δεν εκπέμπεται προειδοποίηση.

- **Η αντικειμενική διεπαφή `Net::protoent`**. Αν εισαγάγετε το `Net::protoent`, η `getprotobynumber` υπερκαλύπτεται ώστε να επιστρέφει blessed αντικείμενο με κατονομασμένους accessors (`$p->name`, `$p->aliases`, `$p->proto`). Βολικό όταν δεν θέλετε να θυμάστε τη σειρά των πεδίων κατά θέση.

- **Αναζήτηση βάσης δεδομένων, όχι ενδοσκόπηση στοίβας πρωτοκόλλων**. Αυτή η συνάρτηση διαβάζει το `/etc/protocols` (ή την αντίστοιχη πηγή NSS). Σας λέει ποιο όνομα εκχωρεί το σύστημα σε έναν αριθμό· δεν σας λέει αν ο πυρήνας υποστηρίζει όντως αυτό το πρωτόκολλο ούτε αν μπορεί να ανοιχτεί μια υποδοχή αυτού του τύπου.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `getprotobyname` - η αντίστροφη: αναζητά μια εγγραφή πρωτοκόλλου με το όνομα όταν έχετε ετικέτα όπως `"tcp"` και χρειάζεστε τον αριθμό
- `getprotoent` - διατρέχει κάθε εγγραφή στη βάση πρωτοκόλλων, όταν θέλετε να τις απαριθμήσετε ή ευρετηριάσετε όλες
- `setprotoent` - ξανανοίγει / επαναφέρει τον δρομέα της βάσης πρωτοκόλλων· συνδυάστε με `endprotoent` γύρω από μια επανάληψη

- `getservbyport` - το ανάλογο του πίνακα υπηρεσιών, μετατρέπει έναν αριθμό θύρας σε όνομα υπηρεσίας
- `getsockopt` - επιστρέφει αριθμό πρωτοκόλλου ως το όρισμά της `level`· τροφοδοτήστε το στην `getprotobynumber` για ευανάγνωστα logs
- `Socket` - εξάγει τις σταθερές `IPPROTO_*`, των οποίων τις αριθμητικές τιμές περνάτε ως όρισμα

Πληροφορίες δικτύου

getprotoent

Λαμβάνει την επόμενη εγγραφή από τη βάση δεδομένων πρωτοκόλλων.

Η `getprotoent` διασχίζει τη βάση δεδομένων πρωτοκόλλων του συστήματος - σε παραδοσιακά συστήματα τύπου Unix είναι το `/etc/protocols`, σε σύγχρονα συστήματα ό,τι αντιστοιχίζει σε αυτή η C library και η στοίβα NSS - επιστρέφοντας μία εγγραφή ανά κλήση μέχρι να εξαντληθεί η βάση δεδομένων. Είναι το σκέλος επανάληψης της τριάδας `setprotoent` / `getprotoent` / `endprotoent`. Καταφύγετε στην `getprotobyname` ή την `getprotobynumber` όταν θέλετε μια μόνο στοχευμένη αναζήτηση· η `getprotoent` είναι για απαρίθμηση κάθε πρωτοκόλλου που γνωρίζει το σύστημα.

Ο δρομέας επανάληψης μοιράζεται σε επίπεδο διεργασίας και ανήκει στη C library. Η `setprotoent` τον επαναφέρει στην αρχή και προαιρετικά καρφώνει ανοιχτό τον υποκείμενο περιγραφέα· η `endprotoent` τον κλείνει. Μεταξύ αυτών των δύο κλήσεων η `getprotoent` αποδίδει κάθε εγγραφή με τη σειρά, και μετά επιστρέφει την κενή λίστα (ή `undef` σε βαθμωτό περιβάλλον) μόλις δεν απομένει τίποτα άλλο.

Σύνοψη

```
my ($name, $aliases, $proto) = getprotoent;
my $name = getprotoent;
```

Τι επιστρέφεται

Περιβάλλον λίστας - λίστα τριών στοιχείων που αντικατοπτρίζει το C `struct protoent`:

- `$name` - το κανονικό όνομα πρωτοκόλλου, π.χ. `"tcp"`, `"udp"`, `"icmp"`.
- `$aliases` - συμβολοσειρά εναλλακτικών ονομάτων για το ίδιο πρωτόκολλο, διαχωρισμένων με κενά. Συχνά κενή.
- `$proto` - ο αριθμητικός αριθμός πρωτοκόλλου όπως χρησιμοποιείται στις κεφαλίδες IP και γίνεται δεκτός από τις `socket` / `getsockopt`.

Στο τέλος της βάσης δεδομένων η επιστροφή είναι η κενή λίστα. Η σημείωση του upstream perlfunc για *«μια μοναδική άνευ νοήματος αληθή τιμή»* για απούσες εγγραφές αφορά τις αναζητήσεις κατά όνομα και κατά αριθμό, όχι την `getprotoent` - η υπέρβαση του τέλους της επανάληψης είναι καθαρή κενή λίστα.

Βαθμωτό περιβάλλον - μόνο το κανονικό `$name`, ή `undef` στο τέλος της βάσης δεδομένων. Χρησιμοποιήστε περιβάλλον λίστας όποτε χρειάζεστε τον αριθμό ή τα ψευδώνυμα· δεν μπορείτε να τα ανακατασκευάσετε από το όνομα εκ των υστέρων.

Διασπάστε τα ψευδώνυμα σε λίστα με μια απλή `split`:

```
my @alts = split ' ', $aliases;
```

Καθολική κατάσταση που επηρεάζει

- Ο **δρομέας της βάσης δεδομένων πρωτοκόλλων**, ένας πόρος ανά διεργασία που ανήκει στη C library και μοιράζεται με τις `setprotoent`, `endprotoent`, `getprotobyname` και `getprotobynumber`. Μια αναζήτηση κατά όνομα ή κατά αριθμό μπορεί να επαναφέρει τον δρομέα στη μέση της επανάληψης σε κάποιες πλατφόρμες - δείτε *Οριακές περιπτώσεις*.
- Δεν διαβάζονται ή γράφονται ειδικές μεταβλητές σε επίπεδο Perl. Σε αντίθεση με την `gethostent`, οι ρουτίνες πρωτοκόλλων δεν έχουν ισοδύναμο `h_errno`, οπότε η `$?` δεν θίγεται. Το τέλος της βάσης δεδομένων δεν είναι αποτυχία και δεν θέτει την `$!`.

Παραδείγματα

Απαρίθμηση κάθε πρωτοκόλλου που γνωρίζει το σύστημα:

```
setprotoent(1);
while (my ($name, $aliases, $proto) = getprotoent) {
    print "$proto\t$name\t$aliases\n";
}
endprotoent;
```

Βαθμωτό περιβάλλον - παραθέστε μόνο τα κανονικά ονόματα:

```
setprotoent(0);
while (defined(my $name = getprotoent)) {
    print "$name\n";
}
endprotoent;
```

Δομήστε αντιστοιχία ονόματος-σε-αριθμό σε ένα πέρασμα ώστε οι μετέπειτα αναζητήσεις να μην χτυπούν καθόλου τη βάση δεδομένων:

```
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
```

Διεπαφή ανά εγγραφή μέσω της παράκαμψης `Net::protoent` - μέθοδοι προσπέλασης αντί για θεσιακό αποπακετάρισμα:

```
use Net::protoent;
while (my $p = getprotoent) {
    printf "%-10s %3d\n", $p->name, $p->proto;
}
```

Οριακές περιπτώσεις

- **Καμία σιωπηρή επαναφορά.** Η `getprotoent` δεν ανοίγει ούτε επαναφέρει τη βάση δεδομένων από μόνη της. Η πρώτη κλήση μετά την εκκίνηση του προγράμματος λειτουργεί επειδή η C library ανοίγει το αρχείο τεμπέλικα, αλλά μόλις φτάσει το τέλος, οι επόμενες κλήσεις συνεχίζουν να επιστρέφουν τέλος-βάσης μέχρι μια `setprotoent` να επαναφέρει τον δρομέα. Πάντα οριοθετήστε την απαρίθμηση με ρητή `setprotoent` και `endprotoent`.
- **Το τέλος της βάσης δεδομένων είναι η κενή λίστα, όχι `undef`.** Σε περιβάλλον λίστας η συνθήκη βρόχου `while (my @r = getprotoent)` λειτουργεί επειδή η κενή λίστα είναι ψευδής. Σε βαθμωτό περιβάλλον χρησιμοποιήστε `while (defined(my $name = getprotoent))` - όνομα πρωτοκόλλου `"0"` είναι θεωρητικά έγκυρο και θα τερμάτιζε ψευδώς έναν βρόχο που στηρίζεται στην αληθότητα.
- **Τα ψευδώνυμα είναι ένα βαθμωτό, όχι λίστα.** Το `$aliases` είναι μία ενιαία συμβολοσειρά διαχωρισμένη με κενά· διασπάστε σε λευκούς χαρακτήρες αν θέλετε λίστα. Κενό πεδίο ψευδωνύμων είναι η συνηθισμένη περίπτωση, όχι σφάλμα.
- **Παρεμβολή με αναζητήσεις κατά όνομα / κατά αριθμό.** Μια κλήση σε `getprotobyname` ή `getprotobynumber` κατά τη διάρκεια μιας διάσχισης μπορεί να επαναφέρει τον κοινό δρομέα σε κάποιες πλατφόρμες. Ολοκληρώστε πρώτα την επανάληψη, ή επανεκκινήστε την με `setprotoent` μετά την αναζήτηση.
- **Μη επανεισόδιμη.** Ο δρομέας της βάσης δεδομένων είναι ένας μοναδικός καθολικός πόρος μέσα στη C library. Δύο ταυτόχρονες επαναλήψεις στην ίδια διεργασία - σε διαφορετικά threads, χειριστές σήματος ή εμφωλευμένες επιστροφές κλήσεων - διαφθείρουν την κατάσταση η μία της άλλης. Περιορίστε κάθε διάσχιση σε ένα thread τη φορά.
- **Πλατφόρμες χωρίς βάση δεδομένων πρωτοκόλλων.** Σε συστήματα χωρίς ισοδύναμο `/etc/protocols`, η κλήση είναι no-op που επιστρέφει αμέσως την κενή λίστα αντί να εγείρει σφάλμα. Δείτε το `perlport` για την πλήρη λίστα ιδιαιτεροτήτων βάσεων δεδομένων δικτύου.
- **Tainting.** Οι εγγραφές προέρχονται από αρχείο που διαβάζει ο πυρήνας για λογαριασμό σας και δεν σημειώνονται ως tainted. Αντιμετωπίστε τις ως μη έμπιστες έτσι κι αλλιώς όταν το αρχείο είναι εγγράψιμο από τον χρήστη.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `setprotoent` - επαναφέρει τον επαναλήπτη στην αρχή και προαιρετικά καρφώνει ανοιχτό τον υποκείμενο περιγραφέα για όλη τη διάσχιση
- `endprotoent` - κλείνει τον επαναλήπτη και απελευθερώνει το handle της βάσης δεδομένων μόλις ολοκληρωθεί η απαρίθμηση
- `getprotobyname` - στοχευμένη αναζήτηση με όνομα πρωτοκόλλου αντί για πλήρη απαρίθμηση
- `getprotobynumber` - στοχευμένη αναζήτηση με αριθμητικό αριθμό πρωτοκόλλου
- `getservent` - ίδιο μοτίβο επανάληψης για τη βάση δεδομένων υπηρεσιών, η οποία αναφέρεται σε ονόματα πρωτοκόλλων από αυτήν εδώ
- `gethostent` - ίδιο ιδίωμα για τη βάση δεδομένων υπολογιστών

Πληροφορίες χρηστών και ομάδων

getpwent

Επιστροφή της επόμενης εγγραφής από τη βάση δεδομένων κωδικών πρόσβασης του συστήματος.

Η `getpwent` διασχίζει τη βάση δεδομένων κωδικών πρόσβασης μία εγγραφή τη φορά. Η πρώτη κλήση ανοίγει τη βάση δεδομένων (τυπικά `/etc/passwd`, ενδεχομένως διευρυμένη μέσω NSS με LDAP, SSSD ή παρόμοιο)· κάθε επόμενη κλήση επιστρέφει την επόμενη εγγραφή. Όταν η επανάληψη εξαντληθεί, η κλήση επιστρέφει κενή λίστα σε περιβάλλον λίστας ή `undef` σε βαθμωτό περιβάλλον. Συνδυάστε την με `setpwent` για επαναφορά και `endpwent` για απελευθέρωση τυχόν πόρων που κρατά η βιβλιοθήκη βάσης δεδομένων.

Σύνοψη

```
my @fields = getpwent;
my $name   = getpwent;      # scalar context: login name only
```

Τι επιστρέφεται

Σε περιβάλλον λίστας, μια λίστα 10 στοιχείων που περιγράφει έναν λογαριασμό:

```
my ( $name,   $passwd,   $uid,   $gid,   $quota,
     $comment, $gcos,    $dir,   $shell, $expire ) = getpwent;
#    0         1          2       3       4
#    5         6          7       8       9
```

- `$name` - όνομα σύνδεσης.
- `$passwd` - το πεδίο κρυπτογραφημένου κωδικού πρόσβασης. Σε συστήματα με shadow passwords αυτό είναι συνήθως `"x"` ή `"*"` εκτός αν η διεργασία είναι προνομιούχα· η πραγματική τιμή κατακερματισμού βρίσκεται στο `/etc/shadow`. Tainted υπό `-T`.
- `$uid`, `$gid` - αριθμητικά ids χρήστη και πρωτεύουσας ομάδας.

- `$quota` - όριο δίσκου σε συστήματα που το υποστηρίζουν, αλλιώς κενή συμβολοσειρά. Σε ορισμένα συστήματα αυτή η θέση φέρει `$change` ή `$age` (δεδομένα παλαίωσης κωδικού).
- `$comment` - διαχειριστικό σχόλιο σε συστήματα που το υποστηρίζουν, αλλιώς κενό. Σε ορισμένα συστήματα αυτή η θέση φέρει `$class`.
- `$gcos` - το πεδίο GECOS, συμβατικά το πραγματικό όνομα του χρήστη. Εγγράψιμο από τον χρήστη σε πολλά συστήματα, επομένως tainted υπό `-T`.
- `$dir` - αρχικός κατάλογος.
- `$shell` - κέλυφος σύνδεσης. Tainted υπό `-T`.
- `$expire` - λήξη λογαριασμού ή κωδικού πρόσβασης σε συστήματα που το υποστηρίζουν, αλλιώς απουσιάζει.

Σε βαθμωτό περιβάλλον, επιστρέφεται μόνο το όνομα σύνδεσης. Όταν η επανάληψη εξαντληθεί, η βαθμωτή τιμή είναι `undef` και η τιμή λίστας είναι κενή.

Το αν τα `$quota`, `$comment` και `$expire` φέρουν ουσιαστικά δεδομένα στην τρέχουσα έκδοση εξαρτάται από το πώς ρυθμίστηκε η Perl· επιθεωρήστε τις τιμές `Config` `d_pwquota`, `d_pwage`, `d_pwchange`, `d_pwcomment` και `d_pwexpire` για να το ανακαλύψετε.

Καθολική κατάσταση που επηρεάζει

Η `getpwent` διατηρεί τον δρομέα της σε ένα **καθολικό σε επίπεδο διεργασίας** μπλοκ κατάστασης που ανήκει στη βιβλιοθήκη, μέσα στις ρουτίνες pwd της βιβλιοθήκης C. Συνέπειες:

- Δύο ανεξάρτητοι βρόχοι πάνω στη βάση δεδομένων στην ίδια διεργασία αλληλοεπηρεάζονται. Κάθε `getpwent` προωθεί τον έναν κοινόχρηστο δρομέα.
- Η `setpwent` επαναφέρει αυτόν τον δρομέα στην αρχή.
- Η `endpwent` κλείνει τη βάση δεδομένων και αφήνει τυχόν πόρους (ανοιχτούς περιγραφείς αρχείων, κατάσταση αρθρώματος NSS) που κρατούσε η βιβλιοθήκη.
- Τα threads μοιράζονται επίσης τον δρομέα. Η `getpwent` δεν είναι ασφαλής για ταυτόχρονη κλήση από πολλαπλά threads· αν δύο threads επαναλάβουν παράλληλα, κανένα από τα δύο δεν βλέπει κάθε εγγραφή.

Η `getpwent` δεν διαβάζει ούτε γράφει σε ειδικές μεταβλητές Perl.

Παραδείγματα

Εκτύπωση κάθε ονόματος λογαριασμού και αρχικού καταλόγου:

```
while ( my @pw = getpwent ) {
    print "$pw[0]\t$pw[7]\n";
}
endpwent;
```

Η αποδόμηση με ονομαζόμενα πεδία διαβάζεται πιο καθαρά από την αριθμητική ευρετηρίαση:

```
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;
```

Το βαθμωτό περιβάλλον δίνει το όνομα σύνδεσης, χρήσιμο για απαρίθμηση χρηστών χωρίς αποπακετάρισμα των υπολοίπων:

```
my @users;
while ( defined( my $name = getpwent ) ) {
    push @users, $name;
}
endpwent;
```

Επαναφορά στη μέση μιας διάσχισης με `setpwent`:

```
my @first_pass = (getpwent)[0];
setpwent;                               # back to the top
while ( my @pw = getpwent ) { ... }
endpwent;
```

Συλλογή όλων των εγγραφών σε πίνακα κατακερματισμού με κλειδί το uid, και έπειτα απελευθέρωση της βάσης δεδομένων:

```
my %by_uid;
while ( my @pw = getpwent ) {
    $by_uid{ $pw[2] } = \@pw;
}
endpwent;
```

Οριακές περιπτώσεις

- **Τέλος επανάληψης**: επιστρέφει κενή λίστα σε περιβάλλον λίστας και `undef` σε βαθμωτό περιβάλλον. Οι `while (my @pw = getpwent)` και `while (defined(my $name = getpwent))` είναι οι ιδιωματικές μορφές βρόχου.
- **Παράλειψη της `endpwent`**: διαρρέει όποιους περιγραφείς αρχείων ή κατάσταση NSS άνοιξε η βιβλιοθήκη μέχρι να τερματίσει η διεργασία. Δεν είναι σφάλμα ορθότητας, αλλά γίνεται αισθητό σε προγράμματα μακράς διαρκείας και σε λουριά δοκιμών.
- **Η εμφωλευμένη επανάληψη είναι σπασμένη**: δύο βρόχοι στην ίδια διεργασία μοιράζονται τον μοναδικό δρομέα της βιβλιοθήκης. Συλλέξτε μια διάσχιση σε δομή δεδομένων πριν ξεκινήσετε την επόμενη.
- **Shadow passwords**: σε Linux και Solaris η πραγματική τιμή κατακερματισμού κωδικού βρίσκεται στο `/etc/shadow`, αναγνώσιμη μόνο από τον root. Οι μη προνομιούχες διεργασίες βλέπουν `"x"` ή `"*"` στο `$passwd` και πρέπει να καλέσουν τις shadow-aware ρουτίνες (εκτός του πυρήνα της Perl) για να πάρουν την τιμή κατακερματισμού.

- **Εκπλήξεις NSS**: η βάση δεδομένων είναι ό,τι λέει το `/etc/nsswitch.conf`. Σε έναν υπολογιστή δεμένο με LDAP ή SSSD, η `getpwent` μπορεί να επιστρέψει χιλιάδες λογαριασμούς καταλόγου, να πάρει αξιοσημείωτο χρόνο, και να αποτυγχάνει περιοδικά αν το δίκτυο είναι ασταθές. Σκεφτείτε να αναζητάτε λογαριασμούς ονομαστικά με `getpwnam` όταν ξέρετε ποιον θέλετε.
- **Tainting υπό `-T`**: τα `$passwd`, `$gcos` και `$shell` επιστρέφουν tainted επειδή οι χρήστες μπορούν να τα επηρεάσουν. Αφαιρέστε το taint με σκόπιμη αντιστοιχία regex αν χρειάζεται να τα χρησιμοποιήσετε σε ευαίσθητες λειτουργίες.
- **Ασφάλεια threads**: σε αντίθεση με τις `getpwnam` και `getpwuid`, δεν υπάρχει `getpwent_r` σε POSIX που η Perl να μπορεί να υποκαταστήσει διαφανώς. Η ταυτόχρονη επανάληψη από πολλαπλά threads δεν είναι ασφαλής.
- **Διεπαφή ονόματος**: το `User::pwent` υπερισχύει της `getpwent` με μια έκδοση που επιστρέφει αντικείμενο που εκθέτει πεδία ως μεθόδους (`$pw->name`, `$pw->uid`, ...). Χρησιμοποιήστε το όταν το αποπακετάρισμα αριθμητικών θέσεων γίνεται δύσχρηστο.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `getpwnam` - αναζήτηση ενός λογαριασμού κατά όνομα σύνδεσης· η σωστή κλήση όταν ξέρετε ποιον χρήστη θέλετε
- `getpwuid` - αναζήτηση ενός λογαριασμού κατά αριθμητικό uid
- `setpwent` - επαναφορά του δρομέα `getpwent` στην πρώτη εγγραφή
- `endpwent` - κλείσιμο της βάσης δεδομένων κωδικών και απελευθέρωση πόρων βιβλιοθήκης· καλέστε την όταν τελειώσει η διάσχιση
- `getgrent` - το αντίστοιχο της βάσης ομάδων, ίδιο μοντέλο επανάληψης
- `User::pwent` - διεπαφή ονόματος που επιστρέφει αντικείμενο με ονομαζόμενους accessors αντί για λίστα 10 στοιχείων

Πληροφορίες χρηστών και ομάδων

getpwnam

Αναζήτηση εγγραφής passwd ενός χρήστη κατά όνομα σύνδεσης.

Η `getpwnam` περιτυλίγει την κλήση `getpwnam(3)` της βιβλιοθήκης C του συστήματος. Περάστε ένα όνομα σύνδεσης όπως `"root"` ή `"alice"`· επιστρέφεται η εγγραφή passwd για αυτόν τον χρήστη, ή κενή λίστα / `undef` αν δεν υπάρχει τέτοιος χρήστης. Η μορφή επιστροφής εξαρτάται από το περιβάλλον: το περιβάλλον λίστας δίνει κάθε πεδίο της εγγραφής· το βαθμωτό περιβάλλον δίνει μία χρήσιμη τιμή - για τη `getpwnam` συγκεκριμένα, το αριθμητικό UID.

Σύνοψη

```
getpwnam NAME
```

Τι επιστρέφεται

Περιβάλλον λίστας επιστρέφει μια λίστα 10 στοιχείων που αντικατοπτρίζει το `struct passwd` διευρυμένο με τα BSD extras:

```
my ($name,    $passwd, $uid,  $gid,   $quota,
    $comment, $gcos,   $dir,  $shell, $expire) = getpwnam($login);
```

- `$name` - όνομα σύνδεσης (η ίδια συμβολοσειρά που περάσατε)
- `$passwd` - κρυπτογραφημένος κωδικός πρόσβασης, ή `"x"` / `"*"` σε συστήματα shadow-password. Tainted υπό `-T`.
- `$uid` - αριθμητικό ID χρήστη
- `$gid` - αριθμητικό ID πρωτεύουσας ομάδας
- `$quota` - όριο δίσκου, ή `$change` / `$age` σε ορισμένα συστήματα. Κενή συμβολοσειρά όταν δεν υποστηρίζεται.
- `$comment` - διαχειριστικό σχόλιο, μερικές φορές `$class`. Κενή συμβολοσειρά όταν δεν υποστηρίζεται.
- `$gcos` - πεδίο GECOS (συνήθως το πραγματικό όνομα μαζί με άλλες πληροφορίες χρήστη). Tainted υπό `-T` επειδή οι χρήστες μπορούν να το επεξεργαστούν.
- `$dir` - αρχικός κατάλογος
- `$shell` - κέλυφος σύνδεσης. Tainted υπό `-T`.
- `$expire` - λήξη λογαριασμού / κωδικού, όταν υποστηρίζεται.

Βαθμωτό περιβάλλον επιστρέφει μόνο το `$uid`. Για τη `getpwnam` - μια αναζήτηση *κατά όνομα* - το βαθμωτό περιβάλλον σας δίνει «το άλλο πράγμα», ταιριάζοντας τον κανόνα που τεκμηριώνει το upstream για όλες τις συναρτήσεις `getpw*` / `getgr*` / `gethost*` / `getnet*` / `getproto*` / `getserv*`.

Σε αποτυχία αναζήτησης και τα δύο περιβάλλοντα επιστρέφουν την κενή λίστα / `undef`. Ελέγχετε πάντα την επιστροφή πριν το αποπακετάρισμα.

Καθολική κατάσταση που επηρεάζει

- Θέτει την `$!` σε αποτυχία κλήσης συστήματος (σπάνια - η «μη εύρεση χρήστη» δεν είναι αποτυχία, είναι κενή επιστροφή).
- Υπό `-T` (taint mode), τα `$passwd`, `$gcos` και `$shell` επιστρέφουν tainted. Δείτε το `perlsec` για το πώς να τα καθαρίσετε.
- Μοιράζεται την ίδια υποκείμενη κατάσταση επαναλήπτη με τις `getpwent`, `setpwent` και `endpwent`. Μια κλήση `getpwnam` **δεν** διαταράσσει έναν εν εξελίξει βρόχο `getpwent` στη glibc, αλλά φορητός κώδικας δεν θα έπρεπε να το υποθέτει αυτό.

Παραδείγματα

Βασική αναζήτηση ονόματος-σε-UID σε βαθμωτό περιβάλλον:

```
my $uid = getpwnam("root");          # 0
```

Πλήρης εγγραφή σε περιβάλλον λίστας, με αμυντικό έλεγχο:

```
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";
```

Μοτίβο εκχώρησης από το upstream - άμεσο αποπακετάρισμα με `or die`:

```
my ($login, $pass, $uid, $gid) = getpwnam($user)
    or die "$user not in passwd file";
```

Σύγκριση ιδιοκτησίας αρχείου με ονομασμένο χρήστη χωρίς χειρόγραφη ευρετηρίαση πεδίων, μέσω της παράκαμψης `User::pwent`:

```
use User::pwent;
use File::stat;
my $is_theirs = (stat($filename)->uid == getpwnam($whoever)->uid);
```

Το `use User::pwent` αντικαθιστά την `getpwnam` με μια έκδοση που επιστρέφει blessed αντικείμενο με accessors `->name`, `->uid`, `->gid`, `->dir`, `->shell` - βολικό όταν θέλετε ένα πεδίο και δεν θέλετε να μετράτε κόμματα. Οι δύο μορφές δεν αναμειγνύονται: μέσα σε μία εμβέλεια χρησιμοποιήστε είτε την παράκαμψη είτε τη built-in, όχι και τις δύο.

Αλυσίδα εφεδρείας για το «ως ποιος εκτελούμαι»:

```
my $login = getlogin() || getpwuid($<) || "unknown";
```

Η `getpwuid` είναι το αντίστοιχο κατά UID· η `getlogin` διαβάζει την εγγραφή του ελέγχοντος tty και μπορεί να ψεύδεται σε αποσπασμένες διεργασίες, οπότε είναι η πρώτη επιλογή, όχι η μόνη.

Οριακές περιπτώσεις

- **Δεν υπάρχει τέτοιος χρήστης** επιστρέφει κενή λίστα σε περιβάλλον λίστας και `undef` σε βαθμωτό περιβάλλον. Αυτό δεν είναι σφάλμα - η `$!` δεν τίθεται. Διακρίνετε «λείπει χρήστης» από «αποτυχία αναζήτησης» ελέγχοντας την `$!` μόνο όταν θέλετε να αναφέρετε απρόσμενη αποτυχία, όχι όταν αποφασίζετε αν ο χρήστης υπάρχει.
- **Το `$passwd` σπάνια είναι χρήσιμο.** Σε σύγχρονα συστήματα Linux και BSD η πραγματική τιμή κατακερματισμού βρίσκεται στο `/etc/shadow`, αναγνώσιμη μόνο από τον root. Εδώ παίρνετε `"x"` ή `"*"`. Χρησιμοποιήστε PAM ή το άρθρωμα `Authen::PAM` για ταυτοποίηση αντί να συγκρίνετε με αυτό το πεδίο.
- **Φορητότητα `$quota`, `$comment`, `$expire`.** Αυτά τα πεδία είναι κενά ή χωρίς νόημα σε Linux. Το POD του upstream απαριθμεί τιμές `Config` (`d_pwquota`, `d_pwage`,

d_pwchange, d_pwcomment, d_pwexpire) που μπορείτε να ελέγξετε αν χρειάζεται να γνωρίζετε κατά τον χρόνο εκτέλεσης αν η πλατφόρμα τα συμπληρώνει.

- **Tainted πεδία υπό -T.** Τα `$passwd`, `$gcos`, `$shell` είναι tainted επειδή οι χρήστες μπορούν να τα τροποποιήσουν (μέσω `chsh`, `chfn` ή άμεσων επεξεργασιών `/etc/passwd` σε προνομιούχα περιβάλλοντα). Περάστε τα μέσα από σύλληψη regex για να τα καθαρίσετε πριν τη χρήση τους σε κλήση συστήματος.
- **Η τιμή επιστροφής είναι μια «μοναδική άνευ νοήματος αληθής τιμή»** σε λογικό βαθμωτό περιβάλλον *όταν η εγγραφή υπάρχει αλλά θέλατε περιβάλλον λίστας.* Η τεκμηριωμένη επιστροφή σε βαθμωτό περιβάλλον για τη `getpwnam` είναι το UID, που για τον `root` είναι `0` - ψευδής τιμή. Η `if (getpwnam("root")) { ... }` ως εκ τούτου αποτυγχάνει για τον root. Χρησιμοποιήστε `defined` ή περιβάλλον λίστας:

```
if (defined getpwnam("root"))     { ... }   # correct
if (my @pw = getpwnam("root"))    { ... }   # also correct
```

- **Το όνομα είναι συμβολοσειρά, όχι UID.** Η `getpwnam(0)` μετατρέπει το `0` σε συμβολοσειρά και αναζητά τον κυριολεκτικό χρήστη ονόματι `"0"`, που σχεδόν ποτέ δεν υπάρχει. Χρησιμοποιήστε την `getpwuid` για αριθμητικές αναζητήσεις.
- **Μεγάλοι κατάλογοι (LDAP / NIS / SSSD).** Η `getpwnam` περνά από το NSS, οπότε το κόστος κλιμακώνεται όπως απαντά ο τοπικός αναλυτής. Μια αστοχία έναντι ενός λάθος ρυθμισμένου διακομιστή LDAP μπορεί να μπλοκάρει για δευτερόλεπτα. Κάντε cache τα αποτελέσματα σε πίνακα κατακερματισμού αν αναζητάτε πολλούς χρήστες σε στενό βρόχο.
- **Threads.** Το upstream σημειώνει ότι η `getpwnam_r` χρησιμοποιείται αυτόματα όπου η πλατφόρμα την παρέχει· η επανάληψη `getpwent`, αντιθέτως, παραμένει ανά διεργασία. Όχι ανησυχία υπό pperl (χωρίς ithreads), αλλά αξίζει να το γνωρίζετε αν μοιράζεστε κώδικα με την perl5.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `getpwuid` - ίδια εγγραφή, αναζήτηση κατά αριθμητικό UID· συνδυάστε με τη `getpwnam` όταν χρειάζεται να πάτε UID → όνομα
- `getpwent` - επανάληψη όλης της βάσης passwd· χρησιμοποιήστε όταν χρειάζεστε κάθε χρήστη, όχι κάποιον συγκεκριμένο
- `endpwent` - κλείσιμο του επαναλήπτη passwd μετά από `getpwent`
- `getgrnam` - αντίστοιχο της βάσης ομάδων· χρειάζεται για να αναλυθεί το πεδίο `$gid` αυτής της εγγραφής πίσω σε όνομα ομάδας
- `getlogin` - φθηνή αναζήτηση «ποιος είναι σε αυτό το tty»· επιστροφή στις `getpwnam`/`getpwuid` όταν επιστρέφει κενή
- `User::pwent` - περιτύλιγμα αντικειμένου με ονομασμένους accessors· καταφύγετε σε αυτό όταν θέλετε `->uid` αντί να μετράτε θέσεις λίστας

Πληροφορίες χρηστών και ομάδων

getpwuid

Αναζητά την εγγραφή passwd ενός χρήστη βάσει αριθμητικού UID.

Η `getpwuid` περιτυλίγει την κλήση `getpwuid(3)` της βιβλιοθήκης C του συστήματος. Περάστε ένα αριθμητικό ID χρήστη όπως `0` ή `$<`· λαμβάνετε πίσω την εγγραφή passwd για εκείνον τον χρήστη, ή κενή λίστα / `undef` αν δεν υπάρχει τέτοιο UID. Η μορφή επιστροφής εξαρτάται από το περιβάλλον: το περιβάλλον λίστας δίνει κάθε πεδίο της εγγραφής· το βαθμωτό περιβάλλον δίνει μια μοναδική χρήσιμη τιμή - για την `getpwuid` συγκεκριμένα, το όνομα σύνδεσης (login).

Σύνοψη

```
getpwuid UID
```

Τι επιστρέφεται

Περιβάλλον λίστας επιστρέφει μια λίστα 10 στοιχείων που αντικατοπτρίζει το `struct passwd` εμπλουτισμένο με τα BSD επιπλέον πεδία - η ίδια μορφή που επιστρέφει η `getpwnam`:

```
my ($name,    $passwd, $uid,  $gid,   $quota,
    $comment, $gcos,   $dir,  $shell, $expire) = getpwuid($uid);
```

- `$name` - όνομα σύνδεσης (login)
- `$passwd` - κρυπτογραφημένος κωδικός, ή `"x"` / `"*"` σε συστήματα shadow-password. Μολυσμένο (tainted) υπό `-T`.
- `$uid` - αριθμητικό ID χρήστη (ο ίδιος αριθμός που περάσατε)
- `$gid` - αριθμητικό ID πρωτεύουσας ομάδας
- `$quota` - όριο δίσκου, ή `$change` / `$age` σε ορισμένα συστήματα. Κενή συμβολοσειρά όταν δεν υποστηρίζεται.
- `$comment` - διαχειριστικό σχόλιο, ενίοτε `$class`. Κενή συμβολοσειρά όταν δεν υποστηρίζεται.
- `$gcos` - πεδίο GECOS (συνήθως το πραγματικό όνομα συν άλλες πληροφορίες χρήστη). Μολυσμένο υπό `-T` επειδή οι χρήστες μπορούν να το επεξεργαστούν.
- `$dir` - αρχικός κατάλογος (home)
- `$shell` - κέλυφος σύνδεσης (login shell). Μολυσμένο υπό `-T`.
- `$expire` - λήξη λογαριασμού / κωδικού, όταν υποστηρίζεται.

Το βαθμωτό περιβάλλον επιστρέφει μόνο το `$name` (το όνομα σύνδεσης). Για την `getpwuid` - μια αναζήτηση *βάσει UID* - το βαθμωτό περιβάλλον σάς δίνει «το άλλο πράγμα», σύμφωνα με τον κανόνα που το upstream τεκμηριώνει για όλες τις συναρτήσεις `getpw*` / `getgr*` / `gethost*` / `getnet*` / `getproto*` / `getserv*`.

Σε αποτυχία αναζήτησης και τα δύο περιβάλλοντα επιστρέφουν την κενή λίστα / `undef`. Ελέγχετε πάντα την επιστροφή πριν την αποπακετάρετε.

Καθολική κατάσταση που επηρεάζει

- Ορίζει το `$!` σε αποτυχία κλήσης συστήματος (σπάνιο - η «εύρεση μηδενικού UID» δεν είναι αποτυχία, είναι κενή επιστροφή).
- Διαβάζει τα `$<` και `$>` έμμεσα μόνο όταν τα περάσετε ως όρισμα, αλλά το κοινό ιδίωμα `getpwuid($<)` είναι αυτό που μετατρέπει την ειδική μεταβλητή του πραγματικού UID σε χρήσιμη εγγραφή.
- Υπό `-T` (taint mode), τα `$passwd`, `$gcos` και `$shell` επιστρέφουν μολυσμένα (tainted). Δείτε `perlsec` για το πώς να τα ξεπλύνετε.
- Μοιράζεται την ίδια υποκείμενη κατάσταση επαναλήπτη με τις `getpwent`, `setpwent` και `endpwent`. Μια κλήση `getpwuid` **δεν** διαταράσσει έναν εν εξελίξει βρόχο `getpwent` στη glibc, αλλά ο φορητός κώδικας δεν θα έπρεπε να βασίζεται σε αυτό.

Παραδείγματα

Βασική αναζήτηση UID-προς-όνομα σε βαθμωτό περιβάλλον:

```
my $name = getpwuid(0);              # "root"
```

«Ως ποιος εκτελούμαι» - το καθιερωμένο ιδίωμα της `getpwuid`, που επιλύει το πραγματικό UID σε όνομα σύνδεσης:

```
my $me = getpwuid($<);               # e.g. "alice"
```

Συνδυάστε με την `getlogin` για μια αξιόπιστη αλυσίδα εφεδρείας που επιβιώνει σε αποσυνδεδεμένες διεργασίες και συνεδρίες `su`:

```
my $login = getlogin() || getpwuid($<) || "unknown";
```

Η `getlogin` διαβάζει την εγγραφή του ελέγχοντος tty και μπορεί να ψεύδεται (ή να επιστρέψει κενό) όταν δεν υπάρχει tty· η `getpwuid($<)` απαντά από το καταγεγραμμένο στον πυρήνα UID και λειτουργεί πάντα.

Πλήρης εγγραφή σε περιβάλλον λίστας, με αμυντικό έλεγχο:

```
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";
```

Από UID → όνομα ομάδας μέσω του πεδίου `$gid`:

```
my $gid      = (getpwuid($<))[3];
my $groupnam = getgrgid($gid);
```

Αντικειμενοστρεφής προσπέλαση μέσω της παράκαμψης `User::pwent`, χρήσιμη όταν θέλετε ονομασμένες μεθόδους προσπέλασης αντί να μετράτε θέσεις λίστας:

```
use User::pwent;
my $pw = getpwuid($<);
printf "%s lives in %s, runs %s\n", $pw->name, $pw->dir, $pw->shell;
```

Το `use User::pwent` αντικαθιστά την `getpwuid` με μια έκδοση που επιστρέφει ένα blessed αντικείμενο με τις μεθόδους προσπέλασης `->name`, `->uid`, `->gid`, `->dir`, `->shell`. Οι δύο μορφές δεν αναμιγνύονται: μέσα σε μία εμβέλεια χρησιμοποιείτε είτε την παράκαμψη είτε την ενσωματωμένη, όχι και τις δύο.

Οριακές περιπτώσεις

- **Ανύπαρκτο UID** επιστρέφει κενή λίστα σε περιβάλλον λίστας και `undef` σε βαθμωτό περιβάλλον. Αυτό δεν είναι σφάλμα - το `$!` δεν ορίζεται. Διακρίνετε το «UID λείπει» από το «η αναζήτηση απέτυχε» ελέγχοντας το `$!` μόνο όταν θέλετε να αναφέρετε μη αναμενόμενη αποτυχία, όχι όταν αποφασίζετε αν υπάρχει το UID.
- **Το `$passwd` είναι σπάνια χρήσιμο.** Σε σύγχρονα συστήματα Linux και BSD, το πραγματικό hash ζει στο `/etc/shadow`, αναγνώσιμο μόνο από τον root. Εδώ λαμβάνετε `"x"` ή `"*"`. Χρησιμοποιήστε PAM ή το άρθρωμα `Authen::PAM` για πιστοποίηση, αντί να συγκρίνετε με αυτό το πεδίο.
- **Φορητότητα των `$quota`, `$comment`, `$expire`.** Αυτά τα πεδία είναι κενά ή χωρίς νόημα σε Linux. Το upstream POD παραθέτει τιμές `Config` (`d_pwquota`, `d_pwage`, `d_pwchange`, `d_pwcomment`, `d_pwexpire`) που μπορείτε να ελέγξετε αν χρειάζεται να γνωρίζετε σε χρόνο εκτέλεσης αν η πλατφόρμα τα συμπληρώνει.
- **Μολυσμένα πεδία υπό `-T`.** Τα `$passwd`, `$gcos`, `$shell` είναι μολυσμένα επειδή οι χρήστες μπορούν να τα τροποποιήσουν (μέσω `chsh`, `chfn`, ή απευθείας επεξεργασίας του `/etc/passwd` σε προνομιακά πλαίσια). Περάστε τα μέσα από μια σύλληψη regex για ξέπλυμα προτού τα χρησιμοποιήσετε σε κλήση συστήματος.
- **Η βαθμωτή επιστροφή για UID 0 δεν είναι ποτέ ψευδής.** Σε αντίθεση με την `getpwnam`, της οποίας η βαθμωτή επιστροφή για τον `root` είναι το UID `0` (ψευδής τιμή), η βαθμωτή επιστροφή της `getpwuid` είναι το *όνομα* `"root"` - πάντοτε αληθής συμβολοσειρά για υπαρκτούς χρήστες. Έτσι, το `if (getpwuid($uid)) { ... }` λειτουργεί όπως θα αναμένατε, αλλά καταφύγετε στο `defined` ούτως ή άλλως αν σας ενδιαφέρει η διάκριση μεταξύ «λείπει» και «κενό όνομα»:

```
if (defined getpwuid($uid))      { ... }   # unambiguous
if (my @pw = getpwuid($uid))     { ... }   # also correct
```

- **Το UID είναι αριθμός, όχι όνομα.** Η `getpwuid("root")` μετατρέπει το `"root"` σε συμβολοσειρά αριθμού `0` (με προειδοποίηση υπό `use warnings`) και αναζητά το UID `0`. Χρησιμοποιήστε `getpwnam` για αναζητήσεις βάσει ονόματος.
- **Αρνητικά / υπερβολικά μεγάλα UID.** Η κλήση συστήματος δέχεται ό,τι δέχεται ο τύπος `uid_t` στην πλατφόρμα (συνήθως 32-bit μη προσημασμένος). Η Perl μετατρέπει τις αρνητικές τιμές μέσω συμπληρώματος ως προς δύο κατά την είσοδο· το αποτέλεσμα εξαρτάται από την πλατφόρμα και σχεδόν ποτέ δεν είναι αυτό που θέλετε.

- **Μεγάλοι κατάλογοι (LDAP / NIS / SSSD).** Η `getpwuid` περνά μέσα από το NSS, οπότε το κόστος κλιμακώνεται ανάλογα με το πώς απαντά ο τοπικός resolver. Μια αποτυχημένη αναζήτηση σε λανθασμένα διαμορφωμένο διακομιστή LDAP μπορεί να μπλοκάρει για δευτερόλεπτα. Αποθηκεύστε τα αποτελέσματα σε hash αν επιλύετε πολλά UID σε στενό βρόχο (π.χ. σχολιάζοντας έξοδο τύπου `ls -l`).
- **Νήματα.** Το upstream σημειώνει ότι η `getpwuid_r` χρησιμοποιείται αυτόματα όπου την παρέχει η πλατφόρμα· η επανάληψη μέσω `getpwent`, αντιθέτως, παραμένει ανά διεργασία. Δεν είναι έγνοια στην pperl (χωρίς ithreads), αλλά αξίζει να γνωρίζετε αν μοιράζεστε κώδικα με την perl5.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `getpwnam` - η ίδια εγγραφή, αναζητούμενη βάσει ονόματος σύνδεσης· συνδυάστε με την `getpwuid` όταν χρειάζεται να μεταβείτε από όνομα → UID
- `getpwent` - επαναλάβετε σε ολόκληρη τη βάση passwd· χρησιμοποιήστε όταν χρειάζεστε κάθε χρήστη, όχι κάποιον συγκεκριμένο
- `endpwent` - κλείστε τον επαναλήπτη passwd μετά την `getpwent`
- `getgrgid` - αντίστοιχη συνάρτηση για τη βάση ομάδων· χρειάζεται για την επίλυση του πεδίου `$gid` από αυτή την εγγραφή σε όνομα ομάδας
- `getlogin` - φθηνή αναζήτηση «ποιος βρίσκεται σε αυτό το tty»· επανέλθετε στην `getpwuid($<)` όταν επιστρέφει κενό
- `User::pwent` - αντικειμενοστρεφές περιτύλιγμα με ονομασμένες μεθόδους προσπέλασης· κατφύγετε σε αυτό όταν θέλετε `->name` αντί να μετράτε θέσεις λίστας

Πληροφορίες δικτύου

getservbyname

Αναζητά μια υπηρεσία δικτύου με βάση το κειμενικό όνομα και το πρωτόκολλό της.

Η `getservbyname` συμβουλεύεται τη βάση δεδομένων υπηρεσιών του συστήματος (τυπικά `/etc/services`, ή ό,τι το `nsswitch.conf` δρομολογεί ως πηγή `services`) και επιστρέφει την εγγραφή της οποίας το όνομα ή το ψευδώνυμο ταιριάζει με το `NAME` υπό το πρωτόκολλο μεταφοράς που κατονομάζει το `PROTO`. Είναι το Perl wrapper γύρω από την κλήση `getservbyname(3)` της βιβλιοθήκης C, και είναι ο τρόπος για να μετατρέψετε το `"https"` σε 443 χωρίς να κωδικοποιήσετε σταθερά τον αριθμό.

Και τα δύο ορίσματα είναι συμβολοσειρές. Το `NAME` είναι το όνομα υπηρεσίας ή ψευδώνυμο (`"http"`, `"ssh"`, `"imaps"`)· το `PROTO` είναι το όνομα πρωτοκόλλου (`"tcp"`, `"udp"`, `"sctp"`) - **όχι** αριθμός, και **όχι** σταθερά τύπου `SOCK_STREAM`.

Σύνοψη

```
my $port                            = getservbyname NAME, PROTO;   #␣
↪scalar
my ($name, $aliases, $port, $proto) = getservbyname NAME, PROTO;   #␣
↪list
```

Τι επιστρέφεται

Σε **βαθμωτό περιβάλλον**, ο αριθμός θύρας ως απλός ακέραιος σε host byte order - έτοιμος για τροφοδοσία στην `pack` με `"n"` ή σε έναν κατασκευαστή `sockaddr_in`. Αν δεν ταιριάζει καμία εγγραφή, η επιστροφή είναι `undef`.

Σε **περιβάλλον λίστας**, τα τέσσερα πεδία της εγγραφής `servent`:

- `$name` - κανονικό όνομα υπηρεσίας (το πρωτεύον όνομα, όχι το ψευδώνυμο με το οποίο αναζητήσατε).
- `$aliases` - συμβολοσειρά εναλλακτικών ονομάτων διαχωρισμένη με κενά.
- `$port` - αριθμός θύρας, host byte order.
- `$proto` - κανονικό όνομα πρωτοκόλλου όπως επιστρέφεται από τον resolver, κανονικά ίσο με το `PROTO` που περάσατε.

Αν δεν ταιριάζει καμία εγγραφή σε περιβάλλον λίστας, η επιστροφή είναι η κενή λίστα.

Η θύρα είναι σε **host byte order**. Τα APIs δικτύου περιμένουν network byte order, οπότε περάστε την μέσω `pack` με το πρότυπο `"n"` - ή αφήστε το `Socket` να το κάνει μέσω της `sockaddr_in`.

Καθολική κατάσταση που επηρεάζει

- Διατηρεί ανοιχτό handle στη βάση δεδομένων υπηρεσιών μεταξύ κλήσεων όταν η βάση έχει ανοίξει με `setservent` χρησιμοποιώντας μη-μηδενικό STAYOPEN. Η `endservent` την κλείνει.
- Θέτει το `$!` σε αποτυχίες αναζήτησης που προέρχονται από σφάλμα σε επίπεδο συστήματος (αρχείο βάσης δεδομένων που λείπει, μη προσπελάσιμο backend NSS). Ένα απλό «δεν υπάρχει τέτοια υπηρεσία» αναφέρεται με επιστροφή `undef` / κενής λίστας, όχι θέτοντας το `$!`.

Παραδείγματα

Επιλύστε μια γνωστή υπηρεσία στη θύρα της:

```
my $port = getservbyname("https", "tcp");
print $port, "\n";                      # 443
```

Πλήρης εγγραφή σε περιβάλλον λίστας:

```
my ($name, $aliases, $port, $proto) = getservbyname("http", "tcp");
print "$name ($aliases) -> $port/$proto\n";
                                        # www (www-http http) -> 80/
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
↪tcp
                                        # (exact aliases depend on /
↪etc/services)
```

Κατασκευάστε ένα sockaddr_in για την connect, μετατρέποντας τη θύρα από host order σε network order με 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: $!";
```

Διακρίνετε το «δεν βρέθηκε» από το «λάθος πρωτόκολλο» - μια υπηρεσία που υπάρχει υπό tcp αλλά όχι udp επιστρέφει undef όταν το πρωτόκολλο δεν ταιριάζει:

```
defined(getservbyname("https", "tcp")) or die "expected 443";
defined(getservbyname("https", "udp")) or warn "no https/udp entry";
```

Αναζήτηση με ψευδώνυμο - τα ψευδώνυμα επιλύονται το ίδιο με το κανονικό όνομα, αλλά το επιστρεφόμενο $name είναι πάντα το κανονικό:

```
my ($name) = getservbyname("www", "tcp");
print $name, "\n";                      # "http" on most systems
```

Οριακές περιπτώσεις

- **Άγνωστη υπηρεσία**: επιστρέφει undef σε βαθμωτό περιβάλλον, την κενή λίστα σε περιβάλλον λίστας. Το $! **δεν** τίθεται για απλή αστοχία - μόνο για υποκείμενα σφάλματα συστήματος.
- **Άγνωστο πρωτόκολλο**: το ίδιο με άγνωστη υπηρεσία. Η getservbyname("ssh", "xyzzy") δεν επιστρέφει τίποτα· δεν εγείρεται ξεχωριστό σφάλμα.
- **Διάκριση πεζών-κεφαλαίων**: τα ονόματα υπηρεσίας και πρωτοκόλλου συγκρίνονται χωρίς διάκριση πεζών-κεφαλαίων από τον resolver του συστήματος σε κάθε πλατφόρμα που υποστηρίζουμε· τα "HTTPS" και "https" δίνουν την ίδια απάντηση.
- **Η θύρα είναι σε host byte order**: το να την περάσετε απευθείας σε μορφή επί της γραμμής είναι σφάλμα. Χρησιμοποιήστε pack("n", $port) ή sockaddr_in($port, $addr) - και τα δύο χειρίζονται τη μετατροπή.
- **Τα NAME και PROTO είναι και τα δύο απαιτούμενα**. Δεν υπάρχει μορφή με ένα όρισμα· η παράλειψη του PROTO είναι σφάλμα κατά τη μεταγλώττιση.
- **Παγίδα προτεραιότητας**: όπως κάθε υποψήφιο ονομασμένου μοναδιαίου τελεστή σε αυτή την οικογένεια, η getservbyname είναι τελεστής λίστας. Βάλτε παρενθέσεις όταν συνδυάζετε με τελεστές:

```
getservbyname "http", "tcp" == 80      # WRONG: parses as
                                       # getservbyname("http", (
 ↪"tcp" == 80))
getservbyname("http", "tcp") == 80     # right
```

- **Threads**: σε συστήματα με `getservbyname_r(3)`, η Perl χρησιμοποιεί διαφανώς την επανεισερχόμενη παραλλαγή. Στις λίγες πλατφόρμες που δεν τη διαθέτουν, ταυτόχρονες αναζητήσεις από πολλαπλά threads μπορούν να αλλοιώσουν τα αποτελέσματα η μία της άλλης.
- **Διεπαφή ανά όνομα**: το `Net::servent` (core) περιτυλίγει αυτή την κλήση και επιστρέφει αντικείμενο με ονομασμένες προσπελάσεις, με κόστος μίας εκχώρησης ανά αναζήτηση.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `getservbyport` - αντίστροφη αναζήτηση: θύρα + πρωτόκολλο → εγγραφή υπηρεσίας
- `getservent` - διασχίστε κάθε εγγραφή στη βάση δεδομένων υπηρεσιών, μία εγγραφή τη φορά
- `setservent` - επαναφέρετε στην αρχή (ή ανοίξτε μόνιμα) τη βάση δεδομένων υπηρεσιών πριν από μια διάσχιση
- `endservent` - κλείστε τη βάση δεδομένων υπηρεσιών που άνοιξε η `setservent`
- `getprotobyname` - συμπληρωματική αναζήτηση για το όρισμα `PROTO` όταν χρειάζεστε επίσης τον αριθμητικό αριθμό πρωτοκόλλου του
- `Socket` - οι `sockaddr_in` και `pack_sockaddr_in` καταναλώνουν τη θύρα σε host order που επιστρέφεται εδώ

Πληροφορίες δικτύου

getservbyport

Αναζήτηση μιας υπηρεσίας δικτύου από την αριθμητική θύρα και το πρωτόκολλό της.

Η `getservbyport` συμβουλεύεται τη βάση δεδομένων υπηρεσιών του συστήματος (`/etc/services` σε Linux, συν όποιες πηγές NSS είναι ρυθμισμένες) και επιστρέφει την εγγραφή για την υπηρεσία που ακούει στη `PORT` υπό το πρωτόκολλο `PROTO`. Η `PORT` είναι ένας ακέραιος **σε network byte order** - η τιμή που θα περνούσατε στην `bind` ή θα λαμβάνατε από το `sockaddr_in`, όχι το δεκαδικό που εμφανίζεται στο `/etc/services`. Το `PROTO` είναι μια συμβολοσειρά με πεζά όπως `"tcp"` ή `"udp"`. Η κλήση περιτυλίγει την `getservbyport(3)` της βιβλιοθήκης C.

Σύνοψη

```
getservbyport PORT, PROTO          # list context: full record
scalar getservbyport PORT, PROTO   # scalar context: service name
```

Τι επιστρέφεται

Σε **περιβάλλον λίστας**, μια λίστα πέντε στοιχείων που περιγράφει την υπηρεσία:

```
my ($name, $aliases, $port, $proto) = getservbyport($port_n, "tcp");
```

Δείκτης	Όνομα	Σημασία
0	`$name`	Κανονικό όνομα υπηρεσίας, π.χ. `"http"`
1	`$aliases`	Λίστα ψευδωνύμων χωρισμένη με κενά, π.χ. `"www www-http"`
2	`$port`	Αριθμός θύρας σε **network byte order**
3	`$proto`	Όνομα πρωτοκόλλου, π.χ. `"tcp"`

Αν η εγγραφή δεν υπάρχει η λίστα είναι κενή, οπότε μια εκχώρηση σε περιβάλλον λίστας αφήνει κάθε στόχο `undef`. Μια σκέτη κλήση σε περιβάλλον λίστας επιστρέφει μια μοναδική άνευ νοήματος αληθή τιμή σε αστοχία - εκχωρείτε πάντα σε λίστα και ελέγχετε το πρώτο στοιχείο, ή καλείτε σε βαθμωτό περιβάλλον.

Σε **βαθμωτό περιβάλλον**, παίρνετε το κανονικό όνομα υπηρεσίας (το ίδιο `$name` όπως παραπάνω), ή `undef` αν καμία υπηρεσία δεν είναι καταχωρημένη σε αυτή τη θύρα για αυτό το πρωτόκολλο:

```
my $name = getservbyport($port_n, "tcp")
    or warn "port $port not registered for tcp";
```

Καθολική κατάσταση που επηρεάζει

- Η `$!` **δεν** τίθεται αξιόπιστα σε αστοχία - η συνάρτηση C `getservbyport(3)` δεν έχει τυποποιημένο συμβόλαιο errno για «δεν βρέθηκε». Αντιμετωπίστε μια επιστροφή κενής λίστας ή `undef` ως το μόνο αξιόπιστο σήμα αποτυχίας.
- Η κατάσταση επανάληψης που κρατά η βάση δεδομένων υπηρεσιών (κοινόχρηστη με τις `getservent`, `setservent` και `endservent`) είναι καθολική σε επίπεδο διεργασίας και δεν είναι ασφαλής για threads.

Παγίδα byte-order

Η PORT πρέπει να είναι σε network byte order. Η στήλη της θύρας στο `/etc/services` είναι ανθρώπινο δεκαδικό· η τιμή που χρησιμοποιεί ο πυρήνας και η βιβλιοθήκη C είναι η big-endian 16-bit κωδικοποίηση. Μετατρέψτε με `pack` / `unpack` ή με τους βοηθούς `Socket`:

```
use Socket;

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
# 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");
```

Το πέρασμα ενός ακεραίου σε host-order όπως `80` λειτουργεί σε υλικό little-endian μόνο κατά λάθος για χαμηλές θύρες των οποίων η κωδικοποίηση τυχαίνει να είναι μη μηδενική στο χαμηλό byte - οι περισσότερες θύρες απλώς θα αστοχήσουν. Αυτός είναι ο πιο κοινός λόγος που μια κλήση `getservbyport` «επιστρέφει σιωπηρά τίποτα».

Παραδείγματα

Αναγνώριση της υπηρεσίας στη θύρα 443 μέσω TCP:

```
use Socket;
my $port_n = pack("n", 443);
my ($name) = getservbyport($port_n, "tcp");
print $name, "\n";                      # https
```

Συνδυασμός με `getpeername` για ονομασία της θύρας μιας εισερχόμενης σύνδεσης:

```
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";
```

Πλήρης εκτύπωση εγγραφής - χρήσιμη όταν θέλετε το πρωτόκολλο ή τα ψευδώνυμα:

```
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;
}
```

Βαθμωτό περιβάλλον, στυλ ρήτρας φύλακα:

```
my $svc = getservbyport(pack("n", $port), "tcp")
    or die "no TCP service registered on port $port";
```

Επανάληψη σε κάθε καταχωρημένη υπηρεσία (η υποκείμενη βάση δεδομένων είναι κοινόχρηστη με την `getservent`):

```
while (my ($n, $a, $p, $proto) = getservent()) {
    next unless $proto eq "tcp";
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    # …do something with $n / $p…
}
endservent();
```

Οριακές περιπτώσεις

- **Εγγραφή που λείπει**: κενή λίστα σε περιβάλλον λίστας, `undef` σε βαθμωτό περιβάλλον. Ένα σκέτο `if (getservbyport(...))` σε περιβάλλον λίστας είναι αληθές ακόμη και για εγγραφή που λείπει λόγω της παλαιάς επιστροφής «μοναδικής άνευ νοήματος αληθούς τιμής» - εκχωρείτε πάντα σε λίστα ή εξαναγκάζετε σε βαθμωτό περιβάλλον.
- **Άγνωστο πρωτόκολλο**: μια μη αναγνωρισμένη συμβολοσειρά `PROTO` (π.χ. `"sctp"` σε έναν υπολογιστή του οποίου η βάση δεδομένων δεν την περιλαμβάνει) αντιμετωπίζεται από τη βιβλιοθήκη C ως μη-αντιστοιχία. Η κλήση επιστρέφει κενό/undef, όχι σφάλμα.
- **Πεζά/κεφαλαία του `PROTO`**: ευαίσθητο σε πεζά/κεφαλαία στις περισσότερες υλοποιήσεις libc Linux. Χρησιμοποιήστε πεζά `"tcp"` / `"udp"`· το `"TCP"` θα αστοχήσει.
- **Θύρα σε host-order**: το πέρασμα `80` αντί για `pack("n", 80)` είναι το κλασικό σφάλμα - δείτε *Παγίδα byte-order* παραπάνω.
- **Αριθμητική έναντι συμβολοσειράς `PORT`**: η Perl μετατρέπει αριθμητικά την `PORT` πριν την περάσει στη libc. Μια πακεταρισμένη συμβολοσειρά δύο bytes λειτουργεί επειδή η `pack` με `"n"` παράγει bytes που μετατρέπονται αριθμητικά ντετερμινιστικά στις υποστηριζόμενες πλατφόρμες· αν υπάρχει αμφιβολία, `unpack("n", $packed)` πρώτα και περάστε τον ακέραιο ώστε η πρόθεση να μείνει ρητή.
- **Προτεραιότητα με τελεστές λίστας**: όπως άλλοι ονομασμένοι μοναδιαίοι / λίστας τελεστές, η `getservbyport` συνδέει χαλαρά. Βάλτε σε παρενθέσεις όταν το αποτέλεσμα τροφοδοτεί άλλον τελεστή:

```
getservbyport $port_n, "tcp" eq "https"     # WRONG
(getservbyport($port_n, "tcp")) eq "https"  # correct
```

- **Threads**: η υποκείμενη αναζήτηση μοιράζεται κατάσταση με τις `getservent` / `setservent` / `endservent`. Δύο threads που επαναλαμβάνουν ταυτόχρονα θα διαπλέξουν εγγραφές. Χρησιμοποιήστε thread-local περιτυλίγματα όταν αυτό έχει σημασία.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `getservbyname` - αντίστροφη αναζήτηση, όνομα υπηρεσίας σε θύρα· καταφύγετε σε αυτή όταν ξέρετε το όνομα και θέλετε τον αριθμό
- `getservent` - επανάληψη σε κάθε εγγραφή υπηρεσίας· χρησιμοποιήστε όταν χρειάζεστε πλήρη σάρωση παρά μία μεμονωμένη αναζήτηση
- `setservent` - επαναφορά του επαναλήπτη υπηρεσιών, προαιρετική σημαία `STAYOPEN` για να κρατά το αρχείο ανοιχτό μέσω `fork`
- `endservent` - κλείσιμο της βάσης δεδομένων υπηρεσιών μετά από μια επανάληψη
- `getprotobynumber` - η αντίστοιχη αναζήτηση από την πλευρά πρωτοκόλλου· συχνά χρησιμοποιείται αμέσως μετά τη `getservbyport` για να μεταφραστεί το πεδίο `$proto`
- `pack` - κατασκευή της τιμής θύρας σε network-order που η `getservbyport` αναμένει ως πρώτο όρισμα
- `Socket` - σταθερές και βοηθητικές `sockaddr_in` που παράγουν θύρες σε network-order απευθείας από δομές πυρήνα

Πληροφορίες δικτύου

getservent

Διαβάζει την επόμενη εγγραφή από τη βάση δεδομένων υπηρεσιών του συστήματος.

Η `getservent` διασχίζει τη βάση δεδομένων `/etc/services` μία εγγραφή τη φορά, αποδίδοντας την ίδια λίστα πεδίων όπως οι `getservbyname` και `getservbyport` αλλά χωρίς κλειδί αναζήτησης. Κάθε κλήση προωθεί έναν εσωτερικό δρομέα· ο δρομέας ανοίγεται στην πρώτη κλήση, επαναφέρεται στην αρχή από την `setservent` και απελευθερώνεται από την `endservent`. Αντικατοπτρίζει τη ρουτίνα `getservent(3)` της C library του POSIX και κληρονομεί κάθε ιδιαιτερότητα φορητότητας που αυτό συνεπάγεται.

Σύνοψη

```
my ($name, $aliases, $port, $proto) = getservent;
my $name = getservent;        # scalar context
```

Τι επιστρέφεται

Σε περιβάλλον λίστας, τέσσερα πεδία που ταιριάζουν με τη σειρά που τεκμηριώνεται για κάθε παραλλαγή `getserv*`:

- `$name` - κανονικό όνομα υπηρεσίας, π.χ. `"http"`.
- `$aliases` - συμβολοσειρά εναλλακτικών ονομάτων διαχωρισμένων με κενά, π.χ. `"www www-http"`. Δεν είναι πίνακας· διασπάστε την εσείς αν χρειάζεστε πίνακα.
- `$port` - αριθμός θύρας ως εγγενής ακέραιος (σε σειρά bytes host), π.χ. `80`.
- `$proto` - πρωτόκολλο μεταφοράς, τυπικά `"tcp"` ή `"udp"`.

Σε βαθμωτό περιβάλλον, μόνο το `$name`. Όταν ο δρομέας φτάσει στο τέλος αρχείου, η `getservent` επιστρέφει την κενή λίστα σε περιβάλλον λίστας και `undef` σε βαθμωτό περιβάλλον - ο ιδιωματικός βρόχος `while (my @e = getservent)` στηρίζεται σε αυτό.

Για να μετατρέψετε τα ψευδώνυμα σε λίστα:

```
my @alt = split / /, $aliases;
```

Καθολική κατάσταση που επηρεάζει

- **Εσωτερικός δρομέας βάσης δεδομένων** για το `/etc/services` - μοιράζεται με τις `getservbyname`, `getservbyport`, `setservent` και `endservent`. Οποιαδήποτε αναζήτηση κατά όνομα ή κατά θύρα σε κάποιες πλατφόρμες επαναφέρει ή κλείνει αυτόν τον δρομέα ως παρενέργεια· επαναλάβετε σε σφιχτό βρόχο ή περιτυλίξτε την επανάληψη σε `setservent` / `endservent` για ασφάλεια.
- `$!` - τίθεται όταν η υποκείμενη κλήση C αναφέρει σφάλμα.
- Δεν είναι ασφαλής για threads. Η `getservent` κρατάει κατάσταση ανά διεργασία, όχι ανά thread· δύο threads που επαναλαμβάνουν ταυτόχρονα θα αλληλομπλέξουν εγγραφές και κανένα δεν θα δει την πλήρη λίστα.

Παραδείγματα

Διασχίστε ολόκληρη τη βάση δεδομένων υπηρεσιών και εκτυπώστε κάθε εγγραφή:

```
while (my ($name, $aliases, $port, $proto) = getservent) {
    print "$name/$proto $port\n";
}
endservent;
```

Επαναφέρετε τον δρομέα πριν την επανάληψη ώστε προηγούμενες κλήσεις αλλού στο πρόγραμμα να μην τον αφήσουν στη μέση της ροής:

```
setservent(0);
while (my @e = getservent) {
    # @e is ($name, $aliases, $port, $proto)
}
endservent;
```

Συλλέξτε μόνο υπηρεσίες TCP σε πίνακα κατακερματισμού με κλειδί τη θύρα:

```
my %tcp;
while (my ($name, undef, $port, $proto) = getservent) {
    $tcp{$port} = $name if $proto eq 'tcp';
}
endservent;
```

Χρησιμοποιήστε την αντικειμενοστρεφή διεπαφή κατά όνομα από την `Net::servent`, η οποία παρακάμπτει την ενσωματωμένη και επιστρέφει blessed αντικείμενο με μεθόδους προσπέλασης:

```
use Net::servent;
while (my $s = getservent) {
    printf "%-20s %5d/%s\n", $s->name, $s->port, $s->proto;
}
```

Οριακές περιπτώσεις

- **Το τέλος αρχείου επιστρέφει κενό / undef, όχι σφάλμα.** Μια ψευδής τιμή ενός βαθμωτού σε περιβάλλον λίστας (το `my @e = getservent` δίνοντας κενή λίστα) είναι ο τερματιστής. Μην ελέγχετε την `$!` για να εντοπίσετε τέλος επανάληψης.
- **Απόν `/etc/services`**: σε συστήματα χωρίς το αρχείο, η πρώτη κλήση επιστρέφει αμέσως την κενή λίστα και η `$!` τίθεται. Τίποτα στην κλήση δεν σηματοδοτεί τη διαφορά μεταξύ κενής και απούσας βάσης δεδομένων - επιθεωρήστε την `$!` πριν υποθέσετε ότι η βάση ήταν απλώς κενή.
- **Διαμοιρασμός δρομέα με αναζητήσεις κατά όνομα / κατά θύρα**: η κλήση `getservbyname` ή `getservbyport` στη μέση ενός βρόχου `getservent` μπορεί να επαναφέρει τον δρομέα σε κάποιες πλατφόρμες. Είτε ολοκληρώστε πρώτα τη διάσχιση είτε ξανανοίξτε με `setservent`.
- **Το `$aliases` είναι συμβολοσειρά, όχι λίστα**. Το `split / /, $aliases` σας δίνει τη μορφή πίνακα. Η άμεση αντιμετώπιση του `$aliases` ως λίστας σας δίνει σιωπηρά το μήκος λίστας ενός στοιχείου.
- **Προγράμματα με threads**: η επανάληψη είναι ανά διεργασία. Συντονίστε την πρόσβαση με κλείδωμα, ή προτιμήστε `getservbyname` / `getservbyport` για σημειακά ερωτήματα σε κώδικα με threads.
- **Το βαθμωτό περιβάλλον επιστρέφει μόνο το όνομα.** Αν χρειάζεστε τη θύρα, καλέστε σε περιβάλλον λίστας ακόμη και όταν ενδιαφέρεστε για ένα μόνο πεδίο.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `getservbyname` - αναζητήστε μία μόνο υπηρεσία με το κειμενικό της όνομα αντί να επαναλαμβάνετε
- `getservbyport` - αναζητήστε μία μόνο υπηρεσία με την αριθμητική της θύρα
- `setservent` - επαναφέρετε τον κοινό δρομέα πριν την επανάληψη ώστε προηγούμενες κλήσεις να μην τον αφήσουν στη μέση της ροής
- `endservent` - απελευθερώστε τον δρομέα όταν η διάσχιση ολοκληρωθεί, ελευθερώνοντας τον υποκείμενο περιγραφέα αρχείου
- `getprotoent` - ίδιο μοτίβο επαναλήπτη για το `/etc/protocols`, τυπικά συνδυάζεται όταν επιλύεται η πλειάδα `(port, proto)`
- `Net::servent` - αντικειμενοστρεφές περιτύλιγμα του οποίου οι μέθοδοι προσπέλασης απαλείφουν την ανάγκη να θυμάστε τη σειρά των πεδίων

Υποδοχές (sockets)

getsockname

Επιστρέφει την τοπική διεύθυνση μιας συνδεδεμένης ή δεσμευμένης υποδοχής.

Η `getsockname` ρωτά τον πυρήνα σε ποια διεύθυνση και θύρα είναι δεσμευμένη η `SOCKET` σε αυτή την άκρη της σύνδεσης. Το αποτέλεσμα είναι η ίδια πακεταρισμένη δομή sockaddr που ο πυρήνας παραδίδει στην `accept` ή που θα δίνατε στην `bind` - αποπακετάρετέ την με τον κατάλληλο βοηθό από το `Socket` (για παράδειγμα `sockaddr_in` για IPv4, `sockaddr_in6` για IPv6). Χρήσιμη όταν ένας υπολογιστής έχει πολλαπλές διεπαφές και χρειάζεται να ξέρετε από ποια σας προσέγγισε ένας πελάτης, ή όταν ζητήσατε από τον πυρήνα μια εφήμερη θύρα μέσω `bind` με θύρα `0` και τώρα θέλετε να μάθετε τη θύρα που επέλεξε.

Σύνοψη

```
getsockname SOCKET
```

Τι επιστρέφεται

Σε επιτυχία, το πακεταρισμένο sockaddr για την τοπική άκρη της `SOCKET` ως συμβολοσειρά bytes της οποίας η διάταξη ταιριάζει με την οικογένεια διευθύνσεων της υποδοχής. Σε αποτυχία, `undef` με ορισμένη την `$!` (τυπικά `ENOTSOCK` αν η `SOCKET` δεν είναι υποδοχή, ή `EBADF` αν δεν είναι ανοιχτό filehandle).

Η τιμή επιστροφής είναι **αδιαφανή bytes**. Μην την επιθεωρείτε απευθείας - δώστε την στον κατάλληλο για την οικογένεια αποπακεταριστή:

```
use Socket;
my $packed = getsockname($sock) or die "getsockname: $!";
my ($port, $addr) = sockaddr_in($packed);      # IPv4
```

Για κώδικα ανεξάρτητο από οικογένεια διευθύνσεων, καλέστε πρώτα την `sockaddr_family` στο αποτέλεσμα και κάντε dispatch βάσει της επιστρεφόμενης σταθεράς `AF_*`.

Καθολική κατάσταση που επηρεάζει

Καμία. Η `getsockname` είναι ένα καθαρό περιτύλιγμα κλήσης συστήματος - δεν διαβάζει ούτε γράφει την `$_`, εκτός της `$!`, ούτε καμία άλλη καθολική του διερμηνευτή.

Παραδείγματα

Ανακαλύψτε την εφήμερη θύρα που εκχώρησε ο πυρήνας μετά από `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";
```

Εντοπίστε σε ποια τοπική IP σας προσέγγισε ένας συνδεδεμένος πελάτης, σε έναν υπολογιστή με πολλαπλές διεπαφές:

```
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);
```

Παραλλαγή IPv6 - αποπακετάρισμα με sockaddr_in6:

```
use Socket;
my ($port, $addr) = sockaddr_in6(getsockname($sock));
printf "local: [%s]:%d\n", inet_ntop(AF_INET6, $addr), $port;
```

Dispatch ανεξάρτητο από οικογένεια όταν η υποδοχή μπορεί να είναι είτε IPv4 είτε 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) {
    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;
}
```

Οριακές περιπτώσεις

- **Μη δεσμευμένη υποδοχή**: Μια υποδοχή που έχει δημιουργηθεί αλλά δεν έχει ποτέ δεσμευθεί ή συνδεθεί τυπικά επιστρέφει ένα sockaddr με τη διεύθυνση wildcard και θύρα `0`. Δεν είναι σφάλμα - ο πυρήνας απλώς αναφέρει «δεν έχει εκχωρηθεί ακόμη τοπικό όνομα».
- **Δεν είναι υποδοχή**: Αν δοθεί ανοιχτό filehandle που δεν είναι υποδοχή (απλό αρχείο, σωλήνωση), επιστρέφει undef με την `$!` ορισμένη σε `ENOTSOCK`.
- **Κλειστό filehandle**: Επιστρέφει undef με την `$!` ορισμένη σε `EBADF`. Υπό `use warnings` εκπέμπεται προειδοποίηση `getsockname() on closed socket`.
- **Υποδοχές πεδίου Unix**: Η πακεταρισμένη μορφή είναι sockaddr `AF_UNIX`· αποπακετάρετε με unpack_sockaddr_un. Η διαδρομή μπορεί να είναι μικρότερη από αυτήν που δώσατε στην `bind`, και στο Linux μπορεί να αρχίζει με ένα byte NUL για υποδοχές αφηρημένου χώρου ονομάτων.
- **Διεύθυνση wildcard με IPv4**: Μια υποδοχή δεσμευμένη στο `INADDR_ANY` αναφέρει εδώ τη διεύθυνση wildcard (`0.0.0.0`), όχι τη συγκεκριμένη διεύθυνση της διεπαφής που προσέγγισε ο ομότιμος. Για να μάθετε τη διεύθυνση-στόχο του ομοτίμου πρέπει να χρησιμοποιήσετε την getsockname *αφού* ο πυρήνας έχει δρομολογήσει τη σύνδεση (μετά την `accept`), ή να συμβουλευτείτε το `IP_PKTINFO` μέσω getsockopt.

- **Κλήση πριν από `bind`/`connect`**: Η συμπεριφορά εξαρτάται από την πλατφόρμα αλλά είναι αβλαβής. Το Linux επιστρέφει ένα μηδενισμένο sockaddr της οικογένειας της υποδοχής· μερικά BSD επιστρέφουν `undef` με `EINVAL`. Στηριχτείτε σε αυτήν μόνο αφού έχουν ολοκληρωθεί η `bind`, η `connect` ή η `accept`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `getpeername` - η κατοπτρική κλήση: επιστρέφει το sockaddr της **απομακρυσμένης** άκρης για μια συνδεδεμένη υποδοχή
- `bind` - θέτει την τοπική διεύθυνση· η `getsockname` διαβάζει πίσω αυτό που πράγματι δέσμευσε ο πυρήνας, συμπεριλαμβανομένων εφήμερων θυρών που επελέγησαν για εσάς
- `accept` - επιστρέφει το απομακρυσμένο sockaddr απευθείας, οπότε σπάνια χρειάζεστε `getsockname` στην αποδεκτή υποδοχή, εκτός αν ο ακροατής δεσμεύτηκε σε wildcard και θέλετε τη συγκεκριμένη τοπική διεύθυνση
- `socket` - δημιουργεί την υποδοχή της οποίας το τοπικό όνομα αναφέρει η `getsockname`
- `Socket` - το άρθρωμα που παρέχει τους αποπακεταριστές (`sockaddr_in`, `sockaddr_in6`, `unpack_sockaddr_un`, `sockaddr_family`) που εφαρμόζετε στην τιμή επιστροφής
- `getsockopt` - για πληροφορίες διεύθυνσης προορισμού ανά πακέτο (`IP_PKTINFO`) όταν μια υποδοχή δεσμευμένη σε wildcard χρειάζεται να γνωρίζει τη συγκεκριμένη τοπική διεύθυνση ενός εισερχόμενου datagram

Υποδοχές (sockets)

getsockopt

Διαβάζει μία επιλογή υποδοχής από τον πυρήνα ως αδιαφανή πακεταρισμένη συμβολοσειρά.

Η `getsockopt` ζητά από τον πυρήνα την τρέχουσα τιμή της επιλογής με όνομα `OPTNAME` στο επίπεδο πρωτοκόλλου `LEVEL` στην `SOCKET`. Ο πυρήνας δεν επιστρέφει τιμή με τύπο - επιστρέφει έναν ωμό ενταμιευτή bytes του οποίου η διάταξη καθορίζεται από το ζεύγος `(LEVEL, OPTNAME)`. Αποκωδικοποιείτε αυτόν τον ενταμιευτή με `unpack`, συνήθως με το πρότυπο `"i"` ή `"I"`, επειδή οι περισσότερες επιλογές είναι απλοί ακέραιοι.

Σύνοψη

```
getsockopt SOCKET, LEVEL, OPTNAME
```

Τι επιστρέφεται

Μια πακεταρισμένη συμβολοσειρά που περιέχει την απάντηση του πυρήνα σε περίπτωση επιτυχίας, ή undef σε περίπτωση αποτυχίας με το $! ρυθμισμένο στο σφάλμα του συστήματος. Τα περιεχόμενα της συμβολοσειράς είναι **αδιαφανή bytes** - το μήκος και η διάταξή της εξαρτώνται εξ ολοκλήρου από το (LEVEL, OPTNAME):

- Οι επιλογές λογικής τιμής και ακεραίου (SO_KEEPALIVE, SO_REUSEADDR, TCP_NODELAY, ...) επιστρέφουν ως πακεταρισμένο int. Αποκωδικοποιήστε με unpack χρησιμοποιώντας "i" ή "I".
- Τα χρονικά όρια (SO_RCVTIMEO, SO_SNDTIMEO) επιστρέφουν ως πακεταρισμένο struct timeval - δύο long στις περισσότερες πλατφόρμες.
- Οι επιλογές δεσμευμένες σε διεύθυνση (SO_PEERCRED, IP_MULTICAST_IF, structs linger, ...) έχουν τις δικές τους διατάξεις· συμβουλευτείτε το getsockopt(2) και τις σχετικές σελίδες man πρωτοκόλλων για το ακριβές struct.

Πάντα ελέγχετε την τιμή επιστροφής πριν την αποκωδικοποίηση - η κλήση unpack σε undef αποδίδει κενό αποτέλεσμα και κρύβει το πραγματικό σφάλμα που καταγράφεται στο $!.

Καθολική κατάσταση που επηρεάζει

- Το $! τίθεται σε περίπτωση αποτυχίας στην τιμή errno από την υποκείμενη κλήση getsockopt(2). Δεν εκκαθαρίζεται σε περίπτωση επιτυχίας, οπότε ελέγξτε την τιμή επιστροφής, όχι το $! άμεσα.

Παραδείγματα

Έλεγχος αν ο αλγόριθμος του Nagle είναι αυτή τη στιγμή απενεργοποιημένος (TCP_NODELAY ενεργό) σε μια υποδοχή TCP:

```
use Socket qw(IPPROTO_TCP TCP_NODELAY);

my $packed = getsockopt($sock, IPPROTO_TCP, TCP_NODELAY)
    or die "getsockopt TCP_NODELAY: $!";
my $nodelay = unpack("I", $packed);
print "Nagle is ", $nodelay ? "off" : "on", "\n";
```

Ίδια ιδέα, αλλά με λήψη του αριθμού πρωτοκόλλου TCP κατά τον χρόνο εκτέλεσης από τη βάση πρωτοκόλλων του συστήματος αντί από σταθερά του Socket:

```
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: $!";
```

Ανάγνωση του SO_ERROR για εκκένωση και εκκαθάριση εκκρεμούς ασύγχρονου σφάλματος σε μη μπλοκαριστική υποδοχή μετά από connect ή αφότου η select την επισημάνει ως εγγράψιμη:

```
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: $!";
}
```

Ανάγνωση του τύπου υποδοχής (SOCK_STREAM, SOCK_DGRAM, ...) ενός ήδη ανοιγμένου 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;
```

Ανάγνωση επιλογής `struct timeval`. Το SO_RCVTIMEO επιστρέφει δύο εγγενή `long` - δευτερόλεπτα και μικροδευτερόλεπτα:

```
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;
```

Οριακές περιπτώσεις

- **Η SOCKET δεν είναι ανοιχτή υποδοχή**: η κλήση αποτυγχάνει, επιστρέφει undef, και θέτει το `$!` σε ENOTSOCK (`"Socket operation on non-socket"`) ή EBADF (`"Bad file descriptor"`) για κλειστό handle.
- **Άγνωστο LEVEL / OPTNAME**: ο πυρήνας απορρίπτει το αίτημα με ENOPROTOOPT (`"Protocol not available"`). Δεν υπάρχει τρόπος να διακρίνετε αυτή την περίπτωση από την περίπτωση επιτυχίας εκτός από τον έλεγχο της τιμής επιστροφής.
- **Ιδίωμα επιστροφής με αλήθεια**: το συνηθισμένο `getsockopt(...) or die` λειτουργεί επειδή μια επιτυχημένη κλήση επιστρέφει μη κενή πακεταρισμένη συμβολοσειρά. Μια απάντηση μηδενικού μήκους θα έκανε το ιδίωμα να αστοχήσει, αλλά καμία επιλογή του πυρήνα δεν επιστρέφει ενταμιευτή μηδενικού μήκους στην πράξη - κάθε ορισμένη επιλογή έχει τουλάχιστον ένα byte payload.
- **Πλάτος αποκωδικοποίησης ακεραίου**: χρησιμοποιήστε `"i"` ή `"I"` για επιλογές μεγέθους `int` (SO_KEEPALIVE, TCP_NODELAY, SO_ERROR, ...)· χρησιμοποιήστε `"l!"` ή `"L!"` μόνο όταν η τεκμηριωμένη διάταξη της επιλογής χρησιμοποιεί όντως εγγενές `long`. Λάθος πλάτος επιστρέφει σιωπηρά τον λάθος αριθμό σε πλατφόρμες LP64.

- **Μη πακετάρετε την απάντηση χειροκίνητα**: η `getsockopt` επιστρέφει ό,τι έγραψε ο πυρήνας, συμπεριλαμβανομένης κάθε συμπλήρωσης. Επαναπακετάρετε μόνο σε πλήρεις κύκλους μέσω της `setsockopt`, και τότε περάστε τον ενταμιευτή πίσω αυτούσιο αντί να τον ανασυγκροτείτε.
- **Καμία σταθερά χωρίς `Socket`**: τα `SOL_SOCKET`, `IPPROTO_TCP`, `TCP_NODELAY`, `SO_ERROR`, και συγγενή δεν είναι ενσωματωμένα ονόματα. Εισάγετέ τα από το `Socket` με `use Socket qw(...)` ή `use Socket qw(:all)`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `setsockopt` - το μισό εγγραφής· ορίζει την επιλογή της οποίας την τιμή ανακτά η `getsockopt`
- `Socket` - πηγή των σταθερών `SOL_*`, `IPPROTO_*`, `SO_*`, `TCP_*`, `IP_*` που περνάτε ως `LEVEL` και `OPTNAME`
- `unpack` - απαιτείται για την αποκωδικοποίηση της πακεταρισμένης συμβολοσειράς bytes που παραδίδει ο πυρήνας
- `getprotobyname` - τρόπος κατά τον χρόνο εκτέλεσης για την απόκτηση του αριθμού πρωτοκόλλου `LEVEL` όταν δεν θέλετε να εισάγετε τις σταθερές `IPPROTO_*` από το `Socket`
- `getsockname` - αδελφή κλήση ενδοσκόπησης· επιστρέφει την τοπική διεύθυνση που είναι δεσμευμένη στην `SOCKET` αντί για μία επιλογή
- `socket` - δημιουργεί την υποδοχή την οποία στη συνέχεια ερωτά η `getsockopt`

Filehandles, αρχεία, κατάλογοι

glob

Επεκτείνει ένα μοτίβο ονόματος αρχείου σε στιλ κελύφους στη λίστα διαδρομών που ταιριάζουν.

Η `glob` παίρνει μια συμβολοσειρά μοτίβου όπως `"*.c"` ή `"src/**/*.pm"` και επιστρέφει τα ονόματα αρχείων στον δίσκο που ταιριάζουν, στο ίδιο στιλ που θα παρήγαγε ένα κέλυφος Unix. Σε περιβάλλον λίστας επιστρέφει όλα τα ταιριάσματα ταυτόχρονα· σε βαθμωτό περιβάλλον λειτουργεί ως επαναλήπτης με κατάσταση, επιστρέφοντας ένα ταίριασμα ανά κλήση και `undef` όταν η λίστα εξαντληθεί. Χωρίς όρισμα, η `glob` επεκτείνει το μοτίβο στο `$_`.

Σύνοψη

```
glob EXPR
glob
<PATTERN>
```

Τι επιστρέφεται

Σε περιβάλλον λίστας, μια (πιθανώς κενή) λίστα ονομάτων αρχείων. Κενή λίστα σημαίνει ότι το μοτίβο δεν ταίριαξε - αυτό **δεν** είναι σφάλμα και δεν ορίζει το `$!`.

Σε βαθμωτό περιβάλλον, η `glob` είναι επαναλήπτης: κάθε κλήση επιστρέφει το επόμενο ταίριασμα, και μετά το τελευταίο ταίριασμα μία ακόμη κλήση επιστρέφει `undef`. Η κατάσταση του επαναλήπτη είναι ανά σημείο κλήσης, οπότε δύο βρόχοι `while (my $f = glob ...)` σε διαφορετικές πηγαίες θέσεις δεν παρεμβαίνουν ο ένας στον άλλον.

```
my @mp3  = glob "*.mp3";              # list context
while (my $f = glob "*.mp3") { }      # scalar context, iterator
```

Καθολική κατάσταση που επηρεάζει

- `$_` - διαβάζεται όταν η `glob` καλείται χωρίς όρισμα.
- Η ίδια η `glob` **δεν** ορίζει το `$!` σε «μηδέν ταιριάσματα». Η μη ύπαρξη ταιριάσματος αποτελεί νόμιμη κενή λίστα, όχι αποτυχία.
- Όταν μια έκφραση `glob` χρησιμοποιείται ως συνθήκη βρόχου `while` ή `for`, η Perl εκχωρεί έμμεσα το αποτέλεσμα στο `$_` και ελέγχει για **οριστική τιμή**, όχι για αληθή - έτσι ένα όνομα αρχείου `"0"` εξακολουθεί να κρατά τον βρόχο σε εκτέλεση.

Η συντόμευση `<...>` και πώς την επιλέγει η Perl

Το `<*.c>` είναι εναλλακτική σύνταξη για το `glob("*.c")`. Η μορφή με γωνιώδεις αγκύλες μοιράζεται έναν αναγνώστη με το `<FH>` (`readline`)· ο αναλυτής αποφασίζει ποια από τις δύο εννοούσατε **κατά τη μεταγλώττιση** με βάση το τι βρίσκεται ανάμεσα στις γωνιώδεις αγκύλες:

- Bareword που θα μπορούσε να είναι filehandle (`<STDIN>`, `<FH>`) → `readline`.
- Ένα απλό βαθμωτό (`<$fh>`) → `readline` σε εκείνο το handle.
- Οτιδήποτε μοιάζει με μοτίβο (`<*.c>`, `<"*.txt">`, `<${dir}/*>`) → `glob`.

Η ονομασμένη συνάρτηση `glob` είναι μονοσήμαντη και πιο εύκολα αναζητήσιμη. Προτιμήστε την σε νέο κώδικα· καταφεύγετε στο `<...>` μόνο για τα πιο σύντομα μοτίβα μιας χρήσης.

```
my @txt = <"*.txt">;                  # same as glob '"*.txt"'
my @txt = glob "*.txt";               # recommended
```

Οι λευκοί χαρακτήρες διασπούν το μοτίβο

Η `glob` διασπά το όρισμά της σε λευκούς χαρακτήρες και χειρίζεται κάθε τμήμα ως ξεχωριστό μοτίβο. Τα αποτελέσματα συνενώνονται:

```
my @sources = glob "*.c *.h";         # all .c OR .h files
my @all     = glob ".* *";            # every entry in cwd
```

Για να εφαρμόσετε glob σε όνομα αρχείου που περιέχει κενά, περικλείστε το σε ένα εσωτερικό ζεύγος εισαγωγικών ώστε η διάσπαση να δει ένα τμήμα:

```
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
```

Παρεμβολή μεταβλητής σε μοτίβο με κενά:

```
my @spacey = glob "'*${var}e f*'";
my @spacey = glob qq("*${var}e f*");
```

Αν ο κανόνας διάσπασης σε λευκούς χαρακτήρες σας μπερδεύει, χρησιμοποιήστε απευθείας την bsd_glob του File::Glob - παίρνει το μοτίβο κατά λέξη.

Επέκταση αγκίστρων χωρίς wildcard

Αν οι μόνοι ειδικοί χαρακτήρες είναι μη κενά άγκιστρα, η glob δεν αγγίζει καθόλου το σύστημα αρχείων: απλώς απαριθμεί το Καρτεσιανό γινόμενο των εναλλακτικών. Αυτό είναι χρήσιμο για παραγωγή συνδυασμών συμβολοσειρών:

```
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
```

Το File::Glob - η μηχανή και οι επιλογές της

Η glob υλοποιείται πάνω από την bsd_glob του File::Glob. Η εισαγωγή των ετικετών του αρθρώματος ρυθμίζει τη συμπεριφορά για ολόκληρη τη μονάδα μεταγλώττισης:

- :nocase - αντιστοίχιση χωρίς διάκριση πεζών-κεφαλαίων.
- :case - επιβολή αντιστοίχισης με διάκριση πεζών-κεφαλαίων (η προεπιλογή στα περισσότερα συστήματα αρχείων Unix).
- :globally - αντικαθιστά τον πυρηνικό τελεστή glob με την bsd_glob ώστε κάθε glob(...) και <...> στο αρχείο να υιοθετεί τις επιλογές σας.

```
use File::Glob qw(:globally :nocase);
my @txt = glob "readme*";             # README, readme.txt, Readme.md
```

Καλέστε την bsd_glob απευθείας όταν χρειάζεστε σημαίες ανά κλήση ή όταν θέλετε να παρακάμψετε τον κανόνα διάσπασης σε λευκούς χαρακτήρες:

```
use File::Glob qw(bsd_glob GLOB_TILDE GLOB_BRACE);
my @files = bsd_glob("~/docs/*.{md,rst}", GLOB_TILDE | GLOB_BRACE);
```

Παραδείγματα

Εμφανίστε κάθε αρχείο πηγαίου κώδικα Perl στον τρέχοντα κατάλογο:

```
my @perl_files = glob "*.pl *.pm";
```

Επαναλάβετε σε αρχεία MP3 ένα ένα - χρήσιμο όταν η λίστα μπορεί να είναι μεγάλη και δεν θέλετε να την υλοποιήσετε στη μνήμη:

```
while (my $song = glob "*.mp3") {
    process($song);                 # one filename per iteration
}
```

Τα αναδρομικά μοτίβα **δεν** είναι ενσωματωμένα· το `*` δεν διασχίζει το `/`. Διατρέξτε το δέντρο με `File::Find` ή συνδυάστε την `glob` με ρητή διέλευση:

```
my @all_c;
for my $dir (glob "src/*") {
    push @all_c, glob "$dir/*.c" if -d $dir;
}
```

Χρησιμοποιήστε τη μορφή επαναλήπτη σε συνθήκη `while` - το αποτέλεσμα εκχωρείται στο `$_` και ελέγχεται για οριστική τιμή, οπότε ονόματα αρχείων όπως `"0"` δεν τερματίζουν τον βρόχο:

```
while (<*.log>) {
    unlink $_ if -M $_ > 30;        # older than 30 days
}
```

Αντιστοίχιση χωρίς διάκριση πεζών-κεφαλαίων για ολόκληρο αρχείο:

```
use File::Glob qw(:globally :nocase);
my @readmes = glob "readme*";
```

Οριακές περιπτώσεις

- **Μηδέν ταιριάσματα**: επιστρέφει την κενή λίστα. Καμία προειδοποίηση, κανένα `$!` δεν ορίζεται. Αν χρειάζεστε «το μοτίβο έπρεπε να ταιριάξει», ελέγξτε ο ίδιος το `scalar @files`.
- **`glob` χωρίς όρισμα**: χρησιμοποιεί το `$_`. Μια γυμνή `glob` μέσα σε `while (<>)` σπάνια είναι αυτό που θέλετε - το μοτίβο θα είναι οποιαδήποτε γραμμή εισόδου μόλις διαβάσατε.
- **Το `while (glob ...)` εκχωρεί στο `$_` και ελέγχει την οριστική τιμή**: ένα όνομα αρχείου `"0"` είναι ψευδές σύμφωνα με τους συνηθισμένους κανόνες της λογικής τιμής, αλλά ο βρόχος εξακολουθεί να εκτελείται επειδή η συνθήκη είναι `defined $_`. Αυτό ταυτίζεται με τον κανόνα για την `readline` στην ίδια θέση.
- **Wildcards και λευκοί χαρακτήρες**: ο κανόνας διάσπασης σε λευκούς χαρακτήρες σημαίνει ότι το `glob "a b"` αναζητά τα a και b ως δύο μοτίβα, όχι την κυριολεκτική συμβολοσειρά `"a b"`. Χρησιμοποιήστε εσωτερικά εισαγωγικά ή την `bsd_glob` όταν το όνομα αρχείου νόμιμα περιέχει κενά.
- **Κρυφά αρχεία**: ένα προπορευόμενο `.` δεν ταιριάζει με το `*` - χρειάζεστε ρητό τμήμα `.*` (`glob ".* *"`) για να συμπεριλάβετε dotfiles. Αυτό ταυτίζεται με τη σημασιολογία glob κελύφους, όχι τη σημασιολογία regex.
- **Επέκταση tilde**: το `glob "~/foo"` λειτουργεί επειδή το `File::Glob` επεκτείνει το `~` εξ ορισμού. Καλώντας το μέσω της `bsd_glob` χωρίς το `GLOB_TILDE` στις σημαίες,

λαμβάνετε ένα κυριολεκτικό ~.

- **Παρεμβολή μέσα σε `<...>`**: το `<$var/*.c>` παρεμβάλλει το `$var` σε χρόνο εκτέλεσης. Η χρήση της μορφής με γωνιώδεις αγκύλες δεν σας απαλλάσσει από τους συνηθισμένους κανόνες διπλών εισαγωγικών της Perl.
- **Ασάφεια γωνιωδών αγκυλών**: το `<FOO>` είναι ανάγνωση handle, το `<$foo>` είναι ανάγνωση handle, το `<*.c>` είναι glob. Το `<$foo/*.c>` - ένα βαθμωτό συν χαρακτήρες μοτίβου - είναι glob. Η επίλυση γίνεται κατά τη μεταγλώττιση· οι αλλαγές στο `$foo` σε χρόνο εκτέλεσης δεν την επανακρίνουν.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `opendir` - ανοίγει ένα handle καταλόγου για πιο λεπτομερή διέλευση από αυτή που σας δίνει η αντιστοίχιση μοτίβων
- `readdir` - διαβάζει εγγραφές από ένα handle καταλόγου μία τη φορά· συνδυάστε με `opendir` όταν χρειάζεστε κάθε εγγραφή (περιλαμβανομένων εκείνων που η `glob` κρύβει, όπως τα dotfiles) ή πληροφορίες τύπου αρχείου
- `readline` - η άλλη σημασία του `<...>`· μοιράζεται τη σύνταξη γωνιωδών αγκυλών αλλά διαβάζει γραμμές από filehandle
- `File::Glob` - η υποκείμενη υλοποίηση· χρησιμοποιήστε την `bsd_glob` απευθείας για σημαίες ανά κλήση και για να παρακάμψετε τον κανόνα διάσπασης σε λευκούς χαρακτήρες
- `$_` - προεπιλεγμένη πηγή μοτίβου για γυμνή `glob`, και η μεταβλητή στην οποία εκχωρείται έμμεσα όταν η `glob` χρησιμοποιείται ως συνθήκη βρόχου `while`/`for`

Χρόνος

gmtime

Μετατρέπει έναν χρόνο epoch σε αναλυτικό χρόνο UTC, είτε ως λίστα 9 στοιχείων είτε ως συμβολοσειρά τύπου `ctime(3)`.

Η `gmtime` είναι η `localtime` εκφρασμένη στη ζώνη ώρας Greenwich. Παίρνει έναν ακέραιο αριθμό δευτερολέπτων από την epoch - τον τύπο τιμής που επιστρέφει η `time` - και επιστρέφει είτε τα επιμέρους ημερολογιακά πεδία (σε περιβάλλον λίστας) είτε μια έτοιμη προς εκτύπωση συνοπτική συμβολοσειρά (σε βαθμωτό περιβάλλον). Καμία αναζήτηση ζώνης ώρας, κανένα `TZ`, κανένα DST: το αποτέλεσμα είναι πάντοτε UTC.

Σύνοψη

```
gmtime EXPR
gmtime
```

Η `gmtime` χωρίς όρισμα χρησιμοποιεί την τρέχουσα `time`.

Τι επιστρέφεται

Σε **περιβάλλον λίστας**, μια λίστα εννέα στοιχείων που προέρχεται απευθείας από το C `struct tm`:

```
#    0     1     2      3      4     5      6      7      8
my ($sec, $min, $hour, $mday, $mon, $year, $wday, $yday, $isdst)
    = gmtime(time);
```

Δείκτης	Πεδίο	Εύρος	Σημειώσεις
0	`$sec`	`0..60`	το 60 επιτρέπει ένα θετικό leap second
1	`$min`	`0..59`	
2	`$hour`	`0..23`	
3	`$mday`	`1..31`	**με αρχή το 1**
4	`$mon`	`0..11`	**με αρχή το 0** - ο Ιανουάριος είναι 0, ο Δεκέμβριος είναι 11
5	`$year`		**έτη από το 1900** - προσθέστε `1900` για μ.Χ.
6	`$wday`	`0..6`	η Κυριακή είναι 0
7	`$yday`	`0..365`	0 είναι η 1η Ιανουαρίου· 365 μόνο σε δίσεκτα έτη
8	`$isdst`	`0`	**πάντοτε 0** για την `gmtime` - το UTC δεν έχει DST

Σε **βαθμωτό περιβάλλον**, μία και μόνη αγγλική συμβολοσειρά μορφοποιημένη ως `ctime(3)`:

```
my $s = gmtime;    # "Thu Oct 13 04:54:34 1994"
```

Η βαθμωτή μορφή είναι **πάντοτε αγγλική και ανεξάρτητη τοπικών ρυθμίσεων**. Για μια συμβολοσειρά που αναγνωρίζει τοπικές ρυθμίσεις ή έχει διαφορετική μορφή, φτιάξτε την από τη μορφή λίστας με την `POSIX::strftime`.

Καθολική κατάσταση που επηρεάζει

Καμία. Η `gmtime` δεν διαβάζει το `$ENV{TZ}`, δεν συμβουλεύεται τις τοπικές ρυθμίσεις του διερμηνέα και δεν εξαρτάται ούτε τροποποιεί καμία από τις ειδικές μεταβλητές της Perl. Με το ίδιο EXPR, επιστρέφει πάντα το ίδιο αποτέλεσμα.

Χωρίς όρισμα, διαβάζει τον τρέχοντα πραγματικό χρόνο μέσω της `time`.

Παραδείγματα

Τρέχον UTC αναλυμένο στα πεδία του, με τις παγίδες του 0-δείκτη μήνα και του έτους-από-1900 να αντιμετωπίζονται και οι δύο:

```
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"
```

Η βαθμωτή μορφή - μία συμβολοσειρά σταθερής μορφής, χρήσιμη για αρχεία καταγραφής:

```
my $stamp = gmtime;
print "[$stamp] startup\n";
# "[Thu Apr 23 14:07:09 2026] startup"
```

Μορφοποιήστε τη λίστα με την `POSIX::strftime` όταν θέλετε ISO-8601 ή απόδοση με επίγνωση τοπικών ρυθμίσεων:

```
use POSIX qw(strftime);
my $iso = strftime "%Y-%m-%dT%H:%M:%SZ", gmtime;
# e.g. "2026-04-23T14:07:09Z"
```

Μετατρέψτε μια αποθηκευμένη epoch (όχι τον τρέχοντα χρόνο) - οποιαδήποτε ακέραια δευτερόλεπτα από την 1970-01-01 UTC λειτουργούν:

```
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"
```

Το `$isdst` είναι πάντοτε `0` υπό την `gmtime` - μην το χρησιμοποιείτε για να ανιχνεύσετε το DST. Καλέστε την `localtime` αν χρειάζεστε αυτή την πληροφορία:

```
my $dst_here = (localtime)[8];   # 0 or 1 depending on local zone
my $dst_utc  = (gmtime)[8];      # always 0
```

Οριακές περιπτώσεις

- **Η γυμνή κλήση χρησιμοποιεί τον τρέχοντα χρόνο**: η `gmtime` χωρίς όρισμα ισούται με `gmtime(time)`. Μέσα σε παρεμβολή ή σε λίστα που μπορεί να καταπιεί την κλήση, χρησιμοποιήστε παρενθέσεις: `gmtime()`.
- **Το περιβάλλον λίστας έναντι βαθμωτού είναι φέρον**. Σε περιβάλλον λίστας λαμβάνετε εννέα αριθμούς· σε βαθμωτό περιβάλλον λαμβάνετε μία συμβολοσειρά. Εξαναγκάζοντας βαθμωτό σε γυμνή κλήση μέσα σε λίστα:

  ```
  print gmtime, "\n";                # nine digits, no spaces
  print scalar gmtime, "\n";         # "Thu Apr 23 14:07:09 2026"
  ```

- **Το `$mon` έχει αρχή το 0, το `$year` μετράται από το 1900**. Δύο από τα πιο κοινά σφάλματα της Perl. Για πλήρες ημερολογιακό έτος γράψτε `$year + 1900`· για μήνα με αρχή το 1 γράψτε `$mon + 1`.
- **Το `$mday` έχει αρχή το 1, το `$yday` έχει αρχή το 0**. Αντικατοπτρίζει επακριβώς το C `struct tm` - ασυνεπές, αλλά σταθερό.
- **`$sec == 60`** σε θετικό leap second. Κώδικας που επικυρώνει με `$sec <= 59` θα απορρίψει νόμιμες τιμές.
- **Αρνητικές τιμές epoch** αναπαριστούν ημερομηνίες πριν από την 1970-01-01 UTC και γίνονται δεκτές σε πλατφόρμες των οποίων η `gmtime(3)` τις δέχεται - όπως το Linux. Η `gmtime(-1)` επιστρέφει 1969-12-31 23:59:59 UTC.

- **Τα κλασματικά δευτερόλεπτα απορρίπτονται**. Η EXPR αποκόπτεται προς το μηδέν πριν από τη μετατροπή· το gmtime(1.9) ισούται με gmtime(1). Για υπο-δευτερολεπτική ακρίβεια, κρατήστε ξεχωριστά το δικό σας κλασματικό μέρος και μορφοποιήστε το χωριστά.
- **Καμία συντομογραφία ζώνης ώρας στη βαθμωτή μορφή**. Σε αντίθεση με την έξοδο ctime(3) της localtime σε ορισμένες πλατφόρμες, η βαθμωτή μορφή της gmtime δεν περιλαμβάνει ποτέ όνομα ζώνης. Είναι πάντα η μορφή πέντε πεδίων "Wdy Mon DD HH:MM:SS YYYY".
- **Το $isdst είναι πάντοτε 0**, όχι undef και όχι -1. Μην το ελέγχετε ως προς την αλήθεια υπό την gmtime· η απάντηση δεν έχει νόημα.

Εναλλακτικές και συμπληρωματικά

- **Η αντίστροφη πορεία** - από αναλυτικά πεδία πίσω σε epoch - είναι δουλειά του αρθρώματος Time::Local. Χρησιμοποιήστε εκεί την timegm (UTC) ή την timelocal (τοπική ζώνη):

```
use Time::Local qw(timegm);
my $epoch = timegm($sec, $min, $hour, $mday, $mon, $year);
```

- **Αντικειμενοστρεφή προσπέλαση βάσει ονόματος πεδίου** προσφέρει το Time::Piece, που υπερφορτώνει τις localtime και gmtime ώστε να επιστρέφουν αντικείμενα με μεθόδους ->year, ->mon, ->mday κ.λπ. Οι συμβάσεις πεδίων εκεί έχουν προσαρμοστεί - το year είναι ήδη το πλήρες έτος, το mon είναι ήδη με αρχή το 1 - οπότε κώδικας γραμμένος για το Time::Piece δεν μπορεί να αναμιχθεί απερίσκεπτα με κώδικα γραμμένο για την ενσωματωμένη μορφή λίστας.
- **Η αναγνώσιμη από ανθρώπους μορφοποίηση** ανήκει στην POSIX::strftime· τροφοδοτήστε την απευθείας με τη λίστα που επιστρέφει η gmtime.
- **Εκτενής αριθμητική ημερομηνιών, ζώνες ώρας, διάρκειες** - καταφύγετε στο DateTime του CPAN. Η gmtime είναι το ελάχιστο βιώσιμο πρωτογενές δομικό στοιχείο· το DateTime είναι η πλήρης βιβλιοθήκη.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- localtime - η ίδια μετατροπή, αλλά στην τοπική ζώνη ώρας· εκεί το $isdst έχει νόημα
- time - παράγει την τιμή epoch που καταναλώνει η gmtime
- POSIX::strftime - μορφοποιήστε τη λίστα 9 στοιχείων σε αυθαίρετη συμβολοσειρά ημερομηνίας (ISO-8601, με επίγνωση τοπικών ρυθμίσεων κ.λπ.)
- Time::Local - αντίστροφη πράξη: κατασκευάζει μια epoch από αναλυτικά πεδία UTC ή τοπικά
- Time::Piece - αντικειμενοστρεφές περιτύλιγμα γύρω από τις gmtime / localtime με ονομασμένες μεθόδους προσπέλασης

Ροή ελέγχου

goto

Μεταφορά εκτέλεσης αλλού στο πρόγραμμα χωρίς επιστροφή.

Η goto έχει τρεις ασυσχέτιστες μορφές που μοιράζονται μόνο τη δεσμευμένη λέξη. Η χρήσιμη στη σύγχρονη Perl είναι η goto &NAME - η μορφή **κλήσης ουράς**, η οποία αντικαθιστά το τρέχον πλαίσιο υπορουτίνας με κλήση σε άλλη υπορουτίνα διατηρώντας το @_. Οι μορφές goto LABEL και goto EXPR υπάρχουν, είναι περιορισμένες, και σχεδόν ποτέ δεν είναι το σωστό εργαλείο.

Σύνοψη

```
goto LABEL
goto EXPR
goto &NAME
```

Τι επιστρέφεται

Η goto δεν επιστρέφει. Ο έλεγχος εγκαταλείπει τη δήλωση goto και δεν επιστρέφει. Η τιμή επιστροφής της κατασκευής στην οποία τελικά προσγειώνεται η goto είναι η τιμή της περικλείουσας έκφρασης.

Για την goto &NAME, η τιμή επιστροφής είναι ό,τι επιστρέφει η υπορουτίνα-στόχος, που διαδίδεται στον αρχικό καλούντα σαν να είχε κληθεί ο στόχος άμεσα.

Η μορφή κλήσης ουράς: goto &NAME

Αυτή είναι η μορφή που έχει σημασία. Εξέρχεται από την τρέχουσα υπορουτίνα - αναιρώντας τυχόν συνδέσεις local - και καλεί άμεσα την NAME με το τρέχον @_. Το τρέχον πλαίσιο **αντικαθίσταται**, δεν στοιβάζεται από πάνω.

Συνέπειες:

- Η caller μέσα στην υπορουτίνα-στόχο βλέπει τον **αρχικό καλούντα**, όχι την υπορουτίνα που εξέδωσε την goto. Το ενδιάμεσο πλαίσιο έχει χαθεί.
- Το @_ περνά ως έχει. Τροποποιήσεις στο @_ πριν την goto είναι ορατές στον στόχο.
- Οι αλλαγές local που έγιναν στην αναχωρούσα υπορουτίνα αναιρούνται πριν τρέξει ο στόχος, ακριβώς όπως θα γίνονταν σε κανονική επιστροφή.
- Ο στόχος δεν χρειάζεται να γνωρίζει ότι προσεγγίστηκε μέσω goto &· από τη δική του πλευρά κλήθηκε κανονικά.

Η κανονική χρήση είναι η αποστολή AUTOLOAD: αναλύστε την πραγματική υπορουτίνα, και έπειτα κάντε goto σε αυτή ώστε τα ίχνη στοίβας, η caller και η wantarray να συμπεριφέρονται σαν να είχε κληθεί η πραγματική υπορουτίνα εξαρχής.

Το NAME δεν χρειάζεται να είναι κυριολεκτικό όνομα. Μπορεί να είναι οποιαδήποτε έκφραση που παράγει αναφορά κώδικα - ένα βαθμωτό που κρατά coderef, ένα μπλοκ που επιστρέφει ένα, ή μια αποαναφερόμενη θέση.

```
goto &$coderef;
goto &{ $dispatch{$op} };
```

Για αναδρομικές κλήσεις ουράς, η goto __SUB__ μεταβαίνει στην εκτελούμενη τη στιγμή υπορουτίνα χωρίς να επαναλύει το όνομά της - ασφαλής υπό μετονομασία και ανώνυμες υπορουτίνες.

Η μορφή goto LABEL

Βρίσκει τη δήλωση με ετικέτα LABEL εντός της τρέχουσας δυναμικής εμβέλειας και συνεχίζει την εκτέλεση εκεί. Στην πράξη αυτό σημαίνει ότι μπορεί να πηδήξει εκτός μπλοκ και εκτός υπορουτινών, αλλά **δεν μπορεί να πηδήξει μέσα** σε μια κατασκευή που απαιτεί αρχικοποίηση - σώμα υπορουτίνας, βρόχο foreach, μπλοκ given, την παράμετρο δυαδικού ή λίστας τελεστή, ή κώδικα που αφαίρεσε ο βελτιστοποιητής. Το πήδημα μέσα σε τέτοια κατασκευή είναι μοιραίο σφάλμα από την Perl 5.44.

Τα περισσότερα από όσα θα μπορούσε να κάνει η goto LABEL εκφράζονται καλύτερα με έλεγχο βρόχου με ετικέτα (last LABEL, next LABEL, redo LABEL) ή με εξαιρέσεις (die / eval). Καταφεύγετε πρώτα σε αυτά.

Η μορφή goto EXPR

Αποτιμά την EXPR. Αν παράγει αναφορά κώδικα, συμπεριφέρεται όπως η goto &NAME. Αν παράγει συμβολοσειρά, αυτή η συμβολοσειρά χρησιμοποιείται ως όνομα ετικέτας που αναλύεται κατά τον χρόνο εκτέλεσης - υπολογιζόμενο goto. Οι ίδιοι περιορισμοί όπως στην goto LABEL ισχύουν για την περίπτωση της ετικέτας.

Δύο ιδιαιτερότητες του αναλυτή που πρέπει να θυμάστε:

- Η goto EXPR εξαιρείται από τον κανόνα «μοιάζει με συνάρτηση». Παρενθέσεις μετά την goto δεν οριοθετούν απαραίτητα το όρισμά της: η goto("NE")."XT" είναι goto NEXT, όχι goto "NE" ακολουθούμενη από μια άχρηστη συνένωση.
- Η goto έχει την ίδια προτεραιότητα με την εκχώρηση, σε αντίθεση με τους περισσότερους ονομασμένους τελεστές.

Παραδείγματα

Αποστολή κλήσης ουράς από AUTOLOAD - η υπορουτίνα-στόχος βλέπει τον αρχικό καλούντα, όχι την 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
}
```

Αναδρομή ουράς χωρίς αύξηση της στοίβας κλήσεων. Η __SUB__ αναλύεται στην εκτελούμενη τη στιγμή υπορουτίνα:

```
sub fact {
    my ($n, $acc) = @_;
    $acc //= 1;
    return $acc if $n <= 1;
    @_ = ($n - 1, $acc * $n);
    goto __SUB__;
}
```

Υπολογιζόμενο goto μέσω goto EXPR. Η ετικέτα επιλέγεται κατά τον χρόνο εκτέλεσης από μια λίστα:

```
goto ("FOO", "BAR", "GLARCH")[$i];
FOO:    handle_foo();    return;
BAR:    handle_bar();    return;
GLARCH: handle_glarch(); return;
```

Ανάθεση σε coderef που κρατείται σε πίνακα αποστολής. Ο καλών δεν βλέπει ποτέ το ενδιάμεσο πλαίσιο:

```
my %ops = (
    add => sub { $_[0] + $_[1] },
    mul => sub { $_[0] * $_[1] },
);

sub run_op {
    my $op = shift;
    goto &{ $ops{$op} // die "unknown op: $op" };
}
```

Τροποποίηση του @_ πριν από κλήση ουράς - οι αλλαγές είναι ορατές στον στόχο:

```
sub wrap {
    unshift @_, "prefix";
    goto &real_handler;       # real_handler sees ("prefix", @orig_
↪args)
}
```

Οριακές περιπτώσεις

- **Η goto LABEL δεν μπορεί να εισέλθει σε υπορουτίνα, foreach ή given.** Από την Perl 5.44 αυτό είναι μοιραίο σφάλμα, όχι προειδοποίηση. Επίσης δεν μπορεί να πηδήξει σε κώδικα που απαλείφθηκε από βελτιστοποίηση ή στους περισσότερους τελεστέους τελεστών - η μοναδική εξαίρεση είναι ο πρώτος τελεστέος ενός δυαδικού τελεστή (για το =, η δεξιά πλευρά).
- **Η goto LABEL δεν μπορεί να βγει από μπλοκ σύγκρισης `sort`.** Ο συγκριτής τρέχει σε περιβάλλον που δεν επιτρέπει το πήδημα.
- **Η goto &NAME απορρίπτει συνδέσεις `local`** από την αναχωρούσα υπορουτίνα πριν τρέξει ο στόχος. Ό,τι έχει εντοπίσει η υπορουτίνα αποκαθίσταται σαν να είχε επιστρέψει κανονικά η υπορουτίνα.

- **Η `caller` είναι αόρατη πέρα από `goto &`.** Το αναχωρόν πλαίσιο έχει χαθεί. Αυτός είναι όλος ο σκοπός της μορφής για την AUTOLOAD - μη βασίζεστε στο ότι θα δείτε την ενδιάμεση υπορουτίνα σε ίχνος στοίβας.
- **Η ψευδωνυμοποίηση του `@_` διατηρείται.** Το `@_` της υπορουτίνας-στόχου ψευδωνυμεί τα ίδια βαθμωτά που ψευδωνύμιζε το `@_` της αναχωρούσας υπορουτίνας. Η εκχώρηση στο `$_[0]` στον στόχο εξακολουθεί να τροποποιεί το όρισμα του καλούντος.
- **Η `goto &$coderef` απαιτεί πραγματικό coderef.** Μια συμβολοσειρά που μοιάζει με όνομα υπορουτίνας δεν λειτουργεί στη μορφή & - χρησιμοποιήστε `goto EXPR` (χωρίς το &) για αυτό, ή αναλύστε πρώτα το όνομα σε coderef με `\&{$name}`.
- **Η `goto` έχει προτεραιότητα εκχώρησης.** Η `goto "A" . "B"` είναι `goto "AB"`, όχι `(goto "A") . "B"`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `last` - έξοδος από τον πιο εσωτερικό ή έναν ονομασμένο περικλείοντα βρόχο· το σωστό εργαλείο για τις περισσότερες περιπτώσεις «πήδηξε έξω από εδώ»
- `next` - μετάβαση στην επόμενη επανάληψη ενός βρόχου, με προαιρετική ετικέτα
- `redo` - επανεκκίνηση της τρέχουσας επανάληψης βρόχου χωρίς επαναποτίμηση της συνθήκης
- `return` - έξοδος από την τρέχουσα υπορουτίνα με τιμή· συγκρίνετε με την `goto &NAME` που αφήνει την υπορουτίνα **αντικαθιστώντας** την
- `caller` - επιθεώρηση της στοίβας κλήσεων· σημειώστε ότι η `goto &` κάνει το αναχωρόν πλαίσιο αόρατο σε αυτή

Λίστες

grep

Φιλτράρει μια λίστα στα στοιχεία για τα οποία το μπλοκ ή η έκφραση είναι αληθή.

Η `grep` διασχίζει τη `LIST` από αριστερά προς τα δεξιά, αποτιμά το `BLOCK` ή την `EXPR` μία φορά ανά στοιχείο με την `$_` ψευδωνυμοποιημένη σε αυτό το στοιχείο, και επιστρέφει τα στοιχεία για τα οποία το τεστ παρήγαγε αληθή τιμή. Είναι φίλτρο λίστας, όχι βρόχος - η λίστα που λαμβάνετε πίσω είναι υποσύνολο της λίστας που περάσατε, στην αρχική σειρά.

Σύνοψη

```
grep BLOCK LIST
grep EXPR, LIST
my @kept  = grep { COND } @list;
my @match = grep /pattern/, @list;
```

Οι δύο μορφές είναι ισοδύναμες σε ισχύ· το BLOCK είναι η γενική περίπτωση, και η EXPR είναι ευκολία για τεστ μίας έκφρασης. Όταν η EXPR είναι σκέτη regex (`/ . . . /`, `m/ . . . /` ή `qr/ . . . /`), ταιριάζεται σιωπηρά με την `$_` - αυτό είναι το ιδίωμα πίσω από το `grep /^#/, @lines`.

Τι επιστρέφεται

- **Περιβάλλον λίστας**: η φιλτραρισμένη λίστα, στην αρχική σειρά, με κάθε επιστρεφόμενο στοιχείο να είναι ψευδώνυμο μέσα στη λίστα-πηγή (δείτε *Καθολική κατάσταση* και *Οριακές περιπτώσεις*).
- **Βαθμωτό περιβάλλον**: ο μετρητής των στοιχείων για τα οποία το τεστ ήταν αληθές. Όχι λογική τιμή - είναι το μέγεθος αυτού που θα παρήγαγε το περιβάλλον λίστας, οπότε το `0` είναι ψευδές, οποιοσδήποτε θετικός μετρητής είναι αληθής.

```
my @keep  = grep { $_ > 0 } @nums;   # list context: filtered list
my $count = grep { $_ > 0 } @nums;   # scalar context: count
```

Καθολική κατάσταση που επηρεάζει

- Η `$_` είναι **ψευδωνυμοποιημένη** σε κάθε στοιχείο της LIST για τη διάρκεια μιας αποτίμησης του BLOCK ή της EXPR. Κατά την έξοδο από τη grep, η `$_` επαναφέρεται σε ό,τι κρατούσε πριν την κλήση. Επειδή το ψευδώνυμο είναι προς το πραγματικό στοιχείο - όχι αντίγραφο - η ανάθεση στην `$_` μέσα στο μπλοκ μεταβάλλει τη λίστα-πηγή επιτόπια. Αυτό είναι το ίδιο συμβόλαιο με την map και με τη μεταβλητή ευρετηρίου ενός βρόχου for.
- Η μορφή σκέτης regex της EXPR διαβάζει και γράφει όλες τις συνήθεις μαγικές μεταβλητές που σχετίζονται με αντιστοιχία (`$1`..`$9`, `$&`, `${^MATCH}`, `@+`, `@-`, `%+`, `%-`, `pos`) σε κάθε επανάληψη, ακριβώς σαν να είχατε γράψει την αντιστοιχία αναλυτικά.

Παραδείγματα

Φιλτράρετε έξω τις απροσδιόριστες τιμές:

```
my @vals    = (1, undef, 2, undef, 3);
my @defined = grep { defined $_ } @vals;
# @defined = (1, 2, 3)
```

Φίλτρο regex μέσω της μορφής έκφρασης - η σκέτη regex ταιριάζει έναντι της `$_`:

```
my @lines    = ("# comment", "code", "# another", "more code");
my @code     = grep !/^#/, @lines;       # drop comments
my @comments = grep  /^#/, @lines;       # keep comments
```

Προσαρμοσμένο τεστ μέσω της μορφής μπλοκ, όταν η συνθήκη είναι περισσότερες από μία εκφράσεις:

```
my @users    = (
    { name => "alice", active => 1, age => 30 },
    { name => "bob",   active => 0, age => 25 },
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    { name => "carol", active => 1, age => 17 },
);
my @adults_active = grep {
    $_->{active} && $_->{age} >= 18
} @users;
```

Μετρήστε χωρίς να δομήσετε τη λίστα - χρησιμοποιήστε βαθμωτό περιβάλλον:

```
my @errors = ("ok", "fail", "ok", "fail", "fail");
my $nfail  = grep { $_ eq "fail" } @errors;   # 3
if (grep { $_ eq "fail" } @errors) {          # boolean use
    warn "had failures";
}
```

Αφαιρέστε κενές γραμμές και γραμμές μόνο από λευκούς χαρακτήρες:

```
my @clean = grep { /\S/ } @lines;
```

Φιλτράρετε τα κλειδιά ενός πίνακα κατακερματισμού βάσει κάποιας ιδιότητας της τιμής:

```
my %scores  = (alice => 42, bob => 17, carol => 95);
my @passing = grep { $scores{$_} >= 50 } keys %scores;
```

Οριακές περιπτώσεις

- **Η `$_` είναι ψευδώνυμο, όχι αντίγραφο.** Η τροποποίηση της `$_` μέσα στο μπλοκ τροποποιεί το αρχικό στοιχείο της λίστας. Αυτό περιστασιακά είναι χρήσιμο αλλά συχνότερα είναι σφάλμα· γράψτε τεστ μόνο για ανάγνωση:

  ```
  my @bumped = grep { $_++; $_ > 10 } @nums;   # MUTATES @nums
  ```

  Αν τα στοιχεία δεν είναι τα ίδια μεταβλητές (π.χ. κυριολεκτικές τιμές, σταθερές ή αποτέλεσμα κλήσης συνάρτησης), η ανάθεση στην `$_` είναι σφάλμα χρόνου εκτέλεσης - «Modification of a read-only value».

- **Τα επιστρεφόμενα στοιχεία είναι επίσης ψευδώνυμα.** Τα στοιχεία της επιστρεφόμενης λίστας μοιράζονται αποθήκευση με τη λίστα-πηγή, ακριβώς όπως η μεταβλητή ευρετηρίου ενός βρόχου `for`. Η εγγραφή σε αυτά μέσω της επιστρεφόμενης λίστας διαδίδεται πίσω:

  ```
  my @arr  = (1, 2, 3, 4);
  my @even = grep { $_ % 2 == 0 } @arr;
  $even[0] = 99;      # @arr is now (1, 99, 3, 4)
  ```

  Αυτό σπάνια είναι το επιθυμητό. Αντιγράψτε ρητά με `map { $_ } grep ...` ή με μια φρέσκια `my` όταν χρειάζεστε ανεξάρτητη αποθήκευση.

- **Μορφή έκφρασης έναντι μορφής μπλοκ - προτεραιότητα.** Η `grep EXPR, LIST` χρησιμοποιεί κόμμα για να χωρίσει το τεστ από τη λίστα· η `grep BLOCK LIST` όχι. Η ανάμειξή τους σας μπερδεύει:

```
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
```

- **Η grep σε αλυσιδωτές εκφράσεις.** Η grep παίρνει μια LIST, που σημαίνει ότι οτιδήποτε μετά το πρώτο όρισμα καταναλώνεται ως μέρος της λίστας. Βάλτε παρενθέσεις όταν θέλετε να ακολουθήσετε τη grep με περισσότερους όρους:

```
my @out = (grep { /x/ } @a), @b;   # concatenates filtered @a␣
↪and @b
my @bad =  grep { /x/ } @a,  @b;   # filters BOTH @a and @b␣
↪together
```

- **Δεν είναι βρόχος.** Η grep είναι έκφραση, όχι δομή ελέγχου. Οι last, next και redo μέσα στο μπλοκ στοχεύουν τον πλησιέστερο περικλείοντα *βρόχο*, όχι την ίδια τη grep - θα εξέλθουν ή θα επανεκκινήσουν όποιον for/while περιέχει τη grep, όχι τη grep. Χρησιμοποιήστε μια πρόωρη return από μια βοηθητική sub αν χρειάζεται να βραχυκυκλώσετε, ή στρέψτε σε first της List::Util όταν το μόνο που θέλετε είναι η πρώτη αντιστοιχία.
- **Κενή είσοδος σημαίνει κενή έξοδος.** Η grep πάνω σε κενή λίστα είναι κενή λίστα σε περιβάλλον λίστας και 0 σε βαθμωτό περιβάλλον. Καμία ειδική περίπτωση δεν χρειάζεται.
- **Ισοπέδωση πίνακα κατακερματισμού.** Η grep { ... } %h διασχίζει την εναλλασσόμενη ισοπέδωση κλειδιού/τιμής του πίνακα κατακερματισμού· το μπλοκ βλέπει κλειδιά και τιμές μπλεγμένα στην $_. Αυτό σχεδόν ποτέ δεν είναι αυτό που θέλετε· φιλτράρετε τα κλειδιά και ξαναχτίστε αντ" αυτού:

```
my %filtered = map  { $_ => $h{$_} }
               grep { some_test($_, $h{$_}) } keys %h;
```

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- map - ίδιο συμβόλαιο επανάληψης με την ίδια σημασιολογία ψευδώνυμου της $_, αλλά επιστρέφει αυτό που *παρήγαγε* το μπλοκ για κάθε στοιχείο αντί για το ίδιο το στοιχείο
- sort - αναδιάταξη λίστας· συχνά αλυσιδώνεται με τη grep ως sort { ... } grep { ... } @list
- any / all - λογικά τεστ με βραχυκύκλωμα πάνω σε λίστα από την List::Util· χρησιμοποιήστε αυτά αντί για scalar grep όταν χρειάζεστε μόνο απάντηση ναι/όχι και η λίστα είναι μεγάλη ή το τεστ είναι δαπανηρό
- first (από την List::Util) - επιστρέφει το πρώτο στοιχείο που ταιριάζει με μπλοκ και σταματάει· το σωστό εργαλείο όταν η grep χρησιμοποιείται μόνο για

να ληφθεί μία αντιστοιχία

- `for` / `foreach` - η κατασκευή βρόχου με τον ίδιο κανόνα ψευδωνυμοποίησης της `$_`· καταφύγετε σε αυτήν όταν θέλετε παρενέργειες, όχι φιλτραρισμένη λίστα
- `$_` - το προεπιλεγμένο βαθμωτό που η `grep` ψευδωνυμοποιεί σε κάθε επανάληψη

Βαθμωτά και συμβολοσειρές · Αριθμητικές συναρτήσεις

hex

Ερμηνεύει μια συμβολοσειρά ως δεκαεξαδικό αριθμό και επιστρέφει την αριθμητική της τιμή.

Η `hex` διαβάζει την `EXPR` ως ακολουθία δεκαεξαδικών ψηφίων - προαιρετικά με πρόθεμα `0x` ή `x` - και επιστρέφει τον ακέραιο που αντιπροσωπεύουν τα ψηφία αυτά. Αν η `EXPR` παραλειφθεί, χρησιμοποιείται η `$_`. Η συμπληρωματική λειτουργία, μετατροπή αριθμού πίσω σε δεκαεξαδική συμβολοσειρά, είναι η `sprintf` με μορφή `"%x"` ή `"%X"`.

Σύνοψη

```
hex EXPR
hex                 # operates on $_
```

Τι επιστρέφεται

Ένας μη αρνητικός αριθμός. Η ίδια η `hex` δεν επιστρέφει ποτέ αρνητική τιμή: δεν έχει έννοια πρόσημου, αποκωδικοποιεί μόνο ψηφία.

- Αν το αποτέλεσμα χωράει σε εγγενή ακέραιο (τυπικά 64-bit μη προσημασμένο σε 64-bit build), παίρνετε ακέραιο.
- Αν η αποκωδικοποιημένη τιμή υπερβαίνει το εύρος αυτό, η `hex` υποχωρεί σε IEEE 754 double. Εκπέμπεται προειδοποίηση υπό `use warnings` και μπορεί να χαθεί ακρίβεια για τιμές πάνω από `2**53`.

Για μετατροπή προς την αντίθετη κατεύθυνση:

```
sprintf "%x", 255           # "ff"
sprintf "0x%08X", 4096      # "0x00001000"
```

Παραδείγματα

Σκέτα δεκαεξαδικά ψηφία, με ή χωρίς το πρόθεμα `0x`:

```
print hex '0xff';           # 255
print hex 'ff';             # 255
print hex 'DEADBEEF';       # 3735928559
```

Επιτρέπονται κάτω παύλες ανάμεσα σε ψηφία ως οπτικοί διαχωριστές, όπως ακριβώς τις δέχονται και τα αριθμητικά κυριολεκτικά:

```
print hex '1_0000';          # 65536
print hex 'cafe_f00d';       # 3405700109
```

Δράση επί της $_:

```
for ('10', '20', 'ff') {
    print hex, "\n";         # 16, 32, 255
}
```

Πλήρης γύρος μιας τιμής μέσω hex:

```
my $n = 0xC0FFEE;
my $s = sprintf "%x", $n;    # "c0ffee"
print hex $s;                # 12648430
```

Τιμές που υπερχειλίζουν το πλάτος εγγενούς ακέραιου προειδοποιούν και επιστρέφουν float:

```
use warnings;
my $big = hex 'ffffffffffffffffff';   # 19 'f's - too wide for u64
# "Integer overflow in hexadecimal number" warning
# $big is now an IEEE 754 double, with lost precision
```

Οριακές περιπτώσεις

- **Μη δεκαεξαδικοί χαρακτήρες τερματίζουν την ανάλυση.** Ο πρώτος χαρακτήρας που δεν είναι δεκαεξαδικό ψηφίο, κάτω παύλα ή αναγνωρισμένο πρόθεμα τερματίζει τη σάρωση. Υπό `use warnings` αυτό εκπέμπει `Illegal hexadecimal digit '…' ignored`:

  ```
  use warnings;
  print hex '1f garbage';    # 31, with a warning
  print hex '0xAZ';          # 10, with a warning on 'Z'
  ```

- **Τα προπορευόμενα κενά δεν παραλείπονται.** Σε αντίθεση με την `oct`, η hex αντιμετωπίζει τα κενά ως μη δεκαεξαδικό χαρακτήρα, οπότε ένα προπορευόμενο κενό σταματά την ανάλυση πριν εμφανιστεί οποιοδήποτε ψηφίο:

  ```
  print hex ' ff';           # 0
  ```

- **Η κενή συμβολοσειρά και η `undef` επιστρέφουν `0`.** Μια κενή EXPR δίνει `0` χωρίς προειδοποίηση. Μια σκέτη `undef` δίνει `0` και εκπέμπει τη συνήθη προειδοποίηση `Use of uninitialized value` υπό `use warnings`.

- **Καμία διαχείριση πρόσημου.** Ένα προπορευόμενο `-` ή `+` είναι μη δεκαεξαδικός χαρακτήρας και τερματίζει την ανάλυση πριν από οποιοδήποτε ψηφίο, οπότε η hex `'-ff'` επιστρέφει `0`. Για να αναλύσετε προσημασμένη δεκαεξαδική συμβολοσειρά, αφαιρέστε μόνοι σας το πρόσημο:

  ```
  my $s = '-ff';
  my $n = ($s =~ s/^-//) ? -hex($s) : hex($s);   # -255
  ```

- **Μορφές προθέματος.** Τα αποδεκτά προθέματα είναι `0x`, `0X`, `x` και `X`. Ένα σκέτο `0` δεν είναι πρόθεμα - είναι ψηφίο. Δεν υπάρχει πρόθεμα `#` ή `$`.
- **Η μορφή πεζών/κεφαλαίων είναι αδιάφορη** για τα ψηφία `a-f` και για το γράμμα προθέματος `x`.
- **Τοποθέτηση κάτω παύλας.** Μία μεμονωμένη κάτω παύλα μπορεί να προηγείται κάθε ψηφίου. Δύο διαδοχικές κάτω παύλες, μία προπορευόμενη κάτω παύλα πριν το πρόθεμα ή μία τελική κάτω παύλα τερματίζουν την ανάλυση ως μη δεκαεξαδικοί χαρακτήρες.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `oct` - αποκωδικοποιεί συμβολοσειρές με αυτο-εντοπιζόμενη βάση: `0x…` ως δεκαεξαδικό, `0b…` ως δυαδικό, `0…` ως οκταδικό, οτιδήποτε άλλο ως δεκαδικό. Χρησιμοποιήστε την όταν η βάση της εισόδου δεν είναι γνωστή εκ των προτέρων
- `sprintf` - η αντίστροφη κατεύθυνση· `"%x"` / `"%X"` μορφοποιούν έναν αριθμό σε δεκαεξαδική συμβολοσειρά με πεζά ή κεφαλαία, `"%#x"` προσθέτει το πρόθεμα `0x`, `"%08x"` συμπληρώνει με μηδενικά σε σταθερό πλάτος
- `printf` - ίδια μορφοποίηση `"%x"`, γραμμένη απευθείας σε filehandle
- `pack` - τα πρότυπα `H` (υψηλό nibble πρώτο) και `h` (χαμηλό nibble πρώτο) μετατρέπουν μεταξύ συμβολοσειράς δεκαεξαδικών ψηφίων και πακεταρισμένων δυαδικών bytes· χρησιμοποιήστε τα όταν μεταφέρετε δεκαεξαδικά δεδομένα από και προς ένα δυαδικό πρωτόκολλο
- `unpack` - το αντίστροφο της `pack`· με `"H*"` μετατρέπει μια συμβολοσειρά bytes σε συμβολοσειρά δεκαεξαδικών ψηφίων με μία κλήση

Εμβέλεια · Αρθρώματα

import

Συμπλήρωση του χώρου ονομάτων του καλούντος με ονόματα που επιλέγει να εξάγει ένα άρθρωμα.

Η `import` **δεν** είναι ενσωματωμένη συνάρτηση. Είναι μέθοδος κλάσης που ορίζει ένα άρθρωμα ώστε η `use` να έχει κάπου να παραδώσει τη λίστα ορισμάτων. Όταν γράφετε `use Module LIST`, η Perl φορτώνει το άρθρωμα και έπειτα καλεί `Module->import(LIST)` κατά τη μεταγλώττιση, στο πακέτο που περιέχει τη δήλωση `use`. Ό,τι κάνει αυτή η μέθοδος - εγκατάσταση υπορουτινών στο stash του καλούντος, αναστροφή bits χαρακτηριστικών, ορισμός σημαιών pragma - είναι ολόκληρο το συμβόλαιο `import` του αρθρώματος.

Τα περισσότερα αρθρώματα δεν γράφουν ποτέ `import` χειροκίνητα· την κληρονομούν από τον `Exporter` και δηλώνουν αντί αυτού `@EXPORT` / `@EXPORT_OK`. Τα pragmas (`strict`, `warnings`, `feature`) γράφουν τη δική τους `import` επειδή ό,τι εγκαθιστούν είναι λεξιλογική κατάσταση, όχι υπορουτίνες.

Σύνοψη

```
Module->import;
Module->import(LIST);
Module->import(@symbols);
```

Μέσα σε ένα άρθρωμα, ο ορισμός μοιάζει με:

```
package My::Module;
sub import {
    my ($class, @args) = @_;
    # install names into caller's package
}
```

Τι επιστρέφεται

Η τιμή επιστροφής της `import` απορρίπτεται από τη `use`. Ένα άρθρωμα που καλείται άμεσα ως μέθοδος κληρονομεί όποια σύμβαση επέλεξε ο συντάκτης - κατά σύμβαση, μην επιστρέφετε τίποτα χρήσιμο.

Το ορατό αποτέλεσμα είναι στο πακέτο του καλούντος: μετά την επιστροφή της `use Module qw(foo bar)`, οι `foo` και `bar` είναι καλέσιμες χωρίς πρόθεμα πακέτου από τον κώδικα που εκτέλεσε την `use`.

Πώς η use καλεί την import

Η `use Module LIST` ισοδυναμεί, ουσιαστικά, με:

```
BEGIN {
    require Module;
    Module->import(LIST);
}
```

Τρεις συνέπειες προκύπτουν από αυτή την ισοδυναμία:

- **Χρόνος μεταγλώττισης.** Η `import` τρέχει όσο το περικλείον αρχείο ακόμη αναλύεται. Τυχόν πρωτότυπα υπορουτίνας ή λεξιλογικά pragmas που εγκαθιστά είναι ορατά στο υπόλοιπο του αρχείου από εκείνο το σημείο και έπειτα. Οτιδήποτε γίνεται κατά τον χρόνο εκτέλεσης - άνοιγμα αρχείων, ανάγνωση ρυθμίσεων - συμβαίνει νωρίτερα από ό,τι υποδηλώνει μια αφελής ανάγνωση της πηγής.
- **Η `caller` αντικατοπτρίζει το μπλοκ `BEGIN`**, όχι την περιβάλλουσα υπορουτίνα. Μια μέθοδος `import` που επιθεωρεί την `caller(0)` βλέπει το πακέτο που έγραψε `use Module`, που είναι ακριβώς αυτό που θέλει· **δεν** βλέπει την υπορουτίνα που περιέχει τη δήλωση `use`.
- **Απούσα λίστα έναντι κενής λίστας.** Η `use Module` χωρίς λίστα καλεί `Module->import` χωρίς ορίσματα - το άρθρωμα εγκαθιστά τα προεπιλεγμένα του. Η `use Module ()` με **ρητή κενή λίστα** παρακάμπτει εντελώς την κλήση `import`. Αυτό είναι το ιδίωμα για «φόρτωσε το αρχείο αλλά μην αγγίξεις τον χώρο ονομάτων μου».

Η no Module LIST είναι η κατοπτρική εικόνα: καλεί Module->unimport(LIST) αντί για import. Τα pragmas το χρησιμοποιούν για να απενεργοποιήσουν χαρακτηριστικά για την περικλείουσα λεξιλογική εμβέλεια.

Ορισμός μιας μεθόδου import

Η κοινή διαδρομή είναι η κληρονομικότητα από τον Exporter:

```
package My::Utils;
use Exporter 'import';                  # installs Exporter::import
our @EXPORT_OK = qw(slurp trim);
sub slurp { ... }
sub trim  { ... }
1;
```

Οι καλούντες έπειτα γράφουν:

```
use My::Utils qw(slurp trim);
slurp($path);
```

Μια χειρόγραφη import έχει την ίδια μορφή με κάθε μέθοδο - λαμβάνει το όνομα κλάσης ως πρώτο όρισμα και τη λίστα ορισμάτων της use ως τα υπόλοιπα:

```
package My::Feature;
sub import {
    my ($class, @flags) = @_;
    my $caller = caller;
    no strict 'refs';
    for my $name (@flags) {
        *{ "${caller}::${name}" } = \&{ "${class}::${name}" };
    }
}
```

Το όνομα πακέτου του καλούντος προέρχεται από την caller, όχι από το $class - το $class είναι όποιο άρθρωμα ονομάτισε ο χρήστης στη γραμμή use.

Παραδείγματα

Προεπιλεγμένη εισαγωγή - ό,τι περιέχει η λίστα @EXPORT του αρθρώματος:

```
use List::Util;                         # imports nothing by default
use File::Basename;                     # imports basename, dirname,␣
↪...
```

Ρητή λίστα εισαγωγής - μόνο τα ονόματα που ζητάτε:

```
use List::Util qw(sum min max);
my $total = sum(1, 2, 3);               # 6
```

Κενή λίστα - φορτώστε το αρχείο, μην αγγίξετε τίποτα:

```
use POSIX ();                           # no POSIX::* names imported
my $pi = POSIX::acos(-1);               # fully qualified call
```

Εισαγωγή από pragma - λεξιλογικό αποτέλεσμα στο περικλείον μπλοκ:

```
use strict;
use warnings;
{
    no warnings 'uninitialized';        # unimport in this block only
    print "x=", $x, "\n";
}
```

Άμεση κλήση της `import` μετά από `require` κατά τον χρόνο εκτέλεσης:

```
require My::Plugin;
My::Plugin->import(@config);            # deferred to runtime on
↪purpose
```

Αυτό το μοτίβο φορτώνει το άρθρωμα μόνο όταν μια διαδρομή κώδικα πραγματικά το χρειάζεται, ανταλλάσσοντας έλεγχο κατά τη μεταγλώττιση με ταχύτητα εκκίνησης.

Οριακές περιπτώσεις

- **Δεν υπάρχει ενσωματωμένη `import`.** Η `print import(@list)` καλεί την υπορουτίνα ονόματι `import` στο τρέχον πακέτο (αν υπάρχει)· δεν είναι τελεστής σε επίπεδο γλώσσας. Οι κανόνες ανάλυσης ισχύουν όπως για οποιαδήποτε υπορουτίνα ορισμένη από τον χρήστη.
- **Παράλειψη λίστας έναντι κενής λίστας** είναι η πιο κοινή πηγή σύγχυσης. Η `use Mod` καλεί την `import`· η `use Mod ()` όχι. Το `()` δεν είναι στιλιστικό κενό - είναι σήμα.
- **Το όρισμα έκδοσης εκτελεί την `VERSION`, όχι την `import`.** Η `use Mod 1.23 LIST` καλεί `Mod->VERSION(1.23)` **πριν** την `Mod->import(LIST)`. Μια σκέτη `use Mod 1.23` εξακολουθεί να είναι έλεγχος έκδοσης ακολουθούμενος από προεπιλεγμένη `import`. Για έλεγχο έκδοσης χωρίς εισαγωγές, γράψτε `use Mod 1.23 ()`.
- **Καθόλου ορισμένη `import`.** Αν το άρθρωμα ούτε γράφει ούτε κληρονομεί `import`, η `use Module LIST` με μη κενή λίστα δεν παράγει σφάλμα - η κλήση μεθόδου περνά σιωπηρά καθώς η `UNIVERSAL::can('import')` επιστρέφει ψευδές και η `use` την αντιμετωπίζει ως «τίποτα να γίνει». Η μη κενή `LIST` τότε αγνοείται στην πράξη, κάτι που σχεδόν ποτέ δεν είναι ό,τι ήθελε ο καλών.
- **Η `import` τρέχει μία φορά ανά δήλωση `use`**, όχι μία φορά ανά διεργασία. Η `use Mod qw(foo); use Mod qw(bar);` καλεί την `import` δύο φορές, παρότι το αρχείο φορτώνεται μόνο μία - οι `@INC` / `%INC` ελέγχουν τη φόρτωση του αρχείου, όχι την κλήση μεθόδου.
- **Η εγκατάσταση συμβόλων δεν είναι αυτόματη.** Η γραφή μεθόδου `import` που απλώς επιστρέφει δεν κάνει τίποτα· πρέπει να εκχωρήσετε στο stash του καλούντος (`*{"${caller}::name"} = \&sub`) ή να αναθέσετε στον `Exporter`.

- **Τα pragmas τυπικά εγκαθιστούν λεξιλογική κατάσταση**, όχι υπορουτίνες σε επίπεδο πακέτου. Η `import` τους καλεί συναρτήσεις όπως `warnings::import_into` ή χειρίζεται την %^H· γι" αυτό το αποτέλεσμα ενός pragma έχει εμβέλεια το περικλείον μπλοκ αντί να διατηρείται στο πακέτο.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `use` - ο φορτωτής κατά τη μεταγλώττιση που επικαλείται την `import`· η κατανόηση του ενός απαιτεί το άλλο
- `no` - ο κάτοπτρος της `use`· καλεί την `unimport` με τους ίδιους κανόνες πέρασης ορισμάτων
- `require` - φορτώνει το αρχείο χωρίς να καλεί `import`· συνδυάστε με ρητό `Module->import(...)` όταν θέλετε να αναβάλετε ή να παρακάμψετε την αποστολή κατά τη μεταγλώττιση
- `package` - ορίζει τον χώρο ονομάτων που μια μέθοδος `import` συμπληρώνει από την οπτική του καλούντος
- `caller` - πώς μια μέθοδος `import` ανακαλύπτει σε ποιο πακέτο να εγκαταστήσει ονόματα
- `bless` - άσχετη με την `import`, αλλά συχνά επικαλείται στα ίδια αρχεία αρθρωμάτων· και τα δύο διαμορφώνουν τον τρόπο με τον οποίο ένα πακέτο παρουσιάζεται στους χρήστες του

Βαθμωτά και συμβολοσειρές

index

Βρίσκει τη θέση μιας υποσυμβολοσειράς μέσα σε μια συμβολοσειρά.

Η `index` σαρώνει τη `STR` από αριστερά προς τα δεξιά αναζητώντας την πρώτη εμφάνιση της `SUBSTR` και επιστρέφει τη θέση με βάση το μηδέν όπου ξεκινά. Καμία μετα-χαρακτήρας κανονικής έκφρασης, καμία πτύχωση πεζών-κεφαλαίων, κανένα wildcard - η `SUBSTR` αντιστοιχίζεται κυριολεκτικά, χαρακτήρα προς χαρακτήρα. Όταν η αναζήτηση αποτύχει, η `index` επιστρέφει `-1`.

Σύνοψη

```
index STR, SUBSTR
index STR, SUBSTR, POSITION
```

Τι επιστρέφεται

Έναν ακέραιο. Σε αντιστοιχία, το μηδενοβάσει offset του πρώτου χαρακτήρα της `SUBSTR` εντός της `STR`. Σε καμία αντιστοιχία, `-1`. Η τιμή-φρουρά είναι ο ιδιωματικός τρόπος ελέγχου:

```
if (index($line, $needle) >= 0) { ... }   # found
if (index($line, $needle) == -1) { ... }  # not found
```

Το POSITION και η τιμή επιστροφής χρησιμοποιούν την **ίδια** μηδενοβάσει κλίμακα, οπότε μπορείτε να τροφοδοτείτε τη μία στην άλλη για να διατρέξετε κάθε εμφάνιση:

```
my $pos = -1;
while (($pos = index($text, $needle, $pos + 1)) != -1) {
    push @hits, $pos;
}
```

Πώς ερμηνεύεται το POSITION

Το POSITION είναι το πρωιμότερο offset στο οποίο επιτρέπεται να ξεκινήσει η αντιστοιχία. Η αναζήτηση εξακολουθεί να προχωρά μέχρι το τέλος της STR· το POSITION δεν περιορίζει την αναζήτηση, απλώς μετατοπίζει το σημείο εκκίνησής της.

- POSITION παραλειπόμενο ή undef - αναζήτηση από offset 0.
- POSITION αρνητικό ή αλλιώς πριν την αρχή - αντιμετωπίζεται ως 0.
- POSITION πέρα από το τέλος της STR - αντιμετωπίζεται ως το τέλος, οπότε ο μόνος τρόπος αντιστοιχίας είναι αν η SUBSTR είναι η κενή συμβολοσειρά (που αντιστοιχίζεται σε κάθε offset, συμπεριλαμβανομένου του τέλους).

Μια κενή SUBSTR ταιριάζει πάντα, και ταιριάζει στο POSITION (περικομμένο εντός εύρους). Αυτό απορρέει από τον κανόνα «η πρώτη θέση όπου εμφανίζεται η SUBSTR»: η κενή συμβολοσειρά εμφανίζεται παντού.

```
index("hello", "");      # 0
index("hello", "", 3);   # 3
index("hello", "", 99);  # 5   (clamped to end of string)
```

Παραδείγματα

Εύρεση ενός μόνου χαρακτήρα ή μιας ολόκληρης λέξης:

```
index("Perl is great", "P");     # 0
index("Perl is great", "g");     # 8
index("Perl is great", "great"); # 8
```

Αναφορά αστοχίας:

```
index("Perl is great", "Z");     # -1
```

Παράκαμψη προηγούμενης αντιστοιχίας με POSITION για εύρεση της δεύτερης εμφάνισης:

```
index("Perl is great", "e", 5);  # 10
```

Διάσχιση κάθε εμφάνισης μιας υποσυμβολοσειράς:

```
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
```

Ένα συνηθισμένο ιδίωμα - έλεγχος εμπεριέχειας χωρίς δημιουργία regex:

```
if (index($path, "/tmp/") != -1) {
    warn "path touches /tmp";
}
```

Συνδυάζεται φυσικά με `substr` για διαχωρισμό στην πρώτη εμφάνιση ενός διαχωριστή:

```
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);
```

Οριακές περιπτώσεις

- **Κενή SUBSTR** ταιριάζει στο POSITION (περικομμένο εντός της STR). Η `index($s, "")` είναι `0`· η `index($s, "", $n)` είναι το `$n` περιορισμένο μέχρι το `length $s`. Ποτέ `-1`.
- **Κενή STR με μη κενή SUBSTR** επιστρέφει `-1`. Κενή STR με κενή SUBSTR επιστρέφει `0`.
- **Αρνητικό POSITION** περικόπτεται στο `0`. Η `index` **δεν** ερμηνεύει αρνητικά offsets ως «από το τέλος» - αυτή είναι η δουλειά της `rindex`, και ακόμη και εκεί η σημασιολογία διαφέρει.
- **POSITION πέρα από το τέλος της STR** περικόπτεται στο `length STR`, οπότε μόνο μια κενή SUBSTR μπορεί να ταιριάξει.
- **Ορίσματα undef** μετατρέπονται σε `""` και πυροδοτούν προειδοποίηση `uninitialized` υπό `use warnings`. Η `index(undef, "x")` είναι `-1`· η `index("abc", undef)` είναι `0` (κανόνας κενής υποσυμβολοσειράς).
- **Χαρακτήρες, όχι bytes.** Η `index` λειτουργεί επί της λογικής ακολουθίας χαρακτήρων της συμβολοσειράς. Για συμβολοσειρά πλατιών χαρακτήρων, το επιστρεφόμενο offset είναι offset χαρακτήρων, όχι bytes. Αν χρειάζεστε offsets bytes, υποβαθμίστε ή κωδικοποιήστε πρώτα τη συμβολοσειρά (`use bytes` για λεξιλογική θέαση σε bytes, ή `Encode::encode_utf8` για να εργαστείτε σε συμβολοσειρά οκτάδων).
- **Διάκριση πεζών-κεφαλαίων**: η `index` λειτουργεί με διάκριση πεζών-κεφαλαίων. Μετατρέψτε πρώτα και τα δύο ορίσματα σε πεζά αν θέλετε αναζήτηση χωρίς διάκριση πεζών-κεφαλαίων, ή χρησιμοποιήστε `=~ /\Q$needle\E/i` και `@-` / `$-[0]` για να ανακτήσετε τη θέση.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `rindex` - ίδιοι κανόνες αντιστοίχισης, σαρώνει από τα δεξιά και επιστρέφει το offset της **τελευταίας** εμφάνισης στο ή πριν το `POSITION`
- `substr` - εξαγωγή της περιοχής αντιστοίχισης αφού την εντοπίσει η `index`, ή αντικατάστασή της επιτόπου
- `length` - άνω όριο για έγκυρο `POSITION`· επιστρέφει μήκος χαρακτήρων στην ίδια κλίμακα που χρησιμοποιεί η `index`
- `pos` - παρακολούθηση θέσης για σάρωση βασισμένη σε regex· χρησιμοποιήστε μαζί με `m//g` όταν χρειάζεστε συλλήψεις αντί για ακατέργαστο offset
- `sprintf` - δημιουργία της συμβολοσειράς αναζήτησης όταν η `SUBSTR` συναρμολογείται από μέρη· η `index` δέχεται κυριολεκτική, οπότε προετοιμάστε την πρώτα

Αριθμητικές συναρτήσεις

int

Επιστρέφει το ακέραιο μέρος ενός αριθμού, με αποκοπή προς το μηδέν.

Η `int` παίρνει ένα βαθμωτό, το εξαναγκάζει σε αριθμητικό περιβάλλον, και απορρίπτει το κλασματικό του μέρος. Η απόρριψη γίνεται πάντα **προς το μηδέν** - όχι προς το αρνητικό άπειρο - οπότε η `int(3.9)` δίνει `3` και η `int(-3.9)` δίνει `-3`, όχι `-4`. Οι μη αριθμητικές συμβολοσειρές εξαναγκάζονται με τους συνήθεις κανόνες αριθμητικής μετατροπής της Perl (αρχικά κενά, προαιρετικό πρόσημο, ψηφία· το υπόλοιπο απορρίπτεται, με προειδοποίηση `Argument "..." isn't numeric` υπό `use warnings`). Αν παραλειφθεί το `EXPR`, η `int` λειτουργεί στο `$_`.

Σύνοψη

```
int EXPR
int
int($x)
```

Τι επιστρέφεται

Ένας αριθμός χωρίς κλασματικό μέρος, με το ίδιο πρόσημο με την είσοδο. Το αποτέλεσμα είναι ακέραια τιμή αν το αποκομμένο μέτρο χωρά στο εγγενές εύρος ακεραίων της πλατφόρμας (τυπικά 64-bit)· διαφορετικά η Perl το διατηρεί ως τιμή κινητής υποδιαστολής της οποίας το κλασματικό μέρος τυχαίνει να είναι μηδέν. Σε κάθε περίπτωση συγκρίνεται και εκτυπώνεται ως ακέραιος.

```
my $n = int(7.8);           # 7
my $m = int(-7.8);          # -7
```

Η `int` δεν είναι συνάρτηση στρογγυλοποίησης. Για στρογγυλοποίηση, δείτε *Δείτε επίσης*.

Παραδείγματα

Αποκοπή θετικού - το κλασματικό μέρος απορρίπτεται:

```
print int(3.9), "\n";               # 3
print int(3.1), "\n";               # 3
print int(3.0), "\n";               # 3
```

Αποκοπή αρνητικού - η παγίδα. Η `int` αποκόπτει προς το μηδέν, οπότε η `int(-3.9)` είναι `-3`, που είναι *μεγαλύτερο* από `-3.9`. Αυτό είναι το αντίθετο της μαθηματικής floor:

```
print int(-3.9), "\n";              # -3   (not -4)
print int(-3.1), "\n";              # -3
```

Λειτουργεί στο `$_` όταν καλείται χωρίς ορίσματα - χρήσιμο μέσα σε `map` ή `while (<>)`:

```
my @whole = map { int } @floats;    # truncate every element
```

Οι συμβολοσειρές εξαναγκάζονται, διατηρώντας το αρχικό αριθμητικό πρόθεμα:

```
print int("42.7 bottles"), "\n";    # 42
print int("  -5.5xyz"), "\n";       # -5
print int("hello"), "\n";           # 0    (with a warning under
↪`use warnings`)
```

Διαίρεση που παράγει κλάσμα, μετά αποκόπτεται - συνηθισμένο ιδίωμα για ακέραια διαίρεση προσημασμένων τιμών, **με την παραπάνω επιφύλαξη για τα αρνητικά**:

```
my $pages = int($total_items / $per_page);   # floor only for non-
↪negative $total_items
```

Η κλασική έκπληξη της κινητής υποδιαστολής. Το `-6.725 / 0.025` είναι *μαθηματικά* `-269`, αλλά το αποτέλεσμα κατά IEEE-754 είναι ελαφρώς μεγαλύτερο από `-269` (περίπου `-268.99999999999994`). Η `int` αποκόπτει προς το μηδέν, δίνοντας `-268`:

```
print int(-6.725 / 0.025), "\n";    # -268, not -269
```

Αυτό δεν είναι bug της `int` - κάθε συνάρτηση αποκοπής/στρογγυλοποίησης πρέπει να αντιμετωπίσει εισόδους όπως αυτή. Δείτε *Οριακές περιπτώσεις* παρακάτω.

Οριακές περιπτώσεις

- **Η `int` στο `$_`**: χωρίς όρισμα, λειτουργεί στο `$_`. Μέσα σε `map { int }` αυτό είναι που θέλετε· μέσα σε μεγαλύτερη έκφραση μπορεί να ξαφνιάσει αναγνώστη που έχασε το σιωπηρό όρισμα. Προτιμήστε `int($x)` όταν η μεταβλητή πηγής δεν είναι προφανής από το συγκείμενο.
- **Είσοδος που είναι ήδη ακέραιος**: η `int(5)` δίνει `5`. Καμία μετατροπή, καμία προειδοποίηση, καμία απώλεια ακρίβειας.

- **Οι πολύ μεγάλοι αριθμοί κινητής υποδιαστολής χάνουν ακρίβεια πριν τους δει η `int`**. Ένας αριθμός κινητής υποδιαστολής διπλής ακρίβειας μπορεί να αναπαραστήσει επακριβώς κάθε ακέραιο έως το `2**53`· πέρα από αυτό, διαδοχικές αναπαραστάσιμες τιμές απέχουν 2, μετά 4, μετά 8. Η `int(1e20)` επιστρέφει αριθμό κινητής υποδιαστολής του οποίου η ακέραια τιμή είναι κοντά αλλά όχι ίση με `10**20`. Αν χρειάζεστε ακριβείς μεγάλους ακέραιους, κρατήστε τους ως ακέραιους σε όλη τη διαδρομή - μην τους δρομολογήσετε μέσω κινητής υποδιαστολής.
- **Παρενέργειες αναπαράστασης κινητής υποδιαστολής**: είσοδοι όπως `-6.725 / 0.025` (δείτε *Παραδείγματα*) δεν προσγειώνονται στον μαθηματικό ακέραιο που «θα έπρεπε». Χρησιμοποιήστε `sprintf "%.0f"` ή `POSIX::floor`/`POSIX::ceil` αν χρειάζεστε συμπεριφορά στρογγυλοποίησης που ανέχεται αυτό.
- **Άπειρα και NaN**: η `int('Inf')` επιστρέφει `Inf`· η `int('-Inf')` επιστρέφει `-Inf`· η `int('NaN')` επιστρέφει `NaN`. Η `int` δεν έχει ακέραιο στον οποίο να αποκόψει, οπότε επιστρέφει την ειδική τιμή αμετάβλητη. Αυτό ταιριάζει με κάθε εναλλακτική (`sprintf "%d"`, `POSIX::floor`, `POSIX::ceil`) - καμία τους δεν σας δίνει πεπερασμένο ακέραιο εδώ.
- **`undef`**: η `int(undef)` επιστρέφει `0` και εκπέμπει προειδοποίηση `Use of uninitialized value` υπό `use warnings`.
- **Μη αριθμητικές συμβολοσειρές**: η `int("abc")` επιστρέφει `0` και εκπέμπει `Argument "abc" isn't numeric` υπό `use warnings`. Συμβολοσειρά που ξεκινά με αριθμητικό πρόθεμα (`"42abc"`) αποδίδει την ακέραια τιμή του προθέματος (42) με την ίδια προειδοποίηση - η `int` το κληρονομεί αυτό από τον τυπικό αριθμητικό εξαναγκασμό της Perl, όχι από κάποια προσαρμοσμένη ανάλυση.
- **Λογικές τιμές και dualvars**: η `int` διαβάζει το αριθμητικό μισό. Η `int(!!1)` δίνει `1`· η `int(!!0)` δίνει `0`. Για dualvar του οποίου οι συμβολοσειριακές και αριθμητικές τιμές διαφωνούν, η `int` χρησιμοποιεί την αριθμητική πλευρά.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `abs` - μέτρο χωρίς πρόσημο· συνδυάστε την με την `int` όταν θέλετε το αποκομμένο μέτρο ανεξάρτητα από πρόσημο
- `sprintf` - η `sprintf "%d", $x` επίσης αποκόπτει προς το μηδέν, αλλά επιπλέον μορφοποιεί το αποτέλεσμα ως συμβολοσειρά· χρησιμοποιήστε την όταν θέλετε την αποκομμένη τιμή *και* μια συγκεκριμένη μορφή κειμένου
- `POSIX::floor` - στρογγυλοποίηση *προς το αρνητικό άπειρο*· η `floor(-3.9)` είναι `-4`, που είναι αυτό που πραγματικά θέλουν τα περισσότερα μαθηματικά περιβάλλοντα (γραφικά, bucketing)
- `POSIX::ceil` - στρογγυλοποίηση *προς το θετικό άπειρο*· η `ceil(-3.1)` είναι `-3`, η `ceil(3.1)` είναι `4`· το κάτοπτρο της `floor`
- `POSIX::round` - στρογγυλοποίηση στον πλησιέστερο ακέραιο, με τα μισά μακριά από το μηδέν· χρησιμοποιήστε την όταν θέλετε στατιστική στρογγυλοποίηση αντί για αποκοπή

- `sprintf` με `"%.0f"` - στρογγυλοποίηση στον πλησιέστερο ακέραιο, με τα μισά σε άρτιο (τραπεζική στρογγυλοποίηση) στις περισσότερες πλατφόρμες· διακριτικά διαφορετική από την `POSIX::round` στις μέσες τιμές

Filehandles, αρχεία, κατάλογοι

ioctl

Εκτέλεση κλήσης συστήματος ελέγχου συσκευής `ioctl(2)` σε filehandle.

Η `ioctl` είναι η άμεση σύνδεση Perl για την κλήση συστήματος POSIX `ioctl(2)` - τη γενική διεπαφή πυρήνα για ειδικές λειτουργίες συσκευής που δεν ταιριάζουν στο τυπικό μοντέλο read/write: μέγεθος παραθύρου τερματικού, παράμετροι σειριακής γραμμής, ερωτήματα σε συσκευή υποδοχής, τοποθέτηση ταινίας, και ό,τι άλλο επιλέγει να εκθέσει ένας οδηγός μέσω κωδικού αιτήματος. Αποστέλλει την `FUNCTION` έναντι του `FILEHANDLE`, περνώντας το `SCALAR` ως τρίτο όρισμα - είτε έναν μικρό ακέραιο είτε δείκτη σε buffer bytes, ανάλογα με το τι αναμένει η συγκεκριμένη `FUNCTION`.

Σε αντίθεση με την `fcntl`, της οποίας οι κωδικοί αιτημάτων βρίσκονται στο άρθρωμα `Fcntl`, οι κωδικοί αιτημάτων `ioctl` σχεδόν ποτέ δεν εξάγονται ως έτοιμες σταθερές. Η συνήθης διαδρομή είναι:

```
require "sys/ioctl.ph";
```

που εισάγει το μεταφρασμένο header `<sys/ioctl.h>`. Αν αυτό το αρχείο δεν υπάρχει, ή δεν ορίζει τον κωδικό που χρειάζεστε, κατασκευάστε το από τα headers C με το εργαλείο h2ph που συνοδεύει την Perl, ή ορίστε τη σταθερά μόνοι σας από το header του πυρήνα.

Σύνοψη

```
require "sys/ioctl.ph";
ioctl FILEHANDLE, FUNCTION, SCALAR
ioctl($fh, $request, $buf)
my $rv = ioctl($fh, $request, $arg) || -1;
```

Τι επιστρέφεται

Σε επιτυχία, ο ακέραιος που επέστρεψε ο πυρήνας. Όταν αυτός ο ακέραιος είναι `0`, η Perl υποκαθιστά τη διπλής τιμής συμβολοσειρά `"0 but true"` - αληθής σε λογικό περιβάλλον, αριθμητικό `0` σε αριθμητικό περιβάλλον, και εξαιρούμενη από την προειδοποίηση `Argument "..." isn't numeric`. Αυτό σημαίνει ότι ένας απλός λογικός έλεγχος διακρίνει αξιόπιστα την επιτυχία από την αποτυχία ακόμη και για αιτήματα των οποίων η τιμή επιτυχίας είναι κυριολεκτικά μηδέν:

```
ioctl($fh, $request, $buf) or die "ioctl: $!";
```

Σε αποτυχία, η `ioctl` επιστρέφει `undef` και θέτει την `$!` στο σχετικό `errno`. Η πλήρης αντιστοίχιση:

Επιστροφή OS	Επιστροφή Perl
-1	undef
0	συμβολοσειρά "0 but true"
οποιαδήποτε άλλη τιμή	αυτός ο ακέραιος

Όταν χρειάζεστε την αδρή επιστροφή του πυρήνα - για κωδικούς αιτημάτων που κωδικοποιούν πληροφορία στην ίδια την τιμή επιστροφής - χρησιμοποιήστε το ιδίωμα από το perlfunc:

```
my $retval = ioctl($fh, $request, $arg) || -1;
printf "System returned %d\n", $retval;
```

Πώς χρησιμοποιείται το SCALAR

Το SCALAR παραδίδεται στον πυρήνα ως τρίτο όρισμα της κλήσης C `ioctl`, και ο ρόλος του εξαρτάται από τον κωδικό αιτήματος:

- **Δείκτης σε buffer.** Η συνήθης περίπτωση. Τα περισσότερα αιτήματα `ioctl` διαβάζουν από ή γράφουν σε ένα C struct του οποίου η διάταξη ορίζεται από τον οδηγό. Κατασκευάστε τον buffer με pack πριν την κλήση και, για αιτήματα που γράφουν πίσω, αποκωδικοποιήστε με unpack έπειτα. Προδιαστασιολογήστε το βαθμωτό στο μήκος του struct τουλάχιστον - η Perl θα επεκτείνει αυτόματα ένα μικρότερο βαθμωτό, αλλά η προδιαστασιολόγηση καθιστά την πρόθεση προφανή και αποφεύγει εκπλήξεις αν ο οδηγός γράψει περισσότερα από ό,τι αναμένατε.
- **Μικρός ακέραιος.** Μια μειονότητα αιτημάτων παίρνει σκέτο ακέραιο αντί για δείκτη. Αν το SCALAR δεν έχει συμβολοσειριακή τιμή αλλά έχει αριθμητική, η Perl περνά τον αριθμό αντί για δείκτη. Για να εγγυηθείτε αυτή την αποστολή - ακόμα και για ένα βαθμωτό που μπορεί να έχει μετατραπεί σε συμβολοσειρά παλιότερα - προσθέστε του `0` πρώτα:

  ```
  my $n = 0 + $n;       # force numeric representation
  ioctl($fh, $request, $n);
  ```

- **Αγνοείται.** Για κωδικούς αιτημάτων που δεν παίρνουν τρίτο όρισμα, περάστε `0` ως placeholder. Ο πυρήνας το αγνοεί· η σύμβαση κρατά την κλήση αναγνώσιμη.

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται σε αποτυχία στο `errno` που επιστρέφει η `ioctl(2)`.
- Κατάσταση συσκευής ή περιγραφέα στο FILEHANDLE - η `ioctl` τυπικά μεταβάλλει κατάσταση πυρήνα σχετιζόμενη με τον υποκείμενο περιγραφέα αρχείου ή τη συσκευή του (χαρακτηριστικά τερματικού, επιλογές υποδοχής, θέση ταινίας). Αυτή η κατάσταση δεν είναι ορατή στην Perl μέσω άλλης μεταβλητής· διαβάστε την ξανά με άλλη κλήση `ioctl` χρησιμοποιώντας το αντίστοιχο αίτημα `TIOCG...` / `SIOCG...`.

Παραδείγματα

Ερώτημα μεγέθους παραθύρου τερματικού σε Linux. Το `TIOCGWINSZ` γράφει μια `struct winsize` (δύο `unsigned short` γραμμές/στήλες συν δύο πεδία που αγνοούνται) στον 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";
```

Μέτρηση των διαθέσιμων bytes για ανάγνωση σε υποδοχή χωρίς να καταναλωθούν (το `FIONREAD` υποστηρίζεται ευρέως σε όλα τα Unix):

```
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";
```

Θέση μιας υποδοχής σε μη μπλοκαριστική λειτουργία μέσω του αιτήματος `FIONBIO` - εναλλακτική στην προσέγγιση `fcntl` / `F_SETFL`, και η μορφή που ιστορικά ήταν φορητή σε ορισμένα παλαιότερα συστήματα:

```
require "sys/ioctl.ph";

my $on = pack('L', 1);
ioctl($sock, FIONBIO(), $on)
    or die "FIONBIO: $!";
```

Διάκριση μεταξύ «η κλήση συστήματος επέστρεψε 0» και «η κλήση συστήματος απέτυχε» χρησιμοποιώντας τη σύμβαση `"0 but true"`:

```
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 {
    # kernel returned $rv
}
```

Κατασκευή του δικού σας κωδικού αιτήματος όταν το `sys/ioctl.ph` λείπει ή είναι ελλιπές. Η κωδικοποίηση (`_IOR` / `_IOW` / `_IOWR`) είναι ειδική του πυρήνα· σε Linux η μακροεντολή αναπτύσσεται σε τιμή 32 bit που μπορείτε να προ-υπολογίσετε και να κωδικοποιήσετε σταθερά:

```
use constant TIOCGWINSZ => 0x5413;   # Linux <asm-generic/ioctls.h>

my $winsize = "\0" x 8;
ioctl(STDOUT, TIOCGWINSZ, $winsize) or die $!;
```

Οριακές περιπτώσεις

- **Παράλειψη του `.ph` `require`**: σταθερές όπως `TIOCGWINSZ` αναλύονται ως barewords χωρίς ορισμένη τιμή, πυροδοτώντας `EINVAL` από τον πυρήνα. Είτε `require "sys/ioctl.ph"` είτε ορίστε τη σταθερά μόνοι σας με `use constant`.
- **Επιστροφή `"0 but true"`**: μη συγκρίνετε την επιστροφή με `== 0` ως έλεγχο αποτυχίας. Χρησιμοποιήστε απλή λογική τιμή (`or die`) ή `defined`. Η σύγκριση ισότητας με την κυριολεκτική συμβολοσειρά λειτουργεί επίσης (`$rv eq "0 but true"`) αλλά σπάνια χρειάζεται.
- **Έκπληξη αποστολής βαθμωτό/ακέραιος**: ένα βαθμωτό που ήταν αριθμητικό αλλά μετατράπηκε σε συμβολοσειρά - για παράδειγμα μέσω παρεμβολής σε μήνυμα - αντιστρέφεται από σημασιολογία ορίσματος-ακεραίου σε σημασιολογία δείκτη-buffer στην επόμενη κλήση. Εξαναγκάστε αριθμητική αναπαράσταση με `0 + $x` όταν ο κωδικός αιτήματος αναμένει ακέραιο.
- **Η διάταξη struct είναι ειδική της πλατφόρμας.** Οι `struct winsize`, `struct termios`, `struct ifreq` και άλλες διαφέρουν μεταξύ πυρήνων και ακόμη μεταξύ αρχιτεκτονικών στον ίδιο πυρήνα. Μην κωδικοποιείτε σταθερά πρότυπα `pack` από μνήμη· επαληθεύστε έναντι του header του συστήματος-στόχου.
- **Υπομεγέθης buffer εξόδου.** Αν ο οδηγός γράψει περισσότερα bytes από όσα κρατά αυτή τη στιγμή το `SCALAR`, το βαθμωτό επεκτείνεται, αλλά δεν έχετε καμία ένδειξη στην Perl ότι αυτό συνέβη. Προδιαστασιολογήστε στο μήκος του struct τουλάχιστον με `"\0" x N` ώστε αναγνώσεις πέρα από το τέλος του buffer να είναι σφάλματα οδηγού, όχι Perl.
- **Δεν υλοποιεί κάθε σύστημα κάθε αίτημα.** Η ίδια η `ioctl` είναι POSIX, αλλά οι συγκεκριμένοι κωδικοί αιτημάτων είναι ορισμένοι από τον οδηγό. Ένα μη υποστηριζόμενο αίτημα αποτυγχάνει με `ENOTTY` ή `EINVAL`· μια κλήση `ioctl` σε σύστημα χωρίς καθόλου `ioctl(2)` εγείρει εξαίρεση.
- **`ioctl` έναντι `fcntl`**: πανομοιότυπη επεξεργασία ορισμάτων και σύμβαση τιμής επιστροφής, διαφορετική διεπαφή πυρήνα. Χρησιμοποιήστε την `fcntl` για σημαίες περιγραφέων και POSIX locks· χρησιμοποιήστε την `ioctl` για κωδικούς ελέγχου συσκευής.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `fcntl` - ίδια σύμβαση κλήσης και τιμής επιστροφής· χρησιμοποιείται για σημαίες σε επίπεδο περιγραφέα και advisory locks παρά για ειδικούς κωδικούς ελέγχου συσκευής

- `pack` / `unpack` - απαιτούνται για κατασκευή και αποκωδικοποίηση των struct buffers που τα περισσότερα αιτήματα `ioctl` ανταλλάσσουν με τον πυρήνα
- `open` / `sysopen` - άνοιγμα της συσκευής ή του ειδικού αρχείου με τον οδηγό του οποίου θα μιλήσει το αίτημα `ioctl`
- `binmode` - ορισμένα αιτήματα `ioctl` (π.χ. αλλαγές λειτουργίας τερματικού) αλληλεπιδρούν με την ενταμίευση PerlIO· αλλάξτε σε raw mode όταν ο οδηγός και το PerlIO διαφωνούν για το line discipline
- `select` / `sysread` - η κανονική διαδρομή I/O· καταφεύγετε στην `ioctl` μόνο όταν η λειτουργία που χρειάζεστε δεν εκφράζεται ως read ή write

Κλάσεις και αντικειμενοστρέφεια

isa

Ελέγχει αν ένα αντικείμενο είναι στιγμιότυπο μιας κλάσης ή οποιασδήποτε υποκλάσης που παράγεται από αυτήν.

Η `isa` είναι **δυαδικός τελεστής**, όχι συνάρτηση. Ο αριστερός τελεστέος είναι μια έκφραση που θα πρέπει να αποδώσει αναφορά αντικειμένου· ο δεξιός τελεστέος κατονομάζει μια κλάση. Επιστρέφει αληθές όταν ο αριστερός τελεστέος είναι blessed αναφορά της οποίας η κλάση είναι - ή κληρονομεί από - την κλάση στα δεξιά. Επιστρέφει ψευδές για οτιδήποτε άλλο, περιλαμβανομένων των `undef`, μη-blessed αναφορών και απλών συμβολοσειρών. Δεν εκτοξεύει ποτέ.

Σε αντίθεση με την παλιά μέθοδο `UNIVERSAL::isa`, η `isa` είναι ασφαλής να εφαρμοστεί σε αυθαίρετες τιμές: η κλήση `$x->isa("Foo")` σε απλή συμβολοσειρά που τυχαίνει να κατονομάζει μια κλάση (`$x = "Foo"`) επικαλείται τη μέθοδο κλάσης και επιστρέφει αληθές, που σχεδόν ποτέ δεν είναι αυτό που εννοούσε ο καλών. Η `isa` ως τελεστής αναφέρει αληθές μόνο για **blessed στιγμιότυπα**.

Σύνοψη

```
use feature 'isa';          # or: use v5.36;

$obj isa Some::Class
$obj isa "Different::Class"
$obj isa $name_of_class
```

Τι επιστρέφεται

1 για blessed αντικείμενο του οποίου η κλάση είναι, ή παράγεται από, την `CLASS`. Την ορισμένη κενή συμβολοσειρά `""` (ψευδής τιμή που αριθμητικά είναι μηδέν και εξαιρείται από τις προειδοποιήσεις `uninitialized`) σε κάθε άλλη περίπτωση.

Ο τελεστής δεν εγείρει ποτέ εξαίρεση: τιμές αόριστες, μη αναφερόμενες και λανθασμένα ταυτοποιημένες απλώς δοκιμάζονται ως ψευδείς.

Πύλη χαρακτηριστικού

Η `isa` είναι διαθέσιμη από την Perl 5.31.6 και έπαψε να είναι πειραματική από την Perl 5.36. Είναι **λεξιλογικό χαρακτηριστικό**, που ενεργοποιείται με ένα από τα εξής:

```
use feature 'isa';
use v5.36;                  # enables the 5.36 feature bundle
```

Υπό no `feature 'isa'` (ή σε εμβέλεια που δεν την ενεργοποίησε ποτέ), το bareword `isa` επανέρχεται στο να είναι κανονικό αναγνωριστικό, και η έκφραση αποτελεί συντακτικό σφάλμα.

Η δεξιά πλευρά

Ο δεξιός τελεστέος της `isa` επιλύεται σε όνομα κλάσης κατά τη μεταγλώττιση όταν είναι bareword:

```
$obj isa Some::Class        # bareword - resolved at compile time
```

Κάθε άλλη έκφραση αποτιμάται κατά τον χρόνο εκτέλεσης και πρέπει να αποδώσει συμβολοσειρά που κατονομάζει την κλάση:

```
$obj isa "Different::Class" # string literal
$obj isa $class_name        # scalar variable
$obj isa (ref $other)       # any expression yielding a string
```

Τα barewords στη δεξιά πλευρά **δεν** χρειάζεται να δηλωθούν, να εισαχθούν ή να φορτωθούν· η `isa` συμβουλεύεται μόνο τον πίνακα συμβόλων της `CLASS` για να διασχίσει το `@ISA`. Μια κλάση που δεν φορτώθηκε ποτέ απλώς δεν έχει γονείς, οπότε το `$obj isa Unknown::Class` απλώς δοκιμάζεται ως ψευδές.

Παραδείγματα

Βασικός έλεγχος στιγμιότυπου:

```
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
```

Φραγή κλήσης μεθόδου επί της πραγματικής κλάσης:

```
sub describe {
    my ($thing) = @_;
    return "unknown" unless $thing isa Animal;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    return $thing->sound;
}
```

Ασφαλής σε αυθαίρετες τιμές - καμία εξαίρεση σε μη αναφορές:

```
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
```

Αποστολή σε όνομα κλάσης που συλλήφθηκε κατά τον χρόνο εκτέλεσης:

```
for my $class (qw(Dog Cat Fish)) {
    if ($rex isa $class) {
        say "rex is a $class";
    }
}
```

Γιατί η `isa` προτιμάται έναντι της `UNIVERSAL::isa` ως μέθοδος:

```
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
```

Η πρώτη μορφή κάνει σιωπηλά το λάθος πράγμα σε οποιαδήποτε συμβολοσειρά τυχαίνει να ταιριάζει με όνομα φορτωμένης κλάσης. Η μορφή τελεστή έχει επίγνωση τύπου και αναφέρει αυτό που πραγματικά ήθελε να μάθει ο καλών.

Οριακές περιπτώσεις

- **Ο αριστερός τελεστέος είναι undef**: ο τελεστής επιστρέφει ψευδές. Δεν εκπέμπεται προειδοποίηση `uninitialized` - αυτό είναι σκόπιμα σιωπηλός έλεγχος.
- **Ο αριστερός τελεστέος είναι μη-blessed αναφορά**: ψευδές. Η `isa` απαιτεί blessed αναφορά· αναφορές πινάκων, αναφορές κατακερματισμού και code refs που δεν έχουν περάσει ποτέ μέσω της `bless` αποτυγχάνουν τον έλεγχο.
- **Ο αριστερός τελεστέος είναι απλή συμβολοσειρά**: ψευδές, ακόμη και όταν η συμβολοσειρά ταιριάζει με όνομα φορτωμένης κλάσης. Αυτή είναι η συμπεριφορική απόκλιση από την `UNIVERSAL::isa` που καλείται ως μέθοδος.
- **Ο δεξιός τελεστέος κατονομάζει άγνωστη κλάση**: ο τελεστής δεν κάνει auto-load ούτε require του αρθρώματος. Διασχίζει το `@ISA` της κλάσης· μια άγνωστη κλάση δεν έχει `@ISA`, οπότε ο έλεγχος είναι ψευδής.
- **Πολλαπλή κληρονομικότητα**: η `isa` τιμά όποια σειρά επίλυσης μεθόδου είναι ενεργή στην κλάση (`mro`, `c3` ή η προεπιλεγμένη αναζήτηση κατά βάθος). Η αναζήτηση είναι η ίδια διάσχιση που χρησιμοποιεί η αποστολή μεθόδου.

- **Reblessed αντικείμενα**: ο έλεγχος αντικατοπτρίζει το τρέχον bless. Η εκ νέου κλήση της `bless` σε μια αναφορά αλλάζει αυτό που αναφέρει η `isa`.
- **Προτεραιότητα αναλυτή**: η `isa` είναι μη προσεταιριστικός ονομασμένος τελεστής με την ίδια προτεραιότητα με τους σχεσιακούς τελεστές (`lt`, `gt`, `le`, `ge`). Η αλυσίδωση (`$a isa B isa C`) είναι συντακτικό σφάλμα, όπως και με τους άλλους σχεσιακούς.
- **Χαρακτηριστικό μη ενεργοποιημένο**: χωρίς `use feature 'isa'` (ή `use v5.36`), ο αναλυτής βλέπει την `isa` ως συνηθισμένο bareword και η έκφραση αποτυγχάνει να μεταγλωττιστεί.
- **Ο δεξιός τελεστέος ως κλήση υπορουτίνας**: γράψτε τον με παρενθέσεις. Το `$obj isa some_class()` αναλύεται· το `$obj isa some_class` μεταχειρίζεται το `some_class` ως κυριολεκτικό όνομα κλάσης (bareword).

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `bless` - επισυνάψτε όνομα κλάσης σε μια αναφορά ώστε η `isa` (και η αποστολή μεθόδου) να έχει κάτι να αναφέρει
- `ref` - ανακτήστε το όνομα κλάσης μιας blessed αναφοράς, ή τον υποκείμενο τύπο για μη-blessed· συμπληρώνει την `isa` όταν χρειάζεστε το όνομα αντί για απάντηση ναι/όχι
- `class` - η εγγενής συντακτική σύνταξη δήλωσης κλάσης (χαρακτηριστικό `class`, Perl 5.38+)· τα αντικείμενα που κατασκευάζονται με την `class` είναι συνηθισμένες blessed αναφορές και λειτουργούν με την `isa` αμετάβλητα
- `UNIVERSAL::isa` - η ιστορική μορφή μεθόδου. Λειτουργεί σε κάθε αναφορά, αλλά επιστρέφει επίσης αληθές για απλές συμβολοσειρές που τυχαίνει να κατονομάζουν μια κλάση· σε νέο κώδικα χρησιμοποιήστε τον τελεστή
- `mro` - ελέγχει τη σειρά επίλυσης μεθόδου που η `isa` διασχίζει μέσω του `@ISA`
- `perlop` - πλήρης πίνακας προτεραιότητας τελεστών· η `isa` βρίσκεται με τους μη προσεταιριστικούς σχεσιακούς τελεστές

Λίστες

join

Συνενώνει μια λίστα συμβολοσειρών με έναν διαχωριστή μεταξύ κάθε γειτονικού ζεύγους και επιστρέφει τη μία προκύπτουσα συμβολοσειρά.

Η `join` παίρνει έναν διαχωριστή ως πρώτο όρισμα και οποιοδήποτε πλήθος τιμών ως τα υπόλοιπα. Μετατρέπει σε συμβολοσειρά κάθε τιμή στη `LIST`, τις κολλά μαζί με την `EXPR` τοποθετημένη μεταξύ κάθε ζεύγους, και επιστρέφει ένα βαθμωτό. Ο διαχωριστής εισάγεται **μεταξύ** στοιχείων μόνο - ποτέ στην αρχή, ποτέ στο τέλος. Με μηδέν ή ένα στοιχείο ο διαχωριστής δεν εμφανίζεται καθόλου.

Σύνοψη

```
join $sep, @list
join "", @chars
join "\n", @lines
```

Τι επιστρέφεται

Μία συμβολοσειρά. Το αποτέλεσμα είναι ένα φρέσκο βαθμωτό· η τροποποίησή του δεν φτάνει πίσω στη `LIST`. Σε βαθμωτό περιβάλλον παίρνετε αυτή τη συμβολοσειρά· η `join` είναι ουσιαστικά πάντα βαθμωτή επειδή παράγει πάντα μόνο μία τιμή.

Παραδείγματα

Τιμές διαχωρισμένες με κόμμα, η κανονική χρήση:

```
my @cols = ("alice", 42, "ops");
my $row  = join ",", @cols;          # "alice,42,ops"
```

Κατασκευή διαδρομής αρχείου με `/` ως διαχωριστή:

```
my @parts = ("var", "log", "app.log");
my $path  = "/" . join("/", @parts);  # "/var/log/app.log"
```

Ένωση γραμμών με newline μεταξύ κάθε μιας - προσέξτε ότι η τελευταία γραμμή **δεν** παίρνει newline, επειδή η `join` βάζει τον διαχωριστή *μεταξύ* στοιχείων:

```
my @lines = ("first", "second", "third");
my $text  = join "\n", @lines;       # "first\nsecond\nthird"
```

Συνένωση χωρίς διαχωριστή. Χρησιμοποιήστε την κενή συμβολοσειρά `""`, όχι `undef` - η `undef` προκαλεί προειδοποίηση uninitialized υπό `use warnings` και ούτως ή άλλως μετατρέπεται σε συμβολοσειρά ως `""`:

```
my @chars = ("h", "e", "l", "l", "o");
my $word  = join "", @chars;         # "hello"
```

Μια κενή `LIST` παράγει κενή συμβολοσειρά - είναι ασφαλές να βασιστείτε σε αυτό και δεν χρειάζεται φύλακας:

```
my @empty;
my $s = join ",", @empty;            # ""
```

Οριακές περιπτώσεις

- **Κενή `LIST`**: επιστρέφει `""`. Ο διαχωριστής δεν χρησιμοποιείται ποτέ.
- **`LIST` ενός στοιχείου**: επιστρέφει αυτό το στοιχείο μετατραμμένο σε συμβολοσειρά, χωρίς διαχωριστή πουθενά: `join ",", ("only")` είναι `"only"`.
- **Ο διαχωριστής μετατρέπεται σε συμβολοσειρά**: περάστε αριθμό, αναφορά, αντικείμενο με υπερφορτωμένη `""` - η `join` καλεί την ίδια μετατροπή σε

συμβολοσειρά που θα καλούσε κάθε άλλος τελεστής της Perl. Το `join 0, 1, 2, 3` είναι `"10203"`.

- **Διαχωριστής πολλαπλών χαρακτήρων**: η `EXPR` μπορεί να είναι οποιαδήποτε συμβολοσειρά, όχι μόνο ένας χαρακτήρας. Το `join ", ", @cols` δίνει `"a, b, c"`.
- **Στοιχεία `undef` στη `LIST`**: κάθε `undef` μετατρέπεται σε συμβολοσειρά ως `""` και εκπέμπει προειδοποίηση `uninitialized` υπό `use warnings`. Δύο γειτονικά `undef` εξακολουθούν να παίρνουν διαχωριστή μεταξύ τους: το `join ",", (undef, undef)` είναι `","`.
- **Η `join` δεν είναι η αντίστροφη της `split` σε έναν μόνο χαρακτήρα** όταν ο διαχωριστής που περνάτε διαφέρει από το μοτίβο που χρησιμοποιείται για διαχωρισμό. Η `split` παίρνει *μοτίβο*· η `join` παίρνει κυριολεκτική συμβολοσειρά. Ο πλήρης κύκλος `join "\t", split /\s+/, $s` χάνει τη διάκριση μεταξύ ακολουθιών λευκών χαρακτήρων και ενός μόνο tab.
- **Όχι μοτίβο**: σε αντίθεση με την `split`, το πρώτο όρισμα της `join` δεν είναι regex. Το `join /,/, @x` είναι συντακτική έκπληξη - ο διαχωριστής είναι πάντα συμβολοσειρά.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `split` - η αντίστροφη: παίρνει συμβολοσειρά και μοτίβο, επιστρέφει λίστα τμημάτων
- `sprintf` - όταν χρειάζεστε μορφοποίηση ανά στοιχείο, συμπλήρωση, ή πλάτη πεδίων που η `join` δεν μπορεί να εκφράσει
- `pack` - όταν η έξοδος είναι δυαδική εγγραφή, όχι συμβολοσειρά κειμένου με διαχωριστές
- `map` - μετασχηματισμός στοιχείων πριν την ένωση, π.χ. `join ",", map { uc } @names`

Πίνακες · Hashes

keys

Απαριθμεί όλα τα κλειδιά ενός κατακερματισμού, ή όλους τους δείκτες ενός πίνακα.

Η `keys` είναι η πρωτογενής λειτουργία ενδοσκόπησης για συσχετιστικά και ευρετηριασμένα containers. Δοθέντος `%HASH` επιστρέφει τα κλειδιά του κατακερματισμού· δοθέντος `@ARRAY` (από την Perl 5.12 και μετά) επιστρέφει τους δείκτες του πίνακα `0 .. $#ARRAY`. Η λίστα είναι αυτό που επαναλαμβάνετε, μετράτε, ταξινομείτε, ή τροφοδοτείτε σε άλλον τελεστή που καταναλώνει λίστα. Η `keys` είναι επίσης ο ιδιωματικός τρόπος να επαναφέρετε τον επαναλήπτη που διασχίζει η `each`.

Σύνοψη

```
my @k = keys %h;
my @i = keys @a;
my $n = keys %h;          # scalar context: count
keys(%h) = 1000;          # preallocate buckets (hash only)
```

Τι επιστρέφεται

Σε περιβάλλον λίστας, μια λίστα κάθε κλειδιού που είναι αυτή τη στιγμή στο %HASH, ή κάθε έγκυρου δείκτη του @ARRAY. Σε βαθμωτό περιβάλλον, την καταμέτρηση - χωρίς να κατασκευαστεί η λίστα - οπότε η `scalar keys %h` είναι ο φθηνός τρόπος για να ρωτήσετε «πόσες εγγραφές;»

Τα κλειδιά κατακερματισμού επιστρέφονται σε **αυθαίρετη** σειρά. Η σειρά είναι σταθερή για δεδομένο μη τροποποιημένο κατακερματισμό - οι keys, values και each τον διασχίζουν όλες στην ίδια ακολουθία - αλλά δεν είναι η σειρά εισαγωγής, δεν είναι ταξινομημένη, και σκόπιμα τυχαιοποιείται ανά κατακερματισμό για να αποτραπούν επιθέσεις αλγοριθμικής πολυπλοκότητας (βλέπε perlsec). Οποιαδήποτε εισαγωγή ή διαγραφή μπορεί να αναδιατάξει. Η μία τεκμηριωμένη εξαίρεση: η διαγραφή του κλειδιού που πιο πρόσφατα επέστρεψε η each ή η keys δεν διαταράσσει τη σειρά των υπολοίπων.

Οι δείκτες πίνακα, αντιθέτως, επιστρέφονται με το χαμηλότερο πρώτο: `(0, 1, 2, …, $#ARRAY)`.

Τα επιστρεφόμενα κλειδιά κατακερματισμού είναι **αντίγραφα**. Η τροποποίηση ενός στοιχείου της επιστρεφόμενης λίστας δεν τροποποιεί τον κατακερματισμό. Συγκρίνετε με την values, που επιστρέφει ψευδώνυμα.

Καθολική κατάσταση που επηρεάζει

- **Επαναλήπτης του %HASH / @ARRAY**: η keys επαναφέρει τον εσωτερικό επαναλήπτη που μοιράζεται με τις each και values. Αυτή είναι η ιδιωματική επαναφορά - καλέστε `keys %h` σε κενό περιβάλλον ειδικά για να απορρίψετε οποιαδήποτε μερική διάσχιση της each:

  ```
  keys %h;                  # reset iterator, no list built
  while (my ($k, $v) = each %h) { ... }
  ```

  Χωρίς αυτή την επαναφορά, μια προηγούμενη each που εξήλθε πρόωρα αφήνει τον επαναλήπτη στη μέση του κατακερματισμού και η επόμενη κλήση each συνεχίζει από εκεί.

Παραδείγματα

Επαναλάβετε όλα τα κλειδιά σε αυθαίρετη σειρά:

```
my %h = (apple => 1, pear => 2, plum => 3);
for my $k (keys %h) {
    print "$k => $h{$k}\n";
}
```

Επαναλάβετε σε ντετερμινιστική σειρά περνώντας τα κλειδιά μέσα από τη `sort`:

```
for my $k (sort keys %h) {
    print "$k => $h{$k}\n";     # apple, pear, plum
}
```

Επαναλάβετε σε **σειρά εισαγωγής** παρακολουθώντας τα κλειδιά ξεχωριστά - οι ίδιοι οι κατακερματισμοί δεν θυμούνται τη σειρά εισαγωγής:

```
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)
}
```

Μετρήστε εγγραφές χωρίς να υλοποιήσετε τη λίστα:

```
my $n = keys %h;                # scalar context, O(1)
print "hash has $n entries\n";
```

Προεκχωρήστε buckets όταν γνωρίζετε ότι ο κατακερματισμός θα μεγαλώσει αρκετά. Η εκχώρηση είναι υπόδειξη μεγέθους, στρογγυλοποιημένη προς τα πάνω στην επόμενη δύναμη του δύο:

```
my %big;
keys(%big) = 10_000;            # allocates 16384 buckets
$big{$_} = 1 for 1 .. 10_000;   # no rehash churn during fill
```

Διασχίστε έναν πίνακα κατά δείκτη - χρήσιμο όταν χρειάζεστε τον δείκτη και το στοιχείο μαζί:

```
my @a = ("zero", "one", "two");
for my $i (keys @a) {
    print "$i: $a[$i]\n";
}
```

Ταξινομήστε έναν κατακερματισμό κατά τιμή, αριθμητικά φθίνουσα:

```
for my $k (sort { $h{$b} <=> $h{$a} } keys %h) {
    printf "%4d %s\n", $h{$k}, $k;
}
```

Οριακές περιπτώσεις

- **Κενός κατακερματισμός ή πίνακας**: επιστρέφει την κενή λίστα σε περιβάλλον λίστας, `0` σε βαθμωτό περιβάλλον. Καμία προειδοποίηση, καμία ειδική περίπτωση για παράκαμψη.

- **`require v5.12` για πίνακες**: η `keys @array` είναι συντακτικό σφάλμα σε perl παλαιότερες της 5.12. Βάλτε `use v5.12;` στην κορυφή αρχείων που χρησιμοποιούν τη μορφή για πίνακες αν έχει σημασία η φορητότητα σε αρχαιότερες perl.
- **Η σειρά κατακερματισμού δεν είναι αναπαραγώγιμη μεταξύ εκτελέσεων**: η Perl τυχαιοποιεί τον σπόρο κατακερματισμού στην εκκίνηση. Μη χτίζετε ποτέ δοκιμή, κλειδί cache ή μορφή αρχείου που υποθέτει συγκεκριμένη σειρά `keys %h`. Ταξινομήστε αν χρειάζεστε ντετερμινισμό.
- **Η σειρά κατακερματισμού δεν είναι σταθερή μεταξύ εκδόσεων Perl**: η τυχαιοποιημένη σειρά εντός μιας εκτέλεσης είναι σταθερή· ο ίδιος ο αλγόριθμος δεν εγγυάται μεταξύ εκδόσεων. Μη σειριοποιείτε την έξοδο της `keys` περιμένοντας να κάνει πλήρη κύκλο σε διαφορετική Perl.
- **Διαγραφή κατά τη διάρκεια επανάληψης**: ασφαλής μόνο για το κλειδί που πιο πρόσφατα επέστρεψε η `each` ή η `keys`. Η διαγραφή διαφορετικού κλειδιού μπορεί να προκαλέσει διπλή επίσκεψη ή παράλειψη άλλων κλειδιών.
- **Η lvalue `keys` σε πίνακες είναι συντακτικό σφάλμα**: το `keys(@a) = 100` δεν είναι έγκυρο. Η μορφή προ-εκχώρησης buckets λειτουργεί μόνο σε κατακερματισμούς.
- **Η lvalue `keys` δεν μπορεί να συρρικνώσει**: το `keys(%h) = 10` σε κατακερματισμό που έχει ήδη 1024 buckets δεν έχει αποτέλεσμα. Το `%h = ()` καθαρίζει τα περιεχόμενα αλλά διατηρεί τον αριθμό buckets· η `undef %h` ελευθερώνει τα buckets.
- **Κόστος μνήμης της μορφής λίστας**: η `keys %h` σε περιβάλλον λίστας υλοποιεί κάθε κλειδί ως φρέσκο SV. Για κατακερματισμό ενός εκατομμυρίου εγγραφών αυτό αντιστοιχεί σε ένα εκατομμύριο εκχωρήσεις συν την επιστρεφόμενη λίστα. Επαναλάβετε με `each` ή `while (my ($k, $v) = each %h)` όταν θέλετε ροή πρόσβασης χωρίς την αιχμή μνήμης.
- **Tied κατακερματισμοί**: αποστέλλουν μέσω `FIRSTKEY` / `NEXTKEY`. Η σειρά και οι παρενέργειες είναι ό,τι παρέχει η κλάση tie. Συγκεκριμένα, οι tied κατακερματισμοί μπορούν να αναδιαταχθούν κατά την εισαγωγή με τρόπους που δεν κάνουν οι συνηθισμένοι κατακερματισμοί, και μια tied `keys` μπορεί να είναι ακριβή αν το υποκείμενο αποθηκευτικό μέσο είναι απομακρυσμένο.
- **Βαθμωτό περιβάλλον σε tied κατακερματισμό**: μπορεί να είναι ή να μην είναι O(1). Οι συνηθισμένοι κατακερματισμοί απαντούν στο `scalar keys %h` σε σταθερό χρόνο· μια κλάση tied πρέπει να υλοποιήσει ρητά το `SCALAR` για να αποφύγει πλήρη διάσχιση.
- **Η βαθμωτή `keys` σε πίνακα ισούται με `scalar @array`**: και τα δύο επιστρέφουν τον αριθμό στοιχείων. Προτιμήστε το `scalar @array` για σαφήνεια, εκτός αν ο γύρω κώδικας είναι ήδη διατυπωμένος με όρους `keys`/`values`/`each`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `values` - η συνοδευτική λίστα των τιμών στην ίδια σειρά με την `keys`· επιστρέφει ψευδώνυμα, όχι αντίγραφα
- `each` - διάσχιση ζεύγους `(κλειδί, τιμή)` με ροή που μοιράζεται τον ίδιο επαναλήπτη που επαναφέρει η `keys`
- `exists` - δοκιμάστε αν ένα συγκεκριμένο κλειδί είναι παρόν χωρίς να κατασκευάσετε την πλήρη λίστα κλειδιών
- `delete` - αφαιρέστε ένα κλειδί· ασφαλής στη μέση της επανάληψης μόνο για το πιο πρόσφατα επιστρεφόμενο κλειδί
- `sort` - επιβάλετε ντετερμινιστική σειρά στην αυθαίρετη ακολουθία που επιστρέφει η `keys`
- `scalar` - εξαναγκάστε βαθμωτό περιβάλλον για να λάβετε την καταμέτρηση χωρίς να υλοποιήσετε τη λίστα κλειδιών

Διεργασίες

kill

Αποστολή σήματος σε λίστα διεργασιών.

Η `kill` είναι η εμπρόσθια διεπαφή της Perl για την κλήση συστήματος `kill(2)`. Παραδίδει το `SIGNAL` σε κάθε διεργασία που προσδιορίζεται από τη `LIST` και επιστρέφει τον αριθμό των ορισμάτων στα οποία στάλθηκε επιτυχώς σήμα. Είναι επίσης ο τυπικός τρόπος να ρωτήσετε τον πυρήνα «υπάρχει ακόμα αυτό το PID και μπορώ να του στείλω σήμα;» χωρίς να ενοχλήσετε πραγματικά τον στόχο, μέσω της μορφής signal-zero.

Σύνοψη

```
kill SIGNAL, LIST
kill SIGNAL            # tainting side-effect only; returns 0
```

Τι επιστρέφεται

Ο αριθμός των ορισμάτων στη `LIST` που πέρασαν επιτυχώς στην `kill(2)`. Αυτός **δεν** είναι απαραίτητα ο αριθμός των διεργασιών που πραγματικά έλαβαν το σήμα - όταν το `SIGNAL` είναι αρνητικό, ή όταν ένα PID προσδιορίζει ομάδα διεργασιών, ένα όρισμα μπορεί να στοχεύει πολλές διεργασίες. Σε σφάλματα δικαιωμάτων ή «δεν υπάρχει τέτοια διεργασία» για ένα όρισμα, αυτό το όρισμα δεν μετράται και η `$!` τίθεται στο αντίστοιχο `errno` (`EPERM`, `ESRCH`, `EINVAL`).

```
my $cnt = kill 'HUP', $child1, $child2;
kill 'KILL', @goners;
```

Με κενή `LIST`, δεν στέλνεται κανένα σήμα και η τιμή επιστροφής είναι `0`. Η σκέτη μορφή `kill SIGNAL` γράφεται μερικές φορές αμιγώς για να πυροδοτήσει ελέγχους taint στο `SIGNAL`· δείτε το `perlsec`.

SIGNAL: όνομα ή αριθμός

Το SIGNAL μπορεί να είναι είτε όνομα σήματος (συμβολοσειρά) είτε αριθμός σήματος. Η μορφή συμβολοσειράς συνιστάται για φορητότητα επειδή το ίδιο σήμα μπορεί να έχει διαφορετικούς αριθμούς σε διαφορετικά λειτουργικά συστήματα.

Ένα όνομα σήματος μπορεί να φέρει το πρόθεμα SIG ή όχι - τα 'TERM' και 'SIGTERM' ονοματίζουν το ίδιο σήμα.

```
kill 'TERM', $pid;      # portable
kill 'SIGTERM', $pid;   # also fine
kill 15, $pid;          # SIGTERM on Linux; may differ elsewhere
kill 9, $pid;           # SIGKILL on Linux
```

Η λίστα ονομάτων σημάτων διαθέσιμων στην τρέχουσα πλατφόρμα βρίσκεται στο $Config{sig_name}, που παρέχεται από το άρθρωμα Config.

SIGNAL 0: έλεγχος ζωντάνιας

Αν το SIGNAL είναι ο αριθμός 0 ή η συμβολοσειρά 'ZERO' (ή 'SIGZERO'), κανένα σήμα δεν παραδίδεται πραγματικά. Η kill(2) εξακολουθεί να εκτελεί τον έλεγχο δικαιωμάτων της, οπότε η κλήση σας λέει αν θα ήταν *δυνατό* να σταλεί σήμα στη διεργασία - δηλαδή, η διεργασία υπάρχει και ανήκει στον ίδιο χρήστη, ή ο καλών είναι ο super-user.

```
if (kill 0, $pid) {
    # process exists and we may signal it
}
elsif ($!{ESRCH}) {
    # no such process - it has exited and been reaped
}
elsif ($!{EPERM}) {
    # it exists but we are not allowed to signal it
}
```

Αυτό είναι το κανονικό ιδίωμα για τον έλεγχο ότι ένα παιδί είναι ακόμα ζωντανό, ακόμη και ως zombie, και δεν έχει αλλάξει το UID του. **Δεν** συλλέγει το παιδί - χρησιμοποιήστε την waitpid γι” αυτό.

Αρνητικό SIGNAL: σήμα σε ομάδα διεργασιών

Ένα αρνητικό όνομα ή αριθμός σήματος στέλνει σήμα σε ολόκληρη ομάδα διεργασιών αντί σε μεμονωμένη διεργασία. Οι kill '-KILL', $pgrp και kill -9, $pgrp στέλνουν αμφότερες SIGKILL σε κάθε διεργασία της ομάδας που προσδιορίζεται από το $pgrp.

```
kill '-TERM', $pgrp;    # polite shutdown of the whole group
kill -9, $pgrp;         # forced
```

Τα θετικά σήματα είναι σχεδόν πάντα αυτό που θέλετε. Καταφύγετε σε αρνητικό σήμα μόνο όταν ο στόχος είναι σκόπιμα αρχηγός ομάδας διεργασιών (για παράδειγμα ένα παιδί που τοποθετήσατε στη δική του ομάδα με setpgrp ώστε να τερματίσετε το υποδέντρο μονομιάς).

PID 0, αρνητικά PIDs

Η συμπεριφορά της `kill` όταν ένα *PID* είναι μηδέν ή αρνητικό εξαρτάται από το λειτουργικό σύστημα. Σε συστήματα συμμορφούμενα με POSIX (συμπεριλαμβανομένου του Linux):

- **PID `0`** - αποστολή σήματος στην ίδια την ομάδα διεργασιών του αποστολέα.
- **PID `-1`** - αποστολή σήματος σε κάθε διεργασία στην οποία ο καλών έχει δικαίωμα να στείλει σήμα (super-user: όλο το σύστημα, μείον μερικά ειδικά PIDs).
- **Οποιοδήποτε άλλο αρνητικό PID** - συμπεριφέρεται όπως αρνητικό `SIGNAL` στην απόλυτη τιμή: αποστολή σήματος στην ομάδα διεργασιών της οποίας το PGID ισούται με `abs(PID)`.

Αν τόσο το `SIGNAL` όσο και το PID είναι αρνητικά, το αποτέλεσμα δεν είναι ορισμένο· μια μελλοντική Perl μπορεί να προειδοποιήσει. Μη συνδυάζετε τα δύο.

Παραδείγματα

Ευγενική αίτηση σε ένα παιδί να τερματίσει, και έπειτα κλιμάκωση:

```
kill 'TERM', $child;
sleep 2;
kill 'KILL', $child if kill 0, $child;
waitpid $child, 0;
```

Καμία συλλογή, απλώς έλεγχος αν ένα αρχείο PID ονοματίζει ακόμα ζωντανή διεργασία:

```
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";
}
```

Αποστολή σήματος σε ολόκληρη ομάδα διεργασιών που δημιουργήθηκε από ένα παιδί σωλήνωσης κελύφους:

```
my $pgrp = getpgrp($child);
kill '-TERM', $pgrp;
```

Εγκατάσταση χειριστή και αποστολή σήματος στον εαυτό σας:

```
local $SIG{USR1} = sub { warn "got USR1\n" };
kill 'USR1', $$;        # $$ is our own PID
```

Εκπομπή σε πολλούς εργάτες και ανακάλυψη πόσοι ήταν προσβάσιμοι:

```
my @workers = (1234, 1235, 9999);   # 9999 has exited
my $hit = kill 'HUP', @workers;     # probably 2
warn "lost a worker: $!\n" if $hit < @workers;
```

Οριακές περιπτώσεις

- **Κενή LIST**: η `kill 'TERM'` δεν στέλνει τίποτα και επιστρέφει `0`. Η κλήση εξακολουθεί να αποτιμά το SIGNAL, που είναι ο μόνος λόγος να τη γράψετε (διάδοση taint).
- **Άγνωστο όνομα σήματος**: εγείρει εξαίρεση (`Unrecognized signal name "FOO"`). Επαληθεύστε έναντι του `$Config{sig_name}` αν το όνομα προέρχεται από είσοδο χρήστη.
- **Η `$!` μετά από μερική επιτυχία**: η `kill` διασχίζει τη LIST από αριστερά προς τα δεξιά. Η `$!` αντικατοπτρίζει το **τελευταίο** όρισμα που απέτυχε, όχι όλα. Αν χρειάζεστε σφάλματα ανά PID, καλέστε την `kill` μία φορά ανά PID.
- **Τα zombies εξακολουθούν να μετρούν ως ζωντανά**: η `kill 0, $zombie_pid` επιστρέφει αληθές μέχρι το παιδί να συλλεχθεί με `wait` ή `waitpid`. Αυτό συνήθως είναι αυτό που θέλετε για έλεγχο ζωντάνιας· περιστασιακά εκπλήσσει.
- **Η παράδοση σήματος είναι ασύγχρονη**: μια επιτυχημένη `kill` σημαίνει μόνο ότι το σήμα *μπήκε σε ουρά*. Ο χειριστής στο `%SIG` στη λαμβάνουσα διεργασία μπορεί να τρέξει αργότερα, και η μηχανική ασφαλών σημάτων της Perl αναβάλλει την παράδοση στο επόμενο όριο opcode.
- **Αποστολή στον εαυτό σας**: η `kill SIGNAL, $$` λειτουργεί και είναι ο συνηθισμένος τρόπος για να δοκιμάσετε τους δικούς σας χειριστές `%SIG`.
- **Καμία LIST, με taint**: υπό `-T`, η γραφή `kill 'TERM'` διαδίδει την tainted κατάσταση του `'TERM'` για την παρενέργεια ενεργοποίησης του ελέγχου - η μορφή δεν έχει άλλη χρήση.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `fork` - δημιουργία της θυγατρικής διεργασίας στην οποία θα στείλετε αργότερα σήμα
- `wait` - συλλογή οποιουδήποτε τερματισμένου παιδιού, εκκαθαρίζοντας zombies που η `kill 0` θα ανέφερε ακόμα ως ζωντανά
- `waitpid` - συλλογή συγκεκριμένου παιδιού, προαιρετικά χωρίς μπλοκάρισμα (`WNOHANG`) ώστε να συνδυάζεται καθαρά με την `kill 0`
- `getpgrp` / `setpgrp` - επιθεώρηση και αλλαγή συμμετοχής σε ομάδα διεργασιών, που είναι αυτό στο οποίο λειτουργεί η `kill` με αρνητικό SIGNAL
- `%SIG` - εγκατάσταση χειριστών για τα σήματα που στέλνετε με την `kill`

- `perlipc` - πλήρης πραγμάτευση σημάτων, χειριστών και σημασιολογίας ασφαλών σημάτων

Ροή ελέγχου

last

Εγκαταλείπει αμέσως έναν βρόχο, παραλείποντας το υπόλοιπο του σώματος και το μπλοκ `continue`.

Η `last` είναι το αντίστοιχο του `break` της C στην Perl. Μεταφέρει τον έλεγχο εκτός του περικλείοντος βρόχου στην εντολή που ακολουθεί. Οι υπόλοιπες εντολές της τρέχουσας επανάληψης δεν εκτελούνται, ούτε το μπλοκ `continue` του βρόχου - αυτή είναι η βασική διαφορά από τη `next`. Χωρίς όρισμα στοχεύει τον πιο εσωτερικό περικλείοντα `while`, `until`, `for`, `foreach` ή γυμνό μπλοκ· με `LABEL` στοχεύει έναν επώνυμο βρόχο πιο έξω.

Σύνοψη

```
last
last LABEL
last EXPR       # LABEL computed at runtime, since 5.18
```

Τι επιστρέφεται

Τίποτα. Η `last` εκτελεί έλεγχο ροής· δεν παράγει τιμή. Δεν μπορεί να χρησιμοποιηθεί για να επιστρέψει αποτέλεσμα από μπλοκ που κανονικά αποδίδει ένα, όπως `eval { ... }`, `sub { ... }`, ή `do { ... }`. Σε αυτά τα περιβάλλοντα είτε η κατασκευή δεν παράγει τιμή είτε το πρόγραμμα πεθαίνει με `Can't "last" outside a loop block` - δείτε *Οριακές περιπτώσεις* παρακάτω.

Στόχευση βρόχου με ετικέτα

Χωρίς ετικέτα, η `last` αναφέρεται στον πιο εσωτερικό περικλείοντα βρόχο. Αυτό είναι συνήθως αυτό που θέλετε:

```
while (my $line = <$fh>) {
    last if $line =~ /^__END__$/;
    process($line);
}
```

Με μια ετικέτα μπορείτε να εγκαταλείψετε έναν εξωτερικό βρόχο μέσα από έναν εμφωλευμένο. Οι ετικέτες είναι συμβατικά εξ ολοκλήρου κεφαλαία:

```
OUTER: for my $row (@grid) {
    for my $cell (@$row) {
        last OUTER if $cell eq 'STOP';
    }
}
# control resumes here
```

Η `last EXPR` (Perl 5.18+) υπολογίζει την ετικέτα κατά τον χρόνο εκτέλεσης. Η `EXPR` πρέπει να αποτιμηθεί σε συμβολοσειρά που κατονομάζει μια περικλείουσα ετικέτα:

```
my $target = 'OUTER';
last $target;
```

Αλληλεπίδραση με μπλοκ continue

Το μπλοκ `continue` ενός βρόχου εκτελείται κανονικά στο τέλος κάθε επανάληψης, περιλαμβανομένης της τελευταίας. Η `last` **το παραλείπει** - αυτή είναι η οριστική διαφορά της από τη `next`:

```
my $i = 0;
while ($i < 10) {
    last if $i == 3;
    print "body $i\n";
} continue {
    $i++;                # runs after every iteration EXCEPT
                         # 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.
```

Αν θέλετε να εκτελεστεί η `continue`, χρησιμοποιήστε αντί γι" αυτό τη `next` με φύλακα, ή προωθήστε τον μετρητή πριν την `last`.

Συμπεριφορά σε εμφωλευμένους βρόχους

Η `last` εγκαταλείπει μόνο έναν βρόχο τη φορά - τον πιο εσωτερικό, εκτός αν φέρει ετικέτα. Σε αυτό το παράδειγμα η εσωτερική `last` εγκαταλείπει τη `for` αλλά η `while` συνεχίζει να εκτελείται:

```
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
}
```

Για να ξετυλίξετε πολλά επίπεδα ταυτόχρονα, βάλτε ετικέτα στον εξωτερικό βρόχο και στοχεύστε τον:

```
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
        }
    }
}
```

Τα γυμνά μπλοκ μετράνε ως βρόχοι

Ένα γυμνό μπλοκ `{ ... }` είναι, για σκοπούς ελέγχου ροής, ένας βρόχος που εκτελείται ακριβώς μία φορά. Η `last` μπορεί να εγκαταλείψει τέτοιο μπλοκ πρόωρα, που είναι ο ιδιωματικός τρόπος για να γράψετε «δοκίμασε μια ακολουθία ελέγχων, βγες στο πρώτο ταίριασμα»:

```
{
    last if $cached;
    recompute();
    store_in_cache();
}
# control continues here
```

Αυτή είναι η μόνη μορφή της `last` που δεν περιλαμβάνει επαναληπτική κατασκευή.

Παραδείγματα

Σταματήστε την ανάγνωση αρχείου σε κενή γραμμή - το κλασικό ιδίωμα τερματιστή κεφαλίδας:

```
LINE: while (<$fh>) {
    last LINE if /^$/;
    push @header, $_;
}
```

Εύρεση-και-αναφορά σε δισδιάστατη δομή:

```
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;
        }
    }
}
```

Πρόωρη έξοδος από γυμνό μπλοκ που χρησιμοποιείται ως ακολουθία try-once:

```
my $result;
{
    $result = $cache{$key}, last if exists $cache{$key};
    $result = compute($key);
    $cache{$key} = $result;
}
```

Υπολογισμένη ετικέτα (Perl 5.18+). Σπάνια απαιτείται, αλλά χρήσιμο όταν ο βρόχος-στόχος επιλέγεται δυναμικά:

```
my $level = $deep ? 'OUTER' : 'INNER';
OUTER: for (...) {
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    INNER: for (...) {
        last $level if $done;
    }
}
```

Οριακές περιπτώσεις

- **Παράνομη σε `grep` και `map`.** Και οι δύο περιμένουν το μπλοκ τους να παράγει τιμή για κάθε στοιχείο εισόδου· η `last` το αποκλείει. Ο αναλυτής δέχεται τον κώδικα (το μπλοκ δεν είναι λεξιλογικά βρόχος μέσα από τον οποίο μπορεί να δει), αλλά η εκτέλεσή του πεθαίνει με `Can't "last" outside a loop block`. Φιλτράρετε με υπό συνθήκη μέσα στο μπλοκ αντί γι" αυτό, ή μεταβείτε σε ρητό βρόχο `for`.
- **Το `do { ... }` δεν είναι βρόχος.** Η `last` μέσα σε `do { ... }` δεν τερματίζει τη `do`· ξετυλίγει στον κοντινότερο περικλείοντα πραγματικό βρόχο. Αν δεν υπάρχει, το πρόγραμμα πεθαίνει με `Can't "last" outside a loop block`. Για πρόωρη έξοδο από μια `do` που χρησιμοποιείται ως έκφραση, δείτε τα workarounds στο *perlsyn* υπό «Statement Modifiers» - τυπικά γυμνό μπλοκ συν `last`, ή βρόχος `for` μίας εκτέλεσης.
- **Η `sub { ... }` δεν είναι βρόχος.** Η `last` μέσα σε σώμα sub **δεν** επιστρέφει από τη sub· ξετυλίγει πέρα από τη sub στον περικλείοντα βρόχο στον καλούντα. Χρησιμοποιήστε `return` για να φύγετε από μια sub.
- **Η `last` μέσα σε `eval` δεν διαφεύγει της `eval`.** Η `eval { ... }` δεν είναι η ίδια βρόχος, οπότε η `last` κοιτάζει προς τα έξω πέρα από την `eval` για τον κοντινότερο περικλείοντα βρόχο και ξετυλίγει σε αυτόν. Η `eval` εγκαταλείπεται ως παρενέργεια αυτού του ξετυλίγματος, αλλά το `$@` μένει κενό - δεν εγέρθηκε εξαίρεση. Αν η `eval` δεν έχει περικλείοντα βρόχο από πάνω, το πρόγραμμα πεθαίνει με `Can't "last" outside a loop block`. Για να φύγετε κανονικά από μια `eval`, αφήστε την ροή να τελειώσει ή χρησιμοποιήστε `die` (συλλαμβάνεται από την ίδια την `eval`) ή `return` (αν η `eval` βρίσκεται μέσα σε sub και θέλετε να φύγετε από τη sub).
- **Προτεραιότητα.** Σε αντίθεση με τους περισσότερους ονομασμένους τελεστές, η `last` έχει την ίδια προτεραιότητα με την εκχώρηση και εξαιρείται από τον κανόνα «μοιάζει με συνάρτηση». Έτσι το `last ("foo") . "bar"` περνά το `"foo" . "bar"` ως ετικέτα, όχι μόνο το `"foo"`. Χρησιμοποιήστε κενό, ή παραλείψτε τις παρενθέσεις, για να αποφύγετε εκπλήξεις:

  ```
  last LABEL;
  last "LABEL";
  ```

- **Άγνωστη ετικέτα.** Η `last NOPE` όπου κανένας περικλείων βρόχος δεν φέρει την ετικέτα `NOPE` πεθαίνει με `Label not found for "last NOPE"`. Αυτό είναι σφάλμα χρόνου εκτέλεσης, όχι μεταγλώττισης - η ετικέτα επιλύεται έναντι της δυναμικής στοίβας κλήσεων.
- **Η μορφή τροποποιητή βρόχου δεν έχει μπλοκ προς έξοδο.** Το `EXPR if COND` και παρόμοιες μορφές τροποποιητή εντολής δεν είναι βρόχοι· η `last` μέσα στην έκφραση ξετυλίγει πέρα από αυτές.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `next` - μεταβείτε στην επόμενη επανάληψη, εκτελώντας πρώτα το μπλοκ `continue`· το αντίστοιχο της `last` που κρατά τον βρόχο ζωντανό
- `redo` - επανεκκινήστε την τρέχουσα επανάληψη χωρίς να επανελέγξετε τη συνθήκη και χωρίς να εκτελέσετε την `continue`
- `return` - φύγετε από υπορουτίνα με τιμή· το σωστό εργαλείο όταν αναζητάτε την `last` μέσα σε `sub { ... }`
- `continue` - το μπλοκ-ουραίο του βρόχου που η `last` παραλείπει και η `next` τιμά· διαβάστε τη σελίδα του για την τριμερή απεικόνιση των `last` / `next` / `redo`

Βαθμωτά και συμβολοσειρές

lc

Επιστρέφει αντίγραφο μιας συμβολοσειράς μετατραμμένο σε πεζά.

Η `lc` παίρνει μια συμβολοσειρά, μετατρέπει κάθε χαρακτήρα με διάκριση πεζών-κεφαλαίων στο πεζό ισοδύναμό του και επιστρέφει το αποτέλεσμα. Το πρωτότυπο δεν τροποποιείται ποτέ. Αν παραλειφθεί το όρισμα, η `lc` λειτουργεί στην `$_`. Το ακριβές σύνολο χαρακτήρων που αλλάζει εξαρτάται από την κωδικοποίηση της συμβολοσειράς και τις pragmas που είναι σε εμβέλεια στο σημείο κλήσης - μόνο ASCII εξ ορισμού, πλήρες Unicode υπό `use feature 'unicode_strings'` ή όταν η είσοδος είναι ήδη συμβολοσειρά χαρακτήρων με τη σημαία UTF-8.

Σύνοψη

```
lc EXPR
lc
```

Τι επιστρέφεται

Μια νέα συμβολοσειρά. Η τιμή επιστροφής είναι πάντα ένα καινούργιο βαθμωτό - η τροποποίησή της δεν επηρεάζει το όρισμα, και η εκ των υστέρων τροποποίηση του ορίσματος δεν επηρεάζει την επιστρεφόμενη συμβολοσειρά. Το μήκος σε χαρακτήρες διατηρείται: η `lc` ποτέ δεν προσθέτει ή αφαιρεί χαρακτήρες, ακόμη και στις σπάνιες περιπτώσεις Unicode όπου οι μορφές πεζών και κεφαλαίων έχουν διαφορετικά μήκη σε άλλες κατευθύνσεις (βλ. `fc` για casefold, που μπορεί να αλλάξει το μήκος).

```
my $str = lc("Perl is GREAT");   # "perl is great"
```

Καθολική κατάσταση που επηρεάζει

- Διαβάζει την `$_` όταν καλείται χωρίς όρισμα.
- Παρατηρεί την `use locale` για `LC_CTYPE` - όταν είναι ενεργό το locale, ο πίνακας μετατροπής σε πεζά του τρέχοντος locale εφαρμόζεται σε σημεία κώδικα κάτω του 256.
- Παρατηρεί την `use bytes` - υπό `use bytes` μόνο τα `A-Z` αλλάζουν, σε `a-z`.
- Παρατηρεί την `use feature 'unicode_strings'` - επιβάλλει κανόνες Unicode ανεξάρτητα από τη σημαία UTF-8 της εισόδου.

Ποιοι κανόνες πεζών-κεφαλαίων εφαρμόζονται

Η Perl επιλέγει ένα από τέσσερα σύνολα κανόνων, με σειρά προτεραιότητας. Νικάει το πρώτο του οποίου η συνθήκη ισχύει:

1. **`use bytes` ενεργή** - κανόνες ASCII. Μόνο τα `A-Z` αντιστοιχίζονται σε `a-z`· κάθε άλλο byte παραμένει αμετάβλητο, συμπεριλαμβανομένων bytes στο εύρος 128-255 που διαφορετικά θα ήταν γράμματα Latin-1.
2. **`use locale` για `LC_CTYPE` ενεργή** - οι πίνακες του τρέχοντος locale εφαρμόζονται σε σημεία κώδικα κάτω του 256· τα σημεία κώδικα 256 και άνω (προσβάσιμα μόνο όταν η συμβολοσειρά φέρει ήδη τη σημαία UTF-8) χρησιμοποιούν κανόνες Unicode. Από την Perl 5.20 και μετά, ένα UTF-8 locale χρησιμοποιεί πλήρεις κανόνες Unicode παντού.
3. **Το όρισμα έχει σημαία UTF-8 ορισμένη** - κανόνες Unicode εφαρμόζονται σε κάθε χαρακτήρα.
4. **`use feature 'unicode_strings'` ή `use locale ':not_characters'` ενεργή** - κανόνες Unicode εφαρμόζονται σε κάθε χαρακτήρα, ανεξάρτητα από τη σημαία UTF-8.
5. **Διαφορετικά** - κανόνες ASCII. Χαρακτήρες εκτός των `A-Z` επιστρέφονται αμετάβλητοι, συμπεριλαμβανομένων κεφαλαίων γραμμάτων Latin-1 όπως τα À-Þ.

Το συμπέρασμα: αν θέλετε προβλέψιμη συμπεριφορά Unicode σε κάθε συμβολοσειρά ανεξάρτητα από τον τρόπο κατασκευής της, ενεργοποιήστε την `use feature 'unicode_strings'` (ή την `use v5.12` και άνω, που την ενεργοποιεί για εσάς).

Παραδείγματα

Βασική μετατροπή ASCII σε πεζά:

```
my $s = lc("Hello, World!");        # "hello, world!"
```

Προεπιλογή στην `$_`:

```
for ("FOO", "Bar", "BAZ") {
    print lc, "\n";                 # foo / bar / baz
}
```

Οι χαρακτήρες Unicode μετατρέπονται σε πεζά μόνο όταν οι κανόνες το επιτρέπουν:

```
use feature 'unicode_strings';
my $s = lc("ÄÖÜ");                    # "äöü"
```

Χωρίς unicode_strings και χωρίς τη σημαία UTF-8 στην είσοδο, οι μη-ASCII χαρακτήρες περνούν αμετάβλητοι:

```
my $bytes = "\xC4\xD6\xDC";           # Ä Ö Ü as Latin-1 bytes
my $lower = lc $bytes;                # unchanged: "\xC4\xD6\xDC"
```

Η lc είναι αυτή που υποστηρίζει την ακολουθία διαφυγής \L...\E μέσα σε συμβολοσειρές με διπλά εισαγωγικά:

```
my $str = "Perl is \LGREAT\E";        # "Perl is great"
```

Σύγκριση χωρίς διάκριση πεζών-κεφαλαίων με μετατροπή και των δύο πλευρών σε πεζά:

```
sub ieq { lc($_[0]) eq lc($_[1]) }
ieq("Perl", "PERL");                  # true
```

Για σωστή σύγκριση χωρίς διάκριση πεζών-κεφαλαίων σε όλο το Unicode, προτιμήστε την fc (τη συνάρτηση casefold του Unicode) έναντι της lc - βλ. *Δείτε επίσης*.

Οριακές περιπτώσεις

- **Όρισμα undef** εγείρει προειδοποίηση uninitialized υπό use warnings και επιστρέφει την κενή συμβολοσειρά.
- **Κενή συμβολοσειρά** επιστρέφει την κενή συμβολοσειρά.
- **Οι αριθμοί μετατρέπονται πρώτα σε συμβολοσειρά**: η lc(42) επιστρέφει "42".
- **Το LATIN CAPITAL LETTER SHARP S (U+1E9E)** μετατρέπεται σε πεζό U+00DF (ß) υπό κανόνες Unicode, αλλά μόνο αν το αποτέλεσμα μπορεί να αναπαρασταθεί χωρίς να διασχιστεί το όριο 255/256 με τρόπο που αποδέχεται το τρέχον σύνολο κανόνων. Υπό use locale σε μη-UTF-8 locale, η Perl αφήνει τον χαρακτήρα αμετάβλητο αντί να μαντέψει - και από την Perl 5.22 και μετά αυτό εγείρει προειδοποίηση locale.
- **Το locale και η σημαία UTF-8 αλληλεπιδρούν**: υπό use locale σε μη-UTF-8 locale, οι χαρακτήρες κάτω του 256 ακολουθούν το locale ενώ οι χαρακτήρες 256 και άνω ακολουθούν το Unicode. Μια συμβολοσειρά που περιέχει και τα δύο εύρη θα δεχτεί δύο διαφορετικούς πίνακες πεζών-κεφαλαίων.
- **Tied μεταβλητές** καλούν την FETCH μία φορά. Η lc δεν τροποποιεί την tied τιμή· επιστρέφει ένα απλό (μη-tied) βαθμωτό.
- **Η lc είναι ονομασμένος μοναδιαίος τελεστής**, όχι τελεστής λίστας. Η lc $a, $b αναλύεται ως (lc $a), $b - μόνο το $a μετατρέπεται σε πεζά, και ο τελεστής κόμματος απορρίπτει το αποτέλεσμα. Χρησιμοποιήστε παρενθέσεις ή την map όταν θέλετε να μετατρέψετε πολλαπλές συμβολοσειρές σε πεζά:

  ```
  my @lower = map { lc } @words;
  ```

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `uc` - το αντίστροφο που μετατρέπει σε κεφαλαία, με τους ίδιους κανόνες pragma και locale
- `lcfirst` - μετατρέπει σε πεζά μόνο τον πρώτο χαρακτήρα· χρήσιμη για αποκεφαλαιοποίηση προτάσεων ή αναγνωριστικών
- `fc` - Unicode casefold, η σωστή επιλογή για σύγκριση χωρίς διάκριση πεζών-κεφαλαίων όταν η είσοδος μπορεί να περιέχει μη-ASCII
- `$_` - το προεπιλεγμένο υποκείμενο που διαβάζει η `lc` όταν καλείται χωρίς όρισμα
- `use locale` - ελέγχει αν εφαρμόζονται πίνακες locale ή κανόνες ASCII/Unicode στους χαρακτήρες κάτω του 256
- `use feature 'unicode_strings'` - επιβάλλει πλήρη σημασιολογία πεζών-κεφαλαίων Unicode ανεξάρτητα από τη σημαία UTF-8 της εισόδου

Βαθμωτά και συμβολοσειρές

lcfirst

Επιστρέφει αντίγραφο συμβολοσειράς με τον πρώτο χαρακτήρα της μετατραμμένο σε πεζό.

Η `lcfirst` αγγίζει **μόνο** τον πρώτο χαρακτήρα· το υπόλοιπο της συμβολοσειράς αντιγράφεται αυτούσιο. Η είσοδος δεν τροποποιείται - η `lcfirst` επιστρέφει νέα συμβολοσειρά. Είναι η εσωτερική συνάρτηση που υλοποιεί την ακολουθία διαφυγής `\l` σε συμβολοσειρές με διπλά εισαγωγικά, οπότε η `"\l$name"` και η `lcfirst($name)` παράγουν το ίδιο αποτέλεσμα.

Αν παραλειφθεί η EXPR, η `lcfirst` λειτουργεί στην `$_`.

Σύνοψη

```
lcfirst EXPR
lcfirst
```

Τι επιστρέφεται

Μια νέα συμβολοσειρά ίση με την EXPR με τον πρώτο χαρακτήρα της αντικατεστημένο από την πεζή μορφή που παράγουν οι κανόνες case-folding του χαρακτήρα. Το υπόλοιπο της συμβολοσειράς παραμένει ανέπαφο. Αν η EXPR είναι η κενή συμβολοσειρά, επιστρέφεται η κενή συμβολοσειρά. Αν η EXPR είναι `undef`, το αποτέλεσμα είναι η κενή συμβολοσειρά και εκπέμπεται προειδοποίηση `uninitialized` υπό `use warnings`.

Η `lcfirst` δεν μεταλλάσσει ποτέ το όρισμά της. Για να μετατρέψετε επιτόπια τον πρώτο χαρακτήρα σε πεζό, αναθέστε το αποτέλεσμα πίσω:

```
$name = lcfirst $name;
```

Για την παραλλαγή όλων-των-χαρακτήρων, βλ. `lc`.

Καθολική κατάσταση που επηρεάζει

- `$_` - διαβάζεται όταν η EXPR παραλείπεται.
- locale `LC_CTYPE` - συμβουλεύεται όταν είναι ενεργή η `use locale` (βλ. *Pragmas και κανόνες πεζών-κεφαλαίων* παρακάτω).

Pragmas και κανόνες πεζών-κεφαλαίων

Η `lcfirst` συμπεριφέρεται με τον ίδιο τρόπο υπό διάφορες pragmas όπως η `lc`. Το ενεργό σύνολο κανόνων για τον πρώτο χαρακτήρα εξαρτάται από το ποιες pragmas είναι σε εμβέλεια στο σημείο κλήσης:

- **`use bytes`** - μόνο κανόνες ASCII. Μόνο τα bytes A-Z μετατρέπονται σε a-z· κάθε άλλο byte επιστρέφεται αμετάβλητο.
- **`use locale`** (για `LC_CTYPE`) - το τρέχον locale αποφασίζει τη μετατροπή για σημεία κώδικα κάτω του 256. Τα σημεία κώδικα 256 και άνω μετατρέπονται με κανόνες Unicode όταν η συμβολοσειρά έχει ορισμένη τη σημαία UTF-8. Ένα UTF-8 locale ενεργοποιεί πλήρεις κανόνες Unicode από την Perl 5.20 και μετά.
- **Συμβολοσειρά με σημαία UTF-8** - εφαρμόζονται κανόνες case-folding Unicode.
- **`use feature 'unicode_strings'`** ή **`use locale ':not_characters'`** - εφαρμόζονται κανόνες Unicode ανεξάρτητα από τη σημαία UTF-8.
  - Unicode rules apply regardless of the UTF-8 flag.
- **Διαφορετικά** - κανόνες ASCII. Χαρακτήρες εκτός ASCII επιστρέφονται αμετάβλητοι.

Αυτό σημαίνει ότι η ίδια `lcfirst("É...")` μπορεί να επιστρέψει `"é..."` υπό `use utf8` ή `use feature 'unicode_strings'`, και `"É..."` (αμετάβλητη) υπό τις προεπιλογές καθαρού ASCII. Όταν έχει σημασία η φορητότητα μεταξύ pragmas, δηλώστε ρητά το σύνολο κανόνων:

```
use feature 'unicode_strings';
my $s = lcfirst $input;        # Unicode rules regardless of UTF-8␣
↪flag
```

Παραδείγματα

Βασική κλήση, μετατροπή του πρώτου γράμματος μιας λέξης σε πεζό:

```
my $s = lcfirst "Hello, World";    # "hello, World"
```

Κανονικοποίηση title-case για μια λίστα ονομάτων, όπου ο καλών θέλει το πρώτο γράμμα σε πεζό για ένα URL slug:

```
my @slugs = map { lcfirst } @CamelCaseNames;
```

Η μορφή προεπιλεγμένου ορίσματος διαβάζει από την `$_`:

```
for ("Apple", "Banana", "Cherry") {
    print lcfirst, "\n";           # "apple", "banana", "cherry"
}
```

Αλλάζει μόνο ο **πρώτος** χαρακτήρας - τα μεταγενέστερα κεφαλαία διατηρούνται. Αυτό έχει σημασία για αναγνωριστικά `camelCase` που προέρχονται από `PascalCase`:

```
my $method = lcfirst "GetUserName";   # "getUserName"
```

Οι κανόνες Unicode πρέπει να ζητηθούν ρητά, είτε μέσω συμβολοσειράς εισόδου με σημαία UTF-8 είτε μέσω του χαρακτηριστικού `unicode_strings`:

```
use feature 'unicode_strings';
my $s = lcfirst "Éclair";          # "éclair"
```

```
my $s = lcfirst "Éclair";          # without unicode_strings and no
                                   # UTF-8 flag: "Éclair" unchanged
```

Η ακολουθία διαφυγής `\l` σε συμβολοσειρά με διπλά εισαγωγικά είναι ακριβώς `lcfirst` στον επόμενο χαρακτήρα:

```
my $name = "Perl";
print "\l$name\n";                 # "perl\n"
```

Οριακές περιπτώσεις

- **Κενή συμβολοσειρά**: η `lcfirst("")` επιστρέφει `""`. Καμία προειδοποίηση.
- **Είσοδος `undef`**: η `lcfirst(undef)` επιστρέφει `""`. Εκπέμπει προειδοποίηση `uninitialized` υπό `use warnings`.
- **Πρώτος χαρακτήρας ήδη πεζός**: επιστρέφεται αμετάβλητος· καμία ειδική περίπτωση, καμία συντόμευση που πρέπει να γνωρίζει ο χρήστης.
- **Ο πρώτος χαρακτήρας δεν έχει αντιστοίχιση σε πεζά** (ψηφίο, σημείο στίξης, σύμβολο, οι περισσότεροι CJK): επιστρέφεται αμετάβλητος.
- **Case folds πολλαπλών σημείων κώδικα**: ορισμένοι χαρακτήρες Unicode μετατρέπονται σε πεζά σε περισσότερα από ένα σημεία κώδικα (π.χ. τα τουρκικά ζεύγη dotted/dotless I σε locale-tailored folds). Η `lcfirst` ακολουθεί την *απλή* αντιστοίχιση σε πεζά του Unicode, που είναι πάντα μονού-σημείου-κώδικα - δεν εκτελεί πλήρες case folding. Χρησιμοποιήστε την `lc` σε συνδυασμό με κανονικοποίηση, ή ρητό `Unicode::CaseFold`, όταν έχει σημασία η σημασιολογία πλήρους fold.
- **Εκπλήξεις locale γύρω από το 255/256**: υπό `use locale` με μη-UTF-8 locale, ένας πρώτος χαρακτήρας του οποίου η μετατροπή σε πεζό θα διέσχιζε το όριο 255/256 επιστρέφεται αμετάβλητος, και από την Perl 5.22 και μετά εκπέμπεται προειδοποίηση `locale`.

- **Δεν είναι list-aware**: η `lcfirst` είναι βαθμωτός τελεστής. Η `lcfirst @list` παίρνει το `scalar(@list)` - το πλήθος στοιχείων - και μετατρέπει σε πεζό τον πρώτο χαρακτήρα του, σχεδόν ποτέ αυτό που ήθελε ο καλών. Χρησιμοποιήστε `map { lcfirst } @list` για μετατροπή σε πεζά ανά στοιχείο.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `lc` - μετατρέπει κάθε χαρακτήρα σε πεζό, όχι μόνο τον πρώτο· ίδιοι κανόνες ευαισθησίας σε pragma
- `ucfirst` - μετατρέπει τον πρώτο χαρακτήρα σε κεφαλαίο· η ακριβώς αντίστροφη λειτουργία
- `uc` - μετατρέπει κάθε χαρακτήρα σε κεφαλαίο
- `fc` - Unicode case-folding κατάλληλο για σύγκριση χωρίς διάκριση πεζών-κεφαλαίων, όχι για εμφάνιση
- `$_` - προεπιλεγμένο όρισμα όταν η EXPR παραλείπεται

Βαθμωτά και συμβολοσειρές

length

Επιστρέφει τον αριθμό των χαρακτήρων μιας συμβολοσειράς.

Η `length` μετατρέπει το όρισμά της σε συμβολοσειρά και επιστρέφει πόσους **χαρακτήρες** περιέχει η συμβολοσειρά αυτή. «Χαρακτήρες» σημαίνει λογικούς χαρακτήρες της Perl, όχι bytes: μια συμβολοσειρά που κρατά δέκα κωδικοσημεία έχει length `10` ανεξάρτητα από το πόσα bytes καταλαμβάνουν αυτά τα κωδικοσημεία όταν κωδικοποιηθούν ως UTF-8 σε δίσκο ή στο καλώδιο.

Σύνοψη

```
length EXPR
length
```

Τι επιστρέφεται

Ένας μη αρνητικός ακέραιος - το πλήθος των χαρακτήρων στη συμβολοσειροποιημένη τιμή της EXPR. Αν η EXPR παραλειφθεί, η `length` δρα επί της `$_`. Αν η EXPR είναι `undef`, το αποτέλεσμα είναι `undef`, όχι `0`· αυτή είναι η μοναδική περίπτωση που η `length` δεν επιστρέφει αριθμό, και είναι σκόπιμη - σας επιτρέπει να διακρίνετε ένα μη ορισμένο βαθμωτό από μια κενή συμβολοσειρά χωρίς ξεχωριστό έλεγχο `defined`:

```
length(undef)       # undef
length("")          # 0
length("abc")       # 3
```

Χαρακτήρες, όχι bytes

Σε μια κανονική συμβολοσειρά Perl, η `length` μετρά χαρακτήρες. Το πλήθος των bytes της ίδιας συμβολοσειράς κωδικοποιημένης ως UTF-8 είναι εν γένει διαφορετικό και είναι αυτό που θέλουν τα εξωτερικά συστήματα (μεγέθη αρχείων, ωφέλιμα φορτία δικτύου, κεφαλίδες `Content-Length`). Πάρτε το κωδικοποιώντας πρώτα:

```
use Encode;
my $s     = "na\x{EF}ve";          # five characters
my $chars = length $s;              # 5
my $bytes = length(encode('UTF-8', $s));  # 6
```

Δείτε το `Encode` και το `perlunicode` για ολόκληρη την ιστορία για το πώς οι συμβολοσειρές Perl φέρουν εσωτερική σημαία χαρακτήρων-έναντι-bytes και πότε αυτή η σημαία έχει σημασία.

Παραδείγματα

Βασική μέτρηση:

```
length "hello"                      # 5
length ""                           # 0
```

Το προεπιλεγμένο όρισμα είναι η `$_`:

```
for ("a", "bb", "ccc") {
    print length, "\n";             # 1, 2, 3
}
```

Διάκριση μη ορισμένου από κενό:

```
my $x;
my $y = "";
print defined(length $x) ? "set" : "undef", "\n";   # undef
print defined(length $y) ? "set" : "undef", "\n";   # set
```

Μια συμβολοσειρά Unicode - μέτρηση χαρακτήρων έναντι μέτρησης bytes:

```
my $s = "caf\x{E9}";                # four characters, one of them
↪non-ASCII
length $s;                          # 4
length encode('UTF-8', $s);         # 5
```

Οι αριθμοί μετατρέπονται σε συμβολοσειρά πριν τη μέτρηση, πράγμα ενίοτε χρήσιμο και ενίοτε εκπληκτικό:

```
length 12345                        # 5
length 3.14                         # 4  ("3.14")
length 1e6                          # 7  ("1000000")
```

Οριακές περιπτώσεις

- **`undef` μέσα, `undef` έξω.** Η `length(undef)` επιστρέφει `undef`, όχι `0`. Υπό `use warnings`, η ανάγνωση ενός μη ορισμένου βαθμωτού μέσω `length $x` όπου το `$x` είναι undef **δεν** εκπέμπει προειδοποίηση `uninitialized`, σε αντίθεση με τους περισσότερους άλλους τελεστές.
- **Ολόκληρος πίνακας ή hash**: η `length` δεν μετρά στοιχεία. Συμβολοσειροποιεί το όρισμά της, οπότε η `length @arr` πρώτα αποτιμά τον `@arr` σε βαθμωτό περιβάλλον (το πλήθος των στοιχείων του) και κατόπιν μετρά τα ψηφία αυτού του αριθμού. Για το πραγματικό μέγεθος χρησιμοποιήστε `scalar @arr` ή `scalar keys %hash`:

```
my @arr = (1) x 100;
length @arr;                    # 3  (digits of "100")
scalar @arr;                    # 100
```

- **Δεμένα ή μαγικά βαθμωτά**: η `length` ενεργοποιεί FETCH, οπότε η μετρούμενη τιμή είναι ό,τι επιστρέφει το στρώμα tie εκείνη τη στιγμή.
- **Αντικείμενα με υπερφόρτωση**: η `length $obj` καλεί την υπερφόρτωση συμβολοσειροποίησης `""` (ή υποχωρεί στην προεπιλεγμένη μορφή αναφοράς) και μετρά χαρακτήρες στη συμβολοσειρά που προκύπτει.
- **Πλατείς χαρακτήρες από είσοδο bytes**: μια συμβολοσειρά που διαβάστηκε από handle χωρίς στρώμα κωδικοποίησης είναι συμβολοσειρά bytes. Η `length` τότε μετρά bytes - επειδή για την Perl κάθε byte είναι ένας χαρακτήρας. Αποκωδικοποιήστε πρώτα αν θέλετε μέτρηση χαρακτήρων:

```
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 και pragma**: σε αντίθεση με τις `lc` / `uc` / `fc`, η `length` δεν μεταβάλλεται υπό `use locale`. Ένας χαρακτήρας είναι χαρακτήρας ανεξαρτήτως.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `scalar` - επιβολή βαθμωτού περιβάλλοντος· ο σωστός τρόπος να πάρετε πλήθος στοιχείων από έναν πίνακα (`scalar @arr`) αντί για το length του συμβολοσειροποιημένου πλήθους του
- `substr` - εξάγει υποσυμβολοσειρά κατά μετατόπιση χαρακτήρα· ζευγαρώνει φυσικά με την `length` για εργασίες τεμαχισμού
- `index` - βρίσκει τη θέση μιας υποσυμβολοσειράς· επιστρέφει μετατόπιση χαρακτήρα μετρημένη με τον ίδιο τρόπο που μετρά η `length`
- `sprintf` - τα πλάτη πεδίου `%s` είναι πλήθη χαρακτήρων, ίδια έννοια με την `length`

- `reverse` - σε βαθμωτό περιβάλλον αντιστρέφει μια συμβολοσειρά χαρακτήρα προς χαρακτήρα· ίδιο μοντέλο χαρακτήρα με την `length`
- `pos` - θέση αντιστοίχισης regex, επίσης σε χαρακτήρες

Filehandles, αρχεία, κατάλογοι

link

Δημιουργεί σκληρό σύνδεσμο από το `NEWFILE` προς το υπάρχον `OLDFILE`.

Η `link` προσθέτει μια δεύτερη εγγραφή καταλόγου που αναφέρεται στο ίδιο υποκείμενο αρχείο με το `OLDFILE`. Μετά από επιτυχημένη κλήση, το αρχείο έχει δύο ονόματα: `OLDFILE` και `NEWFILE`. Και τα δύο ονόματα είναι ισότιμα - το ίδιο το αρχείο δεν διαγράφεται μέχρι να αφαιρεθεί κάθε όνομα και καμία διεργασία να μην κρατά το αρχείο ανοιχτό. Η κλήση είναι λεπτό περιτύλιγμα γύρω από την κλήση συστήματος [`link(2)`] και κληρονομεί τους περιορισμούς της: και οι δύο διαδρομές πρέπει να επιλύονται στο **ίδιο σύστημα αρχείων**, και σε συστήματα POSIX μόνο ο superuser μπορεί να κάνει hard-link σε κατάλογο (οι περισσότεροι πυρήνες Linux αρνούνται ακόμα και για τον `root`).

Σύνοψη

```
link OLDFILE, NEWFILE
```

Τι επιστρέφεται

`1` σε περίπτωση επιτυχίας, `0` σε αποτυχία. Σε αποτυχία, το `$!` τίθεται στο σφάλμα του συστήματος - τυπικές τιμές είναι `EXDEV` (σύνδεσμος μεταξύ συσκευών), `EPERM` (ο στόχος είναι κατάλογος, ή το σύστημα αρχείων απαγορεύει σκληρούς συνδέσμους), `EEXIST` (το `NEWFILE` υπάρχει ήδη), `EACCES` (ανεπαρκή δικαιώματα), και `ENOENT` (λείπει το `OLDFILE`).

```
link $src, $dst
    or die "link $src -> $dst failed: $!";
```

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται σε αποτυχία στον κωδικό σφάλματος του συστήματος.

Παραδείγματα

Δημιουργία δεύτερου ονόματος για υπάρχον αρχείο:

```
link "report.txt", "report.bak"
    or die "link failed: $!";
```

Χρησιμοποιήστε `stat` για να παρατηρήσετε την αύξηση του link count. Το τρίτο στοιχείο της λίστας `stat` είναι ο αριθμός των σκληρών συνδέσμων που δείχνουν στο inode:

```
link "data", "data.alt" or die $!;
my $nlink = (stat "data")[3];
print "data has $nlink names\n";    # "data has 2 names"
```

Ατομική αντικατάσταση αρχείου χτίζοντας τη νέα έκδοση με προσωρινό όνομα, και μετά εναλλαγή με `link` + `rename`:

```
open my $fh, ">", "config.new" or die $!;
print $fh $new_contents;
close $fh;
rename "config.new", "config" or die "rename: $!";
```

(Για τη γνήσια ατομική-σε-crash παραλλαγή θέλετε `link` + `unlink` + `rename`· δείτε τα σχόλια στο `perlport`.)

Ανίχνευση αποτυχίας μεταξύ συσκευών και πτώση σε αντιγραφή:

```
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: $!";
    }
}
```

Οριακές περιπτώσεις

- **Κατάλογοι**: η `link` σε κατάλογο αποτυγχάνει με `EPERM` ουσιαστικά σε κάθε σύγχρονο σύστημα. Το POSIX το απαγορεύει για μη-root, και το Linux το απαγορεύει ακόμα και για τον root. Χρησιμοποιήστε `symlink` όταν θέλετε δεύτερο όνομα για κατάλογο.
- **Μεταξύ συστημάτων αρχείων**: ο πυρήνας επιστρέφει `EXDEV` αν τα `OLDFILE` και `NEWFILE` επιλύονται σε διαφορετικά συστήματα αρχείων. Οι σκληροί σύνδεσμοι δεν μπορούν να εκτείνονται μεταξύ σημείων προσάρτησης επειδή μοιράζονται αριθμό inode, ο οποίος έχει σημασία μόνο εντός ενός συστήματος αρχείων.
- **Υπάρχων στόχος**: αν το `NEWFILE` υπάρχει ήδη, η κλήση αποτυγχάνει με `EEXIST`. Η `link` δεν επικαλύπτει ποτέ. Αφαιρέστε πρώτα τον στόχο με `unlink` αν προτίθεστε αντικατάσταση.
- **Symbolic link ως `OLDFILE`**: ο σύνδεσμος γίνεται προς τον **στόχο** του symlink, όχι προς το ίδιο το symlink, στα περισσότερα συστήματα. Αν χρειάζεται να συνδέσετε το symlink, αποαναφέρετέ το ρητά ή χρησιμοποιήστε μια πλατφόρμα που εκθέτει `linkat` με `AT_SYMLINK_FOLLOW` σβησμένο (δεν είναι διαθέσιμη μέσω του core της Perl).
- **Οι σχετικές διαδρομές** επιλύονται έναντι του τρέχοντος καταλόγου εργασίας τη στιγμή της κλήσης, και για τους δύο τελεστέους.
- **Δικαιώματα**: η `link` δεν απαιτεί δικαίωμα εγγραφής στο `OLDFILE` - μόνο δικαίωμα

αναζήτησης στον κατάλογό του και δικαίωμα εγγραφής στον κατάλογο που περιέχει το NEWFILE.

- **Ίδια διαδρομή και στις δύο πλευρές**: η `link $f, $f` αποτυγχάνει με EEXIST.
- **Unlinking**: η αφαίρεση του OLDFILE με unlink μετά από επιτυχημένη link αφήνει το αρχείο πλήρως προσβάσιμο μέσω του NEWFILE. Τα δεδομένα απελευθερώνονται μόνο όταν αφαιρεθεί με unlink το τελευταίο όνομα και κανένας ανοιχτός περιγραφέας αρχείου δεν αναφέρεται σε αυτό.
- **Φορητότητα**: δεν υποστηρίζει κάθε σύστημα αρχείων σκληρούς συνδέσμους. Τα FAT, exFAT, και πολλά δικτυακά συστήματα αρχείων είτε αρνούνται είτε συμπεριφέρονται σιωπηρά ως αντιγραφές. Δείτε το `perlport` για τη μήτρα πλατφορμών.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- symlink - δημιουργία symbolic link αντί για σκληρό σύνδεσμο· λειτουργεί μεταξύ συστημάτων αρχείων και σε καταλόγους
- unlink - αφαιρεί εγγραφή καταλόγου· το ίδιο το αρχείο επιβιώνει για όσο παραμένει άλλος σύνδεσμος ή ανοιχτός περιγραφέας
- rename - μετακίνηση ή μετονομασία αρχείου μέσα σε ένα σύστημα αρχείων· ατομική εκεί όπου το link + unlink δεν είναι
- stat - εξετάστε το link count (πεδίο 3) για να δείτε πόσα ονόματα αναφέρονται στο ίδιο inode
- `$!` - το σφάλμα του συστήματος που τίθεται όταν η link επιστρέφει `0`

Υποδοχές (sockets)

listen

Σημαίνει μια υποδοχή ως παθητική ώστε να μπορεί να δέχεται εισερχόμενες συνδέσεις.

Η `listen` είναι το τρίτο βήμα της κλασικής εγκατάστασης διακομιστή - αφού η socket δημιουργήσει το άκρο και η bind το καρφώσει σε τοπική διεύθυνση, η `listen` το μετάγει στην κατάσταση ακρόασης και διαστασιολογεί την ουρά backlog εκκρεμών συνδέσεων του πυρήνα. Οι εισερχόμενες συνδέσεις στραγγίζονται κατόπιν μία τη φορά με accept. Το QUEUESIZE είναι ο μέγιστος αριθμός συνδέσεων που θα κρατήσει για εσάς ο πυρήνας πριν αρχίσει να αρνείται νέες.

Σύνοψη

```
listen SOCKET, QUEUESIZE
```

Τι επιστρέφεται

Αληθές σε επιτυχία, ψευδές σε αποτυχία (με την `$!` ορισμένη στο errno της υποκείμενης `listen(2)`). Ελέγχετε πάντα την τιμή επιστροφής - ένας διακομιστής που αποτυγχάνει σιωπηρά να εισέλθει σε κατάσταση ακρόασης δεν θα δεχθεί καμία σύνδεση, και η αιτία (αρχείο υποδοχής που έχει μείνει, σύγκρουση θύρας, μη δεσμευμένος περιγραφέας) είναι ορατή μόνο μέσω της `$!`.

```
listen $srv, SOMAXCONN
    or die "listen on $path: $!";
```

Καθολική κατάσταση που επηρεάζει

Η `listen` δεν διαβάζει ειδικές μεταβλητές και γράφει μόνο την `$!` σε αποτυχία. Η κλήση είναι λεπτό περιτύλιγμα της `listen(2)` του πυρήνα· οι επιπτώσεις είναι στον περιγραφέα της υποδοχής, όχι στην κατάσταση της Perl.

Παραδείγματα

Ένας ελάχιστος διακομιστής TCP. `socket`, `bind`, `listen`, `accept` με τη σειρά - παραλείψτε οποιοδήποτε βήμα και η `accept` αποτυγχάνει:

```
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;
}
```

Ένας διακομιστής πεδίου Unix. Ίδια ακολουθία, διαφορετική οικογένεια διευθύνσεων:

```
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: $!";
```

Χρήση του ορισμένου από POSIX ανώτατου ορίου `SOMAXCONN` αντί να κωδικοποιείτε σκληρά κάποιον αριθμό. Ο πυρήνας έτσι κι αλλιώς περικόπτει σιωπηρά μεγαλύτερες τιμές στο δικό του όριο· το `SOMAXCONN` τεκμηριώνει την πρόθεση:

```
use Socket qw(SOMAXCONN);
listen($srv, SOMAXCONN) or die "listen: $!";
```

Μικρό backlog για υπηρεσία που επεξεργάζεται συνδέσεις γρηγορότερα από όσο φτάνουν:

```
listen($srv, 5) or die "listen: $!";
```

Επανάληψη ακρόασης σε υποδοχή που ήδη ακούει. Μια δεύτερη κλήση με διαφορετικό QUEUESIZE είναι νόμιμη στο Linux και αλλάζει επιτόπου το μέγεθος του backlog:

```
listen($srv, 128) or die "listen: $!";
# ... later, under load:
listen($srv, 1024) or die "listen: $!";   # grow the queue
```

Οριακές περιπτώσεις

- **Η σειρά κλήσης έχει σημασία.** Η κλήση `listen` σε υποδοχή που δεν έχει υποστεί `bind` αποτυγχάνει με EDESTADDRREQ (ή σε ορισμένους πυρήνες αυτο-δεσμευμένη εφήμερη θύρα - μη βασίζεστε σε αυτό). Η κλήση `accept` πριν την `listen` αποτυγχάνει με EINVAL.
- **Μόνο για υποδοχές προσανατολισμένες σε σύνδεση.** Η `listen` έχει νόημα σε SOCK_STREAM και SOCK_SEQPACKET. Η κλήση της σε υποδοχή SOCK_DGRAM αποτυγχάνει με EOPNOTSUPP· το UDP δεν έχει έννοια εκκρεμών συνδέσεων.
- **Το QUEUESIZE είναι συμβουλευτικό.** Ο πυρήνας το περικόπτει στο `/proc/sys/net/core/somaxconn` (Linux) ανεξάρτητα από ό,τι ζητάτε. Η τιμή `0` γίνεται αποδεκτή αλλά σημαίνει «βασίσου στο προεπιλεγμένο backlog του πυρήνα»· δεν απενεργοποιεί την ακρόαση.
- **Οι αρνητικές τιμές αντιμετωπίζονται σιωπηρά ως `0`** από τον πυρήνα. Η Perl δεν το διαγιγνώσκει αυτό - ελέγξτε την είσοδό σας πριν την περάσετε στη `listen`.
- **EADDRINUSE στη `listen` έναντι της `bind`.** Οι συγκρούσεις θυρών για TCP/UDP εμφανίζονται στη `bind`· η `listen` σπάνια αποτυγχάνει με αυτό. Για υποδοχές πεδίου Unix, ένα υπολειπόμενο αρχείο υποδοχής από προηγούμενη εκτέλεση προκαλεί αποτυχία της `bind` με EADDRINUSE - αφαιρέστε το με `unlink` πριν τη δέσμευση.
- **Ξεπερασμένες συνδέσεις στην ουρά.** Όταν εκτελεστεί η `accept`, η επιστρεφόμενη σύνδεση μπορεί να έχει ήδη κλείσει από τον ομότιμο (ο πελάτης παράτησε την προσπάθεια όσο βρισκόταν στην ουρά). Χειριστείτε το όπως κάθε άλλο σφάλμα I/O από τη νέα υποδοχή, όχι ως πρόβλημα της `listen`.
- **Μετά από `fork`.** Μια υποδοχή που ακούει και κληρονομείται από παιδί παραμένει σε ακρόαση· τόσο ο γονέας όσο και το παιδί μπορούν να καλέσουν `accept` από την ίδια ουρά. Αυτό είναι το τυπικό μοτίβο διακομιστή pre-fork - μην ξανακαλέσετε `listen` μέσα στο παιδί.
- **Το SIGPIPE είναι άσχετο.** Η ίδια η `listen` δεν εγείρει ποτέ SIGPIPE· αυτό το σήμα προκύπτει σε εγγραφές προς κλειστό ομότιμο μετά την `accept`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `socket` - δημιουργεί το άκρο που η `listen` θα σημάνει ως παθητικό· πάντα η πρώτη κλήση στην ακολουθία
- `bind` - εκχωρεί τοπική διεύθυνση στην υποδοχή· πρέπει να εκτελεστεί πριν από τη `listen` αλλιώς η κλήση αποτυγχάνει
- `accept` - αφαιρεί την επόμενη εκκρεμή σύνδεση από την ουρά backlog που εγκαθίδρυσε η `listen`
- `setsockopt` - θέτει `SO_REUSEADDR` πριν τη `bind` ώστε να αποφύγετε συγκρούσεις `TIME_WAIT` κατά την επανεκκίνηση διακομιστή
- `shutdown` - μισό-κλείνει μια αποδεκτή σύνδεση· δεν χρησιμοποιείται στην ίδια την υποδοχή ακρόασης
- `$!` - κρατά το errno μετά από αποτυχημένη `listen`· ο μοναδικός τρόπος να ξεχωρίσετε το `EADDRINUSE` από το `EACCES` από το `EOPNOTSUPP`

Εμβέλεια

local

Αποθηκεύει την τρέχουσα τιμή μιας μεταβλητής πακέτου και την επαναφέρει όταν η περικλείουσα εμβέλεια τερματίζεται.

Η `local` είναι ο τελεστής δυναμικής εμβέλειας. **Δεν** δηλώνει νέα μεταβλητή - απλώνεται σε μια υπάρχουσα μεταβλητή πακέτου (καθολική), αποθηκεύει την τρέχουσα τιμή σε μια κρυφή στοίβα διάσωσης και κανονίζει η αρχική τιμή να επαναφερθεί όταν ο έλεγχος εγκαταλείπει το τρέχον μπλοκ, την `eval` ή το `do FILE`. Οι καλούμενες υπορουτίνες βλέπουν την προσωρινή τιμή. Για τις περισσότερες ανάγκες τύπου «θέλω μια μεταβλητή τοπική σε αυτό το μπλοκ», το σωστό εργαλείο είναι η `my`· καταφύγετε στην `local` συγκεκριμένα όταν η μεταβλητή πρέπει να είναι καθολική σε επίπεδο πακέτου - οι μεταβλητές στίξης, τα filehandles ή εγγραφές στον πίνακα συμβόλων - και θέλετε αυτή η καθολική να έχει διαφορετική τιμή για τη διάρκεια ενός μπλοκ και κάθε υπορουτίνας που αυτό καλεί.

Σύνοψη

```
local EXPR
local $var
local $var = EXPR
local (@list) = LIST
local $hash{key}
local $array[i]
local *GLOB
```

Τι επιστρέφεται

Την τιμή της τοπικοποιημένης έκφρασης (τη νέα τιμή, μετά από οποιαδήποτε εκχώρηση στην ίδια εντολή). Η παρακολούθηση της στοίβας διάσωσης είναι παρενέργεια· η τιμή επιστροφής έχει σημασία μόνο όταν η `local` εμφανίζεται μέσα σε μεγαλύτερη έκφραση:

```
our $x = 2;
foo($x, local $x = $x + 1, $x);   # foo() receives (2, 3, 3)
```

Δυναμική εμβέλεια, όχι λεξιλογική εμβέλεια

Η `local` και η `my` είναι διαφορετικά είδη εμβέλειας:

- Η `my` δημιουργεί μια **νέα, λεξιλογικά οριοθετημένη** μεταβλητή. Μόνο κώδικας κειμενικά μέσα στο περικλείον μπλοκ μπορεί να τη δει. Οι καλούμενες υπορουτίνες όχι.
- Η `local` **δεν** δημιουργεί τίποτα. Παρακάμπτει προσωρινά την τιμή μιας **υπάρχουσας μεταβλητής πακέτου**. Κάθε κομμάτι κώδικα - αυτό το μπλοκ και κάθε υπορουτίνα που καλεί, άμεσα ή μεταβατικά - βλέπει τη νέα τιμή μέχρι να τερματιστεί η εμβέλεια.

Γι" αυτόν τον λόγο η `local` είναι το σωστό εργαλείο για μεταβλητές στίξης όπως οι `$/`, `$\`, `$,`, `$_` και `$@`: πρόκειται για καθολικές μεταβλητές πακέτου που η Perl runtime (και ό,τι καλείται από τον κώδικά σας) διαβάζει με γνωστά ονόματα. Μια επικάλυψη με `my` δεν θα τις επηρέαζε.

```
sub slurp {
    my ($path) = @_;
    open my $fh, "<", $path or die "open $path: $!";
    local $/;                    # undef $/ for this sub only
    return <$fh>;                # read to EOF in one go
}
```

Η γραμμή `local $/` πιο πάνω είναι το πιο κοινό ιδίωμα `local` στον πραγματικό κώδικα: αναιρεί προσωρινά τον διαχωριστή εγγραφών εισόδου ώστε η `readline` (`<$fh>`) να επιστρέψει ολόκληρο το αρχείο. Όταν η `slurp` επιστρέφει, η `$/` επαναφέρεται σε ό,τι ήταν πριν - ακόμη και αν ο καλών την είχε επίσης αλλάξει.

Τι μπορείτε και τι δεν μπορείτε να τοπικοποιήσετε

Μπορείτε να εφαρμόσετε `local` σε:

- **Βαθμωτά, πίνακες, hashes πακέτου** - `local $Pkg::var`, `local @Pkg::arr`, `local %Pkg::hash`. Το bareword `local $var` στοχεύει τη μεταβλητή στο τρέχον πακέτο.
- **Ήδη δηλωμένες μεταβλητές `our`** - η `our` δημιουργεί λεξιλογικό ψευδώνυμο προς μια μεταβλητή πακέτου, οπότε η `local` πάνω της τοπικοποιεί την υποκείμενη καθολική.
- **Μεμονωμένα στοιχεία και slices πινάκων/hashes πακέτου** - `local $hash{key}`, `local $array[3]`, `local @hash{qw(a b)}`, `local @array[0..2]`.

- **Globs** - η `local *name` δημιουργεί μια καινούργια εγγραφή στον πίνακα συμβόλων, έτσι ώστε οι `$name`, `@name`, `%name`, `&name` και το filehandle `name` να επαναφέρονται δυναμικά όλα μαζί.
- **Υπό συνθήκη lvalues** - `local ($cond ? $v1 : $v2)` όταν η επιλεγμένη διακλάδωση είναι η ίδια τοπικοποιήσιμη.

Δεν μπορείτε να εφαρμόσετε `local` σε:

- Μια μεταβλητή `my`. Οι μεταβλητές my είναι λεξιλογικές που ζουν σε pad, όχι στον πίνακα συμβόλων· δεν έχουν όνομα πακέτου ώστε η `local` να τις αποθηκεύσει κάτω από αυτό. Η προσπάθεια αποτελεί σφάλμα κατά τη μεταγλώττιση: `Can't localize lexical variable $x`.
- Μια μεταβλητή `state`. Για τον ίδιο λόγο.
- Μόνο-για-ανάγνωση μαγικές καθολικές. Το `local $1 = 2` αποτυγχάνει με `Modification of a read-only value attempted` - οι μεταβλητές σύλληψης δεν μπορούν να δεχθούν εκχώρηση. (Η μία εξαίρεση είναι η `$_`: η `local $_`, από την 5.14, ρητά αφαιρεί τη μαγεία ώστε να μπορεί να επαναχρησιμοποιηθεί με ασφάλεια σε υπορουτίνα.)

Τοπικοποίηση στοιχείων σύνθετων τύπων

Τα `local $hash{key}` και `local $array[i]` αποθηκεύουν τη **συγκεκριμένη θέση**, όχι την τιμή που τυχαίνει να βρίσκεται εκεί. Όταν η εμβέλεια τερματίζεται, η αρχική τιμή επαναφέρεται σε εκείνη ακριβώς τη θέση - ακόμη και αν το στοιχείο διαγράφηκε ή ο πίνακας κοντύνε στο μεταξύ. Ένα διαγραμμένο κλειδί hash επαναεμφανίζεται· ένα στοιχείο πίνακα που έγινε pop επανεμφανίζεται, γεμίζοντας τον πίνακα με `undef` αν χρειάζεται.

```
our %hash = (a => "is");
{
    local $hash{a} = "drill";
    delete $hash{a};              # gone for now...
}
print $hash{a}, "\n";             # ...but restored to "is"
```

Παραδείγματα

Βασικά: δώστε σε μια καθολική μια προσωρινή τιμή για ένα μπλοκ. Οι υπορουτίνες που καλούνται μέσα από το μπλοκ βλέπουν τη νέα τιμή.

```
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"
```

Ιδίωμα slurp-αρχείου με `$/`:

```
my $content = do {
    open my $fh, "<", $path or die "open $path: $!";
    local $/;                       # list context for <> now reads
↪all
    <$fh>;
};
```

Παγιδεύστε προσωρινά την $@ ώστε μια εσωτερική eval να μην επικαλύψει το εκκρεμές σφάλμα του καλούντος:

```
sub try_cleanup {
    local $@;                       # caller's $@ preserved
    eval { risky_cleanup() };
    # even if eval failed, caller's $@ is untouched
}
```

Προεπιλογές print σε στιλ Python, οριοθετημένες σε ένα μπλοκ:

```
{
    local $, = " ";
    local $\ = "\n";
    print 1, 2, 3;                  # "1 2 3\n"
}
print 1, 2, 3;                      # "123" - defaults restored
```

Η μορφή λίστας χρειάζεται παρενθέσεις. Οι παρενθέσεις δίνουν επίσης στη δεξιά πλευρά περιβάλλον λίστας - ακριβώς όπως και η my:

```
our (@wid, %get);
local (@wid, %get) = (@defaults, %overrides);
```

Τοπικοποιήστε ένα μεμονωμένο στοιχείο hash - χρήσιμο για να περάσετε παρακάμψεις μέσα από μια κλήση χωρίς να αγγίξετε άσχετα κλειδιά:

```
our %config = (timeout => 30, retries => 3);
{
    local $config{timeout} = 5;
    do_probe();                     # sees timeout => 5
}
# %config back to (timeout => 30, retries => 3)
```

Τοπικοποίηση glob - ολόκληρη η εγγραφή στον πίνακα συμβόλων για το LOG αντικαθίσταται:

```
{
    local *LOG;
    open LOG, ">", "/tmp/trace.$$" or die $!;
    run_traced();                   # sees our private LOG filehandle
}
# original LOG (if any) is back
```

Οριακές περιπτώσεις

- **Τελεστής χρόνου εκτέλεσης, όχι δήλωση.** Η `local` εκτελείται κάθε φορά που ο έλεγχος φτάνει σε αυτήν. Μέσα σε βρόχο αποθηκεύει και επαναφέρει σε κάθε επανάληψη - μετρήσιμη επιβάρυνση. Βγάλτε την έξω από τον βρόχο όταν η πρόθεση είναι «για ολόκληρο τον βρόχο»:

```
{ local $/ = "\n\n";            # once, not per iteration
    while (<$fh>) { ... }
}
```

- **Η εκχώρηση αποτιμάται πριν από την αποθήκευση.** Η δεξιά πλευρά εκτελείται στην περικλείουσα δυναμική εμβέλεια, οπότε η `local $x = $x + 1` διαβάζει την παλιά `$x`, μετά αποθηκεύει, και κατόπιν εκχωρεί τη νέα τιμή.
- **Το όριο εμβέλειας είναι δυναμικό, όχι λεξιλογικό.** Η αποθήκευση αναιρείται όταν το **τρέχον μπλοκ εξέρχεται σε χρόνο εκτέλεσης**, όχι όταν το πηγαίο μπλοκ τερματίζεται κειμενικά. Η έξοδος μέσω `return`, `die`, `last`, `next` ή μη-τοπικού goto εξακολουθεί να ενεργοποιεί την επαναφορά.
- **Η `local` σε μια `my` είναι σφάλμα κατά τη μεταγλώττιση.** Ο αναλυτής απορρίπτει το `my $x; local $x;` με `Can't localize lexical variable $x`. Χρησιμοποιήστε `our` (ή πλήρως προσδιορισμένο όνομα) αν πραγματικά χρειάζεστε δυναμική εμβέλεια σε εκείνο το όνομα.
- **Πίνακες και hashes με tie** δεν συμπεριφέρονται προς το παρόν όπως περιγράφει το POD - η τοπικοποίηση ολόκληρου του συσσωματώματος είναι ελαττωματική στο upstream. Η τοπικοποίηση μεμονωμένων στοιχείων ενός συσσωματώματος με tie είναι ασφαλής. Αυτή η επιφύλαξη κληρονομείται από την upstream Perl.
- **Στις μόνο-για-ανάγνωση μαγικές δεν μπορεί να γίνει εκχώρηση.** Το `local $1 = "x"` πεθαίνει σε χρόνο εκτέλεσης. Μπορείτε να κάνετε `local $_` (η μαγεία αφαιρείται από την 5.14) αλλά όχι `local $1`, `local $&`, κ.λπ.
- **Αρνητικοί δείκτες πίνακα** στο `local $array[-1]` τεκμηριώνονται ως «ιδιαίτερα εκπληκτικοί» στο upstream· αντιμετωπίστε τη συμπεριφορά ως ασταθή και αποφύγετέ την.
- **Το `delete local`** επεκτείνει την `local` στην αφαίρεση στοιχείου: το `delete local $hash{key}` διαγράφει την εγγραφή για το τρέχον μπλοκ και την επαναφέρει κατά την έξοδο. Επιστρέφει την τιμή που υπήρχε πριν από την τοπικοποίηση.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `my` - λεξιλογική (κατά τη μεταγλώττιση, με εμβέλεια μπλοκ) δήλωση· το σωστό εργαλείο όταν θέλετε μια πραγματικά ιδιωτική μεταβλητή
- `our` - λεξιλογικό ψευδώνυμο προς μια μεταβλητή πακέτου· συνηθισμένος εταίρος της `local` όταν η καθολική βρίσκεται σε άλλο πακέτο

- `state` - επίμονη λεξιλογική· δεν μπορεί να τοπικοποιηθεί με `local`
- `delete` - σε συνδυασμό με την `local` (`delete local $h{k}`) για να περιορίσει τη διαγραφή στοιχείου στο τρέχον μπλοκ
- `eval` - συνηθισμένος στόχος της `local`: η `local $@` γύρω από εσωτερική `eval` αποτρέπει τη διαρροή σφαλμάτων προς τον καλούντα
- `$/` - διαχωριστής εγγραφών εισόδου· καθιερωμένος στόχος της `local` για αναγνώσεις αρχείων σε λειτουργία slurp
- `$_` - προεπιλεγμένο βαθμωτό· η `local $_` αφαιρεί τη μαγεία (από την 5.14) ώστε μια υπορουτίνα να μπορεί να την επαναχρησιμοποιήσει με ασφάλεια
- `$@` - τελευταίο σφάλμα `eval`· η `local $@` θωρακίζει τον καλούντα από το σφάλμα μιας εσωτερικής `eval`

Χρόνος

localtime

Μετατρέπει έναν χρόνο epoch σε αναλυτικό ημερολογιακό χρόνο στην τοπική ζώνη ώρας.

Η `localtime` παίρνει μια τιμή Unix epoch - δευτερόλεπτα από την 1970-01-01 UTC, τη μορφή που επιστρέφει η `time` - και την αναλύει σε έτος, μήνα, ημέρα, ώρα, λεπτό, δευτερόλεπτο, ημέρα εβδομάδας, ημέρα έτους και σημαία θερινής ώρας, όλα εκφρασμένα στην **τοπική** ζώνη ώρας. Η ζώνη ώρας διαβάζεται από τη μεταβλητή περιβάλλοντος `TZ` αν είναι ορισμένη, αλλιώς από την προεπιλογή του συστήματος (συνήθως `/etc/localtime`). Για τη συντρόφισσά της σε UTC που αγνοεί το `TZ`, δείτε την `gmtime`.

Σύνοψη

```
localtime EXPR
localtime
```

Τι επιστρέφεται

Η μορφή της επιστροφής εξαρτάται από το περιβάλλον, και η `localtime` είναι μία από τις λίγες ενσωματωμένες όπου η **ίδια έκφραση σε περιβάλλον λίστας και σε βαθμωτό περιβάλλον παράγει εντελώς διαφορετικές απαντήσεις**.

Περιβάλλον λίστας - μια λίστα 9 στοιχείων, όλα αριθμητικά, παρμένα απευθείας από το C `struct tm`:

```
#     0     1     2      3     4     5     6     7     8
my ($sec, $min, $hour, $mday, $mon, $year, $wday, $yday, $isdst) =
                                                  localtime(time);
```

Δείκτης	Όνομα	Εύρος	Σημειώσεις
0	`$sec`	`0..60`	το 60 είναι δυνατό για leap seconds
1	`$min`	`0..59`	
2	`$hour`	`0..23`	
3	`$mday`	`1..31`	ημέρα του μήνα με αρχή το 1
4	`$mon`	`0..11`	**με αρχή το 0**: `0` = Ιανουάριος, 11 = Δεκέμβριος
5	`$year`	έτη - 1900	π.χ. 129 για το 2029
6	`$wday`	`0..6`	`0` = Κυριακή, 3 = Τετάρτη
7	`$yday`	`0..365`	ημέρα του έτους με αρχή το 0· 365 σε δίσεκτο έτος
8	`$isdst`	λογική τιμή	αληθές όταν ισχύει το DST για την τοπική TZ

Το `$mon` με αρχή το 0 και το `$year` με βάση το 1900 αποτελούν μακροχρόνιες κληρονομιές από τη C· τα περισσότερα σφάλματα που γράφονται γύρω από την `localtime` προέρχονται από τη λήθη ακριβώς ενός από αυτά. Η διόρθωση για το `$year` είναι `+= 1900`:

```
my @t = localtime;
my $full_year = $t[5] + 1900;
```

Βαθμωτό περιβάλλον - η οικεία αγγλική συμβολοσειρά `ctime(3)`, πάντοτε 24 χαρακτήρες, πάντοτε στα αγγλικά ανεξάρτητα από τις τοπικές ρυθμίσεις:

```
my $now_string = localtime;      # "Thu Oct 13 04:54:34 1994"
```

Για συμβολοσειρά με επίγνωση τοπικών ρυθμίσεων ή προσαρμοσμένη μορφή, μην προσπαθήσετε να αναλύσετε τη βαθμωτή μορφή. Περάστε μέσα από τη μορφή λίστας και την `POSIX::strftime`:

```
use POSIX qw(strftime);
my $now_string = strftime "%a %b %e %H:%M:%S %Y", localtime;
```

Καθολική κατάσταση που επηρεάζει

- **`TZ`** στο `%ENV` - επιλέγει την τοπική ζώνη ώρας. Ένα πρόγραμμα που θέλει να ερμηνεύσει μια χρονοσήμανση σε συγκεκριμένη ζώνη ορίζει το `$ENV{TZ}` και καλεί την `POSIX::tzset` πριν από την `localtime` (δείτε *Οριακές περιπτώσεις* για την επιφύλαξη).
- Το zoneinfo του συστήματος (`/etc/localtime`, `/usr/share/zoneinfo/...`) όταν το `TZ` δεν είναι ορισμένο - διαβάζεται οκνηρά από τη βιβλιοθήκη C, όχι απευθείας από την Perl.
- Οι αλλαγές στο `%ENV` που αφορούν το `TZ` **δεν** ενεργοποιούνται πάντα στην επόμενη κλήση· δείτε *Οριακές περιπτώσεις*.

Η `localtime` δεν επηρεάζει το `$_` ούτε καμία καθολική σχετική με έξοδο.

Παραδείγματα

Τρέχουσα ημερομηνία πραγματικού χρόνου σε αναγνώσιμη από ανθρώπους συμβολοσειρά:

```
my $now = localtime;              # "Wed Apr 23 14:07:12 2025"
```

Τρέχουσα ημερομηνία ως λίστα 9 στοιχείων - προσέξτε τις δύο μετατοπίσεις:

```
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"
```

Όνομα μήνα από το $mon με χρήση πίνακα αναζήτησης - αυτός είναι ο καθιερωμένος τρόπος, επειδή το $mon έχει αρχή το 0 ακριβώς για να ευρετηριάζει πίνακα:

```
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`

- η σωστή απάντηση όποτε χρειάζεστε κάτι άλλο πέρα από τη γυμνή αγγλική συμβολοσειρά `ctime`:

```
use POSIX qw(strftime);
my @now = localtime;
my $stamp = strftime "%Y-%m-%d %H:%M:%S", @now;   # "2025-04-23␣
 ↪14:07:12"
```

Ερμηνεία μιας αποθηκευμένης epoch σε συγκεκριμένη ζώνη. Ορίστε το `TZ`, καλέστε την `POSIX::tzset`, και κατόπιν την `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"
```

Παγίδα περιβάλλοντος - η `localtime` μέσα σε `print LIST` βρίσκεται σε **περιβάλλον λίστας**, οπότε λαμβάνετε εννέα αριθμούς συνενωμένους χωρίς κενά. Χρησιμοποιήστε `scalar` για να εξαναγκάσετε τη μορφή συμβολοσειράς:

```
print localtime, "\n";            # e.g. "12724232312505230"
print scalar localtime, "\n";     # "Wed Apr 23 14:07:12 2025"
```

Οριακές περιπτώσεις

- **Χωρίς όρισμα**: η `localtime` χωρίς όρισμα χρησιμοποιεί την `time` - την τρέχουσα στιγμή πραγματικού χρόνου.
- **Αρνητική epoch**: ημερομηνίες πριν από την 1970-01-01 UTC γίνονται δεκτές σε συστήματα με προσημασμένο `time_t`. Η φορητότητα πολύ απομακρυσμένων χρόνων στο παρελθόν ή στο μέλλον εξαρτάται από τη βιβλιοθήκη C της πλατφόρμας.

- **Παγίδα του `$year`**: το `$year + 1900` δίνει το πλήρες έτος. Το `"19$year"` ήταν σφάλμα Y2K και εξακολουθεί να είναι λάθος· γράψτε `1900 + $year`.
- **Παγίδα του `$mon`**: το `$mon` έχει αρχή το 0. Η έξοδος της `localtime` που τροφοδοτεί οτιδήποτε έχει αρχή το 1 (περιλαμβανομένης της `POSIX::mktime`) **δεν** χρειάζεται προσαρμογή - η `mktime` αναμένει την ίδια σύμβαση με αρχή το 0. Εξωτερικά συστήματα (βάσεις δεδομένων, συμβολοσειρές ISO-8601) συνήθως θέλουν `$mon + 1`.
- **Το `$isdst` είναι ενδεικτικό**, όχι άμεσα συγκρίσιμο μεταξύ ζωνών. Το 1 σημαίνει «η τοπική ζώνη εφαρμόζει αυτή τη στιγμή τη μετατόπιση DST»· το `0` σημαίνει «τυπική ώρα»· το `-1` εμφανίζεται περιστασιακά όταν οι κανόνες της ζώνης δεν μπορούν να αποφασίσουν (π.χ. η ώρα που δεν υπάρχει σε μέρα που ο δείκτης πηγαίνει μπροστά την άνοιξη).
- **Επιφύλαξη επαναφόρτωσης του `TZ`**: οι περισσότερες βιβλιοθήκες C (συμπεριλαμβανομένης της glibc) αποθηκεύουν στη μνήμη τους τους αναλυμένους κανόνες ζώνης κατά την πρώτη αναζήτηση. Μια μεταγενέστερη εκχώρηση στο `$ENV{TZ}` **δεν** υιοθετείται αυτόματα - καλέστε την `POSIX::tzset` μετά από κάθε αλλαγή, αλλιώς θα συνεχίσει να χρησιμοποιείται η παλιά ζώνη.
- **Η βαθμωτή συμβολοσειρά είναι πάντοτε αγγλική** και **δεν** επηρεάζεται από το `LC_TIME` ή οποιαδήποτε άλλη ρύθμιση τοπικών. Για συμβολοσειρά τοπικοποιημένη, περάστε μέσα από την `POSIX::strftime` με το `LC_TIME` ορισμένο στις επιθυμητές τοπικές ρυθμίσεις.
- **Ασφάλεια νημάτων**: σε συστήματα όπου η κλήση C `localtime(3)` δεν είναι επαναεισερχόμενη, ταυτόχρονα νήματα που μεταβάλλουν το `TZ` μπορούν να εμφανίσουν race. Στην πράξη η pperl και η σύγχρονη glibc χρησιμοποιούν εσωτερικά την επαναεισερχόμενη `localtime_r`, αλλά το `TZ` παραμένει καθολικό σε επίπεδο διεργασίας.
- **Leap seconds**: το `$sec` μπορεί νόμιμα να είναι `60` σε συστήματα με υποστήριξη leap seconds. Τα περισσότερα συστήματα τα διαχέουν (smear) ή τα αγνοούν, και ποτέ δεν θα το δείτε· κώδικας που δεν πρέπει να σπάει στο `60` το χειρίζεται ήδη.
- **Πλάτος των `%a` / `%b`** στην `POSIX::strftime`: τα συντομευμένα ονόματα ημερών και μηνών **δεν εγγυώνται τρεις χαρακτήρες** σε κάθε locale. Κώδικας που υποθέτει σταθερό πλάτος στήλης θα ευθυγραμμιστεί λάθος στα γερμανικά, στα γαλλικά και σε πολλά άλλα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `gmtime` - η ίδια ανάλυση σε UTC· δεν συμβουλεύεται το `TZ` και είναι η σωστή επιλογή όταν θέλετε τιμή ανεξάρτητη από ζώνη ώρας
- `time` - η πηγή epoch που η `localtime` λαμβάνει συχνότερα ως όρισμα
- `POSIX::strftime` - μορφοποιήστε μια λίστα `localtime` σε όποια διάταξη συμβολοσειράς χρειάζεστε, με επίγνωση τοπικών ρυθμίσεων

- `POSIX::mktime` - αντίστροφη κατεύθυνση: μετατρέπει μια λίστα 9 στοιχείων πίσω σε τιμή epoch
- `POSIX::tzset` - επανάγνωση του `TZ` μετά την αλλαγή του, υποχρεωτική αν θέλετε η αλλαγή να τεθεί σε ισχύ
- `$ENV{TZ}` - η μεταβλητή περιβάλλοντος που διαβάζει η `localtime` για να επιλέξει την τοπική ζώνη

Διάφορα

lock

Τοποθετεί προτρεπτικό κλείδωμα σε κοινόχρηστη μεταβλητή, πίνακα, πίνακα κατακερματισμού ή υπορουτίνα μέχρι το κλείδωμα να βγει εκτός εμβέλειας.

Η `lock` είναι το πρωτογενές συγχρονισμού του μοντέλου διερμηνέα-threads της Perl (`use threads; use threads::shared`). Όταν το `THING` είναι μεταβλητή που έχει δηλωθεί ως `:shared` - ή αναφορά σε κοινόχρηστο σύνθετο - η `lock` αποκτά προτρεπτικό mutex σε αυτό το δεδομένο για το υπόλοιπο του περικλείοντος μπλοκ. Όταν το μπλοκ εξέλθει, το κλείδωμα απελευθερώνεται. Οποιοδήποτε άλλο thread καλεί `lock` στο ίδιο δεδομένο μπλοκάρει μέχρι ο κάτοχος να το απελευθερώσει.

Το κλείδωμα είναι **προτρεπτικό**: σταματάει άλλα threads που επίσης καλούν `lock` στην ίδια μεταβλητή, τίποτα άλλο. Ένα thread που απλώς διαβάζει ή γράφει την κοινόχρηστη μεταβλητή χωρίς να την κλειδώνει δεν μπλοκάρεται και δεν σειριοποιείται. Η `lock` είναι σύμβαση, όχι φραγμός.

Σύνοψη

```
lock $shared_scalar
lock @shared_array
lock %shared_hash
lock &shared_sub
lock $ref_to_shared_aggregate
```

Τι επιστρέφεται

Το ίδιο το όρισμα:

- Για βαθμωτό, η βαθμωτή τιμή.
- Για πίνακα, πίνακα κατακερματισμού ή υπορουτίνα, αναφορά σε αυτό.

Η τιμή επιστροφής σπάνια χρησιμοποιείται. Η `lock` καλείται για την παρενέργειά της (απόκτηση του mutex), όχι για την τιμή της.

Το κλείδωμα έχει εμβέλεια μπλοκ, όχι εμβέλεια δήλωσης

Το κλείδωμα απελευθερώνεται όταν το εσώτατο περικλείον μπλοκ εξέλθει, όχι όταν τελειώσει η δήλωση `lock`. Δύο συνέπειες:

- Η `lock` μέσα σε μπλοκ `{ ... }` σειριοποιεί μόνο τη διάρκεια αυτού του μπλοκ. Η έξοδος από το μπλοκ - με φυσιολογική ολοκλήρωση, `return`, `last`, `die`, οτιδήποτε - απελευθερώνει το κλείδωμα.

- Δεν μπορείτε να απελευθερώσετε ένα κλείδωμα νωρίτερα. Δεν υπάρχει unlock. Αν χρειάζεστε στενότερο κρίσιμο τμήμα, εισαγάγετε ένα σφιχτότερο μπλοκ { ... }:

```
{
    lock %shared_hash;
    $shared_hash{$key} = $value;
}   # lock released here
do_unlocked_work();
```

Ασθενής δεσμευμένη λέξη

Η lock είναι **ασθενής δεσμευμένη λέξη**. Αν υπάρχει υπορουτίνα με όνομα lock εντός εμβέλειας στο σημείο της κλήσης (δηλωμένη ή εισαγμένη πριν το σημείο κλήσης), καλείται αυτή η υπορουτίνα αντί για την ενσωματωμένη. Αυτό κάνει την lock ασφαλή να χρησιμοποιείται ως όνομα μεθόδου ή συνάρτησης σε κώδικα που δεν χρησιμοποιεί threads. Για να επιβάλλετε την ενσωματωμένη όταν μια ομώνυμη sub είναι εντός εμβέλειας, καλέστε την μέσω του πακέτου της: CORE::lock($thing).

Παραδείγματα

Σειριοποιήστε μια ενημέρωση πίνακα κατακερματισμού μεταξύ threads:

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

my %counts :shared;

sub bump {
    my $key = shift;
    lock %counts;
    $counts{$key}++;
}
```

Κλειδώστε ένα σύνθετο μέσω αναφοράς. Η lock ακολουθεί ένα επίπεδο αναφοράς για να βρει το δεδομένο που θα κλειδώσει:

```
my @queue :shared;
my $queue_ref = \@queue;

sub enqueue {
    my $item = shift;
    lock $queue_ref;        # locks @queue
    push @queue, $item;
}
```

Στενέψτε το κρίσιμο τμήμα με ένα εσωτερικό μπλοκ ώστε άσχετη εργασία να τρέχει χωρίς κλείδωμα:

```
my $result;
{
    lock %cache;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    $result = $cache{$key};
}
expensive_post_processing($result);   # runs without the lock
```

Χωρίς `use threads::shared`, η `lock` είναι no-op. Αυτό την καθιστά ακίνδυνη ώστε να αφεθεί σε κώδικα που μπορεί να τρέχει με ένα μόνο thread:

```
# no `use threads::shared` here
lock $x;                    # does nothing, does not warn
```

Οριακές περιπτώσεις

- **Μη κοινόχρηστη μεταβλητή**: η `lock` σε μεταβλητή που δεν έχει δηλωθεί `:shared` και δεν είναι αναφορά σε κοινόχρηστα δεδομένα δεν κάνει κάτι χρήσιμο. Στο upstream μπορεί σε κάποιες περιπτώσεις να εκπέμψει σφάλμα· σε άλλες είναι σιωπηρά no-op. Μη βασίζεστε στη `lock` για να πιάσετε το σφάλμα «ξέχασα να μοιραστώ».
- **Αναφορά έναντι σύνθετου**: τόσο η `lock \%h` όσο και η `lock %h` κλειδώνουν τον %h. Η `lock` αποαναφέρει ένα επίπεδο για να βρει το κοινόχρηστο δεδομένο.
- **Αναδρομικό κλείδωμα μέσα σε ένα thread**: το ίδιο thread μπορεί να ξανααποκτήσει ένα κλείδωμα που ήδη κρατάει· το mutex είναι αναδρομικό ανά thread. Άλλο thread παρ" όλα αυτά μπλοκάρει.
- **Καμία `unlock`**: η απελευθέρωση πριν την έξοδο μπλοκ απαιτεί αναδιάρθρωση σε στενότερη εμβέλεια. Τα μοτίβα `undef $lock_guard` από άλλες γλώσσες δεν ισχύουν - η `lock` δεν είναι τιμή που αποθηκεύετε.
- **Αμφισημία ασθενούς δεσμευμένης λέξης**: μια sub `lock` που δηλώθηκε ή εισήχθη νωρίτερα υπερισχύει της ενσωματωμένης. Χρησιμοποιήστε `CORE::lock` για αποσαφήνιση.
- **Όχι μεταξύ διεργασιών**: η `lock` συντονίζει threads μέσα σε μία διεργασία διερμηνέα. Για κλείδωμα σε επίπεδο αρχείου μεταξύ διεργασιών χρησιμοποιήστε την `flock`· για προτρεπτικά κλειδώματα συστήματος αρχείων αυτοί είναι ξεχωριστοί μηχανισμοί χωρίς αλληλεπίδραση.

Διαφορές από το upstream

- Η pperl δεν υλοποιεί διερμηνέα-threads. Ο opcode `lock` αναγνωρίζεται από τον αναλυτή αλλά δεν έχει επίδραση στον χρόνο εκτέλεσης - συμπεριφέρεται ακριβώς όπως η upstream `lock` σε build χωρίς `threads::shared`: σιωπηρό no-op που επιστρέφει το όρισμά του. Κώδικας που χρησιμοποιεί τη `lock` αμυντικά (όπως συνιστά η τεκμηρίωση upstream για συμβατότητα με ένα μόνο thread) τρέχει χωρίς αλλαγή. Κώδικας που εξαρτάται από τη `lock` για πραγματικό συγχρονισμό δεν θα συγχρονιστεί υπό pperl.
- Η συνοδευτική διανομή `threads::shared` δεν είναι διαθέσιμη στην pperl· το `use threads::shared` αποτυγχάνει κατά τη μεταγλώττιση. Ο παραλληλισμός στην pperl εκτίθεται μέσω αυτο-παραλληλοποίησης JIT (Rayon) σε συγκεκριμένα σχήματα βρόχου, όχι μέσω ορατών στον χρήστη ithreads.

Δείτε επίσης

- `flock` - προτρεπτικό κλείδωμα σε filehandle, που συντονίζει μεταξύ διεργασιών αντί για threads
- `wait` - συγκομιδή θυγατρικής διεργασίας· το ισοδύναμο σε επίπεδο διεργασίας της αναμονής για άλλη μονάδα εκτέλεσης
- `fork` - το άλλο πρωτογενές ταυτοχρονικότητας στον πυρήνα της Perl, χωρίς προεπιλεγμένη κοινόχρηστη μνήμη
- `our` - δηλώστε μεταβλητή με εμβέλεια πακέτου· συνήθως συνδυάζεται με `:shared` σε κώδικα με threads
- `local` - σύνδεση με δυναμική εμβέλεια, μερικές φορές μπερδεύεται με τη `lock` επειδή και οι δύο είναι οριοθετημένες σε εμβέλεια και αμφότερες επαναφέρουν στην έξοδο μπλοκ

Αριθμητικές συναρτήσεις

log

Επιστρέφει τον φυσικό λογάριθμο (βάση *e*) ενός αριθμού.

Η `log` είναι η αντίστροφη της `exp`. Παίρνει αριθμητική `EXPR`, την εξαναγκάζει σε float διπλής ακρίβειας αν χρειάζεται, και επιστρέφει τον λογάριθμό της με βάση *e* (αριθμός του Euler, περίπου `2.718281828459045`). Αν η `EXPR` παραλείπεται, η `log` λειτουργεί επί της `$_`.

Δεν υπάρχει ενσωματωμένη για λογαρίθμους σε αυθαίρετη βάση - χρησιμοποιήστε βασική άλγεβρα (δείτε τα παραδείγματα παρακάτω) ή τις `POSIX::log10` / `POSIX::log2` για τις δύο συνηθισμένες σταθερές βάσεις.

Σύνοψη

```
log EXPR
log
```

Τι επιστρέφεται

Float διπλής ακρίβειας: ο φυσικός λογάριθμος της `EXPR`. Συγκεκριμένα:

- Το `log(1)` είναι `0`.
- Το `log(exp(1))` είναι `1` (η ταυτότητα ορισμού).
- Το `log(x)` είναι γνησίως αύξον για `x > 0`.

Οι ειδικές είσοδοι ακολουθούν τους κανόνες IEEE-754 όπως τους παρέχει η `log(3)` της C library της πλατφόρμας:

- Το `log(0)` επιστρέφει αρνητικό άπειρο (`-Inf`).
- Το `log(x)` για `x < 0` επιστρέφει `NaN`.
- Το `log(Inf)` επιστρέφει `Inf`· το `log(NaN)` επιστρέφει `NaN`.

Το αποτέλεσμα είναι πάντα float, ακόμη και όταν η είσοδος είναι ακέραιος ή συμβολοσειρά.

Παραδείγματα

Η ταυτότητα ορισμού - ο φυσικός λογάριθμος του *e* είναι 1:

```
use POSIX ();
print log(exp(1)), "\n";            # 1
```

Λογάριθμος με βάση 10, χτισμένος από τη `log` με αλλαγή βάσης. Ο λογάριθμος με βάση N ενός αριθμού ισούται με τον φυσικό λογάριθμό του διά τον φυσικό λογάριθμο του N:

```
sub log10 {
    my ($n) = @_;
    return log($n) / log(10);
}

print log10(1000), "\n";            # 3
print log10(0.01), "\n";            # -2
```

Το ίδιο ιδίωμα γενικευμένο σε οποιαδήποτε βάση - χρήσιμο για bits (βάση 2), ντεσιμπέλ (βάση 10), ή οτιδήποτε άλλο:

```
sub log_base {
    my ($base, $n) = @_;
    return log($n) / log($base);
}

print log_base(2, 1024), "\n";      # 10
print log_base(16, 65536), "\n";    # 4
```

Λειτουργία επί της `$_` - βολικό μέσα σε `map` ή σε βρόχο γραμμών `while`:

```
my @natural_logs = map { log } 1, 2, 3, 4;
```

Αριθμητικός εξαναγκασμός ορίσματος συμβολοσειράς - η `log` αποτιμά τον τελεστή της σε αριθμητικό περιβάλλον όπως ακριβώς και οι αριθμητικοί τελεστές:

```
print log("7.389056"), "\n";        # ~2 (log of e^2, give or take)
```

Οριακές περιπτώσεις

- **Το `log(0)` είναι `-Inf`**, όχι σφάλμα και όχι εξαίρεση διαίρεσης με το μηδέν. Η Perl δεν πεθαίνει· λαμβάνετε άπειρο σε float και κάθε επόμενη αριθμητική το διαδίδει:

  ```
  my $x = log(0);                   # -Inf
  print $x + 1, "\n";               # -Inf
  ```

- **Η `log` αρνητικού αριθμού είναι `NaN`**. Και πάλι, καμία εξαίρεση - ο καλών είναι υπεύθυνος για τον έλεγχο εύρους αν αυτό έχει σημασία:

```
my $x = log(-1);                    # NaN
print $x == $x ? "num" : "nan", "\n";   # nan (NaN != NaN)
```

Προστατεύστε την είσοδο αν χρειάζεστε ορισμένο αποτέλεσμα:

```
die "log domain error" unless $n > 0;
my $l = log($n);
```

- **Μη αριθμητικές συμβολοσειρές** εξαναγκάζονται σε `0` και συνεπώς παράγουν `-Inf`, και υπό `use warnings` ενεργοποιούν `Argument "..." isn't numeric`:

```
use warnings;
print log("hello"), "\n";          # -Inf, with warning
```

- **Η μορφή χωρίς όρισμα χρησιμοποιεί την `$_`**. Η `log` με κενή λίστα ορισμάτων διαβάζει το τρέχον θέμα· μέσα σε `map { log }` ή `while (<>) { print log; }` αυτή είναι η συνηθισμένη συντόμευση.
- **Ακέραιη έναντι float εισόδου** δεν κάνει διαφορά στον τύπο αποτελέσματος: η τιμή επιστροφής είναι πάντα float. Το `log(2)` και το `log(2.0)` είναι η ίδια τιμή.
- **Precision** is whatever the platform C library's `log(3)` delivers
  - τυπικά ~15–17 σημαντικά δεκαδικά ψηφία σε x86-64 Linux. Τα αποτελέσματα δεν είναι φορητά bit-προς-bit μεταξύ αρχιτεκτονικών.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `exp` - η αντίστροφη: το `exp(log($x))` είναι `$x` για `$x > 0`, και το `log(exp($x))` είναι `$x` για κάθε πεπερασμένο `$x`
- `**` - ο τελεστής ύψωσης σε δύναμη· το `$x ** $y` είναι ισοδύναμο με `exp($y * log($x))` για θετικό `$x`
- `POSIX::log10` - λογάριθμος με βάση 10, απευθείας από τη C library· ταχύτερος και ελαφρώς ακριβέστερος από το `log($x)/log(10)`
- `POSIX::log2` - λογάριθμος με βάση 2, ίδια ιστορία
- `POSIX::log1p` - το `log(1 + $x)` υπολογισμένο με ακρίβεια για μικρά `$x`, εκεί όπου η αφελής μορφή χάνει ακρίβεια

Filehandles, αρχεία, κατάλογοι

lstat

Επιστρέφει τη λίστα κατάστασης 13 στοιχείων για μια διαδρομή **χωρίς** να ακολουθήσει συμβολικό σύνδεσμο.

Η `lstat` κάνει ακριβώς ό,τι η `stat` - ίδιο αποτέλεσμα 13 στοιχείων, ίδια ειδική προσωρινή μνήμη filehandle `_`, ίδια σημασιολογία αποτυχίας - με μία διαφορά: αν το

όρισμα κατονομάζει συμβολικό σύνδεσμο, η `lstat` επιστρέφει πληροφορίες για τον *ίδιο τον σύνδεσμο* (το δικό του mode, ιδιοκτήτη, μέγεθος, mtime), όχι για τον στόχο στον οποίο δείχνει ο σύνδεσμος. Για οτιδήποτε δεν είναι συμβολικός σύνδεσμος, η `lstat` και η `stat` είναι αδιάκριτες. Σε συστήματα χωρίς συμβολικούς συνδέσμους η `lstat` υποβαθμίζεται σιωπηρά στη `stat`.

Σύνοψη

```
lstat FILEHANDLE
lstat EXPR
lstat DIRHANDLE
lstat
```

Τι επιστρέφεται

Σε περιβάλλον λίστας, η ίδια λίστα 13 στοιχείων με τη `stat`:

```
my ($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,
    $atime,$mtime,$ctime,$blksize,$blocks)
       = lstat($path);
```

Οι σημασίες των πεδίων είναι ίδιες με αυτές της `stat` - συσκευή, inode, mode (τύπος + δικαιώματα), αριθμός συνδέσμων, uid/gid ιδιοκτήτη, rdev, μέγεθος, atime, mtime, ctime, προτιμώμενο μέγεθος μπλοκ I/O και κατανεμημένα μπλοκ. Τα bits τύπου στο `$mode` είναι αυτά που ξεχωρίζουν έναν συμβολικό σύνδεσμο: στον ίδιο τον σύνδεσμο αναφέρουν `S_IFLNK`, ενώ η `stat` στο ίδιο όνομα θα ανέφερε τον τύπο του *στόχου*.

Σε βαθμωτό περιβάλλον η `lstat` επιστρέφει αληθή τιμή σε επιτυχία και κενή συμβολοσειρά σε αποτυχία. Σε αποτυχία η μορφή λίστας επιστρέφει την κενή λίστα και η `$!` τίθεται στον λόγο (`ENOENT`, `EACCES`, κ.λπ.).

Αν η `EXPR` παραλείπεται, η `lstat` λειτουργεί επί της `$_`.

Το filehandle _

Κάθε επιτυχημένη `lstat` (και κάθε επιτυχημένη `stat` ή filetest) γεμίζει τον εσωτερικό «ενταμιευτή τελευταίας stat» του διερμηνέα. Το πέρασμα του bareword `_` ως όρισμα επαναχρησιμοποιεί αυτόν τον ενταμιευτή αντί να γίνει άλλη κλήση συστήματος:

```
if (-l $path && (my @s = lstat(_))) {
    # $path was a symlink; @s describes the link, not the target
}
```

Επειδή το ίδιο το `-l` εκτελεί `lstat` εσωτερικά, το παραπάνω `lstat(_)` δεν κοστίζει τίποτα επιπλέον. Αυτός είναι ο κανονικός τρόπος να «κάνετε αρκετές ερωτήσεις για ένα αρχείο χωρίς να ξανακάνετε stat». Σημειώστε ότι ένα *filetest* σε όνομα αρχείου bareword (`-l $path`) αφήνει τον ενταμιευτή να περιγράφει τον σύνδεσμο· ένα filetest σε filehandle ακολουθεί τον ανοιχτό περιγραφέα και έτσι αφήνει ενταμιευτή σχήματος `stat`. Όταν θέλετε δεδομένα symlink, καταφύγετε ρητά στην `lstat`.

Καθολική κατάσταση που επηρεάζει

- `$_` - το όρισμα όταν η EXPR παραλείπεται.
- `$!` - τίθεται σε αποτυχία στο `errno` από την υποκείμενη κλήση `lstat(2)`.
- Το ειδικό filehandle _ - επικαλύπτεται σε κάθε επιτυχημένη κλήση.

Παραδείγματα

Ξεχωρίστε ένα symlink από τον στόχο του. Η `stat` σε symlink το ακολουθεί σιωπηρά· η `lstat` όχι:

```
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")
```

Διασχίστε έναν κατάλογο και διαχωρίστε συνδέσμους από κανονικά αρχεία χωρίς δεύτερη κλήση συστήματος ανά εγγραφή, χρησιμοποιώντας τον ενταμιευτή _:

```
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 _) {
        print "file: $name\n";
    } elsif (-d _) {
        print "dir:  $name\n";
    }
}
```

Εντοπίστε αιωρούμενο symlink - η `lstat` πετυχαίνει στον σύνδεσμο, η `stat` αποτυγχάνει επειδή ο στόχος δεν υπάρχει:

```
if (lstat $path and not stat $path) {
    warn "$path is a dangling symlink: $!\n";
}
```

Ακολουθήστε χειροκίνητα τον σύνδεσμο όταν θέλετε και τα δύο κομμάτια πληροφορίας:

```
my @link   = lstat $path       or die "lstat $path: $!";
my $target = readlink $path    // die "readlink $path: $!";
my @target = stat $path        or die "stat $path: $!";
```

Μορφή χωρίς όρισμα που λειτουργεί επί της `$_`, ιδιωματική σε μπλοκ preprocess της `File::Find` και σε φίλτρα `grep`:

```
my @symlinks = grep { lstat; -l _ } @paths;
```

Οριακές περιπτώσεις

- **Δεν είναι symlink**: η `lstat` συμπεριφέρεται ακριβώς όπως η `stat` - δεν υπάρχει ποινή στο να προτιμάτε την `lstat` όταν πρόκειται να διακλαδωθείτε με `-l`.
- **Συστήματα χωρίς symlinks**: η `lstat` καταφεύγει σιωπηρά στη `stat`. Κώδικας γραμμένος έναντι της `lstat` παραμένει φορητός· απλώς χάνει τη διακριτική συμπεριφορά.
- **Όρισμα filehandle / dirhandle**: ένα FILEHANDLE ή DIRHANDLE που περνιέται αναφέρεται σε ήδη ανοιγμένο περιγραφέα, που δεν μπορεί ο ίδιος να είναι symlink (ο πυρήνας επίλυσε τον σύνδεσμο τη στιγμή της `open`). Η `lstat FILEHANDLE` συνεπώς επιστρέφει το ίδιο αποτέλεσμα με τη `stat` σε αυτό το handle. Χρησιμοποιήστε τη μορφή EXPR στη *διαδρομή* όταν χρειάζεστε τα μεταδεδομένα του συνδέσμου.
- **Η `lstat(_)` μετά από αποτυχημένη `lstat`**: ο ενταμιευτής _ αφήνεται σε όποια κατάσταση τον άφησε η τελευταία *επιτυχημένη* stat. Μια αποτυχημένη `lstat` δεν τον εκκαθαρίζει. Πάντα ελέγχετε την τιμή επιστροφής πριν διαβάσετε το _.
- **Σχετικές διαδρομές**: επιλύονται ως προς τον τρέχοντα κατάλογο εργασίας της διεργασίας, όπως και στη `stat`.
- **Δικαίωμα για `lstat`**: απαιτεί δικαίωμα αναζήτησης (x) σε κάθε συστατικό καταλόγου της διαδρομής, αλλά - σε αντίθεση με τη `stat` - **δεν** απαιτεί κανένα δικαίωμα στον στόχο του συνδέσμου. Η `lstat` συνεπώς πετυχαίνει σε συνδέσμους προς καταλόγους που δεν μπορείτε να διαβάσετε.
- **Bareword έναντι έκφρασης**: η `lstat $fh` όπου το `$fh` κρατάει filehandle κάνει stat στο handle· η `lstat "$fh"` μετατρέπει σε συμβολοσειρά το `$fh` και κάνει stat στην προκύπτουσα (συνήθως άνευ νοήματος) διαδρομή. Βάλτε εισαγωγικά σκόπιμα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `stat` - ο αντίστοιχος που ακολουθεί τους συνδέσμους· ίδιο σχήμα επιστροφής και σημασιολογία _
- `readlink` - διαβάστε τη συμβολοσειρά *στόχου* ενός συμβολικού συνδέσμου χωρίς να του κάνετε stat
- `symlink` - δημιουργία συμβολικού συνδέσμου
- `-l` - τελεστής filetest για το «είναι συμβολικός σύνδεσμος»· εσωτερικά εκτελεί `lstat` και αφήνει το αποτέλεσμα στο _
- `$_` - προεπιλεγμένο όρισμα όταν η EXPR παραλείπεται
- `$!` - λόγος αποτυχίας όταν η `lstat` επιστρέφει την κενή λίστα

Κανονικές εκφράσεις και αντιστοίχιση μοτίβων

m//

Αναζητά σε μια συμβολοσειρά ένα μοτίβο και αναφέρει αν - και τι - ταίριαξε.

Το `m//` είναι ο τελεστής αντιστοίχισης. Μεταγλωττίζει το PATTERN ως κανονική έκφραση (δείτε τον *οδηγό regex*), το εκτελεί έναντι μιας συμβολοσειράς-στόχου, και επιστρέφει μια τιμή με σχήμα που καθορίζεται από το περιβάλλον κλήσης και τους τροποποιητές που εφαρμόζετε. Στόχος είναι ό,τι βρίσκεται αριστερά του `=~` ή του `!~`· χωρίς τελεστή σύνδεσης, ο στόχος είναι η `$_`. Το αρχικό `m` είναι προαιρετικό όταν ο οριοθέτης είναι `/`, οπότε τα `/PATTERN/` και `m/PATTERN/` σημαίνουν το ίδιο πράγμα.

Σύνοψη

```
$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
```

Τι επιστρέφεται

Το περιβάλλον αποφασίζει το σχήμα της τιμής επιστροφής.

- **Βαθμωτό περιβάλλον, χωρίς `/g`**: 1 σε αντιστοιχία, η κενή συμβολοσειρά σε μη αντιστοιχία. Και οι δύο χρησιμοποιούνται ως λογικές τιμές· η κενή συμβολοσειρά είναι ψευδής με διπλή τιμή που ισούται επίσης αριθμητικά με `0`.
- **Περιβάλλον λίστας, χωρίς `/g`**: η λίστα τιμών σύλληψης (`$1, $2, $3, …`) σε επιτυχημένη αντιστοιχία· αν το μοτίβο δεν έχει ομάδες σύλληψης, η μονοσύνολη `(1)`· σε αποτυχία, η κενή λίστα. Έτσι λειτουργεί το `if (my ($x, $y) = $s =~ /(\w+)=(\w+)/)`.
- **Βαθμωτό περιβάλλον, `/g`**: κάθε κλήση προχωρά μέσα στη συμβολοσειρά, επιστρέφοντας αληθές για την επόμενη αντιστοιχία και ψευδές μόλις δεν υπάρχουν άλλες. Η `pos` στον στόχο παρακολουθεί από πού θα ξεκινήσει η επόμενη απόπειρα.
- **Περιβάλλον λίστας, `/g`**: όλες οι αντιστοιχίες με μία κίνηση. Με ομάδες σύλληψης, η επίπεδη λίστα όλων των συλλήψεων από κάθε αντιστοιχία. Χωρίς συλλήψεις, η λίστα κάθε πλήρους αντιστοιχίας.

Οι επιτυχημένες αντιστοιχίες γεμίζουν επίσης τις ειδικές μεταβλητές regex (`$1` έως `$9`, `$&`, `$``, `$'`, `$+`, `%+`, `%-`) για την περικλείουσα δυναμική εμβέλεια. Μια αποτυχημένη αντιστοιχία τις αφήνει να κρατούν τις προηγούμενες τιμές τους - ελέγχετε πάντα την ίδια την αντιστοιχία, ποτέ μια μεταβλητή σύλληψης, για να αποφασίσετε αν συνέβη αντιστοιχία.

Καθολική κατάσταση που επηρεάζει

- `$_` - ο προεπιλεγμένος στόχος όταν δεν δίνεται σύνδεση `=~`.
- `$1`, `$2`, ... - αριθμημένες συλλήψεις, τίθενται σε επιτυχία, αμετάβλητες σε αποτυχία.
- `$&`, `$`` , `$'` - αντιστοιχία, προ-αντιστοιχία, μετα-αντιστοιχία.
- `$+` - η σύλληψη με τον μεγαλύτερο αριθμό που πράγματι ταίριαξε (χρήσιμη με εναλλαγές).
- `%+`, `%-` - hashes ονομαστικών συλλήψεων.
- `${^LAST_SUCCESSFUL_PATTERN}` - το τελευταίο μοτίβο που ταίριαξε στην τρέχουσα δυναμική εμβέλεια· επίσης το μοτίβο που η κενή μορφή `m//` επαναχρησιμοποιεί (δείτε *Οριακές περιπτώσεις*).
- `pos` στη συμβολοσειρά-στόχο - διαβάζεται και ενημερώνεται από αντιστοίχιση `/g`· επανέρχεται σε αποτυχία εκτός αν τεθεί επίσης το `/c`.
- Πηγές κανόνων locale / Unicode όταν ισχύουν οι `/l`, `/u` ή `/d`.

Οριοθέτες

Με `m`, οποιοδήποτε ζεύγος μη-λευκών χαρακτήρων λειτουργεί ως οριοθέτης, και τα ζεύγη αγκυλών εμφωλεύονται:

```
m/pattern/
m{pattern}
m[pattern]
m(pattern)
m<pattern>
m!pattern!
m#pattern#
m,pattern,
```

Επιλέγοντας οριοθέτη που δεν εμφανίζεται στο μοτίβο αποφεύγετε τη συσσώρευση backslash - γνωστή ως LTS, *leaning toothpick syndrome* (σύνδρομο γερτής οδοντογλυφίδας). Ένα μοτίβο αντιστοίχισης διαδρομών διαβάζεται καθαρά με `m{…}` ή `m!…!` και άσχημα με `m/…/`.

Δύο επιλογές οριοθέτη αλλάζουν τη σημασιολογία:

- **`'` (μονό εισαγωγικό)** - καμία παρεμβολή μεταβλητών μέσα στο `PATTERN`. Το `m'$foo'` ταιριάζει τους κυριολεκτικούς τέσσερις χαρακτήρες.
- **`?`** - το `m?PATTERN?` ταιριάζει μόνο μία φορά μεταξύ κλήσεων στην `reset`. Το αρχικό `m` είναι υποχρεωτικό· από την Perl 5.22 και μετά, η γυμνή μορφή `?…?` είναι συντακτικό σφάλμα.

Όταν ο οριοθέτης είναι χαρακτήρας λέξης (γράμμα ή ψηφίο), απαιτείται κενό μετά το `m`: το `m q foo q` είναι νόμιμο, το `mqfooq` δεν είναι.

Τροποποιητές

Τροποποιητές μεταγλώττισης μοτίβου (γίνονται επίσης δεκτοί από `qr`, `s` και `split`):

- **`m`** - πολυγραμμικός: τα ^ και $ ταιριάζουν σε κάθε ενσωματωμένο newline, όχι μόνο στα άκρα της συμβολοσειράς.
- **`s`** - μονογραμμικός: το . ταιριάζει κάθε χαρακτήρα συμπεριλαμβανομένου του newline.
- **`i`** - αντιστοίχιση χωρίς διάκριση πεζών-κεφαλαίων.
- **`x`** - αγνοεί λευκούς χαρακτήρες και σχόλια # στο μοτίβο· το `xx` το επεκτείνει και στις κλάσεις χαρακτήρων.
- **`p`** - διατηρεί αντίγραφα της αντιστοιχισμένης συμβολοσειράς. Από την 5.20 είναι no-op - οι `${^PREMATCH}`, `${^MATCH}`, `${^POSTMATCH}` είναι πάντα διαθέσιμες μετά από επιτυχημένη αντιστοιχία.
- **`a`**, **`u`**, **`l`**, **`d`** - κανόνες συνόλου χαρακτήρων για τα `\d`, `\s`, `\w`, και τις κλάσεις POSIX. Το `/a` τους περιορίζει σε ASCII· το `/aa` επιπρόσθετα απαγορεύει αντιστοίχιση ASCII/μη-ASCII υπό `/i`.
- **`n`** - μη συλλαμβάνων: το `(...)` συμπεριφέρεται σαν `(?:...)` και δεν γεμίζει τις $1, $2, ...
- **`o`** - μεταγλωττίζει το μοτίβο ακριβώς μία φορά ακόμη και αν αλλάξουν οι παρεμβαλλόμενες μεταβλητές. Σχεδόν πάντα λάθος εργαλείο· χρησιμοποιήστε αντ" αυτού `qr` για να φτιάξετε ένα επαναχρησιμοποιήσιμο μεταγλωττισμένο μοτίβο.

Τροποποιητές διαδικασίας αντιστοίχισης (αποκλειστικοί στα `m//` και `s///`):

- **`g`** - καθολική αντιστοίχιση. Σε βαθμωτό περιβάλλον η συμπεριφορά είναι επαναληπτική (προχωρά την `pos` σε κάθε κλήση)· σε περιβάλλον λίστας η συμπεριφορά επιστρέφει όλες τις αντιστοιχίες μαζί.
- **`c`** - έχει νόημα μόνο με `/g`. Μια αποτυχημένη αντιστοιχία `/g` διατηρεί την `pos` εκεί που ήταν αντί να την επαναφέρει στην αρχή· απαραίτητο για σαρωτές τύπου `lex` χτισμένους γύρω από το `\G`.

Παραδείγματα

Έλεγχος αν μια συμβολοσειρά περιέχει ένα μοτίβο:

```
if ($line =~ /error/i) {
    warn "matched: $line";
}
```

Δέσμευση σύλληψης σε μία κίνηση:

```
if (my ($key, $val) = $line =~ /^(\w+)\s*=\s*(.*)$/) {
    $config{$key} = $val;
}
```

Εξαγωγή κάθε αριθμού από μια συμβολοσειρά με `/g` σε περιβάλλον λίστας:

```
my @nums = "x=1 y=22 z=333" =~ /(\d+)/g;
# @nums = (1, 22, 333)
```

Επανάληψη μία προς μία στις αντιστοιχίες με /g σε βαθμωτό περιβάλλον, χρησιμοποιώντας την pos για να δείτε πού βρίσκεται η μηχανή:

```
my $s = "foo 1 bar 22 baz 333";
while ($s =~ /(\d+)/g) {
    printf "matched %s at offset %d\n", $1, pos($s) - length($1);
}
```

Εκτεταμένη μορφή με τον τροποποιητή x και ονομαστικές συλλήψεις:

```
if ($ts =~ m{
        ^ (?<year>\d{4}) -
          (?<mon> \d{2}) -
          (?<day> \d{2}) $
    }x) {
    printf "year=%s mon=%s day=%s\n", $+{year}, $+{mon}, $+{day};
}
```

Αποφύγετε το LTS επιλέγοντας οριοθέτη που δεν εμφανίζεται στο μοτίβο:

```
next if $path =~ m{^/usr/local/};
```

Χρησιμοποιήστε το \G με m//gc για να διασχίζετε μια συμβολοσειρά token-προς-token χωρίς να χάνετε τη θέση σε αποτυχημένη διακλάδωση:

```
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                    }
}
```

Οριακές περιπτώσεις

- **Κενό μοτίβο**: τα // και m// επαναχρησιμοποιούν το τελευταίο μοτίβο που ταίριαξε επιτυχώς στην τρέχουσα δυναμική εμβέλεια. Αν δεν έχει ταιριάξει τίποτα ακόμη, ένα κενό μοτίβο ταιριάζει παντού. Η απευθείας διαβίβαση εισόδου χρήστη στο m/$pat/ όταν η $pat μπορεί να είναι κενή είναι αιχμηρό άκρο - τυλίξτε το σε μη συλλαμβάνουσα ομάδα: m/(?:$pat)/. Το τελευταίο επιτυχημένο μοτίβο είναι επίσης αναγνώσιμο ως ${^LAST_SUCCESSFUL_PATTERN}.
- **Αμφισημία defined-or**: Η Perl αναλύει το $x // $y ως τον τελεστή defined-or, ποτέ ως δύο κενές αντιστοιχίες. Σε παθολογικές θέσεις (print $fh //) η Perl εξακολουθεί να υποθέτει defined-or· εξαναγκάστε αντιστοιχία γράφοντας ρητά m// ή χωρίζοντας με κενά τους οριοθέτες.
- **Αποτυχημένη αντιστοιχία αφήνει τις συλλήψεις παλιωμένες**: η $1 μετά από μια αποτυχημένη /…/ εξακολουθεί να κρατά τη σύλληψη από την *προηγούμενη*

επιτυχημένη αντιστοιχία - βάζετε πάντα φραγμό στη χρήση των συλλήψεων με βάση το αποτέλεσμα της αντιστοιχίας.

- **`/o` με μεταβαλλόμενες μεταβλητές**: το `m/$x/o` κλειδώνει την πρώτη τιμή της `$x` στο μεταγλωττισμένο μοτίβο. Μεταγενέστερες αλλαγές στην `$x` αγνοούνται σιωπηλά. Καταφύγετε στην `qr` όταν θέλετε ρητή, επαναχρησιμοποιήσιμη μεταγλώττιση.
- **Παρεμβολή όταν ο οριοθέτης είναι `'`**: το `m'$var'` είναι κυριολεκτική αντιστοίχιση δολαρίου-var. Αυτό σπάνια είναι αυτό που θέλετε, και το ίδιο αποτέλεσμα είναι διαθέσιμο με `\Q…\E` ή `quotemeta` σε οποιονδήποτε άλλο οριοθέτη.
- **`/g` συν τροποποίηση στόχου**: η τροποποίηση του στόχου μεταξύ επαναλήψεων `/g` επαναφέρει την `pos` στην αρχή. Επαναλάβετε πάνω σε αντίγραφο αν χρειάζεται να μεταβάλλετε το πρωτότυπο καθώς προχωράτε.
- **`\G` εκτός `/g`**: χωρίς `/g`, το `\G` αγκυρώνει στην `pos` που είχε ο στόχος τη στιγμή της κλήσης και ταιριάζει το πολύ μία φορά. Σε συμβολοσειρά στην οποία δεν έχει ποτέ εφαρμοστεί `/g`, το `\G` ισοδυναμεί με `\A`.
- **Εμβέλεια reset του `m?…?`**: η `reset` καθαρίζει την κατάσταση `m??` μόνο για το τρέχον πακέτο. Ένα `m??` σε ένα πακέτο δεν επηρεάζεται από `reset` που κλήθηκε από άλλο.
- **Τελεστές σύγκρισης κοντά σε κενό regex**: το `$x //= 1` είναι πάντα η ανάθεση defined-or· αν θέλετε πραγματικά το κενό regex, γράψτε `m//` αντί για `//`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `qr` - μεταγλωττίζει ένα μοτίβο μία φορά και το επαναχρησιμοποιεί· αποφεύγει το `/o` και διατηρεί το μοτίβο τιμή πρώτης τάξης
- `s` - ίδια σύνταξη μοτίβου, αντικαθιστά ό,τι ταιριάζει
- `tr` - μετάφραση χαρακτήρα-προς-χαρακτήρα· διαφορετικό εργαλείο με επιφανειακά παρόμοιο σχήμα
- `split` - όταν θέλετε τα κομμάτια μεταξύ των αντιστοιχιών αντί για τις ίδιες τις αντιστοιχίες
- `pos` - διαβάζει ή ορίζει τη θέση από την οποία συνεχίζει η αντιστοίχιση `/g`
- *Regular expressions guide*
  - η ίδια η γλώσσα των regex: διεκδικήσεις, κλάσεις χαρακτήρων, οπισθαναφορές, ονομαστικές συλλήψεις

Λίστες

map

Εφαρμόζει ένα μπλοκ ή μια έκφραση σε κάθε στοιχείο μιας λίστας και επιστρέφει τα ισοπεδωμένα αποτελέσματα.

Η map αποτιμά το BLOCK (ή την EXPR) μία φορά ανά στοιχείο της LIST, με το $_ ψευδωνυμοποιημένο προς εκείνο το στοιχείο, και συλλέγει ό,τι επιστρέφει κάθε αποτίμηση σε μία μοναδική επίπεδη λίστα. Επειδή κάθε αποτίμηση μπορεί να αποδώσει μηδέν, μία ή πολλές τιμές, το μήκος της εξόδου είναι ανεξάρτητο από το μήκος της εισόδου - αυτή η ισοπέδωση είναι το νόημα της map. Σε περιβάλλον λίστας η map επιστρέφει τη γενόμενη λίστα· σε βαθμωτό περιβάλλον επιστρέφει τη συνολική καταμέτρηση στοιχείων.

Σύνοψη

```
my @out = map { BLOCK } LIST;        # block form
my @out = map EXPR, LIST;            # expression form
my @out = map { f($_) } grep { ... } sort LIST;   # chained
```

Τι επιστρέφεται

Μία μοναδική επίπεδη λίστα που κατασκευάζεται με συνένωση των τιμών επιστροφής κάθε αποτίμησης, στη σειρά εισόδου:

- Επιστροφή μίας τιμής ανά είσοδο → μετασχηματισμός (ίδιο μήκος εξόδου).
- Επιστροφή λίστας ανά είσοδο → επέκταση 1-προς-N (μεγαλύτερη έξοδος).
- Επιστροφή της κενής λίστας ( ) → παράλειψη αυτής της εισόδου (μικρότερη έξοδος· ιδίωμα map-ως-φίλτρο).

Σε βαθμωτό περιβάλλον η τιμή επιστροφής είναι ο αριθμός των γενόμενων στοιχείων, όχι αναφορά και όχι λογική τιμή. Η εκχώρηση του αποτελέσματος σε κατακερματισμό είναι καλά ορισμένη: η επίπεδη λίστα καταναλώνεται ως εναλλασσόμενα ζεύγη κλειδιού/τιμής.

```
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
```

Καθολική κατάσταση

Το $_ **ψευδωνυμοποιείται** προς κάθε στοιχείο της LIST, δεν αντιγράφεται. Η τροποποίηση του $_ μέσα στο μπλοκ τροποποιεί το πηγαίο στοιχείο. Αυτό είναι περιστασιακά χρήσιμο αλλά συνήθως παγίδα - επιλέξτε έναν απλό βρόχο foreach όταν εννοείτε επιτόπια τροποποίηση.

```
my @v = (1, 2, 3);
my @r = map { $_ *= 10; $_ } @v;
# @v is now (10, 20, 30) - the source was mutated
```

Η ψευδωνυμοποίηση προς μη μεταβλητή (κυριολεκτικό, τιμή επιστροφής συνάρτησης, στοιχείο τομής κατακερματισμού που δεν υπάρχει ακόμη) και έπειτα εκχώρηση στο $_

αποτυγχάνει κατά τον χρόνο εκτέλεσης με `Modification of a read-only value`. Δείτε το `$_`.

Η `map` επίσης τοπικοποιεί το `$_` γύρω από την κλήση, οπότε ένα εξωτερικό `$_` ορατό στον καλούντα αποκαθίσταται όταν η `map` επιστρέψει.

Παραδείγματα

Μετασχηματισμός - μετατροπή κάθε λέξης σε κεφαλαία:

```
my @upper = map { uc } @words;
```

Επέκταση 1-προς-N - διαχωρισμός κάθε γραμμής στους λευκούς χαρακτήρες στα tokens της:

```
my @tokens = map { split /\s+/ } @lines;
```

Ιδίωμα φίλτρου - διατηρήστε μόνο τις αληθείς, μετασχηματισμένες τιμές:

```
my @positives = map { $_ > 0 ? $_ : () } @numbers;
```

Κατασκευή κατακερματισμού από λίστα - κάθε αποτίμηση επιστρέφει λίστα δύο στοιχείων:

```
my %seen = map { $_ => 1 } @keys;
my %idx  = map { ($keys[$_] => $_) } 0 .. $#keys;
```

Συντεθειμένη διοχέτευση - ταξινόμηση, φιλτράρισμα, μετασχηματισμός. Οι `map`, `grep` και `sort` δέχονται όλες μια `LIST` στα δεξιά, οπότε αλυσιδώνονται από δεξιά προς τα αριστερά:

```
my @top = map  { "$_->{name}: $_->{score}" }
          sort { $b->{score} <=> $a->{score} }
          grep { $_->{score} >= 50 }
          @records;
```

Ισοπέδωση δομής - κάθε είσοδος συνεισφέρει μεταβλητό αριθμό εξόδων:

```
my @leaves = map { ref($_) eq 'ARRAY' ? @$_ : $_ } @mixed;
```

Ζευγαρώστε με δείκτη χρησιμοποιώντας το `List::Util::pairs` ή ρητή μεταβλητή δείκτη - η ίδια η `map` δεν εκθέτει τέτοια:

```
my @numbered = do {
    my $i = 0;
    map { sprintf "%d. %s", ++$i, $_ } @items;
};
```

Οριακές περιπτώσεις

- **Το `$_` ψευδωνυμοποιείται, δεν αντιγράφεται.** Η εκχώρηση στο `$_` γράφει πίσω στην πηγή. Αν το πηγαίο στοιχείο είναι μόνο για ανάγνωση (σταθερά, τιμή επιστροφής κλήσης συνάρτησης ισοπεδωμένη στη LIST), η εκχώρηση εγείρει `Modification of a read-only value`. Αντιμετωπίστε το `$_` ως μόνο εισόδου εκτός αν θέλετε συγκεκριμένα την παρενέργεια.
- **Η `map` δεν είναι βρόχος.** Οι `last`, `next`, `redo` και οι ετικέτες βρόχου δεν εφαρμόζονται στη `map`. Αναφέρονται στον περικλείοντα πραγματικό βρόχο και θα συμπεριφερθούν αναλόγως - συνήθως όχι αυτό που εννοούσε ο συγγραφέας. Χρησιμοποιήστε `foreach` όταν χρειάζεστε ροή ελέγχου βρόχου.
- **Το κενό περιβάλλον είναι παρωχημένο.** Η `map { ... } LIST;` με απορριμμένο αποτέλεσμα εκπέμπει προειδοποίηση `Useless use of map in void context` υπό `use warnings`. Όταν θέλετε τις παρενέργειες, όχι τη λίστα, γράψτε `foreach`.
- **Αμφισημία αναλυτή `BLOCK` έναντι `EXPR`.** Το `{` εκκινεί τόσο μπλοκ όσο και κατασκευαστή αναφοράς κατακερματισμού. Τα `map { ... } LIST` και `map { ... }, LIST` φαίνονται πανομοιότυπα στον αναλυτή στην ανοιγόμενη αγκύλη· η Perl μαντεύει βάσει του τι ακολουθεί. Ένα μπλοκ που ξεκινά με κάτι που μοιάζει με κλειδί κατακερματισμού συν `=>` είναι ιδιαίτερα πιθανό να εικαστεί λάθος:

```
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
```

  Για να επιστρέψετε ανώνυμο κατακερματισμό ανά στοιχείο, εξαναγκάστε τον κατασκευαστή αναφοράς κατακερματισμού με `+{`:

```
my @hashes = map +{ lc($_) => 1 }, @array;
```

- **Η σειρά είναι σταθερή.** Η `map` αποτιμά από αριστερά προς τα δεξιά και συνενώνει αποτελέσματα από αριστερά προς τα δεξιά. Η γενόμενη λίστα διατηρεί τη σειρά εισόδου, με τη συνεισφορά κάθε εισόδου να εμφανίζεται συνεχόμενα.
- **Κενή `LIST`.** Η `map` αποτιμά το μπλοκ μηδέν φορές και επιστρέφει την κενή λίστα. Το μπλοκ δεν αποτιμάται για τις παρενέργειές του.
- **Εμφωλευμένη `map`.** Η εσωτερική `map` βλέπει το δικό της `$_`· το εξωτερικό `$_` επικαλύπτεται για όλη τη διάρκεια της εσωτερικής αποτίμησης. Αποτυπώστε πρώτα την εξωτερική τιμή αν χρειάζονται και οι δύο:

```
my @pairs = map {
    my $outer = $_;
    map { [$outer, $_] } @inner
} @outer;
```

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `grep` - ίδιου σχήματος με τη `map`, αλλά διατηρεί στοιχεία για τα οποία το μπλοκ είναι αληθές αντί να τα μετασχηματίζει· αλυσιδώνεται φυσικά με `map` και `sort`
- `sort` - διατάσσει μια λίστα· συνήθως συντίθεται με `map` μέσω του Schwartzian Transform (`map` → `sort` → `map`)
- `for` / `foreach` - χρησιμοποιήστε όταν θέλετε επανάληψη για παρενέργειες, τροποποίηση της πηγής, ή έλεγχο βρόχου (`last`/`next`/`redo`)
- x - τελεστής επανάληψης λίστας· ορθογώνιος με τη `map` αλλά χρησιμοποιείται συχνά για να κατασκευάσει τη `LIST` που καταναλώνει η `map`
- `List::Util::reduce` - αναδιπλώστε μια λίστα σε μία τιμή· χρησιμοποιήστε όταν το αποτέλεσμα δεν είναι λίστα αλλά συγκεντρωτική τιμή (άθροισμα, μέγιστο, πρώτο ταίριασμα)
- `$_` - το ψευδώνυμο στοιχείου μέσα στο μπλοκ· μόνο για ανάγνωση όταν το πηγαίο στοιχείο είναι κυριολεκτικό ή τιμή επιστροφής συνάρτησης

Ροή ελέγχου · Κλάσεις και αντικειμενοστρέφεια

method

Δηλώνει μια ονομασμένη μέθοδο στιγμιότυπου μέσα σε ένα μπλοκ `class`.

Η `method` ορίζει υπορουτίνα που αναμένει να κληθεί σε αντικείμενο της περικλείουσας κλάσης. Συμπεριφέρεται σαν `sub`, με δύο πρόσθετες εγγυήσεις που η `sub` δεν σας δίνει: το λεξιλογικό `$self` δένεται αυτόματα στον επικαλούμενο, και οι υπογραφές είναι πάντα ενεργές - γράφετε τις παραμέτρους σαν να ίσχυε `use feature 'signatures'`, και το `$self` δεν εμφανίζεται σε αυτή την υπογραφή.

Η `method` είναι έγκυρη μόνο μέσα σε δήλωση `class`. Εκτός μπλοκ κλάσης αποτελεί σφάλμα κατά τη μεταγλώττιση.

> **Σημείωση**
>
> Πειραματικό χαρακτηριστικό Οι δεσμευμένες λέξεις `class`, `method` και `field` είναι μέρος του πειραματικού χαρακτηριστικού `class` (εισάχθηκε στην Perl 5.38). Η ενεργοποίησή του με `use feature 'class';` εκπέμπει προειδοποίηση στην κατηγορία `experimental::class`. Η σύνταξη και σημασιολογία ενδέχεται ακόμη να αλλάξουν μεταξύ εκδόσεων της Perl.

Σύνοψη

```
method NAME BLOCK
method NAME SIGNATURE BLOCK
method NAME : ATTRS BLOCK
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
method SIGNATURE BLOCK        # anonymous
method BLOCK                  # anonymous
```

Τι επιστρέφεται

Η `method` είναι δήλωση, όχι έκφραση, στην ονομασμένη μορφή της - εγκαθιστά τη μέθοδο στον πίνακα μεθόδων της περικλείουσας κλάσης και αποτιμάται σε τίποτα χρήσιμο στη θέση εντολής. Η ανώνυμη μορφή επιστρέφει αναφορά κώδικα που θυμάται το περιβάλλον κλάσης και μπορεί να κληθεί αργότερα με επικαλούμενο αντικειμένου.

Το σώμα μιας `method` καλείται με το αντικείμενο ως επικαλούμενο. Η τιμή επιστροφής της μεθόδου είναι ό,τι επιστρέφει το μπλοκ, ακολουθώντας τους ίδιους κανόνες περιβάλλοντος με μια κανονική κλήση υπορουτίνας.

Τι σας δίνει η `method` που δεν σας δίνει η `sub`

Τέσσερα πράγματα διαφέρουν από τη γραφή `sub` μέσα σε μπλοκ `class`:

- **Το `$self` είναι προδηλωμένο.** Ένα λεξιλογικό `$self` δημιουργείται στην εμβέλεια της μεθόδου και αρχικοποιείται στον επικαλούμενο. Δεν - και δεν θα έπρεπε να - το αφαιρείτε με shift από το `@_`.
- **Οι υπογραφές είναι πάντα ενεργές.** Μπορείτε να γράψετε `method greet($name) { ... }` χωρίς `use feature 'signatures'`. Ο επικαλούμενος `$self` καταναλώνεται πριν δέσει η υπογραφή, οπότε δεν εμφανίζεται ποτέ στη λίστα παραμέτρων.
- **Τα πεδία είναι σε εμβέλεια.** Όλες οι μεταβλητές `field` που δηλώνονται στην περικλείουσα κλάση είναι ορατές με τα λεξιλογικά τους ονόματα μέσα στο σώμα της μεθόδου. Μια απλή `sub` μέσα σε μπλοκ `class` δεν βλέπει πεδία.
- **Καταχωρείται ως μέθοδος.** Ο πίνακας μεθόδων της κλάσης καταγράφει το όνομα, το οποίο αναζητά ο μηχανισμός επίλυσης μεθόδων για αποστολή `$obj->name(...)`. Μια απλή `sub` μέσα στο μπλοκ κλάσης είναι βοηθός, όχι μέθοδος.

Με άλλα λόγια: μια `sub` μέσα σε μπλοκ `class` είναι συνηθισμένη υπορουτίνα πακέτου που τυχαίνει να ζει στον χώρο ονομάτων εκείνου του πακέτου. Δεν έχει `$self`, δεν έχει ορατότητα πεδίων, και δεν αποτελεί μέρος της διεπαφής μεθόδων της κλάσης - καλέστε την μέσω `ClassName::helper(...)` αν το χρειάζεστε, όχι μέσω βέλους.

Παραδείγματα

Μια ελάχιστη κλάση με μέθοδο και πεδίο:

```
use v5.38;
use feature 'class';
no warnings 'experimental::class';

class Greeter {
    field $greeting = "Hello";

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    method greet($name) {
        return "$greeting, $name";
    }
}

my $g = Greeter->new;
say $g->greet("world");           # Hello, world
```

Οι προεπιλεγμένες τιμές υπογραφής λειτουργούν ακριβώς όπως για την sub:

```
class Greeter {
    field $greeting = "Hello";

    method greet($name = "someone") {
        return "$greeting, $name";
    }
}

say Greeter->new->greet;          # Hello, someone
```

Μια μέθοδος χωρίς παραμέτρους εξακολουθεί να χρησιμοποιεί τη μορφή υπογραφής - οι κενές `()` δεν είναι απαιτητές, αλλά επιτρέπονται και αποτελούν το σύνηθες στιλ όταν η μέθοδος δέχεται ορίσματα αλλού στην κλάση:

```
class Counter {
    field $n = 0;

    method inc        { $n++ }
    method value ()   { return $n }
}
```

Μια ανώνυμη μέθοδος - χρήσιμη ως εργοστάσιο ή ως callback που χρειάζεται πρόσβαση στο `$self` και στα πεδία:

```
class AnonMethodFactory {
    method make_printer {
        return method { say "called on $self" };
    }
}
```

Μια λεξιλογική (ιδιωτική) μέθοδος δηλωμένη με `my method`. Επικαλεστείτε την μέσω του τελεστή `->&`, που παρακάμπτει την αναζήτηση μεθόδου και καλεί τη λεξιλογική απευθείας σαν να ήταν μέθοδος:

```
class LexicalMethod {
    my method _check ($x, $y) {
        return $x > 0 && $y > 0;
    }

    method process ($x, $y) {
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
        return unless $self->&_check($x, $y);
        # ...
    }
}
```

Κανονική sub μέσα σε κλάση, σε αντιδιαστολή με method - η sub είναι βοηθός, η method είναι η δημόσια διεπαφή:

```
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);
    }
}
```

Οριακές περιπτώσεις

- **Εκτός μπλοκ `class` είναι σφάλμα κατά τη μεταγλώττιση.** Το `method foo { ... }` στο ανώτατο επίπεδο, ή μέσα σε `package` που δεν είναι `class`, δεν αναλύεται. Χρησιμοποιήστε εκεί την `sub`.
- **Το `$self` είναι μόνο για ανάγνωση με την έννοια ότι δεν μπορείτε να δηλώσετε άλλο λεξιλογικό με το ίδιο όνομα** στην αρχή της μεθόδου - βρίσκεται ήδη σε εμβέλεια. Η εκχώρηση στο `$self` (π.χ. `$self = ...`) επιτρέπεται αλλά σπάνια είναι χρήσιμη και δεν αλλάζει τον επικαλούμενο του καλούντος.
- **Το `@_` δεν είναι η λίστα ορισμάτων που περιμένετε.** Υπό τις υπογραφές που ενεργοποιεί η `method`, το `@_` μέσα στο σώμα είναι κενό μετά τη δέσμευση της υπογραφής, όπως ακριβώς σε κάθε sub που χρησιμοποιεί υπογραφή. Διαβάστε τις παραμέτρους από την υπογραφή, όχι από το `@_`.
- **Η σειρά επίλυσης μεθόδου ορίζεται από την περικλείουσα κλάση.** Η ίδια η `method` δεν δέχεται επιλογή MRO· η κληρονομικότητα και η αποστολή ακολουθούν ό,τι διαμόρφωσε η δήλωση `class` μέσω `:isa` και συναφών γνωρισμάτων.
- **Η `method` δεν συμμετέχει σε πρωτότυπα.** Οι μέθοδοι επικαλούνται μέσω αποστολής, όχι με άμεσες κλήσεις πίνακα συμβόλων, οπότε τα πρωτότυπα (το γνώρισμα `:prototype(...)` σε μια sub) δεν έχουν εδώ αποτέλεσμα. Χρησιμοποιήστε την υπογραφή για να περιορίσετε το σχήμα κλήσης.
- **Γνωρίσματα.** Η μορφή `method NAME : ATTRS BLOCK` δέχεται γνωρίσματα με τον ίδιο τρόπο όπως η `sub NAME : ATTRS BLOCK`. Το σύνολο των χρήσιμων γνωρισμάτων στις μεθόδους ακόμη σταθεροποιείται με το χαρακτηριστικό `class`· συμβουλευτείτε το upstream πριν βασιστείτε σε προσαρμοσμένα.
- **Παγίδα επιστροφής από μέθοδο με ανώνυμη μέθοδο.** Το `return method { ... }` επιστρέφει αναφορά κώδικα. Το `method { ... }` ως τελευταία έκφραση μεθόδου χωρίς `return` εξακολουθεί να είναι αναφορά κώδικα, αλλά οι αναγνώστες συχνά την παρερμηνεύουν ως μπλοκ - προτιμήστε ρητή `return` για σαφήνεια.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Το χαρακτηριστικό `class` - και επομένως η `method` - σημειώνεται πειραματικό στο upstream· η pperl κληρονομεί αυτή την κατάσταση και την κατηγορία προειδοποίησης `experimental::class`.

Δείτε επίσης

- `class` - δηλώνει το πακέτο μέσα στο οποίο ζει η `method`· η `method` αναλύεται μόνο μέσα σε μπλοκ `class`
- `field` - ανά στιγμιότυπο μεταβλητές που είναι ορατές με το όνομά τους μέσα σε κάθε `method` της περικλείουσας κλάσης
- `sub` - δηλώστε κανονική υπορουτίνα· χρησιμοποιήστε αυτό για βοηθούς μέσα σε κλάση που δεν χρειάζονται `$self` ή πρόσβαση σε πεδία
- `__CLASS__` - το όνομα της τρέχουσας κλάσης, χρήσιμο μέσα σε μεθόδους που χρειάζεται να αποστείλουν πίσω σε κώδικα επιπέδου κλάσης
- `return` - ρητή επιστροφή από σώμα μεθόδου· συνιστάται όταν η τελευταία έκφραση είναι η ίδια ανώνυμη μορφή `method { ... }`
- `my` - προθέστε `my` σε μια `method` για να δηλώσετε λεξιλογική (ιδιωτική) μέθοδο που επικαλείται μέσω `->&`

Filehandles, αρχεία, κατάλογοι

mkdir

Δημιουργεί έναν μόνο κατάλογο στο σύστημα αρχείων.

Η `mkdir` ζητάει από το λειτουργικό σύστημα να δημιουργήσει τον κατάλογο που κατονομάζει το `FILENAME`, με bits δικαιωμάτων που δίνει το `MODE` (περαιτέρω περιοριζόμενα από την `umask` της διεργασίας). Δημιουργεί ακριβώς **έναν** κατάλογο

- every parent component in `FILENAME` must already exist. For recursive creation, reach for `File::Path::make_path` instead.

Σύνοψη

```
mkdir FILENAME, MODE
mkdir FILENAME
mkdir
```

Τι επιστρέφεται

1 σε επιτυχία, `0` σε αποτυχία με την `$!` τεθειμένη στο errno του συστήματος. Σε αντίθεση με τις περισσότερες πρωτογενείς λειτουργίες I/O, η τιμή αποτυχίας είναι ένα απλό ψευδές `0`, όχι `undef` - ελέγξτε την τιμή επιστροφής, και μετά συμβουλευτείτε την `$!` για τον λόγο:

```
mkdir $path, 0755
    or die "mkdir $path failed: $!";
```

Προεπιλεγμένα ορίσματα

Και τα δύο ορίσματα έχουν προεπιλογές:

- Το `MODE` έχει προεπιλογή `0777`. Ο πυρήνας μασκάρει το παρεχόμενο mode με την `umask` της τρέχουσας διεργασίας, οπότε το πραγματικό mode του καταλόγου είναι `MODE & ~umask`. Μια τυπική `umask 022` μετατρέπει το προεπιλεγμένο `0777` σε `0755` στον δίσκο.
- Το `FILENAME` έχει προεπιλογή την `$_`. Μια σκέτη `mkdir;` μέσα σε βρόχο πάνω σε διαδρομές δημιουργεί καθεμία με τη σειρά:

  ```
  mkdir for qw(logs cache tmp);
  ```

Περάστε το `MODE` ρητά όποτε ο κατάλογος πρέπει να είναι ιδιωτικός (mail spools, αποθήκες κλειδιών, κατάλογοι συνεδριών). Διαφορετικά προτιμήστε ένα επιτρεπτικό `MODE` και αφήστε την `umask` του χρήστη να το στενέψει - η σελίδα της `umask` συζητάει το trade-off αναλυτικά. Η ρύθμιση bits εκτός του εύρους δικαιωμάτων (π.χ. setuid/setgid/sticky) αποδίδει συμπεριφορά καθορισμένη από την υλοποίηση κατά POSIX 1003.1-2008.

Καθολική κατάσταση που επηρεάζει

- Θέτει την `$!` (errno) σε αποτυχία· την αφήνει αμετάβλητη σε επιτυχία.
- Διαβάζει το `FILENAME` από την `$_` όταν καλείται χωρίς ορίσματα.
- Τα πραγματικά bits δικαιωμάτων εξαρτώνται από την `umask` της διεργασίας.

Παραδείγματα

Δημιουργία καταλόγου με ρητό mode:

```
mkdir "build", 0755
    or die "mkdir build: $!";
```

Δημιουργία του μόνο αν λείπει, χωρίς να αντιμετωπίζεται το «υπάρχει ήδη» ως σφάλμα:

```
use Errno qw(EEXIST);
unless (mkdir "cache", 0700) {
    die "mkdir cache: $!" unless $! == EEXIST;
}
```

Ιδιωτικός κατάλογος - μην αφήσετε την `umask` να τον διευρύνει:

```
mkdir "$ENV{HOME}/.secrets", 0700
    or die "mkdir secrets: $!";
```

Σκέτη μορφή πάνω σε λίστα, με την `$_` ως όνομα αρχείου:

```
for (qw(a b c)) {
    mkdir or warn "mkdir $_: $!";
}
```

Η αναδρομική δημιουργία **δεν** είναι δουλειά της `mkdir` - χρησιμοποιήστε την `File::Path`:

```
use File::Path qw(make_path);
make_path("var/log/app/2026", { mode => 0755 })
    or die "make_path failed: $!";
```

Οριακές περιπτώσεις

- **Ο γονέας πρέπει να υπάρχει.** Η `mkdir "a/b/c"` αποτυγχάνει με `ENOENT` αν το `a/b` λείπει. Χρησιμοποιήστε την `File::Path::make_path` για βαθιές διαδρομές.
- **Ο στόχος υπάρχει ήδη.** Αποτυγχάνει με `EEXIST`, ανεξαρτήτως του αν η υπάρχουσα εγγραφή είναι κατάλογος, αρχείο ή symlink. Ελέγξτε την `$!` έναντι της `Errno::EEXIST` όταν το «υπάρχει ήδη» είναι αποδεκτό.
- **Τελικές κάθετοι.** Το POSIX 1003.1-1996 επιτρέπει οποιονδήποτε αριθμό τελικών καθέτων· η Perl τις αφαιρεί πριν την κλήση συστήματος ώστε συστήματα αρχείων που απορρίπτουν το `"foo/"` να συμπεριφέρονται και πάλι συνεπώς.
- **Αναντιστοιχία δικαιωμάτων.** Το `MODE` μασκάρεται από την `umask`. Αν χρειάζεστε ακριβές mode στον δίσκο, είτε αποθηκεύστε και επαναφέρετε την umask, είτε κάντε `chmod` στον κατάλογο μετά τη δημιουργία:

  ```
  mkdir $dir, 0700 or die $!;
  chmod 0700, $dir;                    # defeat any umask surprises
  ```

- **Το `MODE` παραλείπεται εντελώς.** Η `mkdir $path` είναι ίδια με `mkdir $path, 0777`
  - και πάλι μασκάρεται από την `umask`, οπότε το mode στον δίσκο είναι τυπικά `0755`, όχι `0777`.
- **Bits εκτός δικαιωμάτων στο `MODE`.** Τα bits setuid / setgid / sticky σε μια κλήση `mkdir` έχουν αποτελέσματα καθορισμένα από την υλοποίηση. Ρυθμίστε τα με επακόλουθο `chmod` αν τα χρειάζεστε.
- **Σύστημα αρχείων μόνο για ανάγνωση ή απόν δικαίωμα εγγραφής στον γονέα** αποτυγχάνει με `EROFS` ή `EACCES` αντίστοιχα. Η `$!` τα διακρίνει.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `rmdir` - αφαίρεση κενού καταλόγου· η αντίστροφη λειτουργία
- `chmod` - προσαρμογή bits δικαιωμάτων μετά τη δημιουργία όταν η `umask` διαφορετικά θα τα στένευε
- `umask` - η μάσκα που σιωπηρά περιορίζει κάθε `mkdir MODE`
- `stat` - επιθεωρήστε τον τύπο και το mode μιας υπάρχουσας διαδρομής, π.χ. για να ξεχωρίσετε «ήδη κατάλογος» από «υπάρχει ως αρχείο»
- `File::Path::make_path` - αναδρομική δημιουργία καταλόγων· το σωστό εργαλείο όταν οι ενδιάμεσοι γονείς μπορεί να μην υπάρχουν

SysV IPC

msgctl

Εκτελεί μια λειτουργία ελέγχου σε ουρά μηνυμάτων IPC System V.

Η `msgctl` είναι άμεσο wrapper γύρω από την κλήση συστήματος C `msgctl(2)`. Επιθεωρεί, τροποποιεί ή αφαιρεί την ουρά μηνυμάτων που αναγνωρίζεται από το `ID`. Η λειτουργία που επιλέγεται από τη `CMD` καθορίζει πώς χρησιμοποιείται το `ARG` - είτε ως περιγραφέας εισόδου, είτε ως ενταμιευτής εξόδου, είτε αγνοείται εντελώς. Οι συμβολικές σταθερές που χρειάζονται για τη `CMD` (`IPC_STAT`, `IPC_SET`, `IPC_RMID`, και σε Linux οι `IPC_INFO`, `MSG_STAT`, `MSG_INFO`) ζουν στο `IPC::SysV`, το οποίο πρέπει να εισαγάγετε πριν επιλυθούν τα ονόματα.

Σύνοψη

```
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
```

Τι επιστρέφεται

Η τιμή επιστροφής ακολουθεί την ίδια σύμβαση με την `ioctl`:

- `undef` σε σφάλμα, με το `$!` τεθειμένο.
- Τη συμβολοσειρά `"0 but true"` όταν η υποκείμενη κλήση επέστρεψε `0`. Αυτή είναι η περίπτωση επιτυχίας με μηδενική τιμή - δοκιμάζεται ως αληθής σε λογικό περιβάλλον και συγκρίνεται ίση με `0` σε αριθμητικό περιβάλλον, οπότε ένας έλεγχος χειρίζεται τόσο επιτυχία όσο και αποτυχία.
- Την ακατέργαστη μη μηδενική ακέραια επιστροφή διαφορετικά (για παράδειγμα τον δείκτη που επιστρέφει η ειδική του Linux `MSG_STAT`).

Η σύμβαση `"0 but true"` υπάρχει ακριβώς ώστε η `msgctl(...) or die` να κάνει το σωστό πράγμα ακόμη και όταν η κλήση συστήματος επιστρέφει νόμιμα μηδέν:

```
msgctl $id, IPC_RMID, 0
    or die "msgctl IPC_RMID failed: $!";
```

Πώς χρησιμοποιείται το ARG

Το ARG παίζει τρεις διαφορετικούς ρόλους ανάλογα με τη CMD. Ο πυρήνας αγγίζει τη μνήμη μόνο στις κατευθύνσεις που απαιτεί η εντολή:

- **IPC_STAT** - το ARG πρέπει να είναι **εγγράψιμο βαθμωτό**. Ο πυρήνας το γεμίζει με την πακεταρισμένη δομή msqid_ds που περιγράφει την ουρά (δικαιώματα, ιδιοκτήτη, χρόνους τελευταίας αποστολής/τελευταίας λήψης, όρια bytes ουράς, καταμετρήσεις). Κάντε unpack με βοηθούς του IPC::SysV αντί να γράψετε τη διάταξη χειρωνακτικά· η msqid_ds ποικίλλει μεταξύ πυρήνων και εκδόσεων libc.
- **IPC_SET** - το ARG πρέπει να περιέχει πακεταρισμένη msqid_ds **που παρήχθη από προηγούμενη IPC_STAT** στην ίδια (ή συμβατή) ουρά. Τιμώνται μόνο μερικά πεδία: msg_perm.uid, msg_perm.gid, msg_perm.mode (κατώτερα 9 bits) και msg_qbytes. Τα υπόλοιπα αγνοούνται.
- **IPC_RMID** - το ARG δεν χρησιμοποιείται. Περάστε 0. Η ουρά σημειώνεται προς καταστροφή· οποιαδήποτε μπλοκαρισμένη msgsnd / msgrcv σε αυτήν αποτυγχάνει με EIDRM.

Μόνο σε Linux, οι MSG_STAT και MSG_INFO δέχονται επίσης εγγράψιμο βαθμωτό για το ARG και επιστρέφουν επιπλέον δεδομένα· αυτές δεν είναι φορητές και θα πρέπει να φραγούν πίσω από έλεγχο κατά τον χρόνο εκτέλεσης.

Καθολική κατάσταση που επηρεάζει

- $! - τίθεται σε αποτυχία με το errno του συστήματος (EPERM, EACCES, EINVAL, EIDRM, EFAULT είναι οι συνηθισμένες τιμές).
- Καμία άλλη ειδική μεταβλητή δεν επηρεάζεται.

Παραδείγματα

Διαβάστε την κεφαλίδα της ουράς και επιθεωρήστε το τρέχον όριο bytes:

```
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.
```

Συρρικνώστε τον προϋπολογισμό bytes της ουράς. Πλήρης κύκλος μέσω IPC_STAT ώστε τα αμετάβλητα πεδία να διατηρήσουν τις τρέχουσες τιμές τους από τον πυρήνα:

```
use IPC::SysV qw(IPC_STAT IPC_SET);

my $buf = "";
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
msgctl($id, IPC_STAT, $buf) or die "stat: $!";
# ... modify msg_qbytes inside $buf via IPC::SysV ...
msgctl($id, IPC_SET,  $buf) or die "set: $!";
```

Καταστρέψτε μια ουρά όταν τελειώσετε. Το ARG αγνοείται, αλλά πρέπει παρ” όλα αυτά να περαστεί - το `0` είναι ο συμβατικός placeholder:

```
use IPC::SysV qw(IPC_RMID);

msgctl($qid, IPC_RMID, 0)
    or warn "could not remove queue $qid: $!";
```

Ιδιωματικός έλεγχος επιτυχίας που λειτουργεί για την περίπτωση `"0 but true"`:

```
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
```

Οριακές περιπτώσεις

- **Το `IPC::SysV` δεν είναι φορτωμένο**: γυμνά ονόματα σταθερών όπως `IPC_STAT` γίνονται barewords και, υπό `use strict`, σφάλμα κατά τη μεταγλώττιση. Κάνετε πάντα πρώτα `use IPC::SysV`.
- **Το βαθμωτό που περνά στο `IPC_STAT` δεν είναι εγγράψιμο**: πεθαίνει με `Modification of a read-only value attempted`. Χρησιμοποιήστε απλό `my $buf = ""` - όχι σταθερά, όχι κυριολεκτικό συμβολοσειράς.
- **Αναντιστοιχία μήκους στο `IPC_SET`**: αν το ARG είναι μικρότερο από τη `msqid_ds` του πυρήνα, η κλήση αποτυγχάνει με `EFAULT`. Ξεκινάτε πάντα από έναν ενταμιευτή `IPC_STAT` αντί να πακετάρετε έναν από την αρχή.
- **Αφαιρεμένη ουρά (`EIDRM`)**: μόλις εκδοθεί η `IPC_RMID`, οποιαδήποτε περαιτέρω λειτουργία στο ίδιο ID αποτυγχάνει με το `$!` τεθειμένο σε `EIDRM`, όχι `EINVAL`. Σενάρια που κάνουν polling σε ουρά πρέπει να αντιμετωπίζουν το `EIDRM` ως «η ουρά χάθηκε, σταμάτα».
- **Σφάλματα δικαιωμάτων**: οι `IPC_SET` και `IPC_RMID` απαιτούν αντιστοίχιση ενεργού UID ή `CAP_SYS_ADMIN`. Ο πυρήνας επιστρέφει `EPERM`, όχι `EACCES`, για αποτυχίες ιδιοκτησίας.
- **Παγίδα τιμής επιστροφής**: το `if (msgctl(...) == 0)` είναι **λάθος** - σε επιτυχία η τιμή είναι η συμβολοσειρά `"0 but true"`, που μετατρέπεται σε συμβολοσειρά ως μη κενή αλλά συγκρίνεται αριθμητικά ίση με `0`. Δοκιμάστε με `defined` για την περίπτωση σφάλματος, ή βασιστείτε στο `or die`.
- **Διαθεσιμότητα πλατφόρμας**: το System V IPC δεν είναι καθολικό. Σε συστήματα όπου ο πυρήνας το παραλείπει, η `msgctl` εγείρει την εξαίρεση χρόνου εκτέλεσης `msgctl not implemented`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44 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.

Δείτε επίσης

- `msgget` - αποκτήστε το ID ουράς επί του οποίου λειτουργεί η `msgctl`
- `msgsnd` - στείλτε ένα μήνυμα στην ουρά
- `msgrcv` - λάβετε ένα μήνυμα από την ουρά
- `semctl` - το αντίστοιχο για σύνολο σηματοφόρων, ίδια σύμβαση επιστροφής και ίδιο μοτίβο `IPC_STAT` / `IPC_SET` / `IPC_RMID`
- `shmctl` - το αντίστοιχο για τμήμα κοινόχρηστης μνήμης
- `IPC::SysV` - οι σταθερές (`IPC_STAT`, `IPC_SET`, `IPC_RMID`) και οι βοηθοί pack/unpack για `msqid_ds` που απαιτεί αυτή η κλήση

SysV IPC

msgget

Δημιουργεί ή αναζητά μια ουρά μηνυμάτων System V IPC και επιστρέφει το id της.

Η `msgget` είναι το σημείο εισόδου στην οικογένεια ουράς μηνυμάτων System V: κάθε επόμενη κλήση `msgsnd`, `msgrcv` ή `msgctl` χρειάζεται ένα id, και από εδώ προκύπτει αυτό το id. Το KEY ονοματίζει την ουρά σε όλο το σύστημα (ένας ακέραιος `key_t`, τυπικά κατασκευασμένος με `IPC::SysV::ftok` ή η ειδική τιμή `IPC_PRIVATE`). Το `FLAGS` είναι το bitwise-or των bits δικαιωμάτων (τα κάτω εννέα, ίδια διάταξη με τα modes αρχείων) και προαιρετικών σημαιών `IPC_CREAT` / `IPC_EXCL`. Η κλήση είναι άμεσο περιτύλιγμα γύρω από την κλήση συστήματος `msgget(2)`.

Σύνοψη

```
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);
```

Τι επιστρέφεται

Ένας μη αρνητικός ακέραιος id ουράς μηνυμάτων σε επιτυχία, ή `undef` σε αποτυχία (με την `$!` ορισμένη στην τιμή `errno` από την υποκείμενη κλήση συστήματος). Το id είναι αδιαφανές handle - μη κάνετε αριθμητική πάνω του, απλώς περάστε το πίσω στις άλλες ενσωματωμένες `msg*`.

```
my $id = msgget($key, IPC_CREAT | 0600)
    // die "msgget($key) failed: $!";
```

Καθολική κατάσταση που επηρεάζει

- `$!` - ορίζεται σε αποτυχία στο `errno` από την `msgget(2)` (συνήθως `EACCES`, `EEXIST`, `ENOENT`, `ENOSPC`, `ENOMEM`).

Δεν εμπλέκονται άλλες ειδικές μεταβλητές. Η ίδια η ουρά ζει στον πυρήνα, όχι στη διεργασία Perl, και διατηρείται μετά από `exec` και αφότου εξέλθει η διεργασία που τη δημιούργησε - πρέπει να αφαιρεθεί ρητά με `msgctl` και `IPC_RMID`.

Παραδείγματα

Δημιουργήστε ιδιωτική ουρά που καμία άλλη διεργασία δεν μπορεί να ονοματίσει - η συνήθης επιλογή όταν η ουρά μοιράζεται μόνο με παιδιά αυτής της διεργασίας:

```
use IPC::SysV qw(IPC_PRIVATE);
my $id = msgget(IPC_PRIVATE, 0600) // die "msgget: $!";
```

Δημιουργήστε ή προσαρτηθείτε σε επώνυμη ουρά χρησιμοποιώντας ένα `key_t` παραγόμενο από διαδρομή αρχείου. Η `ftok` μετατρέπει διαδρομή συν ένα id έργου ενός χαρακτήρα σε σταθερό κλειδί ώστε άσχετες διεργασίες να μπορούν να συμφωνούν σε μια ουρά:

```
use IPC::SysV qw(ftok IPC_CREAT);
my $key = ftok("/var/run/myapp", 'M');
my $id  = msgget($key, IPC_CREAT | 0600) // die "msgget: $!";
```

Δημιουργία αποκλειστικά - αποτυχία αν η ουρά υπάρχει ήδη. Έτσι ανιχνεύετε ότι ένα προηγούμενο στιγμιότυπο του προγράμματός σας υπάρχει ακόμη:

```
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: $!";
}
```

Προσάρτηση σε υπάρχουσα ουρά χωρίς δημιουργία νέας - παραλείψτε το `IPC_CREAT`. Επιστρέφει `undef` με την `$!` ορισμένη σε `ENOENT` αν δεν υπάρχει τέτοια ουρά:

```
my $id = msgget($key, 0) // die "no queue for key $key: $!";
```

Αφαιρέστε την ουρά όταν τελειώσετε. Η ίδια η `msgget` δεν έχει αντίστοιχο αποδόμησης· η αφαίρεση γίνεται μέσω της `msgctl`:

```
use IPC::SysV qw(IPC_RMID);
msgctl($id, IPC_RMID, 0) or warn "msgctl IPC_RMID: $!";
```

Οριακές περιπτώσεις

- **Το KEY πρέπει να είναι ακέραιος**, τυπικά `key_t`. Η παροχή συμβολοσειράς πυροδοτεί τους συνήθεις κανόνες αριθμητικής μετατροπής και σχεδόν σίγουρα θα ονοματίσει λάθος ουρά. Κατασκευάστε κλειδιά με `IPC::SysV::ftok` ή χρησιμοποιήστε `IPC_PRIVATE`.

- **Το `IPC_PRIVATE` δημιουργεί πάντα νέα ουρά**, ανεξάρτητα από το αν έχει τεθεί το `IPC_CREAT`. Το κλειδί δεν είναι επαναχρησιμοποιήσιμο - δεν μπορείτε αργότερα να καλέσετε `msgget` για την ίδια ουρά από άσχετη διεργασία χρησιμοποιώντας `IPC_PRIVATE`.
- **Τα bits δικαιωμάτων εφαρμόζονται σε κάθε κλήση, όχι μόνο στη δημιουργία**. Η προσάρτηση σε υπάρχουσα ουρά χωρίς `IPC_CREAT` ελέγχει και πάλι τα bits δικαιωμάτων έναντι του ιδιοκτήτη/ομάδας της ουράς και του ενεργού uid/gid της τρέχουσας διεργασίας. Bits που δεν ταιριάζουν δίνουν `EACCES`.
- **Το `IPC_EXCL` χωρίς `IPC_CREAT` δεν έχει νόημα** - ο πυρήνας το αγνοεί. Αν θέλετε αποκλειστική δημιουργία, ζευγαρώνετέ τα πάντα.
- **Όρια σε επίπεδο συστήματος**. Η δημιουργία μπορεί να αποτύχει με `ENOSPC` όταν χτυπηθεί το όριο `MSGMNI` του πυρήνα, ή με `ENOMEM` υπό πίεση μνήμης. Αυτά δεν είναι σφάλματα του προγράμματός σας· είναι λειτουργικές συνθήκες.
- **Ο χρόνος ζωής της ουράς ξεπερνά τις διεργασίες**. Μια ουρά που δημιουργείται εδώ διατηρείται μέχρι να αφαιρεθεί ρητά με `msgctl` και `IPC_RMID`, ή μέχρι να επανεκκινήσει το σύστημα. Ξεχασμένες ουρές διαρρέουν πόρους πυρήνα που τα `ipcs` και `ipcrm` χρησιμοποιούνται για να εντοπίσουν και να καθαρίσουν.
- **Δεν διαθέτουν όλα τα συστήματα ουρές μηνυμάτων SysV.** Σε πλατφόρμες που τις παραλείπουν, η ενσωματωμένη συνάρτηση κράζει με `msgget not implemented on this architecture`. Η προειδοποίηση φορητότητας στο upstream POD ισχύει.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `msgsnd` - στέλνει μήνυμα στην ουρά που επέστρεψε η `msgget`
- `msgrcv` - λαμβάνει μήνυμα από αυτή την ουρά
- `msgctl` - διερωτάται, θέτει επιλογές ή αφαιρεί την ουρά· η αποδόμηση μετά την `msgget` περνά από εδώ
- `semget` - ίδιο σχήμα API για σηματοφόρους System V όταν χρειάζεστε συγχρονισμό αντί για μεταφορά δεδομένων
- `shmget` - ίδιο σχήμα API για κοινή μνήμη System V όταν χρειάζεστε άμεσο διαμοιρασμό μνήμης αντί για ανταλλαγή μηνυμάτων
- `IPC::SysV` - παρέχει το `ftok` και τις σταθερές `IPC_*` / `S_*` που η `msgget` περιμένει στο `FLAGS`

SysV IPC

msgrcv

Λήψη μηνύματος από ουρά μηνυμάτων System V IPC.

Η `msgrcv` είναι η σύνδεση Perl για την κλήση συστήματος `msgrcv(2)`. Αντλεί ένα μήνυμα από την ουρά που προσδιορίζεται από το `ID`, αποθηκεύει τον τύπο μηνύματος και το payload στο `VAR`, και επιστρέφει αληθή τιμή σε επιτυχία ή ψευδή σε σφάλμα. Το `SIZE` είναι ο μέγιστος αριθμός bytes payload που ο καλών είναι διατεθειμένος να αποδεχτεί· το `TYPE` επιλέγει ποιο μήνυμα θα ληφθεί· το `FLAGS` είναι η bitmask που περνά στον πυρήνα (`IPC_NOWAIT`, `MSG_NOERROR`, κ.λπ.).

Ο επιστρεφόμενος buffer **δεν** είναι μόνο το σώμα του μηνύματος - τα πρώτα `sizeof(long)` bytes είναι ο τύπος μηνύματος που πέρασε ο αποστολέας στην `msgsnd`, ακολουθούμενα από το payload. Χρησιμοποιήστε `unpack` με πρότυπο `l! a*` για να τα διαχωρίσετε.

Σύνοψη

```
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;
```

Τι επιστρέφεται

Αληθές σε επιτυχία, ψευδές σε σφάλμα με την `$!` ορισμένη. Σε επιτυχία, το `VAR` αντικαθίσταται με το ληφθέν frame: ένα native `long` (ο τύπος μηνύματος) ακολουθούμενο άμεσα από έως `SIZE` bytes payload. Η μεταβλητή σημειώνεται ως tainted υπό `-T`.

Το `VAR` επεκτείνεται ή συρρικνώνεται για να χωρέσει το πραγματικό μήκος του frame - δεν χρειάζεται να το προδιαστασιολογήσετε. Το περιεχόμενο πριν την κλήση απορρίπτεται.

Για να λάβετε τον τύπο και το payload ως ξεχωριστές τιμές:

```
my ($type, $msg) = unpack "l! a*", $buf;
```

Ορίσματα

- `ID` - το id ουράς που επιστράφηκε προηγουμένως από την `msgget`.
- `VAR` - ένα lvalue βαθμωτό που λαμβάνει το αδρό frame. Η προηγούμενη τιμή του απορρίπτεται σε επιτυχία.
- `SIZE` - το μέγιστο μέγεθος payload σε bytes. Αν το επόμενο μήνυμα που ταιριάζει είναι μεγαλύτερο από αυτό, η `msgrcv` αποτυγχάνει με `E2BIG` και το μήνυμα παραμένει στην ουρά - **εκτός** αν είναι ορισμένο το `MSG_NOERROR` στα `FLAGS`, οπότε το μήνυμα αποκόπτεται σιωπηρά και τα επιπλέον bytes χάνονται.
- `TYPE` - επιλέγει ποιο μήνυμα θα ληφθεί:
  - `0` - το πρώτο μήνυμα στην ουρά, ανεξάρτητα από τον τύπο του.

  - θετικό `N` - το πρώτο μήνυμα του οποίου ο τύπος είναι ακριβώς `N`.
  - αρνητικό `-N` - το πρώτο μήνυμα του χαμηλότερου τύπου που είναι `<= abs(N)`, χρήσιμο για ουρές προτεραιότητας.
- `FLAGS` - bitmask από `IPC_NOWAIT` (άμεση επιστροφή `ENOMSG` αντί για μπλοκάρισμα όταν δεν υπάρχει διαθέσιμο μήνυμα που ταιριάζει), `MSG_NOERROR` (αποκοπή αντί για αποτυχία σε υπερμεγέθη μηνύματα), και τη Linux-ειδική `MSG_EXCEPT` (με θετικό `TYPE`, λήψη του πρώτου μηνύματος του οποίου ο τύπος *δεν* είναι `TYPE`).

Οι σταθερές βρίσκονται στο `IPC::SysV`· τα σκέτα ονόματα δεν γίνονται ορατά μέχρι να τα εισάγετε.

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται σε σφάλμα στο υποκείμενο `errno` (`ENOMSG`, `E2BIG`, `EIDRM`, `EINVAL`, `EACCES`, `EFAULT`, `EINTR`).
- Taint mode - το ληφθέν `VAR` σημειώνεται ως tainted, αφού το περιεχόμενό του προήλθε από άλλη διεργασία.

Παραδείγματα

Λήψη με μπλοκάρισμα οποιουδήποτε μηνύματος, και έπειτα διαχωρισμός τύπου και 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";
```

Λήψη μόνο μηνυμάτων τύπου 42:

```
msgrcv $qid, my $buf, 4096, 42, 0           or die "msgrcv: $!";
```

Δημοσκόπηση χωρίς μπλοκάρισμα - άμεση επιστροφή αν η ουρά είναι κενή:

```
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: $!";
}
```

Αποκοπή υπερμεγέθων μηνυμάτων αντί για αποτυχία:

```
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
```

Λήψη τύπου προτεραιότητας - λήψη του μηνύματος με τον χαμηλότερο αριθμό του οποίου ο τύπος είναι <= 3:

```
msgrcv $qid, my $buf, 4096, -3, 0           or die "msgrcv: $!";
```

Οριακές περιπτώσεις

- **Το VAR πρέπει να είναι lvalue**. Το πέρασμα σταθεράς ή βαθμωτού μόνο για ανάγνωση (για παράδειγμα ένα στοιχείο του @_ που πέρασε με αναφορά σε κυριολεκτικό) εγείρει `Modification of a read-only value attempted`.
- **Η διάταξη του frame είναι native long, όχι τέσσερα bytes**. Σε 64-bit Linux ο τύπος καταλαμβάνει 8 bytes· σε 32-bit συστήματα καταλαμβάνει 4. Το `l!` στο `unpack "l! a*"` - το `!` - είναι υποχρεωτικό· ένα σκέτο `l` υποθέτει 4 bytes και θα διαφθείρει payloads σε 64-bit υπολογιστές.
- **Το SIZE εξαιρεί την επικεφαλίδα τύπου για σκοπούς του MSG_NOERROR**. Ο πυρήνας συγκρίνει το SIZE με το μήκος του payload, όχι του frame. Διαστασιολογήστε τον buffer για το μεγαλύτερο payload που αναμένετε· η επικεφαλίδα τύπου γράφεται πέρα από αυτό.
- **Διακοπή σήματος**. Μια μπλοκαριστική `msgrcv` επιστρέφει ψευδές με `$! == EINTR` όταν φτάνει σήμα πριν να είναι διαθέσιμο μήνυμα. Επαναλάβετε την κλήση εκτός αν το σήμα προορίζεται να ματαιώσει τον βρόχο λήψης.
- **Αφαίρεση ουράς ενώ είναι μπλοκαρισμένη**. Αν άλλη διεργασία καλέσει `msgctl` με `IPC_RMID` ενώ αυτή η διεργασία κοιμάται στην `msgrcv`, η κλήση επιστρέφει ψευδές με `$! == EIDRM`.
- **Δικαιώματα**. Η λήψη απαιτεί δικαίωμα ανάγνωσης στην ουρά. Το `EACCES` σημαίνει ότι η ουρά υπάρχει αλλά τα τρέχοντα uid/gid δεν ταιριάζουν με τα mode bits της ουράς.
- **Το TYPE = 0 βλέπει σειρά FIFO**. Δεν υπάρχει προτεραιότητα «οποιοδήποτε» πέραν του ρητού αρνητικού TYPE - μηνύματα όλων των τύπων αντλούνται με σειρά αποστολής.
- **Υποστήριξη πυρήνα**. Πυρήνες Linux χτισμένοι χωρίς `CONFIG_SYSVIPC`, και container runtimes που δεν εκθέτουν τον χώρο ονομάτων IPC, αποτυγχάνουν κάθε `msgrcv` με `ENOSYS`. Ελέγξτε κατά την εκκίνηση αν ο στόχος ανάπτυξής σας είναι αβέβαιος.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `msgsnd` - η αντίστοιχη που τοποθετεί μηνύματα στην ουρά· το πρότυπο `pack "l! a*"` της είναι το αντίστροφο αυτού που κάνετε unpack εδώ
- `msgget` - απόκτηση του id ουράς `ID` που περνάτε
- `msgctl` - επιθεώρηση ή διαγραφή της ουράς, συμπεριλαμβανομένου του `IPC_RMID` που προκαλεί επιστροφή `EIDRM` από μπλοκαρισμένη `msgrcv`
- `unpack` - διαχωρισμός του επιστρεφόμενου frame σε τύπο και payload με `l! a*`
- `IPC::SysV` - πηγή των σταθερών σημαίας `IPC_NOWAIT`, `MSG_NOERROR` και σχετικών
- `IPC::Msg` - αντικειμενοστρεφές περιτύλιγμα που αποκρύπτει τη διαδικασία pack/unpack πίσω από μια μέθοδο `rcv`

SysV IPC

msgsnd

Στέλνει ένα μήνυμα σε μια ουρά μηνυμάτων System V IPC.

Η `msgsnd` είναι ένα λεπτό περιτύλιγμα γύρω από την κλήση συστήματος `msgsnd(2)`. Προσαρτά το `MSG` στην ουρά που προσδιορίζεται από το `ID` (το queue id που επέστρεψε προηγουμένως η `msgget`), και επιστρέφει αληθές σε επιτυχία ή ψευδές σε σφάλμα με ρυθμισμένο το `$!`. Το `MSG` **δεν** είναι ελεύθερης μορφής: πρέπει να ξεκινά με έναν τύπο μηνύματος εγγενούς `long`, ακολουθούμενο από τα bytes του payload. Τα `FLAGS` ελέγχουν το μπλοκάρισμα και τη συμπεριφορά σφάλματος-σε-υπερχείλιση, τυπικά `0` ή `IPC_NOWAIT`.

Σύνοψη

```
msgsnd ID, MSG, FLAGS
```

Τι επιστρέφεται

Αληθές σε επιτυχία, ψευδές σε αποτυχία. Σε αποτυχία το `$!` τίθεται στο `errno` της `msgsnd(2)` - συνηθισμένες τιμές είναι `EAGAIN` (ουρά γεμάτη, τέθηκε `IPC_NOWAIT`), `EIDRM` (η ουρά αφαιρέθηκε), `EINTR` (σήμα διέκοψε μπλοκαριστική κλήση), και `EACCES` (κανένα δικαίωμα εγγραφής στην ουρά).

Επειδή η τιμή επιστροφής είναι απλή λογική τιμή, ελέγξτε την πάντα πριν υποθέσετε ότι το μήνυμα μπήκε στην ουρά:

```
msgsnd($qid, $msg, 0)
    or die "msgsnd($qid) failed: $!";
```

Μορφή μηνύματος

Ο πυρήνας δεσμεύει τα πρώτα `sizeof(long)` bytes κάθε μηνύματος που μπαίνει στην ουρά για τον **τύπο μηνύματος**, έναν θετικό ακέραιο `long` που οι καλούντες της `msgrcv` χρησιμοποιούν για να επιλέξουν ποια μηνύματα να βγάλουν από την ουρά. Το payload ακολουθεί, χωρίς ερμηνεία.

Δημιουργήστε τον ενταμιευτή με την `pack` χρησιμοποιώντας τους ειδικοποιητές `l!` (εγγενές long) και `a*` (ακατέργαστα bytes):

```
my $type    = 1;
my $payload = "hello queue";
my $msg     = pack("l! a*", $type, $payload);
msgsnd($qid, $msg, 0) or die "msgsnd: $!";
```

Ο τύπος μηνύματος **πρέπει να είναι μεγαλύτερος από μηδέν**. Το μηδέν και οι αρνητικές τιμές είναι δεσμευμένες για τη σημασιολογία επιλογέα της `msgrcv` και προκαλούν αποτυχία της `msgsnd(2)` με `EINVAL`.

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται στο `errno` της αποτυχημένης κλήσης συστήματος όταν η `msgsnd` επιστρέφει ψευδές. Ανέπαφο σε επιτυχία.

Παραδείγματα

Αποστολή ενός απλού πληκτρολογημένου μηνύματος με την προεπιλεγμένη μπλοκαριστική συμπεριφορά. Αν η ουρά είναι γεμάτη, η κλήση περιμένει μέχρι να γίνει διαθέσιμος χώρος:

```
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: $!";
```

Αποστολή χωρίς μπλοκάρισμα - άμεση επιστροφή με `EAGAIN` στο `$!` αν η ουρά είναι γεμάτη, αντί για ύπνο:

```
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: $!";
    }
}
```

Χρήση του πεδίου τύπου ως ετικέτα προτεραιότητας / καναλιού. Οι παραλήπτες επιλέγουν με βάση τον τύπο μέσω της `msgrcv`:

```
my %CHANNEL = (log => 1, metric => 2, alert => 3);

sub send_event {
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    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");
```

Αποστολή δομημένου payload - pack-άρετε τα πεδία της επικεφαλίδας μετά το υποχρεωτικό πρόθεμα τύπου:

```
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: $!";
```

Οριακές περιπτώσεις

- **Ο τύπος μηνύματος πρέπει να είναι θετικός.** Το πέρασμα `0` ή αρνητικού τύπου αποτυγχάνει με EINVAL. Η σημασιολογία επιλογής τύπου στην `msgrcv` δεσμεύει αυτές τις τιμές για λογική επιλογέα.
- **Το μήνυμα υπερβαίνει το `msg_qbytes`.** Αν το MSG είναι μεγαλύτερο από το ρυθμισμένο ανά-μήνυμα όριο της ουράς, η msgsnd αποτυγχάνει με EINVAL ανεξαρτήτως των FLAGS. Ελέγξτε και ανεβάστε το όριο με `msgctl` IPC_SET αν ελέγχετε την ουρά.
- **Ουρά γεμάτη χωρίς `IPC_NOWAIT`.** Η κλήση μπλοκάρει μέχρι να βγάλει μήνυμα από την ουρά μια άλλη διεργασία, να αφαιρεθεί η ουρά (EIDRM), ή να την διακόψει σήμα (EINTR). Επαναλάβετε σε EINTR αν το σήμα είναι αναμενόμενο και αβλαβές.
- **Ουρά γεμάτη με `IPC_NOWAIT`.** Αποτυγχάνει αμέσως με EAGAIN. Αυτός είναι ο μόνος τρόπος επιβολής φραγμένης καθυστέρησης στην πλευρά της αποστολής.
- **Λείπει το πρόθεμα τύπου.** Η παράλειψη pack-αρίσματος του προθέματος τύπου `l!` περνά το ακατέργαστο payload ως έχει· τα πρώτα `sizeof(long)` bytes ερμηνεύονται τότε ως τύπος - τυπικά μεγάλος θετικός ή αρνητικός αριθμός - και τα υπόλοιπα ως σώμα. Οι παραλήπτες που φιλτράρουν κατά τύπο δεν βλέπουν τίποτα.
- **Το ID είναι queue id, όχι κλειδί.** Περάστε τον ακέραιο που επιστρέφει η `msgget`, όχι το KEY που δώσατε στην `msgget`. Η σύγχυση μεταξύ τους οδηγεί σε EINVAL ή, χειρότερα, σε επιτυχημένη αποστολή σε άσχετη ουρά που τυχαίνει να μοιράζεται την αριθμητική τιμή.
- **Δικαιώματα.** Η διεργασία πρέπει να έχει δικαίωμα εγγραφής στην ουρά (το bit w του mode της ουράς για το ενεργό uid/gid της). Διαφορετικά η κλήση αποτυγχάνει με EACCES.
- **Tainting.** Το MSG διαβάζεται ως συμβολοσειρά bytes· δεν γίνεται έλεγχος taint σε αυτό εδώ. Αν το MSG προήλθε από μη έμπιστη πηγή, η πλευρά παραλήπτη (`msgrcv`, που *όντως* taint-άρει την έξοδό της) είναι εκεί όπου ανήκει το φιλτράρισμα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `msgget` - αποκτά το queue id που περνά ως `ID`· καλέστε αυτό πρώτα, ή λάβετε το id από όποιον δημιούργησε την ουρά
- `msgrcv` - η αντίστοιχη πρωτογενής λειτουργία λήψης· το όρισμά της `TYPE` επιλέγει έναντι του προθέματος `l!` που γράφεται εδώ
- `msgctl` - επιθεωρεί το όριο `msg_qbytes` της ουράς ή αφαιρεί την ουρά όταν τελειώσετε
- `pack` - δημιουργία του ενταμιευτή `l! a*` που απαιτεί η `msgsnd`· επίσης το εργαλείο για οποιαδήποτε περαιτέρω δομή μέσα στο payload
- `$!` - μεταφέρει το `errno` από αποτυχημένη `msgsnd`· συγκρίνετε με τις σταθερές `Errno` για διακλάδωση σε συγκεκριμένες αποτυχίες

Εμβέλεια

my

Δηλώνει μία ή περισσότερες μεταβλητές με λεξιλογική εμβέλεια.

Η `my` εισάγει νέες μεταβλητές των οποίων η ορατότητα περιορίζεται από το περικλείον μπλοκ, αρχείο, ή `eval`. Ο μεταγλωττιστής καταγράφει τη δήλωση κατά τη μεταγλώττιση· ο αποθηκευτικός χώρος εκχωρείται και (επαν)αρχικοποιείται κάθε φορά που ο έλεγχος εισέρχεται στην εμβέλεια. Σε αντίθεση με την `local`, η `my` δεν αντικαθιστά προσωρινά μια μεταβλητή πακέτου - δημιουργεί μια γνήσια νέα μεταβλητή που υπάρχει μόνο όσο η εμβέλεια είναι ζωντανή και είναι αόρατη έξω από αυτήν, ακόμη και στις υπορουτίνες που καλεί η εμβέλεια.

- 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.

Σύνοψη

```
my EXPR
my TYPE EXPR
my EXPR : ATTRS
my TYPE EXPR : ATTRS
```

Μία μεταβλητή μπορεί να γραφεί χωρίς παρενθέσεις· λίστα δύο ή περισσότερων πρέπει να είναι σε παρενθέσεις:

```
my $x;
my ($a, $b, @rest);
my Foo $obj;
my $x : shared = 0;
```

Τι επιστρέφεται

Η `my` είναι δήλωση, όχι τελεστής παραγωγής τιμής με τη συνηθισμένη έννοια. Σε θέση rvalue αποδίδει τη μεταβλητή της (ή τη λίστα μεταβλητών) ως lvalue στο οποίο μπορείτε να αναθέσετε. Οι συνήθεις ιδιωματισμοί είναι:

```
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
```

Όταν δεν δίνεται αρχικοποιητής, κάθε δηλωμένη μεταβλητή ξεκινά ως `undef` (βαθμωτό), κενή λίστα (πίνακας) ή κενός hash.

Εμβέλεια και διάρκεια ζωής

- **Εμβέλεια μπλοκ.** Μια μεταβλητή `my` είναι ορατή από το σημείο αμέσως μετά τη δήλωσή της μέχρι το τέλος του πιο εσωτερικού περικλείοντος μπλοκ, αρχείου, `eval`, `do`, `require` ή αρχείου που έχει φορτωθεί με `use`.
- **Εμβέλεια ελεγκτικής έκφρασης.** Η ελεγκτική έκφραση των `if` / `unless` / `elsif` / `else`, `while` / `until`, `for` / `foreach` αποτελεί μέρος της εμβέλειας της δήλωσης που περιέχει. Στο `while (my $line = <$fh>) { ... }` η μεταβλητή `$line` είναι ζωντανή σε όλο το σώμα του βρόχου και στο μπλοκ `continue` του, αλλά όχι πέρα από αυτά.
- **Φρέσκος αποθηκευτικός χώρος σε κάθε είσοδο.** Κάθε φορά που ο έλεγχος εισέρχεται εκ νέου στην εμβέλεια, η μεταβλητή δημιουργείται από την αρχή. Αυτή είναι η βασική διαφορά από την `state`, η οποία διατηρεί την τιμή της μεταξύ εισόδων.
- **Οι κλειστότητες (closures) συλλαμβάνουν αποθηκευτικό χώρο, όχι τιμές.** Μια ανώνυμη υπορουτίνα που δημιουργείται μέσα στην εμβέλεια κρατά αυτό το συγκεκριμένο στιγμιότυπο της μεταβλητής ζωντανό όσο η υπορουτίνα είναι προσπελάσιμη.

```
sub make_counter {
    my $n = 0;
    return sub { ++$n };    # closes over this $n
}
my $c = make_counter();
$c->(); $c->();             # 1, 2
```

Μορφή λίστας και αποδόμηση

Με παρενθέσεις, η `my` παρέχει περιβάλλον λίστας στη δεξιά πλευρά και αποδομεί το αποτέλεσμα:

```
my ($sec, $min, $hour) = localtime;
my ($head, @tail)      = @list;       # @tail absorbs the rest
my ($x, $y)            = ($y, $x);    # swap via list assignment
```

Χρησιμοποιήστε `undef` ως placeholder για να παρακάμψετε μια θέση:

```
my (undef, $min, $hour) = localtime;  # discard seconds
```

Χωρίς παρενθέσεις, το `my $foo` αποτελεί δήλωση μίας μόνο μεταβλητής και ο τελεστής κόμματος εφαρμόζεται σε ό,τι ακολουθεί:

```
my $foo, $bar = 1;          # WRONG: declares $foo only;
                            # $bar is the package variable
my ($foo, $bar) = (0, 1);   # Right.
```

Παγίδες περιβάλλοντος

Η `my` δεν αλλάζει τον τρόπο που αξιολογείται η δεξιά πλευρά - τροποποιεί μόνο τη μεταβλητή στα αριστερά. Ένα βαθμωτό `my $foo` επιβάλλει βαθμωτό περιβάλλον· ένα `my (...)` σε παρενθέσεις επιβάλλει περιβάλλον λίστας:

```
my  $line  = <$fh>;         # scalar context: one line
my ($line) = <$fh>;         # list context: first of a list
my  @lines = <$fh>;         # list context: all remaining lines
```

Το κλασικό λάθος είναι να γράφεται `my ($foo) = <$fh>` ενώ ζητούνταν μία γραμμή - οι παρενθέσεις ενεργοποιούν περιβάλλον λίστας και το filehandle επιστρέφει λίστα ενός στοιχείου· τυχαίνει η τιμή να είναι η πρώτη γραμμή, αλλά το περιβάλλον γύρω είναι λάθος για ό,τι ακολουθεί.

`my` έναντι `local` έναντι `our` έναντι `state`

Τέσσερις δηλώσεις, τέσσερις διαφορετικές σημασίες:

- `my` - εισάγει μια **νέα λεξιλογική** μεταβλητή. Δεν είναι ορατή στον καλούμενο κώδικα. Φρέσκος αποθηκευτικός χώρος σε κάθε είσοδο.
- `local` - **δεν** δηλώνει τίποτα. Αποθηκεύει την τρέχουσα τιμή μιας **υπάρχουσας** μεταβλητής **πακέτου** και την επαναφέρει όταν η περικλείουσα εμβέλεια τερματίζει. Ορατή στον καλούμενο κώδικα μέσω δυναμικής εμβέλειας. Χρησιμοποιήστε την για τη μικρή χούφτα μαγικών καθολικών (`$_`, `$/`, `$\`, `$,`, `$|`, `%ENV`, ...) που δεν μπορούν να γίνουν λεξιλογικές.
- `our` - δηλώνει ένα **λεξιλογικό ψευδώνυμο** για μια μεταβλητή πακέτου (καθολική). Ο αποθηκευτικός χώρος βρίσκεται στον πίνακα συμβόλων του πακέτου· η `our` απλώς σας επιτρέπει να την αναφέρετε χωρίς το πρόθεμα πακέτου για το υπόλοιπο της λεξιλογικής εμβέλειας. Ικανοποιεί το `use strict 'vars'`.
- `state` - όπως η `my` ως προς την ορατότητα, αλλά ο αποθηκευτικός χώρος **διατηρείται** μεταξύ επανεισόδων στην εμβέλεια. Η αρχικοποίηση εκτελείται μία φορά· οι επόμενες είσοδοι αφήνουν την τιμή ανέπαφη.

```
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
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    ++$calls;
    print ++$total;
}
```

Κρίσιμο σημείο: η `my $var` **δεν** καλύπτει μια μεταβλητή πακέτου με το ίδιο όνομα. Η μεταβλητή πακέτου παραμένει προσβάσιμη μέσω της πλήρως ειδικευμένης μορφής της όσο η λεξιλογική είναι εντός εμβέλειας:

```
package main;
our $x = 10;
my  $x = 20;
print "$x and $::x\n";      # "20 and 10\n"
```

TYPE και χαρακτηριστικά

Και οι δύο μορφές αποτελούν μέρος της γλώσσας αλλά η σημασιολογία τους εξακολουθεί να εξελίσσεται (διατύπωση upstream).

- **TYPE** - ένα bareword όνομα κλάσης, μια σταθερά δηλωμένη μέσω `use constant`, ή το `__PACKAGE__`. Ιστορικά συνδεδεμένο με την pragma `fields`· για συνηθισμένη χρήση λειτουργεί ως υπόδειξη και δεν επιβάλλει τύπο κατά τον χρόνο εκτέλεσης.

  ```
  my Foo $obj = Foo->new;
  ```

- **ATTRS** - μια λίστα χαρακτηριστικών εισαγόμενη με άνω-κάτω τελεία που χειρίζεται η pragma `attributes` (και το `Attribute::Handlers` από την 5.8.0).

  ```
  my $x : shared;
  my @buf : Locked;
  ```

Δείτε L<perlsub/»Private Variables via my()»> στο upstream για την έγκυρη διατύπωση.

Η δήλωση δεν είναι ακόμη εντός εμβέλειας στο ίδιο της το RHS

Η νεοδηλωμένη μεταβλητή δεν είναι ορατή παρά **μετά** την ολοκλήρωση της εντολής δήλωσης. Αυτό κάνει αυτόν τον ιδιωματισμό καλά ορισμένο:

```
my $x = $x;                 # new $x initialised from the old $x
```

και αυτή τη συνθήκη εξαρτημένη από προϋπάρχον εξωτερικό `$x`:

```
my $x = 123 and $x == 123;  # false unless the old $x was 123
```

Μέσα σε μία εντολή, οποιεσδήποτε περαιτέρω αναφορές στο όνομα παραπέμπουν στη σύνδεση προ της δήλωσης (ή προκαλούν σφάλμα `strict 'vars'` αν δεν υπάρχει τέτοια σύνδεση):

```
our $x = 2;
foo($x, my $x = $x + 1, $x); # foo() receives (2, 3, 2)
```

Παραδείγματα

Τυπική είσοδος υπορουτίνας - ονομάστε τα ορίσματα:

```
sub distance {
    my ($x1, $y1, $x2, $y2) = @_;
    return sqrt( ($x2-$x1)**2 + ($y2-$y1)**2 );
}
```

Τοπικός δείκτης βρόχου με `for my`:

```
for my $i (1 .. 10) {
    # $i is fresh each iteration; invisible after the loop
}
```

Ιδιωτική κατάσταση εμβέλειας αρχείου - η μεταβλητή υπάρχει για όλη τη ζωή του διερμηνέα αλλά δεν είναι προσπελάσιμη από έξω από αυτό το αρχείο:

```
my $cache = {};
sub lookup { $cache->{ $_[0] } //= _compute($_[0]) }
```

Ιδιωτική υπορουτίνα μέσω λεξιλογικής αναφοράς κώδικα:

```
my $helper = sub { ... };
$helper->($x);
```

Προειδοποίηση επισκίασης - η εκ νέου δήλωση στην ίδια εμβέλεια εκπέμπει προειδοποίηση κατηγορίας shadow υπό το `use warnings`:

```
use warnings 'shadow';
my $x = 1;
my $x = 2;                  # warning: "my" variable $x masks ...
```

Οριακές περιπτώσεις

- **Μόνο αλφαριθμητικοί προσδιοριστές.** Μαγικές ενσωματωμένες μεταβλητές όπως οι `$/`, `$_`, `$\`, `%ENV`, `@ARGV` δεν μπορούν να γίνουν λεξιλογικές. Χρησιμοποιήστε `local` για να τις θέσετε δυναμικά σε εμβέλεια.
- **Καμία ειδίκευση πακέτου.** Το `my $Foo::bar` είναι συντακτικό σφάλμα: οι μεταβλητές `my` δεν ανήκουν σε πακέτο και δεν μπορούν να φέρουν ειδικευτή `::`.
- **Παρενθέσεις που λείπουν γύρω από λίστα.** Το `my $a, $b` δηλώνει μόνο το `$a`· το `$b` είναι η μεταβλητή πακέτου. Πάντα να βάζετε σε παρενθέσεις δύο ή περισσότερα ονόματα.
- **Έκπληξη περιβάλλοντος λίστας.** Το `my ($line) = <$fh>` επιβάλλει περιβάλλον λίστας στην ανάγνωση· το RHS είναι λίστα ενός στοιχείου, όχι βαθμωτό. Αφαιρέστε τις παρενθέσεις αν θέλετε βαθμωτή σημασιολογία.
- **`foreach` χωρίς `my`.** Το `for $i (...)` τοπικοποιεί το `$i` δυναμικά κατά τον τρόπο της `local`. Προτιμήστε `for my $i (...)` για να περιορίσετε τον δείκτη λεξιλογικά.

- **Επισκίαση σε βρόχους.** Μια `my` μέσα σε σώμα βρόχου δηλώνει φρέσκια μεταβλητή σε κάθε επανάληψη. Αν αυτό δεν είναι επιθυμητό (π.χ. μετρητής που πρέπει να επιβιώσει μεταξύ επαναλήψεων), δηλώστε την έξω από τον βρόχο ή χρησιμοποιήστε `state`.
- **Σύλληψη κλειστότητας.** Μια κλειστότητα πάνω σε μια μεταβλητή `my` κρατά εκείνο το συγκεκριμένο στιγμιότυπο ζωντανό. Διαφορετικές κλήσεις μιας περικλείουσας υπορουτίνας παράγουν ανεξάρτητες κλειστότητες πάνω σε ανεξάρτητες μεταβλητές.
- **Αλληλεπίδραση με `use strict 'vars'`.** Η `my` ικανοποιεί το strict-vars εντός της εμβέλειάς της· μεταβλητές χωρίς δήλωση πρέπει να είναι `our`, πλήρως ειδικευμένες, ή προδηλωμένες με `use vars`.
- **Χρόνος μεταγλώττισης έναντι χρόνου εκτέλεσης.** Η σύνδεση του ονόματος συμβαίνει κατά τη μεταγλώττιση· ο αρχικοποιητής εκτελείται κατά τον χρόνο εκτέλεσης, και επανεκτελείται σε κάθε είσοδο στην εμβέλεια. Δαπανηροί αρχικοποιητές μέσα σε καυτούς βρόχους πληρώνουν αυτό το κόστος σε κάθε επανάληψη.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `our` - λεξιλογικό ψευδώνυμο για μεταβλητή πακέτου· ικανοποιεί το `strict 'vars'` χωρίς να δημιουργεί νέο αποθηκευτικό χώρο
- `local` - δυναμικά οριοθετημένη αποθήκευση/επαναφορά υπάρχουσας μεταβλητής πακέτου· το εργαλείο για μαγικές καθολικές
- `state` - λεξιλογική όπως η `my`, αλλά ο αποθηκευτικός χώρος διατηρείται μεταξύ επανεισόδων στην εμβέλεια
- `undef` - placeholder σε λίστα `my` με παρενθέσεις, και η αρχική τιμή ενός μη αρχικοποιημένου βαθμωτού
- `use strict` - κάνει τα `my` / `our` / πλήρως ειδικευμένα ονόματα υποχρεωτικά, εντοπίζοντας τυχαίες καθολικές
- `attributes` - pragma που ερμηνεύει το τμήμα `: ATTRS` μιας δήλωσης `my`

Ροή ελέγχου

next

Ξεκινά αμέσως την επόμενη επανάληψη του περικλείοντος βρόχου.

Η `next` είναι το ισοδύναμο της εντολής `continue` της C στην Perl: εγκαταλείπει τα υπόλοιπα του σώματος της τρέχουσας επανάληψης και μεταπηδά στο βήμα επανάληψης του βρόχου. Αν ο βρόχος έχει μπλοκ `continue`, εκτελείται πρώτα αυτό το μπλοκ, μετά ελέγχεται ξανά η συνθήκη ελέγχου (ή λαμβάνεται το επόμενο στοιχείο για `foreach`), και η εκτέλεση συνεχίζεται από την κορυφή του σώματος. Με ετικέτα, η `next LABEL` στοχεύει σε συγκεκριμένο περικλείοντα βρόχο αντί για τον εσώτατο.

Σύνοψη

```
next
next LABEL
next EXPR
```

Τι επιστρέφεται

Τίποτα. Η `next` είναι τελεστής ελέγχου ροής, όχι έκφραση - ξετυλίγεται από την τρέχουσα επανάληψη και επομένως ποτέ δεν παράγει τιμή προς τον καλούντα της. Η προσπάθεια χρήσης της σε θέση που παράγει τιμή (για παράδειγμα ως δεξί μέλος ανάθεσης μέσα σε `eval {}`, `sub {}` ή `do {}`) είναι κακή χρήση: η περιβάλλουσα κατασκευή δεν ολοκληρώνεται ποτέ φυσιολογικά, οπότε δεν υπάρχει τιμή επιστροφής για παρατήρηση.

Στόχευση βρόχου

Χωρίς ετικέτα, η `next` αναφέρεται στον **εσώτατο περικλείοντα βρόχο** (`while`, `until`, `for`, `foreach`, ή σκέτο μπλοκ, το οποίο μετράει ως βρόχος μίας εκτέλεσης). Με ετικέτα αναφέρεται στον κατονομασμένο βρόχο, ανεξαρτήτως του πόσο βαθιά εμφωλευμένη βρίσκεται η `next` μέσα σε άλλους βρόχους:

```
OUTER: for my $row (@grid) {
    for my $cell (@$row) {
        next OUTER if $cell eq 'skip-row';
        process($cell);
    }
}
```

Η `next EXPR`, διαθέσιμη από την Perl 5.18, αποτιμά την `EXPR` κατά τον χρόνο εκτέλεσης και χρησιμοποιεί την τιμή της ως συμβολοσειρά για το όνομα ετικέτας. Κατά τα άλλα συμπεριφέρεται ακριβώς όπως η `next LABEL`:

```
my $target = pick_label();
next $target;                    # same as: next FOO if $target eq
 ↪"FOO"
```

Χρησιμοποιήστε τη μορφή χρόνου εκτέλεσης με φειδώ - η στατική `next LABEL` είναι ευκολότερο να διαβαστεί και να επαληθευτεί με το μάτι.

Αλληλεπίδραση με μπλοκ `continue`

Το μπλοκ `continue` ενός βρόχου εκτελείται στο τέλος κάθε επανάληψης, είτε αυτή τελείωσε φυσιολογικά είτε μέσω `next`. Αυτό είναι όλο το νόημα της ύπαρξης μπλοκ `continue`: ομαδοποιεί τη λογιστική που πρέπει να συμβαίνει σε κάθε πέρασμα.

```
my $i = 0;
LINE: while (my $line = <$fh>) {
    next LINE if $line =~ /^\s*#/;  # skip comments
    process($line);
}
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
continue {
    $i++;                               # runs even for skipped lines
}
```

Η `last` παρακάμπτει το μπλοκ `continue`· η `next` όχι. Αυτή είναι η απλούστερη διακριτική ιδιότητα των δύο, και ο λόγος που το `continue` υπάρχει ως ξεχωριστή κατασκευή αντί να ενσωματώνεται στο σώμα του βρόχου.

Παραδείγματα

Βασικός βρόχος `while` - απόρριψη γραμμών σχολίων, επεξεργασία των υπολοίπων:

```
while (my $line = <$fh>) {
    next if $line =~ /^\s*#/;
    next if $line =~ /^\s*$/;       # blank
    handle($line);
}
```

`foreach` με συσσωρευτή - παράκαμψη εγγραφών που αποτυγχάνουν σε κατηγόρημα:

```
my $total = 0;
for my $n (@values) {
    next unless defined $n;
    next if $n < 0;
    $total += $n;
}
```

Εμφωλευμένοι βρόχοι με ετικέτα - προχωρούμε στην επόμενη εξωτερική γραμμή μόλις εντοπίσουμε φρουρά στην εσωτερική σάρωση:

```
ROW: for my $row (@grid) {
    for my $cell (@$row) {
        next ROW if $cell eq 'END';
        count($cell);
    }
}
```

Μπλοκ `continue` που εκτελείται πάντα, ακόμα και σε `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
}
```

Διαφυγή από σκέτο μπλοκ - ένα αυτόνομο `{ ... }` είναι βρόχος μίας εκτέλεσης, οπότε η `next` υπερπηδά τις υπόλοιπες εντολές και βγαίνει από αυτό:

```
{
    last_chance() or next;          # bare block: next == fall out
    commit();
}
```

Οριακές περιπτώσεις

- **Παράνομη μέσα σε `grep` / `map` / μπλοκ τιμής `do { }`.** Αυτές οι κατασκευές δεν είναι βρόχοι - το μπλοκ τους αποτιμάται για την τιμή του, δεν επαναλαμβάνεται με την έννοια του ελέγχου βρόχου. Η χρήση της `next` για να βγείτε από αυτά είναι κακή χρήση και επισημαίνεται ρητά στο upstream perlfunc. Χρησιμοποιήστε αντ' αυτού υπό συνθήκη έκφραση:

```
my @kept = grep { wanted($_) } @xs;          # not: grep { next unless ... }
my @mapped = map { transform($_) // () } @ys; # empty list == skip
```

- **Δεν είναι `return`.** Η `next` δεν μπορεί να παραδώσει τιμή έξω από μπλοκ που επιστρέφει τιμή όπως `eval { }`, `sub { }` ή `do { }`. Αν βρεθείτε να καταφεύγετε στην `next` για να φύγετε από τέτοιο μπλοκ, αυτό που θέλετε είναι η `return`, ή μια υπό συνθήκη έκφραση που παράγει την τιμή που χρειάζεστε.
- **Ένα σκέτο μπλοκ είναι βρόχος μίας εκτέλεσης.** Το `{ ... }` χωρίς πρόθεμα `while` / `for` είναι σημασιολογικά βρόχος που εκτελείται ακριβώς μία φορά, οπότε η `next` βγαίνει από αυτό. Αυτό είναι ενίοτε χρήσιμο για μοτίβα πρόωρης εξόδου, αλλά η `last` είναι η συχνότερη επιλογή επειδή καθιστά την πρόθεση σαφέστερη.
- **Άγνωστη ετικέτα είναι μοιραίο σφάλμα.** Η `next NOSUCH` πεθαίνει κατά τον χρόνο εκτέλεσης με `Can't "next" outside a loop block` όταν κανένας περικλείων βρόχος δεν φέρει αυτήν την ετικέτα.
- **Το `continue` εκτελείται σε `next`, όχι σε `last`.** Αυτή είναι η συνηθισμένη αιτία έκπληξης κατά την αναδιάρθρωση: η μετακίνηση λογιστικής έξω από το σώμα του βρόχου στο `continue` αλλάζει συμπεριφορά μόνο για την `last`, όχι για την `next`.
- **Ιδιορρυθμίες προτεραιότητας και ανάλυσης.** Σε αντίθεση με τους περισσότερους κατονομασμένους τελεστές, η `next` έχει την ίδια προτεραιότητα με την ανάθεση, και εξαιρείται από τον κανόνα «μοιάζει-με-συνάρτηση». Αυτό σημαίνει ότι η `next ("foo") . "bar"` περνά το `"foobar"` στην `next`, όχι μόνο το `"foo"`. Χρησιμοποιήστε αμυντικά παρενθέσεις όταν το όνομα ετικέτας προέρχεται από έκφραση:

```
next(($cond ? "A" : "B"));
```

- **Threads, σήματα, καταστροφείς.** Η `next` ξετυλίγει τη λεξιλογική εμβέλεια της τρέχουσας επανάληψης: οι μεταβλητές `my` βγαίνουν από την εμβέλεια, οι τιμές `local` επανέρχονται, και ο `DESTROY` πυροδοτείται σε αντικείμενα εξόδου εμβέλειας, ακριβώς όπως αν η επανάληψη είχε τερματιστεί φυσιολογικά. Οι εκκρεμείς χειριστές `$SIG{...}` δεν επηρεάζονται.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `last` - έξοδος από τον περικλείοντα βρόχο τελείως, παρακάμπτοντας κάθε μπλοκ `continue`
- `redo` - επανεκκίνηση της τρέχουσας επανάληψης χωρίς εκ νέου έλεγχο της συνθήκης και χωρίς εκτέλεση του `continue`
- `return` - το σωστό εργαλείο για έξοδο από `sub { }`, `eval { }` ή `do { }` με τιμή· η `next` δεν μπορεί να το κάνει αυτό
- `grep` - φιλτράρει μια λίστα· χρησιμοποιήστε υπό συνθήκη έκφραση, όχι `next`, για να απορρίψετε στοιχεία
- `map` - μετασχηματίζει μια λίστα· επιστρέψτε την κενή λίστα `()` από το μπλοκ για να απορρίψετε ένα στοιχείο

Αρθρώματα

no

Η αντίστροφη της `use` κατά τη μεταγλώττιση - καλεί τη μέθοδο `unimport` ενός αρθρώματος για να απενεργοποιήσει ό,τι ενεργοποίησε η `use`.

Το `no MODULE LIST` αντιμετωπίζεται από τον αναλυτή ακριβώς όπως το `use MODULE LIST`, εκτός του ότι καλεί `MODULE->unimport(LIST)` αντί για `MODULE->import(LIST)`. Και τα δύο εκτελούνται κατά τη μεταγλώττιση, και τα δύο έχουν λεξιλογική εμβέλεια, και τα δύο απαιτούν το άρθρωμα να είναι ήδη φορτωμένο. Η συνηθισμένη περίπτωση είναι η απενεργοποίηση μιας pragma μέσα σε ένα μπλοκ - `no strict 'refs'`, `no warnings 'uninitialized'`, `no feature 'switch'` - χωρίς να θιγεί το περικλείον αρχείο.

Σύνοψη

```
no MODULE LIST
no MODULE
no MODULE VERSION LIST
no MODULE VERSION
```

Τι επιστρέφεται

Η `no` είναι δήλωση, όχι έκφραση. Δεν έχει χρήσιμη τιμή επιστροφής και δεν μπορεί να εμφανιστεί όπου αναμένεται τιμή. Το αποτέλεσμα είναι παρενέργεια στην κατάσταση μεταγλώττισης της περικλείουσας λεξιλογικής εμβέλειας.

Πώς αναπτύσσεται

Ο αναλυτής ξαναγράφει

```
no MODULE LIST;
```

στο κατά προσέγγιση ισοδύναμο του

```
BEGIN {
    require MODULE;
    MODULE->unimport(LIST);
}
```

- the same shape as `use`, with `unimport` swapped in for `import`. Consequences that follow from this:
- **Χρόνος μεταγλώττισης.** Το μπλοκ `BEGIN` εκτελείται μόλις ο αναλυτής φτάσει στη δήλωση `no`, πριν μεταγλωττιστεί οποιοσδήποτε επόμενος κώδικας στο αρχείο. Αυτός είναι ο λόγος που το `no strict 'refs'` τίθεται σε ισχύ στις δηλώσεις που το ακολουθούν στο ίδιο αρχείο, όχι κατά τον χρόνο εκτέλεσης.
- **Λεξιλογική εμβέλεια.** Pragmas που τιμούν τη λεξιλογική εμβέλεια (`strict`, `warnings`, `feature`, `integer`, `utf8`, και οι άλλες pragmas του πυρήνα) βλέπουν το αποτέλεσμα «off» να περιορίζεται στο εσώτατο περικλείον μπλοκ, `eval` ή αρχείο. Η έξοδος από το μπλοκ επαναφέρει την προηγούμενη κατάσταση.
- **Το `require` εξακολουθεί να συμβαίνει.** Η `no MODULE` φορτώνει το `MODULE` αν δεν έχει ήδη φορτωθεί, και μετά καλεί την `unimport` σε αυτό. Δεν χρειάζεστε αντίστοιχη `use` νωρίτερα στο αρχείο.
- **Η `unimport` είναι υποχρεωτική.** Αν το πακέτο δεν έχει μέθοδο `unimport`, η κλήση πεθαίνει κατά τη μεταγλώττιση με `Can't locate object method "unimport" via package "MODULE"`. Τα περισσότερα μη-pragma αρθρώματα δεν ορίζουν `unimport` - η `no` είναι στην πραγματικότητα χρήσιμη μόνο για αρθρώματα που τη διαφημίζουν.

Παραδείγματα

Απενεργοποίηση μιας μεμονωμένης κατηγορίας προειδοποιήσεων μέσα σε μπλοκ:

```
use warnings;

for my $row (@rows) {
    no warnings 'uninitialized';
    print join(",", @$row), "\n";   # empty fields render as ""
}
# 'uninitialized' warnings are back on here
```

Χαλάρωση της `strict 'refs'` για μία δήλωση που χρειάζεται συμβολική αποαναφορά:

```
use strict;

sub install {
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    my ($name, $code) = @_;
    no strict 'refs';
    *{"main::$name"} = $code;       # symbolic typeglob assignment
}
```

Απενεργοποίηση προειδοποίησης πειραματικού χαρακτηριστικού ενώ χρησιμοποιείται το χαρακτηριστικό:

```
use feature 'signatures';
no warnings 'experimental::signatures';

sub greet ($who) { "hello, $who\n" }
```

Επαναφορά ενός προεπιλεγμένου πακέτου χαρακτηριστικών σε σκέτη σημασιολογία Perl:

```
use v5.36;                          # enables strict, warnings, say,␣
↪signatures…
no feature 'switch';                # …but not given/when
```

Απενεργοποίηση pragma χρήστη που παρέχει δική της `unimport`:

```
use My::Pragma qw(loud);
{
    no My::Pragma;                  # My::Pragma->unimport called␣
↪here
    quiet_code();
}
```

Οριακές περιπτώσεις

- **Το `no VERSION` είναι σφάλμα μεταγλώττισης.** Η `use VERSION` υπάρχει ως έλεγχος έκδοσης κατά τη μεταγλώττιση έναντι του εκτελούμενου διερμηνέα· δεν υπάρχει συνετή αντίστροφη, οπότε το `no 5.36;` πεθαίνει με `No such class 5.36`. Ο έλεγχος έκδοσης πηγαίνει μόνο προς μία κατεύθυνση.
- **Το `no MODULE VERSION` *όντως* αναλύεται.** Απαιτεί το `MODULE` να είναι τουλάχιστον στην έκδοση `VERSION` (ίδιος έλεγχος με την `use`), και μετά καλεί `MODULE->unimport(LIST)`. Ο έλεγχος έκδοσης αφορά το φορτωμένο άρθρωμα, όχι τον διερμηνέα.
- **Άρθρωμα χωρίς `unimport`.** Πεθαίνει κατά τη μεταγλώττιση. Δεν υπάρχει εφεδρεία σε no-op· αν γράφετε άρθρωμα και θέλετε η `no` να είναι σιωπηρό no-op, ορίστε μια κενή `sub unimport {}`.
- **Μόνο bareword `MODULE`.** Όπως και στην `use`, το όνομα αρθρώματος πρέπει να είναι bareword γνωστό κατά τη μεταγλώττιση. Το `no $name;` είναι συντακτικό σφάλμα - ο αναλυτής χρειάζεται το κυριολεκτικό διακριτικό για να αποφασίσει ποια `unimport` να καλέσει.
- **Δεν αναιρεί την `use`.** Η `no Foo` καλεί `Foo->unimport`, η οποία κατά σύμβαση αντιστρέφει ό,τι έκανε η `Foo->import` στην τρέχουσα λεξιλογική εμβέλεια - αλλά

αυτό είναι σύμβαση, όχι εγγύηση της γλώσσας. Μια αμελής `unimport` μπορεί να αφήσει κατάσταση πίσω της, και οι παρενέργειες από `require Foo` (μπλοκ BEGIN, hooks `@INC`, εγκατεστημένες subs σε άλλα πακέτα) **δεν** αναιρούνται.

- **Το αποτέλεσμα τελειώνει στο περικλείον μπλοκ.** Για λεξιλογικές pragmas, η κατάσταση «off» σταματάει στο επόμενο `}`, στο όριο `eval` ή στο τέλος του αρχείου. Κώδικας σε διαφορετικό αρχείο που κάνει `require` ή `use` αυτό εδώ δεν επηρεάζεται.
- **Οι `import`/`unimport` είναι απλώς κλήσεις μεθόδων.** Η `MODULE->unimport(LIST)` εκτελείται ακριβώς όπως οποιαδήποτε άλλη μέθοδος κλάσης - βλέπει το `MODULE` ως πρώτο όρισμα και τη `LIST` μετά. Η μέθοδος μπορεί να επιθεωρήσει τον καλούντα μέσω `caller` για να βρει ποιο πακέτο πρέπει να τροποποιήσει.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `use` - η θετική μορφή· η `no` είναι κυριολεκτικά `use` με `unimport` στη θέση της `import`
- `require` - το βήμα φόρτωσης κατά τον χρόνο εκτέλεσης που εκτελεί η `no` πριν καλέσει την `unimport`
- `package` - δηλώνει το τρέχον πακέτο, που είναι η εμβέλεια που τυπικά τροποποιεί η `unimport` μιας pragma
- `import` - η συμβατική αντίστροφη μέθοδος· η `no` καλεί `unimport`, η `use` καλεί `import`
- `BEGIN` - η φάση μεταγλώττισης στην οποία τρέχει η `no`
- `caller` - πώς μια μέθοδος `unimport` αναγνωρίζει το πακέτο που πρέπει να επηρεάσει

Βαθμωτά και συμβολοσειρές · Αριθμητικές συναρτήσεις

oct

Ερμηνεία μιας συμβολοσειράς ως αριθμού γραμμένου σε οκταδικό, δεκαεξαδικό ή δυαδικό, και επιστροφή του προκύπτοντος ακεραίου.

Το όνομα είναι παραπλανητικό: η `oct` είναι στην πραγματικότητα ένας αναλυτής *προθέματος βάσης*. Επιθεωρεί τους αρχικούς χαρακτήρες της `EXPR` και επιλέγει τη βάση από αυτούς - `0x`/`x` για δεκαεξαδικό, `0b`/`b` για δυαδικό, `0o`/`o` (ή καθόλου πρόθεμα) για οκταδικό. Αν παραλειφθεί η `EXPR`, η `oct` λειτουργεί στην `$_`. Σκεφτείτε την ως το αντίστροφο του `sprintf "%o"` / `"%x"` / `"%b"` για την κοινή περίπτωση όπου έχετε τη συμβολοσειριακή μορφή και χρειάζεστε τον αριθμό.

Σύνοψη

```
oct EXPR
oct                 # operates on $_
```

Τι επιστρέφεται

Ένας μη αρνητικός ακέραιος. Η αριθμητική τιμή της EXPR ερμηνευμένη σε όποια βάση επέλεξε το πρόθεμά της (οκταδική από προεπιλογή). Το αποτέλεσμα είναι πάντα `>= 0` - η `oct` δεν αναλύει αρχικό πρόσημο και δεν επιστρέφει τιμές κινητής υποδιαστολής.

Κενή συμβολοσειρά, συμβολοσειρά χωρίς αναγνωρισμένα ψηφία, ή `undef` επιστρέφει `0`.

Κανόνες προθέματος

Οι αρχικοί χαρακτήρες της EXPR (μετά από τυχόν αρχικούς λευκούς χαρακτήρες, οι οποίοι παραλείπονται σιωπηρά) καθορίζουν τη βάση:

Πρόθεμα	Βάση	Παράδειγμα εισόδου	Αποτέλεσμα
`0x` ή `x`	16 (δεκαεξαδικό)	`"0xff"`	255
`0b` ή `b`	2 (δυαδικό)	`"0b1010"`	10
`0o` ή `o`	8 (οκταδικό)	`"0o755"`	493
(κανένα)	8 (οκταδικό)	`"755"`	493
(κανένα, αρχικό `0`)	8 (οκταδικό)	`"0755"`	493

Το οκταδικό πρόθεμα `0o`/`o` προστέθηκε στην Perl 5.33.5 για να ταιριάζει με τη σύνταξη κυριολεκτικών ακεραίων· οι παλαιότερες μορφές (`0755` και σκέτο `755`) ήταν πάντοτε οκταδικές.

Οι κάτω παύλες μεταξύ ψηφίων αγνοούνται, οπότε τιμές που αντιγράφονται κατευθείαν από κυριολεκτικά ακέραια της Perl αναλύονται καθαρά:

```
oct("0xDEAD_BEEF")      # 3735928559
oct("0b1111_0000")      # 240
oct("07_55")            # 493
```

Παραδείγματα

Αποκωδικοποίηση συμβολοσειράς δικαιωμάτων που πληκτρολόγησε ο χρήστης - ο πλήρης λόγος ύπαρξης της `oct`. Η συμβολοσειρά `"755"` **δεν** είναι ο αριθμός 755· είναι η οκταδική αναπαράσταση του αριθμού 493. Η `chmod` αναμένει τον αριθμό:

```
my $mode = "0755";
chmod oct($mode), "script.sh";          # chmod 0755
```

```
my $mode = "755";                       # no leading zero - still␣
↪octal
chmod oct($mode), "script.sh";          # same result
```

Ανάλυση δεκαεξαδικών και δυαδικών κυριολεκτικών διαβασμένων από αρχείο ρυθμίσεων ή γραμμή εντολών:

```
my $mask  = oct("0xff00");              # 65280
my $flags = oct("0b1010_0101");         # 165
```

Χειρισμός αυθαίρετων συμβολοσειρών με πρόθεμα βάσης - δεκαδικών, οκταδικών, δεκαεξαδικών ή δυαδικών - σε μία έκφραση. Αυτό είναι το κλασικό ιδίωμα oct-ως-διανομέα: ένα αρχικό `0` (σε οποιαδήποτε από τις μορφές του) σημαίνει «όχι δεκαδικό», οπότε δρομολογήστε μέσω της `oct`· αλλιώς κρατήστε την τιμή ως έχει:

```
my $n = $val =~ /^0/ ? oct($val) : $val;
```

Η μορφή με προεπιλεγμένο όρισμα λειτουργεί στην `$_`, βολική μέσα σε `map`:

```
my @perms = map { oct } @mode_strings;
```

Στρογγυλό ταξίδι με `sprintf` για να πάτε προς την άλλη κατεύθυνση:

```
my $perm = (stat $file)[2] & 07777;
my $text = sprintf "%o", $perm;         # e.g. "644"
my $back = oct($text);                  # back to the same integer
```

Οριακές περιπτώσεις

- **Κενή ή undef είσοδος**: τα `oct("")` και `oct(undef)` επιστρέφουν αμφότερα `0`. Η `undef` πυροδοτεί προειδοποίηση `uninitialized` υπό `use warnings`.
- **Καμία πρόθεση εξαρχής σε οκταδικό**: η `oct("12")` είναι `10`, όχι 12. Αυτή είναι συχνή έκπληξη - αν οι συμβολοσειρές σας μπορεί να είναι δεκαδικές, ελέγξτε πρώτα για αρχικό `0` (δείτε το ιδίωμα παραπάνω) ή μη χρησιμοποιήσετε καθόλου την `oct` και αναλύστε ρητά.
- **Οι κάτω παύλες αγνοούνται** όπου κι αν εμφανίζονται μεταξύ ψηφίων. Η `oct("0x_de_ad")` είναι `57005`. Αρχικές και τελικές κάτω παύλες γίνονται επίσης σιωπηρά αποδεκτές.
- **Άκυροι χαρακτήρες τερματίζουν την ανάλυση**: η ανάλυση σταματά στον πρώτο χαρακτήρα που δεν ανήκει στην επιλεγμένη βάση· το υπόλοιπο της συμβολοσειράς απορρίπτεται χωρίς προειδοποίηση. Η `oct("0x1g2")` επιστρέφει 1, η `oct("755.5")` επιστρέφει 493, η `oct("12abc")` επιστρέφει `10` (οκταδικό, σταμάτησε στο a), η `oct("0b102")` επιστρέφει 2 (δυαδικό `10`, σταμάτησε στο 2).
- **Οι αρχικοί λευκοί χαρακτήρες παραλείπονται σιωπηρά.** Αρχικό `+` ή `-` **δεν** είναι πρόσημο - είναι μη-ψηφίο, οπότε η ανάλυση επιστρέφει `0`. Η `oct` δεν μπορεί να παραγάγει αρνητικό αριθμό.
- **Η κινητή υποδιαστολή και η εκθετική σημειογραφία δεν γίνονται κατανοητές.** Δεν υπάρχει υποστήριξη `0x1.8p3`· η `.` τερματίζει την ανάλυση.
- **Τα πεζά/κεφαλαία προθέματος και ψηφίων δεν έχουν σημασία**: τα `0X`, `0B`, `0O`, και δεκαεξαδικά ψηφία a-f/A-F γίνονται όλα δεκτά.
- **Υπερχείλιση**: τιμές μεγαλύτερες από τον ακέραιο της πλατφόρμας προάγονται σιωπηρά σε κινητή υποδιαστολή, χάνοντας ακρίβεια χαμηλής τάξης για πολύ

μεγάλες δεκαεξαδικές/δυαδικές εισόδους.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `hex` - εξαναγκάζει βάση 16 ανεξάρτητα από το πρόθεμα· χρησιμοποιήστε την όταν η είσοδος είναι *πάντα* δεκαεξαδική και δεν θέλετε την αυτόματη ανίχνευση της `oct`
- `int` - αποκοπή αριθμητικής τιμής προς το μηδέν· άσχετη με την ανάλυση βάσης αλλά συχνά χρησιμοποιείται κατά λάθος
- `chmod` - ο κανονικός καταναλωτής της `oct`· δέχεται αριθμητικό mode, οπότε modes σε συμβολοσειριακή μορφή πρέπει να περάσουν πρώτα από την `oct`
- `sprintf` - η αντίστροφη κατεύθυνση: τα format %o, %x, %b μετατρέπουν έναν ακέραιο στην οκταδική/δεκαεξαδική/δυαδική του συμβολοσειριακή μορφή

Filehandles, αρχεία, κατάλογοι

open

Συσχετίζει ένα filehandle με ένα αρχείο, μια εντολή ή ένα βαθμωτό στη μνήμη.

Η open είναι ο κύριος τρόπος με τον οποίο ένα πρόγραμμα Perl αποκτά ένα filehandle. Η μορφή τριών ορισμάτων - `open $fh, MODE, EXPR` - είναι αυτή που πρέπει να χρησιμοποιείτε σε νέο κώδικα: ο τρόπος (mode) δηλώνεται ρητά, το όνομα αρχείου δεν μπορεί να επανερμηνευτεί ως σωλήνωση κελύφους, και οι λευκοί χαρακτήρες στην αρχή ή το τέλος του EXPR διατηρούνται κατά λέξη. Η παλαιότερη μορφή δύο ορισμάτων υπάρχει για συμβατότητα και για τη «μαγική» ερμηνεία ονόματος αρχείου που επιτρέπει· μην της δίνετε μη έμπιστη είσοδο.

Σύνοψη

```
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
```

Τι επιστρέφεται

1 σε επιτυχία· `undef` σε αποτυχία, με το `$!` ορισμένο στο σφάλμα του ΛΣ. Για τις μορφές με σωλήνα (`|-` / `-|`), η επιτυχημένη τιμή επιστροφής είναι το pid της θυγατρικής διεργασίας - χρήσιμο όταν χρειάζεται να εφαρμόσετε αργότερα `waitpid`. Μετά από άνοιγμα σωλήνα ενός ή δύο ορισμάτων με την κυριολεκτική εντολή `-`, η open επιστρέφει

δύο φορές (ο γονέας λαμβάνει το pid του παιδιού, το παιδί λαμβάνει `0`)· χρησιμοποιήστε `defined($pid)` ή `//` για να τα διακρίνετε.

Ελέγχετε πάντα την τιμή επιστροφής. Μια αποτυχημένη `open` αφήνει το filehandle κλειστό, και κάθε επόμενη ανάγνωση ή εγγραφή σε αυτό θα ορίσει το `$!` σε `Bad file descriptor` και θα εκπέμψει προειδοποίηση υπό `use warnings`.

```
open my $fh, "<", $path
    or die "open $path: $!";
```

Καθολική κατάσταση που επηρεάζει

- `$!` - ορίζεται σε αποτυχία στο σφάλμα σε επίπεδο ΛΣ.
- `${^OPEN}` - προεπιλεγμένα στρώματα I/O που εγκαθίστανται από τη pragma `open` ή τον διακόπτη `-CioD`. Ισχύουν μόνο όταν το `MODE` στη μορφή τριών ορισμάτων δεν περιέχει ρητό στρώμα· ένα γυμνό `<` ή `>` τα τιμά, ενώ ένα `<:raw` ή οποιοδήποτε άλλο ρητό στρώμα τα καταστέλλει.
- `$^F` - ο υψηλότερος περιγραφέας αρχείου που δεν έχει σημανθεί ως close-on-exec· ο νέος fd κληρονομεί τη σημαία close-on-exec βάσει αυτού.
- Το επιλεγμένο filehandle μέσω της `select` - δεν τροποποιείται από την `open`, αλλά γίνεται σχετικό τη στιγμή που κάνετε `print` στο νέο handle χωρίς να το ονομάσετε.
- Για τις μορφές σωλήνωσης, τα `$?` και `${^CHILD_ERROR_NATIVE}` ορίζονται όταν αργότερα κλείσει το handle.

Τρόποι (modes)

Το δεύτερο όρισμα ονομάζει έναν ρόλο I/O. Οι συνηθισμένες περιπτώσεις:

Τρόπος (mode)	Σημασία	ισοδύναμο της `fopen(3)`
`<`	ανάγνωση	`r`
`>`	εγγραφή, αποκοπή ή δημιουργία	`w`
`>>`	προσάρτηση, δημιουργία αν λείπει	`a`
`+<`	ανάγνωση και εγγραφή, χωρίς αποκοπή	`r+`
`+>`	ανάγνωση και εγγραφή, αποκοπή πρώτα	`w+`
`|-`	σωλήνωση προς εντολή (γράφουμε)	-
`-|`	σωλήνωση από εντολή (διαβάζουμε)	-
`<&`, `>&`, `<&=`, `>&=`	dup ή fdopen ενός υπάρχοντος handle	-

Τα νεοδημιουργηθέντα αρχεία λαμβάνουν δικαιώματα `0666 & ~umask`. Το `+<` είναι σχεδόν πάντοτε αυτό που θέλετε για ανάγνωση-εγγραφή· το `+>` πρώτα αδειάζει το αρχείο.

Τα στρώματα I/O προσαρτώνται μετά τον τρόπο: `<:encoding(UTF-8)`, `>:raw`, `>>:crlf`, `<:encoding(UTF-8):crlf`. Ένα ρητό στρώμα καταστέλλει τις προεπιλογές του `${^OPEN}`· μία μόνη άνω-κάτω τελεία (`<:`) επανέρχεται στην προεπιλογή του ΛΣ (`:raw` σε Unix, `:crlf` σε Windows).

Παραδείγματα

Διαβάστε ένα αρχείο κειμένου UTF-8 γραμμή προς γραμμή:

```
open my $fh, "<:encoding(UTF-8)", "notes.txt"
    or die "open notes.txt: $!";
while (my $line = <$fh>) {
    chomp $line;
    # ... use $line ...
}
close $fh or warn "close: $!";
```

Γράψτε ένα δυαδικό αρχείο χωρίς μετάφραση κωδικοποίησης:

```
open my $out, ">:raw", "image.bin"
    or die "open image.bin: $!";
print $out $bytes;
close $out or die "close: $!";
```

Filehandle στη μνήμη - `print` μέσα σε βαθμωτό, χρήσιμο για τη σύλληψη εξόδου που αναμένει πραγματικό handle:

```
my $buf = "";
open my $mem, ">", \$buf or die $!;
print $mem "hello\n";
close $mem;
# $buf now contains "hello\n"
```

Ανάγνωση από εντολή, μορφή λίστας ώστε τα ορίσματα να μην υπόκεινται σε ερμηνεία μετα-χαρακτήρων κελύφους:

```
open my $ls, "-|", "ls", "-la", $dir
    or die "ls: $!";
while (<$ls>) { print }
close $ls;                    # $? holds the exit status
```

Προσωρινό ανώνυμο αρχείο - περάστε `undef` ως τρίτο όρισμα με τρόπο ανάγνωσης-εγγραφής, και κατόπιν εφαρμόστε `seek` πίσω στην αρχή για να διαβάσετε όσα γράψατε:

```
open my $tmp, "+>", undef or die $!;
print $tmp "scratch data\n";
seek $tmp, 0, 0;
my $line = <$tmp>;
```

Οριακές περιπτώσεις

- **«Μαγικό» άνοιγμα δύο ορισμάτων**: η `open $fh, $name` τιμά τα προπορευόμενα <, >, >>, ένα τελικό |, και αφαιρεί τους περιβάλλοντες λευκούς χαρακτήρες από το `$name`. Μια χαρά για κυριολεκτικό όνομα αρχείου που πληκτρολογήσατε ο ίδιος· **ποτέ** για όνομα αρχείου που προήλθε από είσοδο χρήστη, από το δίκτυο ή από μεταβλητή περιβάλλοντος. Χρησιμοποιήστε τη μορφή τριών ορισμάτων.

- **Δύο ορίσματα με -**: η `open $fh, "<-"` ή `open $fh, "-"` ανοίγει το STDIN· η `open $fh, ">-"` ανοίγει το STDOUT. Τα ισοδύναμα τριών ορισμάτων είναι `open $fh, "<", "-"` και `open $fh, ">", "-"`.
- **Επανάνοιγμα των STDOUT / STDERR**: κλείστε τα πρώτα, και μετά ξανανοίξτε τα προς τον νέο στόχο. Η `open STDOUT, ">", \$buf` ανακατευθύνει κάθε μετέπειτα έξοδο `print STDOUT` στο `$buf`.
- **Bareword filehandles** (`open FH, ...`) είναι καθολικά πακέτου και απενεργοποιούνται υπό `use v5.36` ή νεότερο μέσω του χαρακτηριστικού `bareword_filehandles`. Προτιμήστε ένα λεξιλογικό βαθμωτό.
- **Στρώματα και ${^OPEN}**: οποιοδήποτε ρητό στρώμα, ακόμη και η `:`, απενεργοποιεί τις προεπιλογές της pragma open. Αν βασίζεστε σε προεπιλογή της pragma, παραλείψτε εντελώς την άνω-κάτω τελεία (`<`, όχι `<:`).
- **Σωλήνωση σε εντολή με μετα-χαρακτήρες κελύφους**: η μορφή σωλήνωσης δύο ορισμάτων (`open $fh, "|cat -n | lpr"`) εκτελείται μέσω `/bin/sh -c`. Η μορφή λίστας τριών ορισμάτων (`open $fh, "|-", "cat", "-n"`) όχι, και είναι η ασφαλέστερη προεπιλογή.
- **Βαθμωτές συμβολοσειρές στη μνήμη**: αντιμετωπίζονται ως συμβολοσειρές οκταδικών. Χωρίς αποκοπή, το βαθμωτό δεν επιτρέπεται να περιέχει σημεία κώδικα πάνω από το `0xFF`.
- **Αυτόματο κλείσιμο κατά την έξοδο εμβέλειας**: ένα λεξιλογικό filehandle κλείνει όταν ο μετρητής αναφορών του φτάσει στο μηδέν, αλλά το έμμεσο κλείσιμο δεν αναφέρει σφάλματα. Για αρχεία στα οποία γράψατε, κλείστε τα ρητά και ελέγξτε την τιμή επιστροφής.
- **Η open έναντι της sysopen**: η open περιτυλίγει τη σημασιολογία της `fopen(3)` με στρώματα PerlIO από πάνω. Για άμεσες σημαίες της `open(2)` όπως `O_EXCL` ή `O_NOFOLLOW`, χρησιμοποιήστε την `sysopen`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `close` - απελευθερώστε ένα filehandle· ελέγξτε την τιμή επιστροφής για εγγραφές με buffer και εντολές σωλήνωσης
- `sysopen` - άνοιγμα με ωμές σημαίες της `open(2)` όταν χρειάζεστε `O_EXCL`, `O_NOFOLLOW` ή ανάλογη ακρίβεια
- `binmode` - τοποθετήστε ή αντικαταστήστε στρώματα I/O σε ήδη ανοιχτό handle
- `read` - αναγνώσεις σταθερού μεγέθους byte από το handle που επέστρεψε η open
- `readline` - αναγνώσεις μία εγγραφή τη φορά (αυτό που καλεί το `<$fh>`)
- `fileno` - ο υποκείμενος περιγραφέας αρχείου του ΛΣ, για `flock`, `select` ή `fcntl`
- `$!` - σφάλμα ΛΣ που ορίζεται όταν η open επιστρέφει `undef`

Filehandles, αρχεία, κατάλογοι

opendir

Ανοίγει έναν κατάλογο για ανάγνωση.

Η opendir συνδέει το DIRHANDLE με τον κατάλογο που υποδεικνύει το EXPR ώστε οι εγγραφές του καταλόγου να μπορούν να διαβαστούν με readdir και η θέση να χειριστεί με telldir, seekdir και rewinddir. Όταν τελειώσετε, η closedir απελευθερώνει το handle. Το DIRHANDLE ζει σε διαφορετικό χώρο ονομάτων από τα filehandles - ένα bareword που έχετε ήδη χρησιμοποιήσει με την open είναι ελεύθερο να επαναχρησιμοποιηθεί εδώ, και αντιστρόφως.

Σύνοψη

```
opendir DIRHANDLE, EXPR
opendir my $dh, EXPR
```

Τι επιστρέφεται

1 σε επιτυχία, 0 σε αποτυχία. Σε αποτυχία το $! κρατά τον λόγο - ο κατάλογος δεν υπάρχει, δεν είναι κατάλογος, δεν είναι αναγνώσιμος, ή η διεργασία έχει χτυπήσει το όριο ανοιχτών αρχείων. Ελέγχετε πάντα την επιστροφή· μια σιωπηλή αποτυχία εδώ μετατρέπεται σε μυστηριωδώς κενή readdir αργότερα.

```
opendir my $dh, $path
    or die "opendir $path: $!";
```

Καθολική κατάσταση που επηρεάζει

- $! - τίθεται στη συμβολοσειρά σφάλματος του συστήματος σε αποτυχία, ανέπαφο σε επιτυχία. Αποτυπώστε το προτού κάποια άλλη κλήση συστήματος το αλλοιώσει.

Σημασιολογία handle

Γίνονται δεκτές τρεις μορφές DIRHANDLE, και η επιλογή έχει σημασία:

- **Bareword** - η opendir DIR, $path δημιουργεί ένα καθολικό σε επίπεδο πακέτου dirhandle με όνομα DIR στο τρέχον πακέτο. Παραμένει ανοιχτό μέχρι να καλέσετε ρητά την closedir ή να τερματιστεί το πρόγραμμα. Αποφύγετε σε νέο κώδικα· ο καθολικός χώρος ονομάτων μοιράζεται σε ολόκληρο το πακέτο.
- **Μη ορισμένο βαθμωτό** - η opendir my $dh, $path αυτο-δημιουργεί το $dh ως αναφορά σε ένα φρέσκο ανώνυμο dirhandle. Όταν το $dh βγει από την εμβέλεια και το refcount του πέσει στο μηδέν, το handle κλείνει αυτόματα. Αυτή είναι η μορφή που πρέπει να χρησιμοποιείτε.
- **Έκφραση** - οποιαδήποτε έκφραση η τιμή της οποίας είναι χρήσιμη ως έμμεσο dirhandle (τυπικά ένα πραγματικό όνομα dirhandle που κρατείται σε μεταβλητή). Η ταυτότητα του handle προέρχεται από την τιμή της έκφρασης, όχι από τη συντακτική της μορφή.

Τα dirhandles και τα filehandles μοιράζονται το ίδιο υποκείμενο αντικείμενο I/O: ένα δεδομένο αντικείμενο μπορεί να είναι ανοιχτό ως το ένα ή το άλλο τη φορά, ποτέ και τα

δύο. Το πέρασμα dirhandle στη `read` ή filehandle στη `readdir` είναι σφάλμα χρήσης, όχι σιωπηλή ανάγνωση μηδέν bytes.

Παραδείγματα

Το κανονικό ιδίωμα `opendir` / `readdir` / `closedir`:

```
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;
```

Ανάγνωση κάθε εγγραφής σε λίστα με μία κίνηση. Η `readdir` σε περιβάλλον λίστας επιστρέφει όλες τις υπόλοιπες εγγραφές:

```
opendir my $dh, '.' or die "opendir .: $!";
my @entries = grep { !/^\.\.?\z/ } readdir $dh;
closedir $dh;
```

Βασιστείτε στη λεξιλογική εμβέλεια για τον καθαρισμό - δεν χρειάζεται ρητή `closedir`:

```
sub list_dir {
    my ($path) = @_;
    opendir my $dh, $path or return;
    return grep { !/^\.\.?\z/ } readdir $dh;
    # $dh goes out of scope here, handle is closed
}
```

Χειρισμός ενός καταλόγου που λείπει ή είναι μη αναγνώσιμος χωρίς θάνατο:

```
if (opendir my $dh, $path) {
    while (defined(my $e = readdir $dh)) { ... }
    closedir $dh;
} else {
    warn "skipping $path: $!";
}
```

Επανάληψη από την αρχή ενός καταλόγου με `rewinddir` αντί για επανάνοιγμα:

```
opendir my $dh, $dir or die $!;
my @first_pass  = readdir $dh;
rewinddir $dh;
my @second_pass = readdir $dh;
closedir $dh;
```

Οριακές περιπτώσεις

- **Η αποτυχία θέτει το $!, όχι το $@**. Η opendir αναφέρει μέσω του $! όπως και η υπόλοιπη οικογένεια I/O· το $@ παραμένει ανέπαφο.
- **Επαναχρησιμοποίηση ανοιχτού dirhandle**. Η opendir σε handle που είναι ήδη ανοιχτό κλείνει σιωπηλά τον παλιό κατάλογο πριν ανοίξει τον νέο. Προτιμήστε ρητή closedir όταν ο κώδικας προορίζεται να είναι ευανάγνωστος.
- **Σχετικές διαδρομές**. Το EXPR αναλύεται ως προς τον τρέχοντα κατάλογο εργασίας τη στιγμή της κλήσης. Αν κάνετε chdir αργότερα, μια σχετική διαδρομή που αποτυπώσατε νωρίτερα ενδέχεται να μη σημαίνει πια το ίδιο - το ήδη ανοιγμένο dirhandle εξακολουθεί να δείχνει στο σωστό inode ανεξαρτήτως.
- **Symlinks**. Η opendir ακολουθεί symlinks εξ ορισμού: το άνοιγμα ενός link προς κατάλογο ανοίγει τον στόχο. Οι εγγραφές που επιστρέφει η readdir είναι οι εγγραφές του στόχου, όχι του link.
- **Διάρκεια ζωής bareword έναντι λεξιλογικής**. Ένα bareword dirhandle είναι καθολικό και ζει περισσότερο από το περικλείον μπλοκ· ένα λεξιλογικό `my $dh` καθαρίζεται όταν τελειώνει η εμβέλεια. Η ακούσια ανάμειξη των δύο (`opendir DH, $path` μέσα σε sub) διαρρέει dirhandles σε επαναλαμβανόμενες κλήσεις.
- **Σύγχυση file vs dir handle**. Δεν μπορείτε να κάνετε print σε dirhandle ούτε readdir από filehandle. Το σφάλμα αναδύεται στη μετέπειτα κλήση, όχι στην opendir.
- **glob έναντι opendir**. Η glob επεκτείνει μοτίβα τύπου κελύφους και επιστρέφει διαδρομές· η opendir απαριθμεί ακατέργαστες εγγραφές καταλόγου (συμπεριλαμβανομένων των . και ..) χωρίς ερμηνεία. Χρησιμοποιήστε glob όταν θέλετε αντιστοίχιση μοτίβων· χρησιμοποιήστε opendir όταν θέλετε κάθε εγγραφή ακριβώς όπως την αποθηκεύει το σύστημα αρχείων.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- readdir - διαβάζει την επόμενη εγγραφή (ή όλες τις υπόλοιπες εγγραφές) από dirhandle που άνοιξε με opendir
- closedir - απελευθερώνει ένα dirhandle· σιωπηρή όταν ένα λεξιλογικό $dh βγαίνει από την εμβέλεια
- rewinddir - επαναφέρει τη θέση ανάγνωσης στην αρχή χωρίς επανάνοιγμα
- seekdir - επανατοποθετεί το dirhandle σε ένα σύμβολο που επέστρεψε προηγουμένως η telldir
- telldir - τρέχουσα θέση ανάγνωσης ως αδιαφανές σύμβολο κατάλληλο για seekdir
- glob - εναλλακτική βασισμένη σε μοτίβα όταν θέλετε επέκταση ονομάτων αρχείων τύπου κελύφους αντί για ακατέργαστη απαρίθμηση καταλόγου

Βαθμωτά και συμβολοσειρές

ord

Επιστρέφει το σημείο κώδικα Unicode του πρώτου χαρακτήρα μιας συμβολοσειράς.

Η `ord` παίρνει μια συμβολοσειρά, εξετάζει τον πρώτο της χαρακτήρα, και επιστρέφει το αριθμητικό σημείο κώδικα αυτού του χαρακτήρα ως ακέραιο. Προσέξτε *χαρακτήρας*, όχι byte - αν η συμβολοσειρά είναι συμβολοσειρά ευρέων χαρακτήρων που περιέχει το γράμμα é (U+00E9), η `ord` επιστρέφει 233, ανεξάρτητα από το πόσα bytes καταλαμβάνει το é στη μνήμη. Για την κενή συμβολοσειρά, η `ord` επιστρέφει `0`. Αν παραλειφθεί το EXPR, η `ord` λειτουργεί στο `$_`.

Για την αντίστροφη κατεύθυνση - μετατροπή σημείου κώδικα πίσω σε χαρακτήρα - δείτε την `chr`.

Σύνοψη

```
ord EXPR
ord
ord($str)
```

Τι επιστρέφεται

Μη αρνητικός ακέραιος. Για είσοδο ASCII είναι στο εύρος `0..127`· για είσοδο Latin-1 `0..255`· για γενική συμβολοσειρά Unicode οτιδήποτε έως το `0x10FFFF`. Η `ord` δεν επιστρέφει ποτέ αρνητικό αριθμό και δεν επιστρέφει ποτέ μη ακέραιο. Συμβουλεύεται μόνο τον πρώτο χαρακτήρα του ορίσματος - οι ακόλουθοι χαρακτήρες αγνοούνται. Για να διασχίσετε κάθε σημείο κώδικα μιας συμβολοσειράς, συνδυάστε με `split //` ή `unpack "U*"`.

Καθολική κατάσταση που επηρεάζει

Χωρίς όρισμα, η `ord` διαβάζει το `$_`. Δεν διαβάζει ούτε γράφει καμία άλλη ειδική μεταβλητή. Η ερμηνεία της εισόδου ως χαρακτήρων (όχι bytes) εξαρτάται από το αν το βαθμωτό είναι εσωτερικά σημειωμένο ως UTF-8 - δείτε *Οριακές περιπτώσεις* παρακάτω.

Παραδείγματα

Βασική αναζήτηση ASCII:

```
print ord("A");             # 65
print ord("0");             # 48
print ord("\n");             # 10
```

Η κενή συμβολοσειρά επιστρέφει μηδέν, σύμφωνα με τη σύμβαση «χωρίς πρώτο χαρακτήρα»:

```
print ord("");              # 0
```

Μορφή προεπιλεγμένου ορίσματος μέσα σε βρόχο `while` πάνω από το `$_`:

```
while (<STDIN>) {
    last if ord == 4;       # stop on EOT (Ctrl-D) as first char
    print;
}
```

Αποκωδικοποίηση ευρέος χαρακτήρα. Η `ord` βλέπει τον χαρακτήρα, όχι τα bytes:

```
use utf8;
print ord("é");             # 233        (U+00E9)
print ord("€");             # 8364       (U+20AC)
print ord("😀");            # 128512     (U+1F600)
```

Διάσχιση κάθε σημείου κώδικα σε συμβολοσειρά - χρήσιμο για αποσφαλμάτωση εκπλήξεων κωδικοποίησης:

```
use utf8;
my $s = "héllo";
print join(",", map { ord } split //, $s), "\n";
# 104,233,108,108,111
```

Ζευγαρώστε με την `chr` για no-op πλήρη κύκλο σε οποιονδήποτε μονό χαρακτήρα:

```
my $c = "Z";
print chr(ord($c)) eq $c ? "same\n" : "changed\n";   # same
```

Οριακές περιπτώσεις

- **Σημασιολογία χαρακτήρα έναντι byte**: η `ord` επιστρέφει το σημείο κώδικα του πρώτου *χαρακτήρα*, που για βαθμωτό σημειωμένο ως UTF-8 δεν είναι το ίδιο με το πρώτο *byte*. Η `ord("\x{100}")` επιστρέφει 256, παρότι αυτός ο χαρακτήρας καταλαμβάνει δύο bytes στη μνήμη. Για να εξετάσετε αντ” αυτού το πρώτο byte, επιβάλετε σημασιολογία bytes:

  ```
  use bytes;
  print ord("\x{100}");       # 196   (first UTF-8 byte, 0xC4)
  no bytes;
  print ord("\x{100}");       # 256
  ```

  Καταφύγετε στο `use bytes` μόνο όταν χρειάζεστε γνήσια εξέταση σε επίπεδο bytes· είναι τοπική, χειρουργική pragma.

- **Μη-UTF-8 βαθμωτό που περιέχει υψηλά bytes**: αν το βαθμωτό είναι απλή συμβολοσειρά bytes (χωρίς σημαία UTF-8) και το πρώτο του byte είναι `0xC4`, η `ord` επιστρέφει 196, όχι 256. Το byte ερμηνεύεται ως Latin-1. Το αν ένα δεδομένο βαθμωτό είναι σημειωμένο ως UTF-8 εξαρτάται από τον τρόπο που χτίστηκε - `use utf8` στον πηγαίο κώδικα, `decode_utf8`, στρώμα I/O `:utf8`, `chr` τιμής πάνω από 255, κ.ο.κ.

- **`undef`**: η `ord(undef)` επιστρέφει `0` και, υπό `use warnings`, εκπέμπει `Use of uninitialized value in ord`.

- **Αριθμητικό όρισμα**: η `ord(65)` πρώτα μετατρέπει το 65 σε συμβολοσειρά `"65"`, και μετά παίρνει τον πρώτο χαρακτήρα - οπότε επιστρέφει `ord("6") == 54`, όχι

65. Αυτό ξαφνιάζει· αν έχετε ακέραιο και θέλετε να κάνετε πλήρη κύκλο μέσω χαρακτήρα, χρησιμοποιήστε πρώτα την `chr` ή παρακάμψτε εντελώς την `ord`.

- **Πολυχαρακτηρικό όρισμα**: σημασία έχει μόνο ο πρώτος χαρακτήρας. Η `ord("ABC")` επιστρέφει 65· τα B και C δεν συμβουλεύονται ποτέ.
- **Σημεία κώδικα surrogate και μη-χαρακτήρα**: η `ord` θα επιστρέψει χωρίς πρόβλημα `0xD800..0xDFFF` ή `0xFFFE`/`0xFFFF` αν αυτά εμφανιστούν στην είσοδο. Η Perl δεν αρνείται να τα αποθηκεύσει· η επικύρωση, αν χρειάζεται, είναι δουλειά του καλούντος.
- **Μέγιστη τιμή**: σε τυπικό build, η `ord` μπορεί να επιστρέψει έως `0x7FFFFFFF` (31-bit) για συμβολοσειρά παραγόμενη από `chr` τιμής σε αυτό το εύρος. Το πραγματικό κείμενο Unicode φτάνει στο ανώτατο όριο `0x10FFFF`.
- **Προτεραιότητα**: η μοναδιαία μορφή `ord $x` δεσμεύει στενότερα από το `,` αλλά χαλαρότερα από τους περισσότερους αριθμητικούς τελεστές, οπότε η `ord $x + 1` αναλύεται ως `ord($x) + 1`, όχι ως `ord($x + 1)`. Παρενθεσιοποιήστε όταν αμφιβάλλετε.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `chr` - η αντίστροφη λειτουργία· μετατρέπει σημείο κώδικα πίσω σε συμβολοσειρά ενός χαρακτήρα
- `sprintf` - χρησιμοποιήστε `%c` για μορφοποίηση ακεραίου ως χαρακτήρα του, ή `%x` / `%04x` για να αποτυπώσετε αποτέλεσμα `ord` σε δεκαεξαδικό
- `unpack` - `unpack "U*"` για κάθε σημείο κώδικα συμβολοσειράς· `unpack "C*"` για κάθε byte
- `lc` / `uc` - εφαρμόστε casefold σε χαρακτήρα πριν πάρετε το σημείο κώδικά του όταν θέλετε συγκρίσεις χωρίς διάκριση πεζών-κεφαλαίων
- `$_` - το προεπιλεγμένο όρισμα όταν η `ord` καλείται χωρίς όρισμα

Εμβέλεια

our

Δηλώνει ένα λεξιλογικά οριοθετημένο ψευδώνυμο σε μια μεταβλητή πακέτου.

Η `our` **δεν** δημιουργεί νέα μεταβλητή. Εισάγει στην τρέχουσα λεξιλογική εμβέλεια ένα όνομα που παραπέμπει στη μεταβλητή πακέτου με το ίδιο όνομα στο τρέχον πακέτο. Καθ” όλη τη διάρκεια αυτής της εμβέλειας, το μη ειδικευμένο όνομα είναι νόμιμο υπό το `use strict 'vars'`, και κάθε ανάγνωση ή εγγραφή μέσω αυτού πηγαίνει στην υποκείμενη μεταβλητή πακέτου.

Αν η μεταβλητή πακέτου δεν υπήρχε ακόμη, η `our` ούτε αυτή την επικαλείται - απλώς της δίνει όνομα. Οι μεταβλητές πακέτου έρχονται στην ύπαρξη όταν τους ανατεθεί τιμή για πρώτη φορά, και μια δήλωση `our` καθιστά αυτήν την πρώτη ανάθεση δυνατή χωρίς πλήρως ειδικευμένο όνομα.

Σύνοψη

```
our $x;
our ($a, $b, $c);
our @EXPORT_OK = qw(foo bar);
our $VERSION : shared = '1.00';
```

Τι επιστρέφεται

Η `our` επιστρέφει τη λίστα των μεταβλητών της, οπότε μπορεί να εμφανιστεί σε οποιαδήποτε θέση έκφρασης μπορεί να εμφανιστεί η `my`:

```
(our $x, our $y) = (1, 2);
our $count = 0;
```

Ως δήλωση δεν έχει χωριστή τιμή επιστροφής πέραν αυτής - το χρήσιμο αποτέλεσμα είναι η παρενέργεια εμβέλειας, όχι η τιμή.

Εμβέλεια και τι ακριβώς είναι το ψευδώνυμο

Το ψευδώνυμο ζει στο **περικλείον λεξιλογικό μπλοκ**, ακριβώς όπως οι `my` και `state`. Τερματίζει στο άγκιστρο κλεισίματος αυτού του μπλοκ (ή στο τέλος του αρχείου για δήλωση ανώτατου επιπέδου). Έξω από την εμβέλεια, το μη ειδικευμένο όνομα δεν είναι πλέον νόμιμο υπό το `use strict 'vars'` - η ίδια η μεταβλητή πακέτου παραμένει ανέπαφη και εξακολουθεί να είναι προσπελάσιμη με το πλήρως ειδικευμένο όνομά της.

```
package Foo;
use strict;

$Foo::foo = 23;

{
    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
```

Επειδή η `our` αποτελεί ψευδώνυμο σε μεταβλητή **πακέτου**, το πακέτο καθορίζεται στο σημείο της δήλωσης, όχι στο σημείο της χρήσης. Η αλλαγή πακέτου με `package` στο μέσο της εμβέλειας δεν επανασυνδέει το ψευδώνυμο:

```
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 έναντι my

Η our και η my μοιάζουν στο χαρτί και συχνά συγχέονται. Κάνουν αντίθετα πράγματα.

- Η my δημιουργεί μια φρέσκια **ιδιωτική** μεταβλητή που ζει μόνο μέσα στη λεξιλογική εμβέλειά της. Δεν έχει όνομα στον πίνακα συμβόλων κανενός πακέτου και τίποτε έξω από την εμβέλεια δεν μπορεί να την προσπελάσει.
- Η our δημιουργεί ένα λεξιλογικά οριοθετημένο **ψευδώνυμο** σε μια **δημόσια** μεταβλητή πακέτου. Η μεταβλητή ζει στον πίνακα συμβόλων· οποιοσδήποτε άλλος κώδικας γνωρίζει το πλήρως ειδικευμένο όνομά της μπορεί να τη διαβάσει ή να τη γράψει.

Χρησιμοποιήστε my ως προεπιλογή. Καταφύγετε στην our όταν θέλετε σκόπιμα καθολική κατάσταση πακέτου: λίστες exporter αρθρώματος (our @EXPORT, our @EXPORT_OK), our $VERSION, ένα hash διαμόρφωσης σε επίπεδο πακέτου που άλλος κώδικας αναμένεται να πειράξει, ή το παραδοσιακό our @ISA για κληρονομικότητα.

```
package My::Module;

our $VERSION   = '1.23';
our @EXPORT_OK = qw(frobnicate);
our @ISA       = ('Exporter');
```

Κάθε ένα από αυτά τα ονόματα είναι ψευδώνυμο στα $My::Module::VERSION, @My::Module::EXPORT_OK, @My::Module::ISA - ακριβώς αυτό που αναζητούν τα Exporter και UNIVERSAL::isa.

our έναντι local

Η local και η our συχνά εμφανίζονται μαζί και είναι συμπληρωματικές, όχι εναλλακτικές.

- Η our κάνει τη μεταβλητή πακέτου προσπελάσιμη με το μη ειδικευμένο όνομά της εντός λεξιλογικής εμβέλειας. **Δεν** αποθηκεύει ούτε επαναφέρει την τιμή.
- Η local αποθηκεύει την τρέχουσα τιμή μιας μεταβλητής πακέτου και φροντίζει να επαναφερθεί όταν τερματίσει η περικλείουσα δυναμική εμβέλεια. **Δεν** καθιστά το όνομα νόμιμο υπό το use strict 'vars'.

Το ιδιωματικό ζευγάρωμα - προσωρινή αλλαγή μιας καθολικής πακέτου μέσα σε υπορουτίνα - χρησιμοποιεί και τις δύο:

```
sub with_verbose {
    our $verbose;        # make the unqualified name legal here
    local $verbose = 1;  # save old value, restore on return
    do_work();
}
```

our έναντι state

Η state είναι η άλλη λεξιλογική δήλωση. Δημιουργεί μια ιδιωτική μεταβλητή της οποίας η τιμή διατηρείται μεταξύ κλήσεων στην περικλείουσα υπορουτίνα. Η our δεν μοιράζεται τίποτα από αυτόν τον μηχανισμό - η μεταβλητή της είναι δημόσια και δεν

έχει σημασιολογία ανά κλήση. Επιλέξτε `state` για memoisation ανά υπορουτίνα, `our` για καθολική κατάσταση πακέτου.

Η εμβέλεια της ίδιας της δήλωσης

Όπως οι `my`, `state` και `local`, η `our` μπορεί να εμφανιστεί οπουδήποτε μπορεί μια έκφραση (έξω από παρεμβολή σε συμβολοσειρά). Η δήλωση τίθεται σε ισχύ **άμεσα** - ακόμη και μέσα στην ίδια εντολή - οπότε η μεταβλητή ικανοποιεί το `use strict 'vars'` από τη δήλωση και έπειτα. Αλλά η δήλωση **δεν** καλύπτει αναδρομικά προηγούμενες χρήσεις του ίδιου ονόματος στην ίδια εντολή:

```
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)
```

Ο κανόνας: κάθε εμφάνιση του ονόματος επιλύεται με βάση τη δική της θέση στην εντολή. Μετά τη δήλωση, το όνομα είναι το ψευδώνυμο· πριν από αυτή, το όνομα σημαίνει ό,τι σήμαινε προηγουμένως.

Η δήλωση επίσης σας επιτρέπει να αναφερθείτε στη μεταβλητή πακέτου στη δεξιά πλευρά του ίδιου του αρχικοποιητή της, κάτι που η `my` δεν επιτρέπει:

```
my  $foo = $foo;   # error under strict: undeclared $foo on RHS
our $foo = $foo;   # fine: RHS $foo is the package variable
```

Πολλαπλές μεταβλητές και placeholders

Περισσότερες από μία μεταβλητές απαιτούν παρενθέσεις:

```
our ($bar, $baz);
```

Μέσα σε λίστα με παρενθέσεις, το `undef` είναι έγκυρο placeholder που απορρίπτει την αντίστοιχη τιμή μιας ανάθεσης λίστας:

```
our ( undef, $min, $hour ) = localtime;
```

Επαναλαμβανόμενες δηλώσεις

Πολλαπλές δηλώσεις `our` με το ίδιο όνομα στην ίδια λεξιλογική εμβέλεια επιτρέπονται όταν βρίσκονται σε **διαφορετικά** πακέτα - κάθε μία επανασυνδέει το ψευδώνυμο στη μεταβλητή του δικού της πακέτου:

```
use warnings;

package Foo;
our $bar;            # declares $Foo::bar for the rest of this scope
$bar = 20;

package Bar;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
our $bar = 30;        # declares $Bar::bar - aliases the Bar variable
print $bar;           # 30
```

Μια δεύτερη δήλωση our για το ίδιο όνομα **στο ίδιο πακέτο και την ίδια εμβέλεια** είναι απλώς πλεοναστική - επανασυνδέει στην ίδια μεταβλητή. Υπό το `use warnings` η Perl την αναφέρει με τον τρόπο που αναφέρει επαναλαμβανόμενη `my`, αλλά δεν υπάρχει φρέσκια μεταβλητή και τίποτα δεν επισκιάζεται:

```
use warnings;
package Bar;
our $bar = 30;
our $bar;             # warning, but still the same $Bar::bar
print $bar;           # 30
```

Χαρακτηριστικά και TYPE

Η our δέχεται την πλήρη συντακτική μορφή δήλωσης της `my`:

```
our $x : shared;
our MyType $obj;
our $VERSION : shared = '1.00';
```

Το TYPE είναι αυτή τη στιγμή συνδεδεμένο με την pragma `fields`. Τα χαρακτηριστικά επεξεργάζονται από την pragma `attributes` (και, από την 5.8.0, από το `Attribute::Handlers`). Δείτε την pragma `attributes` για το σύνολο των τυπικών χαρακτηριστικών. Ο συνηθισμένος κώδικας επιπέδου χρήστη σπάνια χρησιμοποιεί κάποιο από τα δύο.

Παραδείγματα

Πρόλογος αρθρώματος - η πιο συχνή κατά κράτος χρήση:

```
package My::Thing;
use strict;
use warnings;

our $VERSION   = '0.01';
our @ISA       = ('Exporter');
our @EXPORT_OK = qw(thingify);
```

Δήλωση και ανάθεση σε ένα βήμα:

```
package Cfg;
our $timeout = 30;    # same as $Cfg::timeout = 30
```

Προσωρινή υπερκάλυψη μιας καθολικής πακέτου μέσα σε μια κλήση:

```
sub quiet_run {
    our $verbose;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    local $verbose = 0;
    run_one_job();
}
```

Ψευδώνυμο μεταξύ πακέτων - δήλωση στο Foo, χρήση στο 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
```

Λίστα με παρενθέσεις και placeholder:

```
our ( undef, $min, $hour, undef, $mon, $year ) = localtime;
```

Οριακές περιπτώσεις

- **Η our δεν δημιουργεί τη μεταβλητή.** Δηλώνει ψευδώνυμο. Η μεταβλητή πακέτου υπάρχει τη στιγμή που οτιδήποτε της αναθέτει τιμή (ή λαμβάνει αναφορά σε αυτήν)· από μόνο του το `our $x;` δεν εκχωρεί νέα θέση στον πίνακα συμβόλων.
- **Το ψευδώνυμο είναι λεξιλογικό, η μεταβλητή δεν είναι.** Η έξοδος από την εμβέλεια της `our` τερματίζει το ψευδώνυμο, όχι τη μεταβλητή. Μια άλλη εμβέλεια σε άλλο αρχείο που δηλώνει `our $x;` στο ίδιο πακέτο βλέπει τον ίδιο αποθηκευτικό χώρο.
- **Επίλυση στην ίδια εντολή.** Μόνο οι εμφανίσεις του ονόματος **μετά** τη δήλωση `our` σε μια εντολή χρησιμοποιούν το ψευδώνυμο. Οι προηγούμενες εμφανίσεις διατηρούν την προηγούμενη σύνδεσή τους (δείτε το παράδειγμα `foo()` παραπάνω).
- **Πακέτο, όχι τρέχον πακέτο κατά τη χρήση.** Το ψευδώνυμο είναι συνδεδεμένο με το πακέτο που ισχύει στο σημείο της δήλωσης. Η αλλαγή πακέτου αργότερα δεν το ανακατευθύνει.
- **Δεν είναι το ίδιο με το `use vars`.** Το `use vars '$x'` καθιστά το `$x` νόμιμο υπό το `strict` για το υπόλοιπο του **πακέτου**, ανεξάρτητα από εμβέλεια. Το `our $x` είναι λεξιλογικό. Προτιμήστε `our`· το `use vars` είναι παλαιό.
- **Δεν είναι pragma εμβέλειας αρχείου.** Ένα `our $x;` ανώτατου επιπέδου σε αρχείο καλύπτει μέχρι το τέλος του αρχείου, που είναι λεξιλογική εμβέλεια - αλλά το `require` σε αυτό το αρχείο από αλλού δεν επεκτείνει το ψευδώνυμο στην εμβέλεια του αρχείου που κάνει το require.
- **Η σημασιολογία των χαρακτηριστικών και του TYPE εξακολουθεί να εξελίσσεται.** Οτιδήποτε πέρα από το `: shared` σε κώδικα χρήστη είναι σπάνιο· διαβάστε την τεκμηρίωση των pragmas `attributes` και `fields` πριν βασιστείτε σε αυτά.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `my` - δηλώνει μια ιδιωτική λεξιλογική μεταβλητή· η προεπιλεγμένη επιλογή όταν θέλετε νέα μεταβλητή
- `local` - οριοθετεί δυναμικά μια προσωρινή τιμή για μεταβλητή πακέτου· συνδυάζεται με την `our` όταν χρειάζεστε ταυτόχρονα νομιμότητα υπό strict και αποθήκευση/επαναφορά
- `state` - δηλώνει ιδιωτική λεξιλογική μεταβλητή που διατηρείται μεταξύ κλήσεων στην περικλείουσα υπορουτίνα
- `package` - θέτει το τρέχον πακέτο, στο οποίο και αποτελεί ψευδώνυμο η `our`
- `use` - φορτωτής αρθρωμάτων κατά τη μεταγλώττιση· το `use strict 'vars'` είναι η pragma που καθιστά εξαρχής αναγκαία την `our`
- `undef` - έγκυρο placeholder μέσα σε λίστα `our` σε παρενθέσεις για απόρριψη μιας τιμής κατά την ανάθεση

Βαθμωτά και συμβολοσειρές · Δεδομένα σταθερού μήκους

pack

Μετατρέπει μια λίστα τιμών Perl σε δυαδική συμβολοσειρά σύμφωνα με ένα πρότυπο.

Η `pack` είναι ο χαμηλού επιπέδου σειριοποιητής: περιγράφετε τη διάταξη bytes με ένα `TEMPLATE`, της παραδίδετε μια `LIST` τιμών, και επιστρέφει ένα βαθμωτό του οποίου οι χαρακτήρες είναι η συνενωμένη κωδικοποίηση σε επίπεδο μηχανής αυτών των τιμών. Είναι το αντίστροφο της `unpack` και ο καθιερωμένος τρόπος για να κατασκευάζετε εγγραφές σταθερού πλάτους, πρωτόκολλα επί της γραμμής, διατάξεις struct της C, ωμές διευθύνσεις IP για `sockaddr_in`, και κάθε άλλο φορτίο επιπέδου byte που η Perl δεν μοντελοποιεί ως τύπο πρώτης τάξης.

This page is a **directive reference**. For a narrative introduction

- Αυτή η σελίδα είναι μια **αναφορά οδηγιών**. Για μια αφηγηματική εισαγωγή - «έχω αυτό το πρωτόκολλο επί της γραμμής, πώς το αναλύω;» - ξεκινήστε από τον *οδηγό pack/unpack*.

Σύνοψη

```
my $bytes = pack TEMPLATE, LIST;
my $ip    = pack "C4",     split /\./, "192.168.1.1";
my $rec   = pack "Z8 Z8 L", $user, $host, $ts;
```

Τι επιστρέφεται

Ένα απλό βαθμωτό Perl που κρατά τα πακεταρισμένα bytes. Το μήκος του είναι το άθροισμα των πλατών κάθε οδηγίας του προτύπου· το περιεχόμενό του είναι δυαδικό και ενδέχεται να περιέχει ενσωματωμένα bytes NUL. Μεταχειριστείτε το ως συμβολοσειρά bytes, όχι ως κείμενο - γράφοντάς το σε handle με στρώμα `:utf8` ή `:encoding(…)` θα το επανακωδικοποιήσει. Χρησιμοποιήστε `binmode $fh` ή ανοίξτε με `">:raw"` πριν την έκδοση.

Προεπιλεγμένα το αποτέλεσμα είναι σε **κατάσταση χαρακτήρων** (`C0`). Ένα πρότυπο που ξεκινά με `U`, ή που μεταβαίνει σε `U0` στη μέση του προτύπου, παράγει αντ” αυτού μια συμβολοσειρά Unicode κωδικοποιημένη σε UTF-8. Μη χρησιμοποιείτε αυτό ως υποκατάστατο του αρθρώματος Encode.

Σύνταξη προτύπου

Ένα TEMPLATE είναι μια ακολουθία από **οδηγίες**. Κάθε οδηγία είναι ένα γράμμα ASCII, προαιρετικά ακολουθούμενο από:

- μια **μετρητή επανάληψης** - έναν δεκαδικό ακέραιο, `*`, ή `[…]`
- έναν ή περισσότερους **τροποποιητές** - `!`, `<`, `>`
- μια **ομάδα** - (…) συγκεντρώνει οδηγίες ώστε ένας μετρητής επανάληψης ή τροποποιητής endianness να εφαρμόζεται στο σύνολο

Οι λευκοί χαρακτήρες μεταξύ οδηγιών αγνοούνται. Το # εισάγει σχόλιο που τρέχει μέχρι το τέλος της γραμμής - η ίδια σύμβαση με τον πηγαίο κώδικα Perl.

Πίνακας οδηγιών

Όλες οι οδηγίες της `pack` σε ένα μέρος. Το **W** είναι το πλάτος σε bytes ανά καταναλωμένο βαθμωτό (ένα `C` καταναλώνει μία τιμή και παράγει ένα byte· ένα `a3` καταναλώνει μία τιμή και παράγει τρία bytes). Στήλη **Endian**: *native* σημαίνει η σειρά bytes του CPU· *big* / *little* είναι σταθερά ανεξάρτητα από τον υπολογιστή.

Οδηγία	Καταναλών	W	Προσημασ	Endian	Τροποποιητ	Σημειώσεις
a	1 συμβολοσει	μετρητής	-	-	-	Συμπληρώνεται με NUL ως το πλάτος, αποκόπτεται αν είναι μακρύτερη
A	1 συμβολοσει	μετρητής	-	-	-	Συμπληρώνεται με κενά ως το πλάτος

συνέχεια στην επόμενη σελίδα

Πίνακας 1 – συνεχίζεται από την προηγούμενη σελίδα

Οδηγία	Καταναλών	W	Προσημασμ	Endian	Τροποποιητ	Σημειώσεις
Z	1 συμβολοσει	μετρητής	-	-	-	Τερματίζεται με NUL· το Z* πάντα προσαρτά τελικό NUL
b	1 συμβολοσει	μετρητής/8	-	-	-	Συμβολοσειρά bits, LSB πρώτο μέσα σε κάθε byte
B	1 συμβολοσει	μετρητής/8	-	-	-	Συμβολοσειρά bits, MSB πρώτο μέσα σε κάθε byte
h	1 συμβολοσει	μετρητής/2	-	-	-	Δεκαεξαδική συμβολοσειρά, το κάτω nybble πρώτο
H	1 συμβολοσει	μετρητής/2	-	-	-	Δεκαεξαδική συμβολοσειρά, το άνω nybble πρώτο
c	1 ακέραιος	1	ναι	-	-	Προσημασμένο char
C	1 ακέραιος	1	όχι	-	-	Μη προσημασμένο char (οκτάδα)
W	1 ακέραιος	1	όχι	-	-	Μη προσημασμένο char· επιτρέπει τιμές πάνω από 255 σε κατάσταση U0
s	1 ακέραιος	2	ναι	native	! < >	s! = native short
S	1 ακέραιος	2	όχι	native	! < >	S! = native unsigned short

συνέχεια στην επόμενη σελίδα

Πίνακας 1 – συνεχίζεται από την προηγούμενη σελίδα

Οδηγία	Καταναλών	W	Προσημασ	Endian	Τροποποιητ	Σημειώσεις
`l`	1 ακέραιος	4	ναι	native	! < >	`l!` = native `long`
`L`	1 ακέραιος	4	όχι	native	! < >	`L!` = native `unsigned long`
`i`	1 ακέραιος	native	ναι	native	! < >	`sizeof(int)`· τουλάχιστον 32 bits
`I`	1 ακέραιος	native	όχι	native	! < >	`sizeof(unsigned int)`
`q`	1 ακέραιος	8	ναι	native	< >	Απαιτεί Perl με 64-bit ακεραίους
`Q`	1 ακέραιος	8	όχι	native	< >	Απαιτεί Perl με 64-bit ακεραίους
`n`	1 ακέραιος	2	όχι	big	!	Φορητή σειρά δικτύου· `n!` = προσημασμένο
`N`	1 ακέραιος	4	όχι	big	!	Φορητή σειρά δικτύου· `N!` = προσημασμένο
`v`	1 ακέραιος	2	όχι	little	!	Σειρά «VAX»· `v!` = προσημασμένο
`V`	1 ακέραιος	4	όχι	little	!	Σειρά «VAX»· `V!` = προσημασμένο
`j`	1 ακέραιος	μέγεθος IV	ναι	native	< >	Εσωτερικός προσημασμένος ακέραιος της Perl
`J`	1 ακέραιος	μέγεθος UV	όχι	native	< >	Εσωτερικός μη προσημασμένος ακέραιος της Perl

συνέχεια στην επόμενη σελίδα

Πίνακας 1 – συνεχίζεται από την προηγούμενη σελίδα

Οδηγία	Καταναλών	W	Προσημασ	Endian	Τροποποιητ	Σημειώσεις
f	1 αριθμός	4	-	native	< >	IEEE 754 απλής ακρίβειας
d	1 αριθμός	8	-	native	< >	IEEE 754 διπλής ακρίβειας
F	1 αριθμός	μέγεθος NV	-	native	< >	Εσωτερικός float της Perl (NV)
D	1 αριθμός	ποικίλλει	-	native	< >	Long double· η μορφή ποικίλλει ανά πλατφόρμα
p	1 συμβολοσει / undef	ptr	-	native	< >	Δείκτης σε συμβολοσειράτερματισμένη με NUL· η undef → null pointer
P	1 συμβολοσει / undef	ptr	-	native	< >	Δείκτης σε ενταμιευτή σταθερού μήκους· μετρητής = μήκος ενταμιευτή
u	1 συμβολοσει	ποικίλλει	-	-	-	Κωδικοποίηση uuencode· μετρητής = μέγιστα bytes ανά γραμμή εξόδου (προεπιλογή 45)
U	1 σημείο κώδικα	ποικίλλει	-	-	-	Αριθμός χαρακτήρα Unicode· κωδικοποιείται σε UTF-8 σε κατάσταση U0

συνέχεια στην επόμενη σελίδα

Πίνακας 1 – συνεχίζεται από την προηγούμενη σελίδα

Οδηγία	Καταναλών	W	Προσημασ	Endian	Τροποποιητ	Σημειώσεις
w	1 ακέραιος ⩾ 0	ποικίλλει	όχι	-	-	Ακέραιος συμπιεσμένος κατά BER, big-endian βάσης-128
x	τίποτα	1	-	-	!	Εισάγει ένα NUL· το x!N ευθυγραμμίζει σε πολλαπλάσιο του N
X	τίποτα	−1	-	-	!	Επιστροφή κατά ένα byte· το X!N ευθυγραμμίζει προς τα πίσω
@	τίποτα	απόλυτο	-	-	!	Συμπληρώνει με μηδενικά ή αποκόπτει στη θέση N μέσα στην ομάδα
.	1 ακέραιος	απόλυτο	-	-	!	Συμπληρώνει με μηδενικά ή αποκόπτει στη θέση που δίνει η τιμή
(…)	-	-	-	-	< > !	Ομάδα: ο μετρητής επανάληψης και η endianness διαδίδονται προς τα μέσα

συνέχεια στην επόμενη σελίδα

Πίνακας 1 – συνεχίζεται από την προηγούμενη σελίδα

Οδηγία	Καταναλών	W	Προσημασ	Endian	Τροποποιητ	Σημειώσεις
/	-	-	-	-	-	**Μόνο για unpack.** Στην pack, χρησιμοποιήστε *length-item*/*item*

Οι τροποποιητές:

Τροπ	Αποτέλεσμα
!	Στα `s` / `S` / `l` / `L` / `i` / `I`: χρησιμοποιεί το native `sizeof(…)` αντί σταθερού πλάτους. Στα `x` / `X`: μετατρέπονται σε εντολές ευθυγράμμισης. Στα `n` / `N` / `v` / `V`: ερμηνεύονται ως προσημασμένα. Στα `@` / `.`: μετρά bytes της εσωτερικής αναπαράστασης, όχι χαρακτήρες.
>	Εξαναγκάζει σε **big-endian** σειρά bytes («το μεγάλο άκρο αγγίζει την κατασκευή»).
<	Εξαναγκάζει σε **little-endian** σειρά bytes.

Εφαρμοζόμενα σε ομάδα, τα < / > διαδίδονται σε κάθε οδηγία με σειρά bytes μέσα στην ομάδα και αγνοούνται σιωπηλά από τις οδηγίες που δεν τα δέχονται.

Μετρητές επανάληψης

Ένα γράμμα οδηγίας μπορεί να ακολουθείται από:

- έναν **αριθμό** `N` - εφαρμόζει την οδηγία τόσες φορές, καταναλώνοντας `N` τιμές από τη `LIST`
- `*` - καταναλώνει όλες τις υπόλοιπες τιμές· για τα `x` / `X` / `@` αυτό ισοδυναμεί με `0`· για το `u` επιλέγει την προεπιλογή των 45
- `[N]` - ισοδύναμο με γυμνό `N`
- `[template]` - ο μετρητής επανάληψης είναι το μήκος σε bytes του πακεταρισμένου *template*. Το `x[L]` παρακάμπτει τόσα bytes όσα ένα πακεταρισμένο long· το `x![d]` ευθυγραμμίζει σε όριο double

Οι οδηγίες συμβολοσειρών και bit/nybble μεταχειρίζονται τον μετρητή ως **πλάτος μίας μόνο τιμής**, όχι ως πλήθος τιμών: το `pack "A4", "abcdef"` παράγει `"abcd"`, όχι τέσσερα αντίγραφα του `"abcdef"`.

Ομαδοποίηση

Οι παρενθέσεις ομαδοποιούν οδηγίες. Μια ομάδα μπορεί να πάρει μετρητή επανάληψης ή τροποποιητή endianness:

```
pack "(sl)<", -42, 4711     # same as "s<l<", -42, 4711
pack "(CCS)*", @triplets    # repeat group for every triplet
```

Σε κάθε επανάληψη μιας ομάδας, το @ επανεκκινεί στο 0. Αυτό μπορεί να σας εκπλήξει:

- Το `pack '@1A((@2A)@3A)', qw(X Y Z)` παράγει `"\0X\0\0YZ"`.

Φορτία με πρόθεμα μήκους (/)

Γράψτε `length-item/sequence-item`. Το μήκος υπολογίζεται από την τιμή της ακολουθίας και πακετάρεται σύμφωνα με το *length-item*· στη συνέχεια η ίδια η ακολουθία πακετάρεται σύμφωνα με το *sequence-item*:

```
my $msg = pack "n/a*", "hello, world";
# "\x00\x0chello, world" - 16-bit big-endian length, then the bytes
```

Η οδηγία μήκους μπορεί να είναι οποιαδήποτε ακέραια οδηγία (n, N, w, C, ...) ή ακόμη και οδηγία συμβολοσειράς (A4, Z*) όταν θέλετε το μήκος γραμμένο ως ASCII.

Παραδείγματα

Κατασκευή εγγραφής σταθερού πλάτους με δύο συμβολοσειρές τερματισμένες με NUL και ένα 32-bit native χρονοσφραγίδα:

```
my $rec = pack "Z8 Z8 L", "alice", "server01", $now;
# 8 + 8 + 4 = 20 bytes
```

Φορητή σειρά bytes δικτύου - προτιμήστε n / N / v / V έναντι οδηγιών native πλάτους όταν τα bytes εγκαταλείπουν τη μηχανή:

```
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
```

Ευθυγραμμίστε ένα πεδίο μέσα σε ένα C struct. Το x![d] εισάγει ακριβώς τόσα bytes NUL ώστε να φτάσει στο επόμενο πολλαπλάσιο του πλάτους ενός double:

```
# struct { char c; double d; char cc[2]; }
my $s = pack "c x![d] d c2", $c, $d, $c1, $c2;
```

Πλήρης κύκλος μέσα από δεκαεξαδική συμβολοσειρά:

```
my $raw = pack "H*", "deadbeef";    # "\xde\xad\xbe\xef"
my $hex = unpack "H*", $raw;        # "deadbeef"
```

Οριακές περιπτώσεις

- **Πολύ λίγες τιμές**: οι τιμές που λείπουν αντιμετωπίζονται ως "". Το `pack "A4 A4", "hi"` αποδίδει `"hi \0\0\0\0"`, όχι μοιραίο σφάλμα.
- **Πάρα πολλές τιμές**: οι επιπλέον αγνοούνται σιωπηλά.
- **a έναντι A έναντι Z**: και τα τρία συμπληρώνουν σε ακριβές πλάτος. Το a συμπληρώνει με NUL, το A συμπληρώνει με κενά, το Z συμπληρώνει με NUL **και**

εγγυάται τελικό NUL - οπότε το Z8 κωδικοποιεί το πολύ 7 bytes δεδομένων συν τον τερματιστή.

- **Πλάτος χαρακτήρων έναντι πλάτους bytes** - η μοναδική πιο κοινή παγίδα. Οι μετρητές για τα a / A / Z και οι μετατοπίσεις για τα @ / . είναι σε **χαρακτήρες της πακεταρισμένης συμβολοσειράς**, όχι σε bytes. Σε κατάσταση C0 ένας χαρακτήρας είναι ένα byte· σε κατάσταση U0 ένας χαρακτήρας μπορεί να καλύπτει πολλαπλά bytes UTF-8. Χρησιμοποιήστε τον τροποποιητή ! στα @ / . όταν χρειάζεστε μετατοπίσεις σε bytes ανεξάρτητα από την κατάσταση.
- **Endianness στα s / S / l / L / i / I / q / Q**: μόνο native, όχι φορητή. Χρησιμοποιήστε n / N / v / V ή εφαρμόστε ρητά > / <. Τα f / d είναι ομοίως native· από μόνο του το IEEE 754 δεν καθορίζει endianness.
- **Inf / NaN πακεταρισμένα ως ακέραιοι**: μοιραίο σφάλμα. Δεν υπάρχει λογική αντιστοίχιση.
- **q / Q σε Perl μη 64-bit**: εγείρει εξαίρεση.
- **Τα p και P αποτυπώνουν δείκτες** στη μνήμη του καλούντος. Το αντικείμενο αναφοράς πρέπει να παραμείνει ζωντανό μέχρι να καταναλωθεί η πακεταρισμένη συμβολοσειρά - μια προσωρινή συμβολοσειρά που διαβιβάστηκε στο p μπορεί να αποδεσμευτεί πριν τη διαβάσετε πίσω. Αποφύγετέ τα εκτός κώδικα XS.
- **Ομαδοποίηση και @**: η τοποθέτηση με @ ξεκινά ξανά από το 0 μέσα σε κάθε επανάληψη μιας ομάδας. Το `pack '@1A((@2A)@3A)', qw(X Y Z)` παράγει `"\0X\0\0YZ"`, όχι αυτό που υποδηλώνει μια απλοϊκή ανάγνωση.
- **Απώλεια κατά τον κύκλο κινητής υποδιαστολής**: η Perl αποθηκεύει αριθμούς ως double, οπότε γενικά `unpack("f", pack("f", $x))` δεν ισούται με την $x - το πακετάρισμα μέσα από απλή ακρίβεια αποκόπτει.
- **Εγγραφή πακεταρισμένων bytes σε handle κειμένου**: handle με `:utf8` ή `:encoding(…)` θα επανακωδικοποιήσει τα bytes. Χρησιμοποιήστε `binmode $fh` ή ανοίξτε με `">:raw"`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- unpack - η αντίστροφη λειτουργία· ίδια γλώσσα προτύπων
- sprintf - μορφοποιημένη έξοδος κειμένου αντί για δυαδικά bytes
- vec - πρόσβαση επιπέδου bit σε συμβολοσειρά χωρίς πρότυπο
- chr / ord - μετατροπές μονού χαρακτήρα
- syswrite - εγγραφή πακεταρισμένης συμβολοσειράς bytes χωρίς εκπλήξεις PerlIO
- *Οδηγός pack/unpack* - προσανατολισμένη σε εργασίες περιδιάβαση με λυμένα παραδείγματα πρωτοκόλλων και μορφών αρχείων

Εμβέλεια · Αρθρώματα · Κλάσεις και αντικειμενοστρέφεια

package

Δηλώνει τον χώρο ονομάτων κατά τη μεταγλώττιση για τις δηλώσεις που ακολουθούν.

Η package λέει στον μεταγλωττιστή ποιον πίνακα συμβόλων να χρησιμοποιήσει όταν επιλύει μη προσδιορισμένα ονόματα στον κώδικα που πρόκειται να αναλύσει. **Δεν** δημιουργεί πακέτο με οποιαδήποτε σημασία σε χρόνο εκτέλεσης - τα πακέτα έρχονται στην ύπαρξη την πρώτη φορά που αποθηκεύεται κάτι μέσα τους. Αυτό που πραγματικά κάνει μια εντολή package είναι να αναπροσανατολίσει τον μεταγλωττιστή: από εδώ και στο εξής, το `$foo` σημαίνει `$NAMESPACE::foo`, το `&bar` σημαίνει `&NAMESPACE::bar`, και ούτω καθεξής, μέχρι την επόμενη εντολή package ή το τέλος της περικλείουσας εμβέλειας.

Σύνοψη

```
package NAME;
package NAME VERSION;
package NAME     { ... }
package NAME VER { ... }
```

Τι επιστρέφεται

Η package είναι οδηγία μεταγλώττισης, όχι έκφραση. Δεν έχει χρήσιμη τιμή χρόνου εκτέλεσης - αντιμετωπίστε την ως εντολή. Η επίδρασή της είναι στον αναλυτή: όλα τα μετέπειτα μη προσδιορισμένα αναγνωριστικά επιλύονται στον πίνακα συμβόλων του NAME μέχρι να τερματιστεί η εμβέλεια.

Η μορφή με VERSION έχει μία ορατή παρενέργεια: ορίζει το `$NAME::VERSION` σε ένα αντικείμενο `version` χτισμένο από το VERSION. Αυτό συμβαίνει ακριβώς μία φορά ανά εντολή package NAME VERSION, κατά τη μεταγλώττιση.

Καθολική κατάσταση που επηρεάζει

- **Το τρέχον πακέτο μεταγλώττισης.** Κάθε εντολή package το αντικαθιστά για το υπόλοιπο της εμβέλειας. Το `__PACKAGE__` επιστρέφει την τρέχουσα τιμή σε οποιοδήποτε σημείο της πηγής.
- `$NAME::VERSION` - ορίζεται όταν παρέχεται το όρισμα VERSION. Ορίστε το μόνο μία φορά ανά πακέτο.
- Ο στόχος πίνακας συμβόλων (stash) δημιουργείται κατά την πρώτη χρήση. Το stash ενός πακέτου είναι το ίδιο ένα hash με όνομα `%NAME::` (οπότε του `main` είναι το `%main::`), με κλειδιά τα μη προσδιορισμένα ονόματα συμβόλων που κρατά. Τίποτα δεν συμπληρώνεται από την ίδια την εντολή package· μεταγενέστερες δηλώσεις (`sub`, `our`, αναθέσεις στο `$NAME::foo`, κ.λπ.) το γεμίζουν.

Οι τέσσερις μορφές

package NAMESPACE

Αλλάζει το πακέτο μεταγλώττισης σε NAMESPACE από αυτό το σημείο μέχρι το τέλος της τρέχουσας εμβέλειας (περικλείον μπλοκ, αρχείο ή `eval`). Αυτή είναι η κλασική μορφή

«ένα πακέτο ανά αρχείο»:

```
package My::Thing;

sub new { ... }          # defines &My::Thing::new
our $count = 0;          # defines $My::Thing::count

1;
```

package NAMESPACE VERSION

Ίδια εμβέλεια με τη γυμνή μορφή, επιπλέον ορίζει το `$NAMESPACE::VERSION` σε ένα αντικείμενο `version`. Το VERSION πρέπει να είναι **αυστηρό** version literal: είτε ένας θετικός δεκαδικός (`1.23`, `0.001`) είτε ένα v-string με τελείες που ξεκινά με v και έχει τουλάχιστον τρία τμήματα (`v1.2.3`). Η εκθετική σημειογραφία δεν επιτρέπεται.

```
package My::Thing 1.042;
package My::Thing v1.2.3;
```

Ορίστε το $VERSION μόνο μία φορά ανά πακέτο - μια δεύτερη εντολή package NAME VERSION στο ίδιο πακέτο αντικαθιστά την πρώτη, και οι καταναλωτές `use Module VERSION` βλέπουν μόνο την τελική τιμή.

package NAMESPACE BLOCK

Περιορίζει λεξιλογικά την αλλαγή πακέτου στο BLOCK. Μέσα στα άγκιστρα το πακέτο μεταγλώττισης είναι το NAMESPACE· **έξω από το άγκιστρο κλεισίματος το προηγούμενο πακέτο επαναφέρεται**. Αυτή είναι η μόνη μορφή που επαναφέρεται αυτόματα.

```
package Outer;
our $x = 1;              # $Outer::x

package Inner {
    our $y = 2;          # $Inner::y
}

our $z = 3;              # $Outer::z again
```

package NAMESPACE VERSION BLOCK

Η μορφή μπλοκ με version literal. Η εμβέλεια επαναφέρεται στο άγκιστρο κλεισίματος· το `$NAMESPACE::VERSION` παραμένει ορισμένο.

```
package My::Sub v0.9.1 {
    sub ping { "pong" }
}
```

Τι περιορίζει και τι δεν περιορίζει σε εμβέλεια η package

Η package επηρεάζει τις **δυναμικές μεταβλητές (πακέτου)** και την επίλυση μη προσδιορισμένων ονομάτων. Δεν επηρεάζει τα λεξιλογικά:

- my, state - δημιουργούνται στο λεξιλογικό pad, αόρατα σε αναζήτηση πακέτου. Μια εντολή package δεν τους κάνει τίποτα.
- local - οριοθετεί δυναμικά μια μεταβλητή πακέτου, η οποία εξακολουθεί να ανήκει σε όποιο πακέτο κατονομάζει το πλήρως προσδιορισμένο όνομά της.
- our - δηλώνει λεξιλογικό ψευδώνυμο προς μια μεταβλητή πακέτου στο *τρέχον πακέτο μεταγλώττισης*. Αυτός είναι ο μηχανισμός που επιτρέπει στο our $count μέσα σε package My::Thing να γίνει ψευδώνυμο του $My::Thing::count.

Η αλλαγή πακέτου μετά από δήλωση our δεν αναπροσανατολίζει το ψευδώνυμο: το ψευδώνυμο δεσμεύτηκε στο πακέτο που ίσχυε όταν μεταγλωττίστηκε η our.

```
package Foo;
our $x;                 # aliases $Foo::x
package Bar;
$x = 1;                 # still writes $Foo::x
```

Αλληλεπίδραση με την use v5.36 και νεότερες

Η use v5.36 (και κάθε use v5.36 ή νεότερη) ενεργοποιεί έμμεσα τις strict και warnings. Μόλις τεθεί σε ισχύ η strict, οι μη προσδιορισμένες μεταβλητές πακέτου απαιτούν δήλωση our (ή πλήρη προσδιορισμό). Σε συνδυασμό με εντολή package, ο ιδιωματικός σύγχρονος σκελετός αρχείου είναι:

```
package My::Thing 1.00;

use v5.36;              # strict, warnings, say, signatures, ...

our @EXPORT_OK = qw(thingify);

sub thingify { ... }

1;
```

Η μορφή package NAME VERSION και η use VERSION είναι ανεξάρτητες: η πρώτη ορίζει το $My::Thing::VERSION· η δεύτερη ενεργοποιεί χαρακτηριστικά γλώσσας για το τρέχον αρχείο.

Ειδικά αναγνωριστικά που σημαίνουν πάντοτε main::

Ένα μικρό σύνολο αναγνωριστικών τοποθετείται αναγκαστικά στο πακέτο main:: ανεξάρτητα από το τρέχον package:

- STDIN, STDOUT, STDERR, ARGV, ARGVOUT
- @ARGV, @INC, %ENV, %SIG, %INC
- Όλες οι μεταβλητές στίξης: $_, $!, $@, @_ κ.λπ.

- Το αναγνωριστικό ενός χαρακτήρα _, σε όλα τα sigils του: το filehandle κάτω παύλας που χρησιμοποιείται από την `stat` και τους ελέγχους αρχείων `-X`, το βαθμωτό `$_`, και ο πίνακας `@_`.

Η εγγραφή στο `$!` μέσα σε `package Foo` εξακολουθεί να γράφει στο `$main::!`. Αυτό είναι σκόπιμο: αυτά τα ονόματα θα ήταν άχρηστα αν μετακινούνταν με το πακέτο.

Παραδείγματα

Ένα ελάχιστο αρχείο αρθρώματος. Το όνομα του αρχείου πρέπει να ταιριάζει με το όνομα του πακέτου (η `require` μεταφράζει το `::` στον διαχωριστή καταλόγου):

```
# 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;
```

Δύο πακέτα σε ένα αρχείο με τη μορφή μπλοκ. Η εμβέλεια επαναφέρεται αυτόματα, οπότε το `1;` στο κάτω μέρος δεν ανήκει σε κανένα συγκεκριμένο πακέτο - αποτιμάται σε όποιο πακέτο ήταν ενεργό στην κορυφή του αρχείου (`main` για σενάριο, το ίδιο το πακέτο του αρχείου για άρθρωμα):

```
package My::Thing {
    sub do_stuff { ... }
}

package My::Thing::Helper {
    sub helper { ... }
}
```

Εναλλαγή μπρος-πίσω με τη μορφή εντολής - νόμιμο αλλά σπάνια αυτό που θέλετε:

```
package A;
sub one { 1 }           # &A::one

package B;
sub two { 2 }           # &B::two

package A;
sub three { 3 }         # &A::three
```

Ορισμός έκδοσης και ανάγνωσή της πίσω. Σημειώστε ότι το `$VERSION` είναι αντικείμενο `version`, όχι απλή συμβολοσειρά· μετατρέψτε σε συμβολοσειρά ή συγκρίνετε με `version`:

```
package Foo 1.23;

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
say $Foo::VERSION;      # 1.23
say ref $Foo::VERSION;  # version
```

Η `our` μέσα σε ένα πακέτο σάς δίνει λεξιλογικά οριοθετημένο ψευδώνυμο προς μια μεταβλητή πακέτου - ο καθιερωμένος τρόπος να δηλώνετε καθολικές σε επίπεδο αρθρώματος υπό `strict`:

```
package Counter;
use v5.36;

our $count = 0;          # aliases $Counter::count
sub bump { $count++ }
```

Οριακές περιπτώσεις

- **Η package NAMESPACE VERSION δέχεται μόνο αυστηρά version literals.** Η εκθετική σημειογραφία (1e3), οι μη αριθμητικές συμβολοσειρές και τα v-strings δύο τμημάτων (v1.2) αποτελούν σφάλματα μεταγλώττισης.
- **Ο διπλός ορισμός του $VERSION** στο ίδιο πακέτο (είτε με δύο εντολές `package NAME VERSION` είτε με ρητή εκχώρηση μετά από `package NAME VERSION`) είναι νόμιμος αλλά μπερδεύει - η `use Module VERSION` ελέγχει την τελική τιμή, ενώ τα εργαλεία του CPAN διαβάζουν όποια βρουν πρώτη. Δηλώστε την έκδοση μία φορά.
- **Η μορφή μπλοκ είναι ξεχωριστή λεξιλογική εμβέλεια.** Οι μεταβλητές `my` και `state` που δηλώνονται μέσα σε `package NAME { ... }` δεν είναι ορατές απ” έξω, όπως και σε οποιοδήποτε άλλο μπλοκ.
- **Η package NAMESPACE χωρίς ακολουθούσα εντολή ή μπλοκ** επηρεάζει το υπόλοιπο του αρχείου. Μια package στην κορυφή ενός σεναρίου μετακινεί μόνιμα το σενάριο εκτός του `main`, που σημαίνει ότι το `$main::ARGV[0]` εξακολουθεί να είναι η λίστα ορισμάτων σας αλλά οι νεοδηλωμένες καθολικές ανήκουν στο νέο πακέτο. Σχεδόν ποτέ ένα σενάριο δεν θέλει αυτό.
- **Ένα αρχείο που περιέχει μόνο package Foo; και κανέναν άλλο κώδικα** εξακολουθεί να μεταγλωττίζεται καθαρά και να δημιουργεί το stash `Foo::` στην πρώτη αναφορά - αλλά το `require` σε αυτό επιστρέφει την τιμή της τελευταίας εντολής, που είναι η ίδια η εντολή package (ψευδής). Βάλτε `1;` στο τέλος κάθε αρχείου αρθρώματος.
- **Εμφωλευμένες μορφές μπλοκ** επαναφέρουν το άμεσα περικλείον πακέτο, όχι το `main`:

  ```
  package Outer;
  package Inner {
      package Innermost {
          # __PACKAGE__ is Innermost
      }
      # __PACKAGE__ is Inner again
  }
  # __PACKAGE__ is Outer again
  ```

- **Η package μέσα σε `eval`**. Η αλλαγή πακέτου περιορίζεται στο σώμα της `eval`. Τα σύμβολα που δημιουργήθηκαν κατά την `eval` παραμένουν στα stashes τους (τα stashes είναι καθολικά), αλλά το πακέτο μεταγλώττισης επανέρχεται όταν τελειώσει η `eval`.
- **Απόστροφος ως διαχωριστής πακέτου.** Ο απόστροφος ήταν ο αρχικός διαχωριστής χώρου ονομάτων της Perl: το `$main'sail` σήμαινε `$main::sail`, πολύ πριν υπάρξει το `::`. Επιβιώνει μόνο για την ανάγνωση κώδικα εποχής Perl 4 και είναι απενεργοποιημένος υπό τα σύγχρονα πακέτα χαρακτηριστικών, οπότε ένα τρέχον πρόγραμμα γραμμένο με `use v5.36` (ή οποιοδήποτε πρόσφατο πακέτο) δεν θα τον αναγνωρίσει. Μην τον γράφετε· η διπλή άνω-κάτω τελεία `::` είναι ο μόνος διαχωριστής που πρέπει να χρησιμοποιεί ο τρέχων κώδικας.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `use` - φορτώνει ένα άρθρωμα κατά τη μεταγλώττιση και εισάγει σύμβολα από το πακέτο του· συνήθως συνοδεύεται από δήλωση `package` στην κορυφή του φορτωμένου αρχείου
- `require` - φορτώνει άρθρωμα ή αρχείο σε χρόνο εκτέλεσης· η εντολή `package` του φορτωμένου αρχείου καθορίζει ποιο stash γεμίζουν οι δηλώσεις του
- `bless` - συσχετίζει μια αναφορά με ένα πακέτο ώστε οι κλήσεις μεθόδων να διανέμονται στις υπορουτίνες εκείνου του πακέτου· η συγκολλητική ύλη μεταξύ `package` και αντικειμενοστρέφειας
- `my` - δηλώνει λεξιλογική μεταβλητή που η `package` **δεν** επηρεάζει· το αντίστοιχο των καθολικών πακέτου
- `our` - δηλώνει λεξιλογικό ψευδώνυμο προς μεταβλητή πακέτου στο τρέχον πακέτο μεταγλώττισης· ο strict-safe τρόπος να χρησιμοποιείτε καθολικές σε επίπεδο αρθρώματος
- `local` - οριοθετεί δυναμικά την τιμή μιας μεταβλητής πακέτου χωρίς να αλλάζει σε ποιο πακέτο ανήκει

Διεργασίες

pipe

Ανοίγει ένα ζεύγος συνδεδεμένων filehandles - ένα για ανάγνωση, ένα για εγγραφή.

Η `pipe` περιτυλίσσει την κλήση συστήματος `pipe(2)`. Δημιουργεί ένα ανώνυμο, μονόδρομο κανάλι bytes στον πυρήνα και σας παραδίδει δύο filehandles: το `READHANDLE`, από το οποίο μπορούν να διαβαστούν bytes, και το `WRITEHANDLE`, στο οποίο μπορούν να γραφτούν bytes. Οτιδήποτε γράφεται στο `WRITEHANDLE` γίνεται διαθέσιμο στο `READHANDLE` με την ίδια σειρά. Η κλασική χρήση είναι η επικοινωνία μεταξύ γονικής διεργασίας και παιδιού που παράχθηκε με `fork`: η μία πλευρά γράφει, η άλλη διαβάζει.

Σύνοψη

```
pipe READHANDLE, WRITEHANDLE
```

Τι επιστρέφεται

Αληθές σε επιτυχία, ψευδές σε αποτυχία (με το `$!` ρυθμισμένο στον λόγο - τυπικά `EMFILE` ή `ENFILE` όταν ο πίνακας αρχείων της διεργασίας ή του συστήματος είναι γεμάτος). Και τα δύο handles δημιουργούνται ως παρενέργειες στα ορίσματα· δεν υπάρχει τιμή επιστροφής τύπου «αντικείμενο pipe».

Σε επιτυχία κρατάτε πλέον **δύο** ανοιχτούς περιγραφείς αρχείων. Κάθε περιγραφέας που δημιουργείτε είναι περιγραφέας που πρέπει να `close`νετε - ειδικά το άκρο που κάθε διεργασία **δεν** χρησιμοποιεί μετά από `fork` (δείτε *Οριακές περιπτώσεις*).

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται σε αποτυχία στο `errno` από την υποκείμενη κλήση `pipe(2)`.
- `$|` - **autoflush** στο handle εγγραφής. Η `print` ενταμιεύει μέσω PerlIO, οπότε χωρίς autoflush ο αναγνώστης μπορεί να μη δει τίποτα μέχρι ο εγγραφέας να εξέλθει ή να γεμίσει ο ενταμιευτής. Ρυθμίστε το `$|` στο handle εγγραφής (τυπικά μέσω `select`) πριν την εγγραφή, αλλιώς το πρωτόκολλό σας θα μπλοκάρει σε deadlock.
- `$^F` - το **μέγιστο filehandle συστήματος**. Νέοι περιγραφείς με `fileno` μεγαλύτερο από `$^F` (προεπιλογή 2, δηλαδή πάνω από `STDERR`) λαμβάνουν αυτόματα τη σημαία close-on-exec, οπότε δεν διαρρέουν σε προγράμματα που ξεκινούν με `exec`. Χαμηλώστε το `$^F` για να κρατήσετε το pipe ανοιχτό κατά την `exec`· ανεβάστε το, ή κάντε dup σε χαμηλόαριθμο fd, για να επιβάλετε κληρονόμηση.

Παραδείγματα

Ένας γονέας γράφει μια γραμμή· ένα forked παιδί τη διαβάζει:

```
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;
    exit 0;
}

# parent: writes only
close $reader;
$writer->autoflush(1);
print $writer "hello from parent\n";
close $writer;
waitpid $pid, 0;
```

Το παιδί αναφέρει την άλλη κατεύθυνση - ένα παιδί παράγει δεδομένα, ο γονέας τα καταναλώνει:

```
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 από την πλευρά επιλεγμένου handle, για κώδικα που προηγείται των κλήσεων μεθόδων `IO::Handle`:

```
pipe my $reader, my $writer or die "pipe: $!";
my $old = select $writer; $| = 1; select $old;
```

Για τις περισσότερες χρήσεις, η `open` με pipe mode είναι απλούστερη - κάνει fork, pipe και exec για εσάς:

```
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;
```

Καταφύγετε στη σκέτη `pipe` όταν χρειάζεστε και τις δύο κατευθύνσεις, χρειάζεται να διατηρήσετε το παιδί χωρίς exec, ή χτίζετε κάτι που οι δύο pipe-μορφές της `open` δεν καλύπτουν - τότε τα `IPC::Open2` / `IPC::Open3` περιτυλίγουν τον χορό `pipe` + `fork` + `exec` για εσάς.

Οριακές περιπτώσεις

- **Κλείστε το άκρο που δεν χρησιμοποιείτε.** Μετά από `fork`, **και οι δύο** διεργασίες κρατούν **και τους δύο** περιγραφείς. Ο αναγνώστης πρέπει να κλείσει το άκρο εγγραφής, και ο εγγραφέας πρέπει να κλείσει το άκρο ανάγνωσης. Αν το παραλείψετε λαμβάνετε το κλασικό bug του pipe: το `<$reader>` του αναγνώστη δεν επιστρέφει ποτέ EOF, επειδή ο πυρήνας εξακολουθεί να βλέπει ανοιχτό έναν εγγραφέα (το ίδιο, αδρανές, αντίγραφο του `$writer` του αναγνώστη). Το πρόγραμμα κρεμάει.

- **Η εγγραφή σε pipe χωρίς αναγνώστη εγείρει SIGPIPE.** Μόλις κλείσει κάθε αντίγραφο του άκρου ανάγνωσης, η επόμενη `print` στην πλευρά εγγραφής παραδίδει SIGPIPE, του οποίου η προεπιλεγμένη ενέργεια είναι ο τερματισμός της διεργασίας. Εγκαταστήστε `$SIG{PIPE} = 'IGNORE'` (ή χειριστή) για να το μετατρέψετε σε κανονική αποτυχία εγγραφής με το `$!` ρυθμισμένο σε EPIPE, που μπορείτε στη συνέχεια να ελέγξετε.
- **Deadlock σε βρόχους pipe.** Δύο διεργασίες που γράφουν η μία στην άλλη μέσω pipes θα μπλοκάρουν για πάντα αν γεμίσουν και οι δύο ενταμιευτές εγγραφής πριν διαβάσει οποιαδήποτε από τις δύο. Οι ενταμιευτές pipe του πυρήνα είναι μικροί (τυπικά 64 KiB). Αποστραγγίστε τη μία πλευρά, χρησιμοποιήστε I/O χωρίς μπλοκάρισμα, ή χρησιμοποιήστε `select` / `IO::Select` για πολυπλεξία.
- **Το buffering κρύβει εγγραφές.** Χωρίς `$|` (ή `$writer->autoflush(1)`) στην πλευρά εγγραφής, τα δεδομένα κάθονται στον ενταμιευτή του PerlIO μέχρι να γεμίσει ο ενταμιευτής ή να κλείσει το handle. Ένας αναγνώστης μπλοκαρισμένος σε `<$reader>` θα περιμένει - συχνά επ” αόριστον - δεδομένα που έχουν ήδη «γραφτεί» από τη σκοπιά του εγγραφέα.
- **Μόνο μονόδρομη.** Η `pipe` παράγει μονόδρομο κανάλι: εγγραφή στο `WRITEHANDLE`, ανάγνωση από το `READHANDLE`, όχι το αντίστροφο. Για αμφίδρομη επικοινωνία χρησιμοποιήστε `socketpair` ή δύο pipes.
- **Δυαδικά δεδομένα.** Τα handles που επιστρέφει η `pipe` ξεκινούν στην προεπιλεγμένη στοίβα στρωμάτων PerlIO. Αν μεταφέρετε ωμά bytes διαμέσου του pipe, εφαρμόστε `binmode` και στα δύο άκρα· διαφορετικά στρώματα CRLF ή κωδικοποίησης που κληρονομούνται από τις προεπιλογές της διεργασίας μπορούν να ξαναγράψουν τα δεδομένα σας.
- **Close-on-exec και `fileno`.** Σε συστήματα με `FD_CLOEXEC`, κάθε νέος περιγραφέας του οποίου το `fileno` υπερβαίνει το `$^F` δημιουργείται με τη σημαία close-on-exec ενεργοποιημένη. Οι περιγραφείς pipe τυπικά προσγειώνονται αρκετά πάνω από το 2, οπότε εξ ορισμού **δεν** επιβιώνουν της `exec` - που είναι κανονικά αυτό που θέλετε. Χαμηλώστε το `$^F` *πριν* καλέσετε την `pipe` αν προτίθεστε να περάσετε τους περιγραφείς σε πρόγραμμα που εκτελείται μέσω `exec`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `fork` - το άλλο μισό του ιδιώματος pipe· η `pipe` σχεδόν πάντα καλείται ακριβώς πριν από μια `fork`
- `open` - οι μορφές `'|-'` και `'-|'` κάνουν fork, pipe και `exec` παιδιού σε μία κλήση· προτιμήστε τες όταν χρειάζεστε ακριβώς μία κατεύθυνση προς μία εξωτερική εντολή
- `close` - το κλείσιμο του άκρου που δεν χρησιμοποιείται σε κάθε διεργασία είναι υποχρεωτικό, όχι προαιρετικό
- `read` - αναγνώσεις προσανατολισμένες σε bytes από το `READHANDLE` όταν η προσανατολισμένη σε γραμμές μορφή `<$fh>` είναι λάθος σχήμα

- `print` - ο κανονικός τρόπος για να σπρώξετε δεδομένα στο WRITEHANDLE· θυμηθείτε το `$|`
- `fileno` - ανακτά τον υποκείμενο περιγραφέα αρχείου, π.χ. για να τον παραδώσετε σε `select` / `IO::Select` ή σε κλήση συστήματος χαμηλότερου επιπέδου

Πίνακες

pop

Αφαιρεί και επιστρέφει το τελευταίο στοιχείο ενός πίνακα.

Η pop συντομεύει τον πίνακα κατά ένα και σας παραδίδει το στοιχείο που αφαιρέθηκε. Ο πίνακας τροποποιείται επιτόπου. Χωρίς όρισμα, η pop δρα επί του `@_` μέσα σε υπορουτίνα και επί του `@ARGV` οπουδήποτε αλλού - ο ίδιος προεπιλεγμένος στόχος όπως και της `shift`.

Σύνοψη

```
pop @arr
pop @$aref
pop                 # default: @_ inside a sub, @ARGV outside
```

Τι επιστρέφεται

Το στοιχείο που αφαιρέθηκε, ως βαθμωτό. Αν ο πίνακας ήταν κενός, η τιμή επιστροφής είναι `undef` και ο πίνακας παραμένει κενός - δεν αποτελεί σφάλμα και δεν εκπέμπει προειδοποίηση, οπότε κώδικας που καλεί πάντα pop σε βρόχο χρειάζεται δικό του έλεγχο τερματισμού.

```
my @empty;
my $x = pop @empty;         # $x is undef, no warning
```

Σημειώστε ότι η pop επιστρέφει `undef` και όταν το τελευταίο στοιχείο *ήταν* `undef`. Οι δύο περιπτώσεις είναι μη διακρίσιμες μόνο από την τιμή επιστροφής· χρησιμοποιήστε `@arr` σε λογικό περιβάλλον, ή `scalar @arr`, για να τις ξεχωρίσετε πριν την κλήση pop.

```
my @arr = ('one', 'two', undef);
my $x = pop @arr;           # $x is undef - but so would an empty␣
↪pop be
```

Καθολική κατάσταση που επηρεάζει

Η pop χωρίς όρισμα διαβάζει το `@_` (μέσα σε υπορουτίνα) ή το `@ARGV` (σε εμβέλεια αρχείου, σε μπλοκ `BEGIN`/`INIT`/`CHECK`/`END` και μέσα σε `eval STRING`). Δεν αλληλεπιδρά με την `$_`, την `$,`, την `$\` ή οποιαδήποτε άλλη μεταβλητή στίξης.

Παραδείγματα

Βασική χρήση - αφαιρέστε το τελευταίο στοιχείο από έναν απλό πίνακα:

```
my @arr  = ('cat', 'dog', 'mouse');
my $item = pop @arr;        # $item is 'mouse'
# @arr is now ('cat', 'dog')
```

Στοίβα LIFO - push για προσθήκη, pop για αφαίρεση, και τα δύο στο ίδιο άκρο:

```
my @stack;
push @stack, 'a';
push @stack, 'b';
push @stack, 'c';
my $top = pop @stack;       # 'c'; @stack is ('a', 'b')
```

Απορρίψτε το τελευταίο στοιχείο όταν δεν χρειάζεστε την τιμή του. Η pop σε κενό περιβάλλον είναι απολύτως έγκυρη και είναι ο ιδιωματικός τρόπος να εγκαταλείψετε την ουρά:

```
pop @arr;                   # shorten @arr by one, value thrown away
```

Προεπιλεγμένος στόχος μέσα σε υπορουτίνα - η pop χωρίς όρισμα κάνει pop από το @_, και έτσι μια υπορουτίνα ξεφλουδίζει το τελευταίο της όρισμα από τη λίστα ορισμάτων:

```
sub last_arg {
    return pop;             # pops from @_
}
last_arg(1, 2, 3);          # returns 3
```

Pop μέσω αναφοράς πίνακα. Κάντε αποαναφορά με `@$aref` (ή `@{ $aref }` για οποιαδήποτε μη τετριμμένη έκφραση) και η pop λειτουργεί ακριβώς όπως σε επώνυμο πίνακα:

```
my $aref = [10, 20, 30];
my $x    = pop @$aref;      # 30; $aref now points to [10, 20]
```

Οριακές περιπτώσεις

- **Ο κενός πίνακας επιστρέφει undef, όχι σφάλμα.** Δεν εκδίδεται προειδοποίηση, ούτε καν υπό `use warnings`. Ένας βρόχος της μορφής `while (defined(my $x = pop @arr)) { ... }` τερματίζει είτε σε κενό *είτε* σε γνήσιο στοιχείο undef - χρησιμοποιήστε αντί γι" αυτό `while (@arr)` αν ο πίνακας μπορεί νόμιμα να περιέχει undef.
- **Bareword έκφραση στη θέση του ARRAY.** Η pop απαιτεί πίνακα, όχι αυθαίρετη έκφραση. Η `pop @$aref` λειτουργεί επειδή το `@$aref` είναι σύνταξη πίνακα· η `pop $aref` είχε επιτραπεί για λίγο ως πείραμα μεταξύ της 5.14 και της 5.22 και είναι σφάλμα μεταγλώττισης από την 5.24. Γράφετε πάντα ρητά το sigil @ στην αποαναφορά.
- **Δεμένοι πίνακες.** Η pop `@tied` καλεί τη μέθοδο `POP` της δεμένης κλάσης αν είναι ορισμένη· αλλιώς ισχύει η προεπιλεγμένη υποχώρηση `FETCHSIZE` + `FETCH` +

DELETE + STORESIZE. Οι παρενέργειες αυτών των μεθόδων πυροδοτούνται με αυτή τη σειρά.

- **Η pop συντομεύει τον πίνακα.** Μετά την pop @arr, η $#arr μειώνεται κατά ένα και η scalar @arr πέφτει κατά ένα. Αν χρειάζεται να επιθεωρήσετε την ουρά χωρίς να την αφαιρέσετε, διαβάστε απευθείας $arr[-1] ή $arr[$#arr] - δείτε την $#arr στο perlvar.
- **Η μορφή χωρίς όρισμα αλλάζει σημασία ανάλογα με την εμβέλεια.** Σε εμβέλεια αρχείου, η pop ; σημαίνει pop @ARGV. Μέσα σε υπορουτίνα σημαίνει pop @_. Η ίδια λοιπόν γραμμή πηγαίου κώδικα συμπεριφέρεται διαφορετικά ανάλογα με το πού εμφανίζεται - μια συνηθισμένη έκπληξη όταν κώδικας αναδιαρθρώνεται μέσα σε ή έξω από υπορουτίνα. Γράφετε ρητά τον στόχο (pop @ARGV ή pop @_) όποτε η πρόθεση δεν είναι προφανής από το περιβάλλον.
- **Η pop σε πίνακα με ψευδωνυμοποίηση μέσω @_.** Τα ορίσματα υπορουτίνας είναι ψευδώνυμα προς τις τιμές του καλούντος. Η pop @_ αφαιρεί την εγγραφή ψευδωνύμου από το @_ αλλά δεν συντομεύει τον πίνακα του καλούντος· για να συρρικνώσετε τον πίνακα του καλούντος, η υπορουτίνα πρέπει να δεχθεί τον πίνακα *μέσω αναφοράς* και να χρησιμοποιηθεί pop @$aref αντί γι" αυτό.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- push - το αντίστοιχο· προσαρτά στο ίδιο άκρο του πίνακα από όπου αφαιρεί η pop
- shift - αφαιρεί και επιστρέφει το *πρώτο* στοιχείο· ίδιοι κανόνες προεπιλογής με την pop (@_ σε υπορουτίνα, αλλιώς @ARGV)
- unshift - προπενθέτει στην κεφαλή· ζευγαρωμένη με την shift όπως η push είναι ζευγαρωμένη με την pop
- splice - αφαιρεί ή αντικαθιστά αυθαίρετο τμήμα· η splice @arr, -1 είναι η εκτενής μορφή της pop @arr
- last - δεσμευμένη λέξη ελέγχου βρόχου, άσχετη με το τέλος πίνακα· αναφέρεται επειδή τα ονόματα συχνά συγχέονται
- $#arr - τελευταίος δείκτης του @arr· αλλάζει όταν η pop συντομεύει τον πίνακα

Κανονικές εκφράσεις και αντιστοίχιση μοτίβων

pos

Αναφέρει ή ορίζει πού θα συνεχιστεί η επόμενη αντιστοίχιση regex /g σε μια συμβολοσειρά.

Η pos διαβάζει (ή, ως lvalue, γράφει) το offset που η μηχανή regex αποθηκεύει σε ένα βαθμωτό μετά από καθολική αντιστοίχιση (m//g, s///g). Το offset μετράει χαρακτήρες, όχι bytes, και είναι η θέση **μετά** την τελευταία επιτυχημένη αντιστοίχιση - το σημείο όπου η επόμενη επανάληψη /g θα ξεκινήσει τη σάρωση. Χωρίς όρισμα η pos λειτουργεί πάνω στην $_.

Σύνοψη

```
pos SCALAR
pos $str = N
pos
```

Τι επιστρέφεται

Ένα ακέραιο offset, ή `undef` όταν δεν είναι καταγεγραμμένη θέση. Το `0` είναι έγκυρο offset και σημαίνει «αρχή συμβολοσειράς»· δεν είναι το ίδιο με `undef`, που σημαίνει «καμία αντιστοίχιση `/g` δεν έχει τρέξει, ή η τελευταία απέτυχε και επανέφερε τη θέση.» Πάντα διακρίνετε τα δύο με `defined`:

```
if (defined pos $str) {
    # a /g scan is in progress
}
```

Χρησιμοποιούμενη ως lvalue, η `pos SCALAR` επιστρέφει μια θέση όπου μπορείτε να αναθέσετε:

```
pos($str) = 5;          # next /g match starts at char offset 5
```

Καθολική κατάσταση που επηρεάζει

Η `pos` διαβάζει και γράφει τη θέση regex ανά-βαθμωτό που συνδέεται με τον τελεστέο της. Χωρίς όρισμα στοχεύει την `$_`. Το αποθηκευμένο offset είναι αυτό στο οποίο αγκυρώνεται η `\G` στην επόμενη αντιστοίχιση, οπότε κάθε κλήση στην `pos` ενδέχεται να αλλάξει το πού δένει η `\G`.

Παραδείγματα

Διασχίστε κάθε λέξη σε μια συμβολοσειρά με `/g` σε βαθμωτό περιβάλλον, χρησιμοποιώντας την `pos` για να αναφέρετε την πρόοδο:

```
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
```

Παραλείψτε μπροστά πριν ξεκινήσετε τη σάρωση. Η πρώτη αντιστοίχιση αρχίζει στο offset 4, όχι `0`:

```
my $s = "AAA BBB CCC";
pos($s) = 4;
$s =~ /(\w+)/g;
print $1, "\n";             # BBB
```

Αγκυρώστε μια επόμενη αντιστοίχιση στην προηγούμενη με την `\G`. Χωρίς την `\G` η μηχανή θα σαρώσει προς τα μπρος πέρα από οποιοδήποτε κενό· με αυτήν, η αντιστοίχιση πρέπει να ξεκινήσει ακριβώς εκεί όπου τελείωσε η τελευταία:

```
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
```

Επανεκκινήστε μια σάρωση καθαρίζοντας τη θέση:

```
pos($s) = undef;            # next /g starts from offset 0 again
```

Οριακές περιπτώσεις

- **Η σκέτη `pos`** στοχεύει την `$_`. Η `pos` μέσα σε `while (<>) { ... }` συνεπώς αναφέρει τη θέση στην τρέχουσα γραμμή εισόδου.
- **Χαρακτήρες, όχι bytes.** Για συμβολοσειρά που περιέχει multi-byte χαρακτήρες, η `pos` επιστρέφει το offset σε χαρακτήρες. Η (καταργημένη) pragma `use bytes` μεταβαίνει σε byte offsets· νέος κώδικας δεν πρέπει να βασίζεται σε αυτό.
- **Αποτυχημένη αντιστοίχιση `/g` επαναφέρει τη θέση σε `undef`** - η επόμενη `/g` ξεκινάει ξανά στο offset `0`. Προσθέστε τον τροποποιητή `/c` (`m//gc`) για να διατηρήσετε τη θέση σε αποτυχία, που είναι το συνηθισμένο ιδίωμα όταν συνθέτετε πολλά εναλλακτικά μοτίβα αγκυρωμένα με `\G` πάνω στην ίδια συμβολοσειρά.
- **Οι αναγνώσεις κατά τη διάρκεια μιας αντιστοίχισης είναι παλιές.** Η `pos` αντικατοπτρίζει το τέλος της **προηγούμενης** αντιστοίχισης. Εκφράσεις όπως `(?{ pos() = 5 })` ή `s//pos() = 5/e` επηρεάζουν την επόμενη αντιστοίχιση, όχι αυτήν που τρέχει τώρα.
- **Σημαία αντιστοίχισης μηδενικού μήκους.** Ο ορισμός της `pos` επίσης καθαρίζει την εσωτερική σημαία *αντιστοιχήθηκε με μηδενικό μήκος*, ώστε μια επόμενη αντιστοίχιση μηδενικού πλάτους στην ίδια θέση να επιτρέπεται ξανά. Δείτε το *κεφάλαιο επίδοσης* του οδηγού regex υπό την ενότητα *τερματισμός αντιστοίχισης μηδενικού μήκους*.
- **Τελεστέος που δεν είναι lvalue.** Η `pos` απαιτεί πραγματική βαθμωτή μεταβλητή για τη μορφή lvalue· το `pos("literal") = 3` είναι σφάλμα κατά τη μεταγλώττιση.
- **Offset `0` έναντι `undef`.** Η `pos` με τιμή `0` σημαίνει ότι η επόμενη `/g` ξεκινάει από την αρχή· η `undef` σημαίνει ότι δεν είναι ορισμένη θέση. Συμπεριφέρονται πανομοιότυπα για την πρώτη αντιστοίχιση αλλά διαφέρουν όταν ληφθεί υπόψη η κατάσταση της σημαίας αντιστοίχισης μηδενικού μήκους.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `m` - ο τροποποιητής `/g` είναι αυτός που δημιουργεί και προωθεί τη θέση που διαβάζει η `pos`· ο τροποποιητής `/c` σε αποτυχία τη διατηρεί
- `qr` - προμεταγλωττίστε ένα μοτίβο μία φορά, και μετά επαναχρησιμοποιήστε το σε βρόχους `/g` αγκυρωμένους με `\G` χωρίς εκ νέου ανάλυση
- `split` - ο άλλος καθημερινός τρόπος να διασχίσετε μια συμβολοσειρά σε κομμάτια· καμία `pos` δεν εμπλέκεται, αλλά συχνά πιο καθαρή επιλογή όταν οι οριοθέτες είναι απλοί
- *άγκυρα* `\G`
  - άγκυρα μηδενικού πλάτους που δένει στην τρέχουσα `pos`· ο κύριος λόγος για να αγγίξετε την `pos` εξαρχής
- `$_` - προεπιλεγμένος στόχος της σκέτης `pos`

I/O

print

Γράφει μια λίστα τιμών σε ένα filehandle.

Η `print` είναι η καθημερινή πρωτογενής λειτουργία εξόδου. Μετατρέπει κάθε στοιχείο της `LIST` σε συμβολοσειρά, τα ενώνει με την τρέχουσα τιμή της `$,` (του διαχωριστή πεδίων εξόδου, συνήθως κενού), τα γράφει στο `FILEHANDLE` και τελικά προσθέτει την `$\` (τον διαχωριστή εγγραφών εξόδου, συνήθως κενό). Αν παραλειφθεί το `FILEHANDLE`, η έξοδος πηγαίνει στο τρέχον επιλεγμένο handle (βλ. `select`), που εξ ορισμού είναι το `STDOUT`. Αν παραλειφθεί η `LIST`, τυπώνεται η `$_`.

Σύνοψη

```
print LIST
print FILEHANDLE LIST
print {EXPR} LIST
print
```

Τι επιστρέφεται

1 σε επιτυχία, `undef` σε αποτυχία (με την `$!` ορισμένη). Η τιμή επιστροφής σπάνια ελέγχεται - οι περισσότερες κλήσεις `print` στοχεύουν στο `STDOUT` ή στο `STDERR`, και μια αποτυχία εγγραφής σε οποιοδήποτε από τα δύο είναι ήδη συνθήκη τερματισμού του προγράμματος. Ελέγξτε την τιμή επιστροφής όταν γράφετε σε αρχείο, σωλήνα ή υποδοχή όπου έχουν σημασία οι μερικές εγγραφές και οι σπασμένοι σωλήνες.

```
print $fh @records
    or die "write to $path failed: $!";
```

Προεπιλεγμένο filehandle, προεπιλεγμένοι διαχωριστές

Τρία κομμάτια καθολικής κατάστασης διαμορφώνουν κάθε κλήση `print`:

- Το **τρέχον επιλεγμένο filehandle**, που αλλάζει με την `select`. Η `print LIST` (χωρίς ρητό filehandle) γράφει εκεί. Στην έναρξη του προγράμματος είναι το `STDOUT`.
- `$,` - **διαχωριστής πεδίων εξόδου**, παρεμβάλλεται μεταξύ των στοιχείων της λίστας. Προεπιλογή είναι η κενή συμβολοσειρά, οπότε η `print 1, 2, 3` παράγει `123`, όχι `1 2 3`. Επίσης γνωστή ως `$OFS` υπό `use English`.
- `$\` - **διαχωριστής εγγραφών εξόδου**, προστίθεται μετά από ολόκληρη τη λίστα. Προεπιλογή είναι η κενή συμβολοσειρά, οπότε η `print` **δεν** προσθέτει newline. Επίσης γνωστή ως `$ORS` υπό `use English`.

Ο ορισμός και των δύο δίνει σημασιολογία `print` τύπου Python:

```
local $, = " ";
local $\ = "\n";
print 1, 2, 3;          # "1 2 3\n"
```

Χρησιμοποιήστε `local` όταν αλλάζετε αυτές μέσα σε υπορουτίνα ώστε η αλλαγή να μη διαρρεύσει στον καλούντα.

Η `say` είναι ακριβώς `print` με την `$\` αναγκαστικά σε `"\n"` για εκείνη τη μία κλήση - καταφύγετε σε αυτήν όταν θέλετε newline κάθε φορά και δεν θέλετε να αγγίξετε την `$\`.

Παραδείγματα

Βασική εγγραφή στο `STDOUT`:

```
print "hello, world\n";
```

Εγγραφή σε filehandle με τη μορφή έμμεσου αντικειμένου. Παρατηρήστε: **κανένα κόμμα** ανάμεσα στο filehandle και τη λίστα:

```
open my $fh, ">", "out.txt" or die $!;
print $fh "line 1\n", "line 2\n";
close $fh;
```

Εγγραφή σε filehandle που κρατείται σε έκφραση. Οποιοδήποτε filehandle πιο σύνθετο από bareword ή απλό βαθμωτό χρειάζεται τη μορφή μπλοκ `{EXPR}`:

```
my @logs = (\*STDOUT, \*STDERR);
print { $logs[$level] } "message\n";
```

Ένωση στοιχείων λίστας με την `$,` και τερματισμός της εγγραφής με την `$\`:

```
local $, = ", ";
local $\ = ".\n";
print "apples", "pears", "plums";   # "apples, pears, plums.\n"
```

Λίστα που περιέχει `undef` - κάθε `undef` μετατρέπεται σε συμβολοσειρά ως κενή συμβολοσειρά και ενεργοποιεί προειδοποίηση `uninitialized` υπό `use warnings`:

```
use warnings;
my $x;
print "value=", $x, "\n";           # "value=\n" plus a warning
```

Περιβάλλον λίστας έναντι βαθμωτού περιβάλλοντος. Ο `@arr` σε θέση `LIST` επεκτείνεται στα στοιχεία του· τυλίξτε σε `scalar` για να τυπώσετε το πλήθος αντί:

```
my @arr = (10, 20, 30);
print @arr, "\n";                   # "102030\n"
print scalar @arr, "\n";            # "3\n"
```

Καταστολή της προειδοποίησης `undef` για ένα μπλοκ όπου οι τιμές που λείπουν είναι αναμενόμενες και προορίζονται να εμφανιστούν ως κενές:

```
no warnings 'uninitialized';
print join(",", @row), "\n";        # empty fields render as ""
```

Οριακές περιπτώσεις

- **Κανένα όρισμα**: η `print;` τυπώνει την `$_` στο επιλεγμένο handle. Χρησιμοποιείται ιδιωματικά μέσα σε `while (<>) { print; }`.
- **Η αμφισημία filehandle-ή-πρώτο-όρισμα**: η Perl αποφασίζει αν το πρώτο token είναι filehandle κατά την ανάλυση, όχι από τον τύπο χρόνου εκτέλεσης. Ένα bareword ή απλή βαθμωτή μεταβλητή που ακολουθείται αμέσως από όρο (χωρίς κόμμα) λαμβάνεται ως filehandle:

  ```
  print STDERR "oops\n";            # STDERR is the filehandle
  print $fh "oops\n";               # $fh is the filehandle
  ```

  Μια έκφραση που επιστρέφει filehandle **δεν** λειτουργεί σε εκείνη τη θέση - η `print $handles[0] "x"` είναι συντακτικό σφάλμα. Χρησιμοποιήστε τη μορφή μπλοκ: `print { $handles[0] } "x"`.
- **Το περιβάλλον λίστας ισχύει για κάθε όρισμα**. Μια κλήση υπορουτίνας στη `LIST` εκτελείται σε περιβάλλον λίστας, όχι βαθμωτό:

  ```
  print localtime(), "\n";          # prints nine numbers, no␣
  ↪spaces
  print scalar localtime(), "\n";   # prints the familiar date␣
  ↪string
  ```

- **Η `print (…)` αναλύεται ως κλήση συνάρτησης**. Η `print (2+3)*4` τυπώνει 5 και κατόπιν απορρίπτει το `*4`. Βάλτε σε παρενθέσεις ολόκληρη τη λίστα ορισμάτων, ή προθέστε `+`:

  ```
  print((2+3)*4);                   # prints 20
  print +(2+3)*4;                   # prints 20
  ```

- **Κλειστό filehandle**: επιστρέφει `undef` και θέτει την `$!` σε `"Bad file descriptor"`. Υπό `use warnings` εκπέμπεται προειδοποίηση `print() on closed filehandle`.

- **Στρώματα κωδικοποίησης**: η `print` γράφει ό,τι λέει να γράψει η στοίβα PerlIO του handle. Αν το handle έχει στρώμα `:utf8` ή `:encoding(...)`, οι συμβολοσειρές κωδικοποιούνται κατά την έξοδο. Τα handles σε κατάσταση bytes περιμένουν συμβολοσειρές bytes· το πέρασμα συμβολοσειράς με wide characters γράφει ωμά codepoints και εκπέμπει προειδοποίηση `Wide character in print`.
- **Autoflush**: η `print` εντάμιεύει μέσω PerlIO. Ορίστε την `$|` στο επιλεγμένο handle ώστε να εκκενώνεται μετά από κάθε εγγραφή - συνήθως απαιτείται για έξοδο προόδου σε σωλήνες ή κατά τον συντονισμό με άλλη διεργασία:

```
$| = 1;                          # autoflush current handle
```

- **Η εκτύπωση σε κλειστό σωλήνα ή υποδοχή** εγείρει `SIGPIPE`. Η προεπιλεγμένη ενέργεια είναι ο τερματισμός της διεργασίας· εγκαταστήστε χειριστή (`$SIG{PIPE} = 'IGNORE'` ή μια sub) ώστε να τη μετατρέψετε σε κανονική αποτυχία εγγραφής που μπορείτε να ελέγξετε.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `say` - `print` με την `$\` αναγκαστικά σε `"\n"` για την κλήση· χρησιμοποιήστε την όταν κάθε εγγραφή τελειώνει σε newline
- `printf` - ίδιοι κανόνες στόχευσης με την `print`, αλλά το πρώτο όρισμα είναι συμβολοσειρά μορφής `sprintf` και η `$\` **δεν** προστίθεται
- `sprintf` - χτίζει τη μορφοποιημένη συμβολοσειρά χωρίς να τη γράψει· συνδυάστε με `print` όταν θέλετε να την καταγράψετε αλλού πρώτα
- `select` - αλλάζει σε ποιο filehandle γράφει η μορφή χωρίς όρισμα της `print`
- `$,` - διαχωριστής πεδίων εξόδου που παρεμβάλλεται μεταξύ στοιχείων λίστας
- `$\` - διαχωριστής εγγραφών εξόδου που προστίθεται μετά τη λίστα
- `$|` - σημαία autoflush για το τρέχον επιλεγμένο handle

I/O

printf

Γράφει μια μορφοποιημένη συμβολοσειρά σε ένα filehandle.

Η `printf` είναι `print FILEHANDLE sprintf(FORMAT, LIST)` με μία σκόπιμη εξαίρεση: ο διαχωριστής εγγραφών εξόδου `$\` **δεν** προστίθεται. Οι κανόνες στόχευσης είναι κατά τα άλλα οι κανόνες της `print` - bareword ή απλό-βαθμωτό filehandle αμέσως πριν τη λίστα ορισμάτων, ή η μορφή μπλοκ `{EXPR}` για οτιδήποτε πιο σύνθετο. Αν παραλειφθεί το `FILEHANDLE`, η έξοδος πηγαίνει στο τρέχον επιλεγμένο handle (βλ. `select`), που ξεκινά τη ζωή του ως `STDOUT`.

Δείτε την `sprintf` για την αναφορά προδιαγραφών μορφής - όλα τα `%d`, `%s`, `%.3f`, `%*d`, `%1$s`, `%v`, σημαίες διανύσματος, ευαίσθητο σε locale `%f`, κ.ο.κ. Αυτή η σελίδα καλύπτει

μόνο την ειδική για την `printf` επιφάνεια: επίλυση filehandle, τον κανόνα «η μορφή είναι πρώτη» και τις ιδιοτροπίες γύρω από τη μορφή χωρίς όρισμα.

Σύνοψη

```
printf FORMAT, LIST
printf FILEHANDLE FORMAT, LIST
printf {EXPR} FORMAT, LIST
printf FH                       # bareword filehandle, $_ as format
```

Τι επιστρέφεται

1 σε επιτυχία, `undef` σε αποτυχία (με την `$!` ορισμένη). Όπως και με την `print`, η τιμή επιστροφής σπάνια ελέγχεται για εγγραφές στο STDOUT / STDERR, και αξίζει να ελέγχεται για εγγραφές σε αρχεία, σωλήνες και υποδοχές όπου έχουν σημασία οι μερικές εγγραφές και οι σπασμένοι σωλήνες.

```
printf $fh "%-20s %8d\n", $name, $count
    or die "write to $path failed: $!";
```

Καθολική κατάσταση που επηρεάζει

- **Επιλεγμένο filehandle** (αλλάζει μέσω της `select`): ο προορισμός όταν παραλείπεται το `FILEHANDLE`. Ξεκινά ως STDOUT.
- `$\` (διαχωριστής εγγραφών εξόδου): σκόπιμα **δεν** προστίθεται - αυτή είναι η μία σημασιολογική διαφορά ανάμεσα στην `printf FORMAT, LIST` και την `print FILEHANDLE sprintf(FORMAT, LIST)`. Βάλτε το newline στη συμβολοσειρά μορφής.
- `$,` (διαχωριστής πεδίων εξόδου): **δεν** συμβουλεύεται. Η `printf` συναρμολογεί τα στοιχεία της λίστας μέσα στη μορφή σύμφωνα με τις μετατροπές, όχι με διαχωριστή.
- `$_`: χρησιμοποιείται ως **μορφή** όταν παραλείπονται τόσο η `FORMAT` όσο και η `LIST`, δεν προστίθεται στην έξοδο. Σχεδόν ποτέ αυτό που θέλετε· βλ. *Οριακές περιπτώσεις*.
- `$!`: τίθεται σε αποτυχία εγγραφής.
- `$|`: σημαία autoflush για το επιλεγμένο handle. Η `printf` εντάμιεύει μέσω PerlIO όπως η `print`.
- locale `LC_NUMERIC`: όταν η `use locale` είναι ενεργή και έχει κληθεί η `POSIX::setlocale`, η δεκαδική υποδιαστολή για μετατροπές κινητής υποδιαστολής (%f, %e, %g) ακολουθεί το locale.

Παραδείγματα

Βασική μορφοποιημένη εγγραφή στο STDOUT. Σε αντίθεση με την `say` ή την `print`-με-`$\`, το τελικό newline σας ανήκει:

```
printf "%s is %d years old\n", $name, $age;
```

Πλάτος και ακρίβεια για μια τακτοποιημένη στήλη:

```
printf "%-20s %8.2f\n", $item, $price;
```

Εγγραφή σε filehandle. Bareword και απλό βαθμωτό λειτουργούν απευθείας· κανένα κόμμα ανάμεσα στο filehandle και στη μορφή:

```
open my $fh, ">", "report.txt" or die $!;
printf $fh "%5d  %s\n", $n, $msg;
close $fh;
```

Filehandle που κρατείται σε πιο σύνθετη έκφραση - χρησιμοποιήστε τη μορφή μπλοκ `{EXPR}` (ίδιος κανόνας με την `print`):

```
my @logs = (\*STDOUT, \*STDERR);
printf { $logs[$severity] } "[%s] %s\n", $tag, $msg;
```

Επαναλαμβανόμενη μορφή με λίστα που εκτείνεται σε πολλές εγγραφές. Η `printf` **δεν** επανεφαρμόζει τη μορφή ανά εγγραφή - οι μετατροπές καταναλώνονται μία φορά, από αριστερά προς τα δεξιά:

```
printf "%s=%s\n", "user", "alice";       # "user=alice\n"
printf "%s=%s\n%s=%s\n",                  # two records, one call
       "user", "alice", "host", "db1";
```

Δεκαδική υποδιαστολή ευαίσθητη σε locale:

```
use POSIX qw(setlocale LC_NUMERIC);
use locale;
setlocale(LC_NUMERIC, "de_DE.UTF-8");
printf "%.2f\n", 1234.5;                  # "1234,50\n"
```

Οριακές περιπτώσεις

- **Δεν δίνεται LIST, η `$_` γίνεται η μορφή.** Η `printf;` και η `printf FH;` (bareword filehandle) μορφοποιούν την `$_`. Αν η `$_` περιέχει οποιεσδήποτε μετατροπές %, αυτές επεκτείνονται έναντι κενής λίστας ορισμάτων - κάθε %s / %d γίνεται η κενή συμβολοσειρά και η `use warnings` εκπέμπει προειδοποίηση `Missing argument in printf`. Για να γράψετε την `$_` αυτούσια χρησιμοποιήστε `print`, όχι `printf`.
- **Έμμεσο filehandle χωρίς λίστα είναι αμφίσημο** και δεν υποστηρίζεται. Η `printf $fh;` αναλύει το `$fh` ως μορφή, όχι ως filehandle. Αν χρειάζεστε μορφή χωρίς όρισμα με βαθμωτό filehandle, περάστε μέσω `print` ή δώστε ρητή μορφή: `printf $fh "%s", $_`.
- **Το πρώτο όρισμα είναι πάντα η μορφή.** Η `printf(@_)` χρησιμοποιεί το `$_[0]` ως μορφή και τα υπόλοιπα στοιχεία ως ορίσματα - αυτός είναι κανόνας ανάλυσης, όχι σύμβαση. Χτίστε τη μορφή σκόπιμα· μην εκτονώνετε λίστες που παρέχει ο χρήστης σε `printf` εκτός αν το πρώτο στοιχείο είναι γνωστά ασφαλές.
- **Δεν προστίθεται `$\`.** Η ενσωμάτωση του `"\n"` στη μορφή είναι ο ιδιωματικός τερματιστής. Ο ορισμός `local $\ = "\n"` δεν έχει επίδραση στην έξοδο της `printf`.

- **Η `$,` αγνοείται.** Ο διαχωριστής πεδίων εξόδου είναι έννοια της `print`· η `printf` δεν ενώνει τίποτα - υποκαθιστά.
- **Η `printf (…)` αναλύεται ως κλήση συνάρτησης**, ακριβώς όπως η `print`. Η `printf ("%d", 1)*2` υπολογίζει `printf(...)` και κατόπιν πολλαπλασιάζει την τιμή επιστροφής επί 2 και την απορρίπτει. Τυλίξτε ολόκληρη τη λίστα ορισμάτων σε παρενθέσεις ή προθέστε `+` αν χρειάζεται να αποσαφηνιστεί.
- **Κλειστό filehandle**: επιστρέφει `undef` και θέτει την `$!` σε `"Bad file descriptor"`. Υπό `use warnings` εκπέμπεται προειδοποίηση `printf() on closed filehandle`.
- **Στρώματα κωδικοποίησης**: η `printf` γράφει μέσα από τη στοίβα PerlIO του handle. Στρώμα `:utf8` ή `:encoding(...)` κωδικοποιεί κατά την έξοδο· handle σε κατάσταση bytes με συμβολοσειρά wide character εκπέμπει `Wide character in printf` και γράφει ωμά codepoints.
- **SIGPIPE σε κλειστό σωλήνα/υποδοχή**: η προεπιλεγμένη ενέργεια τερματίζει τη διεργασία. Εγκαταστήστε το `$SIG{PIPE}` για να το μετατρέψετε σε ελέγξιμη αποτυχία εγγραφής.
- **Μη χρησιμοποιείτε `printf` όπου αρκεί η `print`.** Η `printf "%s", $msg` είναι πιο αργή και πιο επιρρεπής σε σφάλματα από την `print $msg` - η συμβολοσειρά μορφής αναλύεται σε κάθε κλήση, και ένα ξεμοναχιασμένο `%` στο `$msg` που χρησιμοποιείται ως μορφή γίνεται σφάλμα που περιμένει να συμβεί.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `print` - μη μορφοποιημένη εγγραφή με τους ίδιους κανόνες filehandle· ταχύτερη, και προσθέτει την `$\` εκεί που η `printf` δεν την προσθέτει
- `sprintf` - χτίζει τη μορφοποιημένη συμβολοσειρά χωρίς να τη γράψει· η αναφορά προδιαγραφών μορφής βρίσκεται εδώ
- `say` - `print` με την `$\` αναγκαστικά σε `"\n"`· καταφύγετε σε αυτήν όταν κάθε εγγραφή τελειώνει σε newline
- `select` - αλλάζει σε ποιο filehandle γράφει η μορφή χωρίς filehandle της `printf`
- `$\` - διαχωριστής εγγραφών εξόδου· σκόπιμα δεν προστίθεται από την `printf`
- `$_` - χρησιμοποιείται ως μορφή όταν παραλείπονται και η `FORMAT` και η `LIST`

Διάφορα

prototype

Επιστρέφει τη συμβολοσειρά prototype μιας υπορουτίνας, ή `undef` αν δεν έχει.

Η `prototype` επιθεωρεί μια υπορουτίνα και επιστρέφει τον prototype που θα χρησιμοποιήσει ο αναλυτής κατά τη μεταγλώττιση κλήσεων προς αυτήν. Το `FUNCTION` είναι είτε όνομα (bareword ή συμβολοσειρά) είτε αναφορά κώδικα. Χωρίς όρισμα,

αναζητείται η υπορουτίνα που ονομάζει η `$_`. Ένα πρόθεμα `CORE::` ζητά το prototype-ισοδύναμο μιας πυρηνικής ενσωματωμένης.

Τα prototypes είναι οι παλιές υποδείξεις ανάλυσης κατά τη μεταγλώττιση που δηλώνονται με `sub foo ($$@) { ... }`. Ελέγχουν πώς ο αναλυτής διαμορφώνει τις κλήσεις - **δεν** είναι signatures υπορουτινών (`sub foo ($x, $y) { ... }`), που αποτελούν ξεχωριστό χαρακτηριστικό για το οποίο η `prototype` δεν αναφέρει.

Σύνοψη

```
prototype FUNCTION
prototype
prototype \&foo
prototype 'Some::Package::foo'
prototype 'CORE::push'
```

Τι επιστρέφεται

Μια συμβολοσειρά που περιγράφει το prototype, ή `undef` όταν:

- η υπορουτίνα υπάρχει αλλά δεν έχει δηλωμένο prototype, ή
- το όνομα επιλύεται σε `CORE::` ενσωματωμένη της οποίας τη μορφή ορισμάτων η μικρογλώσσα prototype δεν μπορεί να εκφράσει (για παράδειγμα `system`, `print`, `sort`).

Η συμβολοσειρά είναι ακριβώς το κείμενο που εμφανίστηκε μεταξύ των παρενθέσεων στη δήλωση της υπορουτίνας, σε κανονικοποιημένη μορφή: αφαιρούνται οι λευκοί χαρακτήρες, κάθε χαρακτήρας είναι ένα από τα γράμματα prototype παρακάτω.

Χαρακτήρα	Σημασία στο prototype
`$`	ένα βαθμωτό όρισμα
`@`	λίστα - απορροφά όλα τα υπόλοιπα ορίσματα
`%`	hash - απορροφά όλα τα υπόλοιπα ορίσματα ως λίστα
`&`	μπλοκ κώδικα ή αναφορά κώδικα
`*`	typeglob ή bareword που αντιμετωπίζεται ως typeglob
`\$ \@ \% \& \*`	αναφορά προς βαθμωτό / πίνακα / hash / κώδικα / glob (ο καλών γράφει `@ary`, ο καλούμενος λαμβάνει `\@ary`)
`\[...]`	αναφορά όπου γίνεται δεκτό οποιοδήποτε από τα sigils εντός αγκύλων
`+`	πίνακας ή hash, που περνά αυτόματα με αναφορά
`_`	βαθμωτό που, αν παραλειφθεί, παίρνει εξ ορισμού την `$_`
`;`	τα επόμενα ορίσματα είναι προαιρετικά

Ένα προπορευόμενο `&` χωρίς παρενθέσεις μετά, όπως στο `sub f (&@) { ... }`, είναι αυτό που επιτρέπει σε υπορουτίνες σαν τις `map`, `grep` και `sort` να δέχονται μπλοκ χωρίς κόμμα.

Καθολική κατάσταση που επηρεάζει

Η `prototype` χωρίς όρισμα διαβάζει την `$_` ως όνομα υπορουτίνας. Καμία άλλη καθολική δεν συμβουλεύεται ούτε τροποποιείται.

Παραδείγματα

Διαβάστε ένα prototype καθορισμένο από τον χρήστη:

```
sub add ($$)   { $_[0] + $_[1] }
print prototype 'add';                  # $$
print prototype \&add;                  # $$
```

Μια υπορουτίνα χωρίς prototype επιστρέφει `undef`:

```
sub plain { "hi" }
print defined(prototype 'plain') ? "yes" : "no";   # no
```

Ένα ερώτημα `CORE::` για ενσωματωμένη της οποίας η μορφή χωρά στη γλώσσα prototype:

```
print prototype 'CORE::push';           # \@@
print prototype 'CORE::keys';           # \[%@]
print prototype 'CORE::pos';            # ${\;}
```

Ένα ερώτημα `CORE::` για ενσωματωμένη που δεν μπορεί να εκφραστεί ως prototype:

```
print defined(prototype 'CORE::system') ? "yes" : "no";   # no
print defined(prototype 'CORE::print')  ? "yes" : "no";   # no
```

Χρησιμοποιήστε την για να κατασκευάσετε ένα περιτύλιγμα που κληρονομεί τη σύμβαση κλήσης του καλούμενου - χρήσιμο όταν γράφετε γενικούς διανομείς:

```
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 }";
}
```

Οριακές περιπτώσεις

- **Χωρίς όρισμα χρησιμοποιείται η `$_`.** Η `prototype` χωρίς τίποτα ανάμεσα σε αυτήν και τον επόμενο όρο συμβουλεύεται την `$_` ως όνομα υπορουτίνας. Ένας βρόχος όπως `for (@names) { print prototype, "\n" }` λειτουργεί κατά σύμπτωση αυτού του κανόνα.
- **Τα signatures δεν είναι prototypes.** Το `sub f ($x, $y) { ... }` δηλώνει signature - η `prototype 'f'` επιστρέφει `undef`, επειδή η υπορουτίνα δεν έχει prototype. Για να δηλώσετε και τα δύο, χρησιμοποιήστε το γνώρισμα `:prototype(...)`: `sub f :prototype($$) ($x, $y) { ... }`.

- **Άγνωστο όνομα επιστρέφει `undef`**. Αν η υπορουτίνα δεν έχει δηλωθεί ποτέ, η `prototype` επιστρέφει `undef` χωρίς να εγείρει σφάλμα. Συνδυάστε με `defined` και `exists &Some::Package::name` όταν χρειάζεται να ξεχωρίσετε το «χωρίς prototype» από το «δεν υπάρχει τέτοια υπορουτίνα».
- **Υπορουτίνες με auto-load**. Αν η υπορουτίνα χειρίζεται από `AUTOLOAD`, δεν έχει μεταγλωττιστεί ακόμη, οπότε η `prototype` αναφέρει `undef` ακόμη και όταν η τελικώς καλέσιμη θα είχε. Αναγκάστε πρώτα ένα stub (`*foo = \&foo` μετά τη φόρτωση) αν πρέπει να το ζητήσετε.
- **Οι αναφορές έναντι των ονομάτων είναι εναλλάξιμες**. Τα `prototype \&foo` και `prototype 'foo'` επιστρέφουν την ίδια συμβολοσειρά· επιλέξτε όποιο είναι φθηνότερο στο σημείο κλήσης. Μια αναφορά παρακάμπτει την αναζήτηση ονόματος πακέτου.
- **Πλήρως προσδιορισμένα ονόματα**. Τα μη προσδιορισμένα ονόματα αναζητούνται στο τρέχον πακέτο. Για κλήσεις από κλήσεις, να προσδιορίζετε πάντοτε: `prototype 'Other::Pkg::fn'`.
- **Το κενό prototype δεν είναι `undef`**. Το `sub nullary () { 42 }` δίνει στην `prototype 'nullary'` την κενή συμβολοσειρά `""`, που είναι ορισμένη και ψευδής. Χρησιμοποιήστε `defined`, όχι έλεγχο αλήθειας, για να ελέγξετε αν «έχει prototype».

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `sub` - δηλώστε την υπορουτίνα της οποίας το prototype πρόκειται να διαβάσετε
- `defined` - ο σωστός έλεγχος για το «έχει αυτή η υπορουτίνα prototype;», αφού ένα prototype κενής συμβολοσειράς είναι έγκυρο
- `ref` - συντρόφισσα συνάρτηση επιθεώρησης όταν το `FUNCTION` είναι αναφορά κώδικα και θέλετε να επαληθεύσετε τον τύπο αναφοράς πριν καλέσετε την `prototype`
- `caller` - γειτονική συνάρτηση εσωτερικής σκόπευσης που απαντά «ποιος με κάλεσε», χρησιμοποιείται συχνά μαζί με την `prototype` κατά την κατασκευή περιτυλιγμάτων
- `wantarray` - ερώτημα περιβάλλοντος σε χρόνο εκτέλεσης, το αντίστοιχο της επιθεώρησης prototype κατά τη μεταγλώττιση
- `perlsub` - η μικρογλώσσα prototype και πώς την εφαρμόζει ο αναλυτής στα σημεία κλήσης

Πίνακες

push

Προσαρτά μία ή περισσότερες τιμές στο τέλος ενός πίνακα.

Η push προσθέτει κάθε στοιχείο της LIST στο τέλος του ARRAY, με τη σειρά. Είναι το ζευγαρωμένο αντίστοιχο της pop στην ουρά και των unshift / shift στην κεφαλή. Ο πίνακας μεγαλώνει επιτόπου· δεν επιστρέφεται νέος πίνακας.

Σύνοψη

```
push @arr, $val
push @arr, @more
push @arr, list, of, values
push @$aref, $val
```

Τι επιστρέφεται

Ο **νέος αριθμός στοιχείων** του ARRAY μετά την προσάρτηση - δηλαδή το `scalar @arr` όπως διαμορφώνεται μόλις επιστρέψει η push. Όχι οι ωθημένες τιμές, όχι το παλιό μήκος, όχι ο ίδιος ο πίνακας.

```
my @colors = ("red");
my $n = push @colors, "blue", "green";   # $n is 3
```

Χρήσιμο όταν η push και ο επακόλουθος έλεγχος μήκους θα ήταν διαφορετικά δύο δηλώσεις:

```
if (push @queue, $job > $limit) { ... }  # one call, count + test
```

Παραδείγματα

Βασική προσάρτηση μίας τιμής:

```
my @animals = ("cat");
push @animals, "mouse";                  # ("cat", "mouse")
```

Προσάρτηση πολλαπλών τιμών σε μία κλήση - κάθε όρισμα γίνεται δικό του στοιχείο:

```
my @colors = ("red");
push @colors, "blue", "green";           # ("red", "blue", "green")
```

Προσάρτηση των περιεχομένων άλλου πίνακα. Ο δεύτερος πίνακας ισοπεδώνεται στη λίστα, οπότε προσαρτώνται τα **στοιχεία** του, όχι ο πίνακας ως μονό στοιχείο:

```
my @a = (1, 2);
my @b = (3, 4);
push @a, @b;                             # (1, 2, 3, 4)
```

Πειθαρχία ουράς - push στην ουρά, shift στην κεφαλή:

```
my @queue;
push @queue, $job;                          # enqueue
my $next = shift @queue;                    # dequeue
```

Πειθαρχία στοίβας - push και pop στο ίδιο άκρο:

```
my @stack;
push @stack, $frame;                        # enter
my $top = pop @stack;                       # leave
```

Push σε πίνακα που κρατείται μέσω αναφοράς. Η αναφορά πρέπει να αποαναφερθεί με `@{ ... }` (ή τη σύντομη μορφή `@$ref`) πριν η push δει πίνακα:

```
my $aref = [1, 2];
push @$aref, 3;                             # $aref now refers to [1,␣
↪2, 3]
push @{ $aref }, 4, 5;                      # same, explicit deref
```

Μέτρηση στοιχείων μετά την προσάρτηση χωρίς προσωρινή μεταβλητή:

```
my $size = push @log, $entry;
warn "log is large: $size entries" if $size > 1000;
```

Οριακές περιπτώσεις

- **Το arrayref χρειάζεται αποαναφορά.** Το `push $ref, $x` είναι σφάλμα μεταγλώττισης στην Perl 5.24+. Χρησιμοποιήστε `push @$ref, $x` ή `push @{ $ref }, $x`. Ένα πειραματικό χαρακτηριστικό επέτρεπε βαθμωτό πρώτο όρισμα στις 5.14–5.22 και αφαιρέθηκε· μην βασίζεστε σε αυτό.
- **Τα ορίσματα πίνακα ισοπεδώνονται αυτόματα.** Η `push @a, @b` προσαρτά τα στοιχεία του `@b` μεμονωμένα. Για να προσαρτήσετε το `@b` ως μία εμφωλευμένη εγγραφή χρειάζεστε arrayref: `push @a, \@b` ή `push @a, [ @b ]`.
- **Και τα ορίσματα hash ισοπεδώνονται.** Η `push @a, %h` προσαρτά εναλλάξ κλειδιά και τιμές με τη σειρά επανάληψης του hash - σχεδόν ποτέ αυτό που θέλετε. Χρησιμοποιήστε `push @a, \%h` για να αποθηκεύσετε το hash μέσω αναφοράς.
- **Περιβάλλον λίστας αλλά βαθμωτή επιστροφή.** Η push τεκμηριώνεται ότι δέχεται `LIST` και επομένως αποτιμά τα ορίσματά της σε περιβάλλον λίστας, αλλά η τιμή που η ίδια η push παράγει είναι πάντα ο προκύπτων αριθμός. Μέσα σε `print push @a, @b` εκτυπώνεται ο αριθμός, όχι ο πίνακας.
- **Κενή `LIST`.** Η `push @a;` (χωρίς δεύτερο όρισμα) είναι no-op και επιστρέφει το τρέχον `scalar @a`. Νόμιμη και περιστασιακά χρήσιμη για το «δώσε μου τον αριθμό χωρίς να τροποποιήσεις τίποτα» - αν και το `scalar @a` είναι σαφέστερο.
- **Tied πίνακες.** Η push σε tied πίνακα καλεί την `PUSH` στο tied αντικείμενο αν είναι ορισμένη· διαφορετικά καταφεύγει σε επαναλαμβανόμενες κλήσεις `STORE` σε θέσεις παραγόμενες από `FETCHSIZE`. Δείτε το `perltie`.
- **Αυτο-δημιουργία.** Η push μέσω αναφοράς με τιμή undef δημιουργεί έναν ανώνυμο πίνακα:

```
my %index;
push @{ $index{$key} }, $value;          # $index{$key} becomes
 ↪an arrayref
```

Μετά την πρώτη push, το `$index{$key}` κρατά arrayref που περιέχει το `$value`. Αυτό είναι το τυπικό ιδίωμα για τη σταδιακή δόμηση ενός hash-of-arrays.

- **Πίνακας μόνο για ανάγνωση.** Η push σε πίνακα μόνο για ανάγνωση ή σταθερού μεγέθους (π.χ. στοιχείο του `@_` δεσμευμένο σε μεταβλητή του καλούντος, ή πίνακας κλειδωμένος με `Hash::Util::lock_value`) εγείρει `Modification of a read-only value attempted`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `pop` - αφαιρεί και επιστρέφει το τελευταίο στοιχείο· ζευγαρωμένη με την push σχηματίζει στοίβα
- `shift` - αφαιρεί και επιστρέφει το πρώτο στοιχείο· ζευγαρωμένη με την push σχηματίζει ουρά
- `unshift` - η κατοπτρική εικόνα της push στην κεφαλή του πίνακα· επιστρέφει το νέο μήκος με τον ίδιο τρόπο
- `splice` - εισαγωγή, αφαίρεση, ή αντικατάσταση οποιουδήποτε εύρους· η `push @a, @b` είναι ισοδύναμη με `splice @a, scalar @a, 0, @b`
- `scalar` - επιβολή βαθμωτού περιβάλλοντος σε πίνακα όταν θέλετε τον αριθμό χωρίς push
- `@ARGV` - ο κανονικός πίνακας στον οποίο τα περισσότερα προγράμματα κάνουν push ή τον καταναλώνουν με `shift`

Βαθμωτά και συμβολοσειρές

q//

Συμβολοσειρά με μονά εισαγωγικά και ελεύθερη επιλογή οριοθέτη.

Η `q//` είναι η γενική μορφή της συνηθισμένης συμβολοσειράς `'...'`. Παράγει μια σταθερά συμβολοσειράς κατά τη μεταγλώττιση, **δεν** παρεμβάλλει μεταβλητές και **δεν** επεξεργάζεται τις ακολουθίες διαφυγής με ανάστροφη κάθετο που αναγνωρίζουν η `qq//` και τα regex literals. Καταφύγετε στην `q//` όποτε το literal που θέλετε να γράψετε θα ήταν δύσχρηστο υπό την `'...'` - συνηθέστερα επειδή η ίδια η συμβολοσειρά περιέχει μονά εισαγωγικά - ή όταν ένας διαφορετικός οριοθέτης απλώς διαβάζεται καλύτερα στο πλαίσιο.

Σύνοψη

```
q/STRING/
q(STRING)
q[STRING]
q{STRING}
q<STRING>
q!STRING!
q#STRING#
'STRING'
```

Οποιοσδήποτε μη αλφαριθμητικός χαρακτήρας λειτουργεί ως οριοθέτης. Τα τέσσερα ζεύγη αγκυλών ASCII (`()`, `[]`, `{}`, `<>`) εμφωλεύονται· όλοι οι άλλοι οριοθέτες όχι. Η `'...'` είναι σύντμηση για το `q(...)` με ταυτόσημη σημασιολογία.

Τι επιστρέφεται

Μια συμβολοσειρά σε βαθμωτό περιβάλλον - η ακριβής ακολουθία bytes μεταξύ του οριοθέτη ανοίγματος και κλεισίματος, με δύο επανεγγραφές:

- Το `\\` γίνεται μία ανάστροφη κάθετος.
- Το `\<delim>` γίνεται κυριολεκτικό αντίγραφο του χαρακτήρα οριοθέτη (αυτό σας επιτρέπει να συμπεριλάβετε τον οριοθέτη μέσα στο σώμα).

Κάθε άλλος χαρακτήρας, περιλαμβανομένης κάθε άλλης `\`, μένει κατά λέξη. Δεν υπάρχει `\n`, ούτε `\t`, ούτε επέκταση `$var`, ούτε `\x{...}`, ούτε `\N{...}` - το εσωτερικό της `q//` πλησιάζει όσο μπορεί η Perl στο «ό,τι πληκτρολογήσατε, αυτό παίρνετε».

Σε περιβάλλον λίστας, η `q//` εξακολουθεί να αποτιμάται στην ίδια μοναδική συμβολοσειρά.

Οριοθέτες και εμφώλευση

Οι μη αγκύλωτοι οριοθέτες χρησιμοποιούν τον ίδιο χαρακτήρα στην αρχή και στο τέλος:

```
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
```

Οι αγκυλωτοί οριοθέτες εμφωλεύονται, οπότε ισορροπημένες αγκύλες μέσα στο σώμα είναι εντάξει και μόνο ένα μη ισορροπημένο κλείσιμο τερματίζει τη συμβολοσειρά:

```
my $c = q{foo{bar}baz};    # "foo{bar}baz"
my $d = q(f(o(o)b)ar);     # "f(o(o)b)ar"
```

Οι λευκοί χαρακτήρες μεταξύ της `q` και του οριοθέτη επιτρέπονται **εκτός** όταν ο οριοθέτης είναι `#` - το `q#foo#` είναι η συμβολοσειρά `"foo"`, αλλά το `q #foo#` είναι ο τελεστής `q` ακολουθούμενος από σχόλιο γραμμής, και το σώμα της συμβολοσειράς συνεχίζεται στην επόμενη γραμμή. Όταν ο οριοθέτης είναι χαρακτήρας λέξης (ταιριάζει με `/\w/`), οι λευκοί χαρακτήρες μεταξύ της `q` και του οριοθέτη είναι **υποχρεωτικοί**: το `q XfooX` είναι `"foo"`, ενώ το `qXfooX` αποτελεί συντακτικό σφάλμα.

Κανόνες διαφυγής

Μέσα στην q// η ανάστροφη κάθετος έχει ακριβώς δύο ειδικούς ρόλους:

Ακολουθία	Παράγει
\\	μία ανάστροφη κάθετο
\<delim>	κυριολεκτικό αντίγραφο του οριοθέτη

Οτιδήποτε άλλο ξεκινά με \ παραμένει ως δύο χαρακτήρες:

```
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
```

Αυτή είναι η πιο συχνή πηγή σύγχυσης για όσους έρχονται από συμβολοσειρές με διπλά εισαγωγικά - στην q// και στην '...', το \n είναι μια ανάστροφη κάθετος ακολουθούμενη από το γράμμα n, **όχι** χαρακτήρας νέας γραμμής.

Παραδείγματα

Ενσωμάτωση και των δύο εισαγωγικών χωρίς διαφυγή:

```
my $msg = q{I said "hello", she said 'hi'.};
```

Διαδρομή τύπου Windows, όπου οι ανάστροφες κάθετοι είναι άφθονες και η παρεμβολή με διπλά εισαγωγικά θα ήταν κίνδυνος:

```
my $path = q(C:\Users\alice\Documents);
print $path;                  # C:\Users\alice\Documents
```

Ένα κυριολεκτικό $ χωρίς παρεμβολή - η qq// θα απαιτούσε διαφυγή κάθε $ και @, η q// τα αφήνει ως έχουν:

```
my $expr = q($total = $x + $y);
print $expr;                  # $total = $x + $y
```

Ο μοναδικός τρόπος να βάλετε τον οριοθέτη μέσα στο σώμα: διαφυγή του με ανάστροφη κάθετο. Σημειώστε ότι η ίδια η ανάστροφη κάθετος καταναλώνεται.

```
my $s = q!He said "ouch\!" loudly!;
print $s;                     # He said "ouch!" loudly
```

Ένα μοτίβο regex αποθηκευμένο ως απλή συμβολοσειρά, που μεταβιβάζεται σε qr// στο σημείο χρήσης - η q// κρατά τις ανάστροφες κάθετους κυριολεκτικές ώστε το μοτίβο να επιβιώσει ακέραιο:

```
my $pat = q(\d{3}-\d{4});
if ($phone =~ /$pat/) { ... }
```

Πολυγραμμικό literal χωρίς επεξεργασία διαφυγής - κάθε αλλαγή γραμμής στην πηγή γίνεται αλλαγή γραμμής στη συμβολοσειρά:

```
my $sql = q{
    SELECT id, name
    FROM   users
    WHERE  active = 1
};
```

Οριακές περιπτώσεις

- **Η q// είναι μεταγλώττισης.** Ο αναλυτής επιλύει τη συμβολοσειρά όταν διαβάζεται η πηγή· δεν υπάρχει κόστος χρόνου εκτέλεσης πέρα από τη χρήση της ήδη κατασκευασμένης σταθεράς. Όπως και η `'...'`.
- **Οι `'...'` και `q(...)` είναι αδιαφοροποίητες** μετά τη συντακτική ανάλυση. Η επιλογή είναι αμιγώς θέμα ύφους - διαλέξτε όποια διαβάζεται καλύτερα για το περιεχόμενο.
- **Κενή συμβολοσειρά.** Τόσο το `q()` όσο και το `''` δίνουν τη συμβολοσειρά μηδενικού μήκους.
- **Κανένα `\n`, κανένα `\t`, καμία παρεμβολή**, ποτέ. Αν χρειάζεστε κάτι από αυτά, μεταβείτε σε `qq//` ή `"..."`. Η ανάμιξη των δύο στην ίδια γραμμή είναι εντάξει: `q(literal ) . qq(interpolated $x)`.
- **Ανάστροφη κάθετος στο τέλος του σώματος.** Μία `\` αμέσως πριν τον οριοθέτη κλεισίματος τον διαφεύγει, οπότε το `q(foo\)` είναι συντακτικό σφάλμα - το `\ )` γίνεται κυριολεκτικό `)` και η συμβολοσειρά μένει χωρίς τερματισμό. Γράψτε `q(foo\\)` για να τελειώσετε μια συμβολοσειρά με μία ανάστροφη κάθετο.
- **Μη ισορροπημένες αγκύλες με αγκυλωτούς οριοθέτες.** Επειδή οι αγκύλες εμφωλεύονται, το `q{foo{bar}` αποτελεί συντακτικό σφάλμα - η εσωτερική `{` ανοίγει εμφωλευμένο επίπεδο και η εξωτερική `}` το κλείνει, αφήνοντας τη συμβολοσειρά χωρίς τερματισμό. Χρησιμοποιήστε μη αγκυλωτό οριοθέτη, ή ισορροπήστε τις αγκύλες, ή διαφύγετε μία από αυτές: `q{foo\{bar}`.
- **Το `#` ως οριοθέτης δεν χρειάζεται κενό από την `q`**, αλλά κάθε άλλος οριοθέτης που είναι χαρακτήρας λέξης (γράμματα, ψηφία, `_`) απαιτεί λευκό χαρακτήρα μεταξύ της `q` και του οριοθέτη ανοίγματος.
- **Η εισαγωγή κώδικα Perl σε εισαγωγικά είναι εύθραυστη.** Το `q{ if ($x eq "}") { ... } }` αποτελεί συντακτικό σφάλμα: η `}` μέσα στην ενσωματωμένη συμβολοσειρά με διπλά εισαγωγικά κλείνει το εξωτερικό άγκιστρο πριν η Perl δει την εσωτερική συμβολοσειρά. Εργαλεία όπως το `Text::Balanced` υπάρχουν επειδή το χειρόγραφο γράψιμο αξιόπιστου quoter για πηγαία Perl είναι δυσάρεστο.
- **Ασάφεια αναλυτή με `/` ως οριοθέτη.** Η `q/.../` χρησιμοποιεί τη `/` με τον ίδιο τρόπο που τη χρησιμοποιεί η `m//`. Μέσα στο σώμα δεν ισχύει σύνταξη regex, αλλά μια πλάγια κάθετος στο περιεχόμενο θα τερματίσει τη συμβολοσειρά πρόωρα - επιλέξτε διαφορετικό οριοθέτη (`q{...}`, `q(...)`) όταν εμφανίζονται πλάγιες καθέτους στο κείμενο.
- **Τα heredocs με `<<'TAG'`** ακολουθούν τους ίδιους κανόνες διαφυγής με την `q//`: καμία παρεμβολή, καμία επεξεργασία διαφυγής πέραν της `\\` και του ότι το `\TAG` δεν είναι ειδικό (το tag τερματίζει στη δική του γραμμή). Χρησιμοποιήστε heredoc με μονά εισαγωγικά όταν θέλετε σημασιολογία `q//` σε πολλές γραμμές χωρίς οριοθέτη κλεισίματος στη μέση του κειμένου.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `qq//` - literal διπλών εισαγωγικών· ίδια ευελιξία οριοθέτη με την `q//` συν παρεμβολή μεταβλητών και το πλήρες λεξιλόγιο διαφυγής `\n/\t/\x{...}`
- `qw//` - λίστα λέξεων χωρισμένων με λευκούς χαρακτήρες· αυτό που θέλετε όταν το literal είναι λίστα σύντομων tokens, όχι μία συμβολοσειρά
- `qr//` - μεταγλώττιση literal regex· το μέλος της οικογένειας quote-like που αφορά αντιστοίχιση μοτίβων
- `qx//` - εκτελεί μια εντολή στο κέλυφος και συλλαμβάνει την έξοδό της· ίδιοι κανόνες οριοθέτη, σημασιολογία ανάστροφων εισαγωγικών
- `quotemeta` - αντίστοιχο σε χρόνο εκτέλεσης του `\Q...\E`, χρήσιμο όταν χρειάζεται να τροφοδοτήσετε ένα literal `q//` σε regex ως σταθερή συμβολοσειρά
- `$"` - διαχωριστής λίστας που χρησιμοποιείται όταν πίνακες παρεμβάλλονται μέσα σε `qq//`· άσχετο με την ίδια την `q//`, αλλά ο συνηθισμένος λόγος που κάποιος καταφεύγει στην `q//` είναι ακριβώς για να **αποφύγει** αυτόν τον μηχανισμό

Βαθμωτά και συμβολοσειρές

qq//

Κατασκευάζει μια συμβολοσειρά σε διπλά εισαγωγικά με παρεμβολή, με οριοθέτη της επιλογής σας.

Το `qq//` είναι η μακρά μορφή του κυριολεκτικού συμβολοσειράς `"..."`. Παράγει την ίδια συμβολοσειρά που θα παρήγαγε το `""`, οπότε τα `qq/hello $name\n/` και `"hello $name\n"` είναι εναλλάξιμα. Το νόημα του qq είναι ο οριοθέτης: όταν το περιεχόμενο περιέχει `"` (ή ακολουθίες backslash που θα παρενέβαιναν στα διπλά εισαγωγικά), επιλέγετε διαφορετικό οριοθέτη και διατηρείτε τη συμβολοσειρά αναγνώσιμη αντί να τη γεμίζετε με `\"`.

Σύνοψη

```
qq/STRING/
qq{STRING}
qq(STRING)
qq[STRING]
qq<STRING>
qqXSTRINGX        # any non-word, non-whitespace char works as␣
↪delimiter
```

Τι επιστρέφεται

Μια βαθμωτή συμβολοσειρά. Οι μεταβλητές που αναφέρονται μέσα παρεμβάλλονται τη στιγμή που αποτιμάται το qq// (όχι κατά την ανάλυση)· οι ακολουθίες διαφυγής με backslash επεκτείνονται. Το αποτέλεσμα δεν έχει υπονοούμενο newline - πρέπει να γράψετε ρητά \n.

Επιλογή οριοθέτη

Οποιοσδήποτε χαρακτήρας που δεν είναι χαρακτήρας λέξης (/\w/) και δεν είναι λευκός χαρακτήρας μπορεί να ακολουθήσει το qq. Τα τέσσερα ζεύγη αγκυλών εμφωλεύονται:

```
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>>
```

Οι οριοθέτες που δεν είναι αγκύλες χρησιμοποιούν τον ίδιο χαρακτήρα και στα δύο άκρα και δεν εμφωλεύονται:

```
qq|one|two|                # SYNTAX ERROR - first | closes
qq/a\/b/                   # "a/b" - backslash escapes the delimiter
```

Αν ο οριοθέτης είναι χαρακτήρας λέξης, απαιτείται λευκός χαρακτήρας μεταξύ του qq και του οριοθέτη:

```
qq XfooX                   # OK - "foo"
qqXfooX                    # WRONG - parsed as identifier qqXfooX
```

Το # ως οριοθέτης δεν χρειάζεται κενό και είναι περιστασιακά χρήσιμο για συμβολοσειρές που περιέχουν κάθε άλλο σημείο στίξης:

```
qq#He said "hi" & waved.#  # "He said \"hi\" & waved."
```

Το χαρακτηριστικό `extra_paired_delimiters` (5.36+) αναγνωρίζει ένα ευρύ σύνολο ζευγών αγκυλών Unicode ως οριοθέτες που εμφωλεύονται· ενεργοποιήστε το με `use feature 'extra_paired_delimiters'` όταν θέλετε, για παράδειγμα, εισαγωγικά γαλλικού τύπου ή αγκύλες CJK.

Τι παρεμβάλλεται

- **Βαθμωτά**: `$x`, `$hash{key}`, `$array[0]`, `$ref->{k}[2]`.
- **Πίνακες**: το `@arr` επεκτείνεται σε σειρά, με τα στοιχεία να ενώνονται από το `$"` (προεπιλογή `" "`). Ισοδύναμο με `join $", @arr`.
- **Τομές**: `@h{qw(a b)}`, `@a[1..3]`.
- **Αποαναφορές με βέλη και δείκτες**: `$ref->{key}[0]`.

Τι **δεν** παρεμβάλλεται:

- **Κλήσεις μεθόδων**: το `$obj->meth` μέσα σε qq// παρεμβάλλει την `$obj` και αφήνει το κυριολεκτικό κείμενο `->meth` μέσα στη συμβολοσειρά.

- **Κλήσεις συναρτήσεων**: η συνηθισμένη παράκαμψη είναι το `@{[ func() ]}` - η αποαναφορά πίνακα μέσω αναφοράς εκτελεί αυθαίρετο κώδικα και ενσωματώνει το αποτέλεσμά του.
- **Πίνακες σημείων στίξης** εκτός των `@_`, `@+`, `@-`: παρεμβάλλονται μόνο σε άγκιστρα (`@{*}`). Το γυμνό `@*` είναι κυριολεκτικό `@*`.

```
my $obj = Some::Class->new;
qq/$obj->name/;                          # interpolates $obj, leaves "-
 ↪>name"
qq/@{[ $obj->name ]}/;                   # calls the method,␣
 ↪interpolates result
```

Ακολουθίες διαφυγής

Διαθέσιμες μέσα σε `qq//` (και οπουδήποτε αλλού γίνεται παρεμβολή):

Ακολουθία	Σημασία
`\t`	στηλοθέτης (HT)
`\n`	newline (εικονικός τερματιστής γραμμής της πλατφόρμας)
`\r`	επιστροφή φορείου (CR)
`\f`	αλλαγή σελίδας (FF)
`\b`	οπισθοδιαγραφή (BS)
`\a`	συναγερμός / κουδούνι (BEL)
`\e`	διαφυγή (ESC)
`\0`	NUL
`\\`	κυριολεκτικό backslash
`\"`	κυριολεκτικό διπλό εισαγωγικό
`\x1b`	δεκαεξαδικός χαρακτήρας, 0x00–0xFF
`\x{263A}`	δεκαεξαδικός χαρακτήρας, οποιοδήποτε σημείο κώδικα (μορφή με άγκιστρα)
`\o{23072}`	οκταδικός χαρακτήρας, οποιοδήποτε σημείο κώδικα
`\033`	οκταδικός χαρακτήρας, 000–777 (προτιμήστε το `\o{}`)
`\cA`	χαρακτήρας ελέγχου - 64 XOR με τον επόμενο χαρακτήρα σε κεφαλαία
`\N{GREEK SMALL LETTER ALPHA}`	ονομαστικός χαρακτήρας Unicode (χρειάζεται `use charnames`)
`\N{U+03B1}`	Χαρακτήρας Unicode κατά σημείο κώδικα (πάντα Unicode, ακόμη και σε EBCDIC)

Οι τροποποιητές πεζών-κεφαλαίων και παράθεσης ισχύουν μέχρι το `\E` ή το τέλος της συμβολοσειράς:

Ακολουθία	Σημασία
`\l`	μετατροπή μόνο του επόμενου χαρακτήρα σε πεζό
`\u`	μετατροπή μόνο του επόμενου χαρακτήρα σε titlecase (όχι κεφαλαία!)
`\L`	πεζά μέχρι το `\E`
`\U`	κεφαλαία μέχρι το `\E`
`\F`	foldcase μέχρι το `\E`
`\Q`	παράθεση μεταχαρακτήρων regex μέχρι το `\E` (καλεί την `quotemeta`)
`\E`	τερματίζει το πιο πρόσφατο `\L` / `\U` / `\F` / `\Q`

Τα `\L`, `\U`, `\F`, `\Q` στοιβάζονται - καθένα χρειάζεται το δικό του `\E`.

```
say "\u$name";                          # capitalise first letter
say "\U$name\E's account";              # uppercase $name, rest␣
 ↪untouched
say "match $file literally: \Q$file\E";
```

Το `\Q` μέσα σε `qq//` εφαρμόζεται **μετά** την παρεμβολή, οπότε το `qq/\Q$pat\E/` ισούται με `quotemeta($pat)`. (Στους τελεστές παράθεσης regex `qr//` και `m//`, το `\Q` εφαρμόζεται μετά την παρεμβολή αλλά πριν τη μεταγλώττιση του regex - η διαφορά έχει σημασία όταν η ίδια η `$pat` περιέχει μεταχαρακτήρες regex που θέλετε να μείνουν κυριολεκτικοί.)

Η Perl δεν έχει διαφυγή `\v` στα κυριολεκτικά συμβολοσειράς - είναι μεταχαρακτήρας regex. Χρησιμοποιήστε `\x0b`, `\cK`, ή `\N{VT}` για κατακόρυφο στηλοθέτη.

Καθολική κατάσταση που επηρεάζει

- `$"` - διαχωριστής λίστας που χρησιμοποιείται κατά την παρεμβολή πινάκων και τομών. Προεπιλογή `" "`. Αλλάξτε με `local $" = ","` για μία εμβέλεια.

Καμία άλλη καθολική του διερμηνέα δεν συμμετέχει στην ίδια την αποτίμηση του `qq//`. Οι παρεμβαλλόμενες μεταβλητές είναι φυσικά ό,τι τιμές έχουν τη στιγμή που εκτελείται το `qq//`.

Παραδείγματα

Απλή παρεμβολή, ταυτόσημη σε σημασία με τη μορφή `"..."`:

```
my $name = "world";
my $s = qq/hello, $name!\n/;            # "hello, world!\n"
```

Συμβολοσειρά που περιέχει διπλά εισαγωγικά - το `qq` με διαφορετικό οριοθέτη αποφεύγει τη συσσώρευση backslash:

```
my $msg = qq{He said "yes" and then "maybe".};
# vs:   "He said \"yes\" and then \"maybe\"."
```

Πολυγραμμικό απόσπασμα HTML με εμφωλευμένα άγκιστρα - το ζεύγος `( )` εμφωλεύεται:

```
my $user = "rj";
my $html = qq(
  <p>Welcome, $user.</p>
  <p>You have @{[ scalar @inbox ]} messages.</p>
);
```

Παρεμβολή πίνακα με προσαρμοσμένο διαχωριστή:

```
my @tags = qw(perl rust docs);
local $" = ", ";
say qq/tags: @tags/;                    # "tags: perl, rust, docs"
```

Παγίδα κλήσης μεθόδου - το `->` είναι κυριολεκτικό μέσα σε `qq//`:

```
my $obj = File::Spec->new;
qq/path is $obj->rel2abs('.')/;         # "path is <stringified $obj>-
 ↪>rel2abs('.')"
qq/path is @{[ $obj->rel2abs('.') ]}/;
                                        # actually calls rel2abs()
```

Κατασκευή αποσπάσματος SQL όπου η `$ident` δεν είναι έμπιστη - το `\Q..\E` διατηρεί τους ενσωματωμένους %, _ και τα backslashes κυριολεκτικά υπό μια ρήτρα `LIKE`, μόνο αφού έχετε φροντίσει για κατάλληλη παραμετροποίηση· για το πραγματικό escaping SQL χρησιμοποιήστε την παράθεση του οδηγού, όχι το `\Q`:

```
my $needle = "100% sure";
my $pat = qq/\Q$needle\E/;              # the regex-safe form
# $pat eq "100\\%\\ sure"
```

Στοιβαγμένοι τροποποιητές:

```
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.
```

Οριακές περιπτώσεις

- **Επιλογή οριοθέτη κατά την ανάλυση**. Η Perl επιλέγει τον τερματικό οριοθέτη τη στιγμή που διαβάζει τον αρχικό. Το `qq/.../` αναμένει `/`· δεν υπάρχει τρόπος αλλαγής στη μέση της συμβολοσειράς. Αν χρειάζεστε και τα δύο είδη αγκυλών, επιλέξτε οριοθέτη που δεν είναι αγκύλη.
- **Backslash στο τέλος συμβολοσειράς με οριοθέτη που δεν είναι αγκύλη**. Το `qq/a\/` είναι `a/` (το backslash διαφεύγει του οριοθέτη, η συμβολοσειρά συνεχίζει)· για κυριολεκτικό τελικό backslash, αλλάξτε οριοθέτη: `qq{a\\}`.
- **Καμία εμφώλευση για οριοθέτες που δεν είναι αγκύλες**. Το `qq|a|b|` είναι η συμβολοσειρά `a` ακολουθούμενη από το αναγνωριστικό `b` ακολουθούμενη από `|` - συντακτικό σφάλμα στα περισσότερα περιβάλλοντα. Επιλέξτε οριοθέτες-αγκύλες όταν το περιεχόμενο περιέχει τον χαρακτήρα του οριοθέτη.

- **Το # ως οριοθέτης** είναι ειδική περίπτωση μόνο λόγω της αλληλεπίδρασης με τα σχόλια: το qq#...# είναι συμβολοσειρά, αλλά το qq #...# (με λευκό χαρακτήρα) είναι ο τελεστής qq ακολουθούμενος από σχόλιο, με τη συμβολοσειρά να λαμβάνεται από την επόμενη γραμμή. Αυτό αντικατοπτρίζει τη συμπεριφορά όλων των τελεστών της οικογένειας q.
- **Κενή συμβολοσειρά**. Το qq// είναι η κενή συμβολοσειρά, ίδια με την "".
- **Παρεμβαλλόμενη undef**. Μετατρέπεται σε κενή συμβολοσειρά και, υπό use warnings, πυροδοτεί προειδοποίηση uninitialized στο σημείο της παρεμβολής, όχι στο σημείο όπου η $var ανατέθηκε σε undef.
- **$ ή @ ακολουθούμενο από μη-αναγνωριστικό**. Μένει κυριολεκτικό: τα qq/cost: \$5/ και qq/cost: $5/ παράγουν και τα δύο cost: $5 (δεν υπάρχει μεταβλητή με όνομα $5 με την έννοια του αναγνωριστικού της Perl 5 - η $5 είναι σύλληψη regex, η οποία υπάρχει όντως, οπότε η δεύτερη μορφή παρεμβάλλει την ομάδα σύλληψης 5, ό,τι κι αν είναι αυτή τη στιγμή). Για κυριολεκτικό $ ή @, διαφύγετέ το: \$, \@.
- **\N{...} χωρίς charnames**. Οι ονομαστικοί χαρακτήρες απαιτούν να έχει φορτωθεί στην εμβέλεια use charnames ':full' (ή κάτι αντίστοιχο)· χωρίς αυτό λαμβάνετε σφάλμα κατά τη μεταγλώττιση. Το \N{U+HEX} και οι μορφές μονού ονόματος στο σύνολο :loose λειτουργούν και χωρίς αυτό.
- **Αντιστοίχιση πεζών-κεφαλαίων ευαίσθητη στο locale**. Με use locale που περιλαμβάνει το LC_CTYPE, τα \l, \u, \L, \U χρησιμοποιούν τους πίνακες πεζών-κεφαλαίων του τρέχοντος locale. Χωρίς αυτό, χρησιμοποιούνται οι πίνακες πεζών-κεφαλαίων του Unicode για σημεία κώδικα πάνω από 0xFF, και οι κανόνες ASCII πιο κάτω. Το \F χρησιμοποιεί το foldcase του Unicode ανεξάρτητα από το locale (εκτός υπό locale UTF-8, όπου ταιριάζει με το \L).
- **Μόνο ένα επίπεδο**. Το αποτέλεσμα της παρεμβολής δεν επανασαρώνεται. Το my $v = '$x'; qq/$v/ είναι η τεσσάρων χαρακτήρων συμβολοσειρά $x, όχι η τιμή της $x.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- q// - κυριολεκτικό σε μονά εισαγωγικά, χωρίς παρεμβολή, ίδιοι κανόνες διαχωριστή
- qw// - κυριολεκτικό λίστας λέξεων, παράγει λίστα συμβολοσειρών χωρισμένων σε λευκούς χαρακτήρες
- qx// - παρεμβαλλόμενη συμβολοσειρά που εκτελείται ως εντολή κελύφους, αποτυπώνει το stdout
- qr// - παρεμβαλλόμενο regex, μεταγλωττίζεται μία φορά και είναι επαναχρησιμοποιήσιμο
- sprintf - καταφύγετε σε αυτή όταν η εργασία είναι η μορφοποίηση αριθμών ή η συμπλήρωση πεδίων, όχι η παρεμβολή μεταβλητών σε κείμενο

- `quotemeta` - η συνάρτηση που καλεί το `\Q..\E`
- `join` - αυτό που κάνει στα κρυφά η παρεμβολή πινάκων, με την `$"` ως διαχωριστή
- `$"` - διαχωριστής λίστας που χρησιμοποιείται όταν ένας πίνακας παρεμβάλλεται σε συμβολοσειρά διπλών εισαγωγικών
- *Τελεστές παράθεσης και τελεστές τύπου παράθεσης* - ο πλήρης πίνακας των `q`, qq, `qx`, `qw`, `qr`, `m`, `s`, `tr` και των κοινών τους κανόνων οριοθέτη και διαφυγής

Κανονικές εκφράσεις και αντιστοίχιση μοτίβων

qr//

Μεταγλωττίζει ένα μοτίβο μία φορά και επιστρέφει ένα επαναχρησιμοποιήσιμο αντικείμενο regex.

Η `qr//` περικλείει σε εισαγωγικά το `STRING` της ως κανονική έκφραση, την παρεμβάλλει με τον ίδιο τρόπο που το κάνει η `m`, και επιστρέφει ένα αντικείμενο `Regexp` που μπορείτε να αποθηκεύσετε, να περάσετε ως όρισμα και να ενσωματώσετε σε άλλα μοτίβα. Το μεταγλωττισμένο μοτίβο μεταφέρει μαζί του τους τροποποιητές του, οπότε κάθε μεταγενέστερη χρήση βλέπει τις σημαίες με τις οποίες χτίστηκε.

Σύνοψη

```
my $re = qr/STRING/;
my $re = qr/STRING/msixpodualn;
my $re = qr{STRING};
my $re = qr'STRING';            # single-quote delimiter: no␣
↪interpolation
```

Τι επιστρέφεται

Ένα blessed αντικείμενο `Regexp`. Η `ref` επιστρέφει τη συμβολοσειρά `"Regexp"`. Η μετατροπή του αντικειμένου σε συμβολοσειρά παράγει μια κανονικοποιημένη μορφή του μοτίβου με τους τροποποιητές του κωδικοποιημένους σε ένα περιτύλιγμα `(?flags:...)`, που είναι ακριβώς η μορφή που η μηχανή regex ενσωματώνει όταν το αντικείμενο παρεμβάλλεται σε άλλο μοτίβο:

```
my $re = qr/my.STRING/is;
print $re;                      # (?^si:my.STRING)
```

Το αντικείμενο είναι αδιαφανές - μην το αποαναφέρετε. Η μορφή ως συμβολοσειρά προορίζεται για παρεμβολή και αποσφαλμάτωση, όχι για χειροκίνητη ανάλυση.

Καθολική κατάσταση που επηρεάζει

- `$1`, `$2`, ..., `$+`, `$&`, `$``, `$'`, `%+`, `%-` - **δεν** ορίζονται από την ίδια την `qr//`. Η μεταγλώττιση δεν αντιστοιχίζει. Αυτές συμπληρώνονται μόνο όταν το μεταγλωττισμένο μοτίβο χρησιμοποιείται αργότερα από `m`, `s` ή `split`.

- `$@` - ορίζεται αν το μοτίβο περιέχει συντακτικό σφάλμα και το σφάλμα παγιδεύεται με `eval`. Ένα μη παγιδευμένο σφάλμα μεταγλώττισης εγείρει μοιραία εξαίρεση τύπου `Bareword "..." not allowed` στο σημείο της `qr//`.
- Οι pragmas locale και Unicode που ισχύουν στο σημείο της `qr//` (`use locale`, `use utf8`) ψήνονται μέσα στο μεταγλωττισμένο αντικείμενο. Η αλλαγή της pragma αργότερα δεν αλλάζει το αντικείμενο.

Τροποποιητές

Όλοι οι τροποποιητές που ισχύουν για την `m` ισχύουν και για την `qr//`. Συλλαμβάνονται στο αντικείμενο που επιστρέφεται και διαδίδονται όταν το αντικείμενο παρεμβάλλεται σε άλλο μοτίβο.

Σημα	Σημασία
`m`	Πολυγραμμικός: τα `^` και `$` ταιριάζουν σε εσωτερικές νέες γραμμές.
`s`	Μονογραμμικός: το `.` ταιριάζει με χαρακτήρα νέας γραμμής.
`i`	Αντιστοίχιση χωρίς διάκριση πεζών-κεφαλαίων.
`x`	Εκτεταμένος: οι λευκοί χαρακτήρες και τα σχόλια # αγνοούνται στο μοτίβο. Ο `xx` αγνοεί επιπλέον τους λευκούς χαρακτήρες μέσα σε `[...]`.
`p`	Διατήρηση αντιστοιχίας (χωρίς αποτέλεσμα σε Perl 5.20+· τα `${^PREMATCH}`, `${^MATCH}`, `${^POSTMATCH}` είναι πάντοτε διαθέσιμα).
`o`	Μεταγλώττιση μία φορά· οι παρεμβαλλόμενες μεταβλητές παγώνουν στην πρώτη χρήση. Σπάνια χρειάζεται - η `qr//` ήδη αποθηκεύει σε cache.
`a`	Περιορίζει τα `\d`, `\s`, `\w` και τις κλάσεις POSIX σε ASCII. Ο `aa` απαγορεύει επιπρόσθετα αντιστοιχίσεις ASCII-προς-μη-ASCII υπό το `i`.
`l`	Χρήση των κανόνων του τρέχοντος locale.
`u`	Χρήση κανόνων Unicode.
`d`	Προεπιλεγμένοι διπλοί κανόνες (legacy· συνήθως υιοθετούνται αυτόματα).
`n`	Μη σύλληψη: οι ανώνυμες ομάδες `(...)` **δεν** γεμίζουν τα `$1`, `$2`, ...

Όταν ένα μεταγλωττισμένο μοτίβο ενσωματώνεται σε μεγαλύτερο μοτίβο, οι σημαίες χαρακτηροσυνόλου και `msixn` με τις οποίες χτίστηκε ισχύουν μόνο για εκείνη την περιοχή. Ο τροποποιητής `o` είναι η μία εξαίρεση - δεν διαδίδεται.

Παραδείγματα

Μεταγλωττίστε μία φορά, χρησιμοποιήστε πολλές. Επανάληψη σε μοτίβα χτισμένα εκ των προτέρων αποφεύγει την επαναμεταγλώττιση σε κάθε επανάληψη:

```
my @rules = map qr/$_/i, qw(error warn fail panic);
for my $line (@lines) {
    for my $rx (@rules) {
        print $line if $line =~ $rx;
    }
}
```

Παρεμβάλετε ένα μεταγλωττισμένο μοτίβο μέσα σε άλλο μοτίβο. Το εξωτερικό μοτίβο βλέπει τις σημαίες του εσωτερικού μέσω του περιτυλίγματος `(?flags:...)`:

```
my $word = qr/\w+/;
my $csv  = qr/^$word(?:,$word)*$/;
"alpha,beta,gamma" =~ $csv;          # matches
```

Οριοθέτης μονών εισαγωγικών για κυριολεκτικό μοτίβο χωρίς παρεμβολή - χρήσιμο όταν το μοτίβο περιέχει $ ή @ που πρέπει να μείνουν κυριολεκτικά:

```
my $price = qr'\$\d+\.\d{2}';        # matches "$1.99", not a sigil
"total: \$9.95" =~ $price;           # matches
```

Επαναχρησιμοποίηση μοτίβου ανάμεσα σε κλήσεις. Η αποθήκευση της μεταγλωττισμένης μορφής σε ένα closure αποσβένει το κόστος μεταγλώττισης σε κάθε κλήση της επιστρεφόμενης υπορουτίνας:

```
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);
```

Η σύλληψη μέσα σε παρεμβαλλόμενο `qr//` εξακολουθεί να συμπληρώνει το `$1` τη στιγμή της αντιστοίχισης, όχι τη στιγμή της `qr//`:

```
my $num = qr/(\d+)/;
"port 8080" =~ /:$num$/;
print $1;                            # 8080
```

Χρήση του `/n` για καταστολή της σύλληψης στα $1, $2, ... διατηρώντας τις ονομαστικές συλλήψεις διαθέσιμες:

```
my $rx = qr/(foo)(?<kw>bar)/n;
"foobar" =~ $rx;
print $+{kw};                        # "bar"
print defined $1 ? "yes" : "no";     # "no"
```

Οριακές περιπτώσεις

- **Καμία αντιστοίχιση κατά τη μεταγλώττιση.** Η `qr//` μόνο *μεταγλωττίζει*. Για να ελέγξετε μια συμβολοσειρά, χρησιμοποιήστε το αντικείμενο με `m` ή `=~`: `$string =~ $rx`.
- **Οι παρεμβαλλόμενες μεταβλητές αποτυπώνονται τη στιγμή της `qr//`**, όχι τη στιγμή της χρήσης. Το `my $rx = qr/$pat/; $pat = "other";` αφήνει στο `$rx` το μοτίβο που χτίστηκε από την αρχική `$pat`. Για επανασύλληψη, ξαναχτίστε την `qr//`.
- **Τα σφάλματα μοτίβου κατά τη μεταγλώττιση είναι μοιραία.** Το `qr/(/` εγείρει εξαίρεση στο σημείο της `qr//`. Τυλίξτε σε `eval` όταν το μοτίβο προέρχεται από

είσοδο χρήστη και ένα σφάλμα θα πρέπει να είναι ανακτήσιμο:

```
my $rx = eval { qr/$user_input/ };
die "bad pattern: $@" if $@;
```

- **Η `ref($rx)` επιστρέφει `"Regexp"`** - όχι `"SCALAR"`, όχι `"CODE"`. Χρησιμοποιήστε αυτό για να ανιχνεύσετε ένα μεταγλωττισμένο μοτίβο σε πολυμορφική θέση ορίσματος: `ref($arg) eq 'Regexp'`.
- **Η μετατροπή σε συμβολοσειρά είναι απωλεστική για επανανάλυση**. Η κανονικοποιημένη μορφή είναι νόμιμη ως στόχος παρεμβολής αλλά δεν εγγυάται αναγνώσιμο από ανθρώπους αντίγραφο της αρχικής πηγής. Μην εφαρμόζετε `qr//` ξανά στην έξοδο σε μορφή συμβολοσειράς.
- **Ψήσιμο locale και Unicode**. Μια `qr//` που χτίστηκε μέσα σε `use locale` συμπεριφέρεται με κανόνες locale για πάντα, ακόμη και όταν χρησιμοποιείται εκτός εκείνης της εμβέλειας. Ξαναχτίστε υπό την επιθυμητή pragma αν η συμπεριφορά πρέπει να αλλάξει.
- **`\Q...\E` για κυριολεκτικό κείμενο**. Για να μεταγλωττίσετε ένα μοτίβο που ταιριάζει κατά λέξη μια συμβολοσειρά παρεχόμενη από τον χρήστη, τυλίξτε την με `\Q...\E`: `qr/\Q$literal\E/`. Χωρίς `\Q`, οι μετα-χαρακτήρες στη συμβολοσειρά ερμηνεύονται ως σύνταξη regex.
- **Κενό μοτίβο**. Η `qr//` μεταγλωττίζεται σε κενό μοτίβο, το οποίο αντιμετωπίζεται ειδικά τη στιγμή της αντιστοίχισης: το `$str =~ $empty` επαναχρησιμοποιεί το *τελευταίο επιτυχημένο* μοτίβο σε εκείνη την εμβέλεια. Αυτή είναι συμπεριφορά της `m` τη στιγμή της αντιστοίχισης, όχι ιδιαιτερότητα της `qr//`, αλλά αποτελεί συνηθισμένη παγίδα όταν κυκλοφορεί μια προεπιλεγμένη `qr//` ως όρισμα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `m` - αντιστοιχίζει ένα μεταγλωττισμένο μοτίβο ή κυριολεκτικό μοτίβο σε συμβολοσειρά· ο κύριος καταναλωτής ενός αντικειμένου `qr//`
- `s` - αντικαθιστά χρησιμοποιώντας μεταγλωττισμένο μοτίβο· δέχεται μια `qr//` στη θέση αναζήτησης
- `split` - διασπά πάνω σε μεταγλωττισμένο μοτίβο· η μορφή `qr//` είναι η πιο αποδοτική
- *Regular expressions guide*
  - πλήρης αναφορά σύνταξης regex· διαβάστε το για τη σημασία των τροποποιητών και των ακολουθιών διαφυγής
- `ref` - επιστρέφει `"Regexp"` για αντικείμενο `qr//`· ο τυπικός έλεγχος τύπου
- `eval` - τυλίξτε την `qr//` όταν η πηγή του μοτίβου είναι μη έμπιστη και ένα σφάλμα μεταγλώττισης θα πρέπει να είναι ανακτήσιμο

Κανονικές εκφράσεις και αντιστοίχιση μοτίβων

quotemeta

Επιστρέφει αντίγραφο μιας συμβολοσειράς με κάθε χαρακτήρα σημαντικό για regex διαφυγμένο με ανάστροφη πλαγιοκάθετο, ώστε το αποτέλεσμα να μπορεί να παρεμβληθεί σε μοτίβο και να ταιριάξει με το ίδιο του το κυριολεκτικό περιεχόμενο.

Η quotemeta υπάρχει για να γεφυρώσει δεδομένα συμβολοσειράς και σύνταξη regex. Όταν είσοδος χρήστη, ονόματα αρχείων ή τιμές διαμόρφωσης καταλήγουν στη δεξιά πλευρά του =~, οποιοσδήποτε χαρακτήρας τυχαίνει να είναι μεταχαρακτήρας regex (., *, +, ?, (, [, \, |, ...) θα ενεργοποιούσε αλλιώς την ειδική του συμπεριφορά. Η quotemeta το προλαβαίνει εισάγοντας μια ανάστροφη πλαγιοκάθετο πριν από κάθε ASCII μη-λεκτικό χαρακτήρα, αφήνοντας γράμματα, ψηφία και κάτω παύλα άθικτα.

Σύνοψη

```
quotemeta EXPR
quotemeta
```

Τι επιστρέφεται

Μια νέα συμβολοσειρά. Κάθε χαρακτήρας ASCII που **δεν** ταιριάζει στο `/[A-Za-z_0-9]/` προηγείται από ανάστροφη πλαγιοκάθετο· οι λεκτικοί χαρακτήρες περνούν αμετάβλητοι. Το αποτέλεσμα είναι πάντα ασφαλές να επικολληθεί σε regex ως κυριολεκτικό κομμάτι μοτίβου.

Η quotemeta είναι καθαρός παραγωγός τιμής - δεν τροποποιεί το όρισμά της.

```
my $safe = quotemeta 'a.b*c';     # 'a\.b\*c'
```

Καθολική κατάσταση που επηρεάζει

Διαβάζει την $_ όταν καλείται χωρίς όρισμα. Εντός της εμβέλειας του use locale, επιπλέον μη-ASCII χαρακτήρες Latin-1 διαφυγαίνονται για προστασία απέναντι σε locale που αντιμετωπίζουν στίξη όπως το | ως λεκτικό χαρακτήρα.

Παραδείγματα

Η σκέτη μορφή υποχωρεί στην $_, όπως οι περισσότερες ενσωματωμένες συναρτήσεις λίστας:

```
for ('a+b', 'c.d') {
    print quotemeta, "\n";     # a\+b
}                              # c\.d
```

Παρεμβολή εισόδου χρήστη σε αντικατάσταση. Χωρίς την quotemeta, το .*? μέσα στο $substring θα ήταν ενεργό ως regex και θα ταίριαζε πολύ περισσότερα από όσο σκόπευε:

```
my $sentence  = 'The quick brown fox jumped over the lazy dog';
my $substring = 'quick.*?fox';
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
my $quoted    = quotemeta $substring;
$sentence =~ s{$quoted}{big bad wolf};
# $sentence unchanged - the literal text 'quick.*?fox' is not␣
 ↪present
```

Η ακολουθία διαφυγής `\Q...\E` σε διπλά-εισαγωγικά συμβολοσειρά είναι ακριβώς η `quotemeta` εφαρμοσμένη στην περικλειόμενη περιοχή. Οι δύο παρακάτω μορφές είναι ισοδύναμες:

```
my $pat1 = "\Q$substring\E";
my $pat2 = quotemeta($substring);
```

Δόμηση regex που ταιριάζει κυριολεκτικό όνομα αρχείου οπουδήποτε σε διαδρομή:

```
my $name = 'my.config[prod].ini';
if ($path =~ /\Q$name\E\z/) {
    # matches exactly the filename, no metachar surprises
}
```

Διαφυγή λίστας όρων για εναλλαγή:

```
my @terms = ('C++', 'C#', '.NET');
my $alt   = join '|', map { quotemeta } @terms;
# 'C\+\+|C\#|\.NET'
```

Οριακές περιπτώσεις

- **Χωρίς όρισμα**: η `quotemeta` χωρίς έκφραση διαφυγαίνει την `$_`. Μέσα σε βρόχο όπως `for (@input) { push @safe, quotemeta }` αυτή είναι η ιδιωματική μορφή.
- **Οι ήδη ασφαλείς συμβολοσειρές δεν θίγονται**: είσοδος αμιγώς λεκτικών χαρακτήρων (`[A-Za-z0-9_]+`) γυρίζει πλήρη κύκλο πανομοιότυπα.
- **Οι κυριολεκτικές ανάστροφες πλαγιοκάθετοι μέσα σε `\Q...\E`** αλληλεπιδρούν με την παρεμβολή ανάστροφης πλαγιοκαθέτου διπλών-εισαγωγικών πριν προλάβει το `\Q` να δει το κείμενο. Μια ακολουθία όπως `"\Q\t\E"` διαφυγαίνει έναν κυριολεκτικό χαρακτήρα tab, όχι ανάστροφη πλαγιοκάθετο ακολουθούμενη από `t`. Όταν χρειάζεστε πραγματικές ανάστροφες πλαγιοκάθετους στη διαφυγαμένη περιοχή, κρατήστε τα δεδομένα σε μεταβλητή και χρησιμοποιήστε `quotemeta $var` αντί να τα ενσωματώσετε σε κυριολεκτικό διπλών-εισαγωγικών.
- **Δεν υπάρχει τρόπος να εισαχθεί κυριολεκτικό `$` ή `@` μέσα σε `\Q...\E`**: ένα προστατευμένο `\$` γίνεται η τεσσάρων χαρακτήρων ακολουθία `\\\$` στην έξοδο, και ένα μη προστατευμένο `$` ξεκινά παρεμβολή βαθμωτού πριν εκτελεστεί το `\Q`. Βγείτε από τη διαφυγαμένη περιοχή για να εισαγάγετε sigils.
- **Διαφυγή ενήμερη Unicode** (Perl 5.16+): σε συμβολοσειρές UTF-8 - και σε συμβολοσειρές bytes υπό `use feature 'unicode_strings'` ή `use v5.12` ή νεότερο - οι μη-ASCII χαρακτήρες διαφυγαίνονται μόνο όταν φέρουν τις ιδιότητες Unicode `Pattern_Syntax`, `Pattern_White_Space`, `White_Space`, `Default_Ignorable_Code_Point` ή `General_Category=Control`. Οι

χαρακτήρες κλάσης αναγνωριστικού (γράμματα, σημάδια, ψηφία) μένουν ανέγγιχτοι. Αυτό είναι το σταθερό συμβόλαιο: η Perl υπόσχεται ότι κάθε μελλοντικός μεταχαρακτήρας regex θα έχει ορισμένη την `Pattern_Syntax`, οπότε οι συμβολοσειρές που είναι ασφαλείς σήμερα παραμένουν ασφαλείς.

- **Παλαιές μη-UTF-8 συμβολοσειρές** εκτός εμβέλειας `unicode_strings` έχουν κάθε άνω-Latin-1 κωδικοσημείο (`\x80`–`\xFF`) διαφυγαμένο, για συμβατότητα προς τα πίσω με τη συμπεριφορά πριν την 5.16.
- **Διαφυγή locale**: εντός `use locale`, όλα τα μη-ASCII κωδικοσημεία Latin-1 διαφυγαίνονται είτε η συμβολοσειρά είναι UTF-8 είτε όχι. Η διαφυγή εύρους ASCII δεν επηρεάζεται από locale.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `m` - ο τελεστής αντιστοίχισης· η `quotemeta` υπάρχει για να τροφοδοτεί ασφαλή κυριολεκτικά στο μοτίβο του
- `qr` - μεταγλωττισμένα αντικείμενα regex· η `qr/\Q$var\E/` είναι ο συνήθης τρόπος καρφώματος ενός μοτίβου γύρω από μια κυριολεκτική συμβολοσειρά
- `s` - αντικατάσταση· το συνηθέστερο σημείο διαφυγής τύπου `quotemeta` στο αριστερό μοτίβο
- `\Q` *and* `\E` - το ζεύγος διαφυγής διπλών-εισαγωγικών που είναι απλώς η `quotemeta` με διαφορετική σύνταξη
- `$_` - προεπιλεγμένη είσοδος όταν η `quotemeta` καλείται χωρίς όρισμα

Λίστες

qw//

Δομεί λίστα από barewords κάνοντας split μιας οριοθετημένης συμβολοσειράς σε λευκούς χαρακτήρες, χωρίς εισαγωγικά ή διαχωρισμό με κόμμα σε κάθε λέξη.

Η `qw//` είναι τελεστής τύπου-εισαγωγικών μεταγλώττισης, όχι συνάρτηση χρόνου εκτέλεσης. Ο αναλυτής βλέπει `qw(foo bar baz)` και εκπέμπει τη σταθερή λίστα `("foo", "bar", "baz")` - δεν συμβαίνει split κατά τον χρόνο εκτέλεσης, και η λίστα συγκροτείται μία φορά ανεξάρτητα από το πόσο συχνά εκτελείται ο κώδικας. Καταφύγετε σε αυτήν όπου διαφορετικά θα γράφατε λίστα από σύντομες, απλές κυριολεκτικές τιμές συμβολοσειρών χωρισμένες με κόμμα: λίστες εισαγωγής, `@ISA`, `@EXPORT`, σύνολα κλειδιών hash, πίνακες ονομάτων δοκιμών.

Σύνοψη

```
qw(foo bar baz)
qw[foo bar baz]
qw{foo bar baz}
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
qw<foo bar baz>
qw!foo bar baz!
qw/foo bar baz/
```

Κάθε μη αλφαριθμητικός χαρακτήρας λειτουργεί ως οριοθέτης. Τα τέσσερα ζεύγη αγκυλών (`()`, `[]`, `{}`, `<>`) εμφωλεύονται· όλοι οι άλλοι οριοθέτες δεν εμφωλεύονται.

Τι επιστρέφεται

Λίστα συμβολοσειρών σε περιβάλλον λίστας - ένα στοιχείο ανά συνεχόμενη ακολουθία μη λευκών χαρακτήρων στο σώμα, με τα αρχικά και τελικά λευκά να απορρίπτονται και συνεχόμενες ακολουθίες εσωτερικών λευκών να αντιμετωπίζονται ως ένας διαχωριστής.

Σε βαθμωτό περιβάλλον η `qw//` αποτιμάται στο **τελευταίο** στοιχείο της λίστας, που σχεδόν ποτέ δεν είναι αυτό που εννοούσε ο συγγραφέας. Το `use warnings` επισημαίνει τα συνηθισμένα λάθη βαθμωτού περιβάλλοντος (δείτε *Οριακές περιπτώσεις*).

Παραδείγματα

Λίστα εισαγωγής για άρθρωμα - η κανονική χρήση:

```
use POSIX qw(setlocale localeconv strftime);
```

Πίνακες `@EXPORT` / `@EXPORT_OK` / `@ISA`:

```
our @ISA     = qw(Exporter);
our @EXPORT  = qw(sum min max);
```

Δομήστε ένα μικρό hash όπου κάθε κλειδί αντιστοιχίζεται στο 1 (ένα σύνολο):

```
my %is_keyword = map { $_ => 1 } qw(if unless while until for);
if ($is_keyword{$word}) { ... }
```

Σύγκριση των τριών τρόπων γραφής της ίδιας λίστας - με κόμματα και εισαγωγικά, αυτόματα εισαγωγικά με fat-comma, και `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//
```

Αρχικοποίηση πίνακα με πολλά σύντομα διακριτικά:

```
my @months = qw(
    Jan Feb Mar Apr May Jun
    Jul Aug Sep Oct Nov Dec
);
```

Λίστα ορισμάτων κατά θέση προς συνάρτηση:

```
run_tests(qw(smoke integration regression));
```

Οριακές περιπτώσεις

- **Κανένα κόμμα μέσα στο σώμα.** Το κόμμα λαμβάνεται ως μέρος λέξης: η `qw(foo, bar)` αποδίδει διστοιχειακή λίστα `("foo,", "bar")`, όχι τριστοιχειακή. Το `use warnings` εκπέμπει `Possible attempt to separate words with commas`.
- **Καμία σύνταξη σχολίων.** Το `#` είναι κυριολεκτικός χαρακτήρας μέσα στο σώμα. Η `qw(foo # comment\n bar)` περιέχει τη λέξη `#` και τη λέξη `comment`. Το `use warnings` εκπέμπει `Possible attempt to put comments in qw() list`.
- **Καμία διαφυγή με ανάστροφη κάθετο.** Το `\` είναι κυριολεκτικό - δεν μπορείτε να διαφύγετε λευκό χαρακτήρα για να τον συμπεριλάβετε σε λέξη, και δεν μπορείτε να κάνετε `\`-διαφυγή του οριοθέτη. Το `use warnings` εκπέμπει προειδοποίηση για κάθε `\` στο σώμα.
- **Καμία παρεμβολή.** Η `qw($x @y)` παράγει τις δύο κυριολεκτικές συμβολοσειρές `"$x"` και `"@y"`. Χρησιμοποιήστε `("$x", @y)` (ή `q()`/`qq()`) όταν θέλετε την τιμή μιας μεταβλητής.
- **Ο λευκός χαρακτήρας είναι ο μόνος διαχωριστής.** Tabs και newlines χωρίζουν τις λέξεις ακριβώς όπως τα κενά, γι" αυτό και οι πολυγραμμικές λίστες `qw(...)` είναι ιδιωματικές για μακριούς πίνακες.
- **Λέξεις με ενσωματωμένο λευκό χαρακτήρα είναι αδύνατες.** Αν χρειάζεστε το `"hello world"` ως μονό στοιχείο, η `qw//` είναι λάθος εργαλείο - χρησιμοποιήστε `("hello world", ...)` με αληθινά εισαγωγικά.
- **Κενό σώμα.** Η `qw()` είναι η κενή λίστα, ισοδύναμη με `()`.
- **Το βαθμωτό περιβάλλον** αποδίδει το τελευταίο στοιχείο, όχι τον αριθμό. Τυλίξτε σε `scalar(() = qw(...))` αν θέλετε γνήσια τον αριθμό, ή αναθέστε πρώτα σε πίνακα:

```
my @w = qw(foo bar baz);
print scalar @w;                # 3
```

- **Αμφισημία του αναλυτή με `//`.** Η `qw//` χρησιμοποιεί το `/` ως οριοθέτη ακριβώς όπως οι `m//` και `s///`. Μέσα στο σώμα, δεν εφαρμόζονται μετα-χαρακτήρες regex - είναι επίπεδη συμβολοσειρά που διαιρείται με λευκούς χαρακτήρες. Η οπτική ομοιότητα με regex είναι κίνδυνος ανάγνωσης, όχι σημασιολογικός.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `q//` - κυριολεκτική συμβολοσειρά με μονά εισαγωγικά· χωρίς παρεμβολή, σε αντίθεση με την `qq//`
- `qq//` - κυριολεκτική συμβολοσειρά με διπλά εισαγωγικά με πλήρη παρεμβολή
- `qr//` - μεταγλωττίζει regex literal· το μέλος της οικογένειας `//` για αντιστοίχιση μοτίβων

- `split` - το αντίστοιχο χρόνου εκτέλεσης όταν η λίστα λέξεών σας δεν είναι γνωστή πριν τον χρόνο εκτέλεσης: η `split(" ", $str)` συμπεριφέρεται σχεδόν ακριβώς όπως μια δυναμική `qw//`
- `join` - η αντίστροφη λειτουργία, που κολλάει μια λίστα πίσω σε μία οριοθετημένη συμβολοσειρά

Διεργασίες

qx//

Εκτελεί μια εντολή κελύφους και αποτυπώνει την κανονική έξοδό της.

Η `qx//` (και το δίδυμό της, η μορφή με backticks `` `STRING` ``) παρεμβάλλει τη `STRING` όπως μια συμβολοσειρά με διπλά εισαγωγικά, παραδίδει το αποτέλεσμα στο κέλυφος του συστήματος, περιμένει να ολοκληρωθεί το παιδί και επιστρέφει ό,τι έγραψε το παιδί στο `STDOUT`. Η κανονική έξοδος σφάλματος παραμένει ανέπαφη - εξακολουθεί να πηγαίνει στο `STDERR` του γονέα εκτός αν την ανακατευθύνετε μέσα στην εντολή. Η κατάσταση εξόδου του παιδιού προσγειώνεται στην `$?`.

Σύνοψη

```
my $out   = `date`;
my $out   = qx/date/;
my @lines = `ls -1 /etc`;
my $raw   = qx'echo $HOME';     # single-quote form: no Perl
↪interpolation
```

Τι επιστρέφεται

Εξαρτάται από το περιβάλλον:

- **Βαθμωτό περιβάλλον** - μία συμβολοσειρά που περιέχει ολόκληρο το αποτυπωμένο `STDOUT`, συμπεριλαμβανομένων όσων ενσωματωμένων newlines υπάρχουν. `undef` αν δεν μπόρεσε να ξεκινήσει το κέλυφος ή η εντολή.
- **Περιβάλλον λίστας** - λίστα γραμμών, διασπασμένη στην τρέχουσα τιμή της `$/` (του διαχωριστή εγγραφών εισόδου, newline εξ ορισμού). Κενή λίστα αν δεν μπόρεσε να ξεκινήσει το κέλυφος ή η εντολή.

Η κατάσταση εξόδου του παιδιού τοποθετείται πάντα στην `$?`, είτε χρησιμοποιείται η αποτυπωμένη έξοδος είτε όχι. Ένα παιδί που δεν γράφει τίποτα και τερματίζει καθαρά αποδίδει `""` σε βαθμωτό περιβάλλον, `()` σε περιβάλλον λίστας και `0` στην `$?` - η διάκριση «ξεκίνησε αλλά δεν παρήγαγε έξοδο» από «απέτυχε να ξεκινήσει» απαιτεί έλεγχο της `$?` έναντι του `-1` ή της `$!`.

Καθολική κατάσταση που επηρεάζει

- `$?` - τίθεται στην κατάσταση αναμονής του παιδιού (στιλ `waitpid`) σε κάθε κλήση. Το `-1` σημαίνει ότι το παιδί δεν μπόρεσε να ξεκινήσει, οπότε η `$!` κρατά τον λόγο.
- `$!` - τίθεται όταν το κέλυφος ή η εντολή απέτυχαν να ξεκινήσουν.
- `$/` - διέπει το πώς διασπάται σε γραμμές η έξοδος σε περιβάλλον λίστας.

- `$|` - επηρεάζει αν η ενταμιευμένη έξοδος της Perl φτάνει στο περιβάλλον του παιδιού πριν τρέξει το παιδί· ορίστε την σε handles εξόδου πριν από `qx//` αν έχει σημασία η σειρά με προηγούμενες `print`.
- `%ENV` - εξάγεται στο παιδί ως το περιβάλλον του.
- Ο τρέχων κατάλογος εργασίας, το umask και οι ανοιχτοί περιγραφείς αρχείων κληρονομούνται από το παιδί.

Παραδείγματα

Αποτύπωση μίας γραμμής:

```
my $today = `date`;
chomp $today;
print "today is $today\n";
```

Αποτύπωση γραμμή-γραμμή σε περιβάλλον λίστας:

```
my @users = `who`;
for my $line (@users) {
    my ($name) = split /\s+/, $line;
    print "logged in: $name\n";
}
```

Η παρεμβολή συμβαίνει προτού το κέλυφος δει τη συμβολοσειρά - οι μεταβλητές Perl επεκτείνονται πρώτα, οι μεταβλητές κελύφους επεκτείνονται στο παιδί:

```
my $dir = "/etc";
my $out = `ls $dir`;            # Perl interpolates $dir
my $sh  = qx'echo $HOME';       # single-quote form: $HOME is shell
 ↪'s
```

Ανακατεύθυνση του STDERR στην αποτυπωμένη ροή:

```
my $both = `gcc -v hello.c 2>&1`;
```

Απόρριψη του STDERR, διατήρηση μόνο του STDOUT:

```
my $out = `find / -name core 2>/dev/null`;
```

Έλεγχος της κατάστασης εξόδου:

```
my $out = `make test`;
if ($? == -1) {
    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;
}
```

Οριακές περιπτώσεις

- **Κανείς μετα-χαρακτήρας κελύφους**: αν η `STRING` δεν περιέχει μετα-χαρακτήρες κελύφους, η Perl παρακάμπτει το κέλυφος και κάνει `exec` την εντολή απευθείας - ίδια βελτιστοποίηση με την `system`. Αυτό είναι ταχύτερο και αποφεύγει ένα επίπεδο `sh -c` παραθέσεων.
- **Οριοθέτης μονού εισαγωγικού**: η `qx'...'` καταστέλλει την παρεμβολή διπλών εισαγωγικών της Perl. Το κυριολεκτικό `$var` περνά στο κέλυφος, το οποίο κάνει τη δική του επέκταση. Χρήσιμο όταν τα ονόματα μεταβλητών ανήκουν στο κέλυφος, όχι στο πρόγραμμά σας Perl.
- **Εναλλακτικοί οριοθέτες**: η `qx//` δέχεται οποιονδήποτε ισορροπημένο ή μη-αλφαριθμητικό οριοθέτη - `qx{ls -l}`, `qx(ls -l)`, `qx[ls -l]`, `qx!ls -l!`. Επιλέξτε έναν που δεν συγκρούεται με τη σύνταξη κελύφους στην εντολή σας. Η `qXfooX` είναι συντακτικό σφάλμα - ο οριοθέτης δεν επιτρέπεται να είναι αλφαριθμητικός.
- **Τα backticks δεν αποτυπώνουν το STDERR**: τίποτα γραμμένο στο `STDERR` του παιδιού δεν καταλήγει στην επιστρεφόμενη συμβολοσειρά. Ανακατευθύνετε με `2>&1` μέσα στην εντολή αν το θέλετε, ή τρέξτε την εντολή μέσω της `open` με σωλήνα για πιο λεπτό έλεγχο.
- **Το STDIN κληρονομείται**: το παιδί διαβάζει από ό,τι είχε η διεργασία Perl στο `STDIN` τη στιγμή της κλήσης. Για να τροφοδοτήσετε στο παιδί συγκεκριμένη είσοδο, χρησιμοποιήστε αντί αυτής την `open` σε κατάσταση σωλήνα.
- **Κατάσταση εξόδου έναντι `$!`**: μη μηδενική `$?` σημαίνει ότι το παιδί έτρεξε και τερμάτισε με σφάλμα· η `$!` έχει νόημα μόνο όταν η `$?` είναι `-1` (δεν μπόρεσε να ξεκινήσει το ίδιο το κέλυφος).
- **Κόστος μνήμης σε βαθμωτό περιβάλλον**: ολόκληρη η αποτυπωμένη έξοδος εντάμιεύεται στη μνήμη πριν επιστραφεί. Για εντολές που παράγουν πολύ μεγάλη ή χωρίς όρια έξοδο, ανοίξτε σωλήνα με `open` και διαβάστε τη σταδιακά.
- **Μορφή συνάρτησης**: η `qx//` μπορεί επίσης να γραφτεί ως `readpipe` όταν θέλετε σύνταξη κλήσης συνάρτησης - χρήσιμο όταν η συμβολοσειρά εντολής χτίζεται δυναμικά και προτιμάτε να την περάσετε ως βαθμωτό αντί να τη συναρμολογήσετε σε quote-like.
- **Ενταμίευση εξόδου**: η Perl εκκενώνει τα ενταμιευμένα handles εξόδου της πριν κάνει fork το παιδί, αλλά η φορητότητα ποικίλλει. Όταν προηγείται `print` σε σωλήνα ή αρχείο πριν από `qx//`, ορίστε την `$|` σε εκείνο το handle ώστε να είστε βέβαιοι ότι η έξοδός σας φτάνει στον προορισμό πριν ξεκινήσει το παιδί.
- **Επιλογή κελύφους**: το κέλυφος που χρησιμοποιείται είναι το `/bin/sh` σε Unix. Εντολές που στηρίζονται σε ιδιαιτερότητες του bash χρειάζονται ρητό περιτύλιγμα `bash -c '...'`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `system` - εκτελεί εντολή κελύφους χωρίς να αποτυπώνει την έξοδο· καταφύγετε σε αυτήν όταν σας ενδιαφέρει μόνο η κατάσταση εξόδου
- `exec` - αντικαθιστά την τρέχουσα διεργασία με μια εντολή· καμία αποτύπωση, καμία επιστροφή
- `readpipe` - η ονομασμένη μορφή συνάρτησης της `qx//`· δέχεται την εντολή ως κανονικό όρισμα συμβολοσειράς
- `open` - κατάσταση σωλήνα (`open my $fh, "-|", CMD`) για ροή της εξόδου ενός παιδιού γραμμή-γραμμή αντί για ενταμίευση ολόκληρης της αποτύπωσης
- `$?` - η κατάσταση αναμονής του παιδιού που ορίζει κάθε κλήση `qx//`· αποκωδικοποιήστε με `>> 8` για κωδικό εξόδου και `& 127` για τερματιστικό σήμα
- `$/` - ελέγχει πώς η `qx//` σε περιβάλλον λίστας διασπά την έξοδο σε γραμμές

Αριθμητικές συναρτήσεις

rand

Επιστρέφει έναν ψευδοτυχαίο αριθμό κινητής υποδιαστολής στο ημιανοικτό εύρος `[0, EXPR)`.

Η `rand` αντλεί την επόμενη τιμή από τη ροή ψευδοτυχαίων αριθμών του διερμηνέα και την κλιμακώνει σε `0 <= n < EXPR`. Το αποτέλεσμα είναι κανονικός αριθμός Perl (double), ποτέ ακέραιος από μόνος του - περιβάλετε με `int` όταν θέλετε ακέραιο αριθμό. Το `EXPR` είναι προαιρετικό και έχει προεπιλογή `1`, οπότε η σκέτη `rand` επιστρέφει float στο `[0, 1)`.

Σύνοψη

```
rand EXPR
rand
```

Τι επιστρέφεται

Ένα double-precision float `n` με `0 <= n < EXPR`. Το άνω όριο είναι **αποκλειστικό**: η `rand(6)` ποτέ δεν επιστρέφει `6`, μόνο τιμές αυστηρά κάτω από αυτό. Για ακέραιο αποτέλεσμα, συνδυάστε με `int`:

```
my $die = int(rand(6)) + 1;       # 1..6
```

Η κατανομή είναι (κατά προσέγγιση) ομοιόμορφη σε όλο το εύρος. Η `rand` είναι βαθμωτός τελεστής - επιστρέφει μία τιμή ανά κλήση, ανεξάρτητα από το περιβάλλον.

Καθολική κατάσταση που επηρεάζει

Η `rand` καταναλώνει την εσωτερική κατάσταση PRNG του διερμηνέα, την οποία μοιράζεται με την `srand`. Δύο κανόνες απορρέουν από αυτό:

- **Αυτόματη σπορά**: η πρώτη κλήση της `rand` σε ένα πρόγραμμα καλεί αυτόματα την `srand` αν δεν την έχετε καλέσει ήδη. Ο σπόρος επιλέγεται από εντροπία επιπέδου διεργασίας, οπότε προγράμματα χωρίς σπορά παράγουν διαφορετική ροή σε κάθε εκτέλεση.
- **Κοινόχρηστη ροή**: κάθε επόμενη κλήση `rand` - οπουδήποτε στο πρόγραμμα, σε οποιοδήποτε άρθρωμα - αντλεί από την ίδια ακολουθία. Η κλήση της `srand($seed)` με σταθερό σπόρο καθιστά αυτή την ακολουθία ντετερμινιστική και αναπαραγώγιμη, κάτι που είναι αυτό που θέλετε για test fixtures και debugging. Η κλήση της `srand()` χωρίς όρισμα κάνει εκ νέου σπορά από εντροπία.

Δεν υπάρχει απομόνωση ανά νήμα ή ανά εμβέλεια. Αν χρειάζεστε ανεξάρτητη ροή για ένα κομμάτι κώδικα, μη χρησιμοποιήσετε την `rand`· καταφύγετε σε ένα ειδικό άρθρωμα γεννήτριας.

Παραδείγματα

Βασική κλήρωση στο `[0, 1)`:

```
my $u = rand();                    # e.g. 0.4173...
```

Ρίψη ζαριού - το κανονικό ιδίωμα `int(rand(N)) + 1`:

```
my $roll = int(rand(6)) + 1;       # 1..6
```

Τυχαίο στοιχείο πίνακα:

```
my @colors = qw(red green blue yellow);
my $pick   = $colors[ rand @colors ];
```

Η `rand @colors` χρησιμοποιεί το μήκος του πίνακα (βαθμωτό περιβάλλον επιβαλλόμενο από το πρωτότυπο της `rand`), οπότε αυτό λειτουργεί για κάθε μέγεθος πίνακα.

Αναπαραγώγιμη ροή - καθορίστε τον σπόρο για τα τεστ:

```
srand(42);
my @sample = map { rand() } 1..5; # same five numbers every run
```

Μονόγραμμο ιδίωμα ανακατέματος. Ευανάγνωστο, αλλά με **μεροληψία** - η `sort` συγκρίνει ζεύγη παραπάνω από μία φορά, και μια μονομερής `rand()` ανά σύγκριση δεν παράγει ομοιόμορφη μετάθεση:

```
my @shuffled = sort { rand() <=> 0.5 } @list;   # biased; do not use
```

Για δίκαιο ανακάτεμα, χρησιμοποιήστε την `List::Util::shuffle`:

```
use List::Util qw(shuffle);
my @shuffled = shuffle @list;
```

Προσομοίωση ζυγισμένου νομίσματος - σημειώστε ότι το αποκλειστικό άνω όριο σημαίνει ότι το `rand() < 0.3` πυροδοτείται με πιθανότητα ακριβώς `0.3`:

```
my $heads = rand() < 0.3 ? "heads" : "tails";
```

Οριακές περιπτώσεις

- **Το EXPR είναι `0`**: αντιμετωπίζεται ως `1`, οπότε η `rand(0)` συμπεριφέρεται όπως η `rand(1)`. Αυτή είναι τεκμηριωμένη συμπεριφορά του upstream και διατηρείται εδώ, αλλά είναι ιστορική ιδιορρυθμία - μη βασιστείτε σε αυτήν σε νέο κώδικα.
- **Το EXPR είναι αρνητικό**: το αποτέλεσμα είναι απροσδιόριστο στην πράξη - μπορεί να λάβετε `0`, μπορεί να λάβετε αρνητικό float, και η συμπεριφορά δεν είναι φορητή μεταξύ εκδόσεων της Perl. Αν θέλετε τιμή στο `[-N, 0)`, γράψτε ρητά `rand($n) - $n`.
- **Το EXPR είναι undef**: μετατρέπεται σε `""`, αριθμητικοποιείται σε `0`, και επομένως συμπεριφέρεται όπως `rand(0)` → `rand(1)`. Υπό `use warnings` λαμβάνετε προειδοποίηση `uninitialized`.
- **Το EXPR είναι μη αριθμητικό**: ισχύει ο συνηθισμένος εξαναγκασμός βαθμωτού-προς-αριθμό. Η `rand("3abc")` είναι `rand(3)` με προειδοποίηση μη αριθμητικού.
- **Δεν είναι κρυπτογραφικά ασφαλής**: η `rand` χρησιμοποιεί γρήγορο PRNG κατάλληλο για προσομοίωση, ανακάτεμα, και test fixtures. **Δεν** είναι κατάλληλη για tokens συνεδρίας, salts κωδικών, υλικό κλειδιού, ή οποιαδήποτε ευαίσθητη ως προς την ασφάλεια τιμή. Για αυτά, διαβάστε απευθείας από `/dev/urandom` ή χρησιμοποιήστε `Crypt::URandom` / `Crypt::PRNG`.
- **Συμπεριφορά threads/fork**: μετά από `fork`, γονέας και παιδί κληρονομούν την ίδια κατάσταση PRNG και θα παράγουν την **ίδια** ροή εκτός αν κάποιος από τους δύο καλέσει την `srand()` για εκ νέου σπορά. Συνηθισμένη παγίδα σε pre-forking διακομιστές - κάντε σπορά στο παιδί.
- **Μεγάλο EXPR**: η γεννήτρια παράγει 48 bits τυχαιότητας ανά κλήρωση· για EXPR κοντά ή πάνω από `2**48`, η έξοδος δεν κατανέμεται πλέον λεπτομερώς σε όλο το εύρος. Για κρυπτογραφικά πλάτη ούτως ή άλλως δεν θα έπρεπε να χρησιμοποιείτε την `rand` (δείτε παραπάνω).

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `srand` - σπορά της PRNG ρητά για αναπαραγωγιμότητα, ή εκ νέου σπορά σε forked παιδί
- `int` - αποκόπτει ένα αποτέλεσμα της `rand` προς το μηδέν για να δώσει ομοιόμορφο ακέραιο στο `[0, N)`
- `List::Util::shuffle` - δίκαιο ανακάτεμα Fisher-Yates· προτιμήστε αυτό αντί του ιδιώματος `sort { rand() <=> 0.5 }`

- `Crypt::URandom` - κρυπτογραφικά ασφαλή τυχαία bytes από την πηγή εντροπίας του OS· χρησιμοποιήστε το για tokens, salts και υλικό κλειδιού
- `Crypt::PRNG` - κρυπτογραφικά ασφαλής PRNG με πλούσιο API όταν χρειάζεστε περισσότερα από ακατέργαστα bytes

I/O · Δεδομένα σταθερού μήκους

read

Διαβάζει σταθερή ποσότητα ενταμιευμένης εισόδου από ένα filehandle σε ένα βαθμωτό.

Η `read` τραβά έως `LENGTH` χαρακτήρες από το `FILEHANDLE` και τους αποθηκεύει στο `SCALAR`, επιστρέφοντας πόσοι διαβάστηκαν στην πραγματικότητα. Περνά μέσα από τη στοίβα PerlIO του handle και είναι επομένως ενταμιευμένη επάνω από την υποκείμενη ανάγνωση ΛΣ - σε αντίθεση με την `sysread`, η οποία παρακάμπτει τον ενταμιευτή και καλεί απευθείας την `read(2)`. Το προαιρετικό όρισμα `OFFSET` σας επιτρέπει να μπολιάσετε τα εισερχόμενα δεδομένα στο μέσο του `SCALAR` αντί να το αντικαταστήσετε.

Σύνοψη

```
read FILEHANDLE, SCALAR, LENGTH
read FILEHANDLE, SCALAR, LENGTH, OFFSET
```

Τι επιστρέφεται

- Ο **αριθμός χαρακτήρων που διαβάστηκαν**, ο οποίος μπορεί να είναι μικρότερος από `LENGTH`.
- `0` στο τέλος του αρχείου.
- `undef` σε σφάλμα, με ορισμένο το `$!`.

Το `SCALAR` μεγαλώνει ή μικραίνει ώστε ο τελευταίος χαρακτήρας που διαβάστηκε πραγματικά να γίνει ο τελευταίος χαρακτήρας του βαθμωτού - εκτός αν δοθεί `OFFSET`, οπότε αντικαθίσταται μόνο το τμήμα στη θέση `OFFSET` και ό,τι βρίσκεται πέρα από αυτό αφήνεται ως έχει (δείτε *Το όρισμα OFFSET* παρακάτω).

Μια σύντομη ανάγνωση **δεν** αποτελεί σφάλμα. Σε κανονικό αρχείο σημαίνει συνήθως ότι φτάσατε στο τέλος του αρχείου· σε σωλήνα, υποδοχή ή τερματικό σημαίνει ότι δεν υπάρχουν άλλα δεδομένα διαθέσιμα αυτή τη στιγμή. Επαναλάβετε σε βρόχο μέχρι είτε να έχετε τα bytes που χρειάζεστε είτε η `read` να επιστρέψει `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;
}
```

Καθολική κατάσταση που επηρεάζει

- `$!` - ορίζεται όταν η `read` επιστρέφει `undef`.
- `${^UTF8CACHE}` / τα στρώματα PerlIO του handle - καθορίζουν αν το `LENGTH` μετράται σε bytes ή σε χαρακτήρες (δείτε *Σημασιολογία χαρακτήρων έναντι bytes* παρακάτω).

Η `read` δεν αλληλεπιδρά με τα `$_`, `$/`, `$\`, ή `$,`. Σε αντίθεση με την `readline`, δεν ενδιαφέρεται για τον διαχωριστή εγγραφών εισόδου.

Το όρισμα OFFSET

Το `OFFSET` ελέγχει **πού μέσα στο SCALAR** προσγειώνονται τα εισερχόμενα δεδομένα. Δεν κάνει αναζήτηση στο filehandle.

- **Παραλειπόμενο** - τα δεδομένα αντικαθιστούν ολόκληρο το περιεχόμενο του `SCALAR`.
- **Θετικό, εντός μήκους** - τα δεδομένα γράφονται ξεκινώντας από τη θέση `OFFSET`. Οι χαρακτήρες πριν το `OFFSET` διατηρούνται· οι χαρακτήρες από το `OFFSET` μέχρι το τέλος του `SCALAR` αντικαθίστανται ή επεκτείνονται.
- **Θετικό, πέρα από το μήκος** - το `SCALAR` γεμίζεται πρώτα με bytes `"\0"` μέχρι το `OFFSET`, και έπειτα προσαρτάται η ανάγνωση. Χρήσιμο για ανάγνωση σε σταθερή θέση μέσα σε μεγαλύτερο ενταμιευτή που συναρμολογείτε.
- **Αρνητικό** - μετρά αντίστροφα από το τέλος του `SCALAR`. Το `-1` σημαίνει «αντικατάσταση του τελευταίου χαρακτήρα και προσάρτηση από εκεί».

```
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
```

Σημασιολογία χαρακτήρων έναντι bytes

Το `LENGTH` μετράται σε όποια μονάδα διαχειρίζεται το handle:

- **Handle σε λειτουργία bytes** (η προεπιλογή, και κάθε handle που ανοίγεται χωρίς στρώμα κωδικοποίησης): το `LENGTH` είναι αριθμός bytes. Το `read($fh, $buf, 10)` τραβά 10 bytes και το `length $buf` είναι 10.
- **Στρώμα `:utf8`**: το `LENGTH` είναι αριθμός **χαρακτήρων**. Η Perl αποκωδικοποιεί UTF-8 κατά την είσοδο, και το `$buf` κρατά αποκωδικοποιημένα code points. Ο αριθμός bytes που καταναλώνονται από το αρχείο μπορεί να κυμαίνεται από `LENGTH` έως `4 * LENGTH`, ανάλογα με το κείμενο.
- **Στρώμα `:encoding(...)`**: ίδιος κανόνας με το `:utf8`, για οποιαδήποτε κωδικοποίηση γνωρίζει το στρώμα.

```
open my $fh, "<:utf8", "greek.txt" or die $!;
read($fh, my $buf, 5);                  # 5 characters, not 5 bytes
```

Η ανάμειξη ανάγνωσης σε λειτουργία bytes με δεδομένα UTF-8 παράγει mojibake και, υπό το `use warnings`, μια προειδοποίηση `Malformed UTF-8` αν αποκωδικοποιήσετε αργότερα το αποτέλεσμα. Επιλέξτε το στρώμα κατά τη στιγμή του `open` και μείνετε σε αυτό.

I/O με ενταμιευτή έναντι χωρίς ενταμιευτή

Η `read` είναι **stdio-ενταμιευμένη** μέσω PerlIO - εσωτερικά καλεί `fread(3)` (ή την αντικατάστασή της από το PerlIO) στον ενταμιευτή του handle. Αυτό έχει δύο συνέπειες που αξίζει να θυμάστε:

- Μπορείτε να αναμείξετε ελεύθερα τις `read`, `readline` / `<$fh>`, `getc` και `seek` στο ίδιο handle. Όλες βλέπουν τον ίδιο ενταμιευτή.
- **Δεν** πρέπει να αναμείξετε τη `read` με τη `sysread` στο ίδιο handle. Η `sysread` παρακάμπτει τον ενταμιευτή και πάει κατευθείαν στην `read(2)`· οποιαδήποτε bytes έχουν ήδη τραβηχτεί στον ενταμιευτή από προηγούμενη `read` γίνονται αόρατα στην `sysread`, και αντιστρόφως. Αν χρειάζεστε σημασιολογία πρωτογενούς κλήσης συστήματος, χρησιμοποιήστε αποκλειστικά `sysread` σε αυτό το handle.

Για είσοδο ακριβείας bytes, χωρίς ενταμιευτή - για παράδειγμα σε μη μπλοκάρουσα υποδοχή, ή όταν υλοποιείτε πρωτόκολλο όπου μια σύντομη ανάγνωση έχει νόημα και δεν είναι «δοκίμασε ξανά» - καταφύγετε στη `sysread`.

Παραδείγματα

Ανάγνωση κεφαλίδας σταθερού μεγέθους από δυαδικό αρχείο:

```
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;
```

Προσάρτηση 16 bytes στο τέλος υπάρχοντος ενταμιευτή χρησιμοποιώντας `OFFSET` ίσο με το τρέχον μήκος:

```
my $buf = "PRELUDE:";
read($fh, $buf, 16, length $buf);       # $buf is now "PRELUDE:" . 16␣
↪new bytes
```

Ανάγνωση στη θέση 1024 ενός βαθμωτού, γεμίζοντας το κενό με `"\0"`:

```
my $slot = "";
read($fh, $slot, 64, 1024);             # length($slot) == 1088
                                        # substr($slot, 0, 1024) is
↪"\0" x 1024
```

Επαναλάβετε σε βρόχο μέχρι να έχετε ακριβώς N bytes ή να φτάσετε σε EOF - το σωστό μοτίβο για σωλήνες και υποδοχές όπου μια μεμονωμένη `read` συχνά επιστρέφει λιγότερα bytes από όσα ζητούνται:

```
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;
}
```

Ανάγνωση μετρημένη σε χαρακτήρες μέσω στρώματος UTF-8:

```
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;
};
```

Οριακές περιπτώσεις

- **Κλειστό filehandle**: επιστρέφει `undef` και ορίζει το `$!` σε `"Bad file descriptor"`. Υπό το `use warnings` εκπέμπεται προειδοποίηση `read() on closed filehandle`.
- **Μη ανοιγμένο filehandle**: ίδιο με κλειστό - `undef` και ορίζεται το `$!`.
- **LENGTH ίσο με 0**: η `read` επιστρέφει αμέσως `0` και δεν αγγίζει το SCALAR. **Δεν** είναι αξιόπιστη ανίχνευση EOF· χρησιμοποιήστε `eof` γι” αυτό.
- **Αρνητικό LENGTH**: μοιραίο σφάλμα κατά τον χρόνο εκτέλεσης (`Negative length at ...`). Επικυρώστε το LENGTH πριν την κλήση.
- **Αρνητικό OFFSET του οποίου το μέτρο υπερβαίνει το τρέχον μήκος του SCALAR**: μοιραίο σφάλμα κατά τον χρόνο εκτέλεσης (`Offset outside string`). Περιορίστε με `max($offset, -length $buf)` όταν το offset υπολογίζεται.
- **Σύντομη ανάγνωση σε σωλήνα ή υποδοχή**: δεν είναι σφάλμα. Η `read` επιστρέφει λιγότερους χαρακτήρες από όσους ζητήθηκαν όποτε ο ενταμιευτής PerlIO αδειάζει πριν φτάσει το LENGTH. Επαναλάβετε σε βρόχο αν χρειάζεστε τον πλήρη αριθμό.
- **EOF στη μέση της ανάγνωσης**: επιστρέφει το μερικό πλήθος. Η επόμενη κλήση επιστρέφει `0`. Μετά από αυτό, το `$fh` παραμένει σε EOF μέχρι να κάνετε `seek` ή `clearerr`.
- **Ανάγνωση από συνδεδεμένο (tied) handle**: η `read` αποστέλλεται στη μέθοδο `READ` της κλάσης tie, η οποία είναι υπεύθυνη να τιμά τα LENGTH και OFFSET. Κακά συμπεριφερόμενες κλάσεις tie μπορούν να παραβιάσουν τη σύμβαση «μεγάλωσε

το SCALAR ώστε ο τελευταίος χαρακτήρας που διαβάστηκε να είναι ο τελευταίος χαρακτήρας».

- **Αλληλεπίδραση με `sysread`**: μην τις αναμείξετε σε ένα handle. Η `read` γεμίζει τον ενταμιευτή PerlIO σε τμήματα της επιλογής της· η `sysread` αγνοεί αυτόν τον ενταμιευτή εντελώς.
- **Δυαδικά δεδομένα σε handle λειτουργίας κειμένου**: σε συστήματα Unix-like δεν υπάρχει ξεχωριστή λειτουργία κειμένου, αλλά ένα στρώμα κωδικοποίησης εξακολουθεί να μετασχηματίζει bytes. Χρησιμοποιήστε `binmode $fh` (ή `open ... , "<:raw", ...`) πριν διαβάσετε δυαδικά δεδομένα.
- **`FILEHANDLE` ως έκφραση**: bareword ή απλό βαθμωτό είναι εντάξει. Οτιδήποτε πιο σύνθετο πρέπει να μπει σε παρενθέσεις: `read(($handles[$i]), $buf, $len)`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `open` - αποκτά το filehandle και αποφασίζει αν οι επόμενες `read` μετρούνται σε bytes ή σε χαρακτήρες μέσω του στρώματος I/O
- `sysread` - το αντίστοιχο χωρίς ενταμιευτή, απευθείας κλήση συστήματος `read(2)`· χρησιμοποιήστε το για μη μπλοκάρουσα I/O ή όταν μια σύντομη ανάγνωση έχει νόημα
- `readline` / `<$fh>` - είσοδος προσανατολισμένη σε εγγραφές που σέβεται το `$/` αντί για πλήθος bytes/χαρακτήρων
- `getc` - διαβάζει έναν μόνο χαρακτήρα· περίπου `read($fh, $c, 1)` αλλά με διαφορετική αναφορά EOF/undef
- `binmode` - αφαιρεί ή προσθέτει στρώματα I/O ώστε το `LENGTH` να είναι μονοσήμαντα πλήθος bytes ή πλήθος χαρακτήρων
- `eof` - ο σωστός τρόπος ελέγχου για τέλος αρχείου, αντί να διαβάζετε τμήμα μηδενικού μήκους

I/O

readdir

Διαβάζει την επόμενη εγγραφή, ή όλες τις υπόλοιπες εγγραφές, από ένα handle καταλόγου που άνοιξε η `opendir`.

Η `readdir` είναι το αντίστοιχο για καταλόγους της `readline`: τραβά ονόματα από ένα handle καταλόγου ένα τη φορά σε βαθμωτό περιβάλλον, ή όλα μαζί σε περιβάλλον λίστας. Το handle πρέπει να έχει ήδη ανοιχτεί με `opendir`· η επανάληψη προχωρά μια θέση που οι `rewinddir`, `seekdir` και `telldir` μπορούν να χειριστούν.

Σύνοψη

```
readdir DIRHANDLE
my $name  = readdir $dh;
my @names = readdir $dh;
while (readdir $dh) { ... }     # sets $_
```

Τι επιστρέφεται

Εξαρτάται από το περιβάλλον:

- **Βαθμωτό περιβάλλον**: το όνομα της επόμενης εγγραφής ως συμβολοσειρά, ή `undef` όταν εξαντληθεί ο κατάλογος.
- **Περιβάλλον λίστας**: κάθε υπόλοιπη εγγραφή ως λίστα συμβολοσειρών, ή η κενή λίστα όταν ο κατάλογος είναι ήδη εξαντλημένος.

Οι εγγραφές είναι **σκέτα ονόματα αρχείων**, όχι διαδρομές. Η `readdir` επιστρέφει `"foo.txt"`, ποτέ `"/some/dir/foo.txt"`. Αν σκοπεύετε να παραδώσετε το αποτέλεσμα σε file test, `open`, `stat`, ή οποιαδήποτε άλλη κλήση που αναλύει διαδρομές ως προς τον τρέχοντα κατάλογο εργασίας, προσθέστε εσείς ως πρόθεμα τον κατάλογο - η `readdir` **δεν** κάνει `chdir` στον κατάλογο που διαβάζει.

Η λίστα περιλαμβάνει πάντα τις εγγραφές `.` (τρέχων κατάλογος) και `..` (γονικός κατάλογος) όπως τις αναφέρει το υποκείμενο σύστημα αρχείων. Φιλτράρετέ τις ρητά αν δεν τις θέλετε:

```
my @real = grep { $_ ne '.' && $_ ne '..' } readdir $dh;
```

Η σειρά εξαρτάται από το σύστημα αρχείων, δεν είναι αλφαβητική. Τα ext4, XFS, tmpfs, NFS και FAT επιστρέφουν το καθένα εγγραφές στη δική του εσωτερική σειρά, η οποία μπορεί να είναι σειρά εισαγωγής, σειρά κατακερματισμού, ή κάτι εντελώς άλλο. Μην υποθέτετε ποτέ ταξινομημένη έξοδο. Αν χρειάζεστε προβλέψιμη σειρά, ταξινομήστε το αποτέλεσμα:

```
my @names = sort readdir $dh;
```

Παραδείγματα

Διαβάστε κάθε όνομα σε λίστα, απορρίψτε τις εγγραφές με τελείες, και ταξινομήστε:

```
opendir(my $dh, $some_dir) or die "opendir $some_dir: $!";
my @names = sort grep { !/^\.\.?\z/ } readdir $dh;
closedir $dh;
```

Επανάληψη μία εγγραφή τη φορά. Μια σκέτη `readdir` σε συνθήκη `while` αναθέτει στο `$_` και ελέγχει το αποτέλεσμα για **ορισμένο**, όχι για αλήθεια - οπότε μια εγγραφή με κυριολεκτικό όνομα `"0"` εξακολουθεί να κρατά τον βρόχο σε λειτουργία:

```
opendir(my $dh, $some_dir) or die "opendir $some_dir: $!";
while (readdir $dh) {
    print "$some_dir/$_\n";
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
}
closedir $dh;
```

File test με τον κατάλογο προθεμένο - η παράλειψη του προθέματος είναι το κλασικό σφάλμα της `readdir`:

```
opendir(my $dh, $some_dir) or die "opendir $some_dir: $!";
my @dotfiles = grep { /^\./ && -f "$some_dir/$_" } readdir $dh;
closedir $dh;
```

Βαθμωτό περιβάλλον - ένα όνομα ανά κλήση, το `undef` σηματοδοτεί τέλος καταλόγου:

```
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;
```

Επαναφορά και ανάγνωση για δεύτερη φορά - το handle παραμένει ανοιχτό και χρήσιμο:

```
my @first  = readdir $dh;
rewinddir $dh;
my @second = readdir $dh;   # same entries, possibly different order
```

Οριακές περιπτώσεις

- **Εξαντλημένο handle**: περαιτέρω κλήσεις επιστρέφουν `undef` σε βαθμωτό περιβάλλον, κενή λίστα σε περιβάλλον λίστας. Χρησιμοποιήστε `rewinddir` για να ξεκινήσετε ξανά.
- **Κλειστό ή μη έγκυρο handle**: επιστρέφει `undef` (βαθμωτό) ή κενή λίστα (λίστα) και θέτει το `$!`. Υπό `use warnings` η Perl εκπέμπει `readdir() attempted on invalid dirhandle`.
- **Έλεγχος ορισμένου σε `while`**: τόσο το `while (readdir $dh)` όσο και το `while (my $n = readdir $dh)` (και τα αντίστοιχά τους για `for`) αντιμετωπίζονται ως ειδικές περιπτώσεις ώστε να ελέγχουν `defined`, όχι αλήθεια. Μια εγγραφή με όνομα `"0"` επιστρέφεται και ο βρόχος συνεχίζεται.
- **Τα `.` και `..` είναι πάντα παρόντα** σε συστήματα αρχείων POSIX, ακόμα και σε διαφορετικά κενό κατάλογο. Φιλτράρετέ τα αν θέλετε μόνο πραγματικό περιεχόμενο.
- **Κανένα πρόθεμα διαδρομής**: οι εγγραφές είναι σκέτα ονόματα. Κάθε file test ή `open` πάνω στην ακατέργαστη τιμή επιστροφής ελέγχει σχετικά ως προς τον τρέχοντα κατάλογο εργασίας, όχι ως προς το `$some_dir`.
- **Καμία ταξινόμηση**: η σειρά είναι όποια επιστρέφει ο πυρήνας. Εφαρμόστε ρητά `sort` αν η σταθερότητα έχει σημασία.
- **Ταυτόχρονες αλλαγές καταλόγου**: εγγραφές που προστίθενται ή αφαιρούνται κατά την επανάληψη μπορεί να εμφανιστούν ή όχι, ανάλογα με την πλατφόρμα.

Η μόνη ασφαλής υπόθεση είναι ότι εγγραφές που είναι παρούσες κατά τη στιγμή της `opendir` και δεν αφαιρέθηκαν θα αναφερθούν.

- **Κωδικοποίηση**: η `readdir` επιστρέφει συμβολοσειρές bytes όπως τις αποθηκεύει ο πυρήνας. Σε σύστημα αρχείων που χρησιμοποιεί ονόματα αρχείων UTF-8, εφαρμόστε `Encode::decode_utf8` ή ρυθμίστε το pragma `open` αν χρειάζεστε συμβολοσειρές χαρακτήρων.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `opendir` - ανοίγει το handle καταλόγου από το οποίο διαβάζει η `readdir`· πάντα η πρώτη κλήση στην ακολουθία
- `closedir` - απελευθερώνει το handle μόλις ολοκληρωθεί η επανάληψη
- `rewinddir` - επαναφέρει τη θέση επανάληψης στην αρχή του καταλόγου
- `seekdir` - μεταπηδά σε μια προηγουμένως καταγεγραμμένη θέση
- `telldir` - καταγράφει την τρέχουσα θέση για μετέπειτα `seekdir`
- `glob` - αντιστοίχιση μοτίβων τύπου κελύφους επί του περιεχομένου καταλόγου· υψηλότερου επιπέδου όταν θέλετε `*.txt` αντί για κάθε εγγραφή

I/O

readline

Διαβάζει μία ή περισσότερες εγγραφές από ένα filehandle.

Η `readline` είναι το πρωτογενές στοιχείο εισόδου πίσω από τον τελεστή «διαμάντι» `<FH>`. Διαβάζει από το filehandle που ονομάζει το `EXPR` (typeglob, αναφορά glob ή `IO::Handle`), χρησιμοποιώντας την τρέχουσα τιμή της `$/` για να αποφασίσει πού τελειώνει μία εγγραφή. Αν το `EXPR` παραλειφθεί, διαβάζεται το `*ARGV`. Σε βαθμωτό περιβάλλον επιστρέφει την επόμενη εγγραφή· σε περιβάλλον λίστας επιστρέφει κάθε υπολειπόμενη εγγραφή. Το τέλος αρχείου σηματοδοτείται με `undef` (βαθμωτό) ή με κενή λίστα.

Σύνοψη

```
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
```

Τι επιστρέφεται

- **Βαθμωτό περιβάλλον**: την επόμενη εγγραφή ως συμβολοσειρά, ή `undef` σε τέλος αρχείου ή σε σφάλμα. Η εγγραφή περιλαμβάνει τον τερματιστή της (την τρέχουσα `$/`) εκτός αν στην τελευταία εγγραφή του αρχείου λείπει. Μετά την επιστρεφόμενη τιμή, η `$.` αυξάνεται - δείτε την ενότητα για την καθολική κατάσταση παρακάτω.
- **Περιβάλλον λίστας**: λίστα με κάθε υπολειπόμενη εγγραφή μέχρι το τέλος αρχείου. Κενή λίστα σημαίνει ότι το handle ήταν ήδη στο EOF.
- **Λειτουργία slurp σε κενό αρχείο**: όταν η `$/` είναι `undef` και το περιβάλλον βαθμωτό, η πρώτη κλήση σε handle μηδέν bytes επιστρέφει `''` (κενή συμβολοσειρά, ορισμένη)· η επόμενη κλήση επιστρέφει `undef`. Αυτό επιτρέπει στο `defined($slurp = readline $fh)` να διακρίνει ένα κενό αρχείο από ένα ανύπαρκτο χωρίς ξεχωριστό έλεγχο `-s`.

Καθολική κατάσταση που επηρεάζει

- `$/` - διαχωριστής εγγραφών εισόδου. Ελέγχει τι μετρά ως «μία εγγραφή». Η προεπιλογή είναι `"\n"`. Οι ειδικές τιμές καλύπτονται στην ενότητα *Λειτουργίες εγγραφών* παρακάτω. Γνωστή και ως `$INPUT_RECORD_SEPARATOR` ή `$RS` υπό `use English`.
- `$.` - τρέχων αριθμός γραμμής εισόδου του handle που διαβάστηκε τελευταίο. Η `readline` αυξάνει την `$.` μία φορά ανά εγγραφή που επιστρέφεται. Η `$.` μηδενίζεται όταν το handle κλείνει ρητά με `close`, αλλά **όχι** όταν το `*ARGV` περνά από το ένα αρχείο στο επόμενο - η μέτρηση συνεχίζει σε όλη την είσοδο. Επίσης `$INPUT_LINE_NUMBER` ή `$NR` υπό `use English`.
- `$_` - το προεπιλεγμένο βαθμωτό. Όταν μια κλήση `readline` (ή το ισοδύναμό της `<FH>`) χρησιμοποιείται **ως συνθήκη βρόχου `while` ή `for`**, η Perl εκχωρεί έμμεσα το αποτέλεσμα στο `$_` και ελέγχει `defined`, όχι αλήθεια. Έτσι, το `while (<$fh>) { ... }` είναι ασφαλές σε γραμμές που περιέχουν `"0"` ή `""`.
- `$!` - errno. Σε σφάλμα ανάγνωσης, η `readline` επιστρέφει `undef` (βαθμωτό) ή μια κολοβωμένη λίστα, και ορίζει την `$!` στο σφάλμα του λειτουργικού.
- `@ARGV` - όταν παραλείπεται το `EXPR` (ή το handle είναι το `ARGV`), κάθε τέλος αρχείου κάνει την Perl να ανοίξει με `open` το επόμενο όνομα αρχείου από το `@ARGV` και να συνεχίσει την ανάγνωση. Κενό `@ARGV` κάνει το `ARGV` να διαβάζει το `STDIN`.

Λειτουργίες εγγραφών

Η `$/` επιλέγει μία από τέσσερις λειτουργίες ανάγνωσης:

- **Λειτουργία γραμμής** (προεπιλογή) - η `$/` είναι μη κενή συμβολοσειρά. Μια εγγραφή τελειώνει στην επόμενη εμφάνιση εκείνης της συμβολοσειράς. Με την προεπιλογή `"\n"`, μία κλήση επιστρέφει μία γραμμή, περιλαμβανομένης της νέας γραμμής της.
- **Λειτουργία παραγράφου** - `$/ = ""`. Μια εγγραφή τελειώνει στην επόμενη σειρά δύο ή περισσοτέρων διαδοχικών νέων γραμμών. Οι κενές γραμμές πριν την πρώτη παράγραφο παραλείπονται· η/οι κενή/ές γραμμή/ές που τερματίζουν περιλαμβάνονται στην επιστρεφόμενη συμβολοσειρά. Χρησιμοποιήστε `local $/ = ""` για να περιορίσετε την αλλαγή σε ένα μπλοκ.

- **Λειτουργία slurp** - `$/ = undef`. Η πρώτη κλήση επιστρέφει το σύνολο του υπολειπόμενου περιεχομένου του handle ως μία συμβολοσειρά· η επόμενη κλήση επιστρέφει `undef`. Για κενό αρχείο, δείτε την ειδική περίπτωση στην ενότητα *Τι επιστρέφεται*.
- **Λειτουργία εγγραφών σταθερού μήκους** - `$/ = \N` (αναφορά σε θετικό ακέραιο) ή `$/ = \"N"`. Κάθε κλήση διαβάζει ακριβώς `N` bytes, ή λιγότερα στο τέλος αρχείου. Χρήσιμο για binary framing. Το αποτέλεσμα εξακολουθεί να μετρά ως μία εγγραφή για το `$.`.

Οι αλλαγές στην `$/` τίθενται σε ισχύ στην επόμενη κλήση `readline`. Οριοθετήστε τες με `local` ώστε μια υπορουτίνα να μην αλλάζει σιωπηλά την αντίληψη του καλούντος για το τι σημαίνει «γραμμή».

Παραδείγματα

Ιδιωματική ανάγνωση γραμμή προς γραμμή. Η συνθήκη `while` εκχωρεί έμμεσα στο `$_` και ελέγχει `defined`:

```
open my $fh, "<", "input.txt" or die "open: $!";
while (<$fh>) {
    chomp;
    print "line $.: $_\n";
}
close $fh;
```

Ρητή κλήση `readline` με ανίχνευση σφαλμάτων. Η `eof` πρέπει να ελέγχεται ξεχωριστά από την τιμή επιστροφής, επειδή μια νόμιμη γραμμή μπορεί να είναι η κενή συμβολοσειρά:

```
while ( ! eof($fh) ) {
    defined( my $line = readline $fh )
        or die "readline failed: $!";
    # ... process $line ...
}
```

Slurp ολόκληρου αρχείου σε ένα βαθμωτό:

```
my $contents = do {
    local $/;                   # undef - slurp mode
    open my $fh, "<", $path or die "open $path: $!";
    readline $fh;
};
```

Λειτουργία παραγράφου - ανάγνωση ενός μπλοκ επικεφαλίδας μηνύματος σε στιλ RFC-822:

```
open my $fh, "<", "message.eml" or die $!;
local $/ = "";
my $headers = <$fh>;            # up to the first blank line
my $body    = do { local $/; <$fh> };
```

Λειτουργία εγγραφών σταθερού μήκους - ανάγνωση μπλοκ 512 bytes από αρχείο συσκευής:

```
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 ...
}
```

Περιβάλλον λίστας - τραβήξτε κάθε υπολειπόμενη γραμμή με τη μία. Φθηνό για μικρά αρχεία, καταστροφικό για μεγάλα:

```
open my $fh, "<", "hosts" or die $!;
my @lines = <$fh>;              # every line, including terminators
chomp @lines;
```

Ανάγνωση από το *ARGV χωρίς ρητό handle - το κλασικό ιδίωμα φίλτρου της Perl. Το @ARGV καταναλώνεται ως λίστα ονομάτων αρχείων, ή διαβάζεται το STDIN αν το @ARGV είναι κενό:

```
while (defined(my $line = readline)) {
    print $line;                # cat(1) in one line
}
```

Οριακές περιπτώσεις

- **EOF έναντι σφάλματος**: η `readline` σε βαθμωτό επιστρέφει `undef` και για τα δύο. Καλέστε `eof($fh)` ή ελέγξτε την `$!` για να τα διακρίνετε. Μέσα σε `while (<$fh>)` ο βρόχος εξέρχεται και στις δύο περιπτώσεις· προσθέστε ρητό έλεγχο `defined` μαζί με `$!` όταν χρειάζεται να τα ξεχωρίσετε.
- **Αρχείο χωρίς τελική νέα γραμμή**: η τελευταία εγγραφή επιστρέφεται χωρίς τερματιστή σε λειτουργία γραμμής. Η `chomp` δεν έχει αποτέλεσμα σε τέτοια τιμή, που είναι η σωστή συμπεριφορά.
- **Λειτουργία παραγράφου σε αρχείο με μόνο κενές γραμμές**: επιστρέφει `undef` αμέσως - δεν υπάρχει μη κενή παράγραφος για να αποδοθεί.
- **Λειτουργία slurp, κενό αρχείο**: η πρώτη κλήση επιστρέφει `''` (ορισμένη, μηδενικού μήκους)· η δεύτερη επιστρέφει `undef`. Μην διακλαδώνετε βάσει αλήθειας της πρώτης ανάγνωσης.
- **Λειτουργία σταθερού μήκους με σύντομη ανάγνωση στο EOF**: επιστρέφει τα υπολειπόμενα bytes (πιθανώς λιγότερα από `N`), και κατόπιν `undef` στην επόμενη κλήση. Ελέγξτε `length` αν χρειάζεστε ακριβώς `N`.
- **Το μαγικό handle ARGV**: όταν διαβάζετε από το *ARGV, η `eof()` χωρίς όρισμα επιστρέφει αληθές μόνο στο τέλος **όλων** των αρχείων του @ARGV, ενώ η `eof(ARGV)` είναι αληθής στο τέλος **κάθε** αρχείου. Δεν μπορείτε να χρησιμοποιήσετε το ιδίωμα `defined(readline) or die "...: $!"` έναντι του *ARGV για να διακρίνετε EOF από σφάλμα - ανοίξτε κάθε όνομα αρχείου μόνοι σας αν χρειάζεστε αυτό.
- **Ήδη κλεισμένο handle**: η `readline` επιστρέφει `undef` και ορίζει την `$!` σε "Bad

`file descriptor"`. Υπό `use warnings` εκπέμπεται προειδοποίηση `readline() on closed filehandle`.

- **Στρώματα κωδικοποίησης**: η `readline` επιστρέφει ό,τι παράγει η στοίβα PerlIO του handle. Ένα στρώμα `:utf8` ή `:encoding(...)` αποκωδικοποιεί bytes σε χαρακτήρες· ένα handle `:raw` επιστρέφει bytes. Η ανάμιξη μιας πολυβάιτας `$/` με handle bytes μπορεί να σπάσει χαρακτήρες.
- **Αλλαγή της `$/` ενδιάμεσα σε εγγραφή**: μόνο η επόμενη κλήση `readline` παρατηρεί τον νέο διαχωριστή. Όποια ήδη ενδιάμεσα στον buffer δεδομένα σαρώνονται ξανά ως προς αυτόν.
- **`<FH>` έναντι `<$fh>` έναντι `<>`**: όλες αυτές είναι `readline` εσωτερικά. Το `<FH>` διαβάζει από το typeglob `FH`· το `<$fh>` διαβάζει από το handle στο `$fh`· το `<>` (ή `<<>>` για ασφαλέστερο χειρισμό ονομάτων αρχείων) διαβάζει από το `*ARGV`. Η μορφή `<pattern>` με οποιοδήποτε άλλο περιεχόμενο είναι `glob`, όχι `readline` - εντελώς άλλη συνάρτηση.
- **Εκχώρηση στη συνθήκη βρόχου**: το `while (my $line = <$fh>) { ... }` ελέγχει `defined`, όχι αλήθεια, μόνο επειδή ο αναλυτής το επανεγγράφει. Το `while (($line = <$fh>)) { ... }` με τις επιπλέον παρενθέσεις **δεν** δέχεται την επανεγγραφή, και θα κάνει βρόχο επ” άπειρον σε είσοδο που περιέχει `"0\n"`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `open` - αποκτήστε το filehandle που καταναλώνει η `readline`
- `read` - ανάγνωση σταθερού μήκους σε bytes που αγνοεί την `$/`· χρησιμοποιήστε όταν θέλετε μέτρηση σε bytes, όχι εγγραφή
- `getc` - ανάγνωση ενός χαρακτήρα, που αγνοεί επίσης την `$/`
- `eof` - ελέγχει αν η επόμενη `readline` θα επέστρεφε `undef`· συμπεριφέρεται ειδικά για το `*ARGV`
- `$/` - διαχωριστής εγγραφών εισόδου· ελέγχει τις λειτουργίες γραμμής, παραγράφου, slurp και σταθερού μήκους
- `$.` - τρέχων αριθμός γραμμής εισόδου, αυξάνεται από κάθε επιτυχημένη `readline`

Filehandles, αρχεία, κατάλογοι

readlink

Επιστρέφει τη διαδρομή στόχου στην οποία δείχνει ένας συμβολικός σύνδεσμος.

Η `readlink` ρωτά τον πυρήνα ποια συμβολοσειρά είναι αποθηκευμένη μέσα σε έναν συμβολικό σύνδεσμο, χωρίς να τον ακολουθεί και χωρίς να αγγίζει το αρχείο στο οποίο τελικά αναλύεται ο σύνδεσμος. Το όρισμα είναι η διαδρομή του ίδιου του συνδέσμου· η τιμή επιστροφής είναι η ακατέργαστη συμβολοσειρά στόχου όπως καταγράφηκε όταν δημιουργήθηκε ο σύνδεσμος. Αν παραλειφθεί το EXPR, χρησιμοποιείται το `$_`.

Σύνοψη

```
readlink EXPR
readlink
```

Τι επιστρέφεται

Σε επιτυχία, ο στόχος του συνδέσμου ως συμβολοσειρά - ακριβώς τα bytes που είναι αποθηκευμένα στο symlink, χωρίς καμία ερμηνεία. Σε αποτυχία, `undef`, με το `$!` να τίθεται στο σφάλμα του συστήματος. Τυπικοί κωδικοί σφάλματος:

- `EINVAL` - η διαδρομή υπάρχει αλλά δεν είναι συμβολικός σύνδεσμος
- `ENOENT` - η διαδρομή δεν υπάρχει
- `EACCES` - ένα στοιχείο καταλόγου δεν είναι αναζητήσιμο
- `ELOOP`, `ENAMETOOLONG`, `ENOTDIR` - οι συνηθισμένες αποτυχίες ανάλυσης διαδρομής σε στοιχεία που οδηγούν στον σύνδεσμο

Η επιστρεφόμενη συμβολοσειρά **δεν** κανονικοποιείται και **δεν** μετατρέπεται σε απόλυτη. Ένας σύνδεσμος που δημιουργήθηκε με `ln -s ../lib/libfoo.so libfoo.so` επιστρέφει το κυριολεκτικό `"../lib/libfoo.so"`, σχετικά ως προς τον κατάλογο που περιέχει τον ίδιο τον σύνδεσμο - όχι ως προς τον τρέχοντα κατάλογο εργασίας. Η ανάλυση του στόχου σε απόλυτη, πραγματική διαδρομή είναι ευθύνη του καλούντος· δείτε `Cwd::abs_path`.

Καθολική κατάσταση που επηρεάζει

- `$_` - διαβάζεται ως το προεπιλεγμένο όρισμα όταν παραλείπεται το `EXPR`
- `$!` - τίθεται σε αποτυχία στο C `errno` από την υποκείμενη κλήση `readlink(2)`

Παραδείγματα

Ανάγνωση του στόχου ενός συνδέσμου απευθείας:

```
my $target = readlink "/usr/bin/vi";
print defined $target ? "points to $target\n" : "not a symlink: $!\n
 ↪";
```

Μορφή με προεπιλεγμένο όρισμα μέσα σε βρόχο επί ονομάτων αρχείων:

```
for (@paths) {
    next unless -l;                     # skip anything that isn't a
 ↪symlink
    my $t = readlink;                   # reads $_
    print "$_ -> $t\n" if defined $t;
}
```

Προστατευτείτε ρητά έναντι μη-symlinks. Η `readlink` σε κανονικό αρχείο επιστρέφει `undef` και θέτει το `$!` σε `EINVAL`:

```
my $t = readlink $path;
if (!defined $t) {
    die "readlink $path: $!" unless $!{EINVAL};
    # not a symlink - handle as a plain file
}
```

Αναλύστε έναν σχετικό στόχο συνδέσμου ως προς τον δικό του κατάλογο, που είναι η σωστή βάση - όχι ο τρέχων κατάλογος εργασίας:

```
use File::Basename qw(dirname);
use File::Spec;

my $target = readlink $link // die "readlink $link: $!";
my $resolved = File::Spec->rel2abs($target, dirname($link));
```

Κανονικοποιήστε πλήρως (ακολουθώντας κάθε σύνδεσμο στην αλυσίδα, καταργώντας τα . και .., παράγοντας απόλυτη διαδρομή) με Cwd::abs_path:

```
use Cwd qw(abs_path);
my $real = abs_path($link);              # undef if any component is␣
↪missing
```

Οριακές περιπτώσεις

- **Δεν είναι symlink**: η readlink σε κανονικό αρχείο, κατάλογο, ή οτιδήποτε άλλο που δεν είναι συμβολικός σύνδεσμος επιστρέφει undef και θέτει το $! σε EINVAL. Ελέγξτε πρώτα με -l αν θέλετε να διακρίνετε το «δεν είναι σύνδεσμος» από άλλα σφάλματα.
- **Ελλείπουσα διαδρομή**: ένα στοιχείο του EXPR που δεν υπάρχει δίνει ENOENT, όπως κάθε άλλη αποτυχία αναζήτησης διαδρομής.
- **Προεπιλεγμένο όρισμα**: η σκέτη readlink χρησιμοποιεί το $_. Δεν εκπέμπεται προειδοποίηση όταν το $_ είναι undef· η κλήση απλώς αποτυγχάνει με σφάλμα αναζήτησης διαδρομής.
- **Τελικά στοιχεία**: η readlink λειτουργεί μόνο επί του τελικού στοιχείου - ενδιάμεσα symlinks στη διαδρομή ακολουθούνται κανονικά. Η readlink "/a/b/c" όπου το /a είναι το ίδιο symlink διαβάζει τον σύνδεσμο στο c, όχι στο a.
- **Σχετικός στόχος, αμετάβλητος**: η επιστρεφόμενη συμβολοσειρά αποθηκεύεται αυτολεξεί. Μην υποθέτετε ποτέ ότι η συμβολοσειρά είναι απόλυτη, και ποτέ μην την ενώσετε με τον τρέχοντα κατάλογο εργασίας - ενώστε την με τον κατάλογο που περιέχει τον σύνδεσμο, ή τροφοδοτήστε τη διαδρομή του συνδέσμου (όχι τον στόχο) στην Cwd::abs_path.
- **Κρεμασμένοι σύνδεσμοι**: ένας σύνδεσμος του οποίου ο στόχος δεν υπάρχει εξακολουθεί να διαβάζεται κανονικά. Η readlink δεν κάνει stat στον στόχο. Χρησιμοποιήστε lstat για να επιθεωρήσετε τον ίδιο τον σύνδεσμο, stat για να επιθεωρήσετε σε ό,τι δείχνει (και να αποτύχετε αν ο στόχος λείπει).
- **Πλατφόρμες χωρίς symlinks**: σε συστήματα που δεν υλοποιούν συμβολικούς συνδέσμους, η readlink εγείρει εξαίρεση αντί να επιστρέψει undef. Το Linux (ο

μοναδικός υποστηριζόμενος στόχος του pperl) τα έχει πάντοτε, οπότε αυτό είναι σχετικό μόνο για φορητό κώδικα που μοιράζεται με άλλους στόχους.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `symlink` - δημιουργία συμβολικού συνδέσμου· η αντίστροφη πράξη
- `link` - δημιουργία σκληρού συνδέσμου (δεν διαβάζεται με `readlink`· οι σκληροί σύνδεσμοι είναι μη-διακριτοί από το αρχικό όνομα)
- `lstat` - `stat` στον ίδιο τον συμβολικό σύνδεσμο αντί στον στόχο του· χρησιμοποιήστε μαζί με την `readlink` για πλήρη επιθεώρηση συνδέσμου
- `-l` - file test για «είναι συμβολικός σύνδεσμος»· ελέγξτε πριν την κλήση της `readlink` για να αποσαφηνίσετε το `EINVAL` από άλλα σφάλματα
- `$!` - errno που τίθεται σε αποτυχία· το `$!{EINVAL}` διακρίνει το «δεν είναι symlink» από άλλα σφάλματα

Διεργασίες

readpipe

Εκτελεί μια εντολή κελύφους και επιστρέφει την τυπική της έξοδο.

Η `readpipe` παραδίδει την `EXPR` στο κέλυφος, περιμένει να τελειώσει το παιδί, και επιστρέφει ό,τι έγραψε το παιδί στην τυπική του έξοδο. Είναι η μορφή με όνομα του τελεστή `qx` (backticks) - και το `` `cmd` `` και το `qx/cmd/` μεταγλωττίζονται κάτω σε `readpipe`. Καταφύγετε στη μορφή με όνομα όταν θέλετε την εντολή σε μεταβλητή, όταν θέλετε να καταστείλετε την παρεμβολή διπλών εισαγωγικών που εκτελεί η `qx`, ή όταν θέλετε να διαβάσετε μια αποτυπωμένη εντολή από το `@ARGV`. Αν η `EXPR` παραλείπεται, χρησιμοποιείται η `$_`.

Σύνοψη

```
readpipe EXPR
readpipe
$out  = readpipe $cmd;
@rows = readpipe $cmd;
```

Τι επιστρέφεται

Το περιβάλλον αποφασίζει το σχήμα:

- **Βαθμωτό περιβάλλον** - μία συμβολοσειρά που περιέχει ολόκληρη την αποτυπωμένη έξοδο, μαζί με τα newlines.
- **Περιβάλλον λίστας** - λίστα εγγραφών, διασπασμένων επί της `$/` (ο διαχωριστής εγγραφών εισόδου, newline προεπιλεγμένα). Ο διαχωριστής παραμένει προσαρτημένος σε κάθε στοιχείο εκτός αν τον αφαιρέσετε με `chomp`.

Η $? τίθεται στην κατάσταση αναμονής του παιδιού αφού τελειώσει η εντολή, ακριβώς όπως με τη system. Μια μη μηδενική $? είναι το συνηθισμένο σήμα ότι η εντολή απέτυχε - η τιμή επιστροφής από μόνη της δεν μπορεί να σας το πει, καθώς μια εντολή που αποτυγχάνει μπορεί παρ" όλα αυτά να έχει γράψει έξοδο (ή τίποτα) πριν εξέλθει.

```
my $out = readpipe "uname -a";
die "uname failed: \$? = $?" if $?;
```

Καθολική κατάσταση που επηρεάζει

- $_ - χρησιμοποιείται ως εντολή όταν η EXPR παραλείπεται.
- $/ - ελέγχει πώς η έξοδος σε περιβάλλον λίστας διασπάται σε εγγραφές. Θέστε το σε undef για να ρουφήξετε ολόκληρη την έξοδο σε ένα μόνο στοιχείο, ή σε "" για λειτουργία παραγράφου.
- $? - τίθεται στην κατάσταση αναμονής του παιδιού κατά την επιστροφή.
- $! - τίθεται αν το ίδιο το κέλυφος δεν μπόρεσε να εκκινήσει.
- %ENV - το κέλυφος κληρονομεί το τρέχον περιβάλλον.

Παραδείγματα

Αποτύπωση μίας γραμμής εξόδου:

```
my $host = readpipe "hostname";
chomp $host;
print "running on $host\n";
```

Εντολή αποθηκευμένη σε μεταβλητή - ο λόγος που υπάρχει η readpipe ως συνάρτηση με όνομα. Τα backticks θα λειτουργούσαν επίσης εδώ, αλλά η readpipe καθιστά την πρόθεση ρητή και αποφεύγει τον οπτικό θόρυβο της εμφωλευμένης χρήσης εισαγωγικών:

```
my $cmd = "ls -1 /etc";
my @entries = readpipe $cmd;
chomp @entries;
print scalar(@entries), " entries under /etc\n";
```

Ρούφηγμα όλων σε μία συμβολοσειρά απενεργοποιώντας τον διαχωρισμό εγγραφών:

```
local $/;                                  # slurp mode
my $log = readpipe "dmesg";
print length($log), " bytes of kernel log\n";
```

Λειτουργία παραγράφου - κάθε στοιχείο είναι ένα μπλοκ διαχωρισμένο με κενή γραμμή:

```
local $/ = "";
my @paragraphs = readpipe "cat README";
print scalar(@paragraphs), " paragraphs\n";
```

Ελέγξτε την κατάσταση εξόδου σωστά. Η τιμή επιστροφής είναι η έξοδος· η επιτυχία ζει στην $?:

```
my @files = readpipe "find . -name '*.tmp' 2>/dev/null";
if ($? == 0) {
    chomp @files;
    unlink @files;
} else {
    warn "find exited with status ", ($? >> 8), "\n";
}
```

Χρησιμοποιήστε την $_ ως εντολή. Σπάνια η πιο σαφής μορφή, αλλά νόμιμη:

```
$_ = "date";
my $when = readpipe;                    # same as readpipe $_
```

Οριακές περιπτώσεις

- **Καμία προστασία από παρεμβολή.** Η `readpipe` παρεμβάλλει το όρισμά της ακριβώς όπως μια συμβολοσειρά σε διπλά εισαγωγικά - όπως και η qx. Η `readpipe "rm $path"` με την `$path` να προέρχεται από είσοδο χρήστη είναι τρύπα ένεσης κελύφους. Παραθέστε την τιμή σε εισαγωγικά, ή καλύτερα, στρέψτε σε μορφή λίστας `system` / piped open και παρακάμψτε το κέλυφος εντελώς.
- **Η τυπική έξοδος σφαλμάτων δεν αποτυπώνεται.** Επιστρέφει μόνο η τυπική έξοδος του παιδιού. Ανακατευθύνετε την stderr μέσα στην ίδια την εντολή αν θέλετε να συγχωνευθεί: `readpipe "cmd 2>&1"`.
- **Το κέλυφος είναι υποχρεωτικό.** Ακόμη και μια μονολεκτική `readpipe "true"` περνά μέσα από το `/bin/sh`, που σημαίνει ότι ισχύουν τα εισαγωγικά κελύφους, το globbing και η ανάπτυξη μεταβλητών περιβάλλοντος. Δεν υπάρχει μορφή λίστας.
- **Μεγάλη ενταμίευση εξόδου στη μνήμη.** Ολόκληρη η έξοδος του παιδιού διαβάζεται πριν επιστρέψει η `readpipe`. Για εντολή που παράγει gigabytes, χρησιμοποιήστε αντ” αυτού piped open και διαβάστε σταδιακά.
- **Κενή έξοδος, μηδενική έξοδος.** Μια επιτυχημένη εντολή που δεν γράφει τίποτα επιστρέφει την κενή συμβολοσειρά σε βαθμωτό περιβάλλον και την κενή λίστα σε περιβάλλον λίστας. Ξεχωρίστε το «έτρεξε και δεν είπε τίποτα» από το «δεν έτρεξε ποτέ» ελέγχοντας την $?.
- **Το παιδί πέθανε από σήμα.** Η $? κωδικοποιεί τόσο τον κωδικό εξόδου όσο και το σήμα
  - μασκάρετε με `$? & 127` για να πάρετε τον αριθμό σήματος, `$? >> 8` για τον κωδικό εξόδου.
- **Κανόνες εισαγωγικών στα Windows.** Η `readpipe` παραδίδει τη συμβολοσειρά στο κέλυφος της πλατφόρμας· στα Windows αυτό είναι το `cmd.exe`, του οποίου οι κανόνες εισαγωγικών και ανακατεύθυνσης διαφέρουν από το POSIX. Η pperl είναι μόνο για Linux, οπότε αυτό είναι σχετικό μόνο όταν εισάγετε σενάρια προς αυτήν.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `qx` - η μορφή τελεστή· τα `` `cmd` `` και `qx/cmd/` είναι συντομογραφία της `readpipe` και φέρουν ταυτόσημη σημασιολογία
- `system` - εκτελέστε εντολή χωρίς αποτύπωση εξόδου· χρησιμοποιήστε την όταν σας ενδιαφέρει μόνο η κατάσταση εξόδου
- `exec` - αντικαθιστά την τρέχουσα διεργασία με την εντολή· καμία επιστροφή, καμία αποτύπωση
- `open` - piped open (`open my $fh, "-|", @cmd`) ροεί την έξοδο του παιδιού γραμμή-προς-γραμμή χωρίς να την ενταμιεύει ολόκληρη
- `$?` - κατάσταση αναμονής παιδιού που τίθεται από κάθε ενσωματωμένη που εκκινεί διεργασία
- `$/` - διαχωριστής εγγραφών εισόδου που διασπά την έξοδο σε περιβάλλον λίστας

Υποδοχές (sockets)

recv

Διαβάζει ένα εισερχόμενο μήνυμα από υποδοχή σε βαθμωτό.

Η `recv` διαβάζει μέχρι `LENGTH` bytes από το `SOCKET` στο `SCALAR`, αντικαθιστώντας ό,τι κρατούσε προηγουμένως το `SCALAR`. Το `SCALAR` μεγαλώνει ή συρρικνώνεται στο πραγματικό πλήθος των bytes που ελήφθησαν. Το `FLAGS` είναι η ίδια μάσκα bits που δέχεται η υποκείμενη κλήση συστήματος `recvfrom(2)` - `0` για συνηθισμένες αναγνώσεις· οι συνηθισμένες μη μηδενικές τιμές είναι `MSG_PEEK` (κοίταγμα χωρίς κατανάλωση), `MSG_WAITALL` (μπλοκάρισμα μέχρι να φτάσουν `LENGTH` bytes), και `MSG_DONTWAIT` (ποτέ μπλοκάρισμα). Για υποδοχή χωρίς σύνδεση η `recv` επιστρέφει την πακεταρισμένη διεύθυνση του αποστολέα· για συνδεδεμένη υποδοχή επιστρέφει την κενή συμβολοσειρά. Στο παρασκήνιο αυτό είναι η `recvfrom(2)`, όχι η `recv(2)`.

Σύνοψη

```
recv SOCKET, SCALAR, LENGTH, FLAGS
```

Τι επιστρέφεται

- Σε επιτυχία από υποδοχή χωρίς σύνδεση (UDP, UNIX datagram): η διεύθυνση του αποστολέα ως πακεταρισμένη συμβολοσειρά sockaddr. Κάντε unpack με τις `sockaddr_in`, `sockaddr_in6` ή `sockaddr_un` του `Socket` ανάλογα με την οικογένεια διευθύνσεων.
- Σε επιτυχία από συνδεδεμένη υποδοχή (TCP, συνδεδεμένο UDP): η κενή συμβολοσειρά `""`. Η κενή συμβολοσειρά είναι **αληθής** για τον έλεγχο ορισμένου-έναντι-undef αλλά **ψευδής** σε λογικό περιβάλλον - ελέγξτε με `defined`, όχι με απλή αληθοφάνεια.

- Σε σφάλμα: `undef`, με το `$!` ρυθμισμένο στο `errno` από την αποτυχημένη κλήση `recvfrom(2)`. Τυπικά σφάλματα: `EAGAIN` / `EWOULDBLOCK` σε μη μπλοκαριστική υποδοχή χωρίς δεδομένα έτοιμα, `ECONNRESET` σε επαναφορά TCP ομότιμου, `EBADF` σε κλειστό handle.

Το `SCALAR` πάντα αλλάζει μέγεθος στο πλήθος bytes που διαβάστηκε. Ανάγνωση μηδέν bytes σε συνδεδεμένη υποδοχή ροής σημαίνει ότι ο ομότιμος έκλεισε καθαρά την πλευρά εγγραφής του - η `recv` επιστρέφει `""` και το `SCALAR` γίνεται η κενή συμβολοσειρά.

Καθολική κατάσταση που επηρεάζει

- Το `$!` τίθεται σε σφάλμα στο `errno` του συστήματος.
- Καμία άλλη ειδική μεταβλητή δεν εμπλέκεται. Η `recv` δεν τηρεί τις `$/`, `$,`, ή `$\` - αυτές είναι έννοιες I/O προσανατολισμένες σε γραμμές και η `recv` είναι πρωτογενής λειτουργία ακατέργαστων datagrams / ροής bytes.

Παραδείγματα

Ανάγνωση ενός UDP datagram και αναγνώριση του αποστολέα:

```
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";
```

Ανάγνωση από συνδεδεμένη υποδοχή TCP. Η διεύθυνση αποστολέα είναι κενή, οπότε ελέγξτε με `defined` αντί για λογική αλήθεια:

```
defined(recv($sock, my $buf, 4096, 0))
    or die "recv: $!";
# $buf now holds 0..4096 bytes; 0 means the peer closed.
```

Κοιτάξτε το επόμενο datagram χωρίς να το καταναλώσετε. Η δεύτερη κλήση επιστρέφει τα ίδια 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.
```

Βρόχος μη μπλοκαριστικής ανάγνωσης. Το `EAGAIN` / `EWOULDBLOCK` είναι το φυσιολογικό σήμα «δεδομένα ακόμα όχι», όχι πραγματικό σφάλμα:

```
use Errno qw(EAGAIN EWOULDBLOCK);
while (1) {
    my $from = recv($sock, my $buf, 1500, 0);
    unless (defined $from) {
        last if $! == EAGAIN || $! == EWOULDBLOCK;
        die "recv: $!";
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    }
    handle_datagram($from, $buf);
}
```

Επιβολή ανάγνωσης πλήρους μήκους σε υποδοχή ροής που ενδέχεται να κατατμήσει το μήνυμα - το `MSG_WAITALL` μπλοκάρει μέχρι να φτάσουν `LENGTH` bytes ή να πεθάνει η σύνδεση:

```
use Socket;
defined(recv($sock, my $frame, 64, MSG_WAITALL))
    or die "recv: $!";
# $frame is exactly 64 bytes, or the peer closed early.
```

Οριακές περιπτώσεις

- **Υποδοχή με στρώμα UTF-8**: αν το `SOCKET` έχει στρώμα `:utf8` (απευθείας ή μέσω `:encoding(...)`), η `recv` ρίχνει εξαίρεση. Η `recv` είναι προσανατολισμένη σε bytes και δεν αλληλεπιδρά με στρώματα PerlIO· χρησιμοποιήστε `binmode` για να αφαιρέσετε το στρώμα πριν την ανάγνωση, ή αποκωδικοποιήστε τα αποτελούμενα bytes μόνοι σας.
- **LENGTH έναντι μεγέθους datagram**: για υποδοχές datagram, αν το `LENGTH` είναι μικρότερο από το εισερχόμενο datagram, το πλεόνασμα **απορρίπτεται σιωπηλά**. Διαστασιολογήστε τον ενταμιευτή στο MTU του πρωτοκόλλου (τυπικά 1500 για UDP πάνω από Ethernet, 65507 για το μέγιστο IPv4 UDP).
- **Η επιστροφή `""` δεν είναι αρκετά ψευδής**: η `recv` σε συνδεδεμένη υποδοχή επιστρέφει την κενή συμβολοσειρά, την οποία το `!` αντιμετωπίζει ως ψευδή. Κώδικας όπως `recv(...) or die $!` πεθαίνει σε κάθε επιτυχή συνδεδεμένη ανάγνωση. Χρησιμοποιήστε `defined`:

  ```
  defined(recv($sock, $buf, $len, 0)) or die "recv: $!";
  ```

- **Αποτέλεσμα μηδέν bytes σε υποδοχή ροής**: η `recv` επιστρέφει `""` και το `SCALAR` γίνεται κενό όταν ο ομότιμος έχει κάνει ομαλό κλείσιμο. Διακρίνετε το τέλος ροής από πραγματικό σφάλμα ελέγχοντας `length($buf) == 0` μετά από ορισμένη επιστροφή.
- **Σήματα**: μια μπλοκαριστική `recv` που διακόπτεται από σήμα επιστρέφει `undef` με `$!` == `EINTR`. Επαναλάβετε ή εγκαταστήστε κατάλληλη `sigaction` με `SA_RESTART` αν αυτό δεν είναι αυτό που θέλετε.
- **Κλειστή υποδοχή**: η `recv` σε κλειστό ή ποτέ-ανοιχτό handle επιστρέφει `undef`, θέτει το `$!` σε `EBADF`, και εκπέμπει προειδοποίηση `recv() on closed socket` υπό `use warnings`.
- **Το προϋπάρχον περιεχόμενο του `SCALAR` καταστρέφεται**. Η `recv` δεν προσαρτά· αντικαθιστά. Χρησιμοποιήστε φρέσκο λεξιλογικό ή καθαρίστε το βαθμωτό πριν από κάθε κλήση.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `send` - η πλευρά εγγραφής της ίδιας υποδοχής· ίδιο λεξιλόγιο `FLAGS`, ίδια διάκριση χωρίς σύνδεση έναντι συνδεδεμένης
- `socket` - δημιουργία του handle `SOCKET` από το οποίο διαβάζει η `recv`
- `bind` - απαιτείται πριν την `recv` σε υποδοχή datagram πλευράς διακομιστή ώστε ο πυρήνας να γνωρίζει σε ποια θύρα να παραδώσει
- `connect` - όταν χρησιμοποιείται σε υποδοχή datagram, κάνει την `recv` να επιστρέφει `""` όπως σε υποδοχή ροής και φιλτράρει datagrams από άλλους ομότιμους
- `sysread` - ανάγνωση σε επίπεδο bytes για υποδοχές ροής όταν δεν χρειάζεστε τη διεύθυνση αποστολέα· ενσωματώνεται σε βρόχους βασισμένους σε `select`
- `Socket` - `sockaddr_in`, `inet_ntoa`, `MSG_PEEK`, `MSG_WAITALL`, και οι υπόλοιπες σταθερές που καταναλώνει η `recv`

Ροή ελέγχου

redo

Επανεκκινεί την τρέχουσα επανάληψη ενός βρόχου χωρίς εκ νέου έλεγχο της συνθήκης και χωρίς εκτέλεση του μπλοκ `continue`.

Το `redo` μεταπηδά πίσω στην κορυφή του σώματος του βρόχου και το εκτελεί ξανά πάνω στην *ίδια* είσοδο - η υπό συνθήκη που φυλάει τον βρόχο δεν επαναποτιμάται, το `$_` δεν προχωρά από το `while (<FH>)`, ο επαναλήπτης ενός `foreach` δεν προχωρά, και κάθε μπλοκ `continue { }` συνδεδεμένο με τον βρόχο παρακάμπτεται. Το τρέχον πέρασμα συμβαίνει ξανά από την αρχή.

Σύνοψη

```
redo
redo LABEL
redo EXPR
```

Τι επιστρέφεται

Το `redo` δεν επιστρέφει τιμή. Είναι τελεστής ελέγχου ροής: μεταφέρει τον έλεγχο και ποτέ δεν συνεχίζει στην επόμενη εντολή. Μην προσπαθήσετε να το χρησιμοποιήσετε ως τιμή έκφρασης - ειδικότερα, δεν μπορεί να χρησιμοποιηθεί για να επιστρέψει τιμή από `eval { }`, `sub { }` ή `do { }`.

Ποιον βρόχο στοχεύει

Χωρίς όρισμα, το `redo` αναφέρεται στον εσώτατο περικλείοντα βρόχο. Με `LABEL`, επανεκκινεί τον βρόχο που φέρει αυτή την ετικέτα, επιτρέποντάς σας να φτάσετε εκτός εμφωλευμένων βρόχων:

```
OUTER: while (...) {
    while (...) {
        redo OUTER;      # restart the while (...) above
    }
}
```

Το `redo EXPR` (προστέθηκε στην Perl 5.18) υπολογίζει την ετικέτα κατά τον χρόνο εκτέλεσης. Η έκφραση πρέπει να αποτιμάται σε συμβολοσειρά που κατονομάζει μια ετικέτα εντός εμβέλειας τη δεδομένη στιγμή:

```
my $target = "OUTER";
redo $target;
```

Ένα σκέτο μπλοκ μετράει ως βρόχος που εκτελείται μία φορά, οπότε το `redo` μέσα σε μπλοκ το μετατρέπει σε επαναληπτική κατασκευή:

```
{
    my $line = <STDIN>;
    redo if $line =~ /^\s*$/;   # skip blank lines, re-read
    print $line;
}
```

`redo` έναντι `next` έναντι `last`

Και τα τρία δέχονται προαιρετικό `LABEL` και τα τρία επηρεάζουν τον έλεγχο βρόχου, αλλά διαφέρουν σε αυτό που παρακάμπτουν:

- `last` - έξοδος από τον βρόχο τελείως. Το μπλοκ `continue` **δεν** εκτελείται.
- `next` - προχώρα στην επόμενη επανάληψη. Το μπλοκ `continue` **εκτελείται**, μετά ελέγχεται ξανά η συνθήκη του βρόχου.
- `redo` - επανάληψη της *ίδιας* επανάληψης. Το μπλοκ `continue` **δεν** εκτελείται, και η συνθήκη του βρόχου **δεν** ελέγχεται ξανά.

Η διαφορά στο μπλοκ `continue` είναι αυτή που πιάνει απροετοίμαστους τους χρήστες. Αν ένας βρόχος χρησιμοποιεί `continue { }` για αύξηση μετρητή ή προώθηση δρομέα, το `redo` δεν θα τον προωθήσει - αυτό είναι όλο το νόημα, αλλά είναι επίσης και όλη η παγίδα.

Παραδείγματα

Επανειλημμένη ερώτηση μέχρι ο χρήστης να δώσει έγκυρη είσοδο. Το μπλοκ `continue` παρακάμπτεται, οπότε το `$attempts` μετρά μόνο *ολοκληρωμένες* επαναλήψεις, κάτι που είναι συνήθως αυτό που θέλετε:

```
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 {
    $attempts++;
}
```

Επανεκκίνηση της επεξεργασίας της τρέχουσας γραμμής αφού την μπαλώσετε - το κλασικό μοτίβο «πες ψέματα στον εαυτό σου για το τι μόλις διαβάστηκε», εδώ αφαιρώντας block comments τύπου Pascal που μπορεί να εκτείνονται σε πολλαπλές γραμμές εισόδου:

```
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;
}
```

Στοχεύστε εξωτερικό βρόχο με ετικέτα. Το `redo INNER` θα επανεκκινούσε τον εσωτερικό βρόχο στο τρέχον `$x`· το `redo OUTER` επανεκκινεί τον εξωτερικό βρόχο στο τρέχον `$x` *χωρίς* να προωθήσει τον επαναλήπτη του `foreach`:

```
OUTER: foreach my $x (@xs) {
    INNER: foreach my $y (@ys) {
        redo OUTER if should_restart($x, $y);
    }
}
```

Μετατρέψτε ένα σκέτο μπλοκ σε βρόχο επανάληψης. Χωρίς το `redo` αυτό το μπλοκ εκτελείται ακριβώς μία φορά· με το `redo` εκτελείται μέχρι η ανάκτηση να πετύχει:

```
my $tries = 0;
{
    my $ok = try_fetch();
    if (!$ok && ++$tries < 3) {
        sleep 1;
        redo;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    }
}
```

Οριακές περιπτώσεις

- **Το `continue` δεν εκτελείται.** Αυτή είναι η καθοριστική διαφορά από το `next`. Αν ο βρόχος σας βασίζεται στο `continue { }` για μετρητές, δρομείς, ή καθαρισμό, το `redo` θα τα παρακάμψει όλα αυτά. Είτε μετακινήστε τη δουλειά στο σώμα είτε χρησιμοποιήστε `next`.
- **Η συνθήκη του βρόχου δεν ελέγχεται ξανά.** Το `while (cond) { ... redo; }` θα συνεχίσει να επαναλαμβάνεται ακόμα και αν η cond θα ήταν τώρα ψευδής. Είστε υπεύθυνοι να εξασφαλίσετε ότι η διαδρομή του `redo` τελικά βγαίνει μέσω `last`, `return`, ή υπό συνθήκη φυσιολογικής ολοκλήρωσης.
- **Εύκολος ατέρμων βρόχος.** Το `redo` χωρίς αλλαγή κατάστασης είναι σφιχτός ατέρμων βρόχος - τυπικά δεν είναι αυτό που θέλετε. Συνδυάζετε πάντα το `redo` με μια συνθήκη που τελικά γίνεται ψευδής, ή με μετρητή επανάληψης που πυροδοτεί `last`.
- **Το `foreach` δεν προχωρά.** Το `foreach my $x (@xs) { redo }` κρατά το `$x` καρφωμένο στο τρέχον στοιχείο και επαναλαμβάνεται σε αυτό για πάντα. Ο επαναλήπτης προχωρά μόνο σε φυσιολογική ολοκλήρωση ή `next`.
- **Το `while (<FH>)` δεν διαβάζει την επόμενη γραμμή.** Το `$_` διατηρεί την τρέχουσα γραμμή κατά τη διάρκεια του `redo`, το οποίο είναι ακριβώς ο λόγος για τον οποίο δουλεύει ο stripper σχολίων Pascal παραπάνω.
- **Το `redo` εκτός `grep` / `map` δεν υποστηρίζεται.** Το upstream το τεκμηριώνει ως απροσδιόριστο· μη χρησιμοποιείτε `redo` για να βγείτε ή να επανεκκινήσετε ένα `grep BLOCK LIST` ή `map BLOCK LIST`.
- **Το `redo` εκτός `eval`, `sub`, ή `do` δεν είναι `return`.** Πραγματοποιεί έλεγχο ροής· δεν υπάρχει τιμή προς επιστροφή. Ένα `redo` που θα έπρεπε να διασχίσει τέτοιο όριο για να φτάσει σε βρόχο είναι σφάλμα χρόνου εκτέλεσης («Can't `redo` outside a loop block»).
- **Ιδιορρυθμία προτεραιότητας.** Σε αντίθεση με τους περισσότερους κατονομασμένους τελεστές, το `redo` έχει την προτεραιότητα της ανάθεσης και εξαιρείται από τον κανόνα «μοιάζει-με-συνάρτηση». Το `redo ("foo"). "bar"` περνά ολόκληρη τη συνένωση ως όρισμα στο `redo` - οι παρενθέσεις δεν απομονώνουν το `"foo"`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `last` - έξοδος από τον βρόχο τελείως· το μπλοκ `continue` δεν εκτελείται, ίδια συμπεριφορά με το `redo`

- `next` - προχώρα στην επόμενη επανάληψη· το μπλοκ `continue` **εκτελείται** και η συνθήκη του βρόχου ελέγχεται ξανά
- `return` - επιστροφή από την περικλείουσα υπορουτίνα, όχι μόνο από τον βρόχο· σε αντίθεση με το `redo`, παράγει τιμή
- `continue` - το μπλοκ που το `redo` παρακάμπτει αλλά το `next` εκτελεί· ο πυρήνας της τριμερούς διάκρισης

Κλάσεις και αντικειμενοστρέφεια

ref

Επιστρέφει μια συμβολοσειρά που περιγράφει σε τι δείχνει μια αναφορά.

Η `ref` εξετάζει τον τελεστέο της και επιστρέφει μια συμβολοσειρά που προσδιορίζει το είδος του αναφερόμενου. Για blessed αναφορά επιστρέφει το όνομα της κλάσης· για μη-blessed αναφορά επιστρέφει μία από τις ενσωματωμένες συμβολοσειρές τύπου (`SCALAR`, `ARRAY`, `HASH`, `CODE`, `GLOB`, `LVALUE`, `IO`, `FORMAT`, `REF`, `VSTRING`, `REGEXP`). Για οτιδήποτε δεν είναι αναφορά επιστρέφει την κενή συμβολοσειρά. Χωρίς όρισμα, η `ref` λειτουργεί επί της `$_`.

Σύνοψη

```
ref EXPR
ref
```

Τι επιστρέφεται

Μια συμβολοσειρά. Τρεις αμοιβαία αποκλειόμενες κατηγορίες:

- **Blessed αναφορά** - το όνομα της κλάσης στην οποία έχει γίνει bless ο αναφερόμενος. Για παράδειγμα, μια αναφορά `bless {}, 'Dog'` επιστρέφει `"Dog"`. Οι υποκλάσεις επιστρέφουν το δικό τους όνομα, όχι του γονέα: μια αναφορά που έγινε bless στην Puppy (η οποία είναι `ISA` της Dog) επιστρέφει `"Puppy"`. Δείτε `bless`.
- **Μη-blessed αναφορά** - μία από τις ενσωματωμένες συμβολοσειρές τύπου που παρατίθενται παρακάτω, που προσδιορίζει τον φυσικό τύπο του αναφερόμενου.
- **Μη-αναφορά** - η κενή συμβολοσειρά `""`.

Η κενή συμβολοσειρά είναι η **μοναδική** ψευδής τιμή που επιστρέφει ποτέ η `ref`, οπότε το `if (ref $x)` είναι ασφαλής έλεγχος «είναι αυτό αναφορά;». Δείτε την ενότητα οριακών περιπτώσεων για ποιον λόγο το `if (ref $x eq 'HASH')` *δεν* είναι ασφαλής έλεγχος «είναι αυτό αναφορά hash;».

Ενσωματωμένες συμβολοσειρές τύπου

Επιστρέφονται για μη-blessed αναφορές σύμφωνα με τον φυσικό αναφερόμενο:

Συμβολοσειρά	Αναφερόμενος
SCALAR	αναφορά σε κανονικό βαθμωτό
ARRAY	αναφορά σε πίνακα
HASH	αναφορά σε hash
CODE	αναφορά σε υπορουτίνα
GLOB	αναφορά σε typeglob
LVALUE	αναφορά σε εκχωρήσιμη έκφραση
IO	αναφορά σε handle I/O
FORMAT	αναφορά σε format
REF	αναφορά σε άλλη αναφορά
VSTRING	αναφορά σε βαθμωτό τύπου v-string
REGEXP	αναφορά σε ωμό (μη-blessed) regexp

Ένα μεταγλωττισμένο regex από `qr//` είναι *blessed* στην κλάση `Regexp` κατά τη δημιουργία, οπότε η `ref qr/.../` συνήθως επιστρέφει `"Regexp"`, όχι `"REGEXP"`. Η σκέτη συμβολοσειρά `REGEXP` εμφανίζεται μόνο για μη-blessed regex αναφερόμενους, που είναι σπάνιοι έξω από τα εσωτερικά.

Παραδείγματα

Βασικές δοκιμές τύπου:

```
ref \1              # "SCALAR"
ref []              # "ARRAY"
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           # ""
```

Οι blessed αναφορές επιστρέφουν το όνομα της κλάσης:

```
my $obj = bless {}, 'My::Widget';
ref $obj;                        # "My::Widget"
```

Προστατευτείτε πριν την αποαναφορά:

```
sub stringify {
    my ($v) = @_;
    return "[undef]"           if !defined $v;
    return "[ref: " . ref($v) . "]" if ref $v;
    return $v;
}
```

Η μορφή χωρίς όρισμα διαβάζει την `$_`:

```
for my $item (@things) {
    print "skipping non-ref\n" and next unless ref;
    ...
}
```

Επιλέγοντας τον σωστό έλεγχο

Η `ref` είναι το παλαιότερο πρωτογενές στοιχείο εσωτερικής σκόπευσης της γλώσσας και συνδυάζει τρεις διαφορετικές ερωτήσεις. Ο σύγχρονος κώδικας πρέπει να επιλέγει τον τελεστή που ταιριάζει με την ερώτηση που τίθεται.

Είναι αυτό καθόλου αναφορά;

```
if (ref $x) { ... }            # empty string is the only false␣
↪return
```

Είναι αυτό αντικείμενο μιας δεδομένης κλάσης (ή παραγόμενο από αυτήν); Χρησιμοποιήστε τον τελεστή `isa` (ή τη μέθοδο `UNIVERSAL::isa`) - διατρέχει το `@ISA` και τιμά την υποκλασιοποίηση, κάτι που μια κυριολεκτική σύγκριση με `ref` δεν κάνει:

```
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
```

Ποιος είναι ο υποκείμενος φυσικός τύπος, ανεξάρτητα από το blessing; Χρησιμοποιήστε την `Scalar::Util::reftype` - αγνοεί την κλάση και επιστρέφει την ενσωματωμένη συμβολοσειρά τύπου ακόμη και για blessed αναφορές:

```
use Scalar::Util qw(reftype blessed);
reftype($obj);                  # "HASH" if $obj is blessed hash ref
blessed($obj);                  # class name, or undef if not␣
↪blessed
```

Οριακές περιπτώσεις

- **Ποτέ μη χρησιμοποιείτε τη `ref` ως τιμή αλήθειας του αναφερόμενου.** Μια αναφορά που έγινε bless σε κλάση με κυριολεκτικό όνομα `0` κάνει την `ref $obj` να επιστρέφει τη συμβολοσειρά `"0"`, που είναι ψευδής. Είναι σπάνιο εκ κατασκευής, αλλά η παγίδα είναι παλιά και γνωστή. Συγκρίνετε με την κενή συμβολοσειρά στη θέση της: `if (ref $x ne "")`.
- **Συγκρούσεις ονομάτων κλάσεων με ενσωματωμένες συμβολοσειρές τύπου.** Τίποτα δεν σας εμποδίζει να γράψετε `package HASH; sub new { bless {}, shift }`. Τότε η `ref(HASH->new)` επιστρέφει `"HASH"` - διακρίνεται από μη-blessed αναφορά hash μόνο με τη `ref`. Χρησιμοποιήστε την `Scalar::Util::blessed` για αποσαφήνιση, ή την `reftype` για να θέσετε άμεσα την ερώτηση φυσικού τύπου.
- **Η `ref` σε `qr//` επιστρέφει `"Regexp"`, όχι `"REGEXP"`.** Τα αντικείμενα `qr//` είναι blessed κατά την κατασκευή· η κεφαλαία μορφή παράγεται μόνο για την (ασυνήθιστη) μη-blessed αναφορά regex.

- **Μεταβλητές με tie**: η `ref` βλέπει μέσα από το tie στην υποκείμενη αναφορά. Χρησιμοποιήστε την `tied` για να ανακτήσετε το αντικείμενο που υποστηρίζει το tie.
- **Οι ασθενείς αναφορές** εξακολουθούν να είναι αναφορές: η `ref` συμπεριφέρεται ταυτόσημα για ασθενείς και ισχυρές αναφορές. Χρησιμοποιήστε την `Scalar::Util::isweak` για να ανιχνεύσετε την ασθένεια.
- **Η υπερφορτωμένη μετατροπή σε συμβολοσειρά δεν επηρεάζει τη `ref`.** Η τιμή επιστροφής της `ref` παράγεται από το runtime, όχι από την υπερφόρτωση `""` του αντικειμένου.
- **Η `ref EXPR` έχει πολύ υψηλή προτεραιότητα** - αναλύεται ως ονομασμένος μοναδιαίος τελεστής. Βάλτε παρενθέσεις όταν το όρισμα είναι οτιδήποτε πιο σύνθετο από μια απλή μεταβλητή:

```
ref $a || $b;          # parsed as (ref $a) || $b
ref($a || $b);         # what you probably meant
```

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `bless` - εγκαθιστά ένα όνομα κλάσης σε μια αναφορά· το όνομα που εγκαθίσταται εδώ είναι ακριβώς αυτό που επιστρέφει στη συνέχεια η `ref`
- `isa` - έλεγχος συμμετοχής σε κλάση που τιμά την κληρονομικότητα του `@ISA`· προτιμήστε την έναντι του `ref(...) eq 'ClassName'` για αντικειμενοστρεφή διανομή
- `tied` - ανακτά το αντικείμενο που υλοποιεί μια μεταβλητή με tie· η `ref` σε ένα βαθμωτό με tie ακολουθεί το tie μέχρι τον αναφερόμενο
- `Scalar::Util::reftype` - συμβολοσειρά φυσικού τύπου για αναφορά, αγνοώντας το blessing
- `Scalar::Util::blessed` - όνομα κλάσης αν είναι blessed, `undef` διαφορετικά· το μονοσήμαντο αντίστοιχο της `ref` για αντικειμενοστρεφή κώδικα
- `perlref`, `perlobj` - εγχειρίδια αναφοράς για αναφορές και μοντέλο αντικειμένων

Filehandles, αρχεία, κατάλογοι

rename

Αλλάζει το όνομα ενός αρχείου.

Η `rename` ζητά από το λειτουργικό σύστημα να επαναδέσει τη διαδρομή `OLDNAME` στο `NEWNAME`. Σε ένα σύστημα αρχείων POSIX αυτή είναι μια μοναδική ατομική λειτουργία καταλόγου: αφού επιστρέψει με επιτυχία η κλήση, το `NEWNAME` παραπέμπει στο αρχείο που πριν ήταν προσπελάσιμο ως `OLDNAME`, και το `OLDNAME` δεν υπάρχει πλέον. Είναι η ίδια πρωτογενής λειτουργία που χρησιμοποιεί η εντολή κελύφους `mv` όταν πηγή και προορισμός βρίσκονται στο ίδιο σύστημα αρχείων.

Σύνοψη

```
rename OLDNAME, NEWNAME
```

Τι επιστρέφεται

1 σε επιτυχία, `0` σε αποτυχία με την `$!` ορισμένη στο `errno` που επέστρεψε η υποκείμενη κλήση συστήματος `rename(2)`. Ελέγχετε πάντα την τιμή επιστροφής - μια σιωπηρή αποτυχία εδώ αφήνει το σύστημα αρχείων σε κατάσταση που το υπόλοιπο πρόγραμμα δεν αναμένει:

```
rename $tmp, $final
    or die "rename $tmp -> $final failed: $!";
```

Ατομικότητα και ο κανόνας του ίδιου συστήματος αρχείων

Η `rename` είναι ατομική **μόνο όταν τα `OLDNAME` και `NEWNAME` επιλύονται στο ίδιο σύστημα αρχείων**. Μέσα σε αυτό το σύστημα αρχείων, καμία ενδιάμεση κατάσταση δεν είναι ποτέ ορατή: μια άλλη διεργασία που διαβάζει τον κατάλογο βλέπει είτε το παλιό όνομα είτε το νέο, ποτέ και τα δύο και ποτέ κανένα από τα δύο. Γι" αυτό η κανονική ιδιωματική φράση «γράψε ένα αρχείο με ασφάλεια» γράφει σε προσωρινό αρχείο στον ίδιο κατάλογο και μετά το μετονομάζει στη θέση του:

```
open my $fh, ">", "$path.tmp" or die "open: $!";
print $fh $data;
close $fh or die "close: $!";
rename "$path.tmp", $path
    or die "rename: $!";
```

Διασυστηματικά η κλήση αποτυγχάνει με `EXDEV` (`"Invalid cross-device link"`) και τίποτα δεν μετακινείται. Η εντολή συστήματος `mv` καλύπτει αυτό υποχωρώντας σε copy-then-unlink· η `rename` όχι. Για φορητή διασυσκευική μετακίνηση, χρησιμοποιήστε `move` από το `File::Copy`, που εκτελεί η ίδια την υποχώρηση αντιγραφή-και-unlink.

Παραδείγματα

Μετονομασία μέσα σε κατάλογο:

```
rename "draft.txt", "final.txt"
    or die "rename: $!";
```

Ατομική αντικατάσταση αρχείου - γράψτε σε αδελφικό προσωρινό αρχείο και κατόπιν μετονομάστε:

```
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: $!";
```

Ανιχνεύστε την διασυσκευική περίπτωση και υποχωρήστε σε `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: $!";
    }
}
```

Μετονομάστε κάθε `.log` σε έναν κατάλογο σε `.log.old`:

```
for my $f (glob "*.log") {
    rename $f, "$f.old"
        or warn "rename $f: $!";
}
```

Οριακές περιπτώσεις

- **Το `NEWNAME` υπάρχει ήδη**: συντρίβεται σιωπηρά. Η `rename` **δεν** ρωτά, δεν προειδοποιεί και δεν αρνείται. Αν ο στόχος είναι κανονικό αρχείο, αποσυνδέεται ως μέρος της λειτουργίας· αν είναι κατάλογος η κλήση αποτυγχάνει με `ENOTDIR` ή `EISDIR` ανάλογα με το αν το `OLDNAME` είναι κατάλογος. Για να πάρετε σημασιολογία «αρνήσου να επικαλύψεις», ελέγξτε πρώτα με `-e` (και αποδεχθείτε ότι ο έλεγχος είναι επιρρεπής σε κούρσα - δεν υπάρχει φορητή `renameat2(RENAME_NOREPLACE)` εκτιθέμενη μέσω της ίδιας της `rename`).
- **Διασυστηματική (`EXDEV`)**: αποτυγχάνει, αφήνει αμφότερες τις διαδρομές άθικτες. Χρησιμοποιήστε `File::Copy::move` όταν ο προορισμός μπορεί να βρίσκεται σε άλλο mount.
- **Ανοιχτά filehandles**: στο Linux (και στο POSIX γενικά) η μετονομασία αρχείου που είναι τη στιγμή εκείνη ανοιχτό είναι εντάξει. Το ανοιχτό filehandle εξακολουθεί να δείχνει στο ίδιο inode· οι επόμενες αναγνώσεις και εγγραφές μέσω του continue να λειτουργούν. Νέα ανοίγματα χρησιμοποιούν το νέο όνομα.
- **Κατάλογοι**: η μετονομασία καταλόγου επιτρέπεται, αλλά μόνο όταν το `NEWNAME` δεν υπάρχει ή είναι κενός κατάλογος. Η μετακίνηση καταλόγου σε υποκατάλογο του εαυτού του αποτυγχάνει με `EINVAL`.
- **Δικαιώματα**: η `rename` χρειάζεται δικαίωμα εγγραφής + εκτέλεσης τόσο στον γονικό κατάλογο της πηγής όσο και του προορισμού, όχι στο ίδιο το αρχείο. Η αποτυχία εμφανίζεται ως `EACCES` ή `EPERM`.
- **Κατάλογοι με sticky bit** (`/tmp`): αν ο γονικός του προορισμού έχει το sticky bit ορισμένο, πρέπει να είστε ιδιοκτήτης είτε του αρχείου πηγής είτε του γονικού του προορισμού· αλλιώς η κλήση αποτυγχάνει με `EPERM` ακόμη και όταν ο κατάλογος είναι εγγράψιμος από όλους.
- **Συμβολικοί σύνδεσμοι**: η `rename` δρα στον ίδιο τον σύνδεσμο, ποτέ στον στόχο του. Η `rename "link", "other"` μετονομάζει τον symlink· το αρχείο στο οποίο

δείχνει μένει ανέγγιχτο.

- **Τελική πλαγιοκάθετος στο `OLDNAME`**: πολλοί πυρήνες το απορρίπτουν με `ENOTDIR` όταν η πηγή δεν είναι κατάλογος. Μη βασίζεστε στο ότι η συμπεριφορά της τελικής πλαγιοκαθέτου είναι φορητή.
- **Μετονομασία ίδιας διαδρομής**: η `rename $p, $p` πετυχαίνει και δεν κάνει τίποτα (συμπεριφορά που απαιτεί το POSIX) εφόσον το `$p` υπάρχει.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `unlink` - αφαιρεί αρχείο κατ” όνομα· το αντίστοιχο αποδόμησης για τη δημιουργία ενός μέσω `rename` στη θέση του
- `link` - δημιουργεί επιπλέον όνομα hard-link για ένα υπάρχον αρχείο χωρίς να αφαιρέσει το αρχικό
- `symlink` - δημιουργεί συμβολικό σύνδεσμο· σημειώστε ότι η `rename` σε symlink μετακινεί τον σύνδεσμο, όχι τον στόχο του
- `File::Copy` - φορητές `move` και `copy` που χειρίζονται τη διασυστηματική (`EXDEV`) περίπτωση που η `rename` δεν μπορεί
- `$!` - `errno` μετά από αποτυχημένη `rename`· συγκρίνετε με σταθερές από το `Errno` (`EXDEV`, `EACCES`, `ENOENT`, ...)

Αρθρώματα

require

Φορτώνει ένα αρχείο πηγαίου κώδικα Perl σε χρόνο εκτέλεσης, ή απαιτεί μια ελάχιστη έκδοση Perl.

Η `require` έχει τρεις μορφές που κάνουν αρκετά διαφορετικά πράγματα. Με αριθμό έκδοσης επιβεβαιώνει ότι ο τρέχων διερμηνέας είναι αρκετά νέος. Με bareword φορτώνει ένα άρθρωμα μετατρέποντας το `Foo::Bar` σε `Foo/Bar.pm` και αναζητώντας στο `@INC`. Με συμβολοσειρά κατά τον χρόνο εκτέλεσης ή με `$_` φορτώνει ένα όνομα αρχείου ως έχει. Και οι τρεις είναι λειτουργίες **χρόνου εκτέλεσης** - αντίθετα με την `use`, που είναι `require` + `import` τυλιγμένα σε ένα μπλοκ `BEGIN` κατά τη μεταγλώττιση.

Σύνοψη

```
require VERSION
require EXPR
require
```

Τι επιστρέφεται

Για τη μορφή ελέγχου έκδοσης: 1 αν η τρέχουσα Perl είναι αρκετά νέα· διαφορετικά εκτοξεύεται εξαίρεση (`Perl vX.Y.Z required--this is only ...`). Δεν υπάρχει τίποτα προς έλεγχο.

Για τις μορφές φόρτωσης αρχείων: η αληθής τιμή που επιστρέφει η τελευταία δήλωση του φορτωμένου αρχείου, ή 1 αν το αρχείο ενεργοποιεί το χαρακτηριστικό `module_true` (προεπιλεγμένο υπό `use v5.38` και μεταγενέστερα). Μια δεύτερη `require` της ίδιας διαδρομής επιστρέφει 1 χωρίς να επανεκτελέσει το αρχείο. Η αποτυχία εκτοξεύει εξαίρεση - είτε το σφάλμα μεταγλώττισης/χρόνου εκτέλεσης από το αρχείο, `Can't locate Foo/Bar.pm in @INC ...`, ή `Foo/Bar.pm did not return true value`.

Τυλίξτε σε `eval` όταν θέλετε να ελέγξετε αν ένα άρθρωμα είναι φορτώσιμο:

```
eval { require Some::Optional::Module };
if ($@) { ... }                    # module missing or broken
```

Καθολική κατάσταση που επηρεάζει

- `@INC` - η διαδρομή αναζήτησης για τις μορφές bareword και σχετικού ονόματος αρχείου. Οι εγγραφές μπορεί να είναι συμβολοσειρές καταλόγων, coderefs, αναφορές πινάκων, ή blessed αντικείμενα (δείτε *Γάντζοι* `@INC` παρακάτω).
- `%INC` - αντιστοιχίζει το ζητούμενο όνομα αρχείου (`Foo/Bar.pm`) στην απόλυτη διαδρομή που φορτώθηκε. Η `require` τη συμβουλεύεται για να αποφύγει διπλή φόρτωση και τη θέτει σε επιτυχία. Εγγραφή με τιμή `undef` σημαδεύει αρχείο του οποίου η μεταγλώττιση απέτυχε.
- `$_` - χρησιμοποιείται ως όνομα αρχείου όταν η `require` καλείται χωρίς όρισμα.
- `${^HOOK}{require__before}` και `${^HOOK}{require__after}` - προαιρετικά coderefs που εκτελούνται γύρω από κάθε `require`, διαθέσιμα για περιτυλίγματα ιχνηλασίας και οργανολογίας.

Οι τρεις μορφές

`require VERSION` - έλεγχος ελάχιστης Perl

Το `VERSION` είναι είτε v-string όπως `v5.38.0` (συγκρίνεται έναντι της `$^V`) ή αριθμητικό κυριολεκτικό όπως `5.038000` (συγκρίνεται έναντι της `$]`). Αν ο τρέχων διερμηνέας είναι παλαιότερος, η `require` εκτοξεύει εξαίρεση.

```
require v5.38;        # preferred
require 5.038;        # numeric; same effect
require 5.038_000;    # same; pre-5.6 syntax
```

Αυτός είναι έλεγχος **χρόνου εκτέλεσης**. Αν θέλετε ο έλεγχος να γίνει κατά τη μεταγλώττιση (ώστε το σενάριο να μην εκκινεί ποτέ σε παλιά Perl), χρησιμοποιήστε αντί αυτής `use VERSION`:

```
use v5.38;            # compile-time equivalent
```

`require` EXPR όπου το EXPR είναι bareword - άρθρωμα κατ" όνομα

Ένα bareword ερμηνεύεται ως όνομα αρθρώματος. Η `require` αντικαθιστά κάθε `::` με `/` και προσαρτά `.pm`, και στη συνέχεια αναζητά στο `@INC`:

```
require Foo::Bar;     # looks for Foo/Bar.pm in @INC
```

Αυτή η μορφή επίσης **αυτο-δημιουργεί τη stash** κατά τη μεταγλώττιση - το hash πακέτου `Foo::Bar::` εμφανίζεται ακόμη και αν η φόρτωση αποτύχει μετέπειτα. Δεν εισάγονται σύμβολα· η `require` μόνο φορτώνει κώδικα.

Πριν δοκιμάσει το `Foo/Bar.pm`, η `require` ψάχνει για το `Foo/Bar.pmc` στον ίδιο κατάλογο και το προτιμά αν το βρει. Ο γάντζος `.pmc` επιτρέπει σε εργαλεία build να αποστέλλουν προμεταγλωττισμένα συνοδευτικά αρχεία δίπλα στις πηγές.

`require` EXPR όπου το EXPR είναι συμβολοσειρά - αρχείο κατά διαδρομή

Οποιαδήποτε έκφραση που δεν είναι bareword αντιμετωπίζεται ως κυριολεκτικό όνομα αρχείου. Καμία μετατροπή `::`, καμία προσάρτηση `.pm`, καμία αυτο-δημιουργία stash:

```
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"
```

Η τελευταία γραμμή είναι η κλασική παγίδα: μόλις το όνομα του αρθρώματος βρίσκεται σε βαθμωτό, είναι συμβολοσειρά, όχι bareword. Για να φορτώσετε άρθρωμα του οποίου το όνομα υπολογίζεται, είτε κάντε τη μετατροπή εσείς, είτε περάστε από `eval`:

```
(my $file = "$class.pm") =~ s{::}{/}g;
require $file;

eval "require $class; 1" or die $@;
```

Αν το EXPR παραλειφθεί εντελώς, χρησιμοποιείται η `$_`.

Η προστασία του `%INC` - τα φορτωμένα αρχεία παραμένουν φορτωμένα

Πριν αναζητήσει στο `@INC`, η `require` συμβουλεύεται το `%INC`. Αν το ζητούμενο όνομα αρχείου είναι παρόν και έχει αληθή τιμή, η `require` επιστρέφει αμέσως 1 χωρίς να ξαναδιαβάσει το αρχείο. Αν η τιμή είναι `undef`, μια προηγούμενη φόρτωση απέτυχε και η `require` εκτοξεύει `Compilation failed in require` - η Perl αρνείται να ξαναδοκιμάσει ένα αρχείο που έχει ήδη εκραγεί μία φορά σε αυτόν τον διερμηνέα.

Αυτό σημαίνει ότι ο κώδικας ανώτατου επιπέδου ενός αρθρώματος εκτελείται **ακριβώς μία φορά** ανά διερμηνέα, ανεξάρτητα από το από πόσα μέρη γίνεται `require`. Η αρχικοποίηση στο σώμα του αρχείου του αρθρώματος είναι μονή λειτουργία· μη βασίζεστε στο ότι θα τρέχει για κάθε κλήση `require` σε κάθε καλούντα.

Ένας γάντζος είναι ελεύθερος να θέσει ο ίδιος το `%INC`· αν δεν το κάνει, η `require` καταγράφει τον γάντζο ως «πηγή» της φόρτωσης.

Η σύμβαση του `1;`

Ιστορικά, κάθε φορτώσιμο αρχείο έπρεπε να επιστρέφει **αληθή** τιμή ως τελευταία του έκφραση, διαφορετικά η `require` εκτόξευε `filename did not return true value`. Ο καθιερωμένος τρόπος για να το εγγυηθεί κανείς είναι ένα τελικό `1;`:

```
package My::Thing;
# ... module code ...
1;                                  # mandatory, pre-5.38
```

Από την Perl 5.38, το χαρακτηριστικό `module_true` - προεπιλεγμένα ενεργοποιημένο υπό `use v5.38` ή μεταγενέστερα - κάνει κάθε αρχείο που γίνεται `require` να επιστρέφει σιωπηρά αληθές σε επιτυχημένη μεταγλώττιση. Τα αρχεία που το ενεργοποιούν δεν χρειάζονται πλέον `1;`. Όσα όχι, εξακολουθούν να το χρειάζονται.

Αυτό επηρεάζει μόνο τη μονάδα μεταγλώττισης που ενεργοποίησε το χαρακτηριστικό· παλαιά αρθρώματα που φορτώνονται από σύγχρονο καλούντα εξακολουθούν να χρειάζονται το δικό τους `1;`.

Όταν το φορτωμένο αρχείο επιστρέφει τιμή και το `module_true` είναι ανενεργό, αυτή η τιμή είναι αυτό που επιστρέφει η `require` - αλλά **μόνο στην πρώτη φόρτωση**. Επόμενες κλήσεις `require` του ίδιου αρχείου επιστρέφουν 1 από την προσωρινή μνήμη του `%INC`. Για να αποτυπώσετε αξιόπιστα την τιμή επιστροφής ενός αρχείου, χρησιμοποιήστε `do`:

```
my $config = do "config.pl"
    or die "config.pl: ", $@ || $!;
```

Γάντζοι `@INC` - coderefs, αναφορές πινάκων, και αντικείμενα

Οι εγγραφές στο `@INC` δεν χρειάζεται να είναι συμβολοσειρές καταλόγων. Μια αναφορά αποθηκευμένη στο `@INC` καλείται **γάντζος** και έχει την ευκαιρία να ικανοποιήσει την `require`.

Αναφορά υπορουτίνας

Καλείται με δύο ορίσματα: το ίδιο το coderef και το όνομα αρχείου (`Foo/Bar.pm`). Πρέπει να επιστρέφει είτε τίποτα (που σημαίνει «δεν μπορώ να το χειριστώ αυτό· δοκίμασε την επόμενη εγγραφή») είτε μέχρι τέσσερις τιμές:

1. Μια βαθμωτή αναφορά σε κείμενο πηγαίου κώδικα που προτάσσεται στο αρχείο.
2. Ένα filehandle από το οποίο διαβάζεται ο υπόλοιπος πηγαίος κώδικας.
3. Μια υπορουτίνα γεννήτορα ή φίλτρου (δείτε το `perldoc perlfunc` για το ακριβές πρωτόκολλο).
4. Προαιρετική κατάσταση που μεταβιβάζεται πίσω στον γεννήτορα.

```
push @INC, sub {
    my ($self, $filename) = @_;
    return unless $filename eq 'Virtual/Module.pm';
    open my $fh, '<', \"package Virtual::Module; 1;";
    return $fh;
};
```

Αναφορά πίνακα

Το πρώτο στοιχείο είναι coderef ή αντικείμενο όπως παραπάνω· το υπόλοιπο είναι κατάσταση που ο γάντζος μπορεί να διαβάζει σε κάθε κλήση. Όταν το πρώτο στοιχείο είναι αντικείμενο με μέθοδο `INC` ή `INCDIR`, η μέθοδος λαμβάνει την αναφορά πίνακα ως τρίτο της όρισμα.

```
push @INC, [\&my_loader, $db_handle, $cache];
```

Blessed αντικείμενο

Αν το αντικείμενο έχει μέθοδο `INC` ή `INCDIR`, η `require` καλεί αυτή τη μέθοδο· διαφορετικά, αν το αντικείμενο είναι επίσης coderef, καλείται σαν απλή αναφορά υπορουτίνας. Οι μέθοδοι `INC` πρέπει να δηλώνονται με το πλήρως κατονομασμένο όνομά τους (`sub Foo::INC { ... }`), διότι το ακατονόμαστο σύμβολο `INC` είναι σταθερά συνδεδεμένο με το πακέτο `main`.

Ένας γάντζος είναι ελεύθερος να ενημερώσει το `%INC` ώστε να καταγράψει τη διαδρομή που παρείχε· αν δεν το κάνει, η `require` αποθηκεύει την ίδια την αναφορά του γάντζου.

`require` έναντι `use`

```
use Foo::Bar qw(x y);

# is almost exactly:

BEGIN {
    require Foo::Bar;
    Foo::Bar->import(qw(x y));
}
```

Πτυχή	`use`	`require`
Πότε εκτελείται	Χρόνος μεταγλώττισης (μέσα σε `BEGIN`)	Χρόνος εκτέλεσης
Καλεί την `import`	Ναι	Όχι
Bareword `::` → `/`	Ναι	Ναι, όταν καλείται με bareword
Μορφή ελέγχου έκδοσης	`use v5.38`	`require v5.38`
Κατάλληλη για	Σταθερές εξαρτήσεις	Προαιρετικά / αργά συνδεόμενα αρθρώματα

Καταφύγετε στην `require` όταν το άρθρωμα είναι προαιρετικό, φορτώνεται από όνομα που υπολογίζεται, ή φορτώνεται μόνο μέσα σε διακλάδωση που ίσως δεν εκτελεστεί ποτέ. Καταφύγετε στην `use` για όλα τα υπόλοιπα.

Παραδείγματα

Ελάχιστη έκδοση Perl σε χρόνο εκτέλεσης:

```
require v5.38;
```

Φόρτωση αρθρώματος με bareword - αναζητά στο @INC, αυτο-δημιουργεί το Data::Dumper:: κατά τη μεταγλώττιση:

```
require Data::Dumper;
Data::Dumper->import;
print Data::Dumper::Dumper(\%ENV);
```

Προαιρετικό άρθρωμα - υποχωρήστε χαριτωμένα όταν λείπει:

```
my $have_json = eval { require JSON::PP; 1 };
if ($have_json) {
    print JSON::PP->new->encode($data);
}
```

Φόρτωση αρθρώματος του οποίου το όνομα υπολογίζεται σε χρόνο εκτέλεσης. Μετατρέψτε σε διαδρομή ονόματος αρχείου· το `require $class` θα έψαχνε για ένα κυριολεκτικό αρχείο με όνομα Foo::Bar:

```
my $driver = $ENV{DB_DRIVER} // 'SQLite';
my $class  = "DBD::$driver";
(my $file  = "$class.pm") =~ s{::}{/}g;
require $file;
```

Φόρτωση αποσπάσματος ρυθμίσεων κατά διαδρομή, διατηρώντας την τιμή επιστροφής του - σημειώστε την do, όχι την `require`, διότι θέλουμε την τιμή σε κάθε κλήση:

```
my $config = do '/etc/myapp/config.pl'
    or die "config load failed: ", $@ || $!;
```

Εγκατάσταση γάντζου @INC που εξυπηρετεί ένα μεμονωμένο εικονικό άρθρωμα από συμβολοσειρά στη μνήμη:

```
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;
```

Οριακές περιπτώσεις

- **Bareword έναντι συμβολοσειράς**: τα `require Foo::Bar` και `require "Foo::Bar"` **δεν** είναι το ίδιο. Η μορφή bareword μετατρέπεται σε `Foo/Bar.pm`· η μορφή συμβολοσειράς αναζητά αρχείο με κυριολεκτικό όνομα `Foo::Bar`.
- **Υπολογιζόμενα ονόματα κλάσεων**: το `require $class` όπου η `$class` κρατά `'Foo::Bar'` αναζητά αρχείο με όνομα `Foo::Bar`, όχι `Foo/Bar.pm`. Κάντε εσείς τη μετατροπή του ονόματος, ή χρησιμοποιήστε `eval "require $class"` όταν το όνομα είναι έμπιστο.
- **Χωρίς όρισμα**: η `require;` χρησιμοποιεί την `$_` ως όνομα αρχείου. Σπάνιο σε σύγχρονο κώδικα· σχεδόν πάντα ατύχημα.
- **Οι επανεκτελέσεις δεύτερης ευκαιρίας δεν λειτουργούν**: Αν μια `require` αποτύχει στη μέση της μεταγλώττισης, το `%INC` καταγράφει `undef` για αυτό το όνομα αρχείου. Μια μεταγενέστερη `require` του ίδιου ονόματος βλέπει το undef και εκτοξεύει `Compilation failed in require` χωρίς να ξανατρέξει το αρχείο. Διαγράψτε την εγγραφή του `%INC` αν θέλετε πραγματικά να ξαναδοκιμάσετε (αυτό είναι αιχμηρό άκρο· προτιμήστε να διορθώσετε την υποκείμενη αποτυχία).
- **Αφαίρεση του σχετικού `.`**: το `.` αφαιρέθηκε από το προεπιλεγμένο `@INC` στην Perl 5.26. Ένα σενάριο που παλιά έβρισκε το `Foo/Bar.pm` δίπλα του πρέπει τώρα να προσθέτει ρητά τον δικό του κατάλογο (`use FindBin; use lib $FindBin::Bin;`).
- **Σκίαση `.pmc`**: αν το `Foo/Bar.pmc` υπάρχει στον ίδιο κατάλογο με το `Foo/Bar.pm`, το `.pmc` υπερισχύει. Τα παλιωμένα αρχεία `.pmc` είναι κλασική παγίδα τύπου «γιατί η επεξεργασία μου δεν έχει αποτέλεσμα».
- **Ανάλυση του `require(EXPR)`**: η `require` αναλύεται ως ονομαστικός μοναδιαίος τελεστής· το `require EXPR + 1` σημαίνει `require(EXPR + 1)`. Χρησιμοποιήστε ρητές παρενθέσεις όταν το όρισμα είναι μη τετριμμένη έκφραση.
- **Αυτο-δημιουργία stash**: το `require Foo::Bar` δημιουργεί τη stash `Foo::Bar::` ακόμη και αν το αρχείο λείπει. Επόμενοι έλεγχοι `defined &Foo::Bar::some_sub` μπορεί να περάσουν το στάδιο «το πακέτο υπάρχει» αλλά να αποτύχουν στο στάδιο «η υπορουτίνα υπάρχει».
- **Το filehandle από γάντζο πρέπει να είναι πραγματικό**: Τα tied filehandles αγνοούνται σιωπηλά από το πρωτόκολλο γάντζων· η επεξεργασία ξεπέφτει από το τέλος και η `require` αναφέρει «not found». Χρησιμοποιήστε πραγματικό typeglob.
- **Το `INC` πρέπει να είναι πλήρως κατονομασμένο σε κλάσεις γάντζων**: το `sub Foo::INC { ... }` λειτουργεί· το `sub INC { ... }` μέσα σε `package Foo` εγκαθιστά το `main::INC` λόγω ενός σταθερά κωδικοποιημένου κανόνα πακέτου.
- **Η `require()` είναι δύσκολο να περιτυλιχθεί**: Πολλά αρθρώματα επιθεωρούν τη στοίβα κλήσεων για να βρουν τον καλούντα τους. Ένα περιτύλιγμα που προσθέτει πλαίσιο συχνά τα σπάει. Προτιμήστε τους γάντζους `${^HOOK}{require__before}` / `${^HOOK}{require__after}` (προστέθηκαν στην 5.38) για ιχνηλασία.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `use` - `require` + `import` κατά τη μεταγλώττιση· ο κανονικός τρόπος για να φέρετε ένα άρθρωμα από το οποίο εξαρτάστε ανεπιφύλακτα
- `do` - φορτώνει ένα αρχείο και επιστρέφει την τιμή του σε κάθε κλήση, χωρίς αποθήκευση στο `%INC` και χωρίς τον κανόνα «πρέπει να επιστρέψει αληθές»
- `import` - η μέθοδος που καλεί η `use` μετά την `require`· η `require` δεν την καλεί ποτέ για εσάς
- `@INC` - διαδρομή αναζήτησης που συμβουλεύεται για φορτώσεις με bareword και σχετικό όνομα αρχείου· δέχεται συμβολοσειρές καταλόγων, coderefs, αναφορές πινάκων και αντικείμενα ως γάντζους
- `%INC` - προσωρινή μνήμη μίας εγγραφής ανά φορτωμένο αρχείο, που κάνει την `require` ιδεμποτή και καταγράφει από πού ήρθε κάθε άρθρωμα
- `eval` - τυλίξτε την `require` όταν επιτρέπεται η φόρτωση να αποτύχει

Διάφορα

reset

Καθαρίζει κάθε μεταβλητή πακέτου της οποίας το όνομα ξεκινά με ένα από ένα σύνολο γραμμάτων, και επανοπλίζει αντιστοιχίσεις μίας εκτέλεσης `m?pattern?`.

Η `reset` είναι κατάλοιπο από τον κόσμο της Perl 4 που είχε μόνο καθολικές. Σε σύγχρονο κώδικα επιβιώνει σε δύο στενούς ρόλους: επανόπλιση των regexes μίας αντιστοιχίας `m?...?` ώστε να πυροδοτηθούν ξανά, και - στο τέλος του μπλοκ `continue` ενός βρόχου - εκκαθάριση μιας παρτίδας μεταβλητών πακέτου πριν την επόμενη επανάληψη. Η εμβέλειά της περιορίζεται στο τρέχον πακέτο και στις **μεταβλητές πακέτου**· οτιδήποτε δηλωμένο με `my` παραμένει ανέπαφο.

Σύνοψη

```
reset EXPR
reset
```

Το `EXPR` είναι συμβολοσειρά που ερμηνεύεται ως σύνολο μεμονωμένων χαρακτήρων, με τις παύλες να επιτρέπονται ως εύρη: `'X'`, `'a-z'`, `'abc'`, `'A-Za-z'`. Κάθε βαθμωτό, πίνακας, και πίνακας κατακερματισμού στο **τρέχον πακέτο** του οποίου το όνομα ξεκινά με ένα από αυτά τα γράμματα επαναφέρεται στην παρθένα, μη ορισμένη κατάστασή του.

Χωρίς όρισμα, μόνο οι αναζητήσεις μίας αντιστοιχίας `m?pattern?` στο τρέχον πακέτο επανοπλίζονται - οι μεταβλητές δεν αγγίζονται.

Τι επιστρέφεται

Πάντα 1. Η τιμή επιστροφής δεν φέρει σήμα επιτυχίας· δεν υπάρχει κάτι στο οποίο η `reset` να μπορεί να αποτύχει.

Καθολική κατάσταση που επηρεάζει

- **Κάθε μεταβλητή πακέτου στο τρέχον πακέτο** της οποίας το όνομα ξεκινά με ένα από τα παρατιθέμενα γράμματα: βαθμωτά, πίνακες, και πίνακες κατακερματισμού όλα μαζί. Τα βαθμωτά γίνονται `undef`, οι πίνακες και οι πίνακες κατακερματισμού γίνονται κενοί.
- **Regexes μίας αντιστοιχίας** (`m?pattern?`) μεταγλωττισμένα στο τρέχον πακέτο - η σημαία τους «ήδη αντιστοιχήθηκε» καθαρίζεται ώστε η επόμενη αποτίμηση να αντιστοιχίζει ξανά.
- **Δεν** διασχίζει όρια πακέτων. Η `reset 'X'` μέσα σε `package Foo` αφήνει την `$Bar::X` ήσυχη.
- **Δεν** αγγίζει λεξιλογικές. Οι `my $x` και `our $x` συμπεριφέρονται διαφορετικά εδώ: η `our` είναι ψευδώνυμο προς τη μεταβλητή πακέτου και επαναφέρεται· η `my` όχι.

Παραδείγματα

Επανόπλιση όλων των αντιστοιχίσεων μίας εκτέλεσης στο τρέχον πακέτο - η τυπική σύγχρονη χρήση της `reset`:

```
reset;
```

Καθαρισμός κάθε μεταβλητής πακέτου που ξεκινά με `X` στο τρέχον πακέτο:

```
package Stats;
our $Xtotal = 0;
our @Xsamples = (1, 2, 3);
our %Xindex = (a => 1);
reset 'X';
# $Xtotal, @Xsamples, %Xindex are all emptied
```

Επαναφορά στο τέλος ενός βρόχου ώστε η επόμενη επανάληψη να ξεκινήσει καθαρή:

```
while (more_work()) {
    process();
} continue {
    reset 'abc';   # wipe $a*, @b*, %c* package vars for next pass
}
```

Εύρος χαρακτήρων - μόνο μεταβλητές με πεζά:

```
reset 'a-z';
```

Συνδυασμός μεμονωμένων γραμμάτων και ευρών:

```
reset 'a-cXY';    # a, b, c, X, Y
```

Οριακές περιπτώσεις

- **Η `reset` χωρίς όρισμα δεν αγγίζει καθόλου μεταβλητές.** Επανοπλίζει αντιστοιχίσεις μίας εκτέλεσης `m?...?` στο τρέχον πακέτο και τίποτα άλλο. Αν θέλετε να καθαρίσετε μεταβλητές πρέπει να περάσετε όρισμα.
- **Οι λεξιλογικές είναι αόρατες στη `reset`.** Η `my $Xtotal` παραμένει εντελώς ανεπηρέαστη από την `reset 'X'`· μόνο η μεταβλητή πακέτου `$Xtotal` επηρεάζεται. Κώδικας που μετανάστευσε από `our`/καθολικές πακέτου σε `my` θα διαπιστώσει ότι η `reset` έχει σιωπηλά γίνει no-op για αυτές τις μεταβλητές.
- **Η `reset 'A-Z'` είναι παγίδα.** Στο πακέτο `main` αυτό σβήνει τα `@ARGV`, `@INC`, `%ENV` και κάθε άλλη κεφαλαία καθολική, σχεδόν σίγουρα όχι αυτό που σκόπευε ο κώδικας.
- **Τα βαθμωτά, οι πίνακες και οι πίνακες κατακερματισμού μοιράζονται έναν χώρο ονομάτων εδώ.** Η `reset 'X'` καθαρίζει κάθε μεταβλητή κάθε sigil της οποίας το όνομα ξεκινά με `X`, όχι μόνο βαθμωτά. Δεν υπάρχει τρόπος να περιοριστεί η επαναφορά σε ένα μόνο sigil.
- **Το τρέχον πακέτο αποφασίζεται κατά τη μεταγλώττιση** μέσω της περικλείουσας δήλωσης `package`, όχι από τον καλούντα. Μια κλήση `reset` μέσα σε άρθρωμα καθαρίζει τις μεταβλητές αυτού του αρθρώματος, όχι του καλούντος.
- **Καμία μορφή ορίσματος regex.** Η `reset` δέχεται συμβολοσειρά γραμμάτων, όχι regex ή λίστα ονομάτων. Δεν μπορείτε να γράψετε `reset qr/^X/` ή `reset '$Xtotal'` - το δεύτερο προσπαθεί να επαναφέρει μεταβλητές που ξεκινούν με `$`, `X`, `t`, `o`, `a`, `l`, κάτι που σχεδόν ποτέ δεν είναι αυτό που σκόπευε ο συγγραφέας.

Διαφορές από το upstream

- Η `reset` του pperl είναι no-op - η κλήση αναλύεται, επιστρέφει 1, και δεν έχει παρενέργειες. Οι μεταβλητές πακέτου δεν καθαρίζονται και η κατάσταση μίας αντιστοιχίας `m?pattern?` δεν παρακολουθείται. Κώδικας που βασίζεται στην `reset` για να σβήσει μια παρτίδα καθολικών στο τέλος ενός βρόχου, ή για να ξαναπυροδοτήσει regex μίας αντιστοιχίας, δεν θα δει αυτή τη συμπεριφορά υπό το pperl. Προτιμήστε `my` και συνηθισμένες `m//` regexes - και τα δύο λειτουργούν πανομοιότυπα σε pperl και upstream, και αποτελούν το συνιστώμενο ιδίωμα και upstream.

Δείτε επίσης

- `my` - λεξιλογικές μεταβλητές· προτιμώμενες σε όλο τον νέο κώδικα και ανεπηρέαστες από την `reset`
- `local` - δυναμικά οριοθετημένη αποθήκευση/επαναφορά μεταβλητής πακέτου, η συνηθισμένη απάντηση όταν θέλετε προσωρινή αλλαγή χωρίς χειροκίνητο καθαρισμό
- `m` - η κανονική μορφή `m//`· η `m?...?` είναι η παραλλαγή μίας εκτέλεσης που επανοπλίζει η `reset`
- `continue` - μπλοκ τέλους βρόχου όπου τυπικά τοποθετείται η μαζική επαναφορά της κατάστασης επανάληψης
- `package` - δηλώνει σε ποιου πακέτου τις μεταβλητές θα λειτουργήσει η `reset`

- `undef` - καθαρισμός μίας κατονομασμένης μεταβλητής, η εναλλακτική ανά μεταβλητή έναντι μιας ευρείας `reset`

Ροή ελέγχου

return

Φεύγει από την τρέχουσα υπορουτίνα, `eval`, do `FILE`, μπλοκ `sort`, ή μπλοκ eval regex, αποδίδοντας μια τιμή στον καλούντα.

Η `return` είναι η ρητή έξοδος από ονοματισμένη ή ανώνυμη υπορουτίνα. Σταματά την αξιολόγηση του σώματος της υπορουτίνας στο σημείο κλήσης και παραδίδει την EXPR πίσω σε όποιον κάλεσε την υπορουτίνα. Η τιμή που παράγει η EXPR εξαρτάται από το περιβάλλον του καλούντος - λίστας, βαθμωτό ή κενό - το οποίο το σώμα μπορεί να ελέγξει με `wantarray` πριν αποφασίσει τι να επιστρέψει.

Σύνοψη

```
return EXPR
return
```

Τι επιστρέφεται

Η ίδια η `return` δεν έχει τιμή επιστροφής - δεν συνεχίζει στο τρέχον πλαίσιο. Τερματίζει το πλαίσιο και ο καλών λαμβάνει την EXPR, αξιολογημένη στο περιβάλλον του καλούντος:

- **Περιβάλλον λίστας** - η EXPR αξιολογείται σε περιβάλλον λίστας· η τιμή λίστας της είναι αυτό που βλέπει ο καλών.
- **Βαθμωτό περιβάλλον** - η EXPR αξιολογείται σε βαθμωτό περιβάλλον· η βαθμωτή τιμή της είναι αυτό που βλέπει ο καλών.
- **Κενό περιβάλλον** - η EXPR αξιολογείται ακόμη (για παρενέργειες), αλλά η τιμή απορρίπτεται.

Χωρίς EXPR ο καλών λαμβάνει:

- **περιβάλλον λίστας** - την κενή λίστα `()`
- **βαθμωτό περιβάλλον** - `undef`
- **κενό περιβάλλον** - τίποτα

Αυτή η ασυμμετρία έχει σημασία: το `return;` (γυμνό) **δεν** είναι το ίδιο με το `return undef;`. Ένα γυμνό `return` σε περιβάλλον λίστας αποδίδει `()`, που είναι ψευδές και έχει μηδέν στοιχεία· το `return undef;` σε περιβάλλον λίστας αποδίδει λίστα ενός στοιχείου που περιέχει `undef`, το οποίο είναι **αληθές** (επειδή η λίστα έχει ένα στοιχείο). Υπορουτίνες που σηματοδοτούν σφάλματα πρέπει σχεδόν πάντα να χρησιμοποιούν γυμνό `return`.

```
sub find_it {
    my ($key) = @_;
    return unless exists $cache{$key};  # bare: () in list, undef in␣
↪scalar
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    return $cache{$key};
}

if (my @hits = find_it($k)) { ... }     # () is false; undef-in-list␣
↪is true
```

Πού είναι νόμιμη η return

Η `return` ξετυλίγει αυτά τα πλαίσια:

- ονοματισμένη ή ανώνυμη **υπορουτίνα**
- συμβολοσειρά ή μπλοκ **`eval`** - το `return` από μέσα σε `eval { ... }` φεύγει από το `eval`, όχι από την περικλείουσα υπορουτίνα
- ένα **`do FILE`** - το `return EXPR` είναι η τιμή του αρχείου
- ένα μπλοκ συγκριτή της **`sort`** - επιστρέφει το αποτέλεσμα της σύγκρισης
- ένα **μπλοκ eval regex** - μπλοκ κώδικα `/(?{ ... })/` και `/(?&NAME)/`

Η `return` **δεν** είναι νόμιμη μέσα σε αυτά τα μπλοκ - αποτελεί σφάλμα κατά τη μεταγλώττιση ή κατά τον χρόνο εκτέλεσης ανάλογα με τα συμφραζόμενα:

- **`grep BLOCK LIST`** και **`map BLOCK LIST`** - το μπλοκ δεν είναι πλαίσιο υπορουτίνας· χρησιμοποιήστε αντ” αυτού την τελευταία έκφραση του μπλοκ
- **`do BLOCK`** - το μπλοκ είναι έκφραση, όχι κλήση· η τιμή του είναι ήδη η τελευταία έκφραση του μπλοκ
- γυμνά μπλοκ, σώματα `if` / `while` / `for` - για τον ίδιο λόγο· αυτά δεν σχηματίζουν πλαίσιο κλήσης

Αν χρειάζεστε «να επιστρέψετε τιμή από αυτό το μπλοκ», κάντε την την τελευταία αξιολογούμενη έκφραση του μπλοκ:

```
my @evens = grep { $_ % 2 == 0 } @nums;   # not: grep { return ... }
```

Διάδοση περιβάλλοντος με wantarray

Το περιβάλλον του καλούντος φτάνει στον καλούμενο μέσω της `wantarray`:

- Η `wantarray` είναι **αληθής** σε περιβάλλον λίστας
- Η `wantarray` είναι **ορισμένη και ψευδής** (`""`) σε βαθμωτό περιβάλλον
- Η `wantarray` είναι **`undef`** σε κενό περιβάλλον

Μια υπορουτίνα που θέλει να επιστρέψει τιμές κατάλληλου σχήματος την ελέγχει:

```
sub stats {
    my @xs = @_;
    return unless @xs;                    # bare: () / undef as␣
↪above
    my $sum = 0; $sum += $_ for @xs;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    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
```

Το ίδιο σημείο κλήσης μπορεί να αξιολογηθεί σε διαφορετικά περιβάλλοντα σε διαφορετικές επικλήσεις· η `return` επανεπιλύει την EXPR κάθε φορά.

Έμμεση επιστροφή: κερδίζει η τελευταία έκφραση

Αν ο έλεγχος φτάσει στο τέλος μιας υπορουτίνας (ή `eval`, ή do FILE) χωρίς να συναντήσει `return`, επιστρέφεται η τιμή της τελευταίας αξιολογούμενης έκφρασης, στο περιβάλλον του καλούντος. Αυτό είναι το ιδιωματικό στιλ «τελικής έκφρασης»:

```
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
}
```

Χρησιμοποιήστε ρητό `return` όταν εξέρχεστε νωρίς, ή όταν το τέλος της υπορουτίνας είναι εντολή (π.χ. βρόχος, ανάθεση) της οποίας η τιμή δεν είναι αυτό που θέλετε να δει ο καλών.

Παραδείγματα

Πρόωρη έξοδος σε φύλακα:

```
sub lookup {
    my ($id) = @_;
    return unless defined $id;
    return $table{$id};
}
```

Επιστροφή από μπλοκ `eval` χωρίς να ξετυλιχθεί η περικλείουσα υπορουτίνα:

```
sub try_parse {
    my ($text) = @_;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    my $parsed = eval {
        return {} if $text eq "";          # leaves eval, not try_
↪parse
        parse($text);
    };
    return $@ ? undef : $parsed;
}
```

Επιστροφή ευαίσθητη στο περιβάλλον:

```
sub users {
    return wantarray ? @all_users : scalar @all_users;
}

my @u = users();                           # the users themselves
my $n = users();                           # how many there are
```

Εξαίρεση του αναλυτή - η `return` δεν είναι τελεστής που μοιάζει με συνάρτηση:

```
sub greet { return ("hello") . " world" } # returns "hello world"
                                           # the . " world" is part␣
↪of EXPR
greet();                                   # "hello world"
```

Γι" αυτό το `return (EXPR)` **δεν** μονώνει την EXPR από έναν τελεστή που ακολουθεί· οτιδήποτε μετά την παρένθεση κλεισίματος εξακολουθεί να είναι όρισμα της `return`. Ίδιος κανόνας με την `print`.

Παράνομο: `return` μέσα σε `grep` / `map`:

```
my @odd = grep { return $_ % 2 } @n;       # wrong; leaves the sub
my @odd = grep { $_ % 2 } @n;              # right
```

Μέσα σε `grep` / `map` η τελευταία έκφραση του μπλοκ είναι ήδη η τιμή· ένα `return` εκεί εξέρχεται από την **περικλείουσα υπορουτίνα**, κάτι που σχεδόν ποτέ δεν είναι αυτό που εννοείτε.

Οριακές περιπτώσεις

- **Γυμνό `return` έναντι `return undef`**: όπως παραπάνω, το γυμνό `return` αποδίδει `()` σε περιβάλλον λίστας, πράγμα που επιτρέπει στο `if (my @x = f())` να λειτουργεί ως έλεγχος «κανένα αποτέλεσμα». Το `return undef` βάζει ένα στοιχείο στη λίστα και αναποδογυρίζει αυτόν τον έλεγχο σε αληθές. Προτιμήστε γυμνό `return` εκτός κι αν θέλετε συγκεκριμένα ορισμένο βαθμωτό με τιμή undef.
- **`return` από συγκριτή `sort`**: νόμιμο και συνηθισμένο σε σύνθετους συγκριτές:

  ```
  sort { return $a->[0] <=> $b->[0] || $a->[1] cmp $b->[1] }␣
  ↪@rows;
  ```

- **`return` από `do FILE`**: γίνεται η τιμή του αρχείου, την οποία ελέγχουν τα `require` και `use` για να αποφασίσουν την επιτυχία φόρτωσης αρθρώματος. Ένα `.pm` που

τελειώνει με `1;` είναι ο συνήθης ιδιωματισμός· το `return 1;` κοντά στην κορυφή του αρχείου είναι ισοδύναμο.

- **`return` από μπλοκ regex `(?{ ... })`**: τερματίζει μόνο το μπλοκ κώδικα, όχι την αντιστοίχιση regex. Σπάνια χρήσιμο· η κανονική ροή μπλοκ είναι σχεδόν πάντα σαφέστερη.
- **Οι παρενέργειες σε κενό περιβάλλον συμβαίνουν παρ" όλα αυτά**: το `return expensive();` σε κενό περιβάλλον εξακολουθεί να καλεί την `expensive()`. Η `return` δεν είναι υπόδειξη βελτιστοποίησης.
- **Όχι συνάρτηση, δεν χρειάζονται παρενθέσεις**: η `return` είναι τελεστής, όχι συνάρτηση. Τα `return(1, 2, 3)` και `return 1, 2, 3` είναι ίδια. Η εξαίρεση του αναλυτή παραπάνω σημαίνει ότι οι παρενθέσεις γύρω από την EXPR δεν την οριοθετούν από το υπόλοιπο της εντολής - οι παρενθέσεις είναι αμιγώς για ομαδοποίηση μέσα στην EXPR.
- **Επιστροφή σε μορφή μπλοκ της `eval`**: το `eval { return $x }` φεύγει από το `eval` με `$x`· το `$@` καθαρίζει (καμία εξαίρεση). Για επιστροφή από την περικλείουσα υπορουτίνα μέσω εξαίρεσης, χρησιμοποιήστε `die` αντ" αυτού.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `wantarray` - ελέγχει το περιβάλλον του καλούντος ώστε η `return` να αποδώσει το σωστό σχήμα
- `caller` - το άλλο μισό της ενδοσκόπησης καλούντος· χρήσιμη όταν μια υπορουτίνα αποφασίζει τι θα επιστρέψει με βάση το ποιος την κάλεσε
- `eval` - η `return` μέσα σε μπλοκ `eval` φεύγει από το `eval`, όχι από την περικλείουσα υπορουτίνα
- `sub` - πώς ορίζονται οι υπορουτίνες και πώς η έμμεση επιστροφή αλληλεπιδρά με την τελευταία έκφραση
- `die` - η μη τοπική έξοδος· χρησιμοποιήστε όταν θέλετε να ξετυλίξετε πέρα από ενδιάμεσα πλαίσια αντί να επιστρέφετε μέσα από καθένα τους
- `last` - εξέρχεται από βρόχο, όχι από υπορουτίνα· μη μπερδεύετε τα δύο

Βαθμωτά και συμβολοσειρές · Λίστες

reverse

Αντιστρέφει μια λίστα - ή, σε βαθμωτό περιβάλλον, αντιστρέφει τους χαρακτήρες μιας συμβολοσειράς.

Η `reverse` είναι δύο τελεστές που φορούν το ίδιο όνομα. Το περιβάλλον αποφασίζει ποιον λαμβάνετε. Σε περιβάλλον λίστας επιστρέφει τη `LIST` με τα στοιχεία της σε αντίθετη σειρά· οι τιμές των στοιχείων παραμένουν ανέπαφες. Σε βαθμωτό περιβάλλον συνενώνει τη `LIST` σε μία συμβολοσειρά και επιστρέφει αυτή τη συμβολοσειρά με όλους τους **χαρακτήρες** της αντεστραμμένους. Το συνηθισμένο λάθος να γραφτεί

`my $s = reverse $str` περιμένοντας αντεστραμμένη λίστα (ή `print reverse $str` περιμένοντας αντεστραμμένη συμβολοσειρά) είναι η μεγαλύτερη παγίδα αυτής της συνάρτησης - δείτε *Οριακές περιπτώσεις*.

Σύνοψη

```
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 $_
```

Τι επιστρέφεται

Εξαρτάται από το περιβάλλον. Ο τελεστής επιθεωρεί το περιβάλλον του καλούντος του και διακλαδώνει:

- **Περιβάλλον λίστας** → νέα λίστα με τα στοιχεία της `LIST` σε αντίστροφη σειρά. Οι τιμές των στοιχείων δεν μετατρέπονται σε συμβολοσειρά, δεν τροποποιούνται, δεν αντιγράφονται σε επίπεδο χαρακτήρων. Μια λίστα αναφορών παραμένει λίστα των ίδιων αναφορών.
- **Βαθμωτό περιβάλλον** → μία συμβολοσειρά: η συνένωση των μετατραμμένων σε συμβολοσειρά στοιχείων της `LIST`, με όλους τους χαρακτήρες σε αντίστροφη σειρά. Υπό `use utf8` / συμβολοσειρές πλατιών χαρακτήρων, η αντιστροφή γίνεται κατά χαρακτήρα, όχι κατά byte.

Αυτό σημαίνει ότι το *ίδιο* σημείο κλήσης επιστρέφει ριζικά διαφορετικά σχήματα ανάλογα με τον τρόπο που το χρησιμοποιείτε:

```
my @back  = reverse "abc", "def";    # ("def", "abc")
my $back  = reverse "abc", "def";    # "fedcba"
```

Όταν χρησιμοποιείται χωρίς ορίσματα σε βαθμωτό περιβάλλον, η `reverse` αντιστρέφει το `$_`. Όταν χρησιμοποιείται χωρίς ορίσματα σε περιβάλλον λίστας, η `reverse` επιστρέφει την κενή λίστα - **όχι** ένα αντεστραμμένο `$_`. Το `print reverse;` δεν παράγει έξοδο.

Καθολική κατάσταση που επηρεάζει

Διαβάζει το `$_` όταν καλείται χωρίς ορίσματα σε βαθμωτό περιβάλλον. Αυτή είναι η μόνη καθολική που αγγίζει. Δεν τροποποιεί το `$_`· επιστρέφει την αντεστραμμένη συμβολοσειρά.

Παραδείγματα

Αντιστροφή λίστας. Τα ίδια τα στοιχεία παραμένουν αμετάβλητα:

```
my @a = (1, 2, 3, 4, 5);
my @r = reverse @a;              # (5, 4, 3, 2, 1)
```

Αντιστροφή συμβολοσειράς. Επιβάλετε βαθμωτό περιβάλλον με `scalar` όταν η περιβάλλουσα έκφραση θα ήταν διαφορετικά λίστα:

```
my $s = reverse "Hello, world";  # "dlrow ,olleH"
print scalar reverse "Hello";    # "olleH"
```

Επανάληψη πίνακα προς τα πίσω χωρίς κατασκευή δεύτερου πίνακα. Η `reverse` στην κεφαλίδα του βρόχου είναι κλήση σε περιβάλλον λίστας· η Perl καταναλώνει την αντεστραμμένη λίστα τεμπέλικα από τους μηχανισμούς του βρόχου:

```
my @stack = ("first", "second", "third");
for my $elem (reverse @stack) {
    print "$elem\n";             # third / second / first
}
```

Φθίνουσα ταξινόμηση. Το `reverse sort` είναι το ιδίωμα όταν η σύγκρισή σας είναι αύξουσα και θέλετε το αντίθετο. Για οτιδήποτε πιο σύνθετο από την προεπιλεγμένη σύγκριση συμβολοσειρών, ανταλλάξτε τους τελεσταίους στον συγκριτή:

```
my @asc  = sort { $a <=> $b } @nums;
my @desc = reverse sort { $a <=> $b } @nums;     # works
my @desc2 = sort { $b <=> $a } @nums;            # preferred
```

Αντιστροφή hash. Οι τιμές γίνονται κλειδιά, τα κλειδιά γίνονται τιμές:

```
my %by_name    = (alice => 1, bob => 2, carol => 3);
my %by_id      = reverse %by_name;               # (1 => "alice", ..
 ↪.)
print $by_id{2};                                 # "bob"
```

Το κόλπο `"dlrow ,olleH"` - χρήση της `reverse` σε βαθμωτό περιβάλλον ως μονομερής αναστροφή χαρακτήρων σε μεταβλητή συμβολοσειράς:

```
my $word = "stressed";
my $rev  = scalar reverse $word;                  # "desserts"
```

Οριακές περιπτώσεις

- **Η No.1 παγίδα: περιβάλλον.** Το `my $s = reverse @arr` **δεν** σας δίνει αντεστραμμένο πίνακα· σας δίνει την αντεστραμμένη *συμβολοσειρά* που σχηματίζεται με συνένωση των στοιχείων του πίνακα και αναστροφή κάθε χαρακτήρα. Αν θέλετε αντεστραμμένο πίνακα αποθηκευμένο μέσω βαθμωτού, χρησιμοποιήστε αναφορά: `my $ref = [ reverse @arr ]`.
- **Μονό βαθμωτό, λάθος περιβάλλον.** Το `my @r = reverse $str` επιστρέφει `($str)` - λίστα ενός στοιχείου που περιέχει το `$str` αμετάβλητο. Δεν υπάρχει τίποτα να αναδιαταχθεί. Για να αντιστρέψετε τους χαρακτήρες, επιβάλετε βαθμωτό περιβάλλον: `my $r = reverse $str` ή `my @r = (scalar reverse $str)`.
- **Η αντιστροφή hash χάνει δεδομένα σε διπλές τιμές.** Η `reverse %h` διατρέχει τα ζεύγη κλειδιού/τιμής ως επίπεδη λίστα και ξαναχτίζει hash. Αν δύο κλειδιά μοιράζονται μία τιμή, ένα από τα δύο κερδίζει και το άλλο απορρίπτεται σιωπηλά·

ποιο κερδίζει δεν είναι ορισμένο. Ελέγξτε πρώτα για μοναδικότητα αν αυτό έχει σημασία:

```
my %seen;
$seen{$_}++ for values %h;
die "non-unique values" if grep { $_ > 1 } values %seen;
my %inv = reverse %h;
```

- **Χωρίς ορίσματα σε περιβάλλον λίστας δεν κάνει τίποτα χρήσιμο.** Η `reverse` χωρίς ορίσματα επιστρέφει `()` σε περιβάλλον λίστας. Το `$_` συμβουλεύεται μόνο σε βαθμωτό περιβάλλον. Το `print reverse;` δεν τυπώνει τίποτα· το `print scalar reverse;` τυπώνει την αντιστροφή του `$_`.
- **Κατά χαρακτήρα, όχι κατά byte.** Για συμβολοσειρές πλατιών χαρακτήρων, η `reverse` αντιστρέφει χαρακτήρες, όχι bytes. Μια συμβολοσειρά `:utf8` τριών code points παραμένει τριών code points μετά την αντιστροφή, ανεξάρτητα από τα πλάτη τους σε bytes. Η αντιστροφή ακατέργαστης συμβολοσειράς bytes με χαρακτήρες πολλαπλών bytes που δεν αποκωδικοποιήσατε ποτέ θα την αλλοιώσει - αποκωδικοποιήστε πρώτα.
- **Η αυτο-ανάθεση διατηρεί τις τρύπες.** Το `@a = reverse @a` σε μη μαγικό ή καλοδιατεθειμένο tied πίνακα διατηρεί τα ανύπαρκτα στοιχεία στις αντεστραμμένες θέσεις τους· ένας αραιός πίνακας παραμένει αραιός.
- **Το `reverse sort` δεν είναι ίδιο με το `sort { $b cmp $a }`.** Και τα δύο δίνουν φθίνουσα σειρά για σταθερή προεπιλεγμένη σύγκριση, αλλά το `reverse sort` κάνει δύο περάσματα πάνω στα δεδομένα και είναι μετρήσιμα πιο αργό σε μεγάλες λίστες. Προτιμήστε την ανταλλαγή των τελεσταίων του συγκριτή.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `sort` - συνδυάστε με τη `reverse` για φθίνουσα σειρά, ή ανταλλάξτε τα `$a`/`$b` στον συγκριτή για να αποφύγετε το δεύτερο πέρασμα
- `pop` - αφαίρεση από το τέλος ενός πίνακα όταν θέλετε απλώς το τελευταίο στοιχείο, όχι αντιστροφή όλης της λίστας
- `shift` - αφαίρεση από την αρχή· συνδυασμένη με αντίγραφο `reverse`-αρισμένο σας δίνει πρόσβαση τύπου στοίβας στην αρχική ουρά
- `split` - παράγει λίστα χαρακτήρων (με `//`) ή πεδίων που μπορείτε στη συνέχεια να `reverse`-άρετε
- `join` - ο φυσικός σύντροφος όταν αντιστρέφετε λίστα συμβολοσειρών και θέλετε πίσω μία ενιαία συμβολοσειρά
- `$_` - ο προεπιλεγμένος στόχος όταν η `reverse` καλείται χωρίς ορίσματα σε βαθμωτό περιβάλλον

I/O

rewinddir

Επαναφέρει ένα handle καταλόγου στην αρχή της λίστας του.

Η `rewinddir` θέτει την τρέχουσα θέση του `DIRHANDLE` πίσω στην αρχή του καταλόγου, ώστε η επόμενη κλήση `readdir` να επιστρέψει ξανά την πρώτη εγγραφή. Είναι το αντίστοιχο της `seek` σε file handle για handle καταλόγου: κανένα εκ νέου άνοιγμα, καμία εκ νέου σάρωση του γονικού καταλόγου - απλώς μια επαναφορά θέσης στο ανοιχτό handle.

Σύνοψη

```
rewinddir DIRHANDLE
```

Τι επιστρέφεται

1 σε επιτυχία, `0` σε αποτυχία (με το `$!` τεθειμένο). Η αποτυχία είναι σπάνια στην πράξη - η συνηθισμένη αιτία είναι ένα handle που δεν ανοίχτηκε ποτέ ή έχει ήδη κλείσει. Ελέγξτε την τιμή επιστροφής αν σας ενδιαφέρει είτε η μία είτε η άλλη περίπτωση:

```
rewinddir $dh
    or die "rewinddir failed: $!";
```

Τυπική χρήση

Έχετε ήδη διασχίσει έναν κατάλογο με `readdir` και θέλετε να τον διασχίσετε ξανά χωρίς να πληρώσετε για ένα ακόμη `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 $entry =~ /^\.\.?\z/;
    process("$path/$entry");
}

closedir $dh;
```

Ένα δεύτερο συνηθισμένο μοτίβο είναι η επαναχρησιμοποίηση ενός handle σε αρκετές σαρώσεις με διαφορετικά φίλτρα, όπου το εκ νέου άνοιγμα θα ήταν σπατάλη ή θα έμπαινε σε race με ταυτόχρονες αλλαγές του καταλόγου:

```
opendir my $dh, $path or die $!;

my @perl = grep { /\.p[lm]\z/ } readdir $dh;
rewinddir $dh;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
my @text = grep { /\.txt\z/  } readdir $dh;

closedir $dh;
```

Οριακές περιπτώσεις

- **Μη ανοιγμένο ή κλειστό handle**: επιστρέφει `0` και θέτει το `$!`. Υπό `use warnings` λαμβάνετε επίσης την προειδοποίηση `rewinddir() attempted on invalid dirhandle`.
- **Αλλαγές καταλόγου κατά τη μέση της σάρωσης**: η `rewinddir` **δεν** εγγυάται ότι θα αναδείξει εγγραφές που δημιουργήθηκαν (ή θα κρύψει εγγραφές που αφαιρέθηκαν) αφότου επέστρεψε η `opendir`. Η υποκείμενη `rewinddir(3)` λειτουργεί στην προσωρινά αποθηκευμένη ροή καταλόγου του πυρήνα· το αν μια επαναφερμένη σάρωση παρατηρεί ταυτόχρονες `mkdir`, `unlink`, ή `rename` στον ίδιο κατάλογο εξαρτάται από το σύστημα αρχείων και το λειτουργικό σύστημα. Αν χρειάζεστε ένα φρέσκο στιγμιότυπο, ξανακάντε `closedir` και `opendir`.
- **Δεν είναι αναζήτηση σε αυθαίρετη θέση**: η `rewinddir` επαναφέρει στην αρχή και τίποτε άλλο. Για να επιστρέψετε σε μια απομνημονευμένη θέση στη μέση του καταλόγου χρησιμοποιήστε `telldir` / `seekdir` αντ” αυτής.
- **Η σειρά εγγραφών δεν είναι προδιαγεγραμμένη**: η επαναφορά αναπαράγει την ίδια ροή που σας έδωσε ο πυρήνας, αλλά η σειρά των εγγραφών είναι ό,τι παράγει το σύστημα αρχείων - όχι ταξινομημένη, όχι σειρά δημιουργίας, ούτε η σειρά που είδατε την προηγούμενη φορά σε διαφορετικό handle. Ταξινομήστε ρητά αν η σειρά έχει σημασία.
- **Φορητότητα**: η `rewinddir` είναι κλήση POSIX. Σε πλατφόρμες χωρίς λειτουργικό `rewinddir(3)` ενδέχεται να υλοποιείται με κλείσιμο και εκ νέου άνοιγμα του καταλόγου· δείτε το L<perlport/rewinddir> για την ιστορική λίστα. Στο Linux (την πλατφόρμα που στοχεύει το pperl) αυτό δεν αποτελεί πρόβλημα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `opendir` - ανοίγει έναν κατάλογο και αποκτά το handle στο οποίο λειτουργεί η `rewinddir`
- `readdir` - διαβάζει την επόμενη εγγραφή· συνεχίζει από όπου άφησε τη ροή η `rewinddir` (από την αρχή)
- `closedir` - απελευθερώνει το handle· συνδυάστε με `opendir` όταν μια επαναφερμένη σάρωση δεν αρκεί και χρειάζεστε ένα φρέσκο στιγμιότυπο του καταλόγου
- `telldir` - καταγράφει την τρέχουσα θέση ώστε να μπορείτε να επιστρέψετε σε αυτή αργότερα

- `seekdir` - επαναφέρει μια θέση που είχε προηγουμένως αποτυπωθεί από την `telldir`· το `rewinddir $dh` είναι ισοδύναμο με το `seekdir $dh, 0` σε συμμορφωμένα συστήματα, αλλά η `rewinddir` είναι η φορητή γραφή

Βαθμωτά και συμβολοσειρές

rindex

Βρίσκει τη θέση της **τελευταίας** εμφάνισης μιας υποσυμβολοσειράς μέσα σε μια συμβολοσειρά.

Η `rindex` είναι το αντίστοιχο της `index` από δεξιά προς τα αριστερά. Σαρώνει τη `STR` για τη `SUBSTR` και επιστρέφει το offset με βάση το μηδέν της δεξιότερης αντιστοιχίας, ή `-1` αν δεν υπάρχει αντιστοιχία. Όταν δίνεται `POSITION`, η αναζήτηση περιορίζεται σε αντιστοιχίες των οποίων το offset εκκίνησης είναι μικρότερο ή ίσο της `POSITION` - δηλαδή, η τελευταία εμφάνιση σε αυτή τη θέση ή πριν από αυτήν.

Σύνοψη

```
rindex $str, $substr
rindex $str, $substr, $position
```

Τι επιστρέφεται

Έναν ακέραιο. Μηδέν ή θετικό σημαίνει το offset με βάση το μηδέν της δεξιότερης αντιστοιχίας· `-1` σημαίνει ότι η `SUBSTR` δεν εμφανίζεται στη `STR` (υποκείμενο στον περιορισμό `POSITION`, αν υπάρχει).

Τα offsets μετρώνται σε **χαρακτήρες** όταν η `STR` περιέχει συμβολοσειρά χαρακτήρων, και σε **bytes** όταν η `STR` είναι συμβολοσειρά bytes. Αυτό ταιριάζει με την `index` και τις υπόλοιπες ενσωματωμένες συναρτήσεις συμβολοσειρών της Perl - η μονάδα είναι ό,τι χρησιμοποιεί η ίδια η συμβολοσειρά.

Πώς η POSITION διαμορφώνει την αναζήτηση

Η `POSITION` είναι ένα **άνω φράγμα στο offset εκκίνησης της αντιστοιχίας**, όχι στο σημείο όπου ξεκινά η σάρωση. Το αποτέλεσμα ικανοποιεί:

- `result <= POSITION`
- `result + length(SUBSTR) <= length(STR)` (όπως πάντα)

Συγκεκριμένα:

- Η `POSITION` παραλείπεται ή είναι `undef` - αναζητήστε ολόκληρη τη συμβολοσειρά· επιστρέψτε τη δεξιότερη αντιστοιχία.
- Η `POSITION` είναι μεγαλύτερη ή ίση του `length(STR)` - συμπεριφέρεται σαν η `POSITION` να ήταν `length(STR)`· λαμβάνεται υπόψη ολόκληρη η συμβολοσειρά.
- Η `POSITION` είναι μικρότερη του `0` - συμπεριφέρεται σαν η `POSITION` να ήταν `0`· μόνο αντιστοιχία που ξεκινά στο offset `0` μπορεί να ικανοποιήσει τον περιορισμό, οπότε η τιμή επιστροφής είναι `0` όταν η `STR` ξεκινά με τη `SUBSTR` και `-1` διαφορετικά.

- Κενή SUBSTR - κάθε offset από `0` έως `length(STR)` αποτελεί έγκυρη θέση αντιστοιχίας, οπότε η `rindex` επιστρέφει `min(POSITION, length(STR))` (ή `length(STR)` όταν παραλείπεται η POSITION).

Σκεφτείτε την POSITION ως «βρες την τελευταία αντιστοιχία εδώ ή πριν από εδώ», όχι ως «ξεκίνα τη σάρωση εδώ».

Παραδείγματα

Βασική αναζήτηση από δεξιά προς τα αριστερά:

```
rindex("banana", "a");          # 5  (the final 'a')
rindex("banana", "na");         # 4  (last "na")
rindex("banana", "z");          # -1 (not found)
```

Βηματισμός προς τα πίσω σε κάθε εμφάνιση χρησιμοποιώντας κάθε αποτέλεσμα, μείον ένα, ως την επόμενη 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;
}
```

Η POSITION ως άνω φράγμα - η αντιστοιχία πρέπει να **ξεκινά** σε αυτή τη θέση ή πριν από αυτήν:

```
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)
```

Η κενή SUBSTR ταιριάζει σε κάθε offset, οπότε το αποτέλεσμα είναι η περικομμένη θέση:

```
rindex("banana", "", 4);        # 4
rindex("banana", "");           # 6  (length of "banana")
```

Συνδυασμός με την `index` για τον εντοπισμό του τελευταίου τμήματος μιας οριοθετημένης συμβολοσειράς - συνηθισμένο ιδίωμα για την απομόνωση τελικής επέκτασης ή συστατικού διαδρομής:

```
my $path = "/usr/local/bin/perl";
my $slash = rindex($path, "/");
my $file  = substr($path, $slash + 1);   # "perl"
```

Οριακές περιπτώσεις

- **Υποσυμβολοσειρά μεγαλύτερη από συμβολοσειρά**: η `rindex("abc", "abcd")` επιστρέφει `-1`. Καμία αντιστοιχία δεν μπορεί να χωρέσει.
- **Υποσυμβολοσειρά ίση με τη συμβολοσειρά**: η `rindex("abc", "abc")` επιστρέφει `0`. Η τελευταία (και μοναδική) αντιστοιχία ξεκινά στο offset `0`.

- **Η POSITION πέρα από το τέλος**: η `rindex($str, $sub, 9999)` συμπεριφέρεται σαν η POSITION να ήταν `length($str)` - σαρώνεται ολόκληρη η συμβολοσειρά.
- **Η POSITION είναι αρνητική**: αντιμετωπίζεται ως `0`. Η `rindex("abc", "b", -5)` επιστρέφει `-1` (καμία αντιστοιχία δεν μπορεί να ξεκινήσει στο offset `0` ή νωρίτερα)· η `rindex("abc", "a", -5)` επιστρέφει `0`.
- **Η POSITION είναι undef**: το ίδιο με την παράλειψή της. Υπό `use warnings` αυτό εκπέμπει προειδοποίηση `uninitialized`.
- **Κενή STR, μη κενή SUBSTR**: επιστρέφει `-1`.
- **Κενή STR, κενή SUBSTR**: επιστρέφει `0`.
- **Unicode**: για συμβολοσειρές χαρακτήρων (αυτές που ο διερμηνέας γνωρίζει ότι είναι κείμενο), τα offsets μετρούν χαρακτήρες, όχι bytes. Μια συμβολοσειρά μεταβεβλημένη σε σημασιολογία bytes μέσω `use bytes` ή `encode_utf8` θα αναφέρει offsets bytes αντί γι" αυτό. Ο κανόνας αντικατοπτρίζει τις `index` και `substr`.
- **Tainted ορίσματα**: το αποτέλεσμα κληρονομεί taint από τις STR και SUBSTR υπό `-T`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `index` - το αντίστοιχο από αριστερά προς τα δεξιά· πανομοιότυπες συμβάσεις ορισμάτων, επιστρέφει την **πρώτη** εμφάνιση
- `substr` - τυπικά η επόμενη κλήση μετά από επιτυχημένη `rindex`, για να φετάρει την αντιστοιχισμένη περιοχή ή ό,τι ακολουθεί
- `length` - χρήσιμη κατά τον υπολογισμό της ουραίας φέτας: `substr($s, $hit + length($sub))`
- `pos` - για σάρωση οδηγούμενη από regex όπου χρειάζεστε την κατάσταση αντιστοιχίας της μηχανής αντί για ένα μοναδικό offset υποσυμβολοσειράς
- `split` - όταν θέλετε κάθε τμήμα οριοθετημένο από τη SUBSTR, όχι μόνο το τελευταίο
- `reverse` - σε συνδυασμό με την `index` θεωρείται μερικές φορές κατά λάθος υποκατάστατο της `rindex`· η `rindex` είναι σαφέστερη και γρηγορότερη

Filehandles, αρχεία, κατάλογοι

rmdir

Αφαιρεί κενό κατάλογο.

Η `rmdir` ζητά από το λειτουργικό σύστημα να διαγράψει τον κατάλογο που ονομάζεται `FILENAME`. Ο κατάλογος πρέπει να είναι κενός - δεν πρέπει να περιέχει άλλες εγγραφές εκτός των `.` και `...` Αν παραλειφθεί το `FILENAME`, η `rmdir` λειτουργεί στο `$_`. Πρόκειται για λεπτό περιτύλιγμα της κλήσης συστήματος `rmdir(2)` και φέρει όλους

τους περιορισμούς της: δεν μπορείτε να αφαιρέσετε μη κενό κατάλογο, δεν μπορείτε να αφαιρέσετε κατάλογο του οποίου τον γονέα δεν έχετε δικαίωμα εγγραφής, και στα περισσότερα συστήματα δεν μπορείτε να αφαιρέσετε τον τρέχοντα κατάλογο εργασίας σας.

Σύνοψη

```
rmdir FILENAME
rmdir
```

Τι επιστρέφεται

1 σε επιτυχία, `0` σε αποτυχία (με το `$!` ρυθμισμένο στο `errno` από την αποτυχημένη κλήση συστήματος). Σε αντίθεση με τις περισσότερες πρωτογενείς λειτουργίες I/O η `rmdir` δεν επιστρέφει `undef` σε αποτυχία - επιστρέφει τον ακέραιο `0`, οπότε ο συνηθισμένος έλεγχος `rmdir $dir or die $!` εξακολουθεί να λειτουργεί επειδή το `0` είναι ψευδές.

Πάντα ελέγχετε την τιμή επιστροφής. Διαφορετικά μια αποτυχημένη `rmdir` είναι σιωπηρή· η επόμενη γραμμή κώδικα θα προχωρήσει χαρούμενα σαν να είχε αφαιρεθεί ο κατάλογος.

```
rmdir $dir
    or die "rmdir $dir failed: $!";
```

Καθολική κατάσταση που επηρεάζει

- `$_` - χρησιμοποιείται ως `FILENAME` όταν δεν δίνεται όρισμα.
- `$!` - τίθεται στο `errno` της αποτυχημένης κλήσης `rmdir(2)` σε αποτυχία· μένει ανέγγιχτο σε επιτυχία. Διαβάστε το άμεσα· οποιαδήποτε ενδιάμεση κλήση συστήματος μπορεί να το αντικαταστήσει.

Παραδείγματα

Απλή κλήση, ρητός κατάλογος:

```
rmdir "build/tmp"
    or die "rmdir build/tmp: $!";
```

Προεπιλεγμένος στόχος. Μέσα σε `while (readdir ...)` ή σε βρόχο που αφήνει το όνομα στο `$_`:

```
for (@empty_dirs) {
    rmdir or warn "rmdir $_: $!";
}
```

Διάκριση μεταξύ «έχει ήδη φύγει» και πραγματικού σφάλματος. Το `ENOENT` σημαίνει ότι ο κατάλογος δεν υπάρχει, που είναι συχνά εντάξει για κώδικα καθαρισμού:

```
use Errno qw(ENOENT);
unless (rmdir $dir) {
    die "rmdir $dir: $!" unless $! == ENOENT;
}
```

Αφαίρεση καταλόγου μόνο αφού πρώτα τον αδειάσετε:

```
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;
rmdir $dir or die "rmdir $dir: $!";
```

Η αναδρομική αφαίρεση **δεν** είναι δουλειά της `rmdir`. Χρησιμοποιήστε την `remove_tree` του `File::Path` - διασχίζει το δέντρο και διαγράφει εγγραφές από κάτω προς τα πάνω:

```
use File::Path qw(remove_tree);
remove_tree("build", { error => \my $err });
die "remove_tree: @$err" if @$err;
```

Οριακές περιπτώσεις

- **Μη κενός κατάλογος**: το `$!` τίθεται σε `ENOTEMPTY` στα περισσότερα συστήματα Unix (ορισμένα BSDs χρησιμοποιούν `EEXIST`). Αυτό είναι η μακράν πιο συνηθισμένη αποτυχία της `rmdir`. Καταγράψτε τα περιεχόμενα του καταλόγου ή καλέστε την `remove_tree` αν στην πραγματικότητα θέλετε αναδρομική διαγραφή.
- **Χωρίς όρισμα**: η `rmdir` χωρίς όρισμα χρησιμοποιεί το `$_`, όχι το `@ARGV` ούτε τον τρέχοντα κατάλογο. Η γραφή `rmdir()` με κενές παρενθέσεις είναι το ίδιο με `rmdir` χωρίς όρισμα - εξακολουθεί να διαβάζει το `$_`.
- **Symlink προς κατάλογο**: η `rmdir` αρνείται να αφαιρέσει symlinks· λειτουργεί μόνο σε καταλόγους. Χρησιμοποιήστε `unlink` στο ίδιο το symlink. Το `$!` τυπικά τίθεται σε `ENOTDIR`.
- **Τρέχων κατάλογος εργασίας**: Στο Linux, η `rmdir "."` συνήθως αποτυγχάνει με `EBUSY` ή `EINVAL` ανάλογα με το σύστημα αρχείων. Μη βασίζεστε στο ότι μπορείτε να αφαιρέσετε τον κατάλογο στον οποίο βρίσκεστε· κάντε πρώτα `chdir` αλλού.
- **Δικαιώματα**: χρειάζεστε δικαίωμα εγγραφής + εκτέλεσης στον **γονικό** κατάλογο, όχι στον κατάλογο που αφαιρείται. Η αφαίρεση του `/a/b/c` ελέγχει το mode του `/a/b`, όχι του `/a/b/c`.
- **Γονέας με sticky bit** (π.χ. `/tmp`): πρέπει επίσης να είστε κάτοχος του καταλόγου που αφαιρείται ή κάτοχος του γονέα· διαφορετικά `EPERM`.
- **Τελική κάθετος**: η `rmdir "foo/"` λειτουργεί και σημαίνει το ίδιο με `rmdir "foo"`.
- **Κούρσα με άλλη διεργασία**: μεταξύ του βρόχου `opendir`/`readdir` σας και της τελικής `rmdir`, μια άλλη διεργασία μπορεί να δημιουργήσει αρχείο στον κατάλογο.

Θα δείτε τότε ENOTEMPTY παρά το ότι μόλις τον αδειάσατε. Δοκιμάστε ξανά ή αποτύχετε με θόρυβο - μην το αγνοήσετε σιωπηρά.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `mkdir` - δημιουργεί κατάλογο· η αντίστροφη λειτουργία, με την ίδια σύμβαση ορίσματος FILENAME-ή-`$_`
- `unlink` - αφαιρεί **αρχεία** (και symlinks)· η `rmdir` αφαιρεί μόνο κενούς καταλόγους
- `chdir` - αλλάζει τον τρέχοντα κατάλογο εργασίας, συχνά χρησιμοποιείται πριν από `rmdir` για να αποφύγετε την προσπάθεια αφαίρεσης του καταλόγου στον οποίο βρίσκεστε
- `opendir` / `readdir` - καταγράφει τα περιεχόμενα ενός καταλόγου πριν τον αδειάσετε και τον αφαιρέσετε
- `$!` - κρατά το `errno` αποτυχημένης `rmdir`· ελέγξτε για ENOENT / ENOTEMPTY / EPERM για να διακλαδώσετε βάσει του λόγου

Κανονικές εκφράσεις και αντιστοίχιση μοτίβων

s///

Αναζητά σε μια συμβολοσειρά ένα μοτίβο και αντικαθιστά κάθε αντιστοιχία με ένα κείμενο αντικατάστασης.

Ο `s///` είναι ο τελεστής αντικατάστασης της Perl. Εντοπίζει κείμενο που ταιριάζει με το PATTERN στη συμβολοσειρά-στόχο, αντικαθιστά κάθε αντιστοιχία με το REPLACEMENT, και εξ ορισμού επιστρέφει τον αριθμό των αντικαταστάσεων που έγιναν. Χωρίς ρητό στόχο (χωρίς =~ ή !~), λειτουργεί επί του `$_`. Ο στόχος πρέπει να είναι βαθμωτό lvalue - μεταβλητή, στοιχείο πίνακα ή hash, ή `substr` / ανάθεση που παράγει ένα - εκτός αν χρησιμοποιηθεί η σημαία `/r`, η οποία λειτουργεί σε οποιαδήποτε τιμή και επιστρέφει το τροποποιημένο αντίγραφο.

Σύνοψη

```
s/PATTERN/REPLACEMENT/FLAGS
$str =~ s/PATTERN/REPLACEMENT/FLAGS
$str !~ s/PATTERN/REPLACEMENT/FLAGS
my $copy = $str =~ s/PATTERN/REPLACEMENT/r
```

Τι επιστρέφεται

Εξ ορισμού, το πλήθος των αντικαταστάσεων που πραγματοποιήθηκαν ως ακέραιος: `0` όταν τίποτα δεν ταίριαξε (το οποίο είναι ψευδές σε λογικό περιβάλλον), θετικός ακέραιος σε επιτυχία. Το βαθμωτό-στόχος τροποποιείται επιτόπια.

Με τη σημαία `/r`, η σημασιολογία αντιστρέφεται: ο στόχος **δεν** τροποποιείται, και ο τελεστής επιστρέφει τη (πιθανώς αλλαγμένη) συμβολοσειρά. Αν τίποτα δεν ταίριαξε, το `/r` εξακολουθεί να επιστρέφει αντίγραφο του πρωτοτύπου. Το αποτέλεσμα του `/r` είναι πάντα απλή συμβολοσειρά, ακόμη και όταν η πηγή είναι tied μεταβλητή ή blessed αντικείμενο με υπερφόρτωση.

Το `!~` αντιστρέφει τη λογική έννοια της τιμής επιστροφής: αποδίδει αληθές όταν δεν έγινε καμία αντικατάσταση, ψευδές όταν έγινε τουλάχιστον μία αντικατάσταση. Ο στόχος εξακολουθεί να τροποποιείται και στις δύο περιπτώσεις.

```
my $n = ($text =~ s/foo/bar/g);      # count of replacements
my $out = $text =~ s/foo/bar/r;      # copy-and-modify, $text
↪untouched
```

Καθολική κατάσταση που επηρεάζει

- `$_` - ο προεπιλεγμένος στόχος όταν δεν δίνεται `=~` ή `!~`. Το `s/old/new/` χωρίς σύνδεση διαβάζει και γράφει το `$_`.
- Τα `$&`, [`$``](../perlvar), [`$"`](../perlvar), [`$1`](../perlvar) και τα όμοιά τους - ορίζονται από κάθε επιτυχημένη αντιστοιχία, ακριβώς όπως τα ορίζει η [`m//`](m). Μέσα στο `REPLACEMENT` αναφέρονται στην τρέχουσα αντιστοιχία, όχι σε προηγούμενη.
- `pos` - εκκαθαρίζεται στο βαθμωτό-στόχο από μια επιτυχημένη μη-`/g` αντικατάσταση· υπό το `/g` προχωρά με κάθε αντιστοιχία και μπορεί να ελεγχθεί μεταξύ επαναλήψεων.
- Το **τελευταίο επιτυχημένο μοτίβο** - χρησιμοποιείται όταν το `PATTERN` αξιολογείται στην κενή συμβολοσειρά (το `s//new/` επαναχρησιμοποιεί την τελευταία επιτυχημένη regex από αυτή την εμβέλεια).
- Τα `$/`, `$\` και οι διαχωριστές εξόδου δεν επηρεάζονται - το `s///` παράγει συμβολοσειρά, δεν γράφει.

Σημαίες

Το πλήρες σύνολο είναι `msixpodualngcer`. Οι σημαίες πλευράς αντιστοίχισης συμπεριφέρονται ακριβώς όπως στα `m//` και `qr`· δείτε το *κεφάλαιο τροποποιητών* του οδηγού regex για τις λεπτομέρειες πλευράς αντιστοίχισης. Σημαίες ειδικές για την αντικατάσταση:

- `e` - αξιολογεί το `REPLACEMENT` ως μπλοκ κώδικα Perl. Η τιμή του μπλοκ είναι η συμβολοσειρά αντικατάστασης. Ισοδύναμο με την περιτύλιξή του σε `do { ... }` και τη συμβολοσειροποίηση του αποτελέσματος.
- `ee` - αξιολογεί ως κώδικα, και έπειτα κάνει `eval` την προκύπτουσα συμβολοσειρά ξανά. Ισοδύναμο με `eval(do { REPLACEMENT })`. Επιπλέον τροποποιητές `e` προσθέτουν περισσότερα στρώματα `eval`.
- `r` - μη καταστροφικό: αφήνει τον στόχο αμετάβλητο και επιστρέφει το τροποποιημένο αντίγραφο (ή αμετάβλητο αντίγραφο σε περίπτωση μη αντιστοιχίας). Δεν μπορεί να συνδυαστεί με τον περιορισμό μη-βαθμωτού lvalue στόχου - οποιαδήποτε έκφραση είναι έγκυρος στόχος υπό το `/r`.

- `g` - καθολικό: αντικαθιστά κάθε μη επικαλυπτόμενη αντιστοιχία, όχι μόνο την πρώτη.
- `c` - γίνεται δεκτό για συντακτική συμμετρία με τη `m//` αλλά δεν έχει επίδραση στη συμπεριφορά του `s///`· ενεργοποιεί προειδοποίηση υπό το `use warnings`.

Σημαίες πλευράς αντιστοίχισης σε μία γραμμή: `m` (πολυγραμμικά ^/$), `s` (το . αντιστοιχίζει νέα γραμμή), `i` (χωρίς διάκριση πεζών-κεφαλαίων), `x` / `xx` (εκτεταμένα κενά, σχόλια), `p` (διατηρεί τις μεταβλητές αντιστοίχισης· παρωχημένο από την 5.20), `o` (μεταγλώττιση μοτίβου μία φορά), `d` / `u` / `a` / `l` (σημασιολογία συνόλου χαρακτήρων), `n` (μη συλλαμβάνον εξ ορισμού). Δείτε το *κεφάλαιο τροποποιητών*.

Οριοθέτες

Οποιοσδήποτε χαρακτήρας που δεν είναι λευκός μπορεί να αντικαταστήσει το `/`. Αν ο επιλεγμένος οριοθέτης είναι ένας από τους `(`, `[`, `{`, ή `<`, ο τελεστής λαμβάνει **δύο** ομάδες σε αγκύλες και ο οριοθέτης μεταξύ τους είναι προαιρετικός:

```
s(foo)(bar)
s{foo}{bar}
s<foo>/bar/
s[foo]{bar}g
```

Οι οριοθέτες με μονά εισαγωγικά απενεργοποιούν την παρεμβολή τόσο στο `PATTERN` όσο και στο `REPLACEMENT` (εκτός αν ισχύει το `/e`, το οποίο αναλύει πάντα το `REPLACEMENT` ως κώδικα):

```
s'$name'$value';    # literal $name → literal $value, no␣
↪interpolation
```

Τα backticks είναι κανονικοί οριοθέτες· το `s`x`y`` δεν εκτελεί εντολή κελύφους.

Όταν ο οριοθέτης είναι χαρακτήρας που είναι επίσης χαρακτήρας προσδιοριστή, απαιτείται κενό μετά το `s` ώστε ο αναλυτής να μη το διαβάσει ως μέρος του ονόματος του τελεστή.

Παραδείγματα

Βασική επιτόπια αντικατάσταση στο `$_`:

```
$_ = 'the quick brown fox';
s/quick/slow/;                  # $_ is now 'the slow brown fox'
```

Ρητός στόχος με =~:

```
my $path = '/usr/bin/perl';
$path =~ s|/usr/bin|/usr/local/bin|;
# $path is '/usr/local/bin/perl'
```

Καθολική αντικατάσταση, με σύλληψη του πλήθους:

```
my $text = 'one two two three two';
my $n = ($text =~ s/two/2/g);   # $text: 'one 2 2 three 2'; $n == 3
```

Μη καταστροφικό `/r` - αφήνει την πηγή ως έχει, επιστρέφει το αντίγραφο. Ιδιωματικό σε `map`:

```
my @clean = map { s/\s+\z//r } @lines;   # @lines untouched
```

Αξιολόγηση κώδικα με `/e`. Η δεξιά πλευρά είναι έκφραση Perl της οποίας η τιμή γίνεται η αντικατάσταση:

```
my $s = 'abc123xyz';
$s =~ s/\d+/$& * 2/e;           # $s is 'abc246xyz'

my %percent = (n => "\n", t => "\t");
$s =~ s/%(.)/$percent{$1} || $&/ge;     # expand %-escapes
```

Αλυσιδωτό `/r` - καθαρά συναρτησιακοί μετασχηματισμοί χωρίς ενδιάμεση μεταβλητή:

```
my $out = $text =~ s/foo/bar/r
                =~ s/baz/qux/r;
```

Ομάδες σύλληψης και οπισθαναφορές στην αντικατάσταση - προσέξτε τη μορφή `$1`, όχι `\1`:

```
# swap the first two whitespace-separated fields
s/(\S+)\s+(\S+)/$2 $1/;
```

Ο ιδιωματισμός `1 while` για αντικαταστάσεις που δημιουργούν νέες αντιστοιχίες:

```
# insert commas into an integer from the right
my $n = '1234567890';
1 while $n =~ s/(\d)(\d\d\d)(?!\d)/$1,$2/;
# $n is '1,234,567,890'
```

Οριακές περιπτώσεις

- **Ο στόχος πρέπει να είναι βαθμωτό lvalue** (εκτός αν χρησιμοποιείται το `/r`): μεταβλητή, στοιχείο, ή lvalue της `substr`. Το `s/x/y/` έναντι κυριολεκτικής συμβολοσειράς ή του αποτελέσματος μιας κλήσης συνάρτησης είναι σφάλμα κατά τη μεταγλώττιση. Υπό το `/r` οποιαδήποτε έκφραση είναι επιτρεπτή αφού τίποτα δεν τροποποιείται.
- **Το κενό μοτίβο επαναχρησιμοποιεί την τελευταία επιτυχημένη αντιστοιχία.** Το `s//new/` αντικαθιστά έναντι όποιου `PATTERN` ταίριαξε πιο πρόσφατα στην περικλείουσα εμβέλεια. Βολικό μέσα σε `while (m/.../g) { s//x/ }`, εκπληκτικό όταν έχει εκτελεστεί άσχετος κώδικας ενδιάμεσα.
- **Η παρεμβολή συμβαίνει κατά τον χρόνο εκτέλεσης.** Το `s/$foo/$bar/` επανα-μεταγλωττίζει το μοτίβο σε κάθε κλήση αν το `$foo` αλλάξει. Χρησιμοποιήστε `/o` για μεταγλώττιση μία φορά και πάγωμα της συλληφθείσας τιμής - ή προμεταγλωττίστε με `qr`, που είναι η σύγχρονη εναλλακτική.
- **Το `/e` αναλύει την αντικατάσταση κατά τη μεταγλώττιση** παρόλο που εκτελείται κατά τη στιγμή της αντιστοίχισης. Τα συντακτικά σφάλματα εμφανίζονται κατά τη φόρτωση του σεναρίου, όχι όταν πυροδοτείται η αντικατάσταση. Κάθε επιπλέον `e` τυλίγει μία ακόμη `eval` γύρω από την τιμή.

- **Το `/g` με κενή αντιστοιχία προχωρά κατά έναν χαρακτήρα** για να αποφευχθεί άπειρος βρόχος: το `s/\b/-/g` εισάγει μεταξύ κάθε λεκτικού, δεν κολλάει στην πρώτη θέση.
- **Το `!~` αντιστρέφει την επιστροφή**, όχι την εργασία. Το `$x !~ s/A/a/g` εξακολουθεί να τροποποιεί το `$x`· απλώς επιστρέφει αληθές όταν δεν βρέθηκε κανένα A.
- **Η αντιστοίχιση οριοθετών για αγκύλες** χρησιμοποιεί τον κατοπτρικό αντίστοιχο: `s{pat}{rep}`, `s<pat><rep>`. Ένας οριοθέτης διαφορετικός από αγκύλη πρέπει να εμφανιστεί τρεις φορές συνολικά: `s/pat/rep/`.
- **Το `/c` είναι no-op στο `s///`**. Γίνεται δεκτό για συντακτική συμμετρία με τη `m//`· η χρήση του παράγει προειδοποίηση υπό το `use warnings` και τίποτε άλλο.
- **Ιδιωματισμός ανάθεσης-ως-στόχου.** Το `($copy = $src) =~ s/x/y/g` γεμίζει το `$copy` με την τροποποιημένη τιμή και αφήνει το `$src` ως έχει. Ο σύγχρονος κώδικας προτιμά το `my $copy = $src =~ s/x/y/gr`.
- **Στόχοι tied ή υπερφορτωμένοι.** Χωρίς το `/r`, καλείται η `STORE` μετά την αντικατάσταση για να γραφεί η νέα τιμή πίσω· η ανακτημένη τιμή αντιγράφεται σε απλό ενταμιευτή πριν εφαρμοστεί το μοτίβο. Με το `/r` το αποτέλεσμα είναι πάντα απλή συμβολοσειρά - η tie / υπερφόρτωση δεν διατηρείται στην επιστροφή.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `m//` - ο τελεστής αντιστοίχισης· μοιράζεται τη σημασιολογία σημαιών πλευράς αντιστοίχισης και τους κανόνες επιστροφής έναντι περιβάλλοντος
- `tr///` - μεταγραμματισμός χαρακτήρα προς χαρακτήρα· πολύ ταχύτερος όταν η εργασία είναι κυριολεκτική μετάφραση χαρακτήρων, όχι regex
- `qr//` - προμεταγλωττίζει ένα μοτίβο μία φορά και το παρεμβάλλει σε πολλές κλήσεις `s///` ή `m//`· η σύγχρονη εναλλακτική για το `/o`
- `split` - όταν ο στόχος είναι το σπάσιμο μιας συμβολοσειράς σε ένα μοτίβο αντί για αντικατάσταση αντιστοιχιών, καταφύγετε στη `split`
- *Κεφάλαιο αντικατάστασης*
  - το κεφάλαιο αφιερωμένο στο `s///`· καλύπτει τα `/e`, `/r`, `/g`, `\K`, και τη διαχείριση αντιστοιχιών μηδενικού μήκους
- *Regular expressions guide*
  - η πλήρης αναφορά της γλώσσας μοτίβων· κάθε σημαία πλευράς αντιστοίχισης στο `s///` ορίζεται εκεί
- `$&` - το κείμενο που ταίριαξε, διαθέσιμο μέσα στο `REPLACEMENT` κάθε `s///`

I/O

say

Εκτυπώνει λίστα τιμών ακολουθούμενη από newline.

Η `say` είναι `print` με τον διαχωριστή εγγραφών εξόδου εξαναγκασμένο σε `"\n"` για εκείνη τη μία κλήση. Μετατρέπει κάθε αντικείμενο της `LIST` σε συμβολοσειρά, τα ενώνει με την `$,` (διαχωριστής πεδίων εξόδου, συνήθως κενός), τα γράφει στο `FILEHANDLE`, και προσαρτά κυριολεκτικό `"\n"` - **όχι** την τιμή της `$\`. Αν το `FILEHANDLE` παραλείπεται, η έξοδος πηγαίνει στο τρέχον επιλεγμένο handle (δείτε την `select`), που έχει προεπιλογή `STDOUT`. Αν η `LIST` παραλείπεται, εκτυπώνεται η `$_`.

Η `say` είναι ενσωματωμένη feature-gated. Είναι διαθέσιμη μόνο όταν το χαρακτηριστικό `"say"` είναι ενεργοποιημένο - αυτόματα υπό `use v5.10` ή υψηλότερα, ή ρητά με `use feature 'say'` - ή όταν καλείται ως `CORE::say`.

Σύνοψη

```
say LIST
say FILEHANDLE LIST
say {EXPR} LIST
say
```

Τι επιστρέφεται

1 σε επιτυχία, `undef` σε αποτυχία (με την `$!` τεθειμένη). Όπως και με την `print`, η τιμή επιστροφής σπάνια ελέγχεται για `STDOUT` ή `STDERR`. Ελέγξτε την όταν γράφετε σε αρχείο, σωλήνα ή υποδοχή όπου αποτυχία εγγραφής είναι ανακτήσιμη κατάσταση:

```
say $fh $record
    or die "write to $path failed: $!";
```

Προεπιλεγμένο filehandle, προεπιλεγμένοι διαχωριστές

Η `say` διαβάζει την ίδια καθολική κατάσταση με την `print`, με μία σκόπιμη εξαίρεση:

- Το **τρέχον επιλεγμένο filehandle**, που αλλάζει με την `select`. Η `say LIST` γράφει εκεί όταν δεν δίνεται filehandle. Προεπιλογή `STDOUT`.
- `$,` - **διαχωριστής πεδίων εξόδου**, παρεμβαλλόμενος μεταξύ στοιχείων της λίστας. Προεπιλογή η κενή συμβολοσειρά, οπότε η `say 1, 2, 3` παράγει `"123\n"`, όχι `"1 2 3\n"`.
- `$\` - **διαχωριστής εγγραφών εξόδου**. Η `say` **αγνοεί την `$\` εντελώς** και αντ” αυτού προσαρτά κυριολεκτικό `"\n"`. Η ρύθμιση της `$\` δεν έχει επίδραση στην `say`.

Αυτό το τελευταίο σημείο είναι ο όλος λόγος που υπάρχει η `say`: μία εγγραφή, ένα newline, χωρίς πειραματισμούς με την `$\`.

Παραδείγματα

Βασική εγγραφή στο STDOUT:

```
use feature 'say';
say "hello, world";                 # "hello, world\n"
```

Εγγραφή σε filehandle. Ίδια μορφή έμμεσου αντικειμένου με την `print` - **χωρίς κόμμα** μεταξύ του filehandle και της λίστας:

```
open my $fh, ">", "out.txt" or die $!;
say $fh "line 1";
say $fh "line 2";
close $fh;
```

Εγγραφή σε filehandle που κρατιέται σε έκφραση. Κάθε filehandle πιο σύνθετο από bareword ή απλό βαθμωτό χρειάζεται τη μορφή μπλοκ `{EXPR}`:

```
my @logs = (\*STDOUT, \*STDERR);
say { $logs[$level] } "message";
```

Η `say` ενώνει τα στοιχεία της λίστας με την `$,` και μετά προσαρτά `"\n"` ανεξαρτήτως της `$\`:

```
local $, = ", ";
local $\ = ".\n";                   # ignored by say
say "apples", "pears", "plums";     # "apples, pears, plums\n"
```

Χωρίς ορίσματα - εκτυπώνει την `$_` συν newline. Το filehandle πρέπει να είναι bareword σε αυτή τη μορφή· η `say $fh;` θα ανέλυε το `$fh` ως το περιεχόμενο, όχι ως τον στόχο:

```
for (qw(alpha beta gamma)) {
    say;                            # "alpha\n", "beta\n", "gamma\n"
}
```

Το περιβάλλον λίστας ισχύει για κάθε όρισμα:

```
my @arr = (10, 20, 30);
say @arr;                           # "102030\n"
say scalar @arr;                    # "3\n"
```

Οριακές περιπτώσεις

- **Πύλη χαρακτηριστικού**: χωρίς `use feature 'say'` ή `use v5.10`, η `say` αναλύεται ως κοινή κλήση υπορουτίνας και θα αποτύχει κατά τη μεταγλώττιση αν δεν υπάρχει τέτοια sub εντός εμβέλειας. Χρησιμοποιήστε `CORE::say(...)` για να την καλέσετε χωρίς συνθήκη.
- **Χωρίς ορίσματα με ρητό filehandle**: η `say FH;` εκτυπώνει την `$_` στο FH, αλλά μόνο για **bareword** filehandles. Η `say $fh;` αναλύει το `$fh` ως το περιεχόμενο, όχι τον στόχο - γράψτε `say $fh $_;` αν αυτό εννοείτε. Αυτό ταιριάζει με την `print`.

- **Η `$\` αγνοείται, δεν παρακάμπτεται**: η `say` δεν θέτει ούτε `local`οποιεί την `$\`. Άλλος κώδικας που διαβάζει την `$\` (συμπεριλαμβανομένων εμφωλευμένων κλήσεων `print`) βλέπει όποια τιμή ήταν ήδη εκεί.
- **Αμφισημία filehandle-ή-πρώτου-ορίσματος**: ίδιος κανόνας με την `print`. Ένα bareword ή απλό βαθμωτό που ακολουθείται αμέσως από όρο (χωρίς κόμμα) λαμβάνεται ως filehandle. Έκφραση που επιστρέφει filehandle χρειάζεται τη μορφή μπλοκ: `say { $handles[0] } "x";`.
- **Η `say (…)` αναλύεται ως κλήση συνάρτησης**. Η `say (2+3)*4` εκτυπώνει 5 και μετά απορρίπτει το `*4`. Βάλτε ολόκληρη τη λίστα ορισμάτων σε παρενθέσεις, ή προθέστε με `+`:

```
say((2+3)*4);                   # "20\n"
say +(2+3)*4;                   # "20\n"
```

- **Κλεισμένο filehandle**: επιστρέφει `undef` και θέτει την `$!` σε `"Bad file descriptor"`. Υπό `use warnings` εκπέμπεται προειδοποίηση `say() on closed filehandle`.
- **Στρώματα κωδικοποίησης**: η `say` γράφει μέσω της στοίβας PerlIO του handle ακριβώς όπως η `print`. Handle με στρώμα `:utf8` ή `:encoding(...)` κωδικοποιεί κατά την έξοδο· το προσαρτημένο newline είναι ένα μόνο byte `"\n"` και κωδικοποιείται από το ίδιο στρώμα. Συμβολοσειρά με ευρείς χαρακτήρες που γράφεται σε handle σε λειτουργία bytes πυροδοτεί προειδοποίηση `Wide character in say`.
- **Autoflush**: η `say` ενταμιεύει μέσω PerlIO όπως η `print`. Θέστε την `$|` στο επιλεγμένο handle για έξοδο ενταμιευμένη γραμμή-γραμμή.
- **`SIGPIPE`**: η εγγραφή σε κλειστό σωλήνα ή υποδοχή εγείρει `SIGPIPE`. Η προεπιλεγμένη ενέργεια τερματίζει τη διεργασία· εγκαταστήστε χειριστή για να το μετατρέψετε σε κανονική αποτυχία εγγραφής.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `print` - ίδιοι κανόνες στόχευσης και διαχωριστή, αλλά προσαρτά την `$\` (κενή προεπιλεγμένα) αντί για εξαναγκασμένο newline
- `printf` - μορφοποιημένη έξοδος· δεν προσαρτά newline, το βάζετε εσείς στη συμβολοσειρά μορφής
- `select` - αλλάζει σε ποιο filehandle γράφει η μορφή χωρίς όρισμα της `say`
- `$,` - διαχωριστής πεδίων εξόδου που παρεμβάλλεται μεταξύ στοιχείων λίστας· η `say` τον τιμά
- `$\` - διαχωριστής εγγραφών εξόδου· η `say` **τον αγνοεί** και αντ” αυτού προσαρτά `"\n"`
- `$|` - σημαία autoflush για το τρέχον επιλεγμένο handle

Διάφορα

scalar

Εξαναγκάζει την αποτίμηση της EXPR σε βαθμωτό περιβάλλον και επιστρέφει την τιμή της.

Η `scalar` δεν είναι στην πραγματικότητα τελεστής με δική της υπόσταση - είναι σημάδι εξαναγκασμού περιβάλλοντος που ο αναλυτής αναγνωρίζει μπροστά από μια μοναδιαία έκφραση. Η Perl αποτιμά την έκφραση σε βαθμωτό περιβάλλον και αποδίδει ό,τι παράγει αυτή η έκφραση εκεί. Η συνάρτηση υπάρχει επειδή οι περισσότερες θέσεις σε ένα πρόγραμμα Perl επιβάλλουν ήδη ένα περιβάλλον (λίστας ή βαθμωτό) στους τελεστές τους, και περιστασιακά χρειάζεται να παρακάμψετε το περιβάλλον λίστας με βαθμωτό περιβάλλον ρητά.

Σύνοψη

```
scalar EXPR
scalar @array
scalar %hash
scalar localtime
```

Τι επιστρέφεται

Η τιμή της EXPR όπως αποτιμάται σε βαθμωτό περιβάλλον. Το ακριβές σχήμα εξαρτάται εξ ολοκλήρου από το τι είναι η EXPR:

- Για πίνακα, ο μετρητής στοιχείων.
- Για πίνακα κατακερματισμού, αληθής τιμή (μη κενός) ή `0` (κενός)· σε παλαιότερες Perl αυτή ήταν συμβολοσειρά χρήσης κάδων, αλλά από την 5.26 και μετά είναι ο μετρητής κλειδιών.
- Για συνάρτηση ευαίσθητη στο περιβάλλον λίστας, ό,τι επιστρέφει αυτή η συνάρτηση όταν καλείται σε βαθμωτό περιβάλλον (για παράδειγμα, η `localtime` επιστρέφει τη συμβολοσειρά τύπου ctime αντί για τη λίστα εννέα στοιχείων).
- Για ήδη βαθμωτή έκφραση, η ίδια τιμή - η `scalar` είναι no-op εκεί.

Καθολική κατάσταση που επηρεάζει

Καμία. Η `scalar` είναι καθαρή μετατροπή περιβάλλοντος· δεν διαβάζει ούτε γράφει δικές της ειδικές μεταβλητές. Η *έκφραση* που περνάτε μπορεί φυσικά να αγγίζει όποιες καθολικές κανονικά αγγίζει.

Παραδείγματα

Μέτρηση των στοιχείων ενός πίνακα μέσα σε θέση περιβάλλοντος λίστας:

```
my @items = (10, 20, 30);
my @counts = (scalar @items);       # (3)
print scalar @items, "\n";          # 3
```

Χωρίς τη `scalar`, ο `@items` θα αναπτυσσόταν στα στοιχεία του και ο `@counts` θα ήταν `(10, 20, 30)`.

Εκτυπώστε την αναγνώσιμη από άνθρωπο ημερομηνία από την `localtime`. Σε περιβάλλον λίστας η `localtime` επιστρέφει εννέα πεδία· σε βαθμωτό περιβάλλον επιστρέφει την οικεία συμβολοσειρά ctime:

```
print localtime, "\n";              # nine numbers run together
print scalar localtime, "\n";       # "Thu Apr 23 14:02:11 2026"
```

Η ίδια παγίδα εμφανίζεται οπουδήποτε μια ενσωματωμένη είναι ευαίσθητη στο περιβάλλον λίστας:

```
my @parts = localtime;              # (sec, min, hour, mday, ...)
my $now   = scalar localtime;       # ctime string
```

Ελέγξτε αν ένας πίνακας κατακερματισμού είναι μη κενός χωρίς να κατανείμετε τη λίστα κλειδιών του:

```
if (scalar %config) {
    # %config has at least one key
}
```

Το `if (%config)` επιβάλλει ήδη βαθμωτό (λογικό) περιβάλλον, οπότε η `scalar` εδώ είναι περιττή αλλά ρητή· οι συγγραφείς μερικές φορές την περιλαμβάνουν για να γίνει η πρόθεση ορατή στους αναγνώστες.

Δομήστε μια λίστα μετρητών από πολλούς πίνακες σε μία έκφραση:

```
my @counts = (scalar @a, scalar @b, scalar @c);
```

Εξαναγκάστε μια κλήση συνάρτησης που διαφορετικά θα «έσπαγε» σε περιβάλλουσα λίστα:

```
my @row = ($id, scalar get_tags($id), $mtime);
```

Χωρίς τη `scalar`, η `get_tags` θα έτρεχε σε περιβάλλον λίστας και η επιστρεφόμενη λίστα της θα ισοπεδωνόταν μέσα στον `@row` δίπλα στα `$id` και `$mtime`.

Οριακές περιπτώσεις

- **Δεν υπάρχει αντίστροφος για περιβάλλον λίστας.** Δεν υπάρχει τελεστής `list EXPR`· το περιβάλλον λίστας είναι προεπιλεγμένο σχεδόν σε κάθε θέση όπου θα θέλατε να το επιβάλετε. Αν χρειάζεστε πραγματικά να εξαναγκάσετε μια βαθμωτή έκφραση σε περιβάλλον λίστας, το ιδίωμα είναι `@{[ EXPR ]}`, αν και στις περισσότερες περιπτώσεις αρκεί απλό `(EXPR)`.
- **Παγίδα παρενθετικής λίστας.** Η `scalar` είναι μοναδιαίος τελεστής. Το να της περάσετε μια παρενθετική έκφραση κόμματος **δεν** αποτιμά αυτή την έκφραση ως λίστα· την αποτιμά ως *βαθμωτό τελεστή κόμματος*, που εκτελεί κάθε υπο-έκφραση σε κενό περιβάλλον εκτός από την τελευταία και επιστρέφει την τελευταία σε βαθμωτό περιβάλλον:

  ```
  print uc(scalar(foo(), $bar)), $baz;
  # equivalent to:
  ```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
foo();                                  # void context
print(uc($bar), $baz);                  # $bar in scalar context
```

Αν θέλατε να συμπεριληφθούν οι τιμές επιστροφής της `foo()`, η `scalar (list)` είναι το λάθος εργαλείο.

- **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 5.44 and so always returns the key count.
- **Πρωτότυπα υπορουτινών.** Μια υπορουτίνα που δηλώθηκε με βαθμωτό πρωτότυπο (`($)`) επιβάλλει ήδη βαθμωτό περιβάλλον στο όρισμά της, οπότε η `scalar` είναι περιττή μπροστά από την κλήση. Πρωτότυπο `(@)` ή `(\@)` δεν επιβάλλει, οπότε η `scalar` εξακολουθεί να έχει σημασία:

```
sub count ($) { $_[0] }
count @items;                           # @items in scalar context:
↪3
```

- **Περιττή σε ήδη βαθμωτές εκφράσεις.** Οι `scalar $x`, `scalar 42`, `scalar "text"` είναι no-ops. Η Perl δεν προειδοποιεί για αυτές· χρησιμοποιούνται περιστασιακά για έμφαση αλλά δεν προσθέτουν τίποτα.
- **Δεν είναι κλήση συνάρτησης.** Επειδή η `scalar` είναι σημάδι περιβάλλοντος σε επίπεδο αναλυτή, δεν μπορείτε να πάρετε `\&scalar`, δεν μπορείτε να την παρακάμψετε με `sub scalar { ... }`, και δεν μπορείτε να την περάσετε ως αναφορά κώδικα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `wantarray` - επιθεωρήστε το περιβάλλον στο οποίο κληθήκατε, αντί να επιβάλετε ένα σε υπο-έκφραση
- `defined` - ομοίως μοναδιαίος τελεστής που εφαρμόζεται σε έκφραση· χρήσιμη παράλληλα με τη `scalar` όταν ελέγχετε αποτελέσματα συναρτήσεων
- `localtime` - κανονικό παράδειγμα ενσωματωμένης της οποίας το σχήμα επιστροφής εξαρτάται από το περιβάλλον
- `reverse` - επίσης ευαίσθητη στο περιβάλλον· η `scalar reverse $s` αντιστρέφει χαρακτήρες, η `reverse @a` αντιστρέφει λίστα
- `keys` - η `scalar keys %h` είναι η εκτεταμένη μορφή της `scalar %h` με το ίδιο αποτέλεσμα στην 5.26+

I/O

seek

Επανατοποθετεί ένα filehandle για αναγνώσεις ή εγγραφές τυχαίας πρόσβασης.

Η `seek` μετακινεί τον δείκτη ανάγνωσης/εγγραφής του `FILEHANDLE` σε νέο offset bytes, αντικατοπτρίζοντας την κλήση C `fseek(3)`. Μετά από επιτυχημένη `seek`, η επόμενη `read`, `readline` ή `print` επί του handle ξεκινά από τη νέα θέση. Το `POSITION` είναι προσημασμένο offset bytes· το `WHENCE` επιλέγει το σημείο αναφοράς από το οποίο μετράται.

Σύνοψη

```
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
```

Τι επιστρέφεται

1 σε επιτυχία, ψευδής τιμή σε αποτυχία (με το `$!` τεθειμένο). Ελέγχετε πάντοτε την τιμή επιστροφής - η μετάβαση πέρα από κοντό αρχείο, σε μη-seekable handle (σωλήνας, υποδοχή, TTY), ή μετά από σφάλμα εγγραφής αποτυγχάνουν όλες εδώ αντί στην επόμενη ανάγνωση.

```
seek $fh, $offset, 0
    or die "seek to $offset failed: $!";
```

Τιμές WHENCE

Το `WHENCE` είναι ακέραιος με τρεις νοηματικές τιμές. Χρησιμοποιήστε τις συμβολικές σταθερές από το `Fcntl` για αναγνωσιμότητα:

- `0` / `SEEK_SET` - το `POSITION` μετράται από την **αρχή του αρχείου**. Το `POSITION` πρέπει να είναι μη αρνητικό.
- `1` / `SEEK_CUR` - το `POSITION` προστίθεται στην **τρέχουσα θέση**. Αρνητικές τιμές μετακινούν προς τα πίσω, θετικές προς τα εμπρός. Το `seek $fh, 0, 1` είναι το κανονικό ιδίωμα «μη μετακινείσαι, αλλά καθάρισε το EOF».
- `2` / `SEEK_END` - το `POSITION` προστίθεται στο offset **τέλους αρχείου**. Το `POSITION` είναι τυπικά μηδέν ή αρνητικό.

```
use Fcntl qw(SEEK_SET SEEK_CUR SEEK_END);

seek $fh, 0,     SEEK_SET;   # rewind
seek $fh, 0,     SEEK_END;   # go to EOF (e.g. to append)
seek $fh, -$n,   SEEK_CUR;   # $n bytes back from here
```

Bytes, όχι χαρακτήρες

Ακόμη και όταν το handle έχει στρώμα προσανατολισμένο σε χαρακτήρες όπως `:encoding(UTF-8)`, οι `seek`, `tell` και η οικογένεια της `sysseek` λειτουργούν επί **offsets bytes**. Μια `seek` σε offset bytes που προσγειώνεται στη μέση πολυβυτικής ακολουθίας θα παραγάγει σφάλματα αποκωδικοποίησης ή χαρακτήρες αντικατάστασης στην επόμενη ανάγνωση.

Αν χρειάζεται να τοποθετηθείτε κατά χαρακτήρες, διαβάστε προς τα εμπρός από γνωστό όριο bytes αντί να προσπαθήσετε να μεταφράσετε καταμετρήσεις χαρακτήρων σε offsets bytes. Οι τιμές της `tell` είναι ασφαλές να τροφοδοτηθούν πίσω στη `seek` επειδή παράχθηκαν σε όρια bytes που το στρώμα I/O είχε ήδη διασχίσει καθαρά.

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται στο μήνυμα σφάλματος συστήματος σε αποτυχία.
- Ο ενταμιευτής PerlIO του filehandle απορρίπτεται σε επιτυχία, οπότε όποια δεδομένα έχουν προαναγνωσθεί από την ενταμίευση χάνονται και η επόμενη ανάγνωση τραβά φρέσκα bytes από το υποκείμενο αρχείο.
- Η σημαία τέλους αρχείου επί του `FILEHANDLE` καθαρίζεται σε επιτυχία, ακόμη και όταν τα `POSITION` και `WHENCE` αφήνουν τη θέση αμετάβλητη.

Παραδείγματα

Επαναφορά στην αρχή αρχείου πριν την επανανάγνωσή του:

```
seek $fh, 0, 0 or die "rewind failed: $!";
while (my $line = <$fh>) { ... }
```

Προσθήκη με τοποθέτηση στο τέλος αρχείου, και μετά εγγραφή. Για χρήση μόνο προσθήκης, προτιμήστε άνοιγμα με >> - αυτή η μορφή είναι για handles ήδη ανοιγμένα για ανάγνωση-εγγραφή:

```
seek $fh, 0, 2 or die $!;           # SEEK_END
print $fh "appended line\n";
```

Ανάγνωση εγγραφής σταθερού πλάτους κατά δείκτη, όπου κάθε εγγραφή είναι 128 bytes:

```
my $record = 42;
seek $fh, $record * 128, 0 or die $!;
read $fh, my $buf, 128;
```

Εξομοίωση `tail -f` - η `seek $fh, 0, 1` επαναφέρει τη συνθήκη EOF ώστε η επόμενη `readline` να ξαναδοκιμάσει το αρχείο για νέα δεδομένα:

```
while (1) {
    while (my $line = <$fh>) { print $line }
    sleep 1;
    seek $fh, 0, 1;                 # clear EOF, keep position
}
```

Αποθηκεύστε μια θέση με `tell`, διαβάστε προς τα εμπρός, και έπειτα αποκαταστήστε:

```
my $mark = tell $fh;
my $peek = <$fh>;
seek $fh, $mark, 0 or die $!;       # back to where we were
```

Οριακές περιπτώσεις

- **Μη-seekable handles**: σωλήνες, υποδοχές, TTYs και οι περισσότερες ειδικές συσκευές επιστρέφουν ψευδές και θέτουν το `$!` σε `ESPIPE` (`Illegal seek`). Ελέγξτε την τιμή επιστροφής· μη θεωρείτε δεδομένο ότι κάθε filehandle είναι seekable.
- **Ανάμειξη αναγνώσεων και εγγραφών στο ίδιο handle** σε αρχείο ανοιγμένο με `+<` ή `+>` απαιτεί μια `seek` (ή `tell`, ή ρητή εκκένωση) κατά την αλλαγή κατεύθυνσης. Ένα WHENCE `1` με `POSITION` `0` είναι ο συνηθισμένος no-op διαχωριστής:

  ```
  seek $fh, 0, 1;                    # allowed to switch read <->␣
  ↪write
  ```

- **Μετάβαση πέρα από το EOF** σε αρχείο ανοιγμένο για εγγραφή είναι νόμιμη και δημιουργεί αραιή τρύπα έως το `POSITION` σε filesystems που υποστηρίζουν τρύπες· η επόμενη εγγραφή γεμίζει μέρος της τρύπας και τα bytes ενδιάμεσα διαβάζονται πίσω ως `"\0"`.
- **Χρήστες `sysread` / `syswrite`**: **μην** αναμειγνύετε τη `seek` με `sysread` ή `syswrite`. Η `seek` λειτουργεί στο στρώμα ενταμιευτή PerlIO· η μη ενταμιευμένη sys-οικογένεια το παρακάμπτει, οπότε οι ενεργές τους θέσεις αποκλίνουν. Χρησιμοποιήστε την `sysseek` για handles που προσπελάζετε μέσω της sys-οικογένειας.
- **Offsets χαρακτήρων μέσω αριθμητικής**: ο πολλαπλασιασμός καταμέτρησης χαρακτήρων με υποτιθέμενο πλάτος κωδικοποίησης είναι εσφαλμένος για `UTF-8` και άλλες κωδικοποιήσεις μεταβλητού μήκους. Χρησιμοποιήστε offsets bytes που λαμβάνονται από την `tell`, ή διαβάστε προς τα εμπρός από γνωστό όριο.
- **Handles καταλόγου**: η `seek` δεν λειτουργεί σε handles καταλόγου. Χρησιμοποιήστε `seekdir` με θέση από την `telldir`.
- **Κλειστό filehandle**: επιστρέφει ψευδή τιμή και θέτει το `$!`· υπό `use warnings` εκπέμπεται προειδοποίηση `seek() on closed filehandle`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `tell` - διαβάστε το τρέχον offset bytes· η τιμή κάνει πλήρη κύκλο με ασφάλεια μέσω της `seek` με `WHENCE` `0`
- `sysseek` - μη ενταμιευμένη seek για handles που προσπελάζονται μέσω `sysread` / `syswrite`· χρησιμοποιήστε αυτή αντί της `seek` όταν παρακάμπτετε το PerlIO
- `read` - ενταμιευμένη ανάγνωση σταθερού μήκους· ο φυσικός συνεργάτης για τοποθέτηση με offset bytes

- `readline` - γραμμοκεντρική ανάγνωση· η `seek $fh, 0, 1` πριν από επανάληψη είναι το ιδίωμα `tail -f`
- `eof` - δοκιμάστε για τέλος αρχείου· η `seek` καθαρίζει τη σημαία EOF που μπορεί να έθεσε προηγούμενη ανάγνωση
- `Fcntl` - πηγή των σταθερών `SEEK_SET`, `SEEK_CUR`, `SEEK_END`

I/O

seekdir

Επαναφέρει ένα handle καταλόγου σε μια θέση που είχε προηγουμένως αποτυπωθεί από την `telldir`.

Η `seekdir` μετακινεί τον δρομέα ανάγνωσης του `DIRHANDLE` στο `POS`, ώστε η επόμενη κλήση `readdir` να επιστρέψει την εγγραφή που θα είχε επιστραφεί όταν το `POS` αποκτήθηκε για πρώτη φορά από την `telldir` στο ίδιο handle. Είναι το αντίστοιχο για κατάλογο της `seek` σε file handle - με τον κρίσιμο περιορισμό ότι το `POS` δεν είναι byte-offset που μπορείτε να υπολογίσετε, είναι ένα αδιαφανές cookie που πρέπει να έχετε λάβει νωρίτερα από την `telldir`.

Σύνοψη

```
seekdir DIRHANDLE, POS
```

Τι επιστρέφεται

Καμία ουσιαστική τιμή. Η `seekdir` επιστρέφει ό,τι επιστρέφει η υποκείμενη `seekdir(3)`, το οποίο στο Linux δεν είναι τίποτα χρήσιμο - **μην** διακλαδώνετε με βάση την τιμή επιστροφής. Για να ανιχνεύσετε πρόβλημα, ελέγξτε το `$!` μετά την κλήση, ή επαληθεύστε ότι το handle είναι ακόμη ανοιχτό πριν την κλήση.

Τυπική χρήση

Διασχίζετε έναν κατάλογο, παρατηρείτε μια εγγραφή που θέλετε να επανεπισκεφθείτε αργότερα, και θυμάστε πού βρισκόσασταν ώστε να μπορείτε να επιστρέψετε σε αυτή αφού συνεχίσετε την ανάγνωση:

```
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\z/;
}

# later: re-read the entry that follows the first .conf file
seekdir $dh, $bookmarks{(sort keys %bookmarks)[0]};
my $next = readdir $dh;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
closedir $dh;
```

Η επιστροφή στην αρχή είναι η συνηθισμένη ειδική περίπτωση - μια κλήση `seekdir $dh, 0` είναι ισοδύναμη με την `rewinddir` σε συμμορφωμένα συστήματα, αλλά η `rewinddir` είναι η ιδιωματική γραφή για αυτή την πρόθεση:

```
seekdir $dh, 0;     # works, but say what you mean:
rewinddir $dh;      # clearer
```

Οριακές περιπτώσεις

- **Το POS πρέπει να προέρχεται από την `telldir` στο ίδιο handle.** Το πέρασμα ενός ακεραίου που επινοήσατε, μιας τιμής από διαφορετικό handle, ή μιας μπαγιάτικης τιμής από handle που έκτοτε έκλεισε και ξανά-ανοίχτηκε είναι απροσδιόριστη συμπεριφορά. Ο πυρήνας μπορεί να την αποδεχτεί σιωπηλά και να σας δώσει σκουπίδια στο επόμενο `readdir`, να παραλείψει εγγραφές, ή να επιστρέψει διπλότυπα.
- **Συμπίεση καταλόγου μεταξύ `telldir` και `seekdir`**: αν το σύστημα αρχείων έχει συμπιέσει ή αναδιοργανώσει τον κατάλογο μεταξύ των δύο κλήσεων (αρχεία που δημιουργήθηκαν, διαγράφηκαν, ή μετονομάστηκαν στον ίδιο κατάλογο από οποιαδήποτε διεργασία), το cookie POS ενδέχεται να μην αναφέρεται πλέον στην ίδια εγγραφή - ή σε καμία εγγραφή. Αυτή είναι ιδιότητα της υποκείμενης `seekdir(3)` και όχι περιορισμός του pperl. Για σταθερή επανάληψη πάνω σε κατάλογο που αλλάζει, λάβετε στιγμιότυπο της λίστας με `readdir` σε περιβάλλον λίστας και δουλέψτε στο αντίγραφο.
- **Μη ανοιγμένο ή κλειστό handle**: θέτει το `$!`. Υπό `use warnings` λαμβάνετε την προειδοποίηση `seekdir() attempted on invalid dirhandle`. Τίποτα στην τιμή επιστροφής δεν σας λέει ότι αυτό συνέβη - ελέγξτε το `$!` ή χρησιμοποιήστε `defined fileno($dh)` / ρητή λογιστική αν χρειάζεστε να προστατευτείτε.
- **Δεν είναι byte-offset**: το POS είναι αδιαφανές διακριτικό. Η αριθμητική πάνω σε αυτό (`seekdir $dh, $pos + 1`) δεν έχει νόημα· οι μόνες ασφαλείς τιμές είναι το `0` (αρχή καταλόγου, και ακόμη και τότε προτιμήστε `rewinddir`) και οι τιμές που επιστρέφονται από την `telldir`.
- **Η σειρά εγγραφών δεν είναι προδιαγεγραμμένη**: η `seekdir` αναπαράγει όποια σειρά επέστρεψε αρχικά ο πυρήνας. Αυτή η σειρά δεν είναι ταξινομημένη, ούτε αλφαβητική, και δεν εγγυάται ότι θα ταιριάξει με ένα φρέσκο `opendir` της ίδιας διαδρομής αργότερα. Αν ο κώδικάς σας ταξινομεί, ταξινομείτε ρητά κάθε φορά.
- **Σχετικές αναζητήσεις τύπου SEEK_CUR δεν υπάρχουν** για handles καταλόγου. Δεν υπάρχει ισοδύναμο του ορίσματος whence της `seek`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `telldir` - παράγει το cookie POS που καταναλώνει η `seekdir`· οι δύο είναι χρήσιμες μόνο ως ζεύγος
- `readdir` - διαβάζει την επόμενη εγγραφή· συνεχίζει από τη θέση που μόλις έθεσε η `seekdir`
- `rewinddir` - ο ιδιωματικός τρόπος για επιστροφή στην αρχή ενός καταλόγου· προτιμήστε την έναντι του `seekdir $dh, 0`
- `opendir` - ανοίγει τον κατάλογο και αποκτά το handle στο οποίο λειτουργεί η `seekdir`
- `seek` - το ανάλογο για file handle, αλλά με πραγματικό byte-offset και όρισμα `whence`· τα handles καταλόγου δεν έχουν ούτε το ένα ούτε το άλλο

I/O · Filehandles, αρχεία, κατάλογοι

select

Είτε ορισμός του προεπιλεγμένου filehandle εξόδου, είτε κλήση της κλήσης συστήματος `select(2)` για πολυπλεξία I/O. Ίδιο όνομα, δύο ασυσχέτιστες δουλειές - διακρίνονται από τον αριθμό ορισμάτων.

Η μορφή ενός ορίσματος (ή μηδέν ορισμάτων) αφορά πού στέλνουν την έξοδό τους οι `print`, `printf`, `say` και `write` όταν δεν ονοματίζετε filehandle. Η μορφή τεσσάρων ορισμάτων είναι άμεσο περιτύλιγμα γύρω από την κλήση συστήματος BSD `select(2)` για αναμονή σε σύνολα περιγραφέων αρχείου.

Σύνοψη

```
select FILEHANDLE              # set default output handle, return␣
↪old
select                         # return current default output␣
↪handle
select RBITS, WBITS, EBITS, TIMEOUT   # select(2) syscall
```

Τι επιστρέφεται

Η `select FILEHANDLE` επιστρέφει το *προηγουμένως* επιλεγμένο filehandle (ως typeglob), οπότε το ιδιωματικό μοτίβο save/restore λειτουργεί:

```
my $old = select $fh;
$| = 1;
select $old;
```

Η `select` (χωρίς ορίσματα) επιστρέφει το τρέχον επιλεγμένο filehandle. Κατά την εκκίνηση του προγράμματος αυτό είναι το STDOUT.

Η `select RBITS, WBITS, EBITS, TIMEOUT` επιστρέφει διαφορετικά πράγματα σε διαφορετικά περιβάλλοντα:

- Σε βαθμωτό περιβάλλον: ο αριθμός των έτοιμων filehandles, ή -1 σε σφάλμα με την `$!` ορισμένη.

- Σε περιβάλλον λίστας: `($nfound, $timeleft)` - το ίδιο `$nfound` συν τον υπολειπόμενο χρόνο από το TIMEOUT. Πολλοί πυρήνες δεν ενημερώνουν το `$timeleft`· σε αυτά τα συστήματα είναι πάντα ίσο με το είσοδο TIMEOUT.

Στη λήξη του timeout το `$nfound` είναι `0`. Οι τρεις bit masks (RBITS, WBITS, EBITS) ενημερώνονται επιτόπου ώστε να υποδείξουν ποιοι περιγραφείς είναι έτοιμοι - περάστε αντίγραφα αν θέλετε να διατηρήσετε τα αρχικά masks.

Καθολική κατάσταση που επηρεάζει

Η μορφή ενός ορίσματος αλλάζει ένα κομμάτι καθολικής κατάστασης διερμηνέα: το τρέχον επιλεγμένο filehandle εξόδου. Όσο αυτό το handle είναι επιλεγμένο, αυτές οι μεταβλητές αναφέρονται σε **αυτό**, όχι στο STDOUT:

- `$|` - σημαία αυτόματης εκκένωσης
- `$,` - διαχωριστικό πεδίου εξόδου
- `$\` - διαχωριστικό εγγραφής εξόδου
- `$#`, `$%`, `$=`, `$-`, `$~`, `$^` - μεταβλητές σχετικές με formats

Γι" αυτό το `my $old = select($fh); $| = 1; select($old);` λειτουργεί: το `$|` είναι *ιδιότητα του επιλεγμένου handle*, όχι καθολικό βαθμωτό.

Η μορφή τεσσάρων ορισμάτων δεν αγγίζει κάποια κατάσταση ανά διερμηνέα. Μπορεί να αφήσει την `$!` ορισμένη σε σφάλμα.

Παραδείγματα

Ενεργοποίηση αυτόματης εκκένωσης σε ένα handle χωρίς ενόχληση του προεπιλεγμένου:

```
my $old = select $fh; $| = 1; select $old;
```

Προσωρινή ανακατεύθυνση της `print` και αποκατάσταση στην έξοδο από το μπλοκ:

```
{
    my $old = select $log_fh;
    print "entry: $msg\n";       # goes to $log_fh
    select $old;
}
```

Αναμονή για 250 χιλιοστά του δευτερολέπτου χρησιμοποιώντας τη μορφή τεσσάρων ορισμάτων:

```
select undef, undef, undef, 0.25;
```

Αναμονή έως 5 δευτερολέπτων μέχρι το STDIN να γίνει αναγνώσιμο:

```
my $rin = '';
vec($rin, fileno(STDIN), 1) = 1;
my $nfound = select my $rout = $rin, undef, undef, 5;
if ($nfound) {
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    my $line = <STDIN>;
}
```

Κατασκευή read mask πάνω σε πολλαπλά 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;
```

Οριακές περιπτώσεις

- **Εναλλακτική σε στυλ μεθόδου.** Το `$fh->autoflush(1)` θέτει τη σημαία αυτόματης εκκένωσης στο `$fh` χωρίς να αλλάζει το προεπιλεγμένο handle. Προτιμήστε το όταν χρειάζεται μόνο να αλλάξετε μία ιδιότητα ενός handle - καμία καθολική κατάσταση δεν εμπλέκεται.
- **Η εντοπιοποίηση typeglob είναι καθαρότερη για ανακατευθύνσεις με εμβέλεια.** Το `local *STDOUT = $fh` επανασυνδέει το ίδιο το `STDOUT` για τη δυναμική εμβέλεια· σε `die` το αρχικό αποκαθίσταται αυτόματα. Η `select` **δεν** αποκαθιστά σε `die`. Τα δύο δεν είναι ισοδύναμα: με τη `select` μια ρητή `print STDOUT ...` εξακολουθεί να φτάνει στο *πραγματικό* `STDOUT`· με τη μορφή `local *STDOUT`, ακόμη και η `print STDOUT ...` πάει στο επανασυνδεδεμένο handle.
- **Η μορφή bit-mask απαιτεί bytes, όχι λίστα ακεραίων fileno.** Ένα bit mask είναι πακεταρισμένη bit string που κατασκευάζεται με `vec`· το πέρασμα λίστας ακεραίων ή αναφοράς παράγει σιωπηρά λάθος αποτελέσματα (ή κενό mask).
- **Το `undef` για mask σημαίνει «μην περιμένεις σε αυτό το σύνολο».** Περάστε `undef` για masks που δεν σας ενδιαφέρουν. Κενή συμβολοσειρά `''` τεχνικά λειτουργεί επίσης αλλά είναι λιγότερο ιδιωματική.
- **Τα κλασματικά timeouts λειτουργούν.** Το `TIMEOUT` είναι float σε δευτερόλεπτα. Το `0.25` είναι 250 ms· το `0` επιστρέφει άμεσα (polling).
- **Το `TIMEOUT` με `undef`** σημαίνει μπλοκάρισμα επ” αόριστον μέχρι τουλάχιστον ένας περιγραφέας να είναι έτοιμος ή να φτάσει σήμα.
- **Τα masks τροποποιούνται επιτόπου.** Κατά την επιστροφή, κάθε mask κρατά μόνο τα bits για τους έτοιμους περιγραφείς. Αν χρειάζεστε τα αρχικά αργότερα, περάστε αντίγραφα - το ιδίωμα `my $rout = $rin` στη γραμμή κλήσης κάνει ακριβώς αυτό.
- **Μην αναμειγνύετε ενταμιευμένο I/O με τη μορφή τεσσάρων ορισμάτων.** Η `select` αναφέρει ετοιμότητα σε επίπεδο πυρήνα· οι `readline`, `read`, και ο τελεστής διαμαντιού `<>` διαβάζουν μέσω buffers PerlIO που μπορεί ήδη να κρατούν μη διαβασμένα δεδομένα τα οποία ο πυρήνας δεν γνωρίζει. Χρησιμοποιήστε `sysread` για δεδομένα μόλις η `select` πει ότι ένας περιγραφέας είναι έτοιμος.
- **Πλασματική ετοιμότητα σε υποδοχές.** Σε ορισμένα Unix η `select(2)` μπορεί να αναφέρει μια υποδοχή ως «έτοιμη για ανάγνωση» όταν μια ακόλουθη `read` θα

μπλόκαρε ακόμα. Θέστε O_NONBLOCK στην υποδοχή αν αυτό έχει σημασία.

- **Η συμπεριφορά επανεκκίνησης μετά από σήμα εξαρτάται από το σύστημα.** Ένα σήμα (π.χ. SIGALRM) μπορεί να προκαλέσει την `select` να επιστρέψει νωρίς με `$! == EINTR`, ή ο πυρήνας μπορεί να επανεκκινήσει την κλήση διαφανώς. Φορητός κώδικας ελέγχει ρητά για EINTR.
- **Ο αριθμός ορισμάτων διακρίνει τις δύο μορφές.** Ένα όρισμα (ή κανένα) είναι η μορφή προεπιλεγμένου handle· ακριβώς τέσσερα είναι η μορφή κλήσης συστήματος. Άλλοι αριθμοί είναι συντακτικό σφάλμα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `print` - διαβάζει το επιλεγμένο handle όταν δεν δίνεται filehandle· η `select` είναι ο τρόπος να αλλάξετε ποιο handle είναι αυτό
- `printf` - ίδιος κανόνας προεπιλεγμένου handle με την `print`
- `write` - έξοδος βασισμένη σε format· οι μεταβλητές top-of-form και μήκους σελίδας ακολουθούν το επιλεγμένο handle, όχι το STDOUT
- `fileno` - αντιστοιχίζει ένα filehandle στον ακέραιο αριθμό περιγραφέα που χρειάζεστε κατά την κατασκευή ενός mask
- `vec` - πακετάρει αριθμούς περιγραφέων στο bit string που αναμένει η μορφή τεσσάρων ορισμάτων
- `sysread` - η μη ενταμιευμένη ανάγνωση που χρησιμοποιείτε αφού η `select` τεσσάρων ορισμάτων πει ότι ένας περιγραφέας είναι έτοιμος
- `IO::Select` - αντικειμενικό περιτύλιγμα πάνω από τη μορφή τεσσάρων ορισμάτων που αποκρύπτει την αριθμητική των bit-masks

SysV IPC

semctl

Εκτελεί μια λειτουργία ελέγχου σε σύνολο σηματοφόρων System V.

Η `semctl` είναι η σύνδεση σε επίπεδο Perl για την κλήση συστήματος `semctl(2)`. Διαβάζει ή τροποποιεί την κατάσταση του συνόλου σηματοφόρων που αναγνωρίζεται από το ID - λήψη ή ορισμός της τιμής ενός μεμονωμένου σηματοφόρου, ανάκτηση ή εγκατάσταση του πλήρους διανύσματος τιμών, ανάγνωση των συνημμένων μεταδεδομένων `semid_ds`, ή αφαίρεση ολόκληρου του συνόλου. Η CMD επιλέγει τη λειτουργία· το SEMNUM επιλέγει ποιος σηματοφόρος εντός του συνόλου (αγνοείται για εντολές σε όλο το σύνολο όπως IPC_STAT, IPC_RMID, GETALL, SETALL)· το ARG είναι ο ενταμιευτής εισόδου ή εξόδου της εντολής.

Η `semctl` είναι λεπτό wrapper - η σημασιολογία, οι σταθερές εντολών και το σχήμα του ARG προέρχονται όλα απευθείας από την κλήση επιπέδου C. Σχεδόν πάντοτε θα θέλετε:

```
use IPC::SysV;
```

πρώτα, για να εισαγάγετε τις σταθερές IPC_STAT, IPC_RMID, GETVAL, SETVAL, GETALL, SETALL, GETPID, GETNCNT, GETZCNT.

Σύνοψη

```
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
```

Τι επιστρέφεται

Η επιστροφή ακολουθεί την ίδια σύμβαση με την ioctl:

- undef σε σφάλμα (με το $! τεθειμένο).
- Τη συμβολοσειρά "0 but true" όταν η υποκείμενη κλήση επέστρεψε 0 - αυτή η συμβολοσειρά είναι ψευδής αριθμητικά (οπότε τυπώνει 0 σε αριθμητικό περιβάλλον) αλλά αληθής σε λογικό περιβάλλον, επιτρέποντάς σας να διακρίνετε την επιτυχία-με-μηδέν από την αποτυχία σε έναν έλεγχο.
- Την πραγματική ακέραια τιμή επιστροφής διαφορετικά.

Ο ιδιωματικός έλεγχος είναι επομένως:

```
my $rc = semctl($id, $n, GETVAL, 0);
defined $rc or die "semctl: $!";
```

Ελέγξτε defined, όχι αλήθεια - μια νόμιμα μηδενική τιμή σηματοφόρου θα αποτύγχανε σε απλό if ($rc).

Για τιμές CMD που διαβάζουν δεδομένα στο ARG (IPC_STAT, GETALL), το ARG πρέπει να είναι **τροποποιήσιμη lvalue** (απλή μεταβλητή, όχι κυριολεκτικό ή έκφραση)· σε επιτυχία ο ενταμιευτής αντικαθίσταται επιτόπου με τα επιστρεφόμενα bytes και κάνετε εσείς unpack.

Καθολική κατάσταση που επηρεάζει

Θέτει το $! σε αποτυχία. Καμία άλλη καθολική του διερμηνέα δεν εμπλέκεται.

Παραδείγματα

Διαβάστε την τρέχουσα τιμή ενός μεμονωμένου σηματοφόρου:

```
use IPC::SysV qw(GETVAL);
my $value = semctl($id, $semnum, GETVAL, 0);
defined $value or die "GETVAL: $!";
```

Θέστε έναν μεμονωμένο σηματοφόρο:

```
use IPC::SysV qw(SETVAL);
semctl($id, $semnum, SETVAL, 1)
    or die "SETVAL: $!";
```

Διαβάστε όλες τις τιμές σηματοφόρων ταυτόχρονα. Ο ενταμιευτής πρέπει να προεκχωρηθεί στο σωστό μέγεθος - nsems εγγενή shorts - πριν την κλήση. Η pack με ανενεργό μετρητή είναι ο συνηθισμένος τρόπος:

```
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);
```

Εγκαταστήστε ένα πλήρες σύνολο τιμών:

```
use IPC::SysV qw(SETALL);
my $buf = pack("s!*", 1, 0, 0, 1);
semctl($id, 0, SETALL, $buf)
    or die "SETALL: $!";
```

Ανακτήστε τη δομή semid_ds για το σύνολο. Το ARG γράφεται επιτόπου με την πακεταρισμένη δομή· η διάταξή της είναι ειδική της πλατφόρμας, οπότε κάντε unpack με τη βοήθεια του IPC::Semaphore αντί χειρωνακτικά εκτός αν χρειάζεται:

```
use IPC::SysV qw(IPC_STAT);
my $ds = "";
semctl($id, 0, IPC_STAT, $ds)
    or die "IPC_STAT: $!";
```

Καταστρέψτε το σύνολο σηματοφόρων. Τα SEMNUM και ARG δεν χρησιμοποιούνται· περάστε 0:

```
use IPC::SysV qw(IPC_RMID);
semctl($id, 0, IPC_RMID, 0)
    or die "IPC_RMID: $!";
```

Οριακές περιπτώσεις

- **Το ARG πρέπει να είναι πραγματική μεταβλητή για εντολές ανάγνωσης.** Οι GETALL και IPC_STAT γράφουν στο ARG· το πέρασμα κυριολεκτικού (semctl($id, 0, GETALL, "")) ή έκφρασης μόνο για ανάγνωση είναι σφάλμα χρόνου εκτέλεσης. Τα υπάρχοντα περιεχόμενα της μεταβλητής αντικαθίστανται, δεν προστίθενται.
- **Το ARG πρέπει να είναι προδιαστασιολογημένο.** Ο πυρήνας γράφει ακριβώς nsems * sizeof(short) bytes για την GETALL, ή sizeof(struct semid_ds) για την IPC_STAT. Το πέρασμα μικρότερου ενταμιευτή είναι απροσδιόριστη συμπεριφορά στην υποκείμενη κλήση C· το pack("s!*", (0) x $nsems) είναι το τυπικό ιδίωμα για την GETALL.
- **Εγγενές short, όχι φορητό short.** Οι τιμές σηματοφόρου είναι C short (προσημασμένα, τυπικά 16-bit). Χρησιμοποιήστε s! στις pack / unpack - το

! επιλέγει τον εγγενή τύπο της πλατφόρμας. Το σκέτο `s` εξαναγκάζει 16-bit little-endian και θα παραγάγει εσφαλμένα αποτελέσματα σε συστήματα big-endian ή LP64-short.

- **Επιστροφές `"0 but true"`.** Η `GETVAL` σε σηματοφόρο του οποίου η τιμή είναι `0`, οι `GETPID`/`GETNCNT`/`GETZCNT` που επιστρέφουν μηδέν, και άλλες επιτυχίες με μηδενική τιμή επιστρέφουν όλες την κυριολεκτική συμβολοσειρά `"0 but true"`. Τα αριθμητικά περιβάλλοντα βλέπουν `0`· τα λογικά περιβάλλοντα βλέπουν αληθές. Χρησιμοποιήστε `defined` για να ελέγξετε αποτυχία.
- **Το `SEMNUM` αγνοείται για εντολές σε όλο το σύνολο.** Οι `IPC_STAT`, `IPC_RMID`, `SETALL`, `GETALL` ενεργούν επί ολόκληρου του συνόλου. Περάστε `0` συμβατικά· το πέρασμα οποιασδήποτε άλλης τιμής είναι αβλαβές αλλά συσκοτίζει την πρόθεση.
- **Μετά την `IPC_RMID` το id είναι άκυρο.** Κάθε επόμενη `semctl` ή `semop` στο `$id` αποτυγχάνει με `EINVAL` / `EIDRM`. Μην αποθηκεύσετε το id σε cache πέρα από την αφαίρεση.
- **Το σύνολο είναι καθολικό στον πυρήνα και διατηρείται μεταξύ εξόδων διεργασιών.** Ένα σύνολο που δημιουργήθηκε από κατεστραμμένο σενάριο παραμένει μέχρι να καλέσει κάποιος `IPC_RMID` (ή να επανεκκινήσει το σύστημα). Χρησιμοποιήστε `ipcs -s` / `ipcrm -s` από το κέλυφος για να επιθεωρήσετε ή να καθαρίσετε διαρρεύσαντα σύνολα.
- **Τα δικαιώματα προέρχονται από τη `semget`.** Το `IPC_CREAT | mode` που περνά κατά τη δημιουργία ελέγχει ποιος μπορεί να εκδώσει ποιες εντολές `semctl`. Το `EACCES` από την `semctl` σημαίνει ότι τα mode bits δεν επιτρέπουν τη ζητούμενη λειτουργία για τον τρέχοντα ενεργό uid.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `semget` - δημιουργήστε ή αναζητήστε το σύνολο σηματοφόρων του οποίου το id περνάτε στη συνέχεια στη `semctl`
- `semop` - οι πραγματικές λειτουργίες wait / signal· η `semctl` είναι για έλεγχο και επιθεώρηση, η `semop` κάνει τη δουλειά
- `msgctl` - η αντίστοιχη κλήση ελέγχου για ουρές μηνυμάτων System V· ίδια σύμβαση τιμής επιστροφής
- `shmctl` - η αντίστοιχη κλήση ελέγχου για κοινόχρηστη μνήμη System V· ίδια σύμβαση τιμής επιστροφής
- `pack` - πώς να κατασκευάσετε τον ενταμιευτή ARG με `s!` για τιμές σηματοφόρου και τη δομή `semid_ds`
- `IPC::Semaphore` - αντικειμενικό wrapper που κρύβει τη `semctl` πίσω από ονομασμένες μεθόδους· επιλέξτε το όταν ένα σενάριο κάνει πάνω από μία ή δύο κλήσεις

SysV IPC

semget

Δημιουργεί ή αναζητά ένα σύνολο σηματοφόρων System V και επιστρέφει το αναγνωριστικό του.

Η `semget` είναι το σημείο εισόδου στο API σηματοφόρων SysV. Περιτυλίγει την κλήση συστήματος `semget(2)`: δοθέντος αριθμητικού `KEY` που κατονομάζει το σύνολο, του αριθμού σημαφόρων `NSEMS` στο σύνολο, και μιας λέξης `FLAGS` που συνδυάζει bits δικαιωμάτων με σημαίες δημιουργίας, ο πυρήνας είτε επιστρέφει το id ενός αντίστοιχου υπάρχοντος συνόλου, είτε, αν το `IPC_CREAT` είναι στις `FLAGS`, δημιουργεί ένα νέο σύνολο και επιστρέφει το id του. Το επιστρεφόμενο id είναι το handle που αναμένει κάθε άλλη κλήση σηματοφόρου (`semop`, `semctl`).

Σύνοψη

```
semget KEY, NSEMS, FLAGS
```

Τι επιστρέφεται

Το αναγνωριστικό του συνόλου σηματοφόρων (μη αρνητικός ακέραιος) σε επιτυχία, ή `undef` σε αποτυχία με το `$!` τεθειμένο στο `errno` της υποκείμενης κλήσης `semget(2)`. Το id **δεν** είναι περιγραφέας αρχείου - είναι ένα handle SysV IPC σε επίπεδο πυρήνα που επιβιώνει της διεργασίας σας και διατηρείται μέχρι να αφαιρεθεί ρητά με `semctl($id, 0, IPC_RMID, 0)`.

```
use IPC::SysV qw(IPC_CREAT IPC_PRIVATE);

my $semid = semget(IPC_PRIVATE, 3, IPC_CREAT | 0600)
    // die "semget: $!";
```

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται στην τιμή `errno` από την `semget(2)` σε αποτυχία (`EACCES`, `EEXIST`, `ENOENT`, `ENOSPC`, `EINVAL`, `ENOMEM`).

Το όρισμα KEY

Το `KEY` είναι ακέραιος που κατονομάζει το σύνολο σηματοφόρων σε επίπεδο συστήματος. Τρεις συμβάσεις έχουν σημασία:

- **`IPC_PRIVATE`** (εισάγεται από το `IPC::SysV`) - ο πυρήνας εκχωρεί ένα φρέσκο κλειδί που δεν χρησιμοποιείται από κανένα άλλο σύνολο. Το προκύπτον id είναι γνωστό μόνο σε αυτή τη διεργασία και σε όποιον της περάσει το id, τυπικά παιδιά μετά από `fork`. Χρησιμοποιήστε αυτό όταν το σύνολο είναι λεπτομέρεια υλοποίησης ενός δέντρου διεργασιών.
- **Σταθερό ακέραιο κυριολεκτικό** - κάθε συνεργαζόμενη διεργασία κωδικοποιεί σταθερά τον ίδιο αριθμό. Εύθραυστο μεταξύ άσχετων προγραμμάτων που μπορεί να επιλέξουν τον ίδιο ακέραιο.
- **Κλειδί παραγόμενο από την `ftok(3)`** - η `IPC::SysV::ftok($path, $proj_id)` μετατρέπει μια διαδρομή στο filesystem συν ένα project byte σε αριθμητικό κλειδί.

Η τυπική συνταγή για άσχετες διεργασίες που χρειάζεται να συμφωνήσουν σε κλειδί χωρίς να κωδικοποιήσουν σταθερά έναν μαγικό αριθμό.

Το όρισμα FLAGS

Το FLAGS συνδυάζει **bits δικαιωμάτων** (κατώτερα 9 bits, όπως ένα mode chmod: `0600`, `0660`, `0666`) με **σημαίες δημιουργίας** από το `IPC::SysV`:

- `IPC_CREAT` - δημιουργεί το σύνολο αν δεν υπάρχει ήδη. Χωρίς αυτή τη σημαία, η `semget` αναζητά μόνο υπάρχον σύνολο με το δοσμένο κλειδί.
- `IPC_CREAT | IPC_EXCL` - δημιουργεί το σύνολο, αλλά αποτυγχάνει με `EEXIST` αν υπάρχει ήδη ένα για το KEY. Το αντίστοιχο SysV του `O_CREAT | O_EXCL` στην `open`.

Τα bits δικαιωμάτων ελέγχονται από τον πυρήνα σε κάθε επόμενη κλήση `semop` και `semctl` έναντι του uid/gid της επικαλούμενης διεργασίας.

Παραδείγματα

Δημιουργήστε ένα ιδιωτικό σύνολο τριών σηματοφόρων που είναι χρησιμοποιήσιμο από αυτή τη διεργασία και τα παιδιά της:

```
use IPC::SysV qw(IPC_PRIVATE IPC_CREAT);
my $semid = semget(IPC_PRIVATE, 3, IPC_CREAT | 0600)
    // die "semget: $!";
```

Αναζητήστε ένα υπάρχον, ονομασμένο σύνολο χωρίς να το δημιουργήσετε. Το `NSEMS` πρέπει να ταιριάζει με το μέγεθος με το οποίο δημιουργήθηκε το σύνολο, ή μπορεί να είναι `0` αν δεν σας ενδιαφέρει:

```
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: $!";
```

Ιδίωμα create-or-open - δοκιμάστε αποκλειστική δημιουργία, με εφεδρεία την αναζήτηση. Η πρώτη διεργασία που περνά αρχικοποιεί το σύνολο· οι επόμενες απλώς συνδέονται:

```
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: $!";
}
```

Αφαιρέστε ένα σύνολο όταν τελειώσετε με αυτό. Τα σύνολα SysV διαρρέουν μεταξύ εξόδων διεργασιών - τίποτα δεν τα καθαρίζει για εσάς:

```
use IPC::SysV qw(IPC_RMID);
semctl($semid, 0, IPC_RMID, 0) or warn "semctl RMID: $!";
```

Οριακές περιπτώσεις

- **NSEMS σε αναζήτηση**: όταν αναζητάτε ένα υπάρχον σύνολο (χωρίς IPC_CREAT), NSEMS ίσο με 0 σημαίνει «οποιοδήποτε μέγεθος»· μια θετική τιμή πρέπει να είναι μικρότερη ή ίση από το πραγματικό μέγεθος του συνόλου, αλλιώς η semget αποτυγχάνει με EINVAL.
- **Το IPC_PRIVATE εξακολουθεί να υπακούει στις FLAGS**: εφαρμόζονται τα bits δικαιωμάτων. Ένα σύνολο που δημιουργήθηκε με 0400 είναι μόνο για ανάγνωση ακόμη και για τη διεργασία που το δημιούργησε, για κλήσεις semop που το τροποποιούν.
- **Όρια πυρήνα**: η semget μπορεί να αποτύχει με ENOSPC όταν το όριο συνόλων σηματοφόρου σε επίπεδο συστήματος φτάσει στο τέλος, ή EINVAL όταν το NSEMS υπερβαίνει το μέγιστο ανά σύνολο. Δείτε `ipcs -sl` για τα τρέχοντα όρια σε Linux.
- **Διαρρεύσαντα σύνολα**: ένα σύνολο διατηρείται μέχρι την IPC_RMID ή την επανεκκίνηση. Δαίμονες μακράς διάρκειας που ξεχνούν να αφαιρέσουν τα σύνολά τους τα συσσωρεύουν· το `ipcs -s` απαριθμεί τους επιζώντες, το `ipcrm -s $id` αφαιρεί έναν.
- **Συγκρούσεις κλειδιών**: δύο άσχετα προγράμματα που επιλέγουν το ίδιο σταθερά κωδικοποιημένο KEY θα βλέπουν το σύνολο το ένα του άλλου - εξ ου και το ιδίωμα `ftok(3)`.
- **Όχι filehandle**: το $semid είναι ακέραιος, όχι handle. Μην το περάσετε σε close, fcntl, ή σε οποιαδήποτε λειτουργία filehandle.
- **Τα forks κληρονομούν ids, όχι ιδιοκτησία**: ένα παιδί βλέπει την ίδια τιμή $semid με τον γονέα και μπορεί να τη χρησιμοποιήσει, αλλά το ίδιο το σύνολο ανήκει στον uid που το δημιούργησε, όχι σε κάποια συγκεκριμένη διεργασία.
- **Αριθμητικό περιβάλλον**: το id είναι απλός ακέραιος, ασφαλές για εκτύπωση, σύγκριση και αποθήκευση σε αρχείο ώστε να το παραλάβει άλλη διεργασία.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- semop - εκτελέστε λειτουργίες wait/signal επί του συνόλου που επιστρέφει η semget
- semctl - λειτουργίες ελέγχου επί του συνόλου: αρχικοποίηση τιμών (SETALL, SETVAL), ανάγνωσή τους (GETALL, GETVAL), και αφαίρεση του συνόλου (IPC_RMID)
- msgget - το αντίστοιχο για ουρά μηνυμάτων, ίδιες συμβάσεις KEY, FLAGS
- shmget - το αντίστοιχο για κοινόχρηστη μνήμη, ίδιες συμβάσεις KEY, FLAGS

- `IPC::SysV` - εισάγει τα `IPC_CREAT`, `IPC_EXCL`, `IPC_PRIVATE`, `IPC_RMID`, `SETALL`, `GETALL`, και τον βοηθό `ftok`
- `IPC::Semaphore` - αντικειμενικό wrapper γύρω από τις ωμές ενσωματωμένες `sem*` όταν προτιμάτε να μη πακετάρετε δομές χειρωνακτικά

SysV IPC

semop

Εκτελεί ατομικά μία ή περισσότερες λειτουργίες σηματοφόρου System V.

Η `semop` είναι το άλογο εργασίας στη χρήση σηματοφόρων SysV: παίρνει ένα αναγνωριστικό συνόλου σηματοφόρων (από την `semget`) και μια πακεταρισμένη λίστα λειτουργιών, και κατόπιν ζητά από τον πυρήνα να τις εφαρμόσει όλες ή καμία. Κάθε λειτουργία ονομάζει έναν μεμονωμένο σηματοφόρο μέσα στο σύνολο, ένα ποσό προς πρόσθεση σε αυτόν (συνήθως `-1` για αναμονή, `+1` για σήμα) και μια λέξη σημαιών. Ο πυρήνας μπλοκάρει τον καλούντα μέχρι κάθε ζητούμενη λειτουργία να μπορεί να προχωρήσει, και κατόπιν τις δεσμεύει ως μία μονάδα. Αυτό κάνει την `semop` αξιοποιήσιμη ως πρωτογενή λειτουργία αμοιβαίου αποκλεισμού ή μέτρησης πόρων μεταξύ άσχετων διεργασιών.

Σύνοψη

```
semop $semid, $opstring
```

Τι επιστρέφεται

Αληθής τιμή σε επιτυχία, ψευδής τιμή σε σφάλμα με την `$!` ορισμένη. Η τιμή αλήθειας είναι το μόνο αποτέλεσμα - η `semop` επικοινωνεί το αποτέλεσμα μέσω της παρενέργειας στο σύνολο σηματοφόρων, όχι μέσω της επιστροφής της. Ελέγξτε την σε κάθε κλήση:

```
semop($semid, $ops)
    or die "semop failed: $!";
```

Η ψευδής επιστροφή σημαίνει ότι καμία λειτουργία στην `OPSTRING` δεν εφαρμόστηκε· η σημασιολογία SysV εγγυάται όλα-ή-τίποτα.

Καθολική κατάσταση που επηρεάζει

Ορίζει την `$!` σε αποτυχία, με τις συνήθεις τιμές errno από την `semop(2)`: `EAGAIN` όταν είχε οριστεί το `IPC_NOWAIT` και η λειτουργία θα μπλοκάριζε, `EIDRM` όταν το σύνολο σηματοφόρων αφαιρέθηκε ενώ περίμενε, `EINTR` όταν σήμα διέκοψε την αναμονή, `EINVAL` για κακό αριθμό ή αναγνωριστικό σηματοφόρου, `EACCES` για αναντιστοιχία δικαιωμάτων έναντι του mode του συνόλου.

Η δομή της λειτουργίας

Η `OPSTRING` είναι συνένωση εγγραφών σταθερού πλάτους. Κάθε εγγραφή είναι τρεις ακέραιοι `short` σε εγγενές πλάτος και παράγεται με `pack`:

```
pack("s!3", $semnum, $semop, $semflag)
```

- `$semnum` - ο δείκτης του σηματοφόρου-στόχου μέσα στο σύνολο (`0` έως `nsems-1`, όπου το `nsems` είναι το μέγεθος που πέρασε στην `semget`).
- `$semop` - το ποσό που προστίθεται στην τιμή του σηματοφόρου. Οι αρνητικές τιμές περιμένουν μέχρι ο σηματοφόρος να είναι τουλάχιστον `|$semop|` πριν αφαιρέσουν· οι θετικές τιμές προσθέτουν χωρίς συνθήκη και ξυπνούν όσους περιμένουν· το μηδέν περιμένει μέχρι ο σηματοφόρος να φτάσει το μηδέν.
- `$semflag` - δυαδικό OR του `IPC_NOWAIT` (αποτυγχάνει με `EAGAIN` αντί να μπλοκάρει) και του `SEM_UNDO` (ζητά από τον πυρήνα να αναιρέσει τη λειτουργία αν η διεργασία τερματίσει χωρίς να την αναιρέσει - απαραίτητο για χρήσεις τύπου κλειδώματος όπου ένα crash δεν πρέπει να αφήσει τον σηματοφόρο κρατημένο).

Το μήκος της `OPSTRING` υπονοεί τον αριθμό των λειτουργιών: ο πυρήνας επεξεργάζεται `length($opstring) / sizeof(struct sembuf)` εγγραφές. Το πέρασμα δύο εγγραφών πακεταρισμένων αλληλένδετα ζητά δύο λειτουργίες που εφαρμόζονται ατομικά.

Το `s!` στο πρότυπο είναι ο **εγγενής** `short`, που ταιριάζει στη διάταξη του `sembuf` στο Linux. Το σκέτο `s` είναι 16-bit προσημασμένο σε network-order και θα κάνει σιωπηρά λανθασμένο πακετάρισμα σε οποιαδήποτε πλατφόρμα όπου `sizeof(short) != 2` ή όπου η endianness διαφέρει από το network order.

Παραδείγματα

Αναμονή στον (μείωση κατά 1) σηματοφόρο 0 του συνόλου στο `$semid`:

```
my $op = pack("s!3", 0, -1, 0);
semop($semid, $op)
    or die "wait on semaphore failed: $!";
```

Σήμα (αύξηση κατά 1) στον ίδιο σηματοφόρο:

```
my $op = pack("s!3", 0, 1, 0);
semop($semid, $op)
    or die "signal failed: $!";
```

Προσπάθεια απόκτησης χωρίς μπλοκάρισμα - αποτυχία γρήγορα αν τον κρατά κάποιος άλλος:

```
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
}
```

Απόκτηση του σηματοφόρου 0 με αυτόματη απελευθέρωση στην έξοδο της διεργασίας - το μοτίβο για αμοιβαίο αποκλεισμό ασφαλή σε crash:

```
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: $!";
```

Δύο λειτουργίες που εφαρμόζονται ατομικά - αναμονή στους σηματοφόρους 0 και 1 μαζί, χωρίς να απελευθερωθεί κανείς αν κάποιος από τους δύο θα μπλόκαρε:

```
my $ops = pack("s!3", 0, -1, 0)
        . pack("s!3", 1, -1, 0);
semop($semid, $ops) or die "paired wait: $!";
```

Αναμονή να φτάσει στο μηδέν ο σηματοφόρος 0 - η μορφή «φράγμα», όπου το `$semop` είναι μηδέν αντί για αρνητικό:

```
my $op = pack("s!3", 0, 0, 0);
semop($semid, $op) or die "barrier: $!";
```

Οριακές περιπτώσεις

- **Σύντομη `OPSTRING`**: το μήκος πρέπει να είναι ακριβές πολλαπλάσιο του μεγέθους εγγραφής `sembuf`. Συμβολοσειρά αποκομμένη ή με padding δίνει `EINVAL`. Χτίζετε πάντα τη συμβολοσειρά από κλήσεις `pack`, ποτέ στο χέρι.
- **Το εγγενές πλάτος έχει σημασία**: `s!3` και όχι `s3`. Ένα σενάριο γραμμένο με σκέτο `s` λειτουργεί σε υπολογιστή με 16-bit short κατά τύχη και σπάει αλλού.
- **Μηδενικού μήκους `OPSTRING`**: ζητά μηδέν λειτουργίες. Ο πυρήνας το αποδέχεται και επιστρέφει επιτυχία χωρίς παρενέργειες - σπάνια αυτό που θέλετε. Προστατέψτε την κλήση με `return unless length $ops`.
- **Διακοπή από σήμα**: μια μπλοκαρισμένη `semop` που επιστρέφει με την `$!` ορισμένη σε `EINTR` δεν είναι σφάλμα - επαναλάβετε εκτός αν εγκαταστήσατε χειριστή που θέλει να ματαιώσει. Ο προεπιλεγμένος χειρισμός `$SIG{*}` της Perl δεν επανεκκινεί αυτόματα την κλήση συστήματος.
- **Το `SEM_UNDO` με ζευγαρωμένα ταιριαστά ζεύγη**: ο πυρήνας παρακολουθεί τη σωρευτική προσαρμογή ανά σηματοφόρο ανά διεργασία. Η απόκτηση δύο φορές με `SEM_UNDO` και κατόπιν μία απελευθέρωση αφήνει ένα εκκρεμές `-1` να αναιρεθεί κατά την έξοδο. Ζευγαρώστε κάθε `-1` με ένα ταιριαστό `+1` στην κανονική ροή· το `SEM_UNDO` είναι ασφάλεια έναντι crashes, όχι υποκατάστατο της απελευθέρωσης.
- **Αφαιρεμένο σύνολο κατά την αναμονή**: αν άλλη διεργασία καλέσει `semctl` με `IPC_RMID`, κάθε μπλοκαρισμένη `semop` επιστρέφει ψευδή τιμή με την `$!` ορισμένη σε `EIDRM`. Το αναγνωριστικό σηματοφόρου είναι άκυρο από εκείνο το σημείο και μετά.
- **Δικαιώματα**: η `semop` ελέγχει το δικαίωμα αλλαγής στο σύνολο. Μια διεργασία που μπορεί να κάνει `semget` ένα read-only handle δεν μπορεί παρ” όλα αυτά να εκτελέσει `semop` σε αυτό.
- **Μεγάλο πλήθος λειτουργιών**: ο πυρήνας περιορίζει τον αριθμό λειτουργιών ανά κλήση στο `SEMOPM` (συνήθως 32). Το χτίσιμο μιας `OPSTRING` με περισσότερες

εγγραφές αποτυγχάνει με E2BIG.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `semget` - αποκτήστε το αναγνωριστικό που περνάτε ως πρώτο όρισμα· οι δύο χρησιμοποιούνται πάντα μαζί
- `semctl` - ορίζει ή ερωτά τις τιμές σηματοφόρων, και αφαιρεί το σύνολο όταν τελειώσετε με αυτό
- `pack` - χτίζει κάθε εγγραφή `sembuf`· `s!3` είναι το πρότυπο που θέλετε
- `msgsnd` - το αντίστοιχο για ουρές μηνυμάτων όταν χρειάζεστε να περάσετε δεδομένα αντί να σηματοδοτήσετε διαθεσιμότητα
- `shmget` - το αντίστοιχο για κοινόχρηστη μνήμη· οι σηματοφόροι χρησιμοποιούνται συνήθως για να προστατέψουν την πρόσβαση σε ένα τμήμα κοινόχρηστης μνήμης SysV
- `IPC::SysV` - εξάγει τις σταθερές `IPC_NOWAIT`, `SEM_UNDO` και `IPC_RMID` που χρησιμοποιούνται στο `$semflag` και στις εντολές `semctl`
- `IPC::Semaphore` - αντικειμενοστραφές περιτύλιγμα πάνω από τις `semget` / `semop` / `semctl` όταν η διαδικαστική διεπαφή φαντάζει υπερβολικά γυμνή

Υποδοχές (sockets)

send

Στέλνει ένα μήνυμα σε μια υποδοχή.

Η `send` γράφει το βαθμωτό `MSG` στο `SOCKET` μέσω του στρώματος υποδοχών του λειτουργικού συστήματος, όχι μέσω PerlIO. Σε συνδεδεμένη υποδοχή (`SOCK_STREAM` μετά από `connect`, ή αποδεκτό ομότιμο) η μορφή τριών ορισμάτων εκδίδει μια `send(2)` - χρησιμοποιήστε τη μορφή τεσσάρων ορισμάτων σε μη συνδεδεμένη υποδοχή datagram (`SOCK_DGRAM`) για να δρομολογήσετε κάθε μήνυμα στο `TO`, κάτι που πυροδοτεί την `sendto(2)` στο παρασκήνιο. Το `FLAGS` είναι η μάσκα bits που δέχεται η κλήση συστήματος (συνήθως `0`· δείτε το man page της πλατφόρμας σας για την `send(2)` σχετικά με `MSG_DONTWAIT`, `MSG_OOB`, `MSG_NOSIGNAL` και παρόμοια).

Σύνοψη

```
send SOCKET, MSG, FLAGS, TO
send SOCKET, MSG, FLAGS
```

Τι επιστρέφεται

Το **πλήθος των bytes** που έγιναν δεκτά από τον πυρήνα, ή `undef` σε σφάλμα με ρυθμισμένο το `$!`. Ελέγχετε πάντα την τιμή επιστροφής:

```
my $n = send $sock, $buf, 0
    or die "send failed: $!";
```

Δύο ιδιότητες δαγκώνουν τους χρήστες:

- Το πλήθος είναι **bytes**, όχι χαρακτήρες. Το `length $buf` σε συμβολοσειρά πλατιών χαρακτήρων και το επιστρεφόμενο πλήθος δεν θα συμφωνούν.
- Σε υποδοχές `SOCK_STREAM` ο πυρήνας ενδέχεται να δεχτεί **λιγότερα bytes από όσα του δώσατε** - σύντομη εγγραφή. Επαναλάβετε σε βρόχο μέχρι να εκκενωθεί ολόκληρος ο ενταμιευτής, ή χρησιμοποιήστε `syswrite` που έχει την ίδια σημασιολογία σύντομης εγγραφής αλλά λειτουργεί σε οποιοδήποτε filehandle.

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται στο σφάλμα του συστήματος σε αποτυχία.
- Η στοίβα στρωμάτων PerlIO της υποδοχής **δεν** συμβουλεύεται. Τα στρώματα εξόδου (`:utf8`, `:encoding(...)`, `:crlf`) δεν έχουν καμία επίδραση σε αυτό που γράφει η `send`. Ειδικότερα, ένα στρώμα `:utf8` κάνει την `send` να **ρίξει εξαίρεση** αντί να κωδικοποιήσει σιωπηλά - το στρώμα υποδοχής και το στρώμα PerlIO δεν μπορούν να κατέχουν και τα δύο το byte framing.

Παραδείγματα

Συνδεδεμένο TCP - μορφή τριών ορισμάτων:

```
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: $!";
```

Μη συνδεδεμένο UDP - μορφή τεσσάρων ορισμάτων με ρητό προορισμό:

```
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: $!";
```

Χειρισμός σύντομων εγγραφών σε υποδοχή ροής:

```
my $offset = 0;
while ($offset < length $buf) {
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    my $n = send $sock, substr($buf, $offset), 0;
    defined $n or die "send: $!";
    $offset += $n;
}
```

Αποστολή με σημαία - το MSG_DONTWAIT κάνει μία μη μπλοκαριστική προσπάθεια και επιστρέφει undef με το $! ρυθμισμένο σε EAGAIN/EWOULDBLOCK αν ο ενταμιευτής του πυρήνα είναι γεμάτος:

```
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: $!";
    }
}
```

Οριακές περιπτώσεις

- **Πλατιοί χαρακτήρες**: το πέρασμα συμβολοσειράς με codepoints πάνω από 0xFF σε υποδοχή bytes γράφει την κωδικοποίηση UTF-8 κάθε χαρακτήρα και εκπέμπει προειδοποίηση Wide character in send υπό use warnings. Κωδικοποιήστε ρητά με Encode::encode πριν την κλήση της send.
- **Στρώμα :utf8 στην υποδοχή**: η send **πεθαίνει**. Είτε ανοίξτε την υποδοχή χωρίς το στρώμα, είτε καλέστε binmode $sock για να το αφαιρέσετε πριν την πρώτη send. Ένα στρώμα :encoding(...) προσθέτει σιωπηρά το :utf8, οπότε απαγορεύεται και αυτό.
- **Μορφή τεσσάρων ορισμάτων σε συνδεδεμένη υποδοχή**: το πέρασμα TO σε συνδεδεμένη υποδοχή SOCK_STREAM είναι σφάλμα - ο πυρήνας επιστρέφει EISCONN. Χρησιμοποιήστε τη μορφή τριών ορισμάτων αφού γίνει η σύνδεση.
- **Μορφή τριών ορισμάτων σε μη συνδεδεμένη υποδοχή datagram**: αποτυγχάνει με ENOTCONN (ή EDESTADDRREQ σε ορισμένες πλατφόρμες). Είτε connect-άρετε πρώτα την υποδοχή datagram (που καρφώνει έναν προεπιλεγμένο ομότιμο), είτε παρέχετε το TO.
- **Κενό MSG σε UDP**: έγκυρο - στέλνει datagram μηδενικού μήκους που ο ομότιμος μπορεί να διαβάσει με recv. Σε TCP είναι no-op που επιστρέφει 0.
- **SIGPIPE σε κλεισμένο ομότιμο ροής**: η εγγραφή σε SOCK_STREAM της οποίας ο απομακρυσμένος άκρος έχει κλείσει εγείρει SIGPIPE, που εξ ορισμού τερματίζει τη διεργασία. Περάστε MSG_NOSIGNAL στα FLAGS (Linux) ή εγκαταστήστε $SIG{PIPE} = 'IGNORE' για να λάβετε αντί αυτού επιστροφή EPIPE.
- **sendmsg(2)**: η Perl εκθέτει τις send/sendto, όχι την sendmsg. Οι scatter/gather εγγραφές και τα ancillary δεδομένα (credentials, file descriptors πάνω από AF_UNIX) απαιτούν άρθρωμα XS όπως το Socket::MsgHdr.

- **Μη-socket filehandle**: η send σε μη-socket επιστρέφει `undef` με το `$!` ρυθμισμένο σε ENOTSOCK. Χρησιμοποιήστε `syswrite` για pipes και κανονικά αρχεία.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `recv` - η αντίστροφη· διαβάζει bytes από υποδοχή με τους ίδιους περιορισμούς στρωμάτων και το ίδιο όρισμα FLAGS
- `syswrite` - ακατέργαστη εγγραφή που παρακάμπτει την ενταμίευση PerlIO σε οποιοδήποτε filehandle· χρησιμοποιήστε την για pipes και αρχεία όπου η send αρνείται να λειτουργήσει
- `connect` - εγκαθιδρύει τη διεύθυνση ομότιμου που σας επιτρέπει να χρησιμοποιήσετε τη μορφή τριών ορισμάτων της send
- `socket` - δημιουργεί το filehandle στο οποίο γράφει η send· ο τύπος υποδοχής (SOCK_STREAM έναντι SOCK_DGRAM) αποφασίζει ποια μορφή της send είναι νόμιμη
- `bind` - εκχωρεί τοπική διεύθυνση· χρειάζεται πριν την send σε υποδοχή datagram αν ο ομότιμος αναμένει απαντήσεις σε γνωστή θύρα
- `binmode` - αφαιρεί ένα αδέσποτο στρώμα `:utf8` που διαφορετικά θα έκανε την send να ρίξει εξαίρεση

Πληροφορίες χρηστών και ομάδων

setgrent

Επανατυλίγει τον επαναλήπτη της βάσης δεδομένων ομάδων πίσω στην πρώτη εγγραφή.

Η `setgrent` λέει στον επαναλήπτη της βάσης δεδομένων ομάδων του συστήματος να ξεκινήσει από την αρχή. Η επόμενη κλήση στην `getgrent` επιστρέφει ξανά την πρώτη εγγραφή ομάδας, ανεξάρτητα από το πόσο μακριά είχε φτάσει η προηγούμενη επανάληψη. Δεν παίρνει ορίσματα, δεν επιστρέφει χρήσιμη τιμή, και υπάρχει ώστε ένα πρόγραμμα να μπορεί να ξανασαρώσει τη βάση ομάδων χωρίς να χρειαστεί να την κλείσει και να την ξανανοίξει χειροκίνητα. Συνδυάστε την με την `endgrent` στο τέλος μιας σάρωσης για να απελευθερώσετε τον υποκείμενο περιγραφέα αρχείου.

Σύνοψη

```
setgrent;
setgrent();
```

Τι επιστρέφεται

Η τιμή επιστροφής δεν είναι προδιαγεγραμμένη και κατά σύμβαση αγνοείται. Καλέστε την `setgrent` για την παρενέργειά της - την τοποθέτηση του επαναλήπτη - όχι για την τιμή της.

Καθολική κατάσταση που επηρεάζει

- Τον **επαναλήπτη βάσης δεδομένων ομάδων ανά διεργασία** που διατηρείται από τη βιβλιοθήκη C (`getgrent(3)`). Είναι καθολικός σε επίπεδο διεργασίας, χωρίς εμβέλεια πακέτου ή λεξιλογικής εμβέλειας: οποιοσδήποτε άλλος κώδικας διασχίζει την `getgrent` στην ίδια διεργασία, συμπεριλαμβανομένων threads που μοιράζονται τον επαναλήπτη, βλέπει το επανατύλιγμα.
- Η `$!` - σε συστήματα των οποίων η `setgrent(3)` αναφέρει σφάλματα (σπάνιο· οι περισσότερες υλοποιήσεις επιστρέφουν `void`), μια αποτυχία αφήνει την `errno` τεθειμένη. Φορητός κώδικας δεν βασίζεται σε αυτό.

Η `setgrent` δεν αγγίζει την `$_`, τον `@ARGV`, ή οποιοδήποτε filehandle.

Παραδείγματα

Επανατύλιγμα μεταξύ δύο διασχίσεων της βάσης δεδομένων ομάδων - πρώτα για συλλογή όλων των ονομάτων ομάδων, μετά για συλλογή των μελών μιας συγκεκριμένης ομάδας:

```
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;
```

Χρησιμοποιήστε `setgrent` στην αρχή μιας sub που καλείται επανειλημμένα και θέλει φρέσκια σάρωση κάθε φορά:

```
sub groups_containing {
    my ($user) = @_;
    setgrent;
    my @groups;
    while (my ($name, undef, undef, $members) = getgrent) {
        push @groups, $name if grep { $_ eq $user } split / /,
↪$members;
    }
    endgrent;
    return @groups;
}
```

Συνδυάστε `setgrent` / `endgrent` γύρω από μια σάρωση ώστε ένας άσχετος καλών που βρισκόταν στη μέση επανάληψης να μην αιφνιδιαστεί όταν ο έλεγχος επιστρέψει:

```
setgrent;
while (my @g = getgrent) {
    # ... process ...
}
endgrent;
```

Οριακές περιπτώσεις

- **Μόνο η μορφή χωρίς ορίσματα.** Σε αντίθεση με τις `sethostent`, `setnetent`, `setprotoent`, και `setservent`, που παίρνουν σημαία `STAYOPEN`, η `setgrent` δεν παίρνει τίποτα. Η μετάδοση ορίσματος είναι σφάλμα κατά τη μεταγλώττιση.
- **Η κλήση της `setgrent` χωρίς ενδιάμεση `getgrent` είναι no-op στα περισσότερα συστήματα** - ο επαναλήπτης απλώς τοποθετείται στην πρώτη εγγραφή, και η πρώτη `getgrent` μετά διαβάζει αυτή την εγγραφή. Μη βασίζεστε στην `setgrent` ως πρωτογενή λειτουργία «άνοιγμα του αρχείου ομάδων»· η μόνη ορισμένη συμπεριφορά της είναι «επανατύλιγμα».
- **Καθολική κατάσταση σε επίπεδο διεργασίας.** Η `setgrent` μέσα σε ένα άρθρωμα επανατυλίγει τον επαναλήπτη για κάθε άλλο άρθρωμα στην ίδια διεργασία. Αν μια βιβλιοθήκη διασχίσει την `getgrent` μέχρι το τέλος και ένας καλών μετά καλέσει `setgrent`, ο δρομέας της βιβλιοθήκης έχει χαθεί. Αντιμετωπίστε τον επαναλήπτη ως κοινόχρηστο πόρο: κρατήστε τον για τη διάρκεια μιας σάρωσης, οριοθετημένο από `setgrent` και `endgrent`.
- **Τα threads μοιράζονται τον επαναλήπτη.** Δύο threads που καλούν `getgrent` ταυτόχρονα θα δουν το καθένα ένα υποσύνολο των εγγραφών· η `setgrent` από ένα thread ενοχλεί το άλλο. Για επανάληψη ανά thread, διαβάστε απευθείας το `/etc/group` ή χρησιμοποιήστε thread-safe περιτυλιγμό· η ενσωματωμένη δεν είναι επανεισόδιμη.
- **Δεν την υλοποιούν όλες οι πλατφόρμες πανομοιότυπα.** Σε συστήματα χωρίς `setgrent(3)` η κλήση είναι no-op και πετυχαίνει σιωπηλά. Σε συστήματα που την υλοποιούν, το ακριβές κόστος (επανάνοιγμα του `/etc/group`, επανεκκίνηση ενός ερωτήματος NSS, επανερώτηση LDAP) εξαρτάται από την υποκείμενη υπηρεσία ονομάτων και μπορεί να είναι μη ασήμαντο - αποφύγετε την κλήση σε σφιχτό βρόχο.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `getgrent` - διαβάστε την επόμενη εγγραφή ομάδας· καταναλώνει τον επαναλήπτη που επανατυλίγει η `setgrent`
- `endgrent` - κλείστε τον επαναλήπτη όταν η σάρωση τελειώσει· το συμμετρικό αντίστοιχο της `setgrent`
- `getgrnam` - αναζήτηση μίας ομάδας με όνομα χωρίς καμία επαφή με τον επαναλήπτη· προτιμήστε την για ερωτήματα μίας εκτέλεσης
- `getgrgid` - αναζήτηση μίας ομάδας με GID· ίδιο σκεπτικό με την `getgrnam`

- `setpwent` - το αντίστοιχο επανατύλιγμα για τη βάση δεδομένων passwd

Πληροφορίες δικτύου

sethostent

Ανοίγει ή επανατυλίγει τη βάση δεδομένων υπολογιστών για επανάληψη.

Η `sethostent` προετοιμάζει τη βάση δεδομένων υπολογιστών του συστήματος (τυπικά `/etc/hosts` συν τυχόν πηγές υπηρεσίας ονομάτων που έχουν ρυθμιστεί από την πλατφόρμα) για σειριακή διάσχιση με την `gethostent`, και επανατυλίγει τον δρομέα αν η επανάληψη βρίσκεται ήδη σε εξέλιξη. Το `STAYOPEN` είναι σημαία που περνάει στην υποκείμενη C κλήση `sethostent(3)`: αληθής τιμή ζητάει από τον resolver να κρατήσει τη βάση ανοιχτή σε όλες τις επόμενες αναζητήσεις `gethostby*`, που σε ορισμένα συστήματα αποφεύγει το ξανα-άνοιγμα του αρχείου ή την επανασύνδεση στην υπηρεσία ονομάτων για κάθε ερώτημα.

Σύνοψη

```
sethostent STAYOPEN
sethostent 0
sethostent 1
```

Τι επιστρέφεται

Δεν επιστρέφει τίποτα χρήσιμο. Η κλήση γίνεται για την παρενέργειά της να εκκινήσει τον επαναλήπτη· αγνοήστε την τιμή επιστροφής.

Καθολική κατάσταση που επηρεάζει

Τροποποιεί **καθολική σε επίπεδο διεργασίας** κατάσταση του resolver που διατηρείται από τη βιβλιοθήκη C: τον περιγραφέα αρχείου της βάσης δεδομένων υπολογιστών ή το handle της υπηρεσίας ονομάτων, και τον δρομέα επανάληψης ανά διεργασία που διαβάζει η `gethostent`. Δύο κομμάτια κώδικα στην ίδια διεργασία που διασχίζουν ταυτόχρονα τη βάση δεδομένων υπολογιστών θα μπερδευτούν - υπάρχει ένας δρομέας, όχι ένας ανά καλούντα. Σε αποτυχία, η `$!` τίθεται από την υποκείμενη κλήση συστήματος· δεν υπάρχει αναφορά τύπου `h_errno` για αυτήν τη συνάρτηση.

Παραδείγματα

Διασχίστε κάθε εγγραφή στη βάση δεδομένων υπολογιστών:

```
sethostent(0);
while (my ($name, $aliases, $addrtype, $length, @addrs)
        = gethostent()) {
    print "$name\n";
}
endhostent();
```

Ζητήστε από τον resolver να κρατήσει τη βάση ανοιχτή ενώ τρέχει μια παρτίδα αναζητήσεων ονόματος-σε-διεύθυνση, και μετά κλείστε την:

```
sethostent(1);
for my $host (@hosts) {
    my $packed = gethostbyname($host);
    # ...
}
endhostent();
```

Επανατυλίξτε μια εν εξελίξει επανάληψη για να ξεκινήσετε ξανά από την πρώτη εγγραφή:

```
sethostent(0);
my $first = (gethostent())[0];
sethostent(0);                          # back to the top
my $first_again = (gethostent())[0];
```

Οριακές περιπτώσεις

- **Το STAYOPEN είναι υποχρεωτικό συντακτικά, όχι σημασιολογικά.** Το όρισμα πρέπει να υπάρχει - το `sethostent;` είναι σφάλμα μεταγλώττισης - αλλά πολλές πλατφόρμες αγνοούν τη σημαία και πάντα κρατούν τη βάση ανοιχτή για τη διάρκεια της επανάληψης. Περάστε `0` όταν δεν έχετε προτίμηση.
- **Δεν έχει κάθε πλατφόρμα ουσιαστικό επαναλήπτη υπολογιστών.** Σε συστήματα όπου η βάση δεδομένων υπολογιστών εξυπηρετείται αποκλειστικά από DNS ή υπηρεσία ονομάτων που δεν εκθέτει απαρίθμηση, η `gethostent` επιστρέφει την κενή λίστα στην πρώτη κλήση ανεξάρτητα από την `sethostent`. Η ίδια η κλήση και πάλι πετυχαίνει.
- **Ασφάλεια σε threads.** Ο δρομέας του resolver είναι ανά διεργασία, όχι ανά thread. Δύο threads που επαναλαμβάνουν ταυτόχρονα θα διαπλέξουν εγγραφές και κανένα δεν θα δει την πλήρη λίστα. Η upstream Perl αντικαθιστά παραλλαγές `*_r` για σημειακές αναζητήσεις όπως η `gethostbyname` όπου είναι διαθέσιμες· δεν υπάρχει thread-safe παραλλαγή του επαναλήπτη.
- **Αλληλεπίδραση με τις cache της `gethostby*`.** Σε ορισμένες υλοποιήσεις libc, ένα `sethostent(1)` πριν από μια ριπή κλήσεων `gethostbyname` / `gethostbyaddr` κρατάει το υποκείμενο αρχείο ανοιχτό μεταξύ ερωτημάτων· σε άλλες δεν έχει κανένα αποτέλεσμα. Μετρήστε πριν βασιστείτε σε αυτό ως βελτιστοποίηση.
- **Συνδυάστε με `endhostent`.** Αφήνοντας τη βάση ανοιχτή μετά την επανάληψη μπορεί να διαρρεύσει περιγραφέας αρχείου ή σύνδεση resolver. Πάντα κλείνετε τη διάσχιση με `endhostent`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `gethostent` - διαβάστε την επόμενη εγγραφή από τη βάση δεδομένων υπολογιστών αφού η `sethostent` έχει εκκινήσει τον δρομέα
- `endhostent` - κλείστε τη βάση και απελευθερώστε την κατάσταση του resolver όταν τελειώσει η επανάληψη

- `gethostbyname` - σημειακή αναζήτηση με όνομα· επωφελείται από `sethostent(1)` σε πλατφόρμες που το τιμούν
- `gethostbyaddr` - σημειακή αναζήτηση με διεύθυνση· ίδια επιφύλαξη με την `gethostbyname`
- `setnetent` - το αντίστοιχο για τη βάση δεδομένων δικτύων, ίδιο μοτίβο επανάληψης
- `setservent` - το αντίστοιχο για τη βάση δεδομένων υπηρεσιών

Πληροφορίες δικτύου

setnetent

Ανοίγει ή επαναφέρει στην αρχή τη βάση δεδομένων δικτύων για επανάληψη.

Η `setnetent` προετοιμάζει τη βάση δεδομένων δικτύων του συστήματος ώστε οι επόμενες κλήσεις στην `getnetent` να επιστρέφουν εγγραφές ξεκινώντας από την πρώτη. Είναι άμεσο περιτύλιγμα της `setnetent(3)` της C library και διαβάζει την ίδια υποκείμενη πηγή με την κλήση C - τυπικά το `/etc/networks` στο Linux, ή ό,τι έχουν διαμορφωθεί να παρέχουν τα `nsswitch.conf` / backends NSS.

Το όρισμα `STAYOPEN` είναι υπόδειξη προς τη C library: αληθής τιμή της ζητά να διατηρήσει το αρχείο της βάσης δεδομένων ανοιχτό κατά τη διάρκεια σχετικών κλήσεων (`getnetbyname`, `getnetbyaddr`, `getnetent`) ώστε οι μεμονωμένες αναζητήσεις να μην πληρώνουν η καθεμία το κόστος ανοίγματος και κλεισίματος του αρχείου. Στη σύγχρονη glibc αυτή η σημαία ουσιαστικά αγνοείται - η βάση δεδομένων χειρίζεται αποδοτικά είτε έτσι είτε αλλιώς - αλλά το να την περάσετε δεν κοστίζει τίποτα και κρατά τον κώδικα φορητό σε συστήματα όπου εξακολουθεί να έχει σημασία.

- but passing it costs nothing and keeps the code portable to systems where it still matters.

Σύνοψη

```
setnetent STAYOPEN
setnetent 0
setnetent 1
```

Τι επιστρέφεται

Η `setnetent` δεν επιστρέφει τίποτα ουσιαστικό. Σε περιβάλλον λίστας επιστρέφει κενή λίστα, σε βαθμωτό περιβάλλον `undef`. Μη βασίζεστε στην τιμή επιστροφής· η επίδραση της κλήσης είναι η παρενέργεια της τοποθέτησης του επαναλήπτη της βάσης στην αρχή.

Αν η υποκείμενη κλήση C αποτύχει (για παράδειγμα το αρχείο της βάσης δεδομένων λείπει ή είναι μη αναγνώσιμο) το σφάλμα είναι σιωπηρό από τη σκοπιά της Perl - η `getnetent` απλώς θα επιστρέψει κενή λίστα στην πρώτη κλήση.

Καθολική κατάσταση που επηρεάζει

- Τον **δρομέα βάσης δικτύων** που είναι καθολικός σε επίπεδο διεργασίας και διατηρείται από τη C library. Κάθε κλήση στην `getnetent` προωθεί αυτόν τον δρομέα· η `setnetent` τον επαναφέρει στην αρχή. Η `endnetent` κλείνει τη βάση δεδομένων και απελευθερώνει όποιους πόρους κρατά μια υπόδειξη `STAYOPEN`.
- Δεν διαβάζεται ούτε γράφεται καμία ειδική μεταβλητή της Perl.

Επειδή ο δρομέας είναι ανά διεργασία, όχι ανά thread, η διαπλοκή επανάληψης βάσης δεδομένων δικτύων από πολλά threads στον ίδιο διερμηνέα θα αλλοιώσει τη θέση κάθε επαναλήπτη. Εκτελέστε την επανάληψη από ένα μόνο thread, ή σειριοποιήστε την πρόσβαση.

Παραδείγματα

Επαναφορά της βάσης δεδομένων και επανάληψη από την πρώτη εγγραφή:

```
setnetent(1);
while (my ($name, $aliases, $addrtype, $net) = getnetent()) {
    printf "%-20s %d\n", $name, $net;
}
endnetent();
```

Επανεκτέλεση της επανάληψης από την κορυφή χωρίς κλείσιμο στο ενδιάμεσο. Η δεύτερη `setnetent` επαναφέρει στην αρχή, οπότε ο δεύτερος βρόχος βλέπει κάθε εγγραφή ξανά:

```
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;
}
```

Ζευγάρωμα με το `Net::netent` για διεπαφή ονομασμένων πεδίων πάνω από την ίδια πηγή δεδομένων - η `setnetent` εξακολουθεί να ελέγχει τον δρομέα:

```
use Net::netent;
setnetent(1);
while (my $n = getnetent()) {
    printf "%s -> %s\n", $n->name, $n->net;
}
endnetent();
```

Οριακές περιπτώσεις

- **Το `STAYOPEN` είναι υπόδειξη, όχι εγγύηση**. Σε συστήματα όπου η C library το αγνοεί, το πέρασμα `1` και το πέρασμα `0` παράγουν πανομοιότυπη παρατηρήσιμη συμπεριφορά.
- **Καμία σιωπηρή `setnetent` πριν την `getnetent`**. Φρέσκια διεργασία που καλεί την `getnetent` χωρίς να καλέσει πρώτα την `setnetent` εξακολουθεί να ξεκινά από την πρώτη εγγραφή στα περισσότερα συστήματα, αλλά αυτό είναι σύμβαση της libc και όχι εγγύηση. Καλέστε ρητά την `setnetent` όταν θέλετε να είστε σίγουροι.
- **Η κλήση `setnetent` στη μέση της επανάληψης επαναφέρει τον δρομέα**. Οποιοσδήποτε εν εξελίξει βρόχος `while (getnetent)` ξαναξεκινά από την κορυφή - χρήσιμο για σκόπιμη δεύτερη διέλευση, bug αν συμβεί κατά λάθος από βοηθητική sub που επίσης επαναλαμβάνει.
- **Αλληλεπίδραση με `getnetbyname` / `getnetbyaddr`**. Σε συστήματα όπου η C library τιμά το `STAYOPEN`, μια αναζήτηση βάσει ονόματος ή διεύθυνσης κατά τη διάρκεια της επανάληψης *δεν* θα διαταράξει τον επαναλήπτη. Σε συστήματα που αγνοούν την υπόδειξη, κάθε αναζήτηση μπορεί να επαναφέρει τον δρομέα. Μην αναμιγνύετε αναζητήσεις βάσει ονόματος με επανάληψη `getnetent` αν χρειάζεστε σταθερή τοποθέτηση.
- **Δικαιώματα**. Η βάση δεδομένων δικτύων είναι τυπικά αναγνώσιμη από όλους, αλλά τα backends NSS (LDAP, `sssd`) μπορεί να επιβάλλουν τη δική τους εξουσιοδότηση. Μια αναζήτηση που δούλεψε ως `root` μπορεί να μην αποδώσει τίποτα ως κανονικός χρήστης χωρίς η Perl να δει σφάλμα.
- **Κενό ή ελλείπον `/etc/networks`**. Στα περισσότερα συστήματα Linux αυτό το αρχείο είναι κενό ή απουσιάζει εξ ορισμού. Η `setnetent` εξακολουθεί να πετυχαίνει· η ακόλουθη `getnetent` επιστρέφει κενή λίστα στην πρώτη κλήση. Αυτό δεν είναι κατάσταση σφάλματος.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `getnetent` - ο επαναλήπτης που προετοιμάζει η `setnetent`· σε κάθε κλήση πιάνει την επόμενη γραμμή
- `endnetent` - κλείνει τη βάση δεδομένων και απελευθερώνει όποιους πόρους κρατά ανοιχτούς το `STAYOPEN`· καλέστε την όταν η επανάληψη ολοκληρωθεί
- `getnetbyname` - αναζήτηση μίας εγγραφής βάσει ονόματος χωρίς εκτέλεση του επαναλήπτη
- `getnetbyaddr` - αναζήτηση μίας εγγραφής βάσει πακεταρισμένης διεύθυνσης χωρίς εκτέλεση του επαναλήπτη
- `sethostent` - το ίδιο ιδίωμα για τη βάση δεδομένων υπολογιστών
- `setservent` - το ίδιο ιδίωμα για τη βάση δεδομένων υπηρεσιών
- `Net::netent` - αντικειμενοστραφές περιτύλιγμα που επιστρέφει ονομασμένα πεδία αντί για λίστα κατά θέση

Διεργασίες

setpgrp

Ορίζει την ομάδα διεργασιών μιας διεργασίας.

Η `setpgrp` μετακινεί τη διεργασία που αναγνωρίζεται από το `PID` στην ομάδα διεργασιών που αναγνωρίζεται από το `PGRP`. Ένα `PID` ίσο με `0` σημαίνει την τρέχουσα διεργασία· ένα `PGRP` ίσο με `0` σημαίνει «χρησιμοποίησε το `PID` ως το νέο id ομάδας» - δηλαδή, κάνε τη διεργασία αρχηγό ομάδας. Χωρίς ορίσματα παίρνει προεπιλογή `setpgrp(0, 0)`, που κάνει την τρέχουσα διεργασία αρχηγό μιας νέας ομάδας διεργασιών της οποίας το id ισούται με το δικό της pid.

Αυτή είναι η διεπαφή Perl προς την υποκείμενη κλήση συστήματος `setpgid(2)` / `setpgrp(2)`. Είναι το δομικό στοιχείο για την αποσύνδεση μιας διεργασίας από το ελέγχον τερματικό της και για σχήματα ελέγχου εργασιών που θέλουν να στείλουν σήματα σε ολόκληρη ομάδα ταυτόχρονα με την `kill`.

Σύνοψη

```
setpgrp PID, PGRP
setpgrp           # same as setpgrp(0, 0)
```

Τι επιστρέφεται

1 σε επιτυχία, `undef` σε αποτυχία με την `$!` τεθειμένη στον λόγο (συνήθως `EPERM` όταν η διεργασία-στόχος δεν είναι ο καλών ή παιδί του καλούντος, ή `ESRCH` όταν δεν υπάρχει τέτοια διεργασία). Ελέγξτε την τιμή επιστροφής - η σιωπηλή συνέχιση με μια διεργασία που εξακολουθεί να βρίσκεται στην παλιά ομάδα δεν είναι σχεδόν ποτέ αυτό που θέλετε.

```
setpgrp(0, 0)
    or die "setpgrp failed: $!";
```

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται στην τιμή `errno` σε αποτυχία. Δεν αγγίζεται σε επιτυχία.

Παραδείγματα

Αποσπάστε την τρέχουσα διεργασία σε δική της νέα ομάδα διεργασιών. Αυτό είναι το κλασικό πρώτο βήμα μιας ακολουθίας daemonisation, που τρέχει μετά τη `fork` και πριν το κλείσιμο των τυπικών handles:

```
setpgrp(0, 0)
    or die "setpgrp failed: $!";
```

Μετακινήστε ένα μόλις δημιουργηθέν παιδί στη δική του ομάδα ώστε ένα `SIGTERM` που στέλνεται στην ομάδα να σκοτώσει το παιδί και τυχόν εγγόνια που δημιουργεί, χωρίς να αγγίξει τον γονέα:

```
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;
```

Διαβάστε το id ομάδας πίσω μετά τον ορισμό του, χρησιμοποιώντας την `getpgrp`:

```
setpgrp(0, 0) or die $!;
my $pgid = getpgrp(0);
print "now leader of group $pgid\n";
```

Χειριστείτε τη λειτουργία αποτυχίας φορητότητας - σε πλατφόρμα χωρίς την υποκείμενη κλήση συστήματος η συνάρτηση εγείρει εξαίρεση αντί να επιστρέφει `undef`:

```
eval { setpgrp(0, 0) };
if ($@) {
    warn "no process-group support on this platform: $@";
}
```

Οριακές περιπτώσεις

- **Η μορφή χωρίς ορίσματα παίρνει προεπιλογή `(0, 0)`.** Η γραφή `setpgrp;` είναι η ίδια με `setpgrp(0, 0)` - κάνει την τρέχουσα διεργασία νέο αρχηγό ομάδας. Η μορφή χωρίς ορίσματα είναι η φορητή μορφή (δείτε παρακάτω).
- **Συμβατότητα BSD 4.2.** Η ιστορική `setpgrp` του BSD 4.2 δεν έπαιρνε ορίσματα. Σενάρια που πρέπει να τρέχουν σε τέτοια συστήματα μπορούν να βασιστούν μόνο στις μορφές `setpgrp(0, 0)` / `setpgrp()`· κάθε άλλος συνδυασμός ορισμάτων είναι μη φορητός.
- **Εγείρει εξαίρεση, δεν επιστρέφει `undef`, σε μη υποστηριζόμενες πλατφόρμες.** Σε μηχάνημα που δεν υλοποιεί ούτε την POSIX `setpgid(2)` ούτε την BSD `setpgrp(2)`, η `setpgrp` εγείρει εξαίρεση. Αυτή είναι η μοναδική λειτουργία αποτυχίας που **δεν** περνάει μέσα από την `$!` - τυλίξτε σε `eval` αν χρειάζεται να υποβαθμιστείτε ομαλά.
- **`EPERM` σε μετακινήσεις μεταξύ συνεδριών.** Ο πυρήνας απορρίπτει προσπάθειες μετακίνησης μιας διεργασίας σε ομάδα διαφορετικής συνεδρίας, ή μετακίνησης μιας διεργασίας που έχει ήδη καλέσει `exec`. Πιάστε την `$! == EPERM` και υποχωρήστε σε εφεδρεία αντί να ξαναπροσπαθείτε.
- **Αποστολή σήματος σε ολόκληρη την ομάδα.** Όταν δημιουργηθεί μια ομάδα, το `kill SIG, -$pgid` παραδίδει το `SIG` σε κάθε διεργασία της ομάδας. Η σύμβαση του αρνητικού pid είναι ιδιότητα της `kill`, όχι της `setpgrp` - αλλά είναι ο λόγος που συνήθως καλείτε την `setpgrp` εξαρχής.
- **Όχι αρχηγός συνεδρίας.** Η `setpgrp` αλλάζει ομάδα διεργασιών, όχι συνεδρία. Για να αποσυνδεθείτε εντελώς από το ελέγχον τερματικό χρειάζεστε `POSIX::setsid`, που δημιουργεί νέα συνεδρία και νέα ομάδα σε ένα βήμα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `getpgrp` - διαβάστε το τρέχον id ομάδας διεργασιών· ο προφανής συνεργάτης για επαλήθευση ότι μια κλήση `setpgrp` είχε αποτέλεσμα
- `POSIX::setpgid` - ο POSIX-ονομασμένος περιτυλιγμός γύρω από την ίδια κλήση συστήματος, προτιμητέος σε νέο κώδικα επειδή η σειρά και η σημασιολογία των ορισμάτων του είναι μη αμφίσημες σε όλες τις πλατφόρμες
- `POSIX::setsid` - δημιουργεί νέα συνεδρία *και* νέα ομάδα διεργασιών σε μία κλήση· χρησιμοποιήστε αυτό, όχι το `setpgrp`, για πλήρη αποσύνδεση από τερματικό
- `fork` - η κλήση που σχεδόν πάντα προηγείται της `setpgrp`· ένα παιδί που θέλει τη δική του ομάδα καλεί `setpgrp(0, 0)` αμέσως μετά την επιστροφή `0` της `fork`
- `kill` - πώς πραγματικά χρησιμοποιείτε μια ομάδα διεργασιών μόλις την έχετε φτιάξει: το `kill SIG, -$pgid` στέλνει σήμα σε κάθε μέλος

Διεργασίες

setpriority

Ορίζει την προτεραιότητα χρονοπρογραμματισμού (nice value) μιας διεργασίας, ομάδας διεργασιών, ή χρήστη.

Η `setpriority` είναι το περιτύλιγμα σε επίπεδο Perl γύρω από την κλήση συστήματος `setpriority(2)`. Το `WHICH` επιλέγει την κλάση στόχου (`PRIO_PROCESS`, `PRIO_PGRP`, ή `PRIO_USER`), το `WHO` κατονομάζει τον συγκεκριμένο στόχο εντός αυτής της κλάσης (το `0` σημαίνει «εαυτός» ή «τρέχων» ανάλογα με το `WHICH`), και το `PRIORITY` είναι η νέα nice value. Υψηλότερες τιμές σημαίνουν *χαμηλότερη* προτεραιότητα χρονοπρογραμματισμού· στο Linux το χρήσιμο εύρος είναι `-20` (πιο ευνοϊκή) έως 19 (λιγότερο ευνοϊκή).

Σύνοψη

```
setpriority WHICH, WHO, PRIORITY
```

Τι επιστρέφεται

Επιστρέφει αληθές σε επιτυχία, `undef` σε αποτυχία (με το `$!` ρυθμισμένο στο υποκείμενο `errno`). Οι συνηθέστερες αποτυχίες είναι `EACCES` ή `EPERM` - η μείωση της nice value κάτω από την τρέχουσα, ή το άγγιγμα διεργασιών άλλου χρήστη, απαιτεί προνόμιο. Ελέγχετε πάντα την τιμή επιστροφής όταν η αλλαγή προτεραιότητας δεν είναι απλώς αισθητική:

```
setpriority(PRIO_PROCESS, $$, 10)
    or die "cannot renice self: $!";
```

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται σε αποτυχία στο errno της υποκείμενης κλήσης `setpriority(2)`.

Οι τρεις κλάσεις WHICH

Το WHICH είναι μία από τρεις ακέραιες σταθερές, καθεμία επιλέγει διαφορετική ερμηνεία του WHO:

- PRIO_PROCESS - το WHO είναι ID διεργασίας. Το 0 σημαίνει την τρέχουσα διεργασία.
- PRIO_PGRP - το WHO είναι ID ομάδας διεργασιών. Το 0 σημαίνει την ομάδα διεργασιών της τρέχουσας διεργασίας.
- PRIO_USER - το WHO είναι αριθμητικό ID χρήστη (UID). Το 0 σημαίνει το πραγματικό UID της τρέχουσας διεργασίας. Ο ορισμός κατά χρήστη αλλάζει την προτεραιότητα *κάθε* διεργασίας που ανήκει σε αυτόν τον χρήστη.

Αυτές οι σταθερές δεν είναι ενσωματωμένες - εισαγάγετέ τες από το POSIX:

```
use POSIX qw(PRIO_PROCESS PRIO_PGRP PRIO_USER);
```

Παραδείγματα

Επαναπροσδιορισμός της nice value της τρέχουσας διεργασίας σε 10 (πιο φιλικό προς άλλη εργασία):

```
use POSIX qw(PRIO_PROCESS);
setpriority(PRIO_PROCESS, 0, 10)
    or die "renice failed: $!";
```

Επαναπροσδιορισμός nice value σε διεργασία παιδί μετά από 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;
}
```

Επαναπροσδιορισμός nice value για κάθε διεργασία στην τρέχουσα ομάδα διεργασιών - χρήσιμο σε επικεφαλής συνεδρίας που θέλει ολόκληρη η pipeline του να παραχωρήσει:

```
use POSIX qw(PRIO_PGRP);
setpriority(PRIO_PGRP, 0, 5)
    or warn "could not renice process group: $!";
```

Συνδυάστε με την getpriority για ρύθμιση σχετικά ως προς την τρέχουσα τιμή αντί για απόλυτο ορισμό:

```
use POSIX qw(PRIO_PROCESS);
my $cur = getpriority(PRIO_PROCESS, 0);
setpriority(PRIO_PROCESS, 0, $cur + 5)
    or die "renice failed: $!";
```

Οριακές περιπτώσεις

- **Οι αρνητικές προτεραιότητες απαιτούν προνόμιο.** Η μείωση της nice value (πιο ευνοϊκός χρονοπρογραμματισμός) κάτω από την τρέχουσα χρειάζεται `CAP_SYS_NICE` στο Linux, ή root. Μια ανεξουσιοδότητη προσπάθεια αποτυγχάνει με `EACCES` ή `EPERM` ανάλογα με την έκδοση του πυρήνα.
- **Οι nice values περικόπτονται.** Ο πυρήνας περικόπτει σιωπηλά το `PRIORITY` στο υποστηριζόμενο εύρος (`-20..19` στο Linux). Το πέρασμα `100` *δεν* αποτυγχάνει· καταλήγει σε 19.
- **Το `WHO = 0` εξαρτάται από το περιβάλλον.** Η σημασία του εξαρτάται από το `WHICH`: τρέχον PID για `PRIO_PROCESS`, τρέχουσα ομάδα διεργασιών για `PRIO_PGRP`, πραγματικό UID για `PRIO_USER`. Δεν είναι καθολικό σύμβολο «εαυτού».
- **Ανύπαρκτος στόχος.** Ένα PID, PGID, ή UID που δεν αντιστοιχεί σε καμία ενεργή διεργασία αποτυγχάνει με `ESRCH`.
- **Πλατφόρμα χωρίς `setpriority(2)`.** Η Perl εγείρει μοιραία εξαίρεση (`setpriority() not implemented`). Το pperl στοχεύει μόνο Linux, οπότε αυτή η περίπτωση δεν προκύπτει στην πράξη, αλλά ο φορητός κώδικας θα πρέπει παρά ταύτα να προστατευτεί με `eval`.
- **Προσημασμένη επιστροφή από την `getpriority` έναντι επιτυχίας της `setpriority`.** Η `getpriority` μπορεί νόμιμα να επιστρέψει `-1` ως nice value, και γι' αυτό θέτει το `$!` για να διακρίνει σφάλμα από αποτέλεσμα· η `setpriority` δεν έχει τέτοια αμφισημία - αντιμετωπίστε την επιστροφή της καθαρά ως λογική τιμή.
- **Η nice value είναι ανά διεργασία, όχι ανά νήμα.** Στο Linux αυτό διαφέρει από το POSIX: ο πυρήνας εφαρμόζει τη nice value στο καλούν task (νήμα). Η Perl είναι μονονηματική σε επίπεδο διερμηνέα, οπότε αυτό έχει σημασία μόνο αν κώδικας XS έχει δημιουργήσει νήματα πυρήνα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `getpriority` - διαβάζει την τρέχουσα nice value· ο φυσικός αντίστοιχος όταν ρυθμίζετε σχετικά ως προς την υπάρχουσα προτεραιότητα
- `POSIX` - πηγή των σταθερών `PRIO_PROCESS`, `PRIO_PGRP` και `PRIO_USER`
- `fork` - τυπικός συνδυασμός: δημιουργήστε παιδί, και μετά αλλάξτε του τη nice value πριν την `exec`
- `$!` - κρατά το `errno` που περιγράφει γιατί μια αποτυχημένη κλήση `setpriority` επέστρεψε `undef`
- `kill` - άλλη πράξη ελέγχου ανά διεργασία με την ίδια σύμβαση «στόχευση κατά PID ή PGID»

Πληροφορίες δικτύου

setprotoent

Ανοίγει τη βάση δεδομένων πρωτοκόλλων και την προετοιμάζει για σειριακές αναγνώσεις.

Η `setprotoent` επανατυλίγει (ή ανοίγει) τη βάση δεδομένων πρωτοκόλλων του συστήματος - στα περισσότερα Unix-like συστήματα αυτό είναι το `/etc/protocols` - ώστε μια επόμενη ακολουθία κλήσεων `getprotoent` να ξεκινήσει από την πρώτη εγγραφή. Είναι το πρώτο σκέλος της τριάδας επανάληψης `setprotoent` / `getprotoent` / `endprotoent`. Το μοναδικό όρισμα `STAYOPEN` είναι hint προς τη βιβλιοθήκη C για το αν το file handle που υποστηρίζει την επανάληψη πρέπει να μένει ανοιχτό μεταξύ κλήσεων: αληθής τιμή ζητάει από τη βιβλιοθήκη να το κρατήσει ανοιχτό (γρηγορότερο για μακρές διασχίσεις), ψευδής τιμή της επιτρέπει να κλείνει και να ξανανοίγει ανά κλήση.

Η κλήση των `getprotobyname` ή `getprotobynumber` σε οποιοδήποτε σημείο στη μέση της επανάληψης μπορεί να επαναφέρει τη θέση διάσχισης σε ορισμένες πλατφόρμες· δείτε *Οριακές περιπτώσεις*.

Σύνοψη

```
setprotoent STAYOPEN
setprotoent(1)
```

Τι επιστρέφεται

Καμία ουσιαστική τιμή επιστροφής. Όπως και οι συγγενείς της `endprotoent` και `sethostent`, η `setprotoent` καλείται για την παρενέργειά της: επανατύλιγμα της επανάληψης και, ανάλογα με το `STAYOPEN`, καθήλωση του περιγραφέα ανοιχτού. Δεν υπάρχει τίποτα που να αξίζει να ελέγξετε στην τιμή επιστροφής της.

Καθολική κατάσταση που επηρεάζει

Η `setprotoent` χειρίζεται έναν **ανά διεργασία** δρομέα στη βάση δεδομένων πρωτοκόλλων που ανήκει στη βιβλιοθήκη C. Αυτή η κατάσταση μοιράζεται σε κάθε επόμενη κλήση `getprotoent` στην ίδια διεργασία - συμπεριλαμβανομένων κλήσεων από άλλα αρθρώματα που δεν γράψατε εσείς. Κώδικας που διασχίζει τη βάση πρέπει είτε να την αποστραγγίσει πλήρως με την `endprotoent` είτε να δεχτεί ότι ένας μεταγενέστερος καλών μπορεί να δει μια επανάληψη ήδη σε εξέλιξη. Δεν διαβάζονται ή γράφονται ειδικές μεταβλητές σε επίπεδο Perl.

Παραδείγματα

Ανοίξτε τη βάση και επαναλάβετε κάθε εγγραφή, κρατώντας τον υποκείμενο περιγραφέα ζωντανό για όλη τη διάσχιση:

```
setprotoent(1);                          # 1 = keep file open between
↪calls
while (my @p = getprotoent()) {
    my ($name, $aliases, $proto) = @p;
    print "$proto\t$name\n";
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
}
endprotoent();
```

Επανεκκινήστε μια εν εξελίξει επανάληψη από την αρχή:

```
setprotoent(1);
my @first  = getprotoent();
my @second = getprotoent();

setprotoent(1);                         # rewind
my @again  = getprotoent();             # same as @first
endprotoent();
```

Αφήστε τη βιβλιοθήκη C να κλείνει και να ξανανοίγει το `/etc/protocols` σε κάθε `getprotoent` - φθηνότερο σε μνήμη, ακριβότερο σε κλήσεις συστήματος, χρήσιμο μόνο όταν διαβάζετε λίγες εγγραφές:

```
setprotoent(0);
my @tcp = getprotoent();
endprotoent();
```

Συνδυάστε με αναζήτηση μίας εκτέλεσης - οι `getprotobyname` και `getprotobynumber` δεν συμμετέχουν στην επανάληψη και δεν χρειάζονται `setprotoent`:

```
my @tcp = getprotobyname('tcp');        # no setprotoent needed
```

Οριακές περιπτώσεις

- **Το STAYOPEN είναι hint, όχι συμβόλαιο**: πολλές βιβλιοθήκες C το αγνοούν εντελώς και συμπεριφέρονται σαν ο περιγραφέας να μένει πάντα ανοιχτός. Μη βασίζεστε στο όρισμα για έλεγχο της χρήσης πόρων σε λεπτομερές επίπεδο.
- **Ανάμειξη με αναζητήσεις μίας εκτέλεσης**: η κλήση των `getprotobyname` ή `getprotobynumber` κατά τη διάρκεια διάσχισης μπορεί να επαναφέρει τον δρομέα επανάληψης σε ορισμένες πλατφόρμες. Φορητός κώδικας αποστραγγίζει πρώτα τη διάσχιση, ή ξανακαλεί την `setprotoent` μετά για να ξεκινήσει από την αρχή.
- **Μη επανεισόδιμη**: ο δρομέας της βάσης δεδομένων πρωτοκόλλων είναι ένας μόνο καθολικός πόρος μέσα στη βιβλιοθήκη C. Δύο ταυτόχρονες επαναλήψεις στην ίδια διεργασία (threads, χειριστές σημάτων, οτιδήποτε) αλληλοαλλοιώνονται. Χρησιμοποιήστε μία διάσχιση τη φορά.
- **Το όρισμα είναι υποχρεωτικό σε επίπεδο γλώσσας**: η `setprotoent` χωρίς όρισμα είναι συντακτικό σφάλμα - η ενσωματωμένη έχει πληθικότητα ένα. Περάστε `0` αν δεν έχετε προτίμηση.
- **Πλατφόρμες χωρίς βάση δεδομένων πρωτοκόλλων**: σε συστήματα όπου η βιβλιοθήκη C δεν έχει αντίστοιχο του `/etc/protocols`, η κλήση είναι no-op αντί για σφάλμα. Δείτε το `perlport` για την πλήρη λίστα ιδιορρυθμιών βάσης δεδομένων δικτύου.
- **Μη tainted αποτέλεσμα της επανάληψης**: οι εγγραφές που επιστρέφει η `getprotoent` μετά την `setprotoent` προέρχονται από αρχείο που διαβάζει

ο πυρήνας εκ μέρους σας και δεν σημειώνονται ως tainted. Παρ” όλα αυτά αντιμετωπίστε τες ως μη έμπιστες αν η βάση μπορεί να επεξεργαστεί από χρήστες.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `getprotoent` - διαβάστε την επόμενη εγγραφή· το σώμα του βρόχου που στήνει η `setprotoent`
- `endprotoent` - κλείστε τη βάση μετά την επανάληψη· ο τερματιστής για την τριάδα που ξεκινάει η `setprotoent`
- `getprotobyname` - αναζήτηση μίας εκτέλεσης με όνομα· δεν συμμετέχει στην επανάληψη
- `getprotobynumber` - αναζήτηση μίας εκτέλεσης με αριθμό πρωτοκόλλου· ομοίως ανεξάρτητη
- `sethostent` - αντίστοιχη opener για τη βάση δεδομένων υπολογιστών· ίδιο ιδίωμα, διαφορετικός πίνακας
- `setservent` - αντίστοιχη opener για τη βάση δεδομένων υπηρεσιών

Πληροφορίες χρηστών και ομάδων

setpwent

Επανατυλίγει τον επαναλήπτη της βάσης δεδομένων passwd ώστε η επόμενη κλήση `getpwent` να επιστρέψει ξανά την πρώτη εγγραφή.

Η `setpwent` τυλίγει την κλήση `setpwent(3)` της βιβλιοθήκης C. (Ξανα)ανοίγει τη βάση δεδομένων passwd του συστήματος και επαναφέρει τον δρομέα επανάληψης που χρησιμοποιεί η `getpwent`, ώστε ένας επόμενος βρόχος να ξεκινήσει από την αρχή. Χρησιμοποιήστε την όταν θέλετε να επαναλάβετε τη βάση passwd πάνω από μία φορά στην ίδια διεργασία, ή όταν θέλετε να επανεκκινήσετε την επανάληψη μετά από μερική διάσχιση. Συνδυάστε την με `endpwent` για να απελευθερώσετε τη βάση όταν τελειώσετε.

Σύνοψη

```
setpwent;
setpwent();
```

Τι επιστρέφεται

Η τιμή επιστροφής **δεν είναι χρήσιμη** και δεν πρέπει να βασίζεστε σε αυτήν. Στα περισσότερα συστήματα η υποκείμενη `setpwent(3)` έχει επιστροφή `void`· η Perl παράγει μια μοναδική ανούσια αληθή τιμή. Καλέστε την `setpwent` για την παρενέργειά της, ποτέ για την τιμή της.

Καθολική κατάσταση που επηρεάζει

- Τον δρομέα ανά διεργασία που μοιράζονται οι `getpwent`, `setpwent`, και `endpwent`. Αυτός ο δρομέας ανήκει στη βιβλιοθήκη C, όχι στην Perl - το fork ενός παιδιού κληρονομεί ένα ανοιχτό handle βάσης δεδομένων, και δύο threads που επαναλαμβάνουν ταυτόχρονα θα διαπλέξουν και θα χάσουν εγγραφές.
- Η `$!` **δεν** τίθεται από την `setpwent`· σφάλματα από την υποκείμενη `setpwent(3)` καταπίνονται από την πλατφόρμα.

Παραδείγματα

Επαναλάβετε τη βάση δεδομένων passwd δύο φορές στο ίδιο πρόγραμμα:

```
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;
```

Επανεκκινήστε την επανάληψη μετά από πρόωρη έξοδο:

```
while (my @pw = getpwent) {
    last if $pw[0] eq 'root';
}
setpwent;                       # rewind so the next walk sees root␣
↪again
```

Τυπικό μοτίβο καθαρισμού - πάντα καλείτε την `endpwent` όταν τελειώσετε, ανεξάρτητα από το αν καλέσατε την `setpwent`:

```
setpwent;
while (my ($name, undef, $uid) = getpwent) {
    print "$name => $uid\n" if $uid >= 1000;
}
endpwent;
```

Οριακές περιπτώσεις

- **Κανένα όρισμα.** Η `setpwent` δεν παίρνει ορίσματα. Η γραφή `setpwent $x` είναι συντακτική παγίδα: όπως και άλλοι τελεστές λίστας, θα προσπαθήσει να καταναλώσει την `$x` ως πρώτο στοιχείο μιας λίστας.
- **Πριν από οποιαδήποτε κλήση getpwent.** Ασφαλές. Η `setpwent` ανοίγει τη βάση αν δεν είναι ήδη ανοιχτή, οπότε διπλασιάζεται ως ρητή κλήση «open».
- **Μετά την endpwent.** Ασφαλές. Η βάση ξανανοίγει.
- **Forked παιδιά.** Κάθε διεργασία έχει τον δικό της δρομέα επανάληψης μετά το fork, αλλά ο υποκείμενος περιγραφέας αρχείου μπορεί να είναι κοινός· διαπλεκόμενες

αναγνώσεις από γονέα και παιδί παράγουν εγγραφές που λείπουν ή είναι διπλές. Καλέστε `setpwent` στο παιδί για να αποκτήσετε καθαρό επαναλήπτη.

- **Threads.** Η επανάληψη της `getpwent` είναι ανά διεργασία, όχι ανά thread. Ταυτόχρονη επανάληψη από πολλαπλά threads χάνει εγγραφές. Η upstream Perl αντικαθιστά thread-safe παραλλαγές `_r` για τις `getpwnam` και `getpwuid` όταν η πλατφόρμα τις παρέχει· δεν υπάρχει `setpwent_r`, και αυτή η συνάρτηση παραμένει καθολική σε επίπεδο διεργασίας.
- **Βάσεις δεδομένων υποστηριζόμενες από NSS.** Σε συστήματα όπου το `/etc/nsswitch.conf` δρομολογεί το `passwd` μέσω LDAP, SSSD, ή παρόμοιων, η `setpwent` μπορεί να μπλοκάρει σε δικτυακό round-trip. Μην την καλείτε από χειριστές σημάτων ή διαδρομές κώδικα κρίσιμες ως προς την καθυστέρηση.
- **Shadow passwords.** Η `setpwent` δεν επηρεάζει τον επαναλήπτη shadow-password (αν υπάρχει)· επαναφέρει μόνο την κανονική διάσχιση `passwd`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `getpwent` - λάβετε την επόμενη εγγραφή· η συνάρτηση που υπάρχει για να επανατυλίγει η `setpwent`
- `endpwent` - απελευθερώστε τη βάση δεδομένων passwd όταν τελειώσετε την επανάληψη
- `getpwnam` - άμεση αναζήτηση με όνομα σύνδεσης· δεν αλληλεπιδρά με τον δρομέα επανάληψης
- `getpwuid` - άμεση αναζήτηση με uid· δεν αλληλεπιδρά με τον δρομέα επανάληψης
- `setgrent` - η ίδια λειτουργία επανατυλίγματος για τη βάση δεδομένων ομάδων
- `User::pwent` - αντικειμενοστραφής περιτυλιγμός που επιστρέφει αντικείμενα με κατονομασμένα πεδία αντί για λίστες 10 στοιχείων

Πληροφορίες δικτύου

setservent

Επανατυλίγει τη βάση δεδομένων υπηρεσιών και προαιρετικά την κρατά ανοιχτή σε όλη τη διάρκεια των αναζητήσεων.

Η `setservent` είναι το αντίστοιχο για βάση δεδομένων υπηρεσιών του `setservent(3)` της βιβλιοθήκης C. Επαναφέρει τον εσωτερικό δρομέα που χρησιμοποιεί η `getservent` ώστε η επόμενη κλήση να ξεκινήσει πάλι από την πρώτη εγγραφή του `/etc/services` (ή όποιας άλλης πηγής έχει διαμορφωθεί να χρησιμοποιεί ο επιλυτής συστήματος), και ελέγχει αν το υποκείμενο αρχείο παραμένει ανοιχτό μεταξύ αναζητήσεων κατ” όνομα και κατά θύρα.

Σύνοψη

```
setservent STAYOPEN
setservent 0                # close file between lookups (default)
setservent 1                # keep file open across lookups
```

Τι επιστρέφεται

The return value is whatever the underlying C `setservent(3)` returns

- στα περισσότερα συστήματα δεν είναι κάτι χρήσιμο και τα σενάρια δεν την ελέγχουν. Αντιμετωπίστε την `setservent` ως καλούμενη για την παρενέργειά της στον επαναλήπτη, όχι για κάποια τιμή.

Καθολική κατάσταση που επηρεάζει

- Ο δρομέας βάσης δεδομένων υπηρεσιών ανά διεργασία, μοιραζόμενος με τις `getservent`, `getservbyname`, `getservbyport` και `endservent`. Και οι τέσσερις διαβάζουν και τροποποιούν την ίδια κατάσταση· η επανάληψη δεν είναι ασφαλής για νήματα ούτε επανεισαγωγίσιμη.
- Η κατάσταση ανοιχτό/κλειστό του αρχείου βάσης δεδομένων υπηρεσιών. Με `STAYOPEN` αληθές το αρχείο παραμένει ανοιχτό μέχρι να κληθεί η `endservent`· με `STAYOPEN` ψευδές (η προεπιλογή) κάθε επόμενη αναζήτηση κατ” όνομα ή κατά θύρα ανοίγει και κλείνει εκ νέου το αρχείο.

Το όρισμα STAYOPEN

Δώστε αληθή τιμή για να κρατήσετε το αρχείο ανοιχτό σε μια σειρά μεμονωμένων αναζητήσεων `getservbyname` ή `getservbyport`. Αυτό είναι βελτιστοποίηση για ομαδική επίλυση, όχι σημασιολογική αλλαγή: τα αποτελέσματα είναι πανομοιότυπα και στις δύο περιπτώσεις.

```
setservent 1;
for my $name (@service_names) {
    my @svc = getservbyname $name, "tcp";
    # ... process @svc ...
}
endservent;
```

Δώστε ψευδή τιμή (ή `0`) όταν θέλετε κάθε αναζήτηση να ξανανοίγει το αρχείο. Αυτή είναι η προεπιλεγμένη συμπεριφορά και αυτή που θέλουν τα περισσότερα προγράμματα.

Παραδείγματα

Διασχίστε κάθε εγγραφή στη βάση δεδομένων υπηρεσιών και εκτυπώστε ζεύγη ονόματος/θύρας:

```
setservent 0;
while (my ($name, undef, $port, $proto) = getservent) {
    print "$name $port/$proto\n";
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
}
endservent;
```

Επανατυλίξτε στα μέσα της επανάληψης για να ξεκινήσετε ξανά:

```
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
my @second_pass = getservent;       # first entry again
endservent;
```

Ομαδική επίλυση με το αρχείο κρατημένο ανοιχτό:

```
my @wanted = qw(http https ssh smtp);
setservent 1;
my %port = map { $_ => scalar getservbyname($_, "tcp") } @wanted;
endservent;
```

Οριακές περιπτώσεις

- **Το STAYOPEN απαιτείται στο επίπεδο της Perl.** Η `setservent` αναλύεται με προτεραιότητα τελεστή λίστας και διαβάζει ένα όρισμα· μια σκέτη `setservent;` χωρίς όρισμα είναι σφάλμα μεταγλώττισης. Δώστε `0` όταν θέλετε την προεπιλεγμένη συμπεριφορά «κλείσιμο μετά από κάθε αναζήτηση».
- **Εξάρτηση από πλατφόρμα.** Η βάση δεδομένων υπηρεσιών είναι ό,τι χρησιμοποιεί ο επιλυτής συστήματος - τυπικά το `/etc/services`, αλλά NIS, LDAP και άλλα backend υπηρεσιών ονομάτων μπορούν να παρέχουν εγγραφές. Η `setservent` επανατυλίγει όποια πηγή επιλέξει η βιβλιοθήκη C· το επίπεδο της Perl δεν έχει λόγο σε αυτό.
- **Πλατφόρμες εκτός Unix.** Σε συστήματα χωρίς `setservent(3)` η κλήση αποτυγχάνει με κράξιμο `The setservent function is unimplemented`. Δείτε το `perlport` για τον πίνακα διαθεσιμότητας.
- **Νήματα.** Η κατάσταση του επαναλήπτη είναι ανά διεργασία, όχι ανά νήμα· δύο νήματα που διασχίζουν τη βάση δεδομένων υπηρεσιών ταυτόχρονα θα μπλεχτούν και κανένα δεν θα δει όλες τις εγγραφές. Δείτε την κοινή σημείωση στο `perlfunc` για την ασφάλεια νημάτων της οικογένειας `getpwnam`.
- **Ζευγάρωμα με την `endservent`.** Όταν το `STAYOPEN` ήταν αληθές, καλέστε την `endservent` για να απελευθερώσετε τον περιγραφέα αρχείου· αλλιώς το handle διαρρέει για όλη τη ζωή της διεργασίας.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `getservent` - διασχίζει τη βάση δεδομένων υπηρεσιών μία εγγραφή τη φορά· αυτός είναι ο επαναλήπτης που επανατυλίγει η `setservent`
- `endservent` - κλείνει το αρχείο βάσης δεδομένων υπηρεσιών που άνοιξε η `setservent 1`
- `getservbyname` - αναζητά μία υπηρεσία κατ” όνομα· τιμά την κατάσταση ανοιχτό/κλειστό που ορίζεται εδώ
- `getservbyport` - αναζητά μία υπηρεσία κατά αριθμό θύρας· ίδια σημασιολογία ανοιχτό/κλειστό
- `setprotoent` - ίδιο μοτίβο για τη βάση δεδομένων πρωτοκόλλων όταν επανατυλίγετε και τις δύο μαζί
- `Net::servent` - αντικειμενοστρεφές περιτύλιγμα πάνω από την οικογένεια `getserv*` για όταν οι θεσιακές λίστες επιστροφής γίνονται δυσχείριστες

Υποδοχές (sockets)

setsockopt

Θέτει σε ανοιχτή υποδοχή μια επιλογή σε επίπεδο πυρήνα.

Η `setsockopt` είναι το περιτύλιγμα Perl γύρω από την κλήση συστήματος POSIX `setsockopt(2)`. Γράφει την επιλογή που κατονομάζει το `OPTNAME` στο `LEVEL` πρωτοκόλλου, στο ήδη ανοιγμένο filehandle `SOCKET`, χρησιμοποιώντας την τιμή πακεταρισμένη στο `OPTVAL`. Είναι το κουμπί για όλα όσα ο πυρήνας σας αφήνει να ρυθμίσετε σε μια υποδοχή - επαναχρήση διεύθυνσης, keepalives, χρονικά όρια, αλγόριθμος του Nagle, συμμετοχή σε multicast, μεγέθη ενταμιευτών αποστολής και λήψης, συμπεριφορά του `SO_LINGER` στο `close`, και ούτω καθεξής. Τα συμβολικά ονόματα για το `LEVEL` και το `OPTNAME` προέρχονται από το άρθρωμα `Socket`.

Σύνοψη

```
setsockopt SOCKET, LEVEL, OPTNAME, OPTVAL
```

Τι επιστρέφεται

Αληθής τιμή σε επιτυχία, `undef` σε αποτυχία (με την `$!` τεθειμένη στο υποκείμενο errno: `EBADF`, `ENOPROTOOPT`, `EINVAL`, `ENOTSOCK`, κ.λπ.). Πάντα ελέγχετε την τιμή επιστροφής - μια σιωπηρά αγνοημένη επιλογή σημαίνει ότι το πρόγραμμα τρέχει με προεπιλογές που διαφέρουν από αυτό που διαβάζει ο κώδικας:

```
setsockopt($sock, SOL_SOCKET, SO_REUSEADDR, pack("l", 1))
    or die "setsockopt SO_REUSEADDR: $!";
```

Καθολική κατάσταση που επηρεάζει

Καμία άμεσα. Η `setsockopt` θέτει την `$!` σε αποτυχία· αυτή είναι η μόνη καθολική του διερμηνέα που εμπλέκεται. Η επίδραση είναι στην κατάσταση της υποδοχής από την πλευρά του πυρήνα, όχι σε καμία μεταβλητή σε επίπεδο Perl.

OPTVAL - πακεταρισμένα bytes ή σκέτος ακέραιος

Οι περισσότερες επιλογές υποδοχών δέχονται σημαία C `int`. Η `setsockopt` δέχεται και τις δύο μορφές για ευκολία:

- Μια **πακεταρισμένη συμβολοσειρά bytes** της οποίας η διάταξη ταιριάζει με τον τύπο C της επιλογής, τυπικά δομημένη με `pack`: `pack("l", 1)` για λογική τιμή, `pack("ll", $on_secs, $on_usecs)` για `struct timeval`, `pack("ii", $on, $linger)` για `struct linger`.
- Έναν **απλό ακέραιο**, που είναι συντομογραφία για το `pack("i", OPTVAL)`. Βολικό για `SO_REUSEADDR`, `TCP_NODELAY`, και άλλες λογικές σημαίες:

```
setsockopt($sock, IPPROTO_TCP, TCP_NODELAY, 1);
```

Επιλογές των οποίων η τιμή είναι struct (`SO_LINGER`, `SO_RCVTIMEO`, `IP_ADD_MEMBERSHIP`) πρέπει να περαστούν ως ρητά πακεταρισμένη συμβολοσειρά μέσω `pack`. Σκέτος ακέραιος εκεί παράγει `EINVAL` ή, χειρότερα, γράφει σιωπηρά μνήμη λάθος μεγέθους που ο πυρήνας παρερμηνεύει.

Παραδείγματα

Απενεργοποιήστε τον αλγόριθμο του Nagle σε υποδοχή TCP ώστε μικρές εγγραφές να φεύγουν αμέσως - χρήσιμο για πρωτόκολλα ευαίσθητα σε καθυστέρηση (διαδραστικά κελύφη, κίνηση παιχνιδιών, RPC):

```
use Socket qw(IPPROTO_TCP TCP_NODELAY);

setsockopt($sock, IPPROTO_TCP, TCP_NODELAY, 1)
    or die "TCP_NODELAY: $!";
```

Επιτρέψτε σε διακομιστή να επανασυνδέσει θύρα που εξακολουθεί να είναι σε `TIME_WAIT` από προηγούμενη διεργασία - η μοναδικά πιο κοινή κλήση `setsockopt` στην πράξη, και πρέπει να γίνει **πριν** την `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: $!";
```

Θέστε χρονικό όριο λήψης ώστε μια μπλοκαριστική `recv` ή ανάγνωση από την υποδοχή να επιστρέψει με `EAGAIN` / `EWOULDBLOCK` μετά από 5 δευτερόλεπτα αντί να κρεμάει για πάντα. Η τιμή της επιλογής είναι πακεταρισμένο `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: $!";
```

Ενεργοποιήστε τα probes keepalive του TCP ώστε ημι-ανοιχτές συνδέσεις (κατέρρευσε ο ομότιμος, τραβήχτηκε το καλώδιο) να γίνονται τελικά αντιληπτές και η υποδοχή να σφάλλει αντί να μπλοκάρει επ” αόριστον:

```
use Socket qw(SOL_SOCKET SO_KEEPALIVE);

setsockopt($sock, SOL_SOCKET, SO_KEEPALIVE, 1)
    or die "SO_KEEPALIVE: $!";
```

Κάντε την `close` να μπλοκάρει για έως 10 δευτερόλεπτα ενώ ο πυρήνας εκκενώνει μη απεσταλμένα δεδομένα, και μετά να απορρίπτει και να κάνει reset. Το `SO_LINGER` παίρνει struct δύο ακεραίων: τη σημαία ενεργοποίησης και το χρονικό όριο σε δευτερόλεπτα:

```
use Socket qw(SOL_SOCKET SO_LINGER);

setsockopt($sock, SOL_SOCKET, SO_LINGER, pack("ii", 1, 10))
    or die "SO_LINGER: $!";
```

Μεγαλώστε τον ενταμιευτή λήψης του πυρήνα για υποδοχή μαζικής μεταφοράς. Ο πυρήνας τυπικά διπλασιάζει την τιμή που ζητάτε και την περιορίζει στο `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: $!";
```

Οριακές περιπτώσεις

- **Το `SOCKET` πρέπει να είναι ήδη ανοιχτό.** Η `setsockopt` δεν δημιουργεί την υποδοχή - καλέστε πρώτα `socket`. Κλεισμένο ή ποτέ ανοιγμένο filehandle αποτυγχάνει με `EBADF`.
- **Η σειρά έχει σημασία.** Η `SO_REUSEADDR` πρέπει να τεθεί *πριν* την `bind`. Η `SO_KEEPALIVE` μπορεί να μπει πριν ή μετά την `connect`, αλλά για υποδοχές διακομιστή θέστε την στο αποδεκτό θυγατρικό filehandle που επιστρέφει η `accept`, όχι σε αυτό που ακούει - οι επιλογές δεν διαδίδονται πάντα.
- **Η συντομογραφία ακεραίου είναι `pack("i", N)`**, όχι `pack("l", N)`. Σε πλατφόρμες όπου το `int` και το `long` διαφέρουν σε μέγεθος, το να περάσετε σκέτο ακέραιο και το να περάσετε `pack("l", N)` γράφουν διαφορετικό αριθμό bytes. Και τα δύο τυχαίνει να λειτουργούν για τις κοινές λογικές επιλογές μεγέθους `int` σε LP64 Linux, αλλά η συντομογραφία είναι η φορητή προεπιλογή.
- **Επιλογές με τιμή τύπου struct χρειάζονται ρητό `pack`.** Οι `SO_LINGER`,

SO_RCVTIMEO, SO_SNDTIMEO, IP_ADD_MEMBERSHIP, IP_MREQ, και παρόμοιες παίρνουν structs σταθερής διάταξης. Περάστε πακεταρισμένη συμβολοσειρά σωστού μήκους ή ο πυρήνας απορρίπτει την κλήση με EINVAL.

- **Το LEVEL επιλέγει τον χώρο ονομάτων της επιλογής.** SOL_SOCKET για γενικές επιλογές, IPPROTO_TCP για ειδικές του TCP (TCP_NODELAY, TCP_KEEPIDLE), IPPROTO_IP για IPv4 (IP_TTL, IP_ADD_MEMBERSHIP), IPPROTO_IPV6 για IPv6. Αναντιστοιχία LEVEL και OPTNAME δίνει ENOPROTOOPT.
- **Επιλογές μόνο για ανάγνωση ή μη υποστηριζόμενες** αποτυγχάνουν με ENOPROTOOPT. Αυτό ενεργοποιείται επίσης όταν ο πυρήνας απλώς δεν γνωρίζει την επιλογή - συνηθισμένο όταν αντιγράφεται κώδικας μεταξύ λειτουργικών συστημάτων.
- **Προνομιακές επιλογές.** Λίγες επιλογές (SO_PRIORITY πάνω από κάποιες τιμές, IP_TRANSPARENT, SO_BINDTODEVICE) απαιτούν CAP_NET_ADMIN σε Linux· μια κλήση χωρίς προνόμια αποτυγχάνει με EPERM.
- **Η ανάγνωση πίσω γίνεται με την getsockopt, όχι με την setsockopt.** Η setsockopt δεν έχει μορφή ερώτησης· χρησιμοποιήστε την getsockopt για να επιβεβαιώσετε τι πραγματικά αποθήκευσε ο πυρήνας (μπορεί να στρογγυλοποιήσει, να περικόψει ή να διπλασιάσει την τιμή).

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- getsockopt - ο αντίστοιχος ερώτησης· διαβάστε πίσω μια επιλογή για να δείτε τι πράγματι εφάρμοσε ο πυρήνας
- socket - δημιουργεί το filehandle υποδοχής που ρυθμίζει η setsockopt· πρέπει να κληθεί πρώτη
- bind - θέστε το SO_REUSEADDR με setsockopt πριν καλέσετε bind σε υποδοχή διακομιστή
- accept - επιλογές που τίθενται στον listener δεν διαδίδονται όλες· εφαρμόστε ξανά τις επιλογές ανά σύνδεση στο αποδεκτό handle
- pack - δομεί τη συμβολοσειρά bytes OPTVAL για επιλογές με τιμή τύπου struct όπως οι SO_LINGER και SO_RCVTIMEO
- Socket - πηγή των SOL_SOCKET, IPPROTO_TCP, SO_REUSEADDR, TCP_NODELAY, και των υπολοίπων συμβολικών σταθερών

Πίνακες

shift

Αφαιρεί και επιστρέφει το πρώτο στοιχείο ενός πίνακα.

Η shift μεταλλάσσει το όρισμά της: αφαιρεί το στοιχείο στη θέση 0, μετατοπίζει κάθε υπόλοιπο στοιχείο κάτω κατά μία θέση, και επιστρέφει την τιμή που αφαιρέθηκε. Ο

πίνακας συρρικνώνεται κατά ένα. Αν ο πίνακας είναι κενός, δεν αφαιρείται τίποτα και επιστρέφεται `undef`.

Σύνοψη

```
shift @arr
shift @$aref
shift                   # @_ inside a sub, @ARGV outside
```

Τι επιστρέφεται

Το βαθμωτό που βρισκόταν στη θέση `0`, ή `undef` αν ο πίνακας ήταν κενός. Επειδή το `undef` είναι επίσης νόμιμο στοιχείο πίνακα, μια σκέτη επιστροφή `undef` δεν σημαίνει από μόνη της ότι ο πίνακας ήταν κενός - χρησιμοποιήστε `scalar @arr` ή προηγούμενο έλεγχο `defined` όταν η διάκριση έχει σημασία:

```
my @arr  = (undef, 'two');
my $item = shift @arr;      # undef - but @arr was not empty
```

Προεπιλεγμένος πίνακας: @ARGV έναντι @_

Όταν καλείται χωρίς όρισμα, η `shift` επιλέγει τον πίνακά της από το περιβάλλον συγκείμενο:

- Μέσα σε κατονομασμένη ή ανώνυμη υπορουτίνα - **`@_`**. Αυτός είναι ο ιδιωματικός τρόπος για να καταναλώνετε ορίσματα κατά θέση ένα-ένα.
- Εκτός οποιασδήποτε sub (συμπεριλαμβανομένης της εμβέλειας αρχείου ενός σεναρίου, και σε μπλοκ `eval STRING`, `BEGIN`, `INIT`, `CHECK`, `UNITCHECK`, `END`)
  - **`@ARGV`**. This is how top-of-script argument consumption is usually written.

Η επιλογή γίνεται κατά τη μεταγλώττιση βάσει της λεξιλογικής θέσης, όχι κατά τον χρόνο εκτέλεσης. Μια `shift` μέσα σε sub στοχεύει πάντα το `@_` ακόμα κι αν η sub καλείται από εμβέλεια αρχείου.

Παραδείγματα

Κατανάλωση ενός ορίσματος κατά θέση μέσα σε μέθοδο. Αυτή είναι η κανονική πρώτη γραμμή σχεδόν κάθε αντικειμενοστραφούς μεθόδου της Perl:

```
sub greet {
    my $self = shift;
    my $name = shift;
    return "hello, $name, from $self";
}
```

Η ίδια μέθοδος γραμμένη με τομή `@_` που αναλύεται ως λίστα. Και οι δύο μορφές είναι ιδιωματικές· η `shift` υπερισχύει όταν θέλετε να αποσπάσετε τον επικαλούμενο πριν αποφασίσετε τι θα κάνετε με το υπόλοιπο `@_`:

```
sub greet {
    my ($self, $name) = @_;
    return "hello, $name, from $self";
}
```

Κατανάλωση του @ARGV στην κορυφή ενός σεναρίου, ένα όρισμα τη φορά:

```
my $mode = shift // 'default';       # first CLI arg, or 'default'
my $path = shift // die "need a path\n";
# @ARGV now holds whatever is left
```

Μια ουρά FIFO: push στην ουρά, shift από την κεφαλή:

```
my @queue;
push @queue, 'a', 'b', 'c';
while (defined(my $job = shift @queue)) {
    handle($job);
}
```

Αποαναφορά αναφοράς πίνακα επιτόπου - η shift λειτουργεί στο @$aref ακριβώς όπως σε κατονομασμένο πίνακα:

```
my $aref = [10, 20, 30];
my $head = shift @$aref;             # 10, $aref now [20, 30]
```

Οριακές περιπτώσεις

- **Κενός πίνακας**: επιστρέφει undef και αφήνει τον πίνακα ανέγγιχτο. Η shift σε κενό πίνακα δεν είναι σφάλμα.
- **Η πολυπλοκότητα είναι O(n)**: κάθε υπόλοιπο στοιχείο μετακινείται κάτω κατά μία θέση. Για μεγάλους πίνακες που χρησιμοποιούνται ως ουρές αυτό συσσωρεύεται. Όταν δεν απαιτείται διάταξη FIFO, η pop είναι O(1)· όταν απαιτείται, σκεφτείτε δομή δεδομένων τύπου deque ή ζεύγος στοιβών αντί να κάνετε shift σε πίνακα ενός εκατομμυρίου στοιχείων μέσα σε βρόχο.
- **Η shift χωρίς όρισμα είναι ευαίσθητη στο περιβάλλον βάσει λεξιλογικής θέσης**: μέσα σε sub στοχεύει το @_, έξω στοχεύει το @ARGV. Η μετακίνηση μιας γραμμής κώδικα μεταξύ αυτών των εμβελειών αλλάζει σιωπηρά ποιον πίνακα μεταλλάσσει.
- **Tied πίνακες**: η shift καλεί τη μέθοδο SHIFT στο tied αντικείμενο αν είναι ορισμένη· διαφορετικά καταφεύγει σε FETCH στο στοιχείο 0, μετά DELETE, μετά σε ακολουθία ζευγών FETCH/STORE για να μετατοπίσει τα υπόλοιπα στοιχεία κάτω, και μετά STORESIZE. Ένας προσαρμοσμένος χειριστής SHIFT σχεδόν πάντα αξίζει να γραφεί για επιδόσεις.
- **Αναφορές πινάκων**: η shift @$aref και η shift @{ $expr } λειτουργούν και οι δύο. Η shift $aref (χωρίς το sigil @) ήταν πειραματικό χαρακτηριστικό που προστέθηκε στην Perl 5.14 και αφαιρέθηκε στην 5.24 - είναι συντακτικό σφάλμα στην έκδοση Perl που στοχεύει η pperl.

- **Η `shift` είναι καταναλωτής-lvalue, όχι lvalue**: δεν μπορείτε να αναθέσετε στο αποτέλεσμα της `shift` για να τροποποιήσετε τον πίνακα. Χρησιμοποιήστε `splice` ή απευθείας ανάθεση με δείκτη για αυτό.
- **Τιμή επιστροφής σε περιβάλλον λίστας**: η `shift` επιστρέφει πάντα ένα μονό βαθμωτό. Η ανάθεση του αποτελέσματός της σε λίστα - `my ($x) = shift @arr` - λειτουργεί μόνο επειδή το μονό βαθμωτό γεμίζει την πρώτη θέση λίστας· οι υπόλοιπες θέσεις είναι `undef`. Χρησιμοποιήστε `splice` για να αφαιρέσετε πολλά στοιχεία ταυτόχρονα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `pop` - αφαιρεί και επιστρέφει το **τελευταίο** στοιχείο· O(1), η σωστή επιλογή όταν δεν χρειάζεται διάταξη FIFO
- `push` - προσαρτά στην ουρά· ζευγαρωμένη με την `shift` σχηματίζει ουρά FIFO
- `unshift` - προσθέτει στην κεφαλή· η αντίστροφη της `shift`, και επίσης O(n) για τον ίδιο λόγο
- `splice` - αφαιρεί, εισάγει, ή αντικαθιστά οποιαδήποτε συνεχόμενη ακολουθία στοιχείων· το γενικό εργαλείο όταν δεν αρκούν οι `shift` / `pop` / `unshift` / `push`
- `@ARGV` - ο προεπιλεγμένος πίνακας τον οποίο καταναλώνει η `shift` σε εμβέλεια αρχείου
- `@_` - ο προεπιλεγμένος πίνακας τον οποίο καταναλώνει η `shift` μέσα σε sub· κρατά τη λίστα ορισμάτων του καλούντος

SysV IPC

shmctl

Έλεγχος ή ερώτηση τμήματος κοινόχρηστης μνήμης System V.

Η `shmctl` είναι η πρωτογενής λειτουργία διαχείρισης για τμήματα κοινόχρηστης μνήμης που έχουν δημιουργηθεί με την `shmget`. Εκτελεί μία από μια μικρή σειρά λειτουργιών που ονομάζονται μέσω του `CMD` - ανάγνωση των μεταδεδομένων του τμήματος, αλλαγή των δικαιωμάτων του, ή σήμανση για αφαίρεση. Το `ID` είναι το αναγνωριστικό τμήματος που επιστρέφει η `shmget`. Το `ARG` είναι βαθμωτός ενταμιευτής που είτε παρέχει εισερχόμενη δομή `shmid_ds` (για `IPC_SET`) είτε λαμβάνει την επιστρεφόμενη (για `IPC_STAT`)· για `IPC_RMID` αγνοείται.

Σύνοψη

```
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: $!";
```

Τι επιστρέφεται

Η τιμή επιστροφής ακολουθεί τη σύμβαση της `ioctl`, όχι το συνηθισμένο μοτίβο Unix 0-σε-επιτυχία:

- `undef` σε σφάλμα, με το `$!` ρυθμισμένο στο errno.
- Η συμβολοσειρά `"0 but true"` όταν η υποκείμενη κλήση επέστρεψε μηδέν. Αυτή συγκρίνεται αριθμητικά ίση με `0` αλλά είναι αληθής λογικά, οπότε το `shmctl(...) or die` εξακολουθεί να λειτουργεί.
- Οποιαδήποτε άλλη μη μηδενική τιμή διαφορετικά.

Ο πρακτικός κανόνας: ελέγξτε για ψεύδος με `shmctl(...) or die "...: $!"` και ποτέ μη συγκρίνετε την τιμή επιστροφής με κυριολεκτικό `0` μέσω `==`.

Η μορφή `IPC_STAT` γράφει πακεταρισμένη δομή `shmid_ds` στο `ARG`. Αποπακετάρετέ την με το βοηθητικό του `IPC::SysV` ή με χειροκίνητο πρότυπο `unpack`· η διάταξη της δομής εξαρτάται από την πλατφόρμα οπότε η εξάρτηση από σταθερό πρότυπο δεν είναι φορητή.

Καθολική κατάσταση που επηρεάζει

- Το `$!` τίθεται σε αποτυχία στο errno από την υποκείμενη κλήση `shmctl(2)`. Τυπικές τιμές: `EINVAL` (κακό `ID` ή `CMD`), `EACCES` (ανεπαρκή δικαιώματα για τη λειτουργία), `EPERM` (μη ιδιοκτήτης που επιχειρεί `IPC_SET` ή `IPC_RMID`), `EFAULT` (κακός ενταμιευτής `ARG` για `IPC_STAT`/`IPC_SET`).

Παραδείγματα

Ανάγνωση μεταδεδομένων τμήματος. Το `IPC::SysV` παρέχει τον αποπακετάρο `shmid_ds`· χωρίς αυτό, η δομή είναι αδιαφανής συμβολοσειρά bytes:

```
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
```

Αλλαγή των δικαιωμάτων σε υπάρχον τμήμα. Το μοτίβο είναι ανάγνωση-τροποποίηση-εγγραφή στην πακεταρισμένη δομή:

```
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 $!;
```

Σήμανση τμήματος για αφαίρεση. Το τμήμα καταστρέφεται μόλις αποσυνδεθεί η τελευταία διεργασία από αυτό· το `IPC_RMID` μόνο το σημαίνει:

```
use IPC::SysV qw(IPC_RMID);

shmctl($id, IPC_RMID, 0) or die "rmid failed: $!";
```

Φύλαξη βάσει της επιστροφής `"0 but true"`. Επειδή η περίπτωση μηδενός είναι συμβολοσειρά που είναι λογικά αληθής, ο ιδιωματικός έλεγχος είναι η μορφή `or die`, όχι αριθμητική σύγκριση:

```
my $rv = shmctl($id, IPC_STAT, $buf);
defined $rv                          or die "error: $!";
# do NOT write: $rv == 0 and ...   # misses the "0 but true" case
```

Οριακές περιπτώσεις

- **Σταθερές που λείπουν**: τα `IPC_STAT`, `IPC_SET`, `IPC_RMID` δεν είναι ενσωματωμένα. Χωρίς `use IPC::SysV` είναι κλήσεις συνάρτησης bareword και ο κώδικας είτε αποτυγχάνει στη μεταγλώττιση υπό `use strict` είτε σιωπηρά περνά τον λάθος ακέραιο. Πάντα `use IPC::SysV qw(...)`.
- **Λειτουργίες μόνο για τον ιδιοκτήτη**: τα `IPC_SET` και `IPC_RMID` απαιτούν το effective uid να ταιριάζει με τον δημιουργό ή τον ιδιοκτήτη του τμήματος, ή η διεργασία να έχει `CAP_SYS_ADMIN`. Διαφορετικά το `$!` γίνεται `EPERM`.
- **Σημασιολογία `IPC_RMID`**: η σήμανση για αφαίρεση δεν αποσυνδέει τον καλούντα και δεν ακυρώνει τις υπάρχουσες συνδέσεις. Το τμήμα παραμένει χρησιμοποιήσιμο μέχρι κάθε διεργασία να καλέσει `shmdt` σε αυτό· μόνο τότε καταστρέφεται πραγματικά. Μια διεργασία που συνδέεται μετά το `IPC_RMID` λαμβάνει `EIDRM` ή `EINVAL` ανάλογα με τον χρονισμό.
- **Η διάταξη της δομής δεν είναι φορητή**: τα πεδία και τα offsets του `shmid_ds` διαφέρουν μεταξύ πυρήνων και εκδόσεων libc. Μην γράφετε χειροκίνητα πρότυπα `unpack` για σενάρια που πρέπει να τρέξουν σε πολλές πλατφόρμες· χρησιμοποιήστε `IPC::SysV::ShmId` ή παρόμοιο περιτύλιγμα.
- **Διαστασιολόγηση ενταμιευτή για `IPC_STAT`**: το `ARG` πρέπει να είναι βαθμωτό τουλάχιστον `sizeof(shmid_ds)` bytes στην έξοδο. Η Perl το μεγαλώνει όπως χρειάζεται· η προ-διαστασιολόγηση με `$buf = "\0" x 256` είναι συνηθισμένη αλλά όχι απαραίτητη.
- **Μη διαμορφωμένοι πυρήνες**: σε Linux που έχει χτιστεί χωρίς `CONFIG_SYSVIPC`, η κλήση αποτυγχάνει με `ENOSYS`. Container images που εξαιρούν το SysV IPC εμφανίζουν την ίδια συμπεριφορά.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `shmget` - δημιουργία ή αναζήτηση του τμήματος του οποίου το ID χειρίζεται η `shmctl`
- `shmread` - αντιγραφή bytes από συνδεδεμένο τμήμα· χρησιμοποιεί το ID που επιστρέφει η `shmget`, όχι αποτέλεσμα της `shmctl`
- `shmwrite` - αντιγραφή bytes σε συνδεδεμένο τμήμα
- `msgctl` - ίδιο μοτίβο ελέγχου για ουρές μηνυμάτων System V· ίδια σύμβαση επιστροφής `"0 but true"`

- `semctl` - ίδιο μοτίβο ελέγχου για σύνολα σηματοφόρων System V· πλουσιότερο σύνολο CMD αλλά πανομοιότυποι κανόνες τιμής επιστροφής
- `IPC::SysV` - παρέχει τις σταθερές `IPC_STAT` / `IPC_SET` / `IPC_RMID` και τους πακετάρους `shmid_ds`

SysV IPC

shmget

Δημιουργεί ή αναζητά τμήμα κοινόχρηστης μνήμης System V και επιστρέφει το αναγνωριστικό του.

Η `shmget` είναι το βήμα εκχώρησης για κοινόχρηστη μνήμη SysV. Δοθέντος αριθμητικού `KEY` (είτε `IPC_PRIVATE` για φρέσκο ανώνυμο τμήμα είτε ενός γνωστού αριθμού που συμφωνείται με άλλες διεργασίες), του αιτούμενου `SIZE` σε bytes, και μάσκας bits `FLAGS` (σημαίες δημιουργίας OR-αρισμένες με 9-bit mode), επιστρέφει ακέραιο αναγνωριστικό που έχει εκχωρηθεί από τον πυρήνα. Αυτό το αναγνωριστικό - το `shmid` - είναι ό,τι χρησιμοποιεί κάθε επόμενη κλήση (`shmctl`, `shmread`, `shmwrite`) για να αναφερθεί στο τμήμα. Το τμήμα παραμένει στον πυρήνα μέχρι να αφαιρεθεί ρητά με `shmctl($shmid, IPC_RMID, 0)` ή να επανεκκινήσει το σύστημα· το κλείσιμο της δημιουργούσας διεργασίας **δεν** το απελευθερώνει.

Σύνοψη

```
shmget KEY, SIZE, FLAGS
```

Τι επιστρέφεται

Ακέραιο αναγνωριστικό κοινόχρηστης μνήμης (`shmid`) σε επιτυχία, κατάλληλο για πέρασμα στις `shmctl`, `shmread`, και `shmwrite`. Σε αποτυχία επιστρέφει ψευδή τιμή και θέτει το `$!` στο υποκείμενο `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: $!";
```

Το επιστρεφόμενο αναγνωριστικό είναι αδιαφανής μικρός ακέραιος. Μην κάνετε αριθμητική σε αυτό, μην το συγκρίνετε με `0` για αλήθεια - το αναγνωριστικό `0` είναι νόμιμη τιμή σε ορισμένα συστήματα. Πάντα ελέγχετε την επιστροφή έναντι της τεκμηριωμένης φρουράς αποτυχίας (δείτε *Διαφορές από το upstream*).

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται σε αποτυχία στο υποκείμενο `errno`: `EACCES` (δικαιώματα), `EEXIST` (τμήμα για αυτό το κλειδί υπάρχει ήδη και δόθηκαν και τα δύο `IPC_CREAT` και `IPC_EXCL`), `EINVAL` (`SIZE` κάτω από `SHMMIN` ή πάνω από `SHMMAX`, ή το `SIZE` διαφέρει από το υπάρχον τμήμα κατά την αναζήτησή του), `ENFILE` / `ENOSPC` (πίνακας τμημάτων πυρήνα γεμάτος), `ENOENT` (κανένα τμήμα για αυτό το κλειδί και δεν δόθηκε `IPC_CREAT`), `ENOMEM` (ανεπαρκής μνήμη).

Η συνάρτηση είναι κατά τα άλλα καθαρή ως προς την κατάσταση που φαίνεται στην Perl: καμία ειδική μεταβλητή πέραν του `$!` δεν αγγίζεται, το `$_` δεν διαβάζεται.

Παραδείγματα

Δημιουργία φρέσκου ιδιωτικού τμήματος - το συνηθισμένο μοτίβο όταν γονέας και παιδιά μοιράζονται μνήμη μέσω `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
```

Σύνδεση σε προϋπάρχον τμήμα μέσω του γνωστού του κλειδιού. Το `SIZE` και τα mode bits αγνοούνται σε αυτή τη μορφή· περάστε `0` και για τα δύο:

```
my $key   = 0xDEADBEEF;
my $shmid = shmget($key, 0, 0)
    or die "shmget: $!";
```

Δημιουργία-ή-άνοιγμα με `IPC_CREAT`. Αν το κλειδί υπάρχει ήδη επιστρέφεται το υπάρχον `shmid`· διαφορετικά δημιουργείται νέο τμήμα μεγέθους `SIZE` bytes με το δοθέν 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: $!";
```

Αποτυχία αν υπάρχει ήδη τμήμα, με χρήση `IPC_EXCL`. Αυτό είναι το ασφαλές μοτίβο για μια διεργασία που θέλει να είναι ο μοναδικός δημιουργός:

```
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): $!";
}
```

Πλήρης κύκλος δημιουργίας-χρήσης-καταστροφής με εγγυημένο καθαρισμό μέσω `END`:

```
use IPC::SysV qw(IPC_PRIVATE IPC_CREAT IPC_RMID S_IRUSR S_IWUSR);

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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: $!";
```

Οριακές περιπτώσεις

- **Το `use IPC::SysV` είναι ουσιαστικά υποχρεωτικό.** Οι σταθερές σημαιών (`IPC_PRIVATE`, `IPC_CREAT`, `IPC_EXCL`, `IPC_RMID`) και τα mode bits (`S_IRUSR`, `S_IWUSR`, `S_IRGRP`, ...) δεν είναι ενσωματωμένα. Χωρίς την εισαγωγή καταλήγετε να περνάτε σκέτες barewords που μετατρέπονται σε `0` και κάνουν σιωπηρά το λάθος πράγμα.
- **Το κλειδί `0` δεν είναι `IPC_PRIVATE`.** Το `IPC_PRIVATE` ορίζεται ως `0` στο Linux, αλλά το πέρασμα κυριολεκτικού `0` χωρίς `IPC_CREAT` ζητά από τον πυρήνα το τμήμα στο κλειδί `0`, που είναι η φρουρά `IPC_PRIVATE` και αποδίδει `EINVAL`. Εισάγετε τη σταθερά βάσει ονόματος και χρησιμοποιήστε την.
- **Το `SIZE` στρογγυλοποιείται προς τα πάνω σε όριο σελίδας** από τον πυρήνα· το επιστρεφόμενο τμήμα μπορεί να είναι μεγαλύτερο από αυτό που ζητήθηκε. Ζητήστε από τον πυρήνα το ενεργό μέγεθος με `shmctl($shmid, IPC_STAT, $ds)` και αποπακετάρετε τη δομή `shmid_ds` αν χρειάζεστε τον πραγματικό αριθμό.
- **Το τμήμα παραμένει πέραν της εξόδου της διεργασίας.** Ένας δημιουργός που έχει καταρρεύσει ή σκοτωθεί αφήνει το τμήμα στον πυρήνα. Χρησιμοποιήστε `ipcs -m` για να καταγράψετε ορφανά τμήματα και `ipcrm -m <shmid>` για να τα αφαιρέσετε. Εγκαταστήστε μπλοκ `END` ή χειριστή σήματος για να καλέσετε `shmctl IPC_RMID` στον τερματισμό.
- **Το `IPC_RMID` είναι αναβληθέν, όχι άμεσο.** Σημαίνει το τμήμα για καταστροφή· η πραγματική αφαίρεση συμβαίνει μόλις πέσει στο μηδέν ο μετρητής συνδέσεων. Ασφαλές να καλεστεί αμέσως μετά την `shmget` αν γνωρίζετε ότι κανείς άλλος δεν θα συνδεθεί.
- **Όρια πυρήνα.** Το `SHMMNI` περιορίζει τον συνολικό αριθμό τμημάτων στο σύστημα (`/proc/sys/kernel/shmmni`). Το `SHMMAX` περιορίζει το μέγεθος ενός τμήματος (`/proc/sys/kernel/shmmax`). Σε κοινόχρηστο υπολογιστή είναι εύκολο να τα φτάσετε· αντιμετωπίστε τα `ENOSPC` και `EINVAL`-από-μεγάλο-μέγεθος ως ζητήματα διαμόρφωσης, όχι ως bugs.
- **Το SysV IPC δεν είναι διαθέσιμο σε κάθε πλατφόρμα.** Η upstream Perl χτισμένη σε σύστημα χωρίς `HAS_SHM` πεθαίνει με `"System V IPC is not implemented on this machine"`. Η pperl στοχεύει μόνο το Linux, όπου το SysV IPC είναι πάντα παρόν, οπότε η κλήση αποστέλλεται πάντα.

Διαφορές από το upstream

- Επιστρέφει `-1` αντί για `undef` σε αποτυχία. Και οι δύο τιμές είναι ψευδείς υπό τους κανόνες αλήθειας της Perl, οπότε το ιδιωματικό `shmget(...) or die "$! "` λειτουργεί αμετάβλητο, αλλά κώδικας που διακρίνει το «όχι αναγνωριστικό» μέσω `defined $shmid` θα διαβάσει το `-1` ως ορισμένο. Προτιμήστε τον έλεγχο για αλήθεια, ή συγκρίνετε ρητά έναντι του `-1` όταν θέλετε να διακλαδώσετε βάσει της φρουράς αποτυχίας. Το `-1` είναι η ωμή τιμή επιστροφής της `shmget(2)`, που είναι η τεκμηριωμένη φρουρά σφάλματος για αυτή την κλήση συστήματος.

Δείτε επίσης

- `shmctl` - λειτουργίες ελέγχου στο τμήμα που επέστρεψε η `shmget`· χρησιμοποιήστε `IPC_RMID` για να το διαγράψετε, `IPC_STAT` για να εξετάσετε το μέγεθος και τα δικαιώματα
- `shmread` - διαβάζει `SIZE` bytes από το τμήμα σε βαθμωτό της Perl
- `shmwrite` - γράφει βαθμωτό της Perl στο τμήμα σε ένα offset
- `msgget` - το ισοδύναμο βήμα εκχώρησης για ουρές μηνυμάτων SysV όταν θέλετε ουρά, όχι επίπεδο ενταμιευτή
- `semget` - το ισοδύναμο για σηματοφόρους SysV όταν ο στόχος είναι συγχρονισμός παρά κοινόχρηστα δεδομένα
- `IPC::SysV` - πηγή των σταθερών σημαιών `IPC_*` και `S_*` που απαιτεί η `shmget`· δεν υπάρχει τοπική σελίδα αναφοράς, δείτε το upstream POD

SysV IPC

shmread

Αντιγραφή bytes από τμήμα κοινόχρηστης μνήμης System V σε βαθμωτό της Perl.

Η `shmread` συνδέεται στο τμήμα κοινόχρηστης μνήμης που αναγνωρίζεται από το `ID`, διαβάζει `SIZE` bytes ξεκινώντας από το byte offset `POS`, τα γράφει στο `VAR`, και αποσυνδέεται. Στο `VAR` ανατίθεται ως σύνολο - ό,τι κρατούσε πριν αντικαθίσταται από ακριβώς `SIZE` bytes δεδομένων. Η μεταβλητή σημειώνεται ως **tainted** υπό `-T`, επειδή τα περιεχόμενά της προέρχονται από έξω από το πρόγραμμα.

Σύνοψη

```
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: $!";
```

Τι επιστρέφεται

Αληθή σε επιτυχία, ψευδή σε αποτυχία με το `$!` ρυθμισμένο. Σε επιτυχία το VAR έχει επικαλυφθεί με ακριβώς SIZE bytes· καμία σύντομη ανάγνωση - είτε όλο το εύρος αντιγράφεται είτε η κλήση αποτυγχάνει. Σε αποτυχία τα περιεχόμενα του VAR είναι απροσδιόριστα· μη βασίζεστε σε αυτά.

Πάντα ελέγχετε την τιμή επιστροφής. Κακό ID, εκτός εύρους POS/ SIZE, ή τμήμα στο οποίο η διεργασία δεν έχει δικαίωμα σύνδεσης, όλα αναδύονται εδώ ως ψευδής επιστροφή:

```
shmread($id, my $buf, $pos, $size)
    or die "shmread(id=$id, pos=$pos, size=$size): $!";
```

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται σε αποτυχία (οποιοδήποτε από τα υποκείμενα σφάλματα shmat/memcpy/shmdt αναδύεται εδώ).
- Σημαία taint στο VAR - τίθεται υπό -T ανεξάρτητα από το πώς δηλώθηκε το VAR, επειδή τα bytes προέρχονται από έξω από το πρόγραμμα.

Παραδείγματα

Ανάγνωση εγγραφής σταθερού μεγέθους από τμήμα:

```
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
```

Ανάγνωση σε ένα offset. Τα POS και SIZE είναι σε bytes, όχι σε στοιχεία - το τμήμα δεν έχει έννοια εγγραφών:

```
shmread($id, my $chunk, 256, 64)
    or die "shmread: $!";                             # bytes 256..
 ↪319
```

Αποπακετάρισμα δυαδικής εγγραφής απευθείας από την κοινόχρηστη μνήμη:

```
shmread($id, my $raw, 0, 16) or die "shmread: $!";
my ($magic, $len, $flags) = unpack "a4 N N", $raw;
```

Ανάγνωση tainted δεδομένων υπό -T. Η ανάθεση εγγυάται να παράγει tainted βαθμωτό ακόμα κι αν το VAR κρατούσε ήδη untainted τιμή:

```
# perl -T
shmread($id, my $buf, 0, $n) or die "shmread: $!";
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
# $buf is tainted; untaint via a regex match before using
# it in a sensitive context.
```

Οριακές περιπτώσεις

- **Στο VAR γίνεται ανάθεση, όχι προσάρτηση.** Οποιαδήποτε προηγούμενη τιμή απορρίπτεται. Χρησιμοποιήστε φρέσκια λεξιλογική μεταβλητή αν χρειάζεστε και τα παλιά και τα νέα δεδομένα.
- **Το SIZE είναι ακριβές, όχι μέγιστο.** Η shmread δεν κάνει σύντομες αναγνώσεις· το να ζητήσετε περισσότερα bytes από αυτά που κρατά το τμήμα από το POS και μετά αποτυγχάνει αντί να επιστρέψει ό,τι χωρά.
- **Τα POS και SIZE πρέπει να βρίσκονται εντός του τμήματος** που δηλώθηκε κατά την κλήση shmget. POS + SIZE πέρα από το τέλος του τμήματος αποτυγχάνει με EINVAL.
- **Μηδενικό SIZE** παράγει κενή συμβολοσειρά στο VAR και πετυχαίνει, υποθέτοντας ότι το ίδιο το POS είναι έγκυρο.
- **Δικαιώματα.** Η καλούσα διεργασία πρέπει να έχει δικαίωμα ανάγνωσης στο τμήμα, όπως υπαγορεύουν τα mode bits που πέρασαν στην shmget (και, όπου ισχύει, από τον ιδιοκτήτη του τμήματος και τα τρέχοντα uid/gid).
- **Το VAR δεν χρειάζεται προ-διαστασιολόγηση.** Σε αντίθεση με την sysread, δεν χρειάζεται να γεμίσετε με substr ή να προεκχωρήσετε το VAR - η shmread αναθέτει το αποτέλεσμα απευθείας.
- **Taint.** Υπό -T, το VAR είναι tainted σε επιτυχία ανεξάρτητα από αυτό που κρατούσε πριν. Ξεπλύνετέ το μέσω αντιστοίχισης με capture-group ($data =~ /\A(.*)\z/s and $data = $1) μόνο αφού έχετε επικυρώσει ότι τα bytes είναι ασφαλή για την ευαίσθητη λειτουργία που πρόκειται να εκτελέσετε.
- **Δεν έχει κάθε πλατφόρμα SysV IPC.** Σε συστήματα χωρίς shmat(2) η κλήση αποτυγχάνει κατά τον χρόνο εκτέλεσης (τυπικά ENOSYS)· φορητός κώδικας θα πρέπει να φυλαχθεί με eval ή έλεγχο δυνατότητας.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- shmwrite - η αντίστροφη: αντιγραφή συμβολοσειράς σε τμήμα σε δοθέν offset
- shmget - απόκτηση ή δημιουργία του ID τμήματος στο οποίο λειτουργεί η shmread
- shmctl - ερώτηση μεταδεδομένων τμήματος (το IPC_STAT γεμίζει ένα shmid_ds που μπορείτε να χρησιμοποιήσετε για να ελέγξετε τα όρια του POS + SIZE) ή αφαίρεση του τμήματος όταν τελειώσετε
- sysread - το αντίστοιχο σε επίπεδο περιγραφέα αρχείου· ίδιο σχήμα «bytes σε βαθμωτό», διαφορετική πηγή

- unpack - μετατροπή των ωμών bytes που επιστρέφονται στο VAR σε πεδία με τύπο
- IPC::SysV - το άρθρωμα που παρέχει τα IPC_PRIVATE, S_IRUSR, και τις άλλες σταθερές που θα περάσετε στην shmget

SysV IPC

shmwrite

Αντιγράφει bytes σε ένα τμήμα κοινής μνήμης System V.

Η shmwrite προσαρτάται στο τμήμα κοινής μνήμης που προσδιορίζεται από το ID, αντιγράφει έως SIZE bytes από το STRING ξεκινώντας από τη μετατόπιση byte POS μέσα στο τμήμα, και αποπροσαρτάται. Αν το STRING είναι μακρύτερο από SIZE, εγγράφονται μόνο τα πρώτα SIZE bytes· αν είναι κοντύτερο, το υπόλοιπο του παραθύρου SIZE bytes γεμίζει με μηδενικά bytes NUL. Το ID τμήματος είναι η τιμή που επέστρεψε προηγούμενη κλήση shmget (ή αποκτήθηκε από άλλη διεργασία μέσω κάποιου εκτός-ζώνης καναλιού).

Σύνοψη

```
shmwrite ID, STRING, POS, SIZE
```

Τι επιστρέφεται

Αληθές σε επιτυχία, ψευδές σε σφάλμα, με την $! ορισμένη στον λόγο (τυπικά EINVAL, EACCES ή EIDRM). Σε αντίθεση με τις περισσότερες ενσωματωμένες I/O, η τιμή επιστροφής είναι απλή λογική τιμή - δεν υπάρχει παραλλαγή με μέτρηση bytes, επειδή το πλήθος είναι πάντα ακριβώς SIZE όταν η κλήση πετυχαίνει.

```
shmwrite($id, $payload, 0, 4096)
    or die "shmwrite($id) failed: $!";
```

Καθολική κατάσταση που επηρεάζει

- Η $! ορίζεται σε αποτυχία στο errno των υποκείμενων shmat / memcpy / shmdt.
- Δεν διαβάζονται ούτε εγγράφονται άλλες ειδικές μεταβλητές. Σε αντίθεση με την shmread, η shmwrite δεν μολύνει με taint τίποτα - είναι η *πηγή* των bytes, όχι αποδέκτης.

Παραδείγματα

Δημοσιεύστε εγγραφή σταθερού μεγέθους στην αρχή ενός τμήματος:

```
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: $!";
```

Ένα STRING κοντύτερο από το SIZE συμπληρώνεται με NUL ώστε να γεμίσει το παράθυρο - χρήσιμο για να καθαρίζετε ξεπερασμένα τελικά bytes από προηγούμενο εγγραφέα:

```
shmwrite($id, "short", 0, 64);   # 5 bytes of "short", then 59 NULs
```

Ένα STRING μακρύτερο από το SIZE περικόπτεται· η ουρά απορρίπτεται σιωπηρά, οπότε διαστασιολογήστε το παράθυρό σας στη μεγαλύτερη εγγραφή που σκοπεύετε ποτέ να δημοσιεύσετε:

```
shmwrite($id, "x" x 10_000, 0, 128);   # only the first 128 x's land
```

Η εγγραφή σε μη μηδενική μετατόπιση επιτρέπει σε πολλαπλές εγγραφές σταθερού μεγέθους να μοιράζονται ένα τμήμα. Ζευγαρώστε την με αντίστοιχη shmread στην πλευρά του αναγνώστη:

```
my $slot = 3;
my $len  = 64;
shmwrite($id, $record, $slot * $len, $len)
    or die "shmwrite slot $slot: $!";
```

Ελέγξτε την τιμή επιστροφής - ένα τμήμα που έχει σημανθεί προς διαγραφή με IPC_RMID εξακολουθεί να δέχεται εγγραφές από διεργασίες ήδη προσαρτημένες, αλλά αποτυγχάνει σε νέες κλήσεις shmwrite με EIDRM:

```
unless (shmwrite($id, $msg, 0, $SIZE)) {
    warn "segment $id gone: $!";
    # reacquire via shmget() and retry
}
```

Οριακές περιπτώσεις

- **Ο πυρήνας πρέπει να έχει χτιστεί με SysV IPC.** Σε συστήματα χωρίς κοινή μνήμη SysV η shmwrite κράζει με shmwrite not implemented. Αυτή είναι ιδιότητα κατά τη μεταγλώττιση του πυρήνα / libc, όχι συνθήκη χρόνου εκτέλεσης που μπορείτε να ανιχνεύσετε.
- **Το SIZE ίσο με 0** γίνεται αποδεκτό από τον πυρήνα και δεν εγγράφει τίποτα· η shmwrite επιστρέφει αληθές. Χρησιμοποιήστε το μόνο όταν εννοείτε πραγματικά «no-op», όχι ως σήμα σφάλματος.
- **Το POS + SIZE πρέπει να χωράει μέσα στο τμήμα.** Η υπέρβαση του ορίου τμήματος αποτυγχάνει με EINVAL και αφήνει το τμήμα ανέγγιχτο - η αντιγραφή δεν είναι μερική.
- **Τα δυαδικά ωφέλιμα φορτία είναι εντάξει.** Το STRING αντιμετωπίζεται ως συμβολοσειρά bytes· τα ενσωματωμένα NUL εγγράφονται κυριολεκτικά. Αν η συμβολοσειρά φέρει τη σημαία UTF-8, τα ακατέργαστα εσωτερικά bytes είναι αυτά που φτάνουν στο τμήμα - χρησιμοποιήστε pack ή Encode::encode πρώτα αν χρειάζεστε συγκεκριμένη κωδικοποίηση στο καλώδιο.
- **Καμία κλείδωση.** Ο πυρήνας εγγυάται ότι κάθε κλήση shmwrite είναι ατομική ως προς τις shmat / shmdt, αλλά δύο ταυτόχρονοι εγγραφείς μπορούν ακόμη να

μπλέξουν εύρη bytes μέσα στο τμήμα. Συντονιστείτε με `semop` ή κλείδωμα χώρου χρήστη αν πολλαπλές διεργασίες γράφουν στην ίδια περιοχή.

- **Τα δικαιώματα** προέρχονται από το mode του τμήματος τη στιγμή της `shmget`. Προσάρτηση μόνο για ανάγνωση και κλήση `shmwrite` αποτυγχάνει με `EACCES`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `shmread` - αντίστροφη πράξη· αντιγράφει bytes *έξω* από τμήμα σε βαθμωτό Perl
- `shmget` - λαμβάνει το `ID` στο οποίο γράφει η `shmwrite`
- `shmctl` - επιθεωρεί χαρακτηριστικά τμήματος, αλλάζει δικαιώματα ή σημαίνει τμήμα προς διαγραφή
- `semop` - η συνήθης συνοδευτική πρωτογενής για σειριοποίηση πρόσβασης σε κοινό τμήμα
- `pack` - δομεί τη συμβολοσειρά bytes σταθερής διάταξης που δίνετε στην `shmwrite` όταν το τμήμα κρατά δομημένες εγγραφές
- `IPC::SysV` - σταθερές (`IPC_PRIVATE`, `S_IRUSR`, `IPC_RMID`) που χρησιμοποιούνται για τη δόμηση των ορισμάτων στις `shmget` και `shmctl`

Υποδοχές (sockets)

shutdown

Τερματίζει τη μία κατεύθυνση μιας σύνδεσης υποδοχής, ή και τις δύο.

Η `shutdown` λέει στον πυρήνα να σταματήσει μια υποδοχή να κάνει περαιτέρω αναγνώσεις, περαιτέρω εγγραφές, ή και τα δύο - χωρίς να κλείσει τον περιγραφέα αρχείου. Σε αντίθεση με την `close`, η επίδραση είναι άμεση και διαδίδεται σε κάθε forked αντίγραφο του περιγραφέα, όχι μόνο σε αυτό της τρέχουσας διεργασίας. Το `HOW` έχει την ίδια σημασία με το όρισμα της κλήσης συστήματος `shutdown(2)`.

Σύνοψη

```
shutdown SOCKET, HOW
```

Τι επιστρέφεται

1 σε επιτυχία. Σε αποτυχία, `undef` αν το `SOCKET` δεν είναι έγκυρο filehandle, ή `0` με την `$!` ορισμένη για κάθε άλλο σφάλμα (η υποδοχή δεν είναι συνδεδεμένη, έχει ήδη τερματιστεί σε εκείνη την κατεύθυνση, κ.ο.κ.).

Δύο διακριτές επιστροφές αποτυχίας, οπότε μερικές φορές χρειάζεται τριμερής έλεγχος:

```
my $r = shutdown($sock, 1);
if    (!defined $r) { die "bad filehandle" }
elsif (!$r)         { die "shutdown failed: $!" }
```

Το όρισμα HOW

Το HOW είναι μικρός ακέραιος που ταιριάζει με τις σταθερές POSIX SHUT_*:

- 0 - σταματά την ανάγνωση. Ο ομότιμος μπορεί ακόμη να στέλνει, αλλά τα δεδομένα θα απορρίπτονται από τον πυρήνα και η read / sysread σε αυτό το άκρο θα επιστρέφει τέλος αρχείου.
- 1 - σταματά την εγγραφή. Η έξοδος που είναι σε ουρά εκκενώνεται, και κατόπιν ο ομότιμος βλέπει τέλος αρχείου στο άκρο ανάγνωσής του. Η print / syswrite σε αυτό το άκρο θα αποτύχει και θα εγείρει SIGPIPE.
- 2 - σταματά και τις δύο κατευθύνσεις. Ισοδύναμο με κλήση της shutdown με 0 και κατόπιν με 1.

Τα συμβολικά ονόματα SHUT_RD, SHUT_WR, SHUT_RDWR βρίσκονται στο Socket αν προτιμάτε ευανάγνωστο κώδικα:

```
use Socket qw(SHUT_WR);
shutdown $sock, SHUT_WR;
```

shutdown έναντι close

Η close απορρίπτει τον περιγραφέα αρχείου μόνο στην τρέχουσα διεργασία. Άλλες διεργασίες που κληρονόμησαν τον περιγραφέα μέσω fork κρατούν τα αντίγραφά τους ανοιχτά, και ο ομότιμος δεν βλέπει τέλος αρχείου μέχρι να κλείσει το τελευταίο αντίγραφο.

Η shutdown ενεργεί στην υποκείμενη υποδοχή, όχι στον περιγραφέα. Μετά από shutdown($sock, 1), κάθε forked αντίγραφο βλέπει το μισό εγγραφής κλειστό και ο ομότιμος διαβάζει τέλος αρχείου αμέσως - αν και ο ίδιος ο περιγραφέας παραμένει ανοιχτός και μπορεί ακόμη να διαβαστεί.

Αυτός είναι ο συνήθης λόγος να καταφύγει κανείς στη shutdown: γονέας και παιδί μοιράζονται μια υποδοχή, το παιδί τελείωσε την αποστολή και ο γονέας πρέπει να συνεχίσει να διαβάζει απαντήσεις. Η close στο παιδί θα ήταν αόρατη για τον ομότιμο· η shutdown($sock, 1) όχι.

Μισό κλείσιμο για πρωτόκολλα αίτησης/απάντησης

Ένας πελάτης που στέλνει πλήρη αίτηση και κατόπιν χρειάζεται ο διακομιστής να γνωρίζει ότι δεν υπάρχει άλλη είσοδος χρησιμοποιεί shutdown με 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";
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
shutdown $sock, 1;          # signal end-of-request to the server

while (my $line = <$sock>) {
    print $line;
}
close $sock;
```

Χωρίς τη shutdown, οι διακομιστές που διαβάζουν μέχρι το τέλος εισόδου θα μπλόκαραν περιμένοντας περισσότερα δεδομένα αίτησης.

Παραδείγματα

Διακοπή ανάγνωσης αλλά συνέχιση εγγραφής - ο ομότιμος μπορεί να συνεχίζει να στέλνει, απλώς δεν θα τα κοιτάμε:

```
shutdown $sock, 0;
```

Διακοπή εγγραφής, συνέχιση ανάγνωσης - τυπικό μισό κλείσιμο μετά την ολοκλήρωση αίτησης:

```
shutdown $sock, 1;
```

Πλήρης τερματισμός, και κατόπιν κλείσιμο του περιγραφέα:

```
shutdown $sock, 2;
close $sock;
```

Χρήση των σταθερών Socket για ευαναγνωσιμότητα:

```
use Socket qw(SHUT_RD SHUT_WR SHUT_RDWR);
shutdown $sock, SHUT_WR;
```

Έλεγχος του συγκεκριμένου σφάλματος όταν ένα shutdown αποτυγχάνει σε ήδη κλειστή κατεύθυνση:

```
unless (shutdown $sock, 1) {
    warn "shutdown write failed: $!";
}
```

Οριακές περιπτώσεις

- **Η εγγραφή μετά από `shutdown($sock, 1)`** αποτυγχάνει και εγείρει SIGPIPE. Η προεπιλεγμένη ενέργεια τερματίζει τη διεργασία· εγκαταστήστε `$SIG{PIPE} = 'IGNORE'` (ή χειριστή) για να τη μετατρέψετε σε κανονική αποτυχία εγγραφής που μπορείτε να ελέγξετε.
- **Η ανάγνωση μετά από `shutdown($sock, 0)`** επιστρέφει τέλος αρχείου αμέσως, όχι σφάλμα. Όσα δεδομένα είναι ήδη σε ουρά στον ενταμιευτή λήψης του πυρήνα απορρίπτονται.
- **Δεν είναι υποδοχή**: η κλήση shutdown σε κανονικό filehandle θέτει την `$!` σε ENOTSOCK και επιστρέφει `0`.

- **Μη συνδεδεμένη**: η `shutdown` σε υποδοχή που δεν συνδέθηκε ποτέ (φρέσκο αποτέλεσμα της `socket`, ή υποδοχή που ακούει) θέτει την `$!` σε `ENOTCONN` και επιστρέφει `0`.
- **Διπλό shutdown**: ο τερματισμός του ίδιου μισού δύο φορές τυπικά επιστρέφει `0` με `ENOTCONN`. Ο φορητός κώδικας αγνοεί τη δεύτερη αποτυχία αντί να την αντιμετωπίζει ως μοιραία.
- **Κλειστό filehandle**: επιστρέφει `undef`· υπό `use warnings` εκπέμπεται προειδοποίηση `shutdown() on closed socket`.
- **Ο περιγραφέας δεν κλείνει**: η `shutdown` απενεργοποιεί τις κατευθύνσεις που ζητήσατε αλλά αφήνει τον περιγραφέα αρχείου ανοιχτό. Καλέστε `close` (ή αφήστε το handle να βγει εκτός εμβέλειας) όταν πραγματικά τελειώσετε με αυτό.
- **Κοινόχρηστοι περιγραφείς μέσω `fork`**: αυτή είναι η περίπτωση για την οποία σχεδιάστηκε η `shutdown`. Όλες οι διεργασίες που κρατούν τον περιγραφέα βλέπουν το shutdown να εφαρμόζεται άμεσα· δεν έχουν ξεχωριστές όψεις.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `close` - απελευθερώνει τον περιγραφέα σε αυτή τη διεργασία· δεν επηρεάζει forked αντίγραφα και δεν σηματοδοτεί τέλος αρχείου στον ομότιμο μέχρι να κλείσει το τελευταίο αντίγραφο
- `socket` - δημιουργεί την υποδοχή στην οποία λειτουργεί η `shutdown`
- `connect` - εγκαθιδρύει τη σύνδεση· η `shutdown` σε μη συνδεδεμένη υποδοχή αποτυγχάνει με `ENOTCONN`
- `accept` - παράγει τη συνδεδεμένη υποδοχή στην πλευρά του διακομιστή που τυπικά μισοκλείνει με `shutdown`
- `Socket` - εξάγει τις `SHUT_RD`, `SHUT_WR`, `SHUT_RDWR` και τις υπόλοιπες σταθερές επιπέδου socket
- `$!` - συμβολοσειρά σφάλματος συστήματος που ελέγχεται μετά από επιστροφή `0`

Αριθμητικές συναρτήσεις

sin

Επιστρέφει το ημίτονο αριθμού δοθέντος σε ακτίνια.

Η `sin` είναι το τυπικό τριγωνομετρικό ημίτονο: παίρνει γωνία μετρημένη σε **ακτίνια** και επιστρέφει τον λόγο της απέναντι πλευράς προς την υποτείνουσα του αντίστοιχου ορθογωνίου τριγώνου. Αν παραλειφθεί το EXPR, η `sin` λειτουργεί στο `$_`. Το αποτέλεσμα είναι πάντα αριθμός κινητής υποδιαστολής στο κλειστό διάστημα `[-1, 1]`.

Σύνοψη

```
sin EXPR
sin
```

Τι επιστρέφεται

Αριθμός κινητής υποδιαστολής διπλής ακρίβειας στο `[-1, 1]`. Η Perl προωθεί την κλήση στη ρουτίνα `sin(3)` C της πλατφόρμας, οπότε η ακρίβεια και η συμπεριφορά στρογγυλοποίησης ταιριάζουν με τη libm σας. Οι ειδικές τιμές ακολουθούν το IEEE 754: η `sin(0)` είναι ακριβώς `0`, οι `sin("inf")` και `sin("nan")` είναι και οι δύο `NaN`.

Ακτίνια, όχι μοίρες

Το μακράν πιο συνηθισμένο λάθος με την `sin` είναι το να της δίνετε μοίρες. Μια πλήρης στροφή είναι `2 * π` ακτίνια, όχι `360`. Μετατρέψτε μία φορά και κρατήστε τη μετατραμμένη τιμή:

```
use POSIX qw(acos);
my $pi = acos(-1);                  # π to full double precision

sub deg2rad { $_[0] * $pi / 180 }

print sin( deg2rad(30) ), "\n";     # 0.5
print sin( 30 ), "\n";              # -0.988031624092862 - wrong if␣
 ↪you meant degrees
```

Το εξάγει τις `deg2rad` και `rad2deg` αν προτιμάτε να μη γράψετε εσείς τη μετατροπή.

Παραδείγματα

Η `sin(π/2)` είναι μαθηματικά ακριβώς 1, αλλά επειδή το π δεν είναι αναπαραστάσιμο σε δυαδική κινητή υποδιαστολή το υπολογιζόμενο αποτέλεσμα υπολείπεται κατά μερικά ulps:

```
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
```

Αντιμετωπίστε κάθε τριγωνομετρικό αποτέλεσμα που «θα έπρεπε να είναι μηδέν» ως κατά προσέγγιση μηδέν. Συγκρίνετε με ανοχή, όχι με ==.

Μετατροπή μοιρών σε ακτίνια ενσωματωμένα, χωρίς εισαγωγή αρθρώματος. Το `3.14159` είναι ακριβές σε έξι ψηφία - εντάξει για γραφικές παραστάσεις, όχι για αριθμητική ανάλυση:

```
for my $deg (0, 30, 45, 60, 90) {
    printf "sin(%2d°) = %+.4f\n",
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
            $deg, sin($deg * 3.14159 / 180);
}
# sin( 0°) = +0.0000
# sin(30°) = +0.5000
# sin(45°) = +0.7071
# sin(60°) = +0.8660
# sin(90°) = +1.0000
```

Για εργασία που χρειάζεται πλήρη διπλή ακρίβεια, αντικαταστήστε το `3.14159` με `acos(-1)` ή `atan2(1, 1) * 4`.

Αριθμητική παράγωγος του `sin` με πεπερασμένες διαφορές. Ο συμμετρικός διαπολυτύπος δύο σημείων `(f(x+h) - f(x-h)) / (2h)` θα έπρεπε να ανακτά το `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);
```

Η επιλογή του $h είναι συμβιβασμός: πολύ μεγάλο και κυριαρχεί το σφάλμα αποκοπής, πολύ μικρό και η αλληλοεξουδετέρωση στην αφαίρεση καταστρέφει το αποτέλεσμα. Το `1e-6` είναι μια λογική προεπιλογή για διπλή ακρίβεια.

Διασχίστε τον μοναδιαίο κύκλο και εκτυπώστε τα `(cos θ, sin θ)` σε οκτώ ισαπέχουσες γωνίες:

```
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);
}
```

Αντίστροφο ημίτονο (τόξο ημιτόνου) μέσω της ταυτότητας από το `perlfunc`:

```
sub asin { atan2( $_[0], sqrt(1 - $_[0] * $_[0]) ) }

print asin(0.5), "\n";              # 0.523598775598299 (= π/6)
```

Οριακές περιπτώσεις

- **Τα μη αριθμητικά ορίσματα εξαναγκάζονται σε `0`.** Η `sin("hello")` επιστρέφει `0` επειδή η συμβολοσειρά μετατρέπεται πρώτα αριθμητικά σε `0`. Υπό `use warnings` αυτό παράγει προειδοποίηση `Argument "hello" isn't numeric in sin`. Επικυρώστε την είσοδο με `looks_like_number` από το `Scalar::Util` αν η πηγή δεν είναι έμπιστη.

- **Τα πολύ μεγάλα ορίσματα χάνουν ακρίβεια**. Η `sin` πρώτα ανάγει το όρισμά της modulo 2π, και για `|x|` της τάξης του `1e16` ο πλησιέστερος αναπαραστάσιμος double διαφέρει από το `x` κατά περισσότερο από π. Το αποτέλεσμα είναι ακόμα αριθμός στο `[-1, 1]`, αλλά είναι ουσιαστικά τυχαίο. Ανάγετε εσείς τη γωνία πριν την κλήση αν συσσωρεύετε φάση σε πολλές επαναλήψεις.
- **Το `undef` εξαναγκάζεται σε `0`** και επιστρέφει `0`, με τη συνήθη προειδοποίηση `uninitialized` υπό `use warnings`.
- **Ειδικές τιμές IEEE**: οι `sin("inf")`, `sin("-inf")`, και `sin("nan")` επιστρέφουν όλες NaN. Συγκρίνετε με `$result != $result` (τη μόνη τιμή που δεν είναι ίση με τον εαυτό της) για να ανιχνεύσετε NaN, ή χρησιμοποιήστε `POSIX::isnan`.
- **Προεπιλεγμένο όρισμα**. Η `sin;` χωρίς όρισμα διαβάζει το `$_`. Μέσα σε βρόχο `while (<>)` αυτό λειτουργεί στην τρέχουσα γραμμή, που σχεδόν πάντα είναι bug.
- **Καμία υποστήριξη μιγαδικών αριθμών** στην ενσωματωμένη. Για μιγαδικά ορίσματα χρησιμοποιήστε το , που υπερφορτώνει την `sin` μέσω υπερφόρτωσης τελεστών:

```
use Math::Complex;
my $z = cplx(1, 2);
print sin($z), "\n";              # 3.16577851321617+1.
↪9596010414216i
```

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `cos` - συνημίτονο γωνίας σε ακτίνια· η συνοδευτική συνάρτηση που χρησιμοποιείται συχνότερα μαζί με την `sin`
- `atan2` - τόξο εφαπτομένης δύο ορισμάτων, το συνηθισμένο δομικό στοιχείο για αντίστροφη τριγωνομετρία (`asin`, `acos`) σε καθαρή Perl
- `sqrt` - τετραγωνική ρίζα· εμφανίζεται στην κλασική ταυτότητα `asin` `asin(x) = atan2(x, sqrt(1 - x*x))`
- - `asin`, `acos`, `atan`, υπερβολικές παραλλαγές, `deg2rad`/`rad2deg`, και βοηθητικά για μέγιστο κύκλο
- - υπερφορτώνει την `sin` (και την υπόλοιπη τριγωνομετρική οικογένεια) για μιγαδικά ορίσματα
- `POSIX` - εκθέτει την `asin` άμεσα, καθώς και τις `isnan`/`isinf` για ανίχνευση των μη πεπερασμένων αποτελεσμάτων που συζητούνται παραπάνω

Διεργασίες

sleep

Παύει τη διεργασία για ένα πλήθος ακεραίων δευτερολέπτων.

Η `sleep` αναστέλλει την εκτέλεση της τρέχουσας διεργασίας για EXPR δευτερόλεπτα, ή - χωρίς όρισμα - μέχρι να την ξυπνήσει κάποιο σήμα. Το όρισμα αποκόπτεται σε ακέραιο· οι καθυστερήσεις κάτω του δευτερολέπτου απαιτούν διαφορετικό εργαλείο (βλ. παρακάτω). Η τιμή επιστροφής είναι ο αριθμός των δευτερολέπτων που κοιμήθηκε πραγματικά, που μπορεί να είναι μικρότερος από αυτόν που ζητήθηκε αν σήμα διέκοψε την κλήση.

Σύνοψη

```
sleep EXPR
sleep
```

Τι επιστρέφεται

Ένας ακέραιος: ο αριθμός των δευτερολέπτων κατά τα οποία ανεστάλη πραγματικά η διεργασία. Σε αδιακόπη κλήση αυτός ισούται με το (αποκομμένο) όρισμα. Αν ένα σήμα που παραδίδεται κατά τη διάρκεια του sleep αποδεσμεύσει την κλήση νωρίτερα, η τιμή επιστροφής είναι ο μικρότερος, πραγματικός χρόνος που πέρασε. Για γυμνή `sleep`, η τιμή είναι ό,τι κοιμήθηκε η υλοποίηση πριν φτάσει σήμα.

```
my $slept = sleep 10;
warn "woken early after $slept s" if $slept < 10;
```

Χειρισμός ορίσματος

- **Θετικός ακέραιος** - η κανονική περίπτωση. Κοιμάται για τόσα δευτερόλεπτα.
- **Μη ακέραιος** (π.χ. `2.9`) - το κλασματικό μέρος **απορρίπτεται**. Η `sleep 2.9` κοιμάται δύο δευτερόλεπτα, όχι σχεδόν τρία. Για κλασματικές καθυστερήσεις χρησιμοποιήστε `Time::HiRes::sleep`, `Time::HiRes::usleep`, ή τη μορφή τεσσάρων ορισμάτων της `select` - βλ. παρακάτω.
- **Αρνητικός ακέραιος** - η `sleep` **δεν** κοιμάται. Εκπέμπει προειδοποίηση, θέτει την `$!` (`errno`) και επιστρέφει μηδέν.
- **Μηδέν** - η `sleep 0` επιτρέπεται, αλλά εκτελεί παρ” όλα αυτά κλήση στη υποκείμενη πρωτογενή λειτουργία sleep της πλατφόρμας με όποιες παρενέργειες συνεπάγεται αυτή. Δεν είναι λοιπόν **απολύτως** ταυτόσημη με το να παραλείψετε την κλήση. Ειδικότερα, η `sleep 0` δεν είναι το ίδιο με γυμνή `sleep` χωρίς όρισμα.
- **Κανένα όρισμα** - η `sleep` χωρίς όρισμα κοιμάται επ” αόριστον, μέχρι να παραδοθεί σήμα. Αυτό διαφέρει από την `sleep 0`.

Διακοπή από σήματα

Η `sleep` επιστρέφει νωρίτερα αν η διεργασία λάβει σήμα του οποίου ο χειριστής τρέχει (ή του οποίου η προεπιλεγμένη ενέργεια δεν τερματίζει τη διεργασία). Το κανονικό μοτίβο

για μια οριοθετημένη αναμονή που μπορεί να ακυρωθεί από αλλού είναι μια `alarm` σε συνδυασμό με χειριστή σήματος:

```
eval {
    local $SIG{ALRM} = sub { die "Alarm!\n" };
    sleep;
};
die $@ unless $@ eq "Alarm!\n";
```

Επειδή η ίδια η `sleep` υλοποιείται συνήθως πάνω από την `alarm` σε συστήματα Unix, γενικά **δεν** μπορείτε να αναμίξετε κλήσεις `alarm` και `sleep` στο ίδιο πρόγραμμα χωρίς εκπλήξεις. Επιλέξτε μία.

Καθυστερήσεις κάτω του δευτερολέπτου

Η `sleep` δέχεται εξ ορισμού μόνο ακεραίους. Για πιο λεπτόκοκκες αναμονές χρησιμοποιήστε ένα από τα:

- `Time::HiRes::sleep($seconds)` - δέχεται κινητής υποδιαστολής αριθμό δευτερολέπτων. Από την Perl 5.8 το `Time::HiRes` διανέμεται με την βασική διανομή.
- `Time::HiRes::usleep($microseconds)` - ακέραια microseconds.
- Η μορφή τεσσάρων ορισμάτων της `select` με τα τρία πρώτα ορίσματα `undef`:

  ```
  select undef, undef, undef, 0.25;   # sleep 250 ms
  ```

- `syscall` στη `setitimer(2)` αν η πλατφόρμα σας την υποστηρίζει και χρειάζεστε στενό έλεγχο πάνω σε χρονομετρητές.

Δείτε επίσης την `POSIX::pause`, που μπλοκάρει τη διεργασία μέχρι να φτάσει οποιοδήποτε σήμα - ουσιαστικά γυμνή `sleep` χωρίς όριο καθορισμένο από την υλοποίηση.

Παραδείγματα

Σταθερή καθυστέρηση:

```
print "working...\n";
sleep 3;
print "done\n";
```

Βρόχος δημοσκόπησης με back-off:

```
until (ready()) {
    sleep 1;
}
```

Ανίχνευση πρόωρου ξυπνήματος:

```
my $want = 30;
my $got  = sleep $want;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
if ($got < $want) {
    warn "interrupted after $got of $want seconds\n";
}
```

Αναμονή για σήμα χωρίς να καίγεται CPU - η γυμνή `sleep` αναστέλλει μέχρι να φτάσει `SIGALRM` (ή οποιοδήποτε άλλο σήμα που πιάνεται):

```
$SIG{USR1} = sub { warn "got USR1\n" };
sleep;                                  # wakes on USR1 delivery
```

Ακύρωση μακράς αναμονής με `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";
```

Οριακές περιπτώσεις

- **Η γυμνή `sleep` δεν κοιμάται πραγματικά «για πάντα».** Η υλοποίηση έχει μέγιστο - στην πράξη, ό,τι δέχεται η υποκείμενη `sleep(3)` της C ή το ισοδύναμο - αλλά για κάθε ρεαλιστικό σκοπό μπλοκάρει μέχρι να φτάσει σήμα. Αντιμετωπίστε την ως αόριστη αναμονή.
- **`sleep 0` ≠ καμία κλήση.** Η κλήση συστήματος `sleep(0)` της πλατφόρμας εκτελείται παρ” όλα αυτά. Στα περισσότερα συστήματα αυτό παραχωρεί την CPU για λίγο· μη βασίζεστε στη συγκεκριμένη συμπεριφορά σε διαφορετικές πλατφόρμες.
- **Η `sleep` έναντι του πραγματικού χρόνου.** Σε παλαιότερα συστήματα η `sleep` μπορούσε να επιστρέψει μέχρι και ένα ολόκληρο δευτερόλεπτο *νωρίτερα* λόγω αδρής μέτρησης δευτερολέπτων. Τα σύγχρονα συστήματα κοιμούνται ολόκληρη την ποσότητα, αλλά μπορούν να υπερβούν: ο χρονοπρογραμματιστής μπορεί να μην ξυπνήσει αμέσως τη διεργασία σας σε ένα απασχολημένο σύστημα. Η `sleep N` εγγυάται *τουλάχιστον* `N` δευτερόλεπτα αναστολής σε σύγχρονους πυρήνες, όχι *ακριβώς* `N`.
- **Τα αρνητικά ορίσματα δεν περικόπτονται.** Είναι σκληρό no-op που θέτει την `$!`. Αν υπολογίζετε το όρισμα, επικυρώστε το πρώτα.
- **Η ανάμιξη με `alarm` είναι εύθραυστη.** Ο χρονομετρητής alarm και ο χρονομετρητής sleep μπορεί να είναι ο ίδιος πόρος. Καθιερώστε μία πειθαρχία (είτε `alarm` + γυμνή `sleep`, είτε σκέτη `sleep N` χωρίς alarms) και μείνετε σε αυτήν μέσα σε ένα πρόγραμμα.
- **Σήματα που απλώς θέτουν σημαία επίσης ξυπνούν την `sleep`.** Οποιοδήποτε σήμα του οποίου η παράδοση κάνει χειριστή να τρέξει επιστρέφει τον έλεγχο στην

Perl· η `sleep` τότε επιστρέφει νωρίτερα ακόμη και αν ο ίδιος ο χειριστής δεν κάνει τίποτα ορατό.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `alarm` - προγραμματίζει `SIGALRM` μετά από δεδομένο αριθμό δευτερολέπτων· ο κλασικός τρόπος να οριοθετηθεί μια `sleep` ή οποιαδήποτε άλλη μπλοκαριστική κλήση
- `select` - στη μορφή τεσσάρων ορισμάτων με μη ορισμένη τριάδα readfds/writefds/exceptfds, παρέχει sleeps κάτω του δευτερολέπτου
- `wait`, `waitpid` - μπλοκάρει σε διεργασίες-παιδιά αντί σε χρόνο
- `Time::HiRes::sleep` - δευτερόλεπτα κινητής υποδιαστολής
- `Time::HiRes::usleep` - ακέραια microseconds
- `POSIX::pause` - μπλοκάρει μέχρι να φτάσει οποιοδήποτε σήμα

Υποδοχές (sockets)

socket

Δημιουργεί filehandle υποδοχής.

Η `socket` ανοίγει μια υποδοχή του είδους που περιγράφουν τα `DOMAIN`, `TYPE`, και `PROTOCOL` και τη συνδέει με το `SOCKET`, το οποίο γίνεται χρησιμοποιήσιμο filehandle της Perl. Είναι το λεπτό περιτύλιγμα της Perl πάνω από την κλήση συστήματος `socket(2)` - τίποτα περισσότερο, τίποτα λιγότερο. Η νέα υποδοχή είναι μη δεσμευμένη και μη συνδεδεμένη· εξακολουθείτε να χρειάζεστε `bind`, `connect`, ή `listen`+`accept` για να κάνετε οτιδήποτε με αυτήν. Εισάγετε τις συμβολικές σταθερές με `use Socket;` πριν την κλήση - η γραφή των ωμών ακεραίων είναι φορητή μόνο κατά τύχη.

Σύνοψη

```
use Socket;
socket SOCKET, DOMAIN, TYPE, PROTOCOL
```

Τι επιστρέφεται

1 σε επιτυχία, `undef` σε αποτυχία (με το `$!` ρυθμισμένο στο `errno` από την υποκείμενη `socket(2)`). Σε επιτυχία το `SOCKET` είναι πλέον filehandle στο οποίο μπορείτε να καλέσετε `bind`, `connect`, `send` / `recv`, ή να διαβάσετε και να γράψετε με τους συνηθισμένους τελεστές I/O μόλις φτάσει σε συνδεδεμένη κατάσταση. Αξίζει να ελέγχετε την τιμή επιστροφής σε κάθε κλήση - μια αποτυχία εδώ σημαίνει ότι δεν έχετε τίποτα χρήσιμο να περάσετε στην επόμενη κλήση socket, η οποία στη συνέχεια θα αποτύχει με πιο συγκεχυμένο τρόπο.

```
socket my $sock, PF_INET, SOCK_STREAM, getprotobyname("tcp")
    or die "socket: $!";
```

Το SOCKET μπορεί να είναι bareword filehandle (SOCK), glob (*SOCK), ή απροσδιόριστο βαθμωτό που θα αυτο-δημιουργηθεί σε αναφορά ανώνυμου filehandle. Η μορφή αυτο-δημιουργίας είναι το σύγχρονο ιδίωμα και η μόνη που περιοριάζεται καθαρά με `my`.

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται στο errno από την `socket(2)` σε αποτυχία.
- `$^F` - το κατώφλι περιγραφέα αρχείου συστήματος. Σε συστήματα με close-on-exec, ο νέος περιγραφέας λαμβάνει σημαία FD_CLOEXEC όταν ο αριθμός του υπερβαίνει το `$^F` (προεπιλογή 2, οπότε οι περιγραφείς 0–2 παραμένουν κληρονομήσιμοι και όλα τα άλλα κλείνουν αυτόματα κατά την `exec`). Ανεβάστε το πριν την `socket` αν θέλετε συγκεκριμένα η νέα υποδοχή να επιβιώσει μιας `exec`.

Παραδείγματα

Υποδοχή πελάτη για σύνδεση TCP. Δημιουργία, σύνδεση, επικοινωνία:

```
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";
```

Υποδοχή διακομιστή TCP που ακούει. Δημιουργία, σήμανση ως επαναχρησιμοποιήσιμη, 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: $!";
```

Υποδοχή UDP - SOCK_DGRAM αντί για SOCK_STREAM, και δεν χρειάζεται `connect` πριν την `send` όταν περνάτε ρητό προορισμό:

```
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: $!";
```

Υποδοχή ροής τομέα Unix για τοπικό IPC. PF_UNIX με διαδρομή συστήματος αρχείων αντί για διεύθυνση IP:

```
use Socket;
socket my $sock, PF_UNIX, SOCK_STREAM, 0 or die "socket: $!";
connect $sock, pack_sockaddr_un("/run/app.sock") or die "connect: $!
 ↪";
```

Αφήνοντας το PROTOCOL στην προεπιλογή 0. Ο πυρήνας επιλέγει το προεπιλεγμένο πρωτόκολλο για το ζεύγος domain/type - για PF_INET + SOCK_STREAM είναι TCP, για PF_INET + SOCK_DGRAM είναι UDP. Η μορφή getprotobyname παραπάνω είναι πιο ρητή αλλά η μορφή 0 είναι ισοδύναμη και αποφεύγει μία αναζήτηση:

```
use Socket;
socket my $sock, PF_INET, SOCK_STREAM, 0 or die "socket: $!";
```

Οριακές περιπτώσεις

- **Το use Socket δεν είναι προαιρετικό.** Χωρίς αυτό, τα PF_INET, SOCK_STREAM, και συγγενή είναι συμβολοσειρές bareword, όχι σταθερές ακέραιοι, και η κλήση αποτυγχάνει στο όριο της κλήσης συστήματος με Invalid argument. Η εισαγωγή είναι αυτό που τα μετατρέπει σε αριθμούς.
- **Το SOCKET αλλοιώνεται σε επιτυχία.** Αν η θυρίδα κρατά ανοιχτό filehandle, κλείνει πρώτα. Δεν υπάρχει προειδοποίηση - η προηγούμενη υποδοχή απλώς εξαφανίζεται σιωπηρά. Χρησιμοποιήστε φρέσκια μεταβλητή my για κάθε κλήση.
- **Η αποτυχία αφήνει το SOCKET αμετάβλητο.** Σε επιστροφή undef το βαθμωτό ούτε ανοίγεται ούτε τροποποιείται· μια προηγούμενη τιμή (συμπεριλαμβανομένης παλαιότερης, ακόμα ανοιχτής υποδοχής) είναι ακόμα εκεί.
- **Όρια περιγραφέων.** Η socket είναι το σημείο όπου εμφανίζεται η εξάντληση πίνακα περιγραφέων EMFILE (ανά διεργασία) και ENFILE (σε επίπεδο συστήματος). Διακομιστές μακράς διάρκειας θα πρέπει να τα αντιμετωπίζουν ως ανακτήσιμα αντί ως μοιραία.
- **EAFNOSUPPORT / EPROTONOSUPPORT / ESOCKTNOSUPPORT.** Ο πυρήνας απέρριψε την τριάδα domain/type/protocol. Συνηθισμένη αιτία: αίτημα για PF_INET6 σε πυρήνα χτισμένο χωρίς IPv6, ή μη έγκυρος αριθμός πρωτοκόλλου που πέρασε από μια παλιωμένη αναζήτηση getprotobyname.
- **Η μη μπλοκαριστική λειτουργία είναι ξεχωριστό βήμα.** Η socket επιστρέφει μπλοκαριστικό περιγραφέα. Χρησιμοποιήστε fcntl με F_SETFL + O_NONBLOCK (ή το αντίστοιχο του IO::Socket) αν χρειάζεστε I/O χωρίς μπλοκάρισμα.
- **Το close-on-exec είναι αυτόματο πέρα από το $^F.** Η Perl θέτει FD_CLOEXEC στον νέο περιγραφέα όταν ο αριθμός fd του είναι μεγαλύτερος από το $^F. Μια υποδοχή επομένως δεν θα επιβιώσει μιας exec εκτός αν χαμηλώσατε αυτό το κατώφλι ή καθαρίσατε τη σημαία με fcntl.
- **Οι αριθμητικές τιμές domain/type δεν είναι φορητές.** Το 2 τυγχάνει να είναι το AF_INET στο Linux· είναι κάτι άλλο αλλού. Πάντα να περνάτε μέσω των σταθερών του αρθρώματος Socket.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `bind` - συνδέει υποδοχή που δημιούργησε η `socket` με τοπική διεύθυνση, απαιτείται πριν την `listen` ή πριν την αποστολή από γνωστή θύρα
- `connect` - εκκινεί εξερχόμενη σύνδεση σε υποδοχή `SOCK_STREAM`, ή ορίζει τον προεπιλεγμένο ομότιμο σε υποδοχή `SOCK_DGRAM`
- `listen` - σημαίνει δεσμευμένη υποδοχή `SOCK_STREAM` ως παθητική ώστε η `accept` να μπορεί να αντλεί συνδέσεις από αυτήν
- `accept` - επιστρέφει την επόμενη εκκρεμή σύνδεση σε υποδοχή που ακούει ως φρέσκο filehandle
- `socketpair` - δημιουργία ζεύγους συνδεδεμένων υποδοχών σε μία κλήση· η συνηθισμένη επιλογή όταν και τα δύο άκρα ζουν στην ίδια διεργασία ή σε γονέα/παιδί
- `Socket` - το άρθρωμα που παρέχει τα `PF_INET`, `SOCK_STREAM`, `inet_aton`, `pack_sockaddr_in` και τις υπόλοιπες σταθερές και packers που περνάτε στην `socket`
- `$^F` - κατώφλι περιγραφέα που ελέγχει το close-on-exec για τη νέα υποδοχή

Υποδοχές (sockets)

socketpair

Δημιουργεί ένα ανώνυμο, συνδεδεμένο ζεύγος υποδοχών που μιλούν μεταξύ τους.

Η `socketpair` ανοίγει δύο φρέσκα filehandles στα `SOCKET1` και `SOCKET2` ήδη συνδεδεμένα μεταξύ τους, οπότε ο,τιδήποτε γράφεται στο ένα βγαίνει από το άλλο. Καμία διεύθυνση, κανένα `bind`, κανένα `listen`, κανένα `accept`. Η συνηθέστερη χρήση είναι η επικοινωνία με παιδική διεργασία δημιουργημένη από `fork` - κάθε πλευρά κρατά ένα handle και κλείνει το άλλο με `close`.

Σύνοψη

```
use Socket;
socketpair(SOCKET1, SOCKET2, DOMAIN, TYPE, PROTOCOL) or die $!;
socketpair(my $s1, my $s2, AF_UNIX, SOCK_STREAM, PF_UNSPEC) or die
 ↪$!;
```

Τι επιστρέφεται

Αληθές σε επιτυχία, ψευδές σε αποτυχία (με ορισμένη την `$!`). Σε επιτυχία και τα δύο ορίσματα filehandle συμπληρώνονται με φρέσκα handles· σε αποτυχία μένουν άθικτα. Ελέγχετε πάντα την τιμή επιστροφής - κάθε συνδυασμός domain/type/protocol εξαρτάται από την πλατφόρμα.

```
socketpair(my $a, my $b, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
    or die "socketpair: $!";
```

Οι σταθερές DOMAIN, TYPE και PROTOCOL προέρχονται από το άρθρωμα Socket. Το AF_UNIX με SOCK_STREAM και PF_UNSPEC είναι η φορητή προεπιλογή.

Καθολική κατάσταση που επηρεάζει

- Η $! ορίζεται σε αποτυχία στο errno του συστήματος.
- $^F - ο μέγιστος περιγραφέας αρχείου συστήματος κάτω από τον οποίο το close-on-exec **δεν** τίθεται. Σε συστήματα με FD_CLOEXEC, οι περιγραφείς πάνω από την $^F αποκτούν αυτόματα close-on-exec· κάτω από αυτήν, επιβιώνουν στο exec. Η προεπιλογή είναι 2 (stdin/stdout/stderr). Αυξήστε την πριν την socketpair αν θέλετε τα νέα handles να κληρονομούνται κατά την exec.

Παραδείγματα

Απλή σωλήνωση γονέα/παιδιού με socketpair ροής. Κάθε πλευρά κλείνει την άκρη που δεν χρειάζεται, ώστε το EOF να διαδίδεται σωστά:

```
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;
```

Αμφίδρομη ροή: και οι δύο διεργασίες διαβάζουν *και* γράφουν στο ίδιο handle. Δεν χρειάζεται shutdown επειδή καμία πλευρά δεν κλείνει μια κατεύθυνση πρόωρα:

```
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";
    }
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    exit;
}
close $b;
print $a "ping\n";
chomp(my $reply = <$a>);            # "echo: ping"
close $a;
waitpid $pid, 0;
```

Εξομοίωση της pipe με socketpair. Η shutdown μετατρέπει μια αμφίδρομη υποδοχή σε μονόδρομο κανάλι:

```
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
```

Ζεύγος datagram - τα μηνύματα διατηρούν τα όρια, οπότε μία recv επιστρέφει ακριβώς μία send:

```
use Socket;
socketpair(my $a, my $b, AF_UNIX, SOCK_DGRAM, PF_UNSPEC) or die $!;
send($a, "one", 0)   or die $!;
send($a, "two", 0)   or die $!;
recv($b, my $m1, 64, 0);            # "one"
recv($b, my $m2, 64, 0);            # "two"
```

Οριακές περιπτώσεις

- **Το AF_UNIX είναι το μόνο φορητό domain.** Στα περισσότερα συστήματα η socketpair λειτουργεί μόνο με AF_UNIX· άλλες οικογένειες επιστρέφουν EAFNOSUPPORT. Χρησιμοποιήστε τη σταθερά, όχι σκληρά κωδικοποιημένο 1.
- **Έξοδος χωρίς ενταμίευση.** Τα handles ροής socketpair ενταμιεύουν όπως κάθε άλλο filehandle. Θέστε την $| στον εγγραφέα (αφού τον επιλέξετε με select) αλλιώς ο αναγνώστης θα μπλοκάρει περιμένοντας έξοδο που βρίσκεται ακόμη στον ενταμιευτή του γονέα. Η autoflush από το IO::Handle είναι η συνήθης μορφή:

  ```
  $parent->autoflush(1);
  ```

- **Κλείστε και τις δύο άκρες αλλιώς το EOF δεν φτάνει ποτέ.** Ένα socketpair ροής παραδίδει EOF σε αναγνώστη μόνο αφού *κάθε* εγγράψιμο αντίγραφο του handle του ομοτίμου κλείσει. Ένα παιδί που ξεχνά να κλείσει με close την άκρη του γονέα θα κρεμάσει τον γονέα στο <$handle> για πάντα.
- **Close-on-exec μέσω της $^F.** Filehandles των οποίων ο αριθμητικός περιγραφέας είναι μεγαλύτερος της $^F αποκτούν FD_CLOEXEC σε πλατφόρμες που το υποστηρίζουν, οπότε εξαφανίζονται κατά την exec. Αυξήστε την $^F πριν την κλήση αν το handle πρέπει να επιβιώσει μιας exec.
- **Μη υλοποιημένη στην πλατφόρμα.** Αν η υποκείμενη κλήση συστήματος socketpair(2) δεν υπάρχει, η perl εγείρει εξαίρεση αντί να επιστρέψει ψευδές.

Παγιδέψτε την με `eval` αν χρειάζεστε υποχώρηση.

- **Datagram έναντι ροής.** Το `SOCK_STREAM` είναι προσανατολισμένο σε bytes (χωρίς όρια μηνυμάτων, χρησιμοποιήστε framing με νέα γραμμή ή προθέματα μήκους). Το `SOCK_DGRAM` διατηρεί τα όρια μηνυμάτων αλλά τα χάνει σιωπηρά αν ο ενταμιευτής λήψης είναι μικρότερος από το μήνυμα.
- **`PROTOCOL`.** Για `AF_UNIX` το πρωτόκολλο αγνοείται· το `PF_UNSPEC` (ή `0`) είναι σύμβαση. Ο ορισμός πρωτοκόλλου που δεν ταιριάζει στον συνδυασμό domain/type επιστρέφει `EPROTONOSUPPORT`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `socket` - δημιουργεί μία υποδοχή όταν μόνο η μία άκρη είναι δική σας (η άλλη έρχεται από το δίκτυο μέσω `accept` ή `connect`)
- `pipe` - μονόδρομο κανάλι bytes· ελαφρύτερη από την `socketpair` αλλά δεν μπορεί να κάνει αμφίδρομη κίνηση σε μία κλήση
- `send` / `recv` - I/O προσανατολισμένη σε μηνύματα, απαιτείται για socketpairs `SOCK_DGRAM`
- `fork` - ο τυπικός σύντροφος: μία `socketpair` πριν το fork δίνει σε γονέα και παιδί έτοιμο αγωγό
- `Socket` - πηγή των `AF_UNIX`, `SOCK_STREAM`, `SOCK_DGRAM`, `PF_UNSPEC` και των άλλων σταθερών που περιμένει η `socketpair`
- `$^F` - κατώφλι πάνω από το οποίο νέοι περιγραφείς αποκτούν close-on-exec· έχει σημασία όταν τα handles πρέπει να επιβιώσουν μιας `exec`

Λίστες

sort

Ταξινομεί μια λίστα, επιστρέφοντας την ταξινομημένη λίστα.

Η `sort` παίρνει μια λίστα και επιστρέφει νέα λίστα με τα στοιχεία αναδιατεταγμένα σε σειρά. Χωρίς συγκριτή χρησιμοποιεί την τυπική σύγκριση συμβολοσειρών. Με ένα `BLOCK` ή ένα `SUBNAME` παρέχετε εσείς τον κανόνα σύγκρισης ως μικρή συνάρτηση που καλείται για κάθε ζεύγος στοιχείων και πρέπει να επιστρέφει αρνητική, μηδενική ή θετική τιμή με τον τρόπο που το κάνουν τα `cmp` και `<=>`.

Η ταξινόμηση είναι σταθερή - στοιχεία που συγκρίνονται ως ίσα διατηρούν την αρχική τους σχετική σειρά. Ο αλγόριθμος είναι mergesort.

Σύνοψη

```
sort LIST
sort BLOCK LIST
sort SUBNAME LIST
```

Τρεις ιδιωματικές μορφές:

```
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
```

Τι επιστρέφεται

Μια **νέα** ταξινομημένη λίστα. Η `sort` **δεν** τροποποιεί την είσοδο επιτόπια. Για να αντικαταστήσετε έναν πίνακα με την ταξινομημένη του εκδοχή, εκχωρείτε πίσω:

```
@a = sort @a;
```

Σε περιβάλλον λίστας η επιστροφή είναι η ταξινομημένη λίστα. Σε βαθμωτό περιβάλλον η τιμή επιστροφής είναι απροσδιόριστη - μη χρησιμοποιείτε `scalar sort`.

Η επιστρεφόμενη λίστα περιέχει **ψευδώνυμα** προς την αρχική λίστα, ακριβώς όπως η μεταβλητή δείκτη ενός `foreach`. Η τροποποίηση στοιχείου του αποτελέσματος μέσα σε μεταγενέστερο `foreach`, `map` ή `grep` επομένως τροποποιεί και το αρχικό στοιχείο. Αυτό σχεδόν ποτέ δεν είναι αυτό που θέλετε· χειριστείτε το αποτέλεσμα ως μόνο-για-ανάγνωση.

Καθολική κατάσταση: $a και $b

Μέσα στο `BLOCK` ή το `SUBNAME` του συγκριτή, τα δύο στοιχεία που συγκρίνονται εκτίθενται ως οι καθολικές πακέτου `$a` και `$b`. Αυτές δεν είναι παράμετροι και δεν είναι λεξιλογικές - είναι πραγματικές μεταβλητές πακέτου που η `sort` τοπικοποιεί για εσάς γύρω από την κλήση.

- Οι `$a` και `$b` ζουν στο **πακέτο που κάλεσε τη `sort`**. Στο `main`, αυτές είναι οι `$main::a` και `$main::b`· στο `Foo`, οι `$Foo::a` και `$Foo::b`.
- **Ποτέ μη δηλώνετε `my $a` ή `my $b`** οπουδήποτε μπορεί να δει το μπλοκ ταξινόμησης. Μια λεξιλογική `$a` ή `$b` επικαλύπτει την καθολική πακέτου· τότε το μπλοκ συγκρίνει δύο άσχετες μεταβλητές και η ταξινόμηση παράγει σιωπηλά σκουπίδια. Υπό `use warnings` δεν λαμβάνετε καμία προειδοποίηση γι" αυτό.
- Αν πραγματικά χρειάζεστε τις `$a` / `$b` ως λεξιλογικές αλλού, αναφερθείτε στις μεταβλητές ταξινόμησης με το πλήρες όνομα μέσα στο μπλοκ:

  ```
  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);
  ```

- Ένας συγκριτής που ορίζεται με prototype `($$)` (ή ισοδύναμο γνώρισμα signature) λαμβάνει το ζεύγος στο `@_` αντί για τα `$a` και `$b`. Αυτό είναι πιο αργό από τη σύμβαση `$a`/`$b`, αλλά αποσυνδέει την υπορουτίνα από το πακέτο του καλούντος.

Και τα δύο στοιχεία περνούν **με αναφορά**· η μεταβολή των `$a` ή `$b` στο μπλοκ μεταβάλλει την αρχική λίστα. Μην το κάνετε.

Παραδείγματα

Προεπιλεγμένη ταξινόμηση συμβολοσειρών:

```
my @sorted = sort qw(banana apple cherry);
# ('apple', 'banana', 'cherry')
```

Αριθμητική αύξουσα - η προεπιλεγμένη `sort @nums` θα ταξινομούσε το `"10"` πριν το `"2"` επειδή η σύγκριση γίνεται ανά συμβολοσειρά:

```
my @nums = sort { $a <=> $b } (10, 2, 33, 4);
# (2, 4, 10, 33)
```

Αριθμητική φθίνουσα - αντιμεταθέστε τα `$a` και `$b`:

```
my @nums = sort { $b <=> $a } (10, 2, 33, 4);
# (33, 10, 4, 2)
```

Ταξινόμηση με πολλαπλά κλειδιά και αποκλεισμό ισοβαθμίας. Το `||` εκτελεί τον δεύτερο συγκριτή μόνο όταν ο πρώτος επιστρέψει `0`:

```
my @people = sort {
    $a->{last}  cmp $b->{last}
 || $a->{first} cmp $b->{first}
} @records;
```

Ταξινομήστε τα κλειδιά ενός hash βάσει της σχετιζόμενης τιμής:

```
my %age = (alice => 30, bob => 25, carol => 40);
my @by_age = sort { $age{$a} <=> $age{$b} } keys %age;
# ('bob', 'alice', 'carol')
```

Schwartzian Transform - όταν το κλειδί ταξινόμησης είναι ακριβό να υπολογιστεί, προυπολογίστε το μία φορά ανά στοιχείο, ταξινομήστε τα ζεύγη `[key, value]`, και κατόπιν αφαιρέστε τα κλειδιά. Τρία στάδια, διαβάστε από κάτω προς τα πάνω:

```
my @sorted =
    map  { $_->[1] }                    # 3. strip key
    sort { $a->[0] <=> $b->[0] }        # 2. sort by key
    map  { [ expensive_key($_), $_ ] }  # 1. attach key
    @input;
```

Ονομασμένη υπορουτίνα συγκριτή. Η υπορουτίνα διαβάζει τα `$a` και `$b` από το πακέτο του καλούντος, οπότε πρέπει να βρίσκεται (ή να αναφέρεται από) εκείνο το πακέτο:

```
sub by_name { $a->{name} cmp $b->{name} }

my @sorted = sort by_name @people;
```

Συγκριτής με prototype - τα ορίσματα φτάνουν στο `@_`, οπότε λειτουργεί από οποιοδήποτε πακέτο:

```
sub numeric ($$) { $_[0] <=> $_[1] }

my @sorted = sort numeric @nums;
```

Οριακές περιπτώσεις

- **Οι $a και $b είναι καθολικές πακέτου, όχι λεξιλογικές.** Ένα αδέσποτο `my $a` στην εμβέλεια σπάει σιωπηλά κάθε ταξινόμηση σε εκείνη την εμβέλεια. Αν πρέπει, χρησιμοποιήστε `our $a` ή `$::a` μέσα στο μπλοκ.
- **Το <=> είναι αριθμητικό, το cmp είναι συμβολοσειράς.** Η χρήση του λάθος είναι το πιο κοινό σφάλμα ταξινόμησης. Η `sort { $a cmp $b } (10, 2, 33)` δίνει `(10, 2, 33)` - λεξικογραφικά, το `"10"` πριν το `"2"`. Χρησιμοποιήστε `<=>` για αριθμούς.
- **Κενή λίστα επιστρέφει κενή λίστα.** Η `sort ()` είναι `()`. Κανένα σφάλμα.
- **Ένα μόνο στοιχείο επιστρέφεται αμετάβλητο.** Δεν γίνεται καμία σύγκριση.
- **Ο συγκριτής πρέπει να είναι συνεπής.** Αν κάποτε λέει `$x < $y` και κάποτε το αντίθετο για το ίδιο ζεύγος, το αποτέλεσμα είναι απροσδιόριστο. Ο συγκριτής πρέπει να επιστρέφει ολική διάταξη: αρνητικό, μηδέν ή θετικό, με σταθερό τρόπο, για κάθε κλήση.
- **Το NaN δηλητηριάζει το <=>.** Η `$a <=> $b` επιστρέφει `undef` αν οποιοσδήποτε από τους τελεστέους είναι `NaN`, την οποία η `sort` στη συνέχεια χειρίζεται ως `0` - ίσα - δίνοντας αυθαίρετη τοποθέτηση. Φιλτράρετε πρώτα:

```
my @clean = sort { $a <=> $b } grep { $_ == $_ } @input;
```

- **Σταθερή ταξινόμηση.** Τα ίσα στοιχεία διατηρούν τη σειρά εισόδου τους. Βασιστείτε σε αυτό για ταξινομήσεις πολλαπλών περασμάτων (ταξινομήστε πρώτα κατά δευτερεύον κλειδί, μετά κατά πρωτεύον). Η `use sort 'stable'` είναι η προεπιλογή και no-op· οι pragmas `use sort '_mergesort'` / `'_qsort'` είναι ιστορικές και δεν έχουν επίδραση στην τρέχουσα Perl.
- **Η sort δεν είναι επιτόπια.** Η `sort @a` δεν αγγίζει το `@a`. Χρησιμοποιήστε `@a = sort @a` για να το αντικαταστήσετε. Η εκχώρηση στη λίστα αποτελέσματος της `sort` ψευδωνυμοποιεί πίσω στο αρχικό, οπότε η `(sort @a)[0] = ...` τροποποιεί το μικρότερο στοιχείο του `@a`.
- **Κανένας έλεγχος βρόχου εκτός του μπλοκ.** Τα `last`, `next`, `redo`, `return` και `goto LABEL` δεν λειτουργούν μέσα σε συγκριτή sort - το μπλοκ δεν είναι σώμα βρόχου.
  - Όταν η εργασία δεν είναι βρόχος
- **Παγίδα αναλυτή: ταξινόμηση της τιμής επιστροφής συνάρτησης.** Η `sort` παίρνει `LIST`, οπότε ένα ακολουθούμενο bareword μπορεί να εκληφθεί ως `SUBNAME`. Για να ταξινομήσετε το αποτέλεσμα της `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
```

Για να χρησιμοποιήσετε την `find_records` ως συγκριτή επί του `@key` αντ” αυτού:

```
my @out = sort find_records @key;                    # comparator
 ↪form
my @out = sort { find_records() } @key;              # block form
```

- **Ταξινόμηση με επίγνωση locale.** Υπό `use locale` (χωρίς `':not_characters'`) η προεπιλεγμένη σύγκριση συμβολοσειρών ακολουθεί το τρέχον locale συνταξιοθεσίας αντί για τη σειρά σημείων κώδικα.
- **Συγκριτές XSUB.** Αν ο συγκριτής είναι XSUB, το ζεύγος περνά στη στοίβα ορισμάτων με τον τρόπο που οι συναρτήσεις XS συνήθως λαμβάνουν ορίσματα· οι `$a` και `$b` δεν ορίζονται.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `reverse` - αντιστρέφει μια λίστα· συνδυάστε με τη `sort` για φθίνουσες ταξινομήσεις όταν ο συγκριτής δύσκολα αντιστρέφεται με αντιμετάθεση των `$a` και `$b`
- `map` - το άλλο μισό του Schwartzian Transform· χρησιμοποιήστε για να επισυνάψετε και να αφαιρέσετε κλειδιά ταξινόμησης γύρω από μια `sort`
- `grep` - προφιλτράρετε τη λίστα (π.χ. αφαιρέστε `NaN` ή `undef`) πριν την παραδώσετε στη `sort`
- `$a` - το πρώτο στοιχείο στο ζεύγος σύγκρισης, ψευδωνυμοποιημένο ως καθολική πακέτου για τη διάρκεια της ταξινόμησης
- `$b` - το δεύτερο στοιχείο στο ζεύγος σύγκρισης, με τους ίδιους κανόνες διάρκειας ζωής και εμβέλειας με την `$a`
- `<=>` - αριθμητική τρίτιμη σύγκριση· ο συνηθισμένος τελεστής μέσα σε αριθμητικό μπλοκ ταξινόμησης
- `cmp` - τρίτιμη σύγκριση συμβολοσειρών· ο συνηθισμένος τελεστής μέσα σε μπλοκ ταξινόμησης συμβολοσειρών

Πίνακες

splice

Αφαιρεί και/ή αντικαθιστά μια τομή ενός πίνακα επιτόπου, και επιστρέφει τα στοιχεία που αφαιρέθηκαν.

Η `splice` είναι ο χειρουργός γενικής χρήσης για πίνακες. Κάθε άλλος τροποποιητής πίνακα - οι `push`, `pop`, `shift`, `unshift`, ανάθεση μεμονωμένου στοιχείου - μπορεί να γραφτεί ως κλήση `splice`. Επιλέξτε τη θέση με το `OFFSET`, επιλέξτε πόσα στοιχεία να αποκόψετε με το `LENGTH`, και προαιρετικά παρέχετε μια `LIST` αντικαταστατών. Ο πίνακας μεγαλώνει ή συρρικνώνεται για να χωρέσει, και λαμβάνετε πίσω ό,τι αποκόπηκε.

Σύνοψη

```
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
```

Τι επιστρέφεται

- **Περιβάλλον λίστας**: η λίστα των στοιχείων που αφαιρέθηκαν, στην αρχική τους σειρά. Κενή λίστα αν δεν αφαιρέθηκε τίποτα.
- **Βαθμωτό περιβάλλον**: το *τελευταίο* στοιχείο που αφαιρέθηκε, ή `undef` αν δεν αφαιρέθηκε τίποτα. Αυτό σπάνια είναι αυτό που θέλετε - καταφύγετε σε περιβάλλον λίστας εκτός αν χρειάζεστε ειδικά μόνο την ουρά.
- **Κενό περιβάλλον**: τίποτα· χρησιμοποιήστε αυτή τη μορφή όταν σας ενδιαφέρει μόνο η επιτόπια τροποποίηση.

Παραδείγματα

Βασική αφαίρεση - αποκοπή δύο στοιχείων ξεκινώντας από τη θέση 1:

```
my @a = ('a', 'b', 'c', 'd', 'e');
my @gone = splice @a, 1, 2;
# @a    == ('a', 'd', 'e')
# @gone == ('b', 'c')
```

Εισαγωγή χωρίς αφαίρεση - το `LENGTH` είναι `0`, οπότε δεν βγαίνει τίποτα και η `LIST` εμβολιάζεται στο `OFFSET`:

```
my @a = (1, 2, 5, 6);
splice @a, 2, 0, 3, 4;
# @a == (1, 2, 3, 4, 5, 6)
```

Αντικατάσταση τμήματος - το `LENGTH` και το μέγεθος αντικατάστασης δεν χρειάζεται να ταιριάζουν. Εδώ τρία στοιχεία φεύγουν, ένα φτάνει:

```
my @a = ('x', 'y', 'z', 'w', 'v');
my @gone = splice @a, 1, 3, 'Y';
# @a    == ('x', 'Y', 'v')
# @gone == ('y', 'z', 'w')
```

Αφαίρεση της ουράς - παραλείψτε εντελώς το `LENGTH` για αποκοπή από το `OFFSET` μέχρι το τέλος:

```
my @a = (1, 2, 3, 4, 5);
my @tail = splice @a, 3;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
# @a    == (1, 2, 3)
# @tail == (4, 5)
```

Αρνητικό OFFSET μετράει από το τέλος. Pop του τελευταίου στοιχείου:

```
my @a = ('a', 'b', 'c');
my $last = splice @a, -1;
# @a    == ('a', 'b')
# $last == 'c'
```

Κλασικό ιδίωμα - κατανάλωση ενός πίνακα $n στοιχεία τη φορά:

```
while (my @chunk = splice @queue, 0, $n) {
    process(@chunk);
}
```

Όλες οι συνηθισμένες πρωτογενείς λειτουργίες πίνακα γραμμένες ως splice (υποθέτοντας $#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
```

Οριακές περιπτώσεις

- **Αρνητικό OFFSET** μετράει από το τέλος: το -1 είναι το τελευταίο στοιχείο, το -2 το προηγούμενο, και ούτω καθεξής. Αν το μέτρο υπερβαίνει το μήκος του πίνακα, η Perl εκπέμπει προειδοποίηση και περικόπτει στο 0.
- **Αρνητικό LENGTH** αφήνει τόσα στοιχεία στο τέλος του πίνακα ανέπαφα. Η splice @a, 2, -1 αφαιρεί από τη θέση 2 μέχρι αλλά όχι συμπεριλαμβανομένου του τελευταίου στοιχείου.
- **OFFSET πέρα από το τέλος**: με ρητό LENGTH, η Perl εκπέμπει προειδοποίηση και εμβολιάζει στο τέλος του πίνακα (ισοδύναμο με splice @a, scalar(@a), 0, LIST). Χωρίς LENGTH, δεν αφαιρείται τίποτα.
- **LENGTH πέρα από το τέλος**: περικόπτεται σιωπηλά. Η splice @a, 1, 999 απλώς αφαιρεί τα πάντα από τη θέση 1 και μετά.
- **Παραλείπονται και το OFFSET και το LENGTH**: αφαιρείται κάθε στοιχείο. Ο πίνακας μένει κενός.
- **Το μέγεθος της λίστας αντικατάστασης διαφέρει από το LENGTH**: ο πίνακας μεγαλώνει ή συρρικνώνεται για να ταιριάξει. Δεν απαιτείται το scalar(@list) να ισούται με το LENGTH - αυτή η ασυμμετρία είναι όλο το νόημα της splice.
- **Αναφορά πίνακα**: κάντε dereference inline. Οι splice @$ref, 0, 1 και splice @{ $obj->{items} }, -2 δουλεύουν και οι δύο.
- **Η splice σε τομή δεν υποστηρίζεται**: η splice @a[1..3], 0, 1 είναι συντακτικό σφάλμα. Η splice απαιτεί κατονομασμένο πίνακα ή αποαναφερόμενο

πίνακα, όχι έκφραση λίστας. Για να λειτουργήσετε σε εύρος, περάστε τα όρια ως OFFSET και LENGTH.

- **Το κενό περιβάλλον είναι εντάξει**. Αν δεν χρειάζεται η τιμή επιστροφής, καλέστε την splice ως εντολή - σε αυτή την περίπτωση δεν γίνεται καμία εκχώρηση της λίστας αφαιρεμένων στοιχείων.
- **Βαθμωτές εκφράσεις ως πρώτο όρισμα**: δεν επιτρέπονται. Ένα πειραματικό χαρακτηριστικό το επέτρεπε από την 5.14 έως την 5.22 και αφαιρέθηκε στην 5.24. Το πρώτο όρισμα πρέπει να είναι πίνακας (ή αποαναφερόμενο arrayref).

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- push - προσάρτηση στο τέλος· ισοδύναμο με `splice @a, scalar(@a), 0, LIST`
- pop - αφαίρεση του τελευταίου στοιχείου· ισοδύναμο με `splice @a, -1`
- shift - αφαίρεση του πρώτου στοιχείου· ισοδύναμο με `splice @a, 0, 1`
- unshift - πρόθεμα στην αρχή· ισοδύναμο με `splice @a, 0, 0, LIST`
- delete - επιφανειακά παρόμοιο αλλά σημασιολογικά διαφορετικό. Η `delete $a[$i]` αφήνει τρύπα (η θέση γίνεται undef) και **δεν** αναριθμεί τις μετέπειτα θέσεις· το μήκος του πίνακα δεν αλλάζει εκτός αν η διαγραμμένη θέση ήταν η τελευταία. Η `splice @a, $i, 1` αφαιρεί φυσικά τη θέση και μετατοπίζει κάθε μετέπειτα στοιχείο μία θέση πίσω. Χρησιμοποιήστε delete για αραιούς πίνακες όπου η ταυτότητα της θέσης έχει σημασία· χρησιμοποιήστε splice όταν θέλετε ο πίνακας να κλείσει.
- grep - φιλτράρει έναν πίνακα σε έναν νέο μέσω κατηγορήματος· καταφύγετε σε αυτό αντί για splice όταν θέλετε «κράτησε μόνο όσα ταιριάζουν» αντί για «κόψε ένα συνεχόμενο εύρος»

Κανονικές εκφράσεις και αντιστοίχιση μοτίβων

split

Κόβει μια συμβολοσειρά σε λίστα πεδίων χρησιμοποιώντας διαχωριστή regex.

Η split σαρώνει το EXPR για αντιστοιχίες του PATTERN και επιστρέφει τα κομμάτια μεταξύ των αντιστοιχιών ως λίστα. Το ίδιο το κείμενο που ταίριαξε είναι ο διαχωριστής και **δεν** περιλαμβάνεται στο αποτέλεσμα - εκτός αν το μοτίβο περιέχει ομάδες σύλληψης, οπότε κάθε σύλληψη γίνεται επιπλέον στοιχείο στην έξοδο. Σε βαθμωτό περιβάλλον επιστρέφεται το μέγεθος της λίστας.

Αν το PATTERN παραλειφθεί, η split συμπεριφέρεται όπως η προεπιλογή του awk: ακολουθίες λευκών χαρακτήρων αποτελούν τον διαχωριστή και οποιοιδήποτε προπορευόμενοι λευκοί χαρακτήρες στο EXPR αφαιρούνται. Αν το EXPR παραλειφθεί, διαχωρίζεται το $_.

Σύνοψη

```
split /PATTERN/, EXPR, LIMIT
split /PATTERN/, EXPR
split /PATTERN/
split
```

Τι επιστρέφεται

Μια λίστα υποσυμβολοσειρών (πεδίων) σε περιβάλλον λίστας· το πλήθος των πεδίων σε βαθμωτό περιβάλλον. Το κείμενο του διαχωριστή δεν αποτελεί ποτέ μέρος πεδίου.

```
my @fields = split /,/, "a,b,c";       # ("a", "b", "c")
my $n      = split /,/, "a,b,c";       # 3
```

Ο διαχωρισμός ενός EXPR που είναι η κενή συμβολοσειρά πάντα αποδίδει μηδέν πεδία, ανεξαρτήτως LIMIT:

```
my @x = split /,/, "", -1;             # ()
```

Πριν την Perl 5.11, η split σε κενό ή βαθμωτό περιβάλλον αντικαθιστούσε το @_. Η σύγχρονη Perl δεν το κάνει· μην βασίζεστε ποτέ σε εκείνη την παλιά παρενέργεια.

Το όρισμα LIMIT

Το LIMIT ελέγχει πόσα πεδία παράγονται και, το πιο κρίσιμο, αν τα κενά πεδία στο τέλος διατηρούνται.

- **Θετικό LIMIT** - άνω φράγμα στον αριθμό των πεδίων. Το EXPR διαχωρίζεται το πολύ LIMIT - 1 φορές· το τελευταίο πεδίο κρατά το υπόλοιπο της συμβολοσειράς αυτούσιο. LIMIT = 1 σημαίνει κανέναν διαχωρισμό - λαμβάνετε ολόκληρη τη συμβολοσειρά ως ένα στοιχείο.

  ```
  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")
  ```

- **Αρνητικό LIMIT** - αντιμετωπίζεται ως αυθαίρετα μεγάλο. Παράγονται όσα περισσότερα πεδία γίνεται, συμπεριλαμβανομένων **όλων** των κενών πεδίων στο τέλος.

  ```
  my @x = split /,/, "a,b,c,,,", -1;   # ("a", "b", "c", "", "", "
  ↪")
  ```

- **Παραλειπόμενο ή μηδενικό LIMIT** - όπως αρνητικό, **εκτός** από το ότι τα κενά πεδία στο τέλος αφαιρούνται. Τα κενά πεδία στην αρχή διατηρούνται πάντα.

  ```
  my @x = split /,/, "a,b,c,,,";       # ("a", "b", "c")
  my @x = split /,/, ",,a,b";          # ("", "", "a", "b")
  ```

- **Έμμεσο LIMIT σε ανάθεση λίστας.** Όταν γίνεται ανάθεση σε λίστα σταθερού μεγέθους, η Perl θέτει το LIMIT σε ένα παραπάνω από τον αριθμό των στόχων, ώστε τα πεδία στο τέλος να μη χρειάζεται να σαρωθούν. Το παρακάτω λαμβάνει αυτόματα `LIMIT = 3`:

```
my ($login, $passwd) = split /:/;
```

  Σε χρονοκρίσιμο κώδικα, περάστε ρητό LIMIT αντί να αφήσετε ολόκληρη τη συμβολοσειρά να σαρωθεί.

Η περίπτωση λευκών χαρακτήρων τύπου awk: `split " "`

A literal single space as the pattern - `split " "`, not `split / /`

- ενεργοποιεί μια ειδική περίπτωση: το PATTERN αντιμετωπίζεται ως `/\s+/` **και** οποιοιδήποτε προπορευόμενοι λευκοί χαρακτήρες στο EXPR αφαιρούνται πριν τον διαχωρισμό. Αυτό ταιριάζει με την κλασική συμπεριφορά του awk.

```
my @x = split " ", "  Quick brown fox\n";
# ("Quick", "brown", "fox")

my @x = split " ", "RED\tGREEN\tBLUE";
# ("RED", "GREEN", "BLUE")
```

Για διαχωρισμό σε ένα *μονό* κυριολεκτικό κενό μόνο, χρησιμοποιήστε τη μορφή regex `/ /` - δεν υπόκειται σε ειδική περίπτωση:

```
my @x = split / /, " abc";              # ("", "abc")
```

Από την Perl 5.18, η ενεργοποίηση είναι οποιαδήποτε έκφραση της οποίας η τιμή είναι η συμβολοσειρά ενός χαρακτήρα `" "` (όχι μόνο κυριολεκτική), και από την Perl 5.28 ο κανόνας λειτουργεί σωστά υπό το `use feature 'unicode_strings'`. Υπό την Perl 5.39.9+ ο προεπιλεγμένος τροποποιητής `/x` **δεν** επηρεάζει το `split STRING`, οπότε το `split " "` εξακολουθεί να σημαίνει εξομοίωση awk ακόμη και μέσα σε `use re "/x"`. Αν θέλετε να διαχωρίσετε σε ένα κυριολεκτικό κενό υπό το `use re "/x"`, γράψτε `split /(?-x: )/` ή `split /\x{20}/`.

Αν το PATTERN παραλειφθεί εντελώς, εξ ορισμού είναι `" "`, οπότε το γυμνό `split` αποτελεί τη μορφή τύπου awk.

Η περίπτωση κενού μοτίβου: διαχωρισμός σε χαρακτήρες

Αν το PATTERN ταιριάζει με την κενή συμβολοσειρά, ο διαχωρισμός συμβαίνει σε κάθε θέση αντιστοιχίας - δηλαδή μεταξύ των χαρακτήρων.

```
my @x = split //, "abc";                # ("a", "b", "c")
```

Ως κανόνας ειδικός για την `split`, ο γυμνός τελεστής αντιστοίχισης `//` **δεν** είναι εδώ η μορφή «επανάληψε την τελευταία επιτυχημένη αντιστοιχία»· είναι το κυριολεκτικά κενό μοτίβο. Μια αντιστοιχία μηδενικού πλάτους ακριβώς στην αρχή του EXPR δεν παράγει ποτέ κενό αρχικό πεδίο, για αυτόν τον λόγο ο διαχωρισμός προπορευόμενου κενού χειρίζεται έτσι:

```
my @x = split //, " abc";              # (" ", "a", "b", "c")   - 4,␣
↪not 5
my @x = split //, " abc", -1;          # (" ", "a", "b", "c", "") -␣
↪trailing empty with -1
```

Μια αντιστοιχία θετικού πλάτους στην αρχή *όντως* παράγει κενό αρχικό πεδίο:

```
my @x = split / /, " abc";             # ("", "abc")
```

Οι ομάδες σύλληψης γίνονται επιπλέον στοιχεία

Αν το PATTERN περιέχει ομάδες σύλληψης, κάθε σύλληψη εισάγεται στη λίστα εξόδου για κάθε αντιστοιχία διαχωριστή, με τη σειρά που δηλώνονται οι ομάδες. Μια ομάδα που δεν συμμετέχει στην αντιστοιχία συνεισφέρει `undef`. Αυτά τα επιπλέον στοιχεία **δεν** μετρούν για το 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")
```

Χρησιμοποιήστε αυτό όταν θέλετε οι διαχωριστές να διατηρούνται μαζί με τα πεδία - τυπικό για tokenisers όπου οι οριοθέτες αποτελούν οι ίδιοι μέρος του ρεύματος εξόδου.

Καθολική κατάσταση που επηρεάζει

- `$_` - το EXPR το έχει ως προεπιλογή όταν δεν δίνεται δεύτερο όρισμα.
- `@_` - η σύγχρονη Perl **δεν** το αγγίζει. Σχετικό μόνο αν υποστηρίζετε perl πριν την 5.11.
- Οι μεταβλητές αντιστοίχισης της μηχανής regex ($1, $2, ..., $&, $', $``) **δεν** ορίζονται ως παρενέργεια της split, ακόμη και όταν το PATTERN συλλαμβάνει - οι συλλήψεις πηγαίνουν στη λίστα επιστροφής.

Παραδείγματα

Κλασική γραμμή τύπου CSV, χωρίς ενσωματωμένα κόμματα:

```
my @cells = split /,/, "alice,bob,carol";    # ("alice", "bob",
↪"carol")
```

Ανάλυση μιας γραμμής `/etc/passwd` σε γνωστά πεδία· το έμμεσο `LIMIT` σταματά στους πρώτους επτά διαχωριστές:

```
my ($user, $pw, $uid, $gid, $gecos, $home, $shell)
    = split /:/, $line;
```

Tokenising λευκών χαρακτήρων τύπου awk, με τα προπορευόμενα κενά να απορρίπτονται:

```
my @words = split " ", "   one\ttwo  three\n";
# ("one", "two", "three")
```

Χαρακτήρες μιας συμβολοσειράς - το κενό μοτίβο:

```
my @chars = split //, "hello";              # ("h", "e", "l", "l",
↪"o")
```

Διατήρηση κάθε κενού πεδίου στο τέλος για αυστηρό αναγνώστη μετρήματος στηλών:

```
my @cols = split /\t/, $line, -1;
```

Διατήρηση των διαχωριστών μέσω σύλληψης - αναπαραγωγή με round-trip:

```
my @parts = split /([,;])/, "a,b;c";        # ("a", ",", "b", ";",
↪"c")
my $same  = join "", @parts;                # "a,b;c"
```

Η προεπιλεγμένη μορφή διαβάζει το `$_` και κάνει διαχωρισμό λευκών χαρακτήρων τύπου awk:

```
for (@lines) {
    my @f = split;                          # split " ", $_
    ...
}
```

Οριακές περιπτώσεις

- **Κενή είσοδος** - το `split /X/, ""` επιστρέφει την κενή λίστα για κάθε `LIMIT`, συμπεριλαμβανομένου του `-1`.
- **`LIMIT = 1`** - δεν γίνεται διαχωρισμός· ολόκληρη η συμβολοσειρά είναι το ένα και μοναδικό πεδίο. Χρήσιμο όταν θέλετε τη σημασιολογία της `split` υπό συνθήκη χωρίς να χάσετε την είσοδο.
- **Μοτίβο `/^/`** - αντιμετωπίζεται σαν `/^/m` ώστε να ταιριάζει στην αρχή κάθε γραμμής, που είναι η μόνη χρήσιμη συμπεριφορά. Διαχωρίζει μια συμβολοσειρά σε γραμμές διατηρώντας τις νέες γραμμές.
- **Αντιστοιχία μηδενικού πλάτους στη θέση 0** - δεν παράγει ποτέ κενό αρχικό πεδίο (δείτε `split //, "abc"` παραπάνω).
- **Αντιστοιχία θετικού πλάτους στη θέση 0** - παράγει κενό αρχικό πεδίο, πάντα.
- **Αντιστοιχία στο τέλος της συμβολοσειράς** - παράγει κενό πεδίο στο τέλος, το οποίο στη συνέχεια αφαιρείται εκτός αν το `LIMIT` είναι μη μηδενικό.

- **Μοτίβα με τροποποιητές** - εφαρμόζονται οι συνήθεις τροποποιητές `qr//` (`/i`, `/m`, `/s`, `/x`, `/u`, `/a`, `/l`, `/n`). Το μοτίβο δεν χρειάζεται να είναι κυριολεκτικό· οποιαδήποτε έκφραση που παράγει regex ή συμβολοσειρά λειτουργεί, και τα αντικείμενα `qr//` γίνονται δεκτά.
- **Συμβολοσειρά μονού κενού που δεν είναι κυριολεκτική** - από την Perl 5.18 και μετά, οποιαδήποτε έκφραση που αξιολογείται σε `" "` ενεργοποιεί την ειδική περίπτωση τύπου awk, όχι μόνο το κυριολεκτικό `" "`.
- **`split " "` έναντι `split / /`** - το πρώτο είναι λευκοί χαρακτήρες τύπου awk (`\s+` με αφαίρεση των προπορευόμενων κενών), το δεύτερο είναι ένα κυριολεκτικό μονό κενό. Τα δύο δεν είναι ανταλλάξιμα.
- **Λευκοί χαρακτήρες Unicode** - υπό το `use feature 'unicode_strings'` (προεπιλογή από την 5.28 για την περίπτωση τύπου awk), το `split " "` αντιμετωπίζει και τους λευκούς χαρακτήρες Unicode ως διαχωριστή. Έξω από αυτή την εμβέλεια η συμπεριφορά επηρεάζεται από το «Unicode Bug».

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `join` - αντίστροφη πράξη· συρράπτει μια λίστα ξανά σε συμβολοσειρά με επιλεγμένη κόλλα
- `m` - ο τελεστής αντιστοίχισης· όταν θέλετε να *βρείτε* κάτι αντί να κόψετε τη συμβολοσειρά γύρω από αυτό
- `qr` - προμεταγλώττιση μοτίβου μία φορά για επανειλημμένες κλήσεις `split` σε καυτό βρόχο
- `index` - εντοπισμός σταθερής υποσυμβολοσειράς χωρίς κατασκευή της πλήρους λίστας πεδίων· φθηνότερο όταν χρειάζεστε μόνο τον πρώτο διαχωρισμό
- `substr` - εξαγωγή με μετατόπιση byte/χαρακτήρα όταν τα όρια των πεδίων είναι θέσεις, όχι οριοθετημένα

Βαθμωτά και συμβολοσειρές

sprintf

Κατασκευάζει μορφοποιημένη συμβολοσειρά από ένα πρότυπο μορφοποίησης και μια λίστα τιμών.

Η `sprintf` παίρνει μια συμβολοσειρά `FORMAT` που περιέχει κυριολεκτικό κείμενο με ενσωματωμένους προσδιοριστές μετατροπής που εισάγονται από `%`, καταναλώνει τιμές από τη `LIST` για να συμπληρώσει κάθε προσδιοριστή, και επιστρέφει τη συμβολοσειρά που προκύπτει. Είναι η ίδια μηχανική που οδηγεί την `printf`, χωρίς την εγγραφή σε filehandle. Η Perl υλοποιεί τον δικό της μορφοποιητή· δεν καλεί την `sprintf(3)` της βιβλιοθήκης C εκτός από την τελική παραγωγή ψηφίων κινητής υποδιαστολής, οπότε μη τυποποιημένες επεκτάσεις που τυχόν υπάρχουν στη δική σας libc δεν είναι διαθέσιμες εδώ.

Σύνοψη

```
my $s = sprintf $format, @args;
my $s = sprintf "%-10s %5d", $name, $count;
my $s = sprintf "%.3f", $x;
```

Τι επιστρέφεται

Μια μοναδική βαθμωτή συμβολοσειρά. Ποτέ λίστα, ανεξάρτητα από το περιβάλλον. Αν το FORMAT δεν περιέχει προσδιοριστές %, η τιμή επιστροφής είναι το FORMAT αμετάβλητο και η LIST αγνοείται. Αν το FORMAT αναφέρει περισσότερα ορίσματα από όσα παρέχει η LIST, οι τιμές που λείπουν μεταχειρίζονται ως undef (αριθμητικό μηδέν ή κενή συμβολοσειρά ανάλογα με τη μετατροπή), και εκδίδεται προειδοποίηση `Missing argument in sprintf` υπό `use warnings`.

Η διαβίβαση πίνακα ως πρώτο όρισμα είναι σχεδόν πάντα λάθος - ο πίνακας αποτιμάται σε βαθμωτό περιβάλλον, οπότε η Perl χρησιμοποιεί το πλήθος των στοιχείων του ως μορφή. Χρησιμοποιήστε `sprintf $format, @rest` με τη μορφή κρατημένη ξεχωριστά, ή `sprintf "@array"` για την περίπτωση πίνακα ως συμβολοσειρά.

Καθολική κατάσταση που επηρεάζει

- Το locale LC_NUMERIC, όταν είναι ενεργή η `use locale` και έχει κληθεί η POSIX::setlocale, καθορίζει τον χαρακτήρα δεκαδικής υποδιαστολής που χρησιμοποιούν τα %e, %f, %g και οι κεφαλαίες παραλλαγές τους. Χωρίς `use locale` η δεκαδική υποδιαστολή είναι πάντα `.`, ακόμη και αν το locale της διεργασίας λέει κάτι διαφορετικό.
- Καμία άλλη ειδική μεταβλητή δεν συμβουλεύεται. Η `sprintf` δεν διαβάζει την $_· απαιτείται ρητή λίστα.

Προσδιοριστές μετατροπής

Το βασικό σύνολο, με τη σειρά που πραγματικά θα τους χρειαστείτε:

Προσ	Σημασία
%s	Συμβολοσειρά. Μετατρέπει το όρισμα σε συμβολοσειρά.
%d, %i	Προσημασμένος ακέραιος σε δεκαδικό. Οι αριθμοί κινητής υποδιαστολής αποκόπτονται προς το μηδέν.
%u	Μη προσημασμένος ακέραιος σε δεκαδικό.
%f, %F	Κινητή υποδιαστολή σταθερής θέσης. Προεπιλογή 6 ψηφία μετά τη δεκαδική.
%e, %E	Επιστημονική σημειογραφία. Το κεφαλαίο E επιλέγει την κεφαλαία μορφή.
%g, %G	%e ή %f, όποιο είναι συντομότερο. Η ακρίβεια αφορά τα *σημαντικά ψηφία*, όχι τα ψηφία μετά τη δεκαδική.
%x, %X	Μη προσημασμένος ακέραιος σε δεκαεξαδικό· το %X χρησιμοποιεί κεφαλαία A–F.
%o	Μη προσημασμένος ακέραιος σε οκταδικό.
%b, %B	Μη προσημασμένος ακέραιος σε δυαδικό.
%c	Μονός χαρακτήρας του οποίου το σημείο κώδικα είναι ο δοθείς ακέραιος.
%%	Κυριολεκτικό %. Δεν καταναλώνει όρισμα.
%p	Διεύθυνση τιμής Perl, σε δεκαεξαδικό. Μόνο για διαγνωστική χρήση.
%n	Αποθηκεύει τον αριθμό των χαρακτήρων που έχουν γραφτεί μέχρι τώρα στο επόμενο όρισμα (που πρέπει να είναι εγγράψιμο βαθμωτό).
%a, %A	Δεκαεξαδική κινητή υποδιαστολή.

Τα κεφαλαία συνώνυμα μεγέθους %D %U %O και %F υπάρχουν για λόγους ιστορικής συμβατότητας· προτιμήστε τις πεζές μορφές.

Σημαίες, πλάτος, ακρίβεια

Μεταξύ % και του γράμματος μετατροπής, με τη σειρά:

1. **Δείκτης ορίσματος** - το N$ επιλέγει το N-στό όρισμα (αρχίζοντας από 1): `sprintf '%2$s %1$s', 'world', 'hello'` αποδίδει `"hello world"`.
2. **Σημαίες** - οποιοσδήποτε συνδυασμός των:
   - `-` στοίχιση αριστερά μέσα στο πεδίο (προεπιλογή είναι η στοίχιση δεξιά).
   - `+` προθεματίζει τους μη αρνητικούς αριθμούς με +.
   - `` ` ` `` (κενό) προθεματίζει τους μη αρνητικούς αριθμούς με κενό. Όταν δίνονται και κενό και +, υπερισχύει το +.
   - `0` συμπληρώνει τις αριθμητικές μετατροπές με μηδενικά αντί για κενά. Αγνοείται όταν είναι ορισμένο και το - ή όταν δίνεται ακρίβεια για ακεραίους.
   - `#` εναλλακτική μορφή: πρόθεμα `0` για οκταδικό, πρόθεμα `0x`/`0X` για μη μηδενικό δεκαεξαδικό, πρόθεμα `0b`/`0B` για μη μηδενικό δυαδικό.
3. **Ελάχιστο πλάτος** - δεκαδικός αριθμός, ή * για να ληφθεί το πλάτος από το επόμενο όρισμα, ή *N$ για ρητό όρισμα. Το αρνητικό πλάτος (μέσω *) είναι ισοδύναμο με τη σημαία -.

4. **Ακρίβεια** - . ακολουθούμενη από δεκαδικό αριθμό, ή . * (επόμενο όρισμα), ή . *N$. Η σημασία εξαρτάται από τη μετατροπή:
   - %f / %e: ψηφία μετά τη δεκαδική υποδιαστολή (προεπιλογή 6).
   - %g: συνολικά σημαντικά ψηφία.
   - %d / %x / %o / %b: ελάχιστα ψηφία· ο αριθμός συμπληρώνεται με μηδενικά μέχρι αυτό το πλάτος. Η σημαία 0 αγνοείται όταν είναι παρούσα ακρίβεια για ακεραίους.
   - %s: μέγιστο πλάτος· αποκόπτει.
5. **Τροποποιητής μεγέθους** - hh, h, j, l, q, L, ll, t, z, V. Ερμηνεύει το όρισμα ως συγκεκριμένο τύπο C. Σχεδόν ποτέ χρήσιμο από Perl - η Perl έχει έναν τύπο ακεραίου και έναν τύπο float, και οι τιμές επιπέδου Perl δεν υπερχειλίζουν στα μεγέθη που επιλέγουν αυτοί οι τροποποιητές. Το V είναι no-op που σημαίνει «το προεπιλεγμένο μέγεθος της Perl».
6. **Γράμμα μετατροπής** - μία από τις εγγραφές του πίνακα παραπάνω.

Η σημαία διανύσματος

Προθεματίστε το γράμμα μετατροπής με v (προαιρετικά *v ή *N$v για να παρέχετε διαχωριστή) και η Perl μεταχειρίζεται το όρισμα ως διάνυσμα σημείων κώδικα, εφαρμόζοντας τη μετατροπή σε κάθε σημείο κώδικα και ενώνοντας τα αποτελέσματα με . (ή τον διαχωριστή που παρασχέθηκε):

```
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"
```

Συνηθισμένες χρήσεις: απεικόνιση της $^V, μορφοποίηση ακολουθιών bytes IPv4 / IPv6, ή απόρριψη bitstrings. Το όρισμα διαβάζεται πάντα ως συμβολοσειρά σημείων κώδικα, όχι ως λίστα.

Παραδείγματα

Συμπλήρωση και στοίχιση συμβολοσειρών σε στήλη αναφοράς σταθερού πλάτους:

```
printf "%-10s %5d\n", "widgets", 42;   # "widgets        42\n"
printf "%-10s %5d\n", "sprockets", 7;  # "sprockets       7\n"
```

Συμπλήρωση ακεραίου σταθερού πλάτους με μηδενικά, π.χ. για λεξικογραφικά κλειδιά ταξινόμησης ή αριθμούς γραμμών:

```
sprintf "%08d", 1234;                  # "00001234"
sprintf "%05d", -7;                    # "-0007"  (sign consumes a␣
↪slot)
```

Στρογγυλοποίηση σε σταθερό αριθμό δεκαδικών θέσεων. Η sprintf κάνει τη στρογγυλοποίηση· η printf είναι ο συνηθισμένος τρόπος εξαναγκασμού μιας τιμής για εμφάνιση:

```
sprintf "%.3f", 3.14159265;           # "3.142"
sprintf "%.0f", 2.5;                  # "2" or "3" - banker's␣
↪rounding
                                      # is platform-dependent
```

Επιστημονική σημειογραφία, με και χωρίς επιλεγμένη ακρίβεια:

```
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)
```

Δεκαεξαδικά, οκταδικά και δυαδικά με το πρόθεμα εναλλακτικής μορφής #:

```
sprintf "%#x",  255;                  # "0xff"
sprintf "%#o",  8;                    # "010"
sprintf "%#b",  10;                   # "0b1010"
sprintf "%08b", 10;                   # "00001010"   (no prefix,␣
↪width 8)
```

Χαρακτήρας από σημείο κώδικα - το %c είναι ουσιαστικά η `chr` δρομολογημένη μέσα από τη μηχανή μορφοποίησης:

```
sprintf "%c", 65;                     # "A"
sprintf "%c", 0x2014;                 # "-"  (em dash)
```

Πλάτος πεδίου από τη λίστα ορισμάτων - χρήσιμο όταν το πλάτος υπολογίζεται σε χρόνο εκτέλεσης:

```
my $w = 12;
sprintf "%-*s|", "name", $w;          # "name        |"
```

Ορίσματα εκτός σειράς - χρήσιμα όταν η ίδια τιμή εμφανίζεται πολλές φορές ή όταν τοπικοποιημένες μορφές αναδιατάσσουν τα placeholders:

```
sprintf "%2\$s loves %1\$s", "Alice", "Bob";   # "Bob loves Alice"
```

Κατασκευή συμβολοσειράς διεύθυνσης IPv4:

```
sprintf "%vd", "\x7f\x00\x00\x01";    # "127.0.0.1"
```

Αποτύπωση μήκους μέσω %n (το λαμβάνον όρισμα πρέπει να είναι εγγράψιμο βαθμωτό):

```
my $written;
my $s = sprintf "hello%n, world", $written;
# $s       is "hello, world"
# $written is 5
```

Οριακές περιπτώσεις

- **Αποκοπή ακεραίων στο %d με floats**: το %d μετατρέπει μέσω εξαναγκασμού σε ακέραιο, που αποκόπτει προς το μηδέν. Το `sprintf "%d", 3.9` είναι `"3"`, όχι `"4"`. Χρησιμοποιήστε `%.0f` αν θέλετε εμφάνιση με τραπεζική στρογγυλοποίηση, ή `int` / `sprintf "%.0f"` ρητά πρώτα.
- **%s σε αντικείμενο**: εκτελείται η μετατροπή σε συμβολοσειρά, που μπορεί να επικαλεστεί υπερφορτωμένη `""` ή να υποχωρήσει σε `"ClassName=HASH(0x…)"`. Δεν υπάρχει τρόπος καταστολής της υπερφόρτωσης μέσα στην `sprintf`· καλέστε πρώτα την `overload::StrVal($obj)` αν χρειάζεστε την ωμή μορφή.
- **Ορίσματα που λείπουν**: το `sprintf "%s %s", "one"` επιστρέφει `"one "` και προειδοποιεί `Missing argument in sprintf at …` υπό `use warnings`. Δεν εγείρεται εξαίρεση.
- **Πάρα πολλά ορίσματα**: τα επιπλέον αγνοούνται σιωπηλά. Καμία προειδοποίηση.
- **Το %s με ακρίβεια αποκόπτει, το %d με ακρίβεια συμπληρώνει με μηδενικά**. Το `sprintf "%.3s", "abcdef"` είναι `"abc"`. Το `sprintf "%.3d", 7` είναι `"007"`.
- **Σημαία %0 έναντι ακρίβειας σε ακεραίους**: υπερισχύει η ακρίβεια. Το `sprintf "%05.3d", 7` είναι `"  007"`, όχι `"00007"` - η σημαία `0` αγνοείται από τη στιγμή που δίνετε ρητή ακρίβεια ακεραίου.
- **Αρνητικό πλάτος μέσω ***: το `sprintf "%*s", -6, "x"` είναι `"x     "`. Η αρνητική ακρίβεια μέσω `.*` μετράει ως καθόλου ακρίβεια.
- **NaN και Inf**: το `sprintf "%f", 9**9**9` αποδίδει `"Inf"`· το `sprintf "%f", -sin(9**9**9)` αποδίδει `"NaN"`. Η ακριβής γραφή (`"Inf"` έναντι `"inf"`, `"NaN"` έναντι `"nan"`) ακολουθεί τη σύμβαση της εν λόγω έκδοσης της Perl, όχι τη διάκριση πεζών-κεφαλαίων %f / %F.
- **Δεκαδική υποδιαστολή locale**: με `use locale` και locale de_DE, το `sprintf "%.2f", 1.5` μπορεί να παράγει `"1,50"`. Χωρίς `use locale` το αποτέλεσμα είναι πάντα `"1.50"`. Κώδικας που γράφει αριθμητική έξοδο για κατανάλωση από άλλα προγράμματα θα έπρεπε είτε να αποφεύγει την `use locale` σε αυτή την εμβέλεια είτε να χρησιμοποιεί `use locale ':not_characters'` και να μορφοποιεί αριθμούς εκτός της επιρροής του locale.
- **Το %p είναι μόνο διαγνωστικό**: η δεκαεξαδική τιμή δείκτη αλλάζει από εκτέλεση σε εκτέλεση και είναι άνευ σημασίας εκτός της τρέχουσας διεργασίας. Μην τη χρησιμοποιείτε ως κλειδί ταυτότητας - δύο αναφορές μπορεί να μοιράζονται την ίδια διεύθυνση αφού η πρώτη έχει αποδεσμευτεί.
- **Πίνακας διαβιβασμένος ως FORMAT**: το `sprintf @a` μορφοποιεί χρησιμοποιώντας τη βαθμωτή μέτρηση του `@a` ως συμβολοσειρά μορφής. Σχεδόν πάντα σφάλμα. Κρατήστε πάντα τη μορφή σε δικό της βαθμωτό.
- **Μη υποστηριζόμενος τροποποιητής μεγέθους**: σε πλατφόρμες που στερούνται π.χ. long doubles, το μέγεθος `L`/`q`/`ll` σε μετατροπή float υποχωρεί σιωπηλά στο προεπιλεγμένο μέγεθος, και εκδίδεται προειδοποίηση `printf` υπό `use warnings`. Για να την προωθήσετε σε εξαίρεση, χρησιμοποιήστε `use warnings FATAL => 'printf'`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `printf` - ίδια μηχανή μορφοποίησης, αλλά γράφει σε filehandle αντί να επιστρέφει τη συμβολοσειρά
- `print` - χρησιμοποιήστε την με ένα προ-κατασκευασμένο αποτέλεσμα της `sprintf` όταν θέλετε να μορφοποιήσετε μία φορά και να γράψετε σε αρκετά handles
- `chr` - μονό σημείο κώδικα σε μονό χαρακτήρα· ισοδύναμο με `sprintf "%c", $n`
- `hex`, `oct` - αντίστροφες λειτουργίες: αναλύουν συμβολοσειρά σε δεκαεξαδικό/οκταδικό/δυαδικό πίσω σε αριθμό
- `pack` - κατασκευάζει δυαδική συμβολοσειρά από πρότυπο· καταφύγετε σε αυτή όταν η έξοδος είναι bytes, όχι αναγνώσιμη από άνθρωπο μορφή
- `$_` - *δεν* συμβουλεύεται από την `sprintf`· σε αντίθεση με πολλές ενσωματωμένες, χρειάζεται πάντα ρητή λίστα

Αριθμητικές συναρτήσεις

sqrt

Επιστρέφει τη μη αρνητική τετραγωνική ρίζα του EXPR.

Η `sqrt` υπολογίζει την **κύρια τετραγωνική ρίζα** - τον μοναδικό μη αρνητικό πραγματικό αριθμό του οποίου το τετράγωνο ισούται με EXPR. Για κάθε `EXPR >= 0` το αποτέλεσμα είναι `>= 0`· η `sqrt` ποτέ δεν επιστρέφει την αρνητική ρίζα. Αν παραλειφθεί το EXPR, χρησιμοποιείται το `$_`.

Ο υπολογισμός εκτελείται σε κινητή υποδιαστολή διπλής ακρίβειας (IEEE 754 `binary64`), οπότε το αποτέλεσμα είναι float ακόμα και όταν η είσοδος είναι ακέραιος και ακόμα και όταν η μαθηματική απάντηση είναι η ίδια ακέραιος.

Σύνοψη

```
sqrt EXPR
sqrt
sqrt(EXPR)
```

Τι επιστρέφεται

Αριθμός κινητής υποδιαστολής: η μη αρνητική τετραγωνική ρίζα του EXPR. Για αρνητική είσοδο το αποτέλεσμα είναι `NaN` (και εκπέμπεται προειδοποίηση υπό `use warnings`)· δείτε *Οριακές περιπτώσεις*. Για ρίζες με μιγαδικές τιμές φορτώστε το , που υπερφορτώνει την `sqrt` ώστε η `sqrt(-1)` να επιστρέφει `i`.

```
my $r = sqrt 2;                     # 1.4142135623731
my $r = sqrt;                       # sqrt of $_
```

Το προεπιλεγμένο όρισμα, η προτεραιότητα, και η ανάλυση

Τρεις λεπτομέρειες του αναλυτή διαμορφώνουν κάθε κλήση `sqrt`:

- **Χωρίς όρισμα χρησιμοποιεί το `$_`**. Η `sqrt` χωρίς όρισμα και χωρίς παρενθέσεις διαβάζει τη μεταβλητή θέματος. Βολικό μέσα σε μπλοκ `map` και `grep` που ήδη δεσμεύουν το `$_`.
- **Προτεραιότητα ονομασμένου μοναδιαίου τελεστή**. Η `sqrt` είναι ονομασμένος μοναδιαίος τελεστής, οπότε δένει πιο σφιχτά από τους περισσότερους δυαδικούς τελεστές αλλά πιο χαλαρά από τον πολλαπλασιασμό και την ύψωση σε δύναμη. Το `sqrt 2 + 3` σημαίνει `sqrt(2) + 3`, όχι `sqrt(2 + 3)` - βάζετε παρενθέσεις όταν αμφιβάλλετε.
- **Το `sqrt(...)` είναι κλήση συνάρτησης**. Όπως με κάθε ονομασμένο μοναδιαίο, αν γράψετε ανοιχτή παρένθεση αμέσως μετά το όνομα μετατρέπει το υπόλοιπο σε λίστα ορισμάτων: το `sqrt(4) * 2` είναι 4 (η τετραγωνική ρίζα του 4) επί 2, δηλαδή 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)
```

Παραδείγματα

Κλασική άρρητη ρίζα. Το εκτυπωμένο δεκαδικό είναι η πλησιέστερη προσέγγιση `binary64`, όχι η ακριβής τιμή:

```
printf "%.15f\n", sqrt(2);          # 1.414213562373095
```

Πυθαγόρεια απόσταση μεταξύ δύο σημείων:

```
sub distance {
    my ($x1, $y1, $x2, $y2) = @_;
    return sqrt(($x2 - $x1) ** 2 + ($y2 - $y1) ** 2);
}

print distance(0, 0, 3, 4), "\n";   # 5
```

Το μηδέν είναι η τετραγωνική ρίζα του εαυτού του, όπως και το ένα:

```
print sqrt(0), "\n";                # 0
print sqrt(1), "\n";                # 1
```

Τα τέλεια τετράγωνα κάνουν ακριβή πλήρη κύκλο για μικρούς ακέραιους, επειδή το μαθηματικό αποτέλεσμα είναι αναπαραστάσιμο σε `binary64`:

```
print sqrt(16), "\n";               # 4
print sqrt(144), "\n";              # 12
print sqrt(1_000_000), "\n";        # 1000
```

Μεγάλα τέλεια τετράγωνα ενδέχεται **να μην** κάνουν ακριβή πλήρη κύκλο - το αποτέλεσμα είναι το πλησιέστερο αναπαραστάσιμο float, το οποίο μπορεί να είναι ελαφρώς διαφορετικό:

```
my $n = 9_007_199_254_740_996;      # 2**53 + 4
printf "%.3f\n", sqrt($n) ** 2 - $n; # non-zero in general
```

Σύγκριση με τον τελεστή ύψωσης σε δύναμη, που είναι ο γενικός τρόπος για να πάρετε οποιαδήποτε πραγματική δύναμη:

```
print 2 ** 0.5, "\n";               # same as sqrt(2)
print 27 ** (1/3), "\n";            # cube root of 27 ≈ 3
```

Χρήση του για να λάβετε μιγαδικό αποτέλεσμα από αρνητική είσοδο:

```
use Math::Complex;
my $z = sqrt(-1);                   # i
print $z, "\n";                     # "i"
print sqrt(-4), "\n";               # "2i"
```

Οριακές περιπτώσεις

- **Αρνητική είσοδος χωρίς** : επιστρέφει NaN και πυροδοτεί προειδοποίηση `Invalid operation` υπό `use warnings`. Το NaN συγκρίνεται ως άνισο με τον εαυτό του, οπότε μην το ελέγχετε ποτέ με `==`· χρησιμοποιήστε `POSIX::isnan` ή ελέγξτε `$x != $x`.

  ```
  my $r = sqrt(-1);                 # NaN
  print "bad\n" if $r != $r;        # NaN self-compare trick
  ```

- **Αρνητική είσοδος με** : όταν φορτωθεί το , η `sqrt` υπερφορτώνεται για μιγαδικά βαθμωτά και επιστρέφει την κύρια μιγαδική ρίζα. Οι πραγματικές είσοδοι εξακολουθούν να περνούν από την ενσωματωμένη.

- **Μη αριθμητική είσοδος**: εξαναγκάζεται σε αριθμό με τους συνηθισμένους κανόνες - οι αρχικοί λευκοί χαρακτήρες και ένα αριθμητικό πρόθεμα καταναλώνονται, τα υπόλοιπα αγνοούνται. Υπό `use warnings` πυροδοτείται προειδοποίηση `isn't numeric` όταν η συμβολοσειρά δεν έχει καθόλου αριθμητικό πρόθεμα. Το `undef` εξαναγκάζεται σε `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
  ```

- **Ανακρίβεια κινητής υποδιαστολής για τέλεια τετράγωνα**: για αρκετά μεγάλους ακέραιους, η μαθηματικά ακριβής ρίζα μπορεί να μην είναι αναπαραστάσιμη. Η `sqrt($n*$n)` δεν εγγυάται ότι ισούται με `$n` για μεγάλο `$n`. Αν χρειάζεστε ακριβή ακέραια τετραγωνική ρίζα, μη χρησιμοποιείτε την `sqrt` - χρησιμοποιήστε την `bsqrt` του ή ειδική υλοποίηση isqrt.

- **Ειδικές τιμές IEEE**: η `sqrt("Inf")` είναι `Inf`· η `sqrt("NaN")` είναι `NaN`· η `sqrt(-0.0)` είναι `-0.0` (αρνητικό μηδέν, που συγκρίνεται ίσο με `0`). Η `sqrt("-Inf")` είναι `NaN`.

- **Η ακέραια υπερχείλιση δεν είναι ζήτημα**: επειδή το αποτέλεσμα είναι πάντα float, η `sqrt` ποτέ δεν υπερχειλίζει για οποιαδήποτε πεπερασμένη αριθμητική είσοδο.
- **Το περιβάλλον λίστας είναι άσχετο**: η `sqrt` επιστρέφει πάντα ένα βαθμωτό. Σε περιβάλλον λίστας αυτό το βαθμωτό είναι το μόνο στοιχείο της επιστρεφόμενης λίστας.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `abs` - απόλυτη τιμή· συνδυάστε με την `sqrt` όταν θέλετε `sqrt(abs($x))` για να αποφύγετε αρνητική είσοδο
- `**` - τελεστής ύψωσης σε δύναμη· το `$x ** 0.5` είναι `sqrt`, και το `$x ** (1/$n)` γενικεύεται σε άλλες ρίζες
- `hypot` - από το `POSIX`, υπολογίζει `sqrt($x*$x + $y*$y)` χωρίς ενδιάμεση υπερχείλιση
- `-` υπερφορτώνει την `sqrt` και παρόμοιες ώστε η `sqrt(-1)` να επιστρέφει `i` αντί για `NaN`
- `,` - τετραγωνικές ρίζες αυθαίρετης ακρίβειας μέσω της `bsqrt` όταν το `binary64` δεν επαρκεί

Αριθμητικές συναρτήσεις

srand

Σπείρει τη γεννήτρια ψευδοτυχαίων αριθμών.

Η `srand` θέτει την εσωτερική κατάσταση της PRNG από την οποία αντλεί η `rand`. Καλέστε την με ρητό ακέραιο για να λάβετε αναπαραγώγιμη ροή, ή καλέστε την χωρίς όρισμα για να ξανασπείρετε από την καλύτερη πηγή εντροπίας που προσφέρει η πλατφόρμα. Από την Perl 5.14 και μετά η `srand` **επιστρέφει τον σπόρο που χρησιμοποίησε**, ώστε να μπορείτε να τον καταγράψετε και να αναπαραγάγετε την ίδια εκτέλεση αργότερα.

Σύνοψη

```
srand EXPR
srand
```

Τι επιστρέφεται

Ο σπόρος που εγκαταστάθηκε, ως ακέραιος. Για `srand($n)` η τιμή επιστροφής είναι το `$n` (μετά τη συνήθη αποκοπή σε ακέραιο). Για `srand()` χωρίς όρισμα η τιμή επιστροφής είναι ο αυτόματα επιλεγμένος σπόρος που αντλείται από την εντροπία του συστήματος - αποτυπώστε την αν ποτέ θέλετε να αναπαραγάγετε την εκτέλεση.

```
my $seed = srand();                    # auto-seed, remember what we
 ↪used
warn "seed for this run: $seed\n";
```

Πριν την Perl 5.14 η `srand` δεν επέστρεφε τίποτα χρήσιμο. Κώδικας που βασίζεται στην επιστροφή του σπόρου θα πρέπει να δηλώνει την ελάχιστη έκδοση:

```
use v5.14;                             # so srand returns the seed
```

Καθολική κατάσταση

Η `srand` μεταλλάσσει μια μοναδική PRNG καθολική σε επίπεδο διεργασίας. Κάθε επόμενη κλήση στην `rand` στην ίδια διεργασία - από οποιοδήποτε πακέτο, οποιοδήποτε άρθρωμα, οποιοδήποτε περιβάλλον χωρίς threads - διαβάζει από αυτήν τη μοναδική κατάσταση. Δεν υπάρχει RNG ανά πακέτο ή ανά εμβέλεια. Συνέπειες:

- Η κλήση `srand($k)` οπουδήποτε στο πρόγραμμα επηρεάζει *κάθε* μεταγενέστερη κλήση `rand`, συμπεριλαμβανομένων εκείνων μέσα σε αρθρώματα που δεν έχετε γράψει εσείς.
- Δύο διεργασίες που καλούν `srand($k)` με το ίδιο `$k` παράγουν πανομοιότυπες ροές `rand`. Αυτή είναι η βάση των αναπαραγώγιμων δοκιμών και ο λόγος που πρέπει να ξανασπείρετε μετά από `fork`.
- Αν η `srand` δεν κληθεί ποτέ ρητά, καλείται σιωπηρά **χωρίς όρισμα** την πρώτη φορά που καλείται η `rand`.

Παραδείγματα

Αναπαραγώγιμη περίπτωση δοκιμής με ρητό σπόρο - ίδια είσοδος, ίδια ακολουθία, κάθε εκτέλεση:

```
srand(42);
my @sample = map { rand() } 1..5;   # identical on every invocation
```

Σπορά από `time()` για τυχαιότητα με σπάνιες συγκρούσεις σε ξεχωριστές εκτελέσεις που ξεκίνησαν σε διαφορετικά δευτερόλεπτα πραγματικού χρόνου. Αυτό είναι το κλασικό μοτίβο· **δεν** είναι κρυπτογραφικού επιπέδου:

```
srand(time() ^ $$);                    # time XOR pid, old-school
 ↪spread
```

Μοτίβο αποθήκευσης-επαναφοράς σπόρου για δοκιμή που επιλέγει τυχαίο υποσύνολο των περιπτώσεών της αλλά καταγράφει αρκετά για να αναπαραγάγει οποιαδήποτε αποτυχία:

```
my $seed = srand();                    # auto-seed, capture it
eval { run_randomised_tests() };
if ($@) {
    warn "FAILED with seed=$seed - reproduce with: srand($seed)\n";
    die $@;
}
```

Ξανασπείρετε μετά από `fork` ώστε γονέας και παιδί να μη μοιράζονται ροή:

```
my $pid = fork();
die "fork: $!" unless defined $pid;
if ($pid == 0) {
    srand();                        # child gets fresh entropy
    exec_child_work();
}
```

Δύο ανεξάρτητες ροές από τον ίδιο σπόρο - χρήσιμο όταν θέλετε να κάνετε checkpoint και να συνεχίσετε, ή να συγκρίνετε δύο εκτελέσεις που απέκλιναν μετά από ένα σημείο απόφασης:

```
srand(12345);
my @first  = map { rand() } 1..10;

srand(12345);
my @second = map { rand() } 1..10;  # identical to @first
```

Οριακές περιπτώσεις

- **Η μορφή χωρίς όρισμα κάνει αυτόματη σπορά με την καλύτερη διαθέσιμη εντροπία.** Σε Linux αυτή αντλείται από το `/dev/urandom` (ή την `getrandom(2)`). Ο σπόρος είναι αρκετά ευρύς ώστε να τον αντιμετωπίζετε ως αδιαφανή, όχι ως μικρό ακέραιο.
- **Μην καλείτε την `srand()` χωρίς ορίσματα περισσότερες από μία φορά ανά διεργασία.** Η εσωτερική κατάσταση της PRNG ήδη φέρει περισσότερη εντροπία από αυτή που μπορεί να επαναφέρει οποιοσδήποτε μεμονωμένος σπόρος, οπότε η εκ νέου σπορά *χάνει* τυχαιότητα αντί να την προσθέτει. Οι δύο νόμιμοι λόγοι να την καλείτε επανειλημμένα είναι (α) με ρητό σπόρο για αναπαραγωγιμότητα, και (β) μία φορά σε κάθε παιδί μετά από `fork`.
- **Τα δεκαδικά ορίσματα αποκόπτονται σιωπηρά σε ακέραιο.** Οι `srand(42)` και `srand(42.9)` εγκαθιστούν τον ίδιο σπόρο. Πάντα περνάτε ακέραιο ώστε η πρόθεση να είναι προφανής.
- **Ίδιος σπόρος ⇒ πανομοιότυπη ροή `rand`.** Αυτό είναι όλο το νόημα για τις δοκιμές· αυτό είναι όλο το πρόβλημα για την ασφάλεια.
- **Όχι κρυπτογραφικά ασφαλές.** Οι `rand`/`srand` χρησιμοποιούν γρήγορη μη κρυπτογραφική PRNG. Μην τις χρησιμοποιείτε για να παράγετε tokens συνεδρίας, nonces, κλειδιά, salts κωδικών πρόσβασης, ή οτιδήποτε δεν πρέπει να μπορεί να προβλέψει ένας αντίπαλος. Χρησιμοποιήστε [`Crypt::URandom`][cu] ή διαβάστε απευθείας από το `/dev/urandom`.
- **Μεταβλητή περιβάλλοντος `PERL_RAND_SEED`.** Αν τεθεί σε μη αρνητικό ακέραιο κατά την εκκίνηση της διεργασίας, η `srand()` χωρίς όρισμα (συμπεριλαμβανομένης της σιωπηρής κλήσης που πυροδοτείται από την πρώτη `rand`) εγκαθιστά ντετερμινιστικό σπόρο που παράγεται από αυτή την τιμή. Η παραγωγή είναι σκόπιμα απροσδιόριστη· εγγυάται μόνο η αναπαραγωγιμότητα ίδια-τιμή-ίδιο-δυαδικό-ίδιος-κώδικας. Προορίζεται για αποσφαλμάτωση και ανάλυση επιδόσεων. Η μεταβλητή διαβάζεται **μία φορά** κατά την εκκίνηση· η μετάλλαξη του `%ENV` αργότερα δεν έχει επίδραση στην τρέχουσα διεργασία.

- **Threads** (`use threads`): κάθε κλώνος διερμηνέα λαμβάνει τη δική του κατάσταση PRNG. Η σπορά σε ένα thread δεν επηρεάζει άλλο.
- **Εμβέλεια του σπόρου**: δεν υπάρχει τρόπος να `local`-οποιήσετε την κατάσταση της PRNG. Αν μια βιβλιοθήκη που καλείτε ενδιάμεσα εκτελέσει `srand`, η ροή σας διαταράσσεται. Αποτυπώστε και επαναφέρετε χειροκίνητα αν αυτό έχει σημασία.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44. `PERL_RAND_SEED` is honoured with the same «deliberately unspecified but reproducible» contract as perl5.

Δείτε επίσης

- `rand` - ο καταναλωτής της κατάστασης που εγκαθιστά η `srand`· αντλεί την επόμενη τιμή από τη σπαρμένη ροή
- `time` - δευτερόλεπτα από την epoch· η κλασική πρόχειρη πηγή σπόρου για κώδικα μη σχετιζόμενο με ασφάλεια (`srand(time() ^ $$)`)
- [`Crypt::URandom`][cu] - κρυπτογραφικά ασφαλή τυχαία bytes από την δεξαμενή εντροπίας του OS· χρησιμοποιήστε αυτό, όχι το `rand`, για tokens και κλειδιά
- [`Math::Random::MT`][mt] - Mersenne Twister PRNG με τη δική του ανεξάρτητη κατάσταση, όταν χρειάζεστε δεύτερη ροή που δεν διαταράσσεται από κλήσεις βιβλιοθήκης προς το `rand`
- `perlrun` - τεκμηριώνει τη μεταβλητή περιβάλλοντος `PERL_RAND_SEED` και άλλα κουμπιά εκκίνησης

Filehandles, αρχεία, κατάλογοι

stat

Λαμβάνει τις πληροφορίες κατάστασης ενός αρχείου.

Η `stat` ζητά από το λειτουργικό σύστημα την εγγραφή μεταδεδομένων ενός αρχείου - μέγεθος, ιδιοκτησία, δικαιώματα, χρονικές σφραγίδες, inode - και την επιστρέφει ως λίστα 13 στοιχείων. Είναι η υποκείμενη πρωτογενής συνάρτηση πίσω από τους ελέγχους αρχείων `-X` (`-e`, `-f`, `-r`, ...), πίσω από αρθρώματα όπως το `File::stat`, και πίσω από κάθε σενάριο που χρειάζεται να εξετάσει ένα αρχείο χωρίς να το ανοίξει. Η `stat` αποαναφέρει τα συμβολικά λινκ και αναφέρει για τον στόχο· χρησιμοποιήστε `lstat` για να λάβετε το ίδιο το λινκ.

Σύνοψη

```
stat FILEHANDLE
stat EXPR
stat DIRHANDLE
stat
stat _
```

Τι επιστρέφεται

Σε περιβάλλον λίστας, μια λίστα 13 αριθμητικών πεδίων. Σε βαθμωτό περιβάλλον, λογική τιμή: αληθές σε επιτυχία, ψευδές σε αποτυχία. Σε κάθε περίπτωση, σε επιτυχία ο ενταμιευτής stat του πυρήνα αποθηκεύεται στο ειδικό filehandle `_`, το οποίο οποιαδήποτε επόμενη `stat`, `lstat` ή έλεγχος `-X` μπορεί να επαναχρησιμοποιήσει χωρίς νέα κλήση συστήματος.

Σε αποτυχία, το περιβάλλον λίστας επιστρέφει την κενή λίστα και το `$!` ορίζεται στην τιμή `errno` από την αποτυχημένη `stat(2)`.

Τα πεδία, με τη σειρά:

Δείκτη	Πεδίο	Σημασία
0	`dev`	αριθμός συσκευής του συστήματος αρχείων που περιέχει το αρχείο
1	`ino`	αριθμός inode
2	`mode`	mode αρχείου - bits τύπου **και** bits δικαιωμάτων μαζί
3	`nlink`	αριθμός σκληρών συνδέσμων προς το αρχείο
4	`uid`	αριθμητικό user ID του ιδιοκτήτη
5	`gid`	αριθμητικό group ID του ιδιοκτήτη
6	`rdev`	αναγνωριστικό συσκευής, για ειδικά αρχεία χαρακτήρα/μπλοκ
7	`size`	συνολικό μέγεθος σε bytes
8	`atime`	χρόνος τελευταίας πρόσβασης, δευτερόλεπτα από το epoch (1970-01-01 00:00 GMT)
9	`mtime`	χρόνος τελευταίας τροποποίησης, δευτερόλεπτα από το epoch
10	`ctime`	χρόνος αλλαγής inode, δευτερόλεπτα από το epoch - **όχι** χρόνος δημιουργίας
11	`blksiz`	προτιμώμενο μέγεθος μπλοκ I/O για αναγνώσεις/εγγραφές σε αυτό το αρχείο
12	`blocks`	αριθμός μπλοκ συστήματος αρχείων που έχουν πραγματικά εκχωρηθεί (συχνά 512 B το καθένα)

Το πεδίο 2 (`mode`) πακετάρει τόσο τον τύπο αρχείου όσο και τα bits δικαιωμάτων σε έναν αριθμό. Μασκάρετε με `07777` για να δείτε μόνο τα δικαιώματα· χρησιμοποιήστε τις σταθερές `S_IF*` και τους βοηθούς `S_IS*` από το `Fcntl` για να ελέγξετε τον τύπο φορητά.

```
my $mode = (stat($filename))[2];
printf "permissions are %04o\n", $mode & 07777;
```

Δεν συμπληρώνει κάθε σύστημα αρχείων κάθε πεδίο. Τα `atime`, `blksize`, και `blocks` μπορεί να επιστρέψουν `0` σε συστήματα αρχείων που δεν τα παρακολουθούν (για παράδειγμα ορισμένες προσαρτήσεις FUSE ή δικτυακά συστήματα αρχείων)· το `ctime` δεν είναι φορητός «χρόνος δημιουργίας» - παρακολουθεί την τελευταία αλλαγή του ίδιου του inode (δικαιώματα, ιδιοκτησία, πλήθος συνδέσμων), όχι το πότε δημιουργήθηκε αρχικά το αρχείο.

Καθολική κατάσταση που επηρεάζει

- `$_` - διαβάζεται όταν η `stat` καλείται χωρίς όρισμα. Η `stat` χωρίς EXPR εκτελεί stat στο όνομα αρχείου που κρατά το `$_`, **όχι** στο αποθηκευμένο handle `_`.
- `$!` - ορίζεται στο `errno` της αποτυχημένης κλήσης συστήματος `stat(2)` όταν η κλήση αποτυγχάνει.
- Το **ειδικό filehandle `_`** - μια cache ανά διερμηνέα του πιο πρόσφατου επιτυχημένου αποτελέσματος `stat`, `lstat` ή `-X`. Η μετάδοση του `_` ως όρισμα επαναχρησιμοποιεί αυτόν τον αποθηκευμένο ενταμιευτή και δεν εκδίδει κλήση συστήματος. Επιτυχημένες κλήσεις τον ανανεώνουν· αποτυχημένες αφήνουν τα προηγούμενα περιεχόμενα στη θέση τους.

Παραδείγματα

Βασικό ξεπακετάρισμα σε περιβάλλον λίστας:

```
my ($dev, $ino, $mode, $nlink, $uid, $gid, $rdev, $size,
    $atime, $mtime, $ctime, $blksize, $blocks) = stat($filename);
```

Επιλογή ενός μόνο πεδίου με τομή λίστας:

```
my $size  = (stat($filename))[7];
my $mtime = (stat($filename))[9];
```

Βαθμωτό περιβάλλον ως ανίχνευση ύπαρξης συν αναγνώσιμων μεταδεδομένων:

```
if (stat($path)) {
    my $size = (stat _)[7];          # reuse the cached buffer
    print "size = $size\n";
} else {
    warn "stat $path: $!";
}
```

Επαναχρησιμοποίηση της cache σε έλεγχο `-X` και σε `stat`:

```
if (-x $file && (($d) = stat(_)) && $d < 0) {
    print "$file is executable NFS file\n";
}
```

Πρόσβαση με ονοματισμένους accessor μέσω του `File::stat`:

```
use File::stat;
my $sb = stat($filename);
printf "file %s, size %s, perm %04o, mtime %s\n",
       $filename, $sb->size, $sb->mode & 07777,
       scalar localtime $sb->mtime;
```

Αποκωδικοποίηση του mode με `Fcntl`:

```
use Fcntl ':mode';
my $mode = (stat($filename))[2];
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
printf "permissions %04o\n", S_IMODE($mode);
print "is a directory\n" if S_ISDIR($mode);
```

Οριακές περιπτώσεις

- **Χωρίς όρισμα**: το `stat;` κάνει stat στο `$_`, **όχι** στο αποθηκευμένο handle `_`. Για να ξαναδιαβάσετε την cache πρέπει να γράψετε `stat(_)` με την κάτω παύλα ως ρητό bareword.
- **Το bareword `_` είναι ειδικό**: ονομάζει την cache του stat, όχι αρχείο. Η τοποθέτησή του σε εισαγωγικά (`stat("_")`) κάνει stat σε ένα αρχείο που λέγεται κυριολεκτικά `_` στον τρέχοντα κατάλογο.
- **Τα συμβολικά λινκ ακολουθούνται**: η `stat` αναφέρει για τον στόχο ενός συμβολικού λινκ. Για το ίδιο το λινκ, χρησιμοποιήστε `lstat`. Ένα σπασμένο symlink κάνει την `stat` να αποτύχει με `ENOENT`· η `lstat` στο ίδιο μονοπάτι επιτυγχάνει.
- **Μορφή DIRHANDLE**: η μετάδοση handle καταλόγου από `opendir` επιστρέφει το stat του ίδιου του ανοιχτού καταλόγου. Δεν υποστηρίζουν όλες οι πλατφόρμες αυτό· σε όσες δεν το υποστηρίζουν, η κλήση αποτυγχάνει και το `$!` ορίζεται.
- **Οι αριθμοί inode μπορεί να υπερχειλίσουν τους ακεραίους**: σε συστήματα αρχείων με αριθμούς inode 64 bit πέρα από το εύρος ακεραίων της Perl, το `ino` επιστρέφεται ως δεκαδική συμβολοσειρά για να διατηρηθεί η πλήρης τιμή. Συγκρίνετε αριθμούς inode με `eq`, όχι `==`, για να παραμείνετε σωστοί σε αυτά τα συστήματα αρχείων. Το `eq` λειτουργεί μια χαρά και σε αριθμητικά inode.
- **Το `ctime` δεν είναι χρόνος δημιουργίας**: είναι ο χρόνος αλλαγής inode - ενημερώνεται από `chmod`, `chown`, `rename`, αλλαγές πλήθους συνδέσμων, και τις ίδιες τις εγγραφές του αρχείου. Το Linux εκθέτει έναν πραγματικό χρόνο δημιουργίας (`btime`) μέσω της `statx(2)`, αλλά αυτός δεν αποτελεί μέρος της λίστας επιστροφής της `stat`.
- **Μηδενικά πεδία σε ορισμένα συστήματα αρχείων**: τα `atime` (σε προσαρτήσεις `noatime`), `blksize`, και `blocks` μπορεί να αναφερθούν ως `0`. Αντιμετωπίστε το `0` ως «μη διαθέσιμο», όχι ως νοηματική τιμή.
- **Το βαθμωτό περιβάλλον δεν συμπληρώνει το `_` σε αποτυχία**: μια αποτυχημένη `stat` αφήνει τον προηγούμενο αποθηκευμένο ενταμιευτή ανέπαφο. Ελέγχετε πάντα την τιμή επιστροφής πριν από επόμενο `stat(_)` ή `-X _`.
- **Παράθυρα race**: τα μεταδεδομένα που επιστρέφονται είναι στιγμιότυπο. Οτιδήποτε ενεργεί σε μονοπάτι βασισμένο στο αποτέλεσμα της `stat` (έλεγχοι δικαιωμάτων, μοτίβα TOCTOU) πρέπει να γραφτεί ώστε να ανέχεται την αλλαγή του αρχείου μεταξύ της `stat` και της επόμενης πράξης - ή καλύτερα, ανοίξτε πρώτα το αρχείο και κάντε `stat` στο filehandle.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `lstat` - όπως η `stat` αλλά **δεν** ακολουθεί συμβολικά λινκ· αναφέρει για το ίδιο το λινκ
- Τελεστές ελέγχου αρχείων `-X` (`-e`, `-f`, `-r`, `-w`, `-x`, ...) - κτισμένοι πάνω στον ίδιο ενταμιευτή stat· μοιράζονται την cache `_` με τις `stat` και `lstat`
- `chmod` - αλλάζει τα bits δικαιωμάτων που αναφέρονται στο πεδίο 2 (mode)
- `chown` - αλλάζει τα `uid` και `gid` που αναφέρονται στα πεδία 4 και 5
- `File::stat` - ονοματισμένοι accessor (`$sb->size`, `$sb->mtime`, ...) πάνω από την ίδια λίστα 13 πεδίων· το `use File::stat` υπερκαλύπτει την `stat` στην τρέχουσα εμβέλεια
- `Fcntl` - η ετικέτα `:mode` εξάγει τις σταθερές `S_I*` και τους βοηθούς `S_IS*` για την αποκωδικοποίηση του πεδίου 2

Εμβέλεια

state

Δηλώνει μια λεξιλογικά οριοθετημένη μεταβλητή της οποίας η τιμή επιμένει μεταξύ κλήσεων της περικλείουσας υπορουτίνας.

Μια μεταβλητή `state` έχει τους ίδιους κανόνες ορατότητας με μια μεταβλητή `my` - είναι ορατή μόνο μέσα στο μπλοκ που τη δηλώνει - αλλά **αρχικοποιείται ακριβώς μία φορά** ανά περικλείον closure, όχι σε κάθε είσοδο στο μπλοκ. Αυτός είναι ο ενσωματωμένος τρόπος της Perl να γράφετε επίμονες ιδιωτικές μεταβλητές: μετρητές, μνήμες memoisation, πίνακες αναζήτησης που υπολογίζονται μία φορά, οτιδήποτε διαφορετικά θα κρύβατε σε λεξιλογική σε εμβέλεια αρχείου πάνω από μια sub.

Σύνοψη

```
state $scalar
state $scalar = EXPR
state ($a, $b, undef, $c)
state @array
state %hash
state TYPE $var
state $var : ATTRS
```

Τι επιστρέφεται

Η `state` είναι δήλωση, όχι συνάρτηση. Όπως η `my`, επιστρέφει την/τις μεταβλητή/ές που δηλώνει, οπότε μπορεί να χρησιμοποιηθεί σε οποιαδήποτε θέση έκφρασης όπου θα εμφανιζόταν η δηλωμένη μεταβλητή:

```
(state $n //= 0)++;                    # declare-and-use in one␣
↪expression
print ++(state $calls), "\n";          # count calls to this line
```

Ο αρχικοποιητής στη δεξιά πλευρά του = αποτιμάται **μόνο την πρώτη φορά που ο έλεγχος περνά μέσα από τη δήλωση** σε εκείνο το συγκεκριμένο στιγμιότυπο closure. Σε κάθε επόμενη είσοδο, η δήλωση επανεκθέτει την ήδη αρχικοποιημένη μεταβλητή και ο αρχικοποιητής παραλείπεται.

Εμβέλεια επιμονής - τι σημαίνει «μία φορά»

«Μία φορά ανά περικλείον closure» είναι ο ακριβής κανόνας. Μια μεταβλητή `state` ζει όσο το CV (code value) που περιέχει τη δήλωσή της:

- Σε μια **ονομασμένη υπορουτίνα**, το CV δημιουργείται μία φορά κατά τη μεταγλώττιση. Η μεταβλητή `state` αρχικοποιείται στην πρώτη κλήση και διατηρεί την τιμή της σε κάθε επόμενη κλήση για όλη τη διάρκεια ζωής της διεργασίας.
- Σε μια **ανώνυμη υπορουτίνα / closure**, κάθε αποτίμηση `sub { ... }` παράγει ένα φρέσκο CV με τη δική του φρέσκια θέση `state`. Δύο closures φτιαγμένα από την ίδια πηγή **δεν** μοιράζονται κατάσταση.
- Σε ένα **εμφωλευμένο μπλοκ** μέσα σε υπορουτίνα, το `state` εξακολουθεί να ανήκει στο περιβάλλον CV - δεν αναδημιουργείται σε κάθε είσοδο στο εσωτερικό μπλοκ.

```
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
```

Πύλη χαρακτηριστικού και διαθεσιμότητα

Η `state` ήταν ανέκαθεν υπό feature-gate. Δύο τρόποι για να ενεργοποιηθεί:

- `use feature 'state';` - ενεργοποιεί μόνο αυτό το χαρακτηριστικό.
- `use v5.10;` (or any later version bundle, including `use v5.36` and `use v5.44`) - implicit via the version feature bundle. Every modern program already has `state` available through its `use vX.Y` line.

Χωρίς το χαρακτηριστικό σε εμβέλεια, η `state` αναλύεται ως κανονικό bareword. Για να φτάσετε την ενσωματωμένη ανεξάρτητα από αυτό, γράψτε `CORE::state`.

```
use v5.36;                             # 'state' feature is on by␣
↪default
sub hits { state $n = 0; ++$n }
```

Δηλώσεις λίστας και αρχικοποιητές

Η `state` υποστηρίζει τη μορφή λίστας με παρενθέσεις για τη δήλωση πολλών μεταβλητών ταυτόχρονα, με `undef` ως σύμβολο κράτησης θέσης:

```
state ($x, $y, undef, $z);
```

Αυτό που **δεν** υποστηρίζει ακόμη είναι η **αρχικοποίηση λίστας** μεταβλητών `state`. Το upstream το σημειώνει ρητά: η αρχικοποίηση μεταβλητών `state` σε λίστα με παρενθέσεις δεν είναι προς το παρόν εφικτή, οπότε η μορφή λίστας είναι χρήσιμη μόνο για δήλωση.

Τα συσσωματώματα - `state @array` και `state %hash` - είχαν ιστορικά τον ίδιο περιορισμό σε αρχικοποιητές (η Perl εξέπεμπε πειραματική προειδοποίηση, και σε αρκετές εκδόσεις συντακτικό σφάλμα, για `state @a = (1, 2, 3)`). Χτίστε το συσσωμάτωμα μέσα στο σώμα αντ" αυτού:

```
sub primes_below_100 {
    state @p;
    unless (@p) {
        @p = _sieve(100);           # fill on first call only
    }
    return @p;
}
```

Το ιδίωμα `unless (@p)` / `unless (%h)` παραπάνω είναι ο φορητός αντικαταστάτης ενός αρχικοποιητή συσσωματώματος.

Θέση δήλωσης και σημασιολογία της ίδιας εντολής

Όπως οι `my`, `our` και `local`, μια δήλωση `state` μπορεί να εμφανιστεί όπου επιτρέπεται έκφραση (εκτός από την παρεμβολή συμβολοσειρών). Η νέα σύνδεση τίθεται σε ισχύ για τις **επόμενες εντολές**, όχι για άλλες αναφορές στην ίδια μεταβλητή στην ίδια εντολή:

```
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)
```

Η πρώτη `$x` και η τελική `$x` στην ίδια κλήση εξακολουθούν να αναφέρονται στην `our $x`· μόνο η επόμενη εντολή βλέπει την `state $x`.

Παραδείγματα

Μετρητής επισκέψεων - το καθιερωμένο μονόγραμμο:

```
sub hits { state $n = 0; ++$n }
hits(); hits(); hits();             # returns 1, 2, 3
```

Memoisation ακριβής καθαρής συνάρτησης:

```
sub fib {
    my ($n) = @_;
    state %cache;
    return $cache{$n} //= $n < 2 ? $n : fib($n-1) + fib($n-2);
}
```

Οκνηρή εφάπαξ εγκατάσταση πίνακα μόνο-για-ανάγνωση:

```
sub months {
    state @m = qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec);
    return @m;
}
```

Εδώ ο αρχικοποιητής είναι μια απλή λίστα, που λειτουργεί επειδή η δεξιά πλευρά του `state @m = ...` αναδιπλώνεται σε σταθερή λίστα και η τρέχουσα Perl επιτρέπει αυτή τη μορφή· η πιο γενική περίπτωση υπολογισμένης λίστας εξακολουθεί να χρειάζεται το μοτίβο `unless (@m) { ... }` παραπάνω για φορητότητα.

Κατάσταση ανά closure (φρέσκια θέση ανά αποτίμηση `sub { ... }`):

```
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
```

Δήλωση και χρήση σε μία και μόνη έκφραση:

```
while (my $line = <$fh>) {
    warn "first line was: $line" if (state $first //= $line) eq
↪$line;
}
```

Οριακές περιπτώσεις

- **Αρχικοποιητές συσσωματωμάτων.** Τα `state @a = (1, 2, 3)` και `state %h = (k => 1)` έχουν μακρά ιστορία ως περιορισμένα ή πειραματικά. Όταν αμφιβάλλετε, δηλώστε το συσσωμάτωμα και γεμίστε το υπό `unless (@a) { ... }` στην πρώτη κλήση. Οι αρχικοποιητές βαθμωτών (`state $x = EXPR`) ήταν πάντοτε εντάξει.
- **Επαναχρησιμοποίηση στην ίδια εντολή.** Μια φρέσκια δήλωση `state` **δεν** ισχύει για άλλες αναφορές στο ίδιο όνομα μέσα στην ίδια εντολή· εκείνες εξακολουθούν να επιλύονται σε ό,τι ήταν σε εμβέλεια πριν τη δήλωση. Δείτε το παράδειγμα `foo($x, state $x = $x + 1, $x)` παραπάνω.
- **Προειδοποίηση επικάλυψης.** Η εκ νέου δήλωση του ίδιου ονόματος στην ίδια εμβέλεια - `state $n; state $n;` - επικαλύπτει την πρώτη. Υπό `use warnings` εκπέμπεται προειδοποίηση στην κατηγορία shadow. Σχεδόν πάντα σφάλμα.

- **Πολλαπλότητα ανώνυμης υπορουτίνας.** Κάθε αποτίμηση `sub { ... }` παράγει νέο CV με τη δική του θέση `state`. Αν χτίζετε πολλά closures από την ίδια πηγή, καθένα έχει τη δική του ανεξάρτητη `state` - αυτό είναι χαρακτηριστικό, αλλά εκπλήσσει όσους περιμένουν επιμονή σε εμβέλεια αρχείου.
- **Νήματα και forks.** Η `state` ζει στο CV, το οποίο κλωνοποιείται στο `fork` (copy-on-write με την έννοια του ΛΣ - κάθε διεργασία συνεχίζει με το δικό της αντίγραφο) και στο `threads->create` (βαθιά κλωνοποίηση). Δύο διεργασίες ή νήματα δεν μοιράζονται έναν μετρητή `state`.
- **Η `local` δεν ισχύει.** Οι μεταβλητές `state` είναι λεξιλογικές, όχι μεταβλητές πακέτου, οπότε η `local` δεν μπορεί να τις αποθηκεύσει και να τις επαναφέρει. Τυλίξτε ολόκληρη την υπορουτίνα στη δική της εμβέλεια ή χρησιμοποιήστε διαφορετικό μηχανισμό αν χρειάζεστε δυναμική εμβέλεια.
- **Προσπέλαση μέσω `CORE::state`.** Όταν γράφετε κώδικα που πρέπει να αναλύεται ακόμη και χωρίς το χαρακτηριστικό ενεργοποιημένο, προθέστε τη δεσμευμένη λέξη: `CORE::state $n = 0;`.
- **Καταστροφή.** Οι μεταβλητές `state` ζουν όσο το CV. Για ονομασμένη υπορουτίνα στο κύριο πρόγραμμα αυτό σημαίνει «μέχρι την καθολική καταστροφή». Μη βάζετε filehandle ή μεγάλο buffer σε μεταβλητή `state` περιμένοντας να απελευθερωθεί κατά την έξοδο της υπορουτίνας - δεν θα απελευθερωθεί.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `my` - συνήθης λεξιλογική· επαναρχικοποιείται σε κάθε είσοδο μπλοκ, η φυσική αντίθεση στη `state`
- `our` - λεξιλογικό ψευδώνυμο μεταβλητής πακέτου· χρησιμοποιήστε όταν οι καλούντες χρειάζεται να δουν ή να εφαρμόσουν `local` στην τιμή
- `local` - δυναμική εμβέλεια για μεταβλητές πακέτου· ορθογώνια προς τη `state` και δεν μπορεί να εφαρμοστεί σε αυτήν
- `sub` - η `state` ζει μέσα σε CV· η κατανόηση των διαφορών μεταξύ ονομασμένων και ανώνυμων υπορουτινών εξηγεί την κατάσταση ανά closure
- `use` - η `use v5.10` και νεότερες ενεργοποιούν το χαρακτηριστικό `state` ως μέρος του πακέτου της έκδοσης
- `undef` - νόμιμο σύμβολο κράτησης θέσης μέσα σε δηλώσεις λίστας `state (...)`

Κανονικές εκφράσεις και αντιστοίχιση μοτίβων

study

Ένα no-op που διατηρείται για συμβατότητα πηγαίου κώδικα με παλαιότερο κώδικα Perl.

Η `study` δέχεται ένα βαθμωτό όρισμα (ή λειτουργεί στο `$_` όταν καλείται χωρίς όρισμα) και δεν κάνει τίποτα με αυτό. Η κλήση είναι έγκυρη, το όρισμα αποτιμάται, και ο έλεγχος επιστρέφει αμέσως. Δεν χτίζεται κανένα ευρετήριο, δεν τροποποιείται καμία κατάσταση

της μηχανής regex, καμία μεταγενέστερη αντιστοίχιση δεν εκτελείται γρηγορότερα ή πιο αργά εξαιτίας της.

Το όνομα επιβιώνει επειδή δεκαετίες κώδικα Perl - βιβλία, FAQ, διανομές CPAN, εσωτερικά scripts - πασπαλίζουν `study` μπροστά από μακριές συμβολοσειρές που πρόκειται να αντιστοιχιστούν με πολλά μοτίβα. Αυτά τα προγράμματα εξακολουθούν να μεταγλωττίζονται και εξακολουθούν να τρέχουν. Απλώς δεν λαμβάνουν πλέον τη βελτιστοποίηση που κάποτε υποσχόταν η δεσμευμένη λέξη.

Σύνοψη

```
study SCALAR
study
```

Τι επιστρέφεται

Πάντα 1. Η τιμή επιστροφής δεν τεκμηριώνεται ως χρήσιμη και κανένας ιδιωματικός κώδικας δεν την επιθεωρεί. Αντιμετωπίστε τη `study` ως μη επιστρέφουσα κάτι άξιο αποτύπωσης.

Ιστορικό

Μέχρι το Perl 5.14 η `study` έχτιζε ένα αντεστραμμένο ευρετήριο των bytes στο `SCALAR`, και μετά προκαταλάμβανε τη μηχανή regex ώστε να αγκυρώνει κάθε επόμενη αντιστοίχιση στον σπανιότερο χαρακτήρα του μοτίβου (η σπανιότητα εκτιμώταν από έναν στατικό πίνακα συχνοτήτων ψημένο μέσα στον διερμηνευτή). Ο επιδιωκόμενος φόρτος εργασίας ήταν «μία μακριά συμβολοσειρά, πολλά `m//` εναντίον της»: το ευρετήριο αποσβέστηκε στις αντιστοιχίσεις.

Η βελτιστοποίηση αφαιρέθηκε στο Perl 5.16. Αλληλεπιδρούσε άσχημα με το Unicode, το casefold `/i`, και τη μετά-την-5.10 μηχανή regex, και το κόστος συντήρησης είχε από καιρό ξεπεράσει τον στενό φόρτο εργασίας που βοηθούσε. Η δεσμευμένη λέξη διατηρήθηκε ως no-op ώστε ο υπάρχων πηγαίος κώδικας να συνεχίσει να μεταγλωττίζεται.

Παραδείγματα

Παλαιό ιδίωμα - εξακολουθεί να μεταγλωττίζεται, δεν έχει πια κανένα αποτέλεσμα:

```
study $big_text;
while ($big_text =~ /$pattern/g) {
    # ...
}
```

Χωρίς όρισμα - εφαρμόζεται στο `$_`, επίσης no-op:

```
for (@lines) {
    study;
    print if /error/;
}
```

Καθοδήγηση αντικατάστασης: αν ο αρχικός κώδικας στηριζόταν στη `study` για ταχύτητα, η σύγχρονη απάντηση είναι να μεταγλωττίσετε το μοτίβο μία φορά με `qr` και να το επαναχρησιμοποιήσετε, ή να αναδιαρθρώσετε τον κώδικα ώστε η επαναλαμβανόμενη εργασία να είναι από την πλευρά του μοτίβου, όχι από την πλευρά της συμβολοσειράς:

```
my $re = qr/$pattern/;
for my $line (@lines) {
    print $line if $line =~ $re;
}
```

Οριακές περιπτώσεις

- **Κανένα αποτέλεσμα στην επίδοση των regex.** Η αφαίρεση κάθε κλήσης `study` από ένα πρόγραμμα δεν αλλάζει τίποτα παρατηρήσιμο, ούτε τους χρόνους πέρα από τον θόρυβο της μέτρησης.
- **Το όρισμα αποτιμάται ούτως ή άλλως.** Το `study some_expensive_call()` εκτελεί την `some_expensive_call()` για τις παρενέργειες και την τιμή επιστροφής της, και μετά απορρίπτει το αποτέλεσμα. Αυτό σχεδόν ποτέ δεν είναι αυτό που σκόπευε ο συγγραφέας.
- **Καμία προειδοποίηση κατά τη χρήση.** Η `study` δεν προειδοποιεί υπό `use warnings` και δεν είναι παρωχημένη, οπότε ένα linter είναι ο μοναδικός τρόπος να σημανθούν παλιές εμφανίσεις σε μια βάση κώδικα.
- **Δεν είναι pragma, δεν είναι hint.** Η `study` είναι κανονική ενσωματωμένη έκφραση. Δεν διατηρείται σε όλες τις δηλώσεις και δεν υπάρχει κατάσταση «studied» προσκολλημένη στο βαθμωτό.
- **Tainting και magic μένουν ανέγγιχτα.** Το `study $tainted` δεν ξεπλένει taint, δεν πυροδοτεί FETCH σε tied βαθμωτό περισσότερο απ” ό,τι μια απλή ανάγνωση, και δεν σημειώνει το βαθμωτό με κανέναν τρόπο.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44. `study` is a no-op in both.

Δείτε επίσης

- `qr` - μεταγλωττίζει ένα μοτίβο μία φορά και επαναχρησιμοποιεί τη μεταγλωττισμένη μορφή· αυτή είναι η σύγχρονη απάντηση στο «κάντε πολλές αντιστοιχίσεις στο ίδιο κείμενο γρηγορότερα»
- `m` - ο τελεστής αντιστοίχισης του οποίου την επίδοση επηρέαζε κάποτε η `study`
- `index` - αναζήτηση σταθερής συμβολοσειράς, γρηγορότερη από regex όταν το μοτίβο είναι κυριολεκτική υπο-συμβολοσειρά
- `pos` - πού άφησε η τελευταία αντιστοίχιση `/g`, σχετικό με τον φόρτο εργασίας «μία μακριά συμβολοσειρά, πολλές αντιστοιχίσεις» που κάποτε στόχευε η `study`

Ροή ελέγχου

sub

Δηλώνει ή ορίζει μια υπορουτίνα.

Η sub είναι η δεσμευμένη λέξη που εισάγει μια υπορουτίνα. Με NAME και BLOCK ορίζει μια κατονομασμένη υπορουτίνα στο τρέχον πακέτο. Χωρίς BLOCK είναι δήλωση προς τα εμπρός. Χωρίς NAME είναι ανώνυμη έκφραση sub που αποτιμάται σε αναφορά κώδικα - το sub { ... } είναι η μόνη μορφή της sub που είναι πράγματι έκφραση· οι κατονομασμένες μορφές είναι δηλώσεις και δεν επιστρέφουν τίποτα χρήσιμο.

Σύνοψη

```
sub NAME BLOCK                          # named definition
sub NAME (PROTO) BLOCK                  # with prototype
sub NAME : ATTRS BLOCK                  # with attributes
sub NAME (PROTO) : ATTRS BLOCK          # prototype + attributes

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
```

Τι επιστρέφεται

Οι **κατονομασμένες** μορφές (sub NAME BLOCK, sub NAME (PROTO) BLOCK, κ.λπ.) είναι δηλώσεις. Εγκαθιστούν την υπορουτίνα στον πίνακα συμβόλων του τρέχοντος πακέτου κατά τη μεταγλώττιση και δεν συνεισφέρουν τίποτα στην περιβάλλουσα έκφραση. Η γραφή my $x = sub foo { 1 }; είναι συνηθισμένο λάθος - αυτό είναι δύο δηλώσεις κολλημένες μαζί, και η $x αφήνεται απροσδιόριστη.

Η **ανώνυμη** μορφή sub BLOCK είναι έκφραση. Αποτιμάται - τη στιγμή που η έκφραση προσεγγίζεται σε χρόνο εκτέλεσης, όχι κατά τη μεταγλώττιση - σε μια αναφορά κώδικα που κλείνει πάνω από τις λεκτικές μεταβλητές στην εμβέλεια. Η ίδια έκφραση sub { ... } αποτιμώμενη δύο φορές παράγει δύο διακριτές αναφορές κώδικα που δεν μοιράζονται τίποτα:

```
my @refs = map { sub { $_ } } 1..3;     # three independent closures
```

Οι δηλώσεις προς τα εμπρός (sub NAME; χωρίς μπλοκ) δεν κάνουν τίποτα σε χρόνο εκτέλεσης και δεν επιστρέφουν τίποτα. Σκοπός τους είναι να αναγγείλουν το όνομα στον αναλυτή ώστε επόμενες κλήσεις του να μπορούν να αναλυθούν χωρίς παρενθέσεις.

Καθολική κατάσταση που επηρεάζει

- Το τρέχον πακέτο (ορίζεται από την `package`) καθορίζει το πλήρως κατονομασμένο όνομα μιας κατονομασμένης υπορουτίνας. Το `sub foo { }` μέσα σε `package Acme;` εγκαθιστά την `Acme::foo`, όχι την `main::foo`.
- Η `@_` μέσα στο σώμα κρατά τη λίστα ορισμάτων (ως ψευδώνυμο στα βαθμωτά του καλούντος, εκτός όταν ισχύει υπογραφή - δείτε *Υπογραφές* παρακάτω). Η ανάθεση στην `@_` ως σύνολο σπάει την ψευδωνυμοποίηση για το υπόλοιπο της κλήσης.
- Τα στοιχεία `$_[N]` είναι ψευδώνυμα προς τα βαθμωτά του καλούντος. Η τροποποίηση της `$_[0]` τροποποιεί τη μεταβλητή του καλούντος· η διαβίβαση κυριολεκτικής τιμής και η μετέπειτα τροποποίηση της `$_[0]` είναι μοιραίο σφάλμα.
- Η `wantarray` μέσα στο σώμα αντικατοπτρίζει το περιβάλλον στο οποίο κλήθηκε η υπορουτίνα.

Κατονομασμένος ορισμός

```
sub greet {
    my ($name) = @_;
    return "Hello, $name";
}

print greet("world"), "\n";            # Hello, world
```

Το σώμα μεταγλωττίζεται κατά τη μεταγλώττιση. Το όνομα είναι ορατό από εκείνο το σημείο του αρχείου και μετά· οι μπροστινές αναφορές εντός του ίδιου πακέτου λειτουργούν διότι η Perl αναβάλλει τις κλήσεις τύπου μεθόδου και τις κλήσεις bareword.

Ο επανορισμός κατονομασμένης υπορουτίνας σε χρόνο εκτέλεσης (`eval "sub foo { ... }"` ή ένα δεύτερο `sub foo { }` στο ίδιο πακέτο) αντικαθιστά τον προηγούμενο ορισμό και εκπέμπει προειδοποίηση `Subroutine foo redefined` υπό `use warnings`. Καταστείλετέ την με `no warnings 'redefine'` για την εμβέλεια του επανορισμού.

Ανώνυμη sub και closures

```
sub make_counter {
    my $n = 0;
    return sub { ++$n };
}

my $c = make_counter();
print $c->(), $c->(), $c->(), "\n";    # 123
```

Η εσωτερική `sub` αποτυπώνει την `$n` από την περικλείουσα εμβέλειά της. Κάθε κλήση της `make_counter` δημιουργεί μια καινούργια `$n` και ένα καινούργιο closure πάνω σε αυτή.

```
my @doublers = map { my $k = $_; sub { $k * 2 } } 1..3;
print $doublers[2]->(), "\n";          # 6
```

Παρατηρήστε το ενδιάμεσο `my $k = $_` - τα closures αποτυπώνουν μεταβλητές, όχι τιμές, και η `$_` είναι κοινή στις επαναλήψεις της `map`. Χωρίς το αντίγραφο, κάθε closure θα έβλεπε την τελευταία τιμή της `$_`.

Υπογραφές

Με `use feature 'signatures'` (ενεργό προεπιλεγμένα υπό `use v5.36` και μεταγενέστερα), η λίστα παραμέτρων εμφανίζεται ανάμεσα στο όνομα και το μπλοκ, και τα ορίσματα δένονται αυτόματα σε λεκτικές μεταβλητές:

```
use feature 'signatures';

sub add ($x, $y) { $x + $y }

sub greet ($name, $greeting = "Hello") {
    return "$greeting, $name";
}

sub log_all ($level, @msgs) {
    print "[$level] $_\n" for @msgs;
}
```

Μορφές υπογραφών:

- `$x` - απαιτούμενη παράμετρος κατά θέση.
- `$x = EXPR` - προαιρετική παράμετρος κατά θέση με προεπιλογή· η `EXPR` αποτιμάται σε κάθε κλήση όταν το όρισμα λείπει.
- `$x = undef` - προαιρετική, με προεπιλογή `undef` (χρήσιμη για να τεκμηριωθεί ότι μια θυρίδα είναι nullable αντί να παραλείπεται).
- `@rest` / `%rest` - slurpy, καταναλώνει όλα τα υπόλοιπα ορίσματα.
- `$` / `@` / `%` - ακατονόμαστο placeholder, επιβεβαιώνει την πληθικότητα χωρίς να δεσμεύει μεταβλητή.

Κλήση με λανθασμένη πληθικότητα κράζει: `Too many arguments for subroutine` ή `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 5.44. Use the signature variables instead.

Πρωτότυπα

```
sub mypush (\@@) {
    my $aref = shift;
    push @$aref, @_;
}

my @a = (1, 2);
mypush @a, 3, 4;                    # @a is now (1, 2, 3, 4)
```

Τα πρωτότυπα είναι **υπόδειξη προς τον αναλυτή κατά τη μεταγλώττιση**, όχι έλεγχος τύπου. Αλλάζουν τον τρόπο που αναλύονται τα σημεία κλήσης: το `\@` στο παραπάνω πρωτότυπο εξαναγκάζει το πρώτο όρισμα να είναι πίνακας και διαβιβάζει αναφορά σε αυτόν. Δείτε την `prototype` για να επιθεωρήσετε το πρωτότυπο μιας υπορουτίνας.

Τα πρωτότυπα δεν κάνουν τίποτα όταν η υπορουτίνα καλείται μέσω `&name(...)` ή μέσω αναφοράς κώδικα. Αν θέλετε υπογραφή, χρησιμοποιήστε `use feature 'signatures'`· τα πρωτότυπα και οι υπογραφές μπορούν να συνυπάρχουν με το γνώρισμα `:prototype(...)`:

```
use feature 'signatures';
sub each_pair :prototype(\@) ($aref) { ... }
```

Γνωρίσματα

Τα γνωρίσματα εμφανίζονται μετά από `:` και τροποποιούν τον τρόπο που η υπορουτίνα μεταγλωττίζεται ή καταχωρείται:

```
sub critical :lvalue { $state }        # usable on the left of =
sub noop     :method { }               # marked as a method (for
↪some tools)
```

Συνηθισμένα βασικά γνωρίσματα: `lvalue`, `method`, `prototype(...)`, `const`. Αρθρώματα όπως το `Attribute::Handlers` επιτρέπουν σε ένα πακέτο να ορίσει τα δικά του. Άγνωστα γνωρίσματα αποτελούν σφάλμα κατά τη μεταγλώττιση.

Δήλωση προς τα εμπρός

```
sub later;                             # announce the name
plan later(3);                         # now parses without parens
sub later { ... }                      # defined later in the file
```

Μια δήλωση προς τα εμπρός είναι χρήσιμη μόνο για να ενημερώσει τον αναλυτή ότι το `later` είναι υπορουτίνα, ώστε το `later ARG` να αναλυθεί ως κλήση και όχι ως κατασκευή έμμεσου αντικειμένου ή έκφραση με κόμμα. Δεν απαιτείται αν καλείτε με παρενθέσεις.

Παραδείγματα

Κατονομασμένη υπορουτίνα, ρητή αποσυσκευασία ορισμάτων:

```
sub area {
    my ($w, $h) = @_;
    return $w * $h;
}
print area(3, 4), "\n";                # 12
```

Ανώνυμη υπορουτίνα που διαβιβάζεται σε συνάρτηση ανώτερης τάξης:

```
my @lengths = map { sub { length shift } }->(@_),
              ("alpha", "beta", "gamma");   # not quite - see below
```

Πιο χρήσιμο: διαβίβαση sub απευθείας:

```
use List::Util qw(first);
my $first_even = first { $_ % 2 == 0 } 1, 3, 4, 7;   # 4
```

Υπονοούμενη επιστροφή - η τιμή της τελευταίας έκφρασης επιστρέφεται:

```
sub double { $_[0] * 2 }                # no return needed
print double(21), "\n";                 # 42
```

Επιστροφή ευαίσθητη στο περιβάλλον:

```
sub names {
    my @n = qw(alice bob carol);
    return wantarray ? @n : scalar @n;
}

my @all = names();                      # ("alice","bob","carol")
my $n   = names();                      # 3
```

Τροποποίηση ορισμάτων καλούντος μέσω ψευδωνυμοποίησης της `@_`:

```
sub upcase_in {
    for (@_) { $_ = uc }
}
my $s = "hello";
upcase_in($s);
print $s, "\n";                         # HELLO
```

Αναδρομή με την `__SUB__` - αναφορά στην τρέχουσα υπορουτίνα χωρίς να την ονομάσετε:

```
use feature 'current_sub';
my $fact = sub {
    my $n = shift;
    $n < 2 ? 1 : $n * __SUB__->($n - 1);
};
print $fact->(5), "\n";                 # 120
```

Οριακές περιπτώσεις

- **Ο κατονομασμένος ορισμός sub είναι δήλωση, όχι έκφραση.** Δεν συνεισφέρει τίποτα στην περικλείουσα έκφρασή του. Το `my $x = sub foo {};` αναλύεται ως δύο δηλώσεις· η `$x` καταλήγει `undef`.
- **Οι μορφές `sub NAME BLOCK` είναι κατά τη μεταγλώττιση.** Το σώμα του μπλοκ αναλύεται και μεταγλωττίζεται όταν μεταγλωττίζεται το περικλείον αρχείο ή `eval`. Οι παρενέργειες μέσα στο μπλοκ τρέχουν μόνο όταν η υπορουτίνα *καλείται*, όχι όταν ορίζεται. Αντιθέτως, η `sub BLOCK` (ανώνυμη) αποτιμάται σε χρόνο εκτέλεσης κάθε φορά που η έκφραση προσεγγίζεται, παράγοντας ένα νέο closure κάθε φορά.
- **Οι εμφωλευμένες κατονομασμένες υπορουτίνες δεν κλείνουν πάνω σε τίποτα χρήσιμο.** Μια `sub` ορισμένη μέσα σε άλλη `sub` εξακολουθεί να εγκαθίσταται στο πακέτο κατά τη μεταγλώττιση και **δεν** αποτυπώνει λεκτικές μεταβλητές από την

περικλείουσα κλήση. Κάθε κλήση της εξωτερικής υπορουτίνας βλέπει την ίδια εσωτερική υπορουτίνα, στρεβλωμένη γύρω από τις λεκτικές της *πρώτης* κλήσης. Χρησιμοποιήστε ανώνυμη υπορουτίνα για closures.

- **Η τελική `return` είναι προαιρετική αλλά η ρητή είναι πιο καθαρή.** Χωρίς `return`, επιστρέφεται η τιμή της τελευταίας *έκφρασης*. Αν η τελευταία δήλωση είναι βρόχος ή μπλοκ, η τιμή επιστροφής είναι απροσδιόριστη - μη βασίζεστε σε αυτή.
- **Η ανάθεση στην `@_` συνολικά σπάει την ψευδωνυμοποίηση.** Το `@_ = @_` ή `@_ = (1, 2)` αντικαθιστά τον πίνακα και αργότερα `$_[0] = X` δεν φτάνει πια στον καλούντα.
- **Η διαβίβαση κυριολεκτικής τιμής και η μετέπειτα τροποποίηση της `$_[0]` είναι μοιραία.** Το `upcase_in("x")` πεθαίνει με `Modification of a read-only value attempted` διότι η `$_[0]` είναι ψευδώνυμο της κυριολεκτικής τιμής.
- **Ο επανορισμός υπορουτίνας** εκπέμπει `Subroutine X redefined` υπό `use warnings`. Το τύλιγμα ενός επανορισμού απαιτεί `no warnings 'redefine'` για την εμβέλεια:

```
no warnings 'redefine';
*Foo::bar = sub { ... };
```

- **`sub` μέσα σε μπλοκ `BEGIN`** ορίζεται κατά τη μεταγλώττιση της περιβάλλουσας μονάδας, όπως κάθε δήλωση `sub`· το τύλιγμα `BEGIN` δεν αλλάζει τον χρονισμό της ίδιας της δήλωσης.
- **Παγίδα πρωτοτύπου μετά από κενό.** Το `sub foo ($$)` με `use feature 'signatures'` αναλύεται ως υπογραφή, όχι ως πρωτότυπο. Για να επισυνάψετε πρωτότυπο υπό υπογραφές, χρησιμοποιήστε `sub foo :prototype($$) ($x, $y) { ... }`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `return` - ρητή επιστροφή από υπορουτίνα· χωρίς αυτή επιστρέφεται η τιμή της τελευταίας έκφρασης
- `wantarray` - επιθεωρεί το περιβάλλον κλήσης (λίστα / βαθμωτό / κενό) από μέσα από το σώμα
- `caller` - πληροφορίες για το ποιος κάλεσε αυτή την υπορουτίνα και από πού
- `prototype` - διαβάστε πίσω το πρωτότυπο μιας κατονομασμένης ή ανώνυμης υπορουτίνας
- `__SUB__` - αναφορά στην τρέχουσα υπορουτίνα χωρίς να την ονομάσετε· το εργαλείο για ανώνυμη αναδρομή
- `my` - ο συνηθισμένος τρόπος για αποσυσκευασία της `@_` σε κατονομασμένες λεκτικές μεταβλητές όταν δεν χρησιμοποιούνται υπογραφές
- `method` - δήλωση μεθόδου με σύνταξη κλάσης· το ισοδύναμο της `sub` υπό `use feature 'class'` για μεθόδους

Βαθμωτά και συμβολοσειρές

substr

Εξαγωγή, αντικατάσταση ή ψευδωνυμοποίηση μιας συνεχόμενης φέτας μιας συμβολοσειράς.

Η `substr` επιλέγει μια σειρά χαρακτήρων από την `EXPR` ξεκινώντας στο `OFFSET` και εκτείνεται σε `LENGTH` χαρακτήρες, και επιστρέφει αυτή τη φέτα. Η ίδια κλήση μπορεί επίσης να *τροποποιήσει* την αρχική συμβολοσειρά με τρεις διακριτούς τρόπους: ως lvalue αριστερά του `=`, μέσω της μορφής τεσσάρων ορισμάτων `REPLACEMENT`, ή μέσω ψευδωνυμοποιημένης lvalue αποθηκευμένης σε μεταβλητή που παρακολουθεί τη φέτα σε επόμενες εγγραφές. Ο χαρακτήρας μηδέν είναι ο πρώτος χαρακτήρας. Αρνητικό `OFFSET` μετρά από το τέλος· αρνητικό `LENGTH` σταματά τόσους χαρακτήρες πριν το τέλος. Αν παραλειφθεί το `LENGTH`, η φέτα τρέχει έως το τέλος της συμβολοσειράς.

Σύνοψη

```
substr EXPR, OFFSET
substr EXPR, OFFSET, LENGTH
substr EXPR, OFFSET, LENGTH, REPLACEMENT
substr(EXPR, OFFSET, LENGTH) = NEWVALUE       # lvalue form
```

Τι επιστρέφεται

Σε θέση rvalue, ένα νέο βαθμωτό που περιέχει την εξαγμένη φέτα. Σε θέση lvalue, ένα μαγικό βαθμωτό που, όταν του εκχωρείται τιμή, εμβολιάζει τη νέα τιμή στην `EXPR` στην καταγεγραμμένη θέση - το επιστρεφόμενο βαθμωτό **δεν** κρατά αντίγραφο των αρχικών χαρακτήρων. Η μορφή τεσσάρων ορισμάτων εκχωρεί το `REPLACEMENT` στην `EXPR` επιτόπου και επιστρέφει την υποσυμβολοσειρά που υπήρχε πριν, οπότε η εξαγωγή και η αντικατάσταση συμβαίνουν σε μία κλήση.

Αν το ζητούμενο εύρος βρίσκεται μερικώς εκτός της συμβολοσειράς, επιστρέφει μόνο το εντός-ορίων τμήμα. Αν βρίσκεται εξ ολοκλήρου εκτός της συμβολοσειράς, οι μορφές rvalue επιστρέφουν `undef` και προειδοποιούν· οι μορφές lvalue εγείρουν εξαίρεση.

Καθολική κατάσταση που επηρεάζει

Καμία. Η `substr` είναι καθαρή ως προς τις καθολικές του διερμηνέα. Δεν διαβάζει ούτε γράφει στην `$_`, `$/`, `$\`, `$!`, ή σε οποιαδήποτε άλλη ειδική μεταβλητή. Οι προειδοποιήσεις για πρόσβαση εκτός εύρους διέπονται από τη συνήθη εμβέλεια `use warnings` / `$^W`.

Θέσεις και αρνητικοί δείκτες

Το `OFFSET` μετριέται από την αρχή της συμβολοσειράς όταν είναι μη αρνητικό, και από το τέλος όταν είναι αρνητικό: το `-1` είναι η θέση του τελευταίου χαρακτήρα, το `-2` του προτελευταίου, και ούτω καθεξής.

Το `LENGTH`, όταν είναι μη αρνητικό, είναι ο αριθμός των χαρακτήρων προς συμπερίληψη. Όταν είναι αρνητικό, είναι δεξιά άγκυρα: η φέτα σταματά τόσους χαρακτήρες πριν το τέλος της συμβολοσειράς. Η παράλειψη του `LENGTH` είναι το ίδιο με «έως το τέλος».

```
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"
```

Χρήση της substr ως lvalue

Αν η ίδια η EXPR είναι lvalue, η substr(EXPR, OFFSET, LENGTH) μπορεί να εμφανιστεί στην αριστερή πλευρά μιας εκχώρησης. Η δεξιά τιμή αντικαθιστά τη φέτα. Η αντικατάσταση δεν χρειάζεται να ταιριάζει με το LENGTH: μια μικρότερη αντικατάσταση μικραίνει τη συμβολοσειρά, μια μεγαλύτερη την επεκτείνει.

```
my $name = 'fred';
substr($name, 0, 2) = 'SH';         # $name is now "SHed"
substr($name, 1, 2) = 'ayfiel';     # $name is now "Shayfield"
```

Η εκχώρηση πέρα από το τρέχον τέλος της συμβολοσειράς επιτρέπεται, μέχρι μία θέση πέραν του τέλους (προσθήκη-στο-τέλος). Πέρα από αυτό η εκχώρηση εγείρει εξαίρεση.

```
my $name = 'fred';
substr($name, 4) = 'dy';            # $name is now "freddy"
substr($name, 7) = 'gap';           # exception
```

Για να διατηρήσετε σταθερό το μήκος της συμβολοσειράς, γεμίστε ή αποκόψτε ρητά την αντικατάσταση - για παράδειγμα με sprintf "%-*s", $len, $val.

Η μορφή αντικατάστασης τεσσάρων ορισμάτων

Η substr EXPR, OFFSET, LENGTH, REPLACEMENT αντικαθιστά τη φέτα επιτόπου και επιστρέφει την υποσυμβολοσειρά που *υπήρχε* εκεί. Αυτή συνήθως προτιμάται από τη μορφή lvalue: η λειτουργία είναι μία έκφραση, και η παλιά τιμή είναι διαθέσιμη ως τιμή επιστροφής.

```
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"
```

Αυτό αντικατοπτρίζει το ιδίωμα εξαγωγής-και-αντικατάστασης της splice σε πίνακες.

Το τέχνασμα της ψευδωνυμοποιημένης lvalue

Το βαθμωτό που επιστρέφει η μορφή τριών ορισμάτων **δεν** είναι αντίγραφο των χαρακτήρων - είναι ένα μαγικό αντικείμενο που θυμάται σε ποια περιοχή της αρχικής συμβολοσειράς αναφέρεται. Η αποθήκευση αυτής της lvalue σε μεταβλητή (μέσω for, ή \ που λαμβάνεται μέσω accessor lvalue) επιτρέπει σε επαναλαμβανόμενες εγγραφές να εμβολιάζουν στην ίδια θέση της αρχικής συμβολοσειράς, ακόμη και καθώς η συμβολοσειρά αυτή αλλάζει μέγεθος:

```
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
}
```

Με αρνητικό OFFSET, το ψευδώνυμο παρακολουθεί το τέλος της συμβολοσειράς:

```
my $x = '1234';
for (substr($x, -3, 2)) {
    $_ = 'a';    print $x, "\n";    # 1a4
    $x = 'abcdefg';
    print $_, "\n";                 # f
}
```

Αυτό είναι χρήσιμο όταν επανεγγράφετε επανειλημμένα την ίδια θέση ενός buffer (π.χ. πεδία επικεφαλίδας σε εγγραφή σταθερής διάταξης) - αλλά είναι επίσης κλασική πηγή εκπλήξεων. Προτιμήστε τη μορφή τεσσάρων ορισμάτων εκτός αν θέλετε συγκεκριμένα την ψευδωνυμοποίηση.

Παραδείγματα

Εξαγωγή σταθερού πεδίου από εγγραφή:

```
my $line = "2026-04-22  richard.jelinek\@example.com";
my $date = substr $line, 0, 10;          # "2026-04-22"
```

Αφαίρεση αρχικού προθέματος γνωστού μήκους:

```
my $msg = "ERROR: disk full";
my $rest = substr $msg, length("ERROR: ");   # "disk full"
```

Αντικατάσταση σταθερού πεδίου επιτόπου με τη μορφή τεσσάρων ορισμάτων, διατηρώντας την παλιά τιμή:

```
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      "
```

Διάσχιση συμβολοσειράς σε παράθυρα τριών χαρακτήρων χωρίς αντιγραφή ολόκληρης της συμβολοσειράς κάθε φορά:

```
my $s = "ABCDEFGH";
for (my $i = 0; $i + 3 <= length $s; $i++) {
    print substr($s, $i, 3), "\n";  # ABC BCD CDE DEF ...
}
```

Επανεγγραφή του τελευταίου byte ενός buffer επανειλημμένα μέσω ψευδωνυμοποιημένης 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"
```

Οριακές περιπτώσεις

- **Rvalue εκτός εύρους, μερικώς εντός ορίων**: επιστρέφει μόνο το εντός-ορίων τμήμα, καμία προειδοποίηση.

```
my $s = "abcd";
substr($s, 2, 100);                 # "cd"
```

- **Rvalue εκτός εύρους, εξ ολοκλήρου πέραν του τέλους**: επιστρέφει `undef` και προειδοποιεί υπό `use warnings`. Η περίπτωση κενής ουράς (ακριβώς στο τέλος) επιστρέφει `""` χωρίς προειδοποίηση:

```
my $name = 'fred';
my $null = substr $name, 4, 2;      # "" (no warning)
my $oops = substr $name, 7;         # undef, with warning
```

- **Lvalue εκτός εύρους**: εγείρει εξαίρεση. Η `substr` αρνείται να επεκτείνει σιωπηρά τη συμβολοσειρά με κενό απροσδιόριστου περιεχομένου.
- **Αρνητικό `OFFSET` μεγαλύτερο από τη συμβολοσειρά**: μετρά ως «πριν την αρχή», και περικόπτεται στη θέση `0` στις τρέχουσες Perls· υπό `use warnings` αυτό εκπέμπει προειδοποίηση «substr outside of string» σε ορισμένες εκδόσεις. Προτιμήστε να περικόπτετε ρητά.
- **Αρνητικό `LENGTH` που υπερβαίνει την υπολειπόμενη ουρά**: παράγει κενή συμβολοσειρά, επειδή η δεξιά άγκυρα πέφτει στην ή πριν την αριστερή άγκυρα.

```
my $s = "abcde";
substr($s, 1, -10);                 # ""
```

- **Unicode**: η `substr` λειτουργεί σε χαρακτήρες, όχι bytes. Σε συμβολοσειρά με τη σημαία UTF-8 ενεργοποιημένη, οι θέσεις μετρούν codepoints· σε συμβολοσειρά bytes, οι θέσεις μετρούν bytes. Η ανάμειξη των δύο αντιμετωπίζοντας μια αποκωδικοποιημένη συμβολοσειρά ως buffer bytes παράγει λάθος offsets - αποκωδικοποιήστε πρώτα, και έπειτα ευρετηριάστε.
- **Δεσμευμένα ή μαγικά βαθμωτά**: η lvalue `substr` τιμά τη μαγεία `STORE`-time στον στόχο, οπότε οι δεσμευμένες μεταβλητές βλέπουν ολόκληρη τη μεταλλαγμένη συμβολοσειρά, όχι τη φέτα.
- **Ψευδωνυμοποιημένη lvalue που διατηρείται πέρα από τη ζωή του στόχου της**: αν η αρχική μεταβλητή εξέλθει της εμβέλειας ενώ το ψευδώνυμο εξακολουθεί να ζει, μεταγενέστερες εγγραφές μέσω του ψευδωνύμου ενεργούν σε ελευθερωμένη συμβολοσειρά· κρατήστε τη ζωή του ψευδωνύμου εντός εκείνης του στόχου.
- **Γωνία πριν την 5.10**: επαναλαμβανόμενες εγγραφές μέσω μίας lvalue είχαν απροσδιόριστη συμπεριφορά σε παλαιότερες perls, όπως και τα αρνητικά offsets

πριν την 5.16. Η σύγχρονη Perl (και η pperl) ακολουθούν τη σημασιολογία που τεκμηριώνεται σε αυτή τη σελίδα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `index` - εύρεση της θέσης μιας υποσυμβολοσειράς, για να τροφοδοτηθεί στην `substr` ως `OFFSET`
- `rindex` - ίδιο, με αναζήτηση από τα δεξιά
- `length` - το ανώτατο όριο σε οποιοδήποτε ουσιαστικό `OFFSET` ή `LENGTH` περνάτε στην `substr`
- `splice` - το ανάλογο σε επίπεδο πίνακα της μορφής αντικατάστασης τεσσάρων ορισμάτων
- `pack` / `unpack` - για εγγραφές σταθερής διάταξης όπου πολλά πεδία διαβάζονται ή επανεγγράφονται μονομιάς, προτιμήστε αυτές αντί μιας αλυσίδας κλήσεων `substr`
- `tr///` - μαζική αντικατάσταση σε επίπεδο χαρακτήρων που δεν χρειάζεται θεσιακούς δείκτες

Filehandles, αρχεία, κατάλογοι

symlink

Δημιουργεί ένα symbolic link στο `NEWFILE` που δείχνει στη συμβολοσειρά `OLDFILE`.

Η `symlink` είναι το λεπτό περιτύλιγμα γύρω από την κλήση συστήματος POSIX `symlink(2)`. Δεν αντιγράφει τίποτα, δεν διαβάζει το `OLDFILE` και δεν απαιτεί το `OLDFILE` να υπάρχει. Γράφει μια νέα εγγραφή καταλόγου στο `NEWFILE` της οποίας το payload είναι τα κυριολεκτικά bytes του `OLDFILE`, και ο πυρήνας αργότερα επιλύει αυτό το payload κάθε φορά που κάτι ανοίγει, στατιστικοποιεί ή διασχίζει το link.

Σύνοψη

```
symlink OLDFILE, NEWFILE
```

Τι επιστρέφεται

1 σε επιτυχία, `0` σε αποτυχία (με την `$!` ορισμένη). Σε συστήματα που δεν υποστηρίζουν καθόλου symbolic links, η `symlink` εγείρει εξαίρεση αντί να επιστρέψει `0`. Για να διερευνήσετε την υποστήριξη κατά τον χρόνο εκτέλεσης χωρίς να τερματίσετε, τυλίξτε μια αβλαβή κλήση σε `eval`:

```
my $symlink_exists = eval { symlink("", ""); 1 };
```

Το μοτίβο `eval` είναι ο ιδιωματικός έλεγχος δυνατότητας - εκτελεί μια no-op `symlink` της οποίας η μόνη δουλειά είναι να αποτυγχάνει με χαμηλό κόστος σε πλατφόρμες όπου η κλήση υπάρχει, και να εκτοξεύει σε πλατφόρμες όπου δεν υπάρχει.

Πώς διαφέρουν τα symlinks από τους σκληρούς συνδέσμους

Ένα symlink και ένας σκληρός σύνδεσμος (`link`) μοιάζουν παρόμοιοι από το κέλυφος αλλά συμπεριφέρονται πολύ διαφορετικά:

- Η `link` δημιουργεί μια δεύτερη εγγραφή καταλόγου που δείχνει στο ίδιο inode. Και τα δύο ονόματα είναι ισότιμα· η διαγραφή του ενός αφήνει το αρχείο ζωντανό κάτω από το άλλο.
- Η `symlink` δημιουργεί ένα μικροσκοπικό αρχείο τύπου `lnk` του οποίου το περιεχόμενο είναι ένα *όνομα διαδρομής*. Ο στόχος επιλύεται κατ" όνομα σε κάθε αναζήτηση. Τα δύο ονόματα **δεν** είναι ισότιμα - το symlink έχει το δικό του inode, τα δικά του bits δικαιωμάτων (που ως επί το πλείστον αγνοούνται) και τις δικές του χρονικές σφραγίδες.

Πρακτικές συνέπειες:

- Το `OLDFILE` αποθηκεύεται **αυτούσιο**. Δεν κανονικοποιείται, δεν επιλύεται έναντι του τρέχοντος καταλόγου εργασίας και δεν ελέγχεται για ύπαρξη. Ένα symlink μπορεί να δημιουργηθεί δείχνοντας σε διαδρομή που δεν υπάρχει ακόμη (ένα *κρεμασμένο* symlink) και θα αρχίσει να λειτουργεί τη στιγμή που εμφανιστεί ο στόχος.
- Επειδή το `OLDFILE` είναι απλώς συμβολοσειρά, μπορεί να είναι σχετική διαδρομή. Οι σχετικοί στόχοι symlink επιλύονται κατά τον χρόνο αναζήτησης σχετικά με τον κατάλογο που περιέχει το `NEWFILE`, **όχι** τον τρέχοντα κατάλογο εργασίας της διεργασίας. Αυτή είναι η μεγαλύτερη μεμονωμένη πηγή εκπλήξεων.
- Τα symlinks μπορούν να διασχίζουν όρια συστημάτων αρχείων. Οι σκληροί σύνδεσμοι όχι.
- Τα symlinks μπορούν να δείχνουν σε καταλόγους. Οι σκληροί σύνδεσμοι προς καταλόγους απαγορεύονται σε κάθε mainstream Unix.
- Η `unlink` σε symlink αφαιρεί το **link**, όχι τον στόχο. Για να αφαιρέσετε τον στόχο, εκτελέστε unlink απευθείας στη διαδρομή του στόχου.

Παραδείγματα

Δημιουργία symlink προς υπάρχον αρχείο:

```
symlink "/etc/hostname", "hostname.lnk"
    or die "symlink failed: $!";
```

Κρεμασμένο symlink - ο στόχος δεν υπάρχει ακόμη, και αυτό είναι εντάξει:

```
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.
```

Σχετικός στόχος που επιλύεται έναντι του καταλόγου του link, όχι του cwd:

```
chdir "/var/log" or die $!;
symlink "syslog", "/tmp/current.lnk"
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    or die "symlink failed: $!";
# readlink("/tmp/current.lnk") eq "syslog"
# opening /tmp/current.lnk resolves syslog in /tmp, not /var/log.
```

Symlink προς κατάλογο - αυτό λειτουργεί, σε αντίθεση με την `link`:

```
symlink "/var/log", "logs"
    or die "symlink failed: $!";
opendir my $dh, "logs" or die $!;   # traverses the symlink
```

Επιθεώρηση και αφαίρεση symlink χωρίς να αγγίξετε τον στόχο του:

```
my $target = readlink "hostname.lnk";   # the stored OLDFILE string
unlink "hostname.lnk" or die $!;        # removes the link only
# /etc/hostname is untouched.
```

Μοτίβο ατομικής αντικατάστασης - αντικατάσταση symlink που δείχνει στον κατάλογο της «τρέχουσας» έκδοσης γράφοντας ένα νέο symlink και κάνοντας rename πάνω από αυτό:

```
symlink "release-2", "current.new"  or die $!;
rename  "current.new", "current"    or die $!;
# rename over an existing symlink is atomic on POSIX filesystems.
```

Οριακές περιπτώσεις

- **Το `OLDFILE` αποθηκεύεται αυτούσιο.** Καμία κανονικοποίηση, καμία επίλυση, κανένας έλεγχος ύπαρξης. Η `symlink "", "foo"` πετυχαίνει στα περισσότερα συστήματα και δημιουργεί symlink με κενό στόχο που κάθε αναζήτηση απορρίπτει με `ENOENT`. Γι" αυτό το ιδίωμα ελέγχου δυνατότητας χρησιμοποιεί `("", "")` - μια χαμηλού κόστους κλήση που αποτυγχάνει πάντα σε υποστηριζόμενες πλατφόρμες.
  - a cheap call that always fails on supporting platforms.
- **Το σχετικό `OLDFILE`** επιλύεται σχετικά με τον κατάλογο του `NEWFILE`, όχι τον τρέχοντα κατάλογο εργασίας. Αν χτίζετε το `NEWFILE` από σχετική διαδρομή, σκεφτείτε προσεκτικά σε ποιον κατάλογο θα το δει ο πυρήνας.
- **Το `NEWFILE` υπάρχει ήδη** → αποτυχία με την `$!` ορισμένη σε `EEXIST`. Δεν υπάρχει σημαία «αντικατάστασης»· για να αντικαταστήσετε, γράψτε σε νέο όνομα και `rename` πάνω από το παλιό (ατομικά).
- **Τα δικαιώματα στο ίδιο το symlink** σχεδόν πάντα αγνοούνται. Ο πυρήνας επιβάλλει τα δικαιώματα του *στόχου*. Στα περισσότερα συστήματα αρχείων το symlink αναφέρεται ως `lrwxrwxrwx` ανεξάρτητα από το τι καλείται η `chmod` σε αυτό.
- **Η `stat` ακολουθεί τα symlinks, η `lstat` όχι.** Για να εξετάσετε το ίδιο το link - τον τύπο του, το link count του, το μήκος της συμβολοσειράς στόχου του - χρησιμοποιήστε `lstat`. Για να εξετάσετε σε τι δείχνει το link, χρησιμοποιήστε `stat`.
- **Η `unlink` αφαιρεί το link, όχι τον στόχο.** Δεν υπάρχει τρόπος να διαγραφεί ο στόχος μέσω του symlink εκτός από το να τον επιλύσετε πρώτα.

- **Πλατφόρμες χωρίς υποστήριξη symlink** εγείρουν εξαίρεση σε οποιαδήποτε κλήση, ακόμα και σε μία που διαφορετικά θα αποτύγχανε. Χρησιμοποιήστε τον έλεγχο `eval` που δείχθηκε παραπάνω αντί να υποθέτετε μια τιμή επιστροφής.
- **Συστήματα αρχείων χωρίς υποστήριξη symlink** (ορισμένες παραλλαγές FAT, ορισμένες προσαρτήσεις δικτύου) αποτυγχάνουν με `EPERM` ή `EOPNOTSUPP` ακόμη και σε πυρήνα που υλοποιεί την `symlink(2)`. Ελέγξτε την `$!` σε αποτυχία αντί να υποθέτετε ότι η κλήση απορρίφθηκε για τον προφανή λόγο.
- **Οι κύκλοι symlink** δεν ανιχνεύονται κατά τη δημιουργία. Η `symlink "a", "b"` ακολουθούμενη από `symlink "b", "a"` πετυχαίνει· ο κύκλος εμφανίζεται μόνο όταν κάτι προσπαθήσει να επιλύσει κάποιο από τα δύο ονόματα, που αποτυγχάνει με `ELOOP` αφού ο πυρήνας φτάσει στο όριο καταδίωξής του (τυπικά 40).

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `link` - δημιουργεί σκληρό σύνδεσμο (ίδιο inode, ίδιο σύστημα αρχείων, μόνο αρχείο)· το αυστηρό αντίστοιχο της `symlink`
- `readlink` - διαβάζει τη συμβολοσειρά `OLDFILE` που είναι αποθηκευμένη σε symlink χωρίς να την επιλύσει
- `lstat` - κάνει stat στο ίδιο το symlink αντί στον στόχο του· ο μόνος τρόπος να δείτε από Perl ότι μια διαδρομή είναι symlink
- `stat` - κάνει stat σε αυτό στο οποίο δείχνει το symlink, ακολουθώντας την αλυσίδα συνδέσμων
- `unlink` - αφαιρεί ένα symlink (το link, όχι τον στόχο)· επίσης ο τρόπος αφαίρεσης κρεμασμένου symlink
- `$!` - η `errno` που τίθεται σε αποτυχία· επιθεωρήστε την για να διακρίνετε `EEXIST`, `EPERM`, `EOPNOTSUPP`, `ENOENT` στον γονικό κατάλογο

I/O · Δεδομένα σταθερού μήκους

syscall

Επικαλείται ωμή κλήση συστήματος με βάση τον αριθμό πυρήνα της, περνώντας τα υπόλοιπα ορίσματα ως `int` ή ως δείκτη σε ενταμιευτή συμβολοσειράς.

Η `syscall` είναι η διέξοδος για κλήσεις λειτουργικού συστήματος που η υπόλοιπη γλώσσα δεν περιτυλίγει. Το πρώτο όρισμα είναι ο αριθμός syscall - σε Linux, η τιμή του `SYS_<name>` από το `syscall.ph`. Κάθε υπόλοιπο όρισμα περνά είτε ως ακέραιος (αν η Perl το θεωρεί ήδη αριθμητικό) είτε ως δείκτης στον ενταμιευτή συμβολοσειράς του βαθμωτού. Επιστρέφει ό,τι επιστρέφει ο πυρήνας. Σε αποτυχία επιστρέφει `-1` και θέτει το `$!`.

Σύνοψη

```
syscall NUMBER, LIST
```

Τι επιστρέφεται

Ό,τι επέστρεψε ο πυρήνας. Οι τιμές επιτυχίας είναι ειδικές ανά syscall - συχνά αριθμός bytes, περιγραφέας αρχείου ή `0`. Η αποτυχία είναι `-1` με το `$!` τεθειμένο στην τιμή errno.

Ορισμένες syscalls (`lseek`, `read` σε σωλήνα στο EOF με σφάλματα, `ptrace`) μπορούν νόμιμα να επιστρέψουν `-1` σε επιτυχία. Για να τις διακρίνετε από αποτυχίες, καθαρίστε το `$!` πριν την κλήση:

```
$! = 0;
my $r = syscall(SYS_lseek(), fileno($fh), 0, 1);
die "lseek: $!" if $r == -1 && $! != 0;
```

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται όταν ο πυρήνας επιστρέφει `-1`. Δεν επηρεάζεται σε επιτυχία.

Κανόνες περάσματος ορισμάτων

Κάθε όρισμα στη `LIST` επιθεωρείται κατά τον χρόνο κλήσης:

- **Αριθμητικό** όρισμα (κυριολεκτικός αριθμός ή βαθμωτό που έχει χρησιμοποιηθεί σε αριθμητικό περιβάλλον) - περνά ως C `int`.
- **Μη αριθμητικό** όρισμα - ο ενταμιευτής συμβολοσειράς του βαθμωτού περνά με δείκτη. Είστε υπεύθυνοι να επεκτείνετε εκ των προτέρων τον ενταμιευτή ώστε να χωρά όσα δεδομένα μπορεί να γράψει ο πυρήνας. Ένας πολύ κοντός ενταμιευτής είναι σφάλμα αλλοίωσης μνήμης, όχι σφάλμα Perl.
- **Συμβολοσειρά μόνο για ανάγνωση** (κυριολεκτικό, συμβολοσειρά από `__DATA__`, οτιδήποτε η Perl αρνείται να γράψει διαμέσου του) - απορρίπτεται, επειδή η `syscall` πρέπει να υποθέσει ότι κάθε δείκτης συμβολοσειράς μπορεί να γραφτεί διαμέσου.
- **Ακέραιος αποθηκευμένος ως συμβολοσειρά** (π.χ. τιμή διαβασμένη από αρχείο που δεν χρησιμοποιήθηκε ποτέ αριθμητικά) - μπορεί να περαστεί ως δείκτης αντί για ακέραιος. Εξαναγκάστε την αριθμητική ερμηνεία προσθέτοντας `0`:

  ```
  syscall(SYS_write(), 0 + $fd, $buf, length $buf);
  ```

Υποστηρίζονται έως 14 ορίσματα μετά τον αριθμό syscall.

Παραδείγματα

Γράψτε έναν ενταμιευτή στο STDOUT χρησιμοποιώντας την ωμή syscall `write(2)`:

```
require 'syscall.ph';
my $s = "hi there\n";
syscall(SYS_write(), fileno(STDOUT), $s, length $s);
```

Προεκτείνετε έναν ενταμιευτή πριν μια ανάγνωση:

```
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
```

Καλέστε την `gettid(2)` για να λάβετε το id thread του πυρήνα (δεν υπάρχει wrapper libc σε παλαιότερη glibc):

```
require 'syscall.ph';
my $tid = syscall(SYS_gettid());
```

Διακρίνετε μια νόμιμη επιστροφή `-1` από πραγματική αποτυχία:

```
$! = 0;
my $off = syscall(SYS_lseek(), fileno($fh), 0, 1);   # SEEK_CUR
if ($off == -1 && $! != 0) {
    die "lseek failed: $!";
}
```

Εξαναγκάστε έναν ακέραιο που έχει μετατραπεί σε συμβολοσειρά να περαστεί ως `int`:

```
my $fd = read_config("fd");           # returns "3"
syscall(SYS_close(), 0 + $fd);        # without "0 +" it'd pass a␣
↪pointer
```

Οριακές περιπτώσεις

- **Λείπει το `syscall.ph`**: η `require 'syscall.ph'` αποτυγχάνει εκτός αν η h2ph έχει τρέξει έναντι των headers του συστήματος. Χωρίς αυτό οι σταθερές `SYS_*` είναι αόριστες και οι κλήσεις γίνονται χειρωνακτικά συντηρούμενοι μαγικοί αριθμοί.
- **Όρισμα μόνο για ανάγνωση**: το πέρασμα κυριολεκτικού συμβολοσειράς ή άλλου βαθμωτού μόνο για ανάγνωση πεθαίνει με `Modification of a read-only value`. Αντιγράψτε πρώτα σε λεξιλογικό: `my $s = "literal"; syscall(..., $s, ...)`.
- **Ενταμιευτής πολύ κοντός**: ο πυρήνας θα γράψει ευχαρίστως πέρα από το τέλος του ενταμιευτή ενός βαθμωτού αν είπατε ψέματα για το μήκος. Αυτό αλλοιώνει το heap της Perl. Προεπεκτείνετε πάντα με `$buf = "\0" x $n` ή `vec($buf, $n-1, 8) = 0` πριν την κλήση, και μην περνάτε ποτέ `length $buf` μεγαλύτερο από την πραγματική χωρητικότητα.
- **Ανώτατο όριο πλήθους ορισμάτων**: 14 ορίσματα μετά τον `NUMBER`. Syscalls που δέχονται περισσότερα (σπάνια) δεν είναι προσπελάσιμες μέσω της `syscall`.

- **Η `SYS_pipe()` σε Linux**: η ωμή syscall επιστρέφει το fd του άκρου ανάγνωσης αλλά δεν σας δίνει handle στο fd του άκρου εγγραφής. Χρησιμοποιήστε αντί γι" αυτό την `pipe`, που εκθέτει και τα δύο.
- **Τιμές 64-bit σε πλατφόρμες 32-bit**: ορισμένες syscalls δέχονται ορίσματα 64-bit (offsets, μεγέθη) που δεν χωρούν σε ένα Perl `int` σε builds 32-bit. Δεν υπάρχει φορητός τρόπος να τα περάσετε μέσω της `syscall` - χρησιμοποιήστε ειδικό wrapper (`Fcntl::lseek`, `sysseek` κ.λπ.) ή binding από CPAN.
- **Μη υλοποιημένη**: σε πλατφόρμες χωρίς σημείο εισόδου `syscall(2)` (ή όταν χτίστηκε χωρίς υποστήριξη), η `syscall` εγείρει `The syscall function is unimplemented`.
- **Διακοπή από σήμα**: μια syscall που διακόπτεται από σήμα επιστρέφει `-1` με το `$!` τεθειμένο σε `EINTR`. Η `syscall` δεν επανεκκινεί αυτόματα - ξαναδοκιμάστε σε επίπεδο Perl αν χρειάζεται.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `syswrite` - εγγραφή με παράκαμψη ενταμιευτή μέσω filehandle της Perl· σχεδόν πάντα αυτό που θέλετε αντί για `syscall(SYS_write(), ...)`
- `sysread` - ανάγνωση με παράκαμψη ενταμιευτή· συνδυάζεται με τη `syswrite` και χειρίζεται τη διαχείριση ενταμιευτή για εσάς
- `sysopen` - `open(2)` με το πλήρες λεξιλόγιο σημαιών, χωρίς να χρειάζεται η ωμή syscall
- `pipe` - ο σωστός τρόπος για να δημιουργήσετε σωλήνα· αποφεύγει το πρόβλημα του άκρου εγγραφής της `SYS_pipe()`
- `pack` - κατασκευάστε τις δυαδικές δομές (`struct timeval`, `struct stat` κ.λπ.) που αναμένουν οι syscalls σε ορίσματα δείκτη
- `unpack` - αποκωδικοποιήστε τις δυαδικές δομές που μια syscall γράφει στον ενταμιευτή σας

Filehandles, αρχεία, κατάλογοι

sysopen

Ανοίγει ένα αρχείο με χαμηλού επιπέδου τρόπο, περνώντας μια ακέραιη μάσκα bits `MODE` απευθείας στην υποκείμενη κλήση συστήματος `open(2)`.

Η `sysopen` είναι η διέξοδος για όταν το DWIM της λειτουργίας συμβολοσειράς της `open` σας εμποδίζει - όταν χρειάζεστε `O_EXCL` για ατομικό create-if-not-exist, `O_NOFOLLOW` για άρνηση συμβολικών λινκ, `O_NONBLOCK` σε FIFO, ή οποιαδήποτε άλλη σημαία που δεν έχει αντίστοιχη συμβολοσειρά `"<"` / `">"` / `">>"`. Οι σταθερές σημαιών βρίσκονται στο `Fcntl` και συνδυάζονται με `|`.

Σύνοψη

```
sysopen FILEHANDLE, FILENAME, MODE
sysopen FILEHANDLE, FILENAME, MODE, PERMS
```

Τι επιστρέφεται

1 σε επιτυχία, `undef` σε αποτυχία (με το `$!` ορισμένο στο `errno` από την αποτυχημένη κλήση συστήματος). Σε αντίθεση με την `open`, δεν υπάρχει μαγεία - το `FILENAME` λαμβάνεται κυριολεκτικά, δεν ερμηνεύεται κανένα προπορευόμενο `>`, `<`, `|`, ή `-`, δεν αναλύεται καμία συμβολοσειρά λειτουργίας με στρώματα.

Αν το `FILEHANDLE` είναι μη ορισμένο βαθμωτό, αυτο-δημιουργείται σε νέο handle:

```
sysopen my $fh, $path, O_RDONLY
    or die "sysopen $path: $!";
```

MODE - η μάσκα bits του Fcntl

Το `MODE` είναι ακέραιος κατασκευασμένος με OR σταθερών από το `Fcntl`. Η λειτουργία πρόσβασης είναι υποχρεωτική και αμοιβαία αποκλειόμενη· οι σημαίες τροποποιητών είναι προαιρετικές και συνδυάζονται ελεύθερα:

Σημαία	Σημασία
`O_RDONLY`	Άνοιγμα μόνο για ανάγνωση.
`O_WRONLY`	Άνοιγμα μόνο για εγγραφή.
`O_RDWR`	Άνοιγμα για ανάγνωση και εγγραφή.
`O_CREAT`	Δημιουργία του αρχείου αν δεν υπάρχει.
`O_EXCL`	Μαζί με `O_CREAT`: αποτυχία αν το αρχείο υπάρχει ήδη.
`O_APPEND`	Κάθε εγγραφή αναζητά πρώτα το τέλος του αρχείου (ατομική προσάρτηση).
`O_TRUNC`	Αποκοπή του αρχείου σε μηδενικό μήκος κατά το άνοιγμα.
`O_NONBLOCK`	Μη μπλοκάρουσα λειτουργία για τον προκύπτοντα περιγραφέα.
`O_NOFOLLOW`	Αποτυχία με `ELOOP` αν το τελικό συστατικό της διαδρομής είναι symlink.

Εισάγετέ τες με το όνομά τους ή με ετικέτα:

```
use Fcntl qw(O_RDWR O_CREAT O_EXCL O_APPEND);
# or
use Fcntl ':DEFAULT';
```

Οι παλιές τιμές `0`, `1`, `2` για read-only, write-only, read-write επίσης λειτουργούν σε κάθε σύστημα που υποστηρίζει η Perl, αλλά ονοματίστε τις σταθερές - ο προκύπτων κώδικας επιβιώνει αναθεώρηση μεταξύ πλατφορμών και διαβάζεται χωρίς γλωσσάρι.

PERMS - τα bits δικαιωμάτων κατά τη δημιουργία

Το PERMS είναι οκταδικό mode (όπως δέχεται η chmod) που εφαρμόζεται στο inode **μόνο όταν το O_CREAT δημιουργεί πραγματικά το αρχείο**. Αν παραλειφθεί το PERMS, η Perl περνά 0666. Η umask της διεργασίας εφαρμόζεται από τον πυρήνα από πάνω, οπότε μια προεπιλεγμένη umask 022 αποδίδει 0644 στον δίσκο.

```
sysopen my $fh, $path, O_WRONLY | O_CREAT | O_EXCL, 0600
    or die "create $path: $!";
```

Μη γράφετε 0644 ως όρισμα PERMS χωρίς λόγο. Ένα σκληρά κωδικοποιημένο 0644 αφαιρεί την εγγραφή ομάδας ακόμη και από χρήστες που σκόπιμα ορίζουν ανεκτικό umask· το 0666 μαζί με το umask του χρήστη είναι η φορητή προεπιλογή και αυτό χρησιμοποιεί η sysopen όταν παραλείπετε το όρισμα εντελώς.

Καθολική κατάσταση που επηρεάζει

- $! - ορίζεται σε αποτυχία στο errno που επιστρέφει η open(2).
- ${^OPEN} - η διαμόρφωση στρωμάτων της pragma open. Η sysopen εφαρμόζει την **ίδια** προεπιλεγμένη στοίβα PerlIO που θα εφαρμόσει η open για κλήση χωρίς στρώματα. Χρησιμοποιήστε binmode αμέσως μετά από επιτυχή sysopen όταν χρειάζεται να υπερκαλύψετε ή να αφαιρέσετε στρώματα (τυπικό για δυαδικά αρχεία, υποδοχές ως αρχεία, ή περιγραφείς O_NONBLOCK).
- umask διεργασίας - μασκάρει το PERMS όταν το O_CREAT δημιουργεί το αρχείο.

Παραδείγματα

Ατομικό create-if-not-exist - ο κανονικός λόγος καταφυγής στη 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;
```

Άνοιγμα-ή-δημιουργία για ανάγνωση/εγγραφή, διατηρώντας το υπάρχον περιεχόμενο:

```
use Fcntl qw(O_RDWR O_CREAT);
sysopen my $fh, $path, O_RDWR | O_CREAT, 0666
    or die "sysopen $path: $!";
```

Καταγραφή μόνο με προσάρτηση με την εγγύηση append - κάθε εγγραφή προσγειώνεται στο EOF ανεξάρτητα από την εναλλαγή με άλλους εγγραφείς:

```
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";
```

Άνοιγμα FIFO χωρίς μπλοκάρισμα στη χειραψία της πλευράς αναγνώστη:

```
use Fcntl qw(O_RDONLY O_NONBLOCK);
sysopen my $fifo, "/tmp/q", O_RDONLY | O_NONBLOCK
    or die "sysopen fifo: $!";
```

Άρνηση ακολούθησης symlink στο φύλλο - χρήσιμο σε καταλόγους εγγράψιμους από μη έμπιστους χρήστες:

```
use Fcntl qw(O_RDONLY O_NOFOLLOW);
sysopen my $fh, "$dir/config", O_RDONLY | O_NOFOLLOW
    or die "sysopen config: $!";
```

Επιβολή I/O σε επίπεδο byte μετά το άνοιγμα - αφαίρεση οποιουδήποτε στρώματος `:utf8` ή `:crlf` που η προεπιλεγμένη στοίβα PerlIO θα εγκαθιστούσε διαφορετικά:

```
sysopen my $fh, $path, O_RDONLY or die $!;
binmode $fh;
```

Οριακές περιπτώσεις

- **Το `O_EXCL` χωρίς `O_CREAT` είναι no-op.** Ο έλεγχος αποκλειστικότητας ενεργοποιείται μόνο όταν το `O_CREAT` ζητά από τον πυρήνα να δημιουργήσει το αρχείο. Γράφετέ τα πάντα μαζί: `O_CREAT | O_EXCL`.
- **Το `O_EXCL` δεν είναι κλείδωμα.** Εμποδίζει επιτυχημένο άνοιγμα σε αρχείο που ήδη υπάρχει, τίποτε άλλο. Μόλις το αρχείο υπάρχει, κάθε επόμενη `sysopen` με `O_CREAT | O_EXCL` αποτυγχάνει. Δεν σειριοποιεί την πρόσβαση στο περιεχόμενο - χρησιμοποιήστε `flock` γι” αυτό.
- **Το `O_EXCL` σε δικτυακά συστήματα αρχείων.** Οι υλοποιήσεις NFS διαφέρουν· παλαιότερες εκδόσεις NFS χάνουν σιωπηρά τη σημασιολογία αποκλειστικότητας. Μη βασίζεστε στο `O_EXCL` σε NFSv2.
- **`O_CREAT | O_EXCL` και symlinks.** Με τις δύο σημαίες ορισμένες, ο πυρήνας αρνείται να ανοίξει προϋπάρχον symlink στο τελικό συστατικό διαδρομής. **Δεν** προστατεύει ενδιάμεσα συστατικά διαδρομής· χρησιμοποιήστε `O_NOFOLLOW` ή επιλύστε τον κατάλογο με `opendir` και σημασιολογία `openat` αν το χρειάζεστε.
- **`O_TRUNC` με `O_RDONLY`.** Η συμπεριφορά είναι απροσδιόριστη από το POSIX. Μην τα συνδυάζετε.
- **Παραλειπόμενο `PERMS`.** Η προεπιλογή είναι `0666`, μασκαρισμένο από το `umask`. **Δεν** είναι `0644`. Παραλείπετε το `PERMS` εκτός αν ενεργά θέλετε να υπερκαλύψετε το `umask` του χρήστη.
- **`FILEHANDLE` ως έκφραση.** Αν το `FILEHANDLE` είναι bareword, ονομάζει μια καθολική μεταβλητή πακέτου. Αν είναι έκφραση που αξιολογείται σε glob, globref, ή `IO::Handle`, ανοίγει αυτό το handle. Αν είναι μη ορισμένο βαθμωτό, αυτο-δημιουργείται μέσα του ένα νέο handle - η λεξιλογική μορφή που χρησιμοποιείται σε όλα αυτά τα παραδείγματα.
- **Καμία μαγεία στο `FILENAME`.** Ένα `FILENAME "-"` ανοίγει αρχείο που ονομάζεται κυριολεκτικά `-`, όχι το `STDIN`. Ένα προπορευόμενο `>` ανοίγει αρχείο που ονομάζεται κυριολεκτικά `>filename`, όχι εγγραφή στο `filename`. Αυτό είναι όλο το νόημα της `sysopen` έναντι της `open`.

- **Τα προεπιλεγμένα στρώματα PerlIO εξακολουθούν να εφαρμόζονται.** Η `sysopen` είναι «χαμηλού επιπέδου» σε σχέση με τη συμβολοσειρά λειτουργίας, όχι σε σχέση με το PerlIO. Το προκύπτον handle έχει την ίδια στοίβα στρωμάτων με μια χωρίς στρώματα `open`· καλέστε `binmode` για να την αλλάξετε.
- **Κλειστή ή μη έγκυρη έκφραση `FILEHANDLE`.** Επιστρέφει `undef` με ορισμένο το `$!` (συνήθως `EBADF` για κακό υπάρχοντα περιγραφέα· `ENOENT`, `EACCES`, `EEXIST`, `ELOOP` για σφάλματα σε επίπεδο διαδρομής).

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `open` - υψηλότερου επιπέδου ανοιχτήρας με ανάλυση συμβολοσειράς λειτουργίας, σωλήνες, `-` ως `STDIN` / `STDOUT`, και λειτουργίες με στρώματα· χρησιμοποιήστε το εκτός αν χρειάζεστε ειδικά κάποια σημαία που εκθέτει η `sysopen`
- `fcntl` - αλλάζει σημαίες σε ήδη ανοιχτό περιγραφέα (π.χ. ορισμός `O_NONBLOCK` εκ των υστέρων)· μοιράζεται το λεξιλόγιο σταθερών του `Fcntl`
- `sysread` - ανάγνωση χωρίς ενταμιευτή· λειτουργεί σε οποιοδήποτε handle, όχι μόνο σε αυτό που ελήφθη από `sysopen`, παρά το όνομα
- `syswrite` - εγγραφή χωρίς ενταμιευτή, ίδια σημείωση
- `Fcntl` - το άρθρωμα που εξάγει τις σταθερές `O_*` και τις ετικέτες `:DEFAULT` / `:flock` / `:mode`
- `binmode` - ορισμός ή αφαίρεση στρωμάτων PerlIO στο handle μετά το άνοιγμα
- `umask` - η μάσκα διεργασίας που αφαιρεί bits από το `PERMS` όταν το `O_CREAT` δημιουργεί το αρχείο

I/O · Δεδομένα σταθερού μήκους

sysread

Διαβάζει ωμά bytes από ένα filehandle καλώντας την υποκείμενη κλήση συστήματος `read(2)`.

Η `sysread` προσπαθεί να διαβάσει `LENGTH` bytes από το `FILEHANDLE` και να τα αποθηκεύσει στο `SCALAR`. Παρακάμπτει τη στοίβα ενταμίευσης του PerlIO, που είναι ο όλος σκοπός - οι αναγνώσεις πηγαίνουν κατευθείαν στον πυρήνα και επιστρέφουν ό,τι επιστρέφει μία μεμονωμένη `read(2)`, που μπορεί να είναι λιγότερα bytes από αυτά που ζητήθηκαν. Χρησιμοποιήστε την για υποδοχές, σωλήνες, συσκευές και κάθε άλλη κατάσταση όπου θέλετε έλεγχο πάνω σε μεμονωμένες κλήσεις συστήματος αντί για είσοδο προσανατολισμένη σε γραμμή ή εγγραφή.

Σύνοψη

```
sysread FILEHANDLE, SCALAR, LENGTH
sysread FILEHANDLE, SCALAR, LENGTH, OFFSET
```

Τι επιστρέφεται

- Ο αριθμός των bytes που διαβάστηκαν πραγματικά, που μπορεί να είναι **λιγότερα από** το LENGTH σε υποδοχές, σωλήνες, ttys και κατά τη διάρκεια διακοπής από σήμα.
- `0` στο τέλος αρχείου.
- `undef` σε σφάλμα, με την `$!` ορισμένη στην `errno` της αποτυχημένης `read(2)`.

Το SCALAR μεγαλώνει ή συρρικνώνεται ώστε το τελευταίο byte που διαβάστηκε να γίνει το τελευταίο byte του βαθμωτού. Μετά από σύντομη ανάγνωση, το βαθμωτό κρατά μόνο τα bytes που ελήφθησαν πραγματικά - τίποτα δεν συμπληρώνεται με μηδενικά μέχρι το 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
```

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται όταν η `sysread` επιστρέφει `undef`. Οι διακοπείσες αναγνώσεις εμφανίζονται ως `EINTR`· οι αναγνώσεις που θα μπλόκαραν σε μη μπλοκαριστικά handles εμφανίζονται ως `EAGAIN` / `EWOULDBLOCK`.
- Το ίδιο το SCALAR - του γίνεται πάντα ανάθεση, ακόμη και σε αποτυχία (αποκόπτεται σε OFFSET bytes όταν δίνεται OFFSET και η ανάγνωση αποτυγχάνει ή επιστρέφει μηδέν).
- Η θέση byte του filehandle προωθείται κατά τον αριθμό των bytes που επιστράφηκαν· αυτή είναι η ίδια θέση σε επίπεδο πυρήνα που χειρίζεται η `sysseek`.

Η `sysread` δεν συμβουλεύεται ούτε αγγίζει τις `$/`, `$\`, `$,`, ή `$_`. Είναι πρωτογενής λειτουργία επιπέδου byte· η σημασιολογία εγγραφών δεν εφαρμόζεται.

OFFSET - πού προσγειώνονται τα bytes μέσα στο SCALAR

Χωρίς OFFSET, το SCALAR αντικαθίσταται από τα bytes που διαβάστηκαν.

Με OFFSET, τα bytes γράφονται **μέσα** στο SCALAR ξεκινώντας από εκείνη τη θέση:

- `OFFSET >= 0` - αρχίζει η εγγραφή στο byte OFFSET. Αν το OFFSET είναι μεγαλύτερο από το τρέχον μήκος του SCALAR, η συμβολοσειρά συμπληρώνεται πρώτα με `"\0"` bytes μέχρι το OFFSET, και κατόπιν προστίθενται τα νέα δεδομένα.
- `OFFSET < 0` - μετριέται προς τα πίσω από το τέλος του SCALAR. Το `-1` σημαίνει «αντικατάσταση του τελευταίου byte», το `-10` σημαίνει «αντικατάσταση των τελευταίων 10 bytes».

Έτσι μεγαλώνετε σταδιακά έναν ενταμιευτή χωρίς επαναλαμβανόμενη συνένωση:

```
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: $!";
```

Παραδείγματα

Ανάγνωση έως 64 bytes από αρχείο. Το `$buf` καταλήγει ακριβώς `$n` bytes σε μήκος:

```
open my $fh, "<", "input.bin" or die $!;
my $n = sysread $fh, my $buf, 64;
defined $n or die "sysread: $!";
```

Ανάγνωση έως 32 bytes και τοποθέτησή τους ξεκινώντας από τη θέση 512 στο `$buf`, με συμπλήρωση `"\0"` αν το `$buf` ήταν μικρότερο:

```
my $buf = "header";
my $n = sysread $fh, $buf, 32, 512;   # $buf is now 512 + $n bytes
```

Αποστράγγιση υποδοχής μέχρι να κλείσει ο ομότιμος. Ο έλεγχος για `0` - όχι για `eof` - είναι ο μόνος σωστός έλεγχος τέλους ροής για την `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);
}
```

Επανεκκίνηση ανάγνωσης που διέκοψε σήμα. Σε μπλοκαριστικό handle, το `EINTR` είναι η μία αποτυχία που τυπικά επανεπιχειρείτε αντί να διαδώσετε:

```
use Errno qw(EINTR);
my $n;
RETRY: {
    $n = sysread $fh, my $buf, $want;
    redo RETRY if !defined $n && $! == EINTR;
}
defined $n or die "read: $!";
```

Γέμισμα εγγραφής σταθερού μεγέθους, ανεκτικά απέναντι σε σύντομες αναγνώσεις. Μία `sysread` μπορεί να επιστρέψει λιγότερα bytes από όσα ζητήθηκαν ακόμη και σε κανονικό αρχείο κοντά στο EOF, και τακτικά το κάνει σε υποδοχές και σωλήνες:

```
sub read_exact {
    my ($fh, $want) = @_;
    my $buf = "";
    while (length($buf) < $want) {
        my $n = sysread $fh, $buf, $want - length($buf), length
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
↪$buf;
        defined $n          or die "read: $!";
        $n == 0 and die "short read: got ", length $buf, " of $want
↪";
    }
    return $buf;
}
```

Οριακές περιπτώσεις

- **Ποτέ μην αναμιγνύετε με ενταμιευμένο I/O στο ίδιο handle.** Οι `readline`, `read`, `<$fh>`, `getc`, `eof`, `seek` και `tell` περνούν όλες από ενταμιευτές PerlIO· η `sysread` τους παρακάμπτει. Η διαπλοκή τους στο ίδιο handle αφήνει δεδομένα εγκλωβισμένα στον ενταμιευτή που η `sysread` δεν θα δει ποτέ, ή προσπερνά δεδομένα που ο ενταμιευμένος αναγνώστης κατανάλωσε ήδη. Χρησιμοποιήστε ένα στυλ ανά handle.
- **Το `:utf8` απαγορεύεται.** Handle με στρώμα `:utf8` (συμπεριλαμβανομένου του έμμεσου στρώματος που προστίθεται από το `:encoding(...)`) κάνει την `sysread` να εκτοξεύσει εξαίρεση. Αφαιρέστε το στρώμα με `binmode` - η `binmode $fh` χωρίς δεύτερο όρισμα επαναφέρει ωμά bytes - πριν καλέσετε την `sysread`.
- **Το `:crlf` / `:perlio` εντάμιεύουν παρ" όλα αυτά.** Ακόμη και χωρίς `:utf8`, η προεπιλεγμένη στοίβα `:perlio` εντάμιεύει αναγνώσεις και μεταφράζει τερματισμούς γραμμών υπό `:crlf`. Η `sysread` αγνοεί εκείνα τα στρώματα, οπότε bytes που ο ενταμιευμένος αναγνώστης τράβηξε ήδη στον ενταμιευτή PerlIO χάνονται σιωπηρά για την `sysread`. Ανοίξτε handles προοριζόμενα για `sysread` με `sysopen` ή καλέστε `binmode $fh` για να απενεργοποιήσετε το `:crlf`.
- **Οι σύντομες αναγνώσεις είναι κανονικές, όχι σφάλματα.** Επιστροφή `$n` με `0 < $n < LENGTH` είναι επιτυχία. Μόνο το `undef` υποδηλώνει αποτυχία· το `0` υποδηλώνει EOF. Κάντε βρόχο μέχρι να έχετε ό,τι χρειάζεστε.
- **Δεν υπάρχει `syseof`.** Δεν υπάρχει ξεχωριστός έλεγχος τέλους-αρχείου για την `sysread`· η `eof` κοιτά τον ενταμιευτή PerlIO και είναι άνευ νοήματος εδώ. Η τιμή επιστροφής `0` είναι το σήμα τέλους-αρχείου.
- **`OFFSET` πέρα από το τρέχον μήκος.** Η `sysread $fh, $buf, 10, 1_000_000` συμπληρώνει το `$buf` με ένα εκατομμύριο `"\0"` bytes πριν γράψει. Αυτό είναι περιστασιακά χρήσιμο για προ-δέσμευση, αλλά συνήθως είναι σφάλμα.
- **Το αρνητικό `OFFSET`** μετριέται από το τρέχον τέλος του `SCALAR`, όχι από το `LENGTH`. Η `sysread $fh, $buf, 4, -4` αντικαθιστά τα τελευταία τέσσερα bytes ό,τι κρατά αυτή τη στιγμή το `$buf`.
- **Μη μπλοκαριστικά handles.** Σε handle με `O_NONBLOCK` ορισμένο, μια ανάγνωση χωρίς διαθέσιμα δεδομένα επιστρέφει `undef` με την `$!` ίση με `EAGAIN` ή `EWOULDBLOCK`. Αντιμετωπίστε το ως «δοκιμάστε αργότερα», όχι ως σφάλμα.
- **Σήματα.** Μια μπλοκαριστική `sysread` που διακόπτεται από σήμα επιστρέφει `undef` με την `$!` ίση με `EINTR`. Επαναλάβετε την κλήση.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `syswrite` - το αντίστοιχο της πλευράς εγγραφής· ίδιοι κανόνες ενταμίευσης, ίδια σημασιολογία σύντομου I/O
- `sysopen` - ανοίγει handle με ωμές σημαίες `open(2)`, ο φυσικός συνεργάτης της `sysread` σε αρχεία
- `sysseek` - αναζήτηση ακριβείας byte που συντίθεται σωστά με την `sysread`· μην αναμιγνύετε `seek` με `sysread`
- `read` - το ενταμιευμένο αντίστοιχο· χρησιμοποιήστε την για είσοδο προσανατολισμένη σε γραμμή ή εγγραφή, όχι για υποδοχές ή μερικές αναγνώσεις
- `binmode` - αφαιρεί `:utf8` ή `:crlf` από handle πριν τη χρήση της `sysread`
- `$!` - η τιμή `errno` μετά από αποτυχημένη `sysread`, συμπεριλαμβανομένων των `EINTR`, `EAGAIN` και `EWOULDBLOCK`

I/O · Δεδομένα σταθερού μήκους

sysseek

Επανατοποθέτηση filehandle σε επίπεδο συστήματος, παρακάμπτοντας την ενταμίευση PerlIO.

Η `sysseek` μετακινεί το offset περιγραφέα αρχείου πυρήνα του `FILEHANDLE` σε νέα θέση byte καλώντας απευθείας την υποκείμενη `lseek(2)`. Είναι η συντρόφισσα των `sysread` και `syswrite`: οι τρεις σχηματίζουν την οικογένεια μη ενταμιευμένου I/O, που μιλά στον πυρήνα χωρίς να περνά μέσω της στοίβας buffer `:perlio` ή τύπου stdio της Perl. Χρησιμοποιήστε την `sysseek` σε handles στα οποία έχετε πρόσβαση με την sys-οικογένεια· χρησιμοποιήστε την `seek` σε handles με `read`, `readline` ή `print`.

Σύνοψη

```
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
```

Τι επιστρέφεται

Το νέο απόλυτο offset byte σε επιτυχία, ή `undef` σε αποτυχία (με την `$!` ορισμένη). Μια θέση μηδέν επιστρέφεται ως η dualvar συμβολοσειρά `"0 but true"` ώστε η τιμή να εκτυπώνεται ως `0` σε αριθμητικό περιβάλλον αλλά να παραμένει αληθής σε λογικό περιβάλλον - ένας έλεγχος όπως `if (sysseek ...)` είναι ασφαλής σε offset μηδέν.

```
defined(sysseek $fh, $offset, 0)
    or die "sysseek to $offset failed: $!";
```

Επειδή η τιμή επιστροφής φέρει το νέο offset, η `sysseek` διπλασιάζεται ως πρωτογενής «πού βρίσκομαι» - ο λόγος που το ιδιωματικό `systell` είναι μία γραμμή:

```
use Fcntl 'SEEK_CUR';
sub systell { sysseek($_[0], 0, SEEK_CUR) }
```

Τιμές WHENCE

Το `WHENCE` είναι ακέραιος με τρεις ουσιαστικές τιμές. Προτιμήστε τις συμβολικές σταθερές από το `Fcntl` - καθιστούν επίσης την κλήση φορητή σε πλατφόρμες που δεν χρησιμοποιούν `0` / `1` / `2`:

- `0` / `SEEK_SET` - το `POSITION` μετριέται από την **αρχή του αρχείου**. Το `POSITION` πρέπει να είναι μη αρνητικό.
- `1` / `SEEK_CUR` - το `POSITION` προστίθεται στην **τρέχουσα θέση**. Αρνητικές τιμές μετακινούν προς τα πίσω, θετικές προς τα εμπρός. Η `sysseek $fh, 0, 1` επιστρέφει το τρέχον offset χωρίς να μετακινήσει τον δείκτη.
- `2` / `SEEK_END` - το `POSITION` προστίθεται στο offset **τέλους αρχείου**. Το `POSITION` τυπικά είναι μηδέν ή αρνητικό.

```
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, όχι χαρακτήρες

Η `sysseek` λειτουργεί σε **offsets byte**, πάντα. Ακόμη και όταν το handle έχει layer προσανατολισμένο σε χαρακτήρες όπως `:encoding(UTF-8)`, το offset που δέχεται και επιστρέφει είναι αδρό πλήθος byte. Αυτό ταιριάζει με την οπτική του πυρήνα αλλά είναι άσχετο στην πράξη: η `sysseek` προορίζεται για handles που δεν χρησιμοποιούν layers PerlIO, και η ανάμειξή της με layer αποκωδικοποίησης είναι μία από τις παγίδες σύγχυσης που απαριθμούνται παρακάτω.

Αν τα δεδομένα είναι προσανατολισμένα σε χαρακτήρες, χρησιμοποιήστε τις `seek` και `tell` στην ενταμιευμένη πλευρά αντί της `sysseek`.

Μην αναμειγνύετε με ενταμιευμένο I/O

Η `sysseek` μιλά απευθείας στον περιγραφέα αρχείου. Οι `read`, `readline`, `print`, `write`, `seek`, `tell` και `eof` μιλούν στον buffer PerlIO που βρίσκεται μπροστά από τον περιγραφέα. Η θέση του buffer και η θέση του πυρήνα είναι ανεξάρτητες - μια `sysseek` μετακινεί τον δείκτη του πυρήνα αλλά δεν εκκενώνει ούτε ακυρώνει τον buffer, οπότε η επόμενη ενταμιευμένη ανάγνωση επιστρέφει παλιά bytes πριν από τη `sysseek`, και η επόμενη ενταμιευμένη εγγραφή προσγειώνεται όπου νόμιζε ο buffer ότι ήταν.

Ο κανόνας είναι απλός: επιλέξτε μία οικογένεια ανά handle.

- Handle που χρησιμοποιείται με `sysread` / `syswrite` → επανατοποθέτηση με `sysseek`.

- Handle που χρησιμοποιείται με `read` / `print` / `readline` → επανατοποθέτηση με `seek`.

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται στο μήνυμα σφάλματος συστήματος σε αποτυχία.
- Σε αντίθεση με την `seek`, η `sysseek` **δεν** εκκενώνει ούτε απορρίπτει τον buffer PerlIO και **δεν** καθαρίζει τη σημαία EOF στο handle. Αυτή η ασυμμετρία είναι η πηγή του κινδύνου «ανάμειξης» παραπάνω και ο λόγος που οι δύο οικογένειες δεν πρέπει να μοιράζονται handle.

Παραδείγματα

Ανάγνωση τυχαίας πρόσβασης σε εγγραφή σταθερού μήκους. Το αρχείο κρατά εγγραφές 128 byte· μετάβαση κατευθείαν στην εγγραφή 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: $!";
```

Αναφορά του τρέχοντος offset χωρίς μετακίνηση του δείκτη - το ιδίωμα `systell`:

```
use Fcntl 'SEEK_CUR';
my $pos = sysseek $fh, 0, SEEK_CUR;
defined $pos or die "sysseek: $!";
print "at byte $pos\n";
```

Προσθήκη σε αρχείο που κρατείται ανοιχτό για ανάγνωση-εγγραφή, και έπειτα επαναφορά για επανάληψη από την αρχή:

```
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 $!;
```

Το μηδέν παραμένει αληθές - η dualvar σας επιτρέπει να γράψετε τη συνοπτική μορφή χωρίς να αντιμετωπίζετε κατά λάθος μια επιτυχή επαναφορά ως αποτυχία:

```
if (my $pos = sysseek $fh, 0, 0) {
    # entered even when $pos stringifies to "0"
    printf "rewound; pos=%d\n", $pos;
}
else {
    die "sysseek failed: $!";
}
```

Μετάβαση πέρα από το τέλος αρχείου για δημιουργία αραιής τρύπας, και έπειτα εγγραφή του δείκτη ουράς:

```
use Fcntl 'SEEK_SET';
sysseek $fh, 1_000_000, SEEK_SET or die $!;
syswrite $fh, "END"              or die $!;
```

Οριακές περιπτώσεις

- **Μη επανατοποθετήσιμα handles**: pipes, υποδοχές, TTYs, και οι περισσότερες ειδικές συσκευές αποτυγχάνουν με την `$!` ορισμένη σε `ESPIPE` (`Illegal seek`). Η τιμή επιστροφής είναι `undef`· ελέγχετε πάντα με `defined`.
- **Μην αναμειγνύετε με ενταμιευμένο I/O στο ίδιο handle**. Η `sysseek` παρακάμπτει τον buffer PerlIO· οι `read`, `readline`, `print`, `write`, `seek`, `tell` και `eof` τον χρησιμοποιούν. Η εναλλαγή των δύο οικογενειών παράγει σιωπηρά λάθος αποτελέσματα. Επιλέξτε μία οικογένεια ανά handle και μείνετε σε αυτή.
- **Το μηδέν μετατρέπεται σε συμβολοσειρά ως `"0 but true"`**. Οι λογικοί έλεγχοι λειτουργούν σε offset μηδέν - η `if (sysseek ...)` παραμένει αληθής - αλλά οι συγκρίσεις συμβολοσειρών με την κυριολεκτική `"0"` όχι:

  ```
  my $pos = sysseek $fh, 0, 0;
  $pos == 0     or die;           # true
  $pos eq "0"   and die "nope";   # false - the string is "0 but␣
  ↪true"
  ```

  Χρησιμοποιήστε `defined` για ανίχνευση επιτυχίας/αποτυχίας και αριθμητική σύγκριση για την τιμή offset.
- **Layers κωδικοποίησης**: η `sysseek` σε handle με layer `:encoding(...)` είναι σχεδόν πάντα λάθος - το offset byte μπορεί να προσγειωθεί στη μέση codepoint, και η sys-οικογένεια προορίζεται έτσι κι αλλιώς για αδρά bytes. Αν τα δεδομένα χρειάζονται αποκωδικοποίηση, χρησιμοποιήστε `seek` με την ενταμιευμένη οικογένεια.
- **Αρνητικά offsets με SEEK_SET**: το `POSITION` πρέπει να είναι μη αρνητικό όταν το `WHENCE` είναι `0`. Αρνητική τιμή αποτυγχάνει με `EINVAL`.
- **Πολύ μεγάλα αρχεία**: η `sysseek` επιστρέφει το πλήρες 64-bit offset του πυρήνα· θέσεις πέραν των 2 GiB επιστρέφουν σωστά μέσω επόμενων κλήσεων `sysseek` σε όλες τις υποστηριζόμενες 64-bit εκδόσεις.
- **Κλειστό filehandle**: επιστρέφει `undef` και θέτει την `$!`· υπό `use warnings` εκπέμπεται προειδοποίηση `sysseek() on closed filehandle`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `seek` - το ενταμιευμένο αντίστοιχο· χρησιμοποιήστε την σε handles που προσπελάζονται μέσω `read`, `readline` ή `print`
- `sysread` - η μη ενταμιευμένη ανάγνωση που συνδυάζεται με τη `sysseek`· και οι δύο παρακάμπτουν το PerlIO και μιλούν απευθείας στον περιγραφέα αρχείου

- `syswrite` - η μη ενταμιευμένη εγγραφή· το τρίτο μέλος της οικογένειας που ανήκει στο ίδιο handle με τη `sysseek`
- `sysopen` - ανοίγει handle σε επίπεδο περιγραφέα, ο συνηθισμένος τρόπος να αποκτήσετε ένα handle που θα οδηγείτε αργότερα με τη sys-οικογένεια
- `tell` - ερώτημα τρέχοντος offset στην ενταμιευμένη πλευρά· συνδυάστε την με την `seek`, όχι με τη `sysseek`
- `Fcntl` - πηγή των σταθερών `SEEK_SET`, `SEEK_CUR`, `SEEK_END`

Διεργασίες

system

Εκτελεί ένα ξεχωριστό πρόγραμμα και περιμένει να τελειώσει.

Η `system` κάνει ακριβώς το ίδιο πράγμα με την `exec`, με τη διαφορά ότι πρώτα γίνεται `fork` και η γονική διεργασία περιμένει το παιδί να τερματίσει. Η τιμή επιστροφής είναι η κατάσταση αναμονής του παιδιού, όχι η έξοδός του - αν θέλετε την έξοδο μιας εντολής, χρησιμοποιήστε αντί αυτής backticks ή `qx//`.

Σύνοψη

```
system LIST
system PROGRAM LIST
```

Η έμμεση μορφή `PROGRAM LIST` ψεύδεται στο πρόγραμμα για το ίδιο του το όνομα: το `PROGRAM` είναι το αρχείο που εκτελείται πραγματικά, ενώ το πρώτο στοιχείο της `LIST` γίνεται `argv[0]` μέσα στο παιδί. Ίδιο κόλπο με την `exec`.

Τι επιστρέφεται

Η τιμή επιστροφής είναι η ωμή λέξη κατάστασης από την υποκείμενη κλήση `wait` - η ίδια τιμή που προσγειώνεται στην `$?` μετά την κλήση. Για να ανακτήσετε τον πραγματικό κωδικό εξόδου του προγράμματος, μετατοπίστε δεξιά κατά οκτώ:

```
system("make", "install");
my $exit = $? >> 8;
```

Το `-1` σημαίνει ότι το πρόγραμμα δεν μπόρεσε καθόλου να ξεκινήσει (ή ότι η ίδια η `wait` απέτυχε)· επιθεωρήστε την `$!` για τον λόγο. Επιστροφή `0` σημαίνει ότι το παιδί εξήλθε με επιτυχία με κατάσταση `0`.

Εκτέλεση μέσω κελύφους έναντι απευθείας

Η επεξεργασία ορισμάτων εξαρτάται από τη μορφή της `LIST`, και αντικατοπτρίζει ακριβώς την `exec`:

- **Περισσότερα από ένα ορίσματα** (ή πίνακας με περισσότερα από ένα στοιχεία): το πρώτο στοιχείο είναι το πρόγραμμα, τα υπόλοιπα τα ορίσματά του. Το κέλυφος **δεν** καλείται· η `execvp` τρέχει το πρόγραμμα απευθείας. Αυτή είναι η ασφαλής μορφή.

- **Ακριβώς ένα βαθμωτό όρισμα**: η Perl το σαρώνει για μετα-χαρακτήρες κελύφους. Αν υπάρχουν, ολόκληρη η συμβολοσειρά περνά στο `/bin/sh -c` για ανάλυση. Αν δεν υπάρχουν, η Perl τη διασπά σε λευκούς χαρακτήρες και καλεί απευθείας την `execvp` - εξοικονομώντας ένα κέλυφος.

Προτιμήστε τη μορφή λίστας όποτε τα ορίσματα προέρχονται από μη έμπιστη είσοδο ή περιέχουν χαρακτήρες που το κέλυφος θα ερμήνευε:

```
system("ls", "-l", $dir);              # safe, no shell
system("ls -l $dir");                  # shell parses $dir -␣
↪hazardous
```

Σήματα έναντι exit

Η λέξη κατάστασης συνδυάζει τρία κομμάτια πληροφορίας: έναν αριθμό σήματος (αν το παιδί τερμάτισε από σήμα), μια σημαία core-dump και την τιμή εξόδου. Αποκωδικοποιήστε την ρητά όταν χρειάζεται να τα ξεχωρίσετε:

```
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;
}
```

Για δομημένη αποκωδικοποίηση, επιθεωρήστε την `${^CHILD_ERROR_NATIVE}` - την πλήρη OS-native λέξη κατάστασης - με τις μακροεντολές `WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG` από το `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);    ... }
```

Τα `SIGINT` και `SIGQUIT` αγνοούνται στον γονέα όσο τρέχει το παιδί. Αν θέλετε το δικό σας πρόγραμμα να τερματίσει όταν ο χρήστης πατήσει Ctrl-C κατά τη διάρκεια της `system`, ελέγξτε οι ίδιοι την κατάσταση επιστροφής και επανεκπέμψτε:

```
system(@cmd);
kill 'INT', $$ if ($? & 127) == 2;    # re-raise SIGINT on self
```

Εκκένωση stdio

Πριν το fork, η Perl επιχειρεί να εκκενώσει κάθε filehandle ανοιχτό για έξοδο. Αυτό αποτρέπει τη διπλή εμφάνιση ενταμιευμένων γραμμών από γονέα και παιδί. Η εκκένωση είναι best-effort· σε πλατφόρμες όπου είναι αναξιόπιστη, ορίστε την `$|` (`$AUTOFLUSH`) ή καλέστε τη μέθοδο `autoflush` από το `IO::Handle` στα handles που σας ενδιαφέρουν.

Το ιδίωμα die

Ο κανονικός έλεγχος - λιτός, και ενσωματώνει τη λέξη κατάστασης στο μήνυμα ώστε ο καλών να μπορεί ακόμη να την αποκωδικοποιήσει:

```
my @args = ("command", "arg1", "arg2");
system(@args) == 0
    or die "system @args failed: $?";
```

Για ένα άρθρωμα που θα έπρεπε να εκτοξεύει σε κάθε αποτυχημένη `system` (και πολλά άλλα κομμάτια της Perl), χρησιμοποιήστε την pragma `autodie`.

Παραδείγματα

Εκτέλεση εντολής χωρίς να ληφθεί υπόψη το αποτέλεσμα:

```
system("clear");
```

Εκτέλεση εντολής, τερματισμός σε αποτυχία, και διατήρηση της κατάστασης ανέπαφης για κατάντη επιθεώρηση:

```
system("tar", "xf", $archive) == 0
    or die "untar $archive failed: $?";
```

Ανίχνευση του αν είναι διαθέσιμο ένα πρόγραμμα τρέχοντάς το με `--version` και απορρίπτοντας την έξοδό του:

```
my $ok = system("which $prog >/dev/null 2>&1") == 0;
```

Εκτέλεση κάτω από συγκεκριμένο κέλυφος ρητά, παρακάμπτοντας την ευρετική ανίχνευσης μετα-χαρακτήρων της Perl:

```
system("/bin/bash", "-c", $pipeline);
```

Ψέμα για το `argv[0]` με χρήση της έμμεσης μορφής - ο πυρήνας εκτελεί το `/usr/bin/vim`, αλλά το πρόγραμμα βλέπει το όνομά του ως `editor`:

```
system { "/usr/bin/vim" } "editor", $file;
```

Οριακές περιπτώσεις

- **Επιστροφή -1**: το παιδί δεν δημιουργήθηκε ποτέ, ή απέτυχε η ίδια η `wait`. Διακριτό από το «το παιδί εξήλθε με κατάσταση 255». Ελέγξτε την `$!` για τον λόγο - τυπικά `ENOENT` (το πρόγραμμα δεν βρέθηκε) ή `EACCES` (μη εκτελέσιμο).

- **Τιμή εξόδου 255 από κέλυφος**: όταν χρησιμοποιείται η μορφή κελύφους και το `/bin/sh` δεν μπορεί να βρει το πρόγραμμα, το ίδιο το κέλυφος εξέρχεται με 127. Προγράμματα που επιστρέφουν 255 από Perl τυπικά προέρχονται μέσω `die` σε άλλο πρόγραμμα Perl.
- **Αλληλεπίδραση χειριστή SIGCHLD**: η `system` εσωτερικά κάνει fork και περιμένει. Ένας χειριστής `$SIG{CHLD}` μπορεί να παρατηρήσει ή να μπει σε race με αυτή την αναμονή· δείτε το `perlipc` για την πειθαρχία συγκομιδής.
- **Ιδιοτροπίες κελύφους**: όταν ενεργοποιείται η μορφή κελύφους, οι κωδικοί επιστροφής και η αναφορά σφαλμάτων είναι ό,τι λέει το `/bin/sh` - όχι πάντα ό,τι επέστρεψε το υποκείμενο πρόγραμμα. Χρησιμοποιήστε τη μορφή λίστας για να το αποφύγετε.
- **Δεν είναι για αποτύπωση εξόδου**: η `system` γράφει την έξοδο του παιδιού στα κληρονομημένα STDOUT/STDERR. Χρησιμοποιήστε `qx//` ή ανοίξτε σωλήνα με `open` αν χρειάζεστε την έξοδο ως συμβολοσειρά.
- **Windows**: μόνο η σύνταξη `system PROGRAM LIST` αποφεύγει αξιόπιστα το κέλυφος. Η pperl τρέχει μόνο σε Linux, οπότε αυτό δεν ισχύει εδώ· σημειώνεται για φορητότητα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `exec` - αντικαθιστά την τρέχουσα διεργασία αντί να κάνει fork και αναμονή· ίδιοι κανόνες επεξεργασίας ορισμάτων με την `system`
- `fork` - δημιουργεί διεργασία-παιδί χωρίς να τρέξει νέο πρόγραμμα· η `system` είναι `fork` + `exec` + `waitpid`
- `waitpid` - συλλέγει συγκεκριμένο παιδί, χρησιμοποιείται όταν διαχειρίζεστε πολλαπλά παιδιά απευθείας
- `qx` - εκτελεί εντολή και αποτυπώνει την έξοδό της ως συμβολοσειρά· αυτό που θέλετε όταν η στόχευση stdio της `system` δεν αρκεί
- `$?` - λέξη κατάστασης σφάλματος παιδιού που ορίζεται από `system`, backticks, `close` σε σωλήνα και `wait`
- `$!` - errno μετά από αποτυχημένο fork ή exec (τιμή επιστροφής -1)
- `${^CHILD_ERROR_NATIVE}` - πλήρης OS-native κατάσταση αναμονής για αποκωδικοποίηση με τις μακροεντολές `W*` του `POSIX`

I/O · Δεδομένα σταθερού μήκους

syswrite

Γράφει bytes σε filehandle με την ακατέργαστη κλήση συστήματος `write(2)`, παρακάμπτοντας το ενταμιευμένο I/O της Perl.

Η `syswrite` παραδίδει `LENGTH` bytes από το `SCALAR` απευθείας στον πυρήνα μέσω της `write(2)`, παρακάμπτοντας την ενταμίευση `PerlIO` από την οποία περνούν οι `print`,

`printf` και `write`. Ο πυρήνας ενδέχεται να δεχτεί λιγότερα bytes από όσα ζητήθηκαν - η `syswrite` επιστρέφει ό,τι όντως έγραψε, και ο καλογραμμένος κώδικας ελέγχει πάντα. Αν παραλειφθεί το LENGTH, γράφεται ολόκληρο το SCALAR. Ένα προαιρετικό OFFSET ξεκινά την εγγραφή σε θέση διαφορετική από την αρχή της συμβολοσειράς.

Σύνοψη

```
syswrite FILEHANDLE, SCALAR, LENGTH, OFFSET
syswrite FILEHANDLE, SCALAR, LENGTH
syswrite FILEHANDLE, SCALAR
```

Τι επιστρέφεται

Το πλήθος των bytes που όντως γράφτηκαν, ή `undef` σε σφάλμα (με ρυθμισμένο το `$!`). **Το επιστρεφόμενο πλήθος μπορεί να είναι μικρότερο από το LENGTH.** Αυτό είναι φυσιολογικό σε pipes, υποδοχές, και μη μπλοκαριστικά handles· δεν είναι σφάλμα. Κώδικας που χρειάζεται να παραδώσει κάθε byte πρέπει να επαναλαμβάνει σε βρόχο:

```
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;
}
```

Επιστροφή `0` είναι νόμιμη και σημαίνει «δεν γράφτηκαν bytes σε αυτή την κλήση, ξαναπροσπαθήστε» - όχι τέλος ροής. Το `undef` είναι το μόνο σήμα σφάλματος.

Καθολική κατάσταση που επηρεάζει

- `$!` - τίθεται σε σφάλμα (οποιαδήποτε επιστροφή `undef`).
- Καμία αλληλεπίδραση με τα `$,`, `$\`, ή `$|`. Η `syswrite` δεν μορφοποιεί, δεν προσαρτά διαχωριστές εγγραφής, και δεν τηρεί τη σημαία autoflush επειδή δεν υπάρχει ενταμιευτής προς εκκένωση - τα bytes πηγαίνουν κατευθείαν στον πυρήνα.
- Καμία αλληλεπίδραση με το επιλεγμένο filehandle από τη `select`. Το `FILEHANDLE` είναι υποχρεωτικό· δεν υπάρχει προεπιλογή.

Παραδείγματα

Εγγραφή συμβολοσειράς αυτολεξεί στο `STDOUT`, χωρίς προσθήκη διαχωριστών:

```
syswrite STDOUT, "hello\n";        # 6 bytes written
```

Εγγραφή εγγραφής σταθερού μεγέθους και έλεγχος για σύντομη εγγραφή:

```
my $rec = pack "A8 N", $name, $id;   # 12 bytes
my $n = syswrite $fh, $rec;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
defined $n or die "syswrite: $!";
$n == length $rec or die "short write: $n of ${\length $rec}";
```

Εγγραφή από τη μέση ενός ενταμιευτή με OFFSET. Χρήσιμο μέσα σε βρόχο εκκένωσης όπου το `$off` παρακολουθεί πόσα έχουν γίνει δεκτά μέχρι στιγμής:

```
my $buf = "ABCDEFGHIJ";
syswrite $fh, $buf, 4, 3;          # writes "DEFG"
```

Αρνητικό OFFSET μετράει από το τέλος:

```
my $buf = "ABCDEFGHIJ";
syswrite $fh, $buf, 3, -4;         # writes "GHI"
```

Εγγραφή σε μη μπλοκαριστική υποδοχή και χειρισμός του 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: $!";
    }
}
```

Οριακές περιπτώσεις

- **Μην αναμειγνύετε με ενταμιευμένο I/O στο ίδιο handle.** Η `syswrite` παρακάμπτει το PerlIO, αλλά οι `print`, `printf`, `write`, `seek`, `tell` και `eof` περνούν όλες από τα στρώματα ενταμίευσης `:perlio` και `:crlf`. Η διαπλοκή τους σε ένα handle αλλοιώνει τη σειρά εξόδου. Επιλέξτε μία λειτουργία ανά handle. Αν πρέπει να αναμείξετε, χρησιμοποιήστε τους κανόνες συμμετρίας `read` / `sysread` και να είστε προσεκτικοί.
- **Οι σύντομες εγγραφές είναι φυσιολογικές.** Τα pipes, οι υποδοχές, τα τερματικά και τα μη μπλοκαριστικά handles τακτικά δέχονται λιγότερα bytes από αυτά που προσφέρονται. Το ιδίωμα βρόχου εκκένωσης παραπάνω είναι το σωστό μοτίβο για κάθε handle όπου οι σύντομες εγγραφές έχουν σημασία.
- **Τιμή επιστροφής `0`** σημαίνει ότι ο πυρήνας δέχτηκε μηδέν bytes σε αυτή την κλήση. Δεν είναι τέλος ροής ούτε σφάλμα· επαναλάβετε.
- **Μήκος μεγαλύτερο από τα διαθέσιμα δεδομένα**: αν το LENGTH υπερβαίνει το `length(SCALAR) - OFFSET`, γράφονται μόνο τα διαθέσιμα bytes και αυτό το πλήθος επιστρέφεται.
- **Κενό SCALAR**: αν το SCALAR έχει μηδενικό μήκος, το OFFSET πρέπει να είναι `0`. Οποιαδήποτε άλλη τιμή εγείρει εξαίρεση.
- **Τα στρώματα `:utf8` και `:encoding(...)` εγείρουν εξαίρεση.** Η `syswrite` λειτουργεί σε bytes, όχι χαρακτήρες, και αρνείται οποιοδήποτε handle φέρει

στρώμα κωδικοποίησης UTF-8. Το `:encoding(...)` προσθέτει σιωπηρά το `:utf8`, οπότε απαγορεύεται εξίσου. Χρησιμοποιήστε `binmode` για να αφαιρέσετε το στρώμα, ή γράψτε μέσω `print` αντ' αυτού.

- **Χαρακτήρες πάνω από το U+00FF** σε handle λειτουργίας bytes εγείρουν επίσης εξαίρεση - δεν υπάρχει τρόπος να σειριοποιηθεί ένα codepoint μεγαλύτερο από `0xFF` ως μονό byte. Κωδικοποιήστε ρητά τη συμβολοσειρά πριν την κλήση της `syswrite`:

```
use Encode qw(encode);
syswrite $fh, encode("UTF-8", $text);
```

- **Σπασμένο pipe / κλειστή υποδοχή** εγείρει `SIGPIPE`. Η προεπιλεγμένη ενέργεια τερματίζει τη διεργασία. Εγκαταστήστε `$SIG{PIPE} = 'IGNORE'` (ή χειριστή) για να μετατρέψετε αυτό σε κανονικό σφάλμα `syswrite` με το `$!` ρυθμισμένο σε `EPIPE`.
- **Κλεισμένο filehandle** επιστρέφει `undef` με το `$!` ρυθμισμένο σε `"Bad file descriptor"`· υπό `use warnings` εκπέμπεται προειδοποίηση `syswrite() on closed filehandle`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `sysread` - το αντίστοιχο ανάγνωσης· ίδια σημασιολογία παράκαμψης ενταμιευτή και ίδιος επιφυλακτικός όρος για σύντομη ανάγνωση
- `sysopen` - ανοίγει handle σε επίπεδο `open(2)` ώστε η `syswrite` να μπορεί να επικοινωνήσει με αυτό χωρίς να παρεμβάλλονται στρώματα μετάφρασης `PerlIO`
- `sysseek` - επανατοποθετεί το handle χρησιμοποιώντας την `lseek(2)`, συνδυάζεται με την `syswrite` για I/O τυχαίας πρόσβασης σε ακατέργαστα handles
- `print` - η ενταμιευμένη αντίστοιχη που σχεδόν πάντα θέλετε αντ' αυτής· τηρεί τα `$,`, `$\` και `$|`
- `binmode` - αφαιρεί στρώμα `:utf8` ή `:encoding(...)` από handle ώστε να μπορεί να χρησιμοποιηθεί η `syswrite` σε αυτό
- `$!` - επιθεωρήστε το μετά από επιστροφή `undef` για να διακρίνετε το `EAGAIN` από πραγματική αποτυχία

I/O

tell

Επιστρέφει την τρέχουσα θέση σε bytes ενός filehandle.

Η `tell` αναφέρει πού θα γίνει η επόμενη ανάγνωση ή εγγραφή στο `FILEHANDLE`, μετρημένη σε bytes από την αρχή του αρχείου. Ζευγαρώστε την με την `seek` για να αποθηκεύσετε μια θέση τώρα και να επιστρέψετε σε αυτήν αργότερα. Αν παραλειφθεί το `FILEHANDLE`, η `tell` αναφέρει τη θέση του handle από το οποίο διαβάστηκε τελευταία - συμπεριλαμβανομένου του σιωπηρού handle μέσα σε βρόχους `while (<FH>)`.

Σύνοψη

```
tell FILEHANDLE
tell
```

Τι επιστρέφεται

Μη αρνητικό ακέραιο byte offset σε επιτυχία, `-1` σε σφάλμα. Η τιμή είναι κατάλληλη ως όρισμα `POSITION` προς την `seek` με `WHENCE` στο `0` (απόλυτη μετάβαση). Σε pipes, FIFOs, υποδοχές, και μερικές φορές στις τυπικές ροές, το υποκείμενο OS δεν έχει θέση επανατοποθέτησης και η `tell` επιστρέφει `-1`· το `$!` δεν τίθεται αναγκαστικά σε εκείνη την περίπτωση, οπότε αντιμετωπίστε το `-1` ως το αυθεντικό σήμα αποτυχίας.

```
my $pos = tell $fh;
die "not a seekable handle" if $pos < 0;
# ... read or write ...
seek $fh, $pos, 0;                  # back to where we were
```

Καθολική κατάσταση που επηρεάζει

- **Filehandle τελευταίας ανάγνωσης**: η μορφή `tell` χωρίς όρισμα χρησιμοποιεί το handle από το οποίο έγινε η πιο πρόσφατη ανάγνωση στην τρέχουσα εμβέλεια. Κάθε `readline`, `<FH>`, ή `<>` το ενημερώνει. Η ανάμιξη της `tell` χωρίς όρισμα με πολλαπλά ενεργά handles είναι πηγή απρόσμενων αποτελεσμάτων - ονομάστε ρητά το handle όταν αμφιβάλλετε.
- Το `$!` δεν είναι αξιόπιστο κανάλι σφαλμάτων εδώ. Ελέγξτε για `-1` απευθείας αντί να ελέγξετε το `$!` μετά την κλήση.

Bytes, όχι χαρακτήρες

Η `tell` επιστρέφει πάντα **byte** offset, ακόμα κι όταν το handle έχει στρώμα PerlIO προσανατολισμένο σε χαρακτήρες όπως `:encoding(UTF-8)` ή `:utf8`. Αυτό είναι σκόπιμο: η μετατροπή μέτρησης χαρακτήρων σε byte offset σε κωδικοποίηση μεταβλητού πλάτους θα απαιτούσε επανάγνωση του αρχείου από την αρχή. Ο ίδιος κανόνας ισχύει για τις `seek` και `sysseek` - όλες μιλούν με bytes.

Συνέπεια: σε αρχείο κειμένου UTF-8 δεν μπορείτε γενικά να χρησιμοποιήσετε την `tell` για να μετρήσετε χαρακτήρες. Χρησιμοποιήστε την μόνο για να σημάνετε θέση που προτίθεστε να τροφοδοτήσετε ξανά στην `seek`.

Παραδείγματα

Απομνημόνευση θέσης, ανάγνωση μπροστά, επιστροφή:

```
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
```

Καταγραφή του offset τέλους κεφαλίδας ώστε ο μεταγενέστερος κώδικας να μπορεί να επανασαρώσει το σώμα χωρίς επανανάλυση της κεφαλίδας:

```
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
```

Η μορφή χωρίς όρισμα χρησιμοποιεί το handle από το οποίο διαβάστηκε τελευταία. Μέσα σε βρόχο `while (<>)` αυτό το handle είναι το σιωπηρό ARGV:

```
while (<>) {
    print "at ", tell, ": $_";      # offset within the current file
}
```

Η `tell` σε pipe επιστρέφει `-1` επειδή τα pipes δεν είναι επανατοποθετήσιμα:

```
open my $ph, "-|", "ls" or die $!;
my $pos = tell $ph;                 # -1
```

Filehandles μέσα σε σύνθετες εκφράσεις χρειάζονται την ίδια μορφή μπλοκ όπως με την `print` - η `tell $handles[0]` είναι συντακτικό σφάλμα. Χρησιμοποιήστε:

```
my @handles = (\*STDIN, $fh);
my $pos = tell { $handles[1] };
```

Οριακές περιπτώσεις

- **Κλειστό filehandle**: επιστρέφει `-1`. Υπό `use warnings` εκπέμπεται προειδοποίηση `tell() on closed filehandle`.
- **Μορφή χωρίς όρισμα, καμία ανάγνωση ακόμα**: επιστρέφει `-1`. Το «filehandle τελευταίας ανάγνωσης» έχει νόημα μόνο μετά από πραγματική ανάγνωση.
- **Μετά από sysread / syswrite / sysseek**: μη χρησιμοποιείτε την `tell` σε handle που έχετε χειριστεί με την οικογένεια `sys*`. Αυτές οι συναρτήσεις παρακάμπτουν το buffering του PerlIO· η `tell` αναφέρει την ενταμιευμένη θέση, που θα είναι λάθος κατά τα περιεχόμενα του ενταμιευτή. Χρησιμοποιήστε `sysseek` με whence 1 και offset `0` ως την ασφαλή για `sys*` ερώτηση θέσης.
- **Τυπικές ροές**: οι `tell STDIN`, `tell STDOUT`, `tell STDERR` μπορεί να επιστρέψουν `-1` ή ένα ουσιαστικό offset ανάλογα με το αν η ροή στηρίζεται σε κανονικό αρχείο (ανακατεύθυνση κελύφους) ή σε τερματικό, pipe, ή υποδοχή.
- **Δεν υπάρχει systell**: δεν υπάρχει ξεχωριστή συνάρτηση για θέση που παρακάμπτει το buffering. Το ιδίωμα είναι:

  ```
  my $pos = sysseek $fh, 0, 1;
  ```

- **Δυαδική κατάσταση έναντι κατάστασης κειμένου σε μη-Linux πλατφόρμες**: η pperl στοχεύει μόνο το Linux, οπότε τα offsets της `tell` ταιριάζουν πάντα με τα

ωμά bytes του αρχείου. Στρώματα μετατροπής CRLF που περιπλέκουν την `tell` σε perls οικογένειας Windows δεν εφαρμόζονται εδώ.

- **Κατάσταση προσάρτησης (>>)**: μετά από `open my $fh, ">>", ...`, η αρχική `tell $fh` μπορεί να αναφέρει `0` μέχρι την πρώτη εγγραφή, επειδή ορισμένες πλατφόρμες τοποθετούν το handle στο τέλος του αρχείου μόνο σε κάθε εγγραφή. Μη βασίζεστε στην αρχική θέση· γράψτε μία φορά, μετά κάντε `tell`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `seek` - άλμα σε απόλυτο byte offset που είχε επιστρέψει προηγουμένως η `tell`
- `sysseek` - ερώτηση και άλμα θέσης για handles που χρησιμοποιούνται με `sysread` / `syswrite`, παρακάμπτοντας το buffering του PerlIO
- `telldir` - το αντίστοιχο για handle καταλόγου· ζευγαρώνει με `seekdir`
- `eof` - έλεγχος για τέλος αρχείου αντί για απόλυτη θέση
- `open` - δημιουργία του filehandle εξαρχής· το mode και τα στρώματα PerlIO καθορίζουν αν το handle είναι επανατοποθετήσιμο

I/O

telldir

Επιστρέφει την τρέχουσα θέση ανάγνωσης ενός handle καταλόγου ως αδιαφανές διακριτικό.

Η `telldir` αναφέρει από πού θα ξεκινήσει η επόμενη `readdir` στο `DIRHANDLE`. Η επιστρεφόμενη τιμή είναι ένας σελιδοδείκτης, όχι byte-offset: παραδώστε την πίσω στη `seekdir` για να συνεχίσετε την επανάληψη στην ίδια εγγραφή, ή αγνοήστε την. Ο αριθμός δεν έχει αριθμητική σημασία - δεν είναι θέση μέσα στον κατάλογο, ούτε μετρητής εγγραφών που διαβάστηκαν, ούτε συγκρίσιμος μεταξύ handles καταλόγων. Αντιμετωπίστε τον όπως θα αντιμετωπίζατε την τιμή επιστροφής της `tell` σε file handle: αδιαφανής για σας, ουσιαστική μόνο για την αντίστοιχη `seekdir`.

Σύνοψη

```
telldir DIRHANDLE
```

Τι επιστρέφεται

Έναν ακέραιο σε επιτυχία, `undef` σε αποτυχία (για παράδειγμα, αν το `DIRHANDLE` δεν είναι ανοιχτό handle καταλόγου). Ο ακέραιος εγγυάται πλήρη κύκλο μέσω της `seekdir` μόνο πάνω στο **ίδιο** ανοιχτό handle καταλόγου· το κλείσιμο και ξανα-άνοιγμα του καταλόγου ακυρώνει κάθε διακριτικό που επιστράφηκε προηγουμένως.

```
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
```

Καθολική κατάσταση που επηρεάζει

Θέτει την `$!` σε αποτυχία.

Παραδείγματα

Σελιδοδείκτης σε μια θέση και μετά επιστροφή σε αυτήν:

```
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;
```

Καταγράψτε ένα διακριτικό πριν από κάθε εγγραφή ώστε να μπορείτε να επαναφέρετε ένα βήμα πίσω:

```
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;
```

Η επιστροφή στην αρχή γίνεται καλύτερα με τη `rewinddir` παρά απομνημονεύοντας το διακριτικό από πριν την πρώτη `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;
```

Ελέγξτε ρητά για αποτυχία όταν το handle μπορεί να είναι κλειστό:

```
my $pos = telldir $dh;
defined $pos or die "telldir failed: $!";
```

Οριακές περιπτώσεις

- **Το διακριτικό είναι αδιαφανές.** Μη συγκρίνετε, προσθέτετε, αφαιρείτε, ή αποθηκεύετε διαρκώς την τιμή. Είναι έγκυρη μόνο για τη `seekdir` πάνω στο ίδιο ανοιχτό handle καταλόγου εντός της ίδιας διεργασίας.
- **Μη φορητό μεταξύ handles.** Δύο κλήσεις opendir στον ίδιο κατάλογο παράγουν ανεξάρτητα handles· ένα διακριτικό από το ένα είναι ανούσιο για το άλλο.
- **Όχι σταθερό σε `closedir` / `opendir`.** Το ξανα-άνοιγμα του καταλόγου ξεκινάει φρέσκια επανάληψη· τα παλιά διακριτικά δεν αναφέρονται πια σε τίποτα.
- **Συμπίεση καταλόγου.** Αν το υποκείμενο σύστημα αρχείων ή άλλη διεργασία τροποποιήσει τον κατάλογο (προσθέσει, αφαιρέσει, ή μετονομάσει εγγραφές) μεταξύ της `telldir` και της αντίστοιχης `seekdir`, η ανανεωμένη θέση μπορεί να παραλείψει εγγραφές, να επαναλάβει εγγραφές, ή να προσγειωθεί σε διαφορετική εγγραφή από αυτή που αρχικά σελιδοποιήθηκε. Αυτό είναι ιδιότητα του ζεύγους `telldir(3)` / `seekdir(3)` της βιβλιοθήκης C και δεν είναι κάτι που η Perl καλύπτει. Πάρτε στιγμιότυπο του καταλόγου με `readdir` σε λίστα αν χρειάζεστε σταθερότητα.
- **Κλειστό ή ποτέ-μη-ανοιγμένο handle.** Επιστρέφει `undef` και θέτει την `$!`.
- **Bareword έναντι βαθμωτού handle.** Και τα δύο `telldir DH` και `telldir $dh` δουλεύουν. Η μορφή bareword είναι καθολική σε επίπεδο πακέτου· η βαθμωτή μορφή κρατάει αναφορά σε ανώνυμο handle καταλόγου που δημιουργήθηκε από `opendir my $dh, ...`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `seekdir` - ο μόνος καταναλωτής ενός διακριτικού `telldir`· επαναφέρει το `DIRHANDLE` στη σελιδοποιημένη θέση
- `readdir` - ο επαναλήπτης του οποίου τη θέση αναφέρει η `telldir`· προωθεί τον δρομέα που παρατηρεί η `telldir`
- `rewinddir` - φθηνότερη από τη `seekdir` σε διακριτικό αρχής ροής όταν θέλετε απλώς να επαναλάβετε ξανά τον κατάλογο
- `opendir` - δημιουργεί το `DIRHANDLE` που ερωτά η `telldir`· κάθε διακριτικό `telldir` είναι δεμένο σε μία κλήση open
- `closedir` - ακυρώνει κάθε διακριτικό `telldir` για το handle
- `tell` - το ανάλογο για file handle· ίδιο συμβόλαιο αδιαφανούς διακριτικού, διαφορετική υποκείμενη κλήση συστήματος

Κλάσεις και αντικειμενοστρέφεια

tie

Δέσιμο μιας μεταβλητής σε μια κλάση ώστε κάθε πρόσβαση στη μεταβλητή να δρομολογείται μέσω των μεθόδων αυτής της κλάσης.

Η `tie` μαγεύει τη `VARIABLE` ώστε επόμενες αναγνώσεις, εγγραφές και άλλες λειτουργίες σε αυτήν να προωθούνται στις μεθόδους της tie-διεπαφής του `CLASSNAME` (`FETCH`, `STORE`, `FIRSTKEY`, `PRINT`, ...). Η `LIST` περνά στον κατασκευαστή της κλάσης (`TIESCALAR`, `TIEARRAY`, `TIEHASH` ή `TIEHANDLE`, ανάλογα με το sigil της `VARIABLE`). Ο κατασκευαστής επιστρέφει το υποκείμενο δεσμευμένο αντικείμενο, και αυτό το αντικείμενο - **όχι** η `VARIABLE` - είναι η τιμή επιστροφής της `tie`. Κρατήστε το αν θέλετε να καλέσετε επιπλέον μεθόδους στην υλοποίηση άμεσα.

Το συμβόλαιο μεθόδων που πρέπει να ικανοποιεί κάθε κλάση τεκμηριώνεται εκτενώς στο `perltie`· αυτή η σελίδα τεκμηριώνει την `tie` ως built-in.

Σύνοψη

```
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
```

Τι επιστρέφεται

Το αντικείμενο που επιστρέφει ο κατασκευαστής `TIE*` της κλάσης - μια συνηθισμένη blessed αναφορά, ίδια με οποιοδήποτε άλλο αντικείμενο. Καλέστε μεθόδους πάνω του άμεσα όταν η κλάση εκθέτει λειτουργίες που δεν αντιστοιχούν σε απλή πρόσβαση μεταβλητής:

```
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
```

Η `tie` δεν επιστρέφει την ίδια τη μεταβλητή. Κώδικας που γράφει `my $x = tie ...` και έπειτα αντιμετωπίζει το `$x` ως το βαθμωτό είναι κοινό λάθος αρχαρίων· το βαθμωτό είναι `$x` μόνο αν δεσμεύσατε επίσης το `$x`, και ο στόχος εκχώρησης είναι το δεσμευμένο αντικείμενο, όχι η μεταβλητή.

Για να ανακτήσετε το δεσμευμένο αντικείμενο από μια μεταβλητή αργότερα, χρησιμοποιήστε την `tied`. Για να διαλύσετε τη σύνδεση, χρησιμοποιήστε την `untie`.

Ποιος κατασκευαστής καλείται

Το sigil της `VARIABLE` επιλέγει τον κατασκευαστή. Η `LIST` προωθείται αυτούσια:

Μορφή VARIABLE	Κατασκευαστής που καλείται	Αναφορά διεπαφής
`$scalar`	`CLASSNAME->TIESCALAR(LIST)`	`Tie::Scalar`
`@array`	`CLASSNAME->TIEARRAY(LIST)`	`Tie::Array`
`%hash`	`CLASSNAME->TIEHASH(LIST)`	`Tie::Hash`
`*FH` (glob)	`CLASSNAME->TIEHANDLE(LIST)`	`Tie::Handle`

Τα ορίσματα του κατασκευαστή είναι συμβατικά αυτά που θα περνούσατε σε κάτι όπως η `dbm_open(3)` - ένα όνομα αρχείου, ένα mode, μια μάσκα δικαιωμάτων - αλλά οτιδήποτε αναμένει η κλάση είναι αποδεκτό.

Μέθοδοι που πρέπει να παρέχει κάθε κλάση

Η ίδια η `tie` καλεί μόνο τις `TIE*`. Κάθε άλλη λειτουργία στη μεταβλητή δρομολογείται σε μέθοδο που ονοματίζεται κατά τη λειτουργία. Δεν είναι όλες οι μέθοδοι υποχρεωτικές - μέθοδοι που λείπουν κάνουν τις αντίστοιχες λειτουργίες να αποτυγχάνουν με σφάλμα κατά τον χρόνο εκτέλεσης. Τα πλήρη μενού είναι:

Δεσμευμένο βαθμωτό (`TIESCALAR`):

```
TIESCALAR classname, LIST
FETCH     this
STORE     this, value
DESTROY   this
UNTIE     this
```

Δεσμευμένος πίνακας (`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
```

Δεσμευμένος πίνακας κατακερματισμού (`TIEHASH`):

```
TIEHASH   classname, LIST
FETCH     this, key
STORE     this, key, value
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
DELETE    this, key
CLEAR     this
EXISTS    this, key
FIRSTKEY  this
NEXTKEY   this, lastkey
SCALAR    this
DESTROY   this
UNTIE     this
```

Δεσμευμένο 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
```

Δείτε το `perltie` για την πλήρη προδιαγραφή (υπογραφές μεθόδων, πότε καλείται καθεμία, προεπιλογές που κληρονομούνται από τις βασικές κλάσεις `Tie::*`).

Παραδείγματα

Δέσιμο ενός πίνακα κατακερματισμού σε αρχείο DBM στον δίσκο και επανάληψη με `each`. Οι `keys` και `values` θα υλοποιούσαν όλο το σύνολο μονομιάς - για μεγάλα αρχεία DBM αυτό είναι ακριβό:

```
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";
}
```

Κρατήστε την τιμή επιστροφής του κατασκευαστή ώστε να μπορείτε να μιλάτε άμεσα στο δεσμευμένο αντικείμενο:

```
my $impl = tie my %cache, 'My::Cache', ttl => 60;
$cache{session} = $data;    # STORE
$impl->expire_now;           # direct method call
```

Δέσιμο βαθμωτού σε κλάση μόνο για ανάγνωση που καταγράφει κάθε πρόσβαση:

```
package Tie::Logged {
    sub TIESCALAR { my ($class, $v) = @_; bless \$v, $class }
    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"
```

Δέσιμο filehandle σε κλάση που συλλαμβάνει την έξοδο σε συμβολοσειρά:

```
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"
```

Οριακές περιπτώσεις

- **Η τιμή επιστροφής είναι το αντικείμενο, όχι η μεταβλητή.** Το `my $r = tie my %h, ...` δένει το `$r` στο δεσμευμένο αντικείμενο. Η εκχώρηση στο `$r` εκχωρεί σε βαθμωτό· δεν γράφει διαμέσου του στο `%h`.
- **Το άρθρωμα πρέπει να φορτωθεί πρώτα.** Σε αντίθεση με την dbmopen, η `tie` **δεν** κάνει `use` ή `require` το `CLASSNAME` για εσάς. Φορτώστε το ρητά:

  ```
  use My::TiedClass;
  tie my %h, 'My::TiedClass';
  ```

  Η παράλειψη αυτού είναι η τυπική αιτία του `Can't locate object method "TIEHASH" via package ...`.
- **Οι μέθοδοι που λείπουν είναι μοιραίες στο σημείο χρήσης**, όχι στη στιγμή του `tie`. Ένας δεσμευμένος πίνακας κατακερματισμού του οποίου η κλάση ορίζει `FETCH` αλλά όχι `STORE` δένεται επιτυχώς, και έπειτα εγείρει `Can't locate object method "STORE" ...` στην πρώτη εκχώρηση.
- **Η αναντιστοιχία sigil είναι σφάλμα στη στιγμή του `tie`.** Η `tie` επιλέγει τον κατασκευαστή από το sigil της μεταβλητής, οπότε μια κλάση που ορίζει μόνο `TIESCALAR` δεν μπορεί να δεθεί σε πίνακα κατακερματισμού - η αναζήτηση του `TIEHASH` αποτυγχάνει στην ίδια την κλήση `tie`.
- **Το `tie` σε μια δεσμευμένη μεταβλητή** σιωπηρά κάνει πρώτα `untie` την προηγούμενη σύνδεση, πυροδοτώντας `UNTIE` και τελικά `DESTROY` στο παλιό αντικείμενο (μόλις πέσει η τελευταία αναφορά σε αυτό).

- **Η `local` σε δεσμευμένη μεταβλητή** αποθηκεύει και αποκαθιστά τη *σύνδεση*, όχι μόνο την τιμή. Κατά την έξοδο από την εμβέλεια η μεταβλητή ξαναδένεται στο αρχικό της αντικείμενο.
- **Η διατήρηση επιπλέον αναφοράς στο δεσμευμένο αντικείμενο** το κρατά ζωντανό πέρα από την `untie`. Υπό `use warnings` αυτό παράγει προειδοποίηση `untie attempted while N inner references still exist`. Απορρίψτε την αναφορά (π.χ. `undef $obj`) πριν καλέσετε την `untie` αν θέλετε η `DESTROY` να πυροδοτηθεί άμεσα.
- **Η `tie` δεν αποθηκεύει το αντικείμενο μέσα στη μεταβλητή με κανέναν ορατό στον χρήστη τρόπο** - χρησιμοποιήστε την `tied` για να το ανακτήσετε· δεν υπάρχει σύνταξη αποαναφοράς που να το εκθέτει.
- **Απόδοση.** Κάθε λειτουργία σε δεσμευμένη μεταβλητή δρομολογείται μέσω κλήσης μεθόδου. Σε στενούς βρόχους αυτό είναι ορατά πιο αργό από απλή μεταβλητή· για μαζικές εργασίες, διαβάστε πρώτα σε απλό πίνακα ή πίνακα κατακερματισμού, λειτουργήστε, και έπειτα γράψτε πίσω.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `tied` - ανάκτηση του αντικειμένου στο οποίο είναι δεσμευμένη μια μεταβλητή αυτή τη στιγμή, ή `undef` αν δεν είναι δεσμευμένη
- `untie` - διάλυση της σύνδεσης και επιτρέπει την καταστροφή του δεσμευμένου αντικειμένου
- `bless` - η χαμηλού επιπέδου πρωτογενής που χρησιμοποιούν οι κατασκευαστές `TIE*` για να μετατρέψουν μια αναφορά σε αντικείμενο
- `dbmopen` - παλαιότερη συντόμευση για τη συγκεκριμένη περίπτωση δεσίματος πίνακα κατακερματισμού σε αρχείο DBM· φορτώνει την κλάση DBM για εσάς, κάτι που η `tie` δεν κάνει
- `ref` - επιθεώρηση της κλάσης του αντικειμένου που επιστρέφει η `tie`

Κλάσεις και αντικειμενοστρέφεια

tied

Επιστρέφει το αντικείμενο που στηρίζει μια δεμένη μεταβλητή.

Η `tied` σας παραδίδει την ίδια αναφορά που είχε αρχικά επιστρέψει η `tie` όταν η `VARIABLE` δεσμεύτηκε σε κλάση. Μέσω αυτής της αναφοράς μπορείτε να καλέσετε μεθόδους απευθείας στην υλοποίηση tie - παρακάμπτοντας τη συνήθη ανάθεση `FETCH` / `STORE` - και έτσι οι κλάσεις tie εκθέτουν επιπλέον λειτουργίες που δεν έχουν φυσική σύνταξη βαθμωτού / πίνακα / hash. Αν η `VARIABLE` δεν είναι δεμένη, η `tied` επιστρέφει `undef`.

Σύνοψη

```
tied VARIABLE
```

Τι επιστρέφεται

Μια αναφορά στο υποκείμενο αντικείμενο tie όταν η VARIABLE είναι δεμένη, και undef όταν δεν είναι. Η αναφορά είναι ακριβώς η τιμή που είχε αρχικά επιστρέψει η αντίστοιχη κλήση tie - ίδια κλάση, ίδια blessed ταυτότητα - οπότε η `ref(tied $v)` ονοματίζει την κλάση tie και οι κλήσεις μεθόδων πάνω της αναθέτονται κανονικά:

```
my $obj = tied %hash;
$obj->flush if defined $obj;
```

Χρησιμοποιήστε defined (ή λογικό έλεγχο, όταν η κλάση tie δεν κάνει ποτέ bless σε πακέτο με υπερφόρτωση ψευδούς) για να διακρίνετε μια δεμένη μεταβλητή από μια μη δεμένη πριν καλέσετε μεθόδους στο αποτέλεσμα.

Το τέχνασμα του «tie handle»

Η tied είναι ο μοναδικός υποστηριζόμενος τρόπος για να μιλήσετε απευθείας στην υλοποίηση tie. Οποιαδήποτε μέθοδος της κλάσης tie που δεν είναι ένα από τα τυπικά hooks (FETCH, STORE, EXISTS, DELETE, CLEAR, FIRSTKEY, NEXTKEY και τα αντίστοιχα για πίνακα / βαθμωτό / handle) είναι προσπελάσιμη μόνο μέσω του αντικειμένου που επιστρέφει η tied:

```
tie my %cache, 'My::LRU', size => 1024;
tied(%cache)->stats;           # class-specific method, no hash␣
↪syntax
tied(%cache)->evict_older_than(3600);
```

Αυτό είναι επίσης το μοτίβο για ενδοσκόπηση - να ρωτήσετε μια δεμένη μεταβλητή «σε ποιον είσαι δεμένη;» χωρίς να προϋποθέτετε γνώση του σημείου δέσμευσης:

```
my $class = ref tied @array;   # e.g. "Tie::File"
```

Παραδείγματα

Βασικός έλεγχος δεσμότητας:

```
tie my %h, 'My::Tied::Hash';
print defined(tied %h) ? "tied\n" : "not tied\n";   # tied
```

Κλήση κλασικής-ειδικής μεθόδου μέσω του αντικειμένου tie:

```
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
```

Ένα απλό βαθμωτό, πίνακας, hash ή filehandle που ποτέ δεν δέθηκε:

```
my $x;
print defined(tied $x) ? "yes" : "no", "\n";        # no
```

Η `tied` **δεν** ξεδένει - η μεταβλητή παραμένει δεμένη μετά την κλήση, και διαδοχικές κλήσεις επιστρέφουν την ίδια αναφορά:

```
tie my $t, 'My::Counter';
my $a = tied $t;
my $b = tied $t;
print $a == $b ? "same\n" : "different\n";          # same
```

Πλήρης γύρος μέσω της `ref` για να μάθετε την κλάση tie:

```
tie my %db, 'DB_File', 'data.db', O_RDWR | O_CREAT, 0644;
print ref tied %db, "\n";      # DB_File
```

Οριακές περιπτώσεις

- **Μη δεμένη**: επιστρέφει `undef`. Καμία προειδοποίηση, καμία εξαίρεση. Αυτός είναι ο ιδιωματικός τρόπος για να ελέγξετε αν μια μεταβλητή είναι καθόλου δεμένη - δεν υπάρχει χωριστή ενσωματωμένη `is_tied`.
- **Filehandles**: η `tied *FH` επιστρέφει το αντικείμενο για ένα δεμένο filehandle (δεσμευμένο με `tie *FH, ...`). Ένα glob που δεν δέθηκε ποτέ δίνει `undef` όπως κάθε άλλη μη δεμένη μεταβλητή.
- **Η `tied` δεν ξεδένει**. Είναι καθαρός accessor. Για να απελευθερώσετε τη δέσμευση, καλέστε την `untie`. Το να κρατάτε την αναφορά που επιστρέφει η `tied` πέρα από το σημείο όπου η μεταβλητή ξεδένεται κρατά το αντικείμενο tie ζωντανό, πράγμα σκόπιμο - έτσι ενεργοποιούνται οι προειδοποιήσεις «untie attempted while N inner references still exist» της `untie`.
- **Δεμένο στοιχείο σύνθετου τύπου**: όταν ένα μεμονωμένο στοιχείο πίνακα ή hash είναι το ίδιο δεμένο (χωριστά από τυχόν tie στον περιέκτη), η `tied $array[0]` ή `tied $hash{key}` επιστρέφει το ανά στοιχείο αντικείμενο, όχι του περιέκτη. Το tie περιέκτη και το tie στοιχείου είναι ανεξάρτητες δεσμεύσεις.
- **Οι αντιγραμμένες τιμές δεν είναι δεμένες.** Η `my $copy = $tied_scalar;` συμβολοσειροποιεί μέσω `FETCH`· η `$copy` είναι απλό βαθμωτό και η `tied $copy` επιστρέφει `undef`. Το tie είναι ιδιότητα της μεταβλητής, όχι της τιμής.
- **Οι αναφορές σε δεμένες μεταβλητές διατηρούν το tie.** Η `\%hash` όπου το `%hash` είναι δεμένο εξακολουθεί να παραπέμπει στον δεμένο περιέκτη, οπότε η `tied %{ $ref }` επιστρέφει το ίδιο αντικείμενο tie.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `tie` - δεσμεύει μια μεταβλητή σε μια κλάση· εγκαθιδρύει τη σχέση tie που διερωτάται η `tied`
- `untie` - απελευθερώνει το tie· μετά την κλήση της, η `tied` στην ίδια μεταβλητή επιστρέφει `undef`
- `ref` - δίνει το όνομα κλάσης του αντικειμένου που επιστρέφει η `tied` χωρίς ξεχωριστή κλήση μεθόδου
- `bless` - η `tied` επιστρέφει blessed αναφορά όταν η κλάση tie έκανε bless στο αντικείμενο που επέστρεψε από τον κατασκευαστή `TIE*`
- `defined` - ο τυπικός έλεγχος «είναι δεμένη αυτή η μεταβλητή;» είναι `defined tied $var`

Χρόνος

time

Επιστρέφει τον τρέχοντα πραγματικό χρόνο ως ακέραιο αριθμό δευτερολέπτων από την epoch του συστήματος.

Η `time` είναι το σημείο εκκίνησης για κάθε άλλη λειτουργία ημερομηνίας και ώρας στην Perl. Διαβάζει το ρολόι του συστήματος και επιστρέφει έναν ακέραιο - το πλήθος των non-leap δευτερολέπτων από την epoch, που σε κάθε υποστηριζόμενη πλατφόρμα είναι `00:00:00 UTC, 1 January 1970` (Unix time). Το αποτέλεσμα είναι κατάλληλο για απευθείας τροφοδότηση στις `gmtime` ή `localtime` για ανάλυση σε ημερολογιακή ημερομηνία, ή για αφαίρεση δύο αναγνώσεων ώστε να μετρηθεί ο χρόνος που πέρασε.

Σύνοψη

```
time
```

Τι επιστρέφεται

Ένας απλός ακέραιος: δευτερόλεπτα από το `1970-01-01 00:00:00 UTC`, αποκομμένα (όχι στρογγυλοποιημένα) προς το μηδέν. Καμία ακρίβεια κάτω του δευτερολέπτου - δύο κλήσεις μέσα στο ίδιο δευτερόλεπτο επιστρέφουν την ίδια τιμή. Για ακρίβεια χιλιοστών ή εκατομμυριοστών δευτερολέπτου, χρησιμοποιήστε `Time::HiRes::time`.

Η `time` δεν παίρνει ορίσματα· το σκέτο όνομα είναι ολόκληρη η κλήση. Η γραφή `time()` είναι επίσης μια χαρά και μερικές φορές πιο σαφής δίπλα σε άλλες κλήσεις συνάρτησης.

Παραδείγματα

Πρωτογενής ανάγνωση του ρολογιού:

```
my $now = time;
print "$now\n";                     # e.g. 1745424000
```

Αναγνώσιμη από άνθρωπο τοπική ημερομηνία και ώρα - η `time` συνδυάζεται με την `localtime` σε βαθμωτό περιβάλλον για μια μορφοποιημένη συμβολοσειρά:

```
print scalar localtime, "\n";          # Wed Apr 23 12:00:00 2026
```

Η ίδια ανάγνωση αναλυμένη σε πεδία για ημερολογιακή αριθμητική:

```
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;
```

Μέτρηση χρόνου που πέρασε γύρω από ένα μπλοκ εργασίας - το ιδιωματικό χρονόμετρο ακέραιων δευτερολέπτων:

```
my $t0 = time;
do_work();
my $elapsed = time - $t0;
print "took ${elapsed}s\n";
```

UTC αντί για τοπική ώρα - τροφοδοτήστε τον ίδιο ακέραιο στην `gmtime`:

```
my @utc = gmtime time;                 # same fields, UTC-based
print scalar gmtime, "\n";             # Wed Apr 23 10:00:00 2026
```

Οριακές περιπτώσεις

- **Καμία ανάλυση κάτω του δευτερολέπτου.** Διαδοχικές κλήσεις στο ίδιο δευτερόλεπτο επιστρέφουν τον ίδιο ακέραιο, οπότε το `time - $t0` για γρήγορη εργασία εκτυπώνει `0`. Χρησιμοποιήστε `Time::HiRes::time` για ανάγνωση κινητής υποδιαστολής με ακρίβεια εκατομμυριοστών:

  ```
  use Time::HiRes qw(time);
  my $t0 = time;                       # now a float
  do_work();
  printf "%.6f s\n", time - $t0;
  ```

- **Leap seconds.** Το επιστρεφόμενο πλήθος εξαιρεί τα leap seconds· είναι POSIX «δευτερόλεπτα από την epoch» όπως τα αναφέρει ο πυρήνας. Κώδικας που αφαιρεί δύο αναγνώσεις της `time` μετράει δευτερόλεπτα πραγματικού χρόνου που πέρασαν, όχι δευτερόλεπτα SI, διασχίζοντας ένα όριο leap-second.
- **Πλάτος ακεραίου.** Η τιμή είναι προσημασμένος ακέραιος επαρκώς πλατύς για τρέχουσες και μακρόχρονα μελλοντικές χρονοσφραγίδες σε όλες τις υποστηριζόμενες πλατφόρμες (64-bit). Το πρόβλημα του 2038 δεν ισχύει εδώ.
- **Ζώνη ώρας.** Η ίδια η `time` είναι αδιάφορη σε ζώνη ώρας - ο ακέραιος ονοματίζει μία μόνο στιγμή σε UTC. Οι ζώνες ώρας μπαίνουν στο παιχνίδι μόνο όταν μετατρέπετε μέσω `localtime`· η `gmtime` παραμένει σε UTC.
- **Μονοτονικότητα.** Η `time` ακολουθεί το ρολόι του συστήματος. Αν το ρολόι μετατοπιστεί προς τα πίσω (ρύθμιση NTP, χειροκίνητη αλλαγή), μια μεταγενέστερη κλήση μπορεί να επιστρέψει μικρότερη τιμή από μια προγενέστερη. Για μετρήσεις που πέρασαν και δεν πρέπει να γυρνάνε πίσω, χρησιμοποιήστε μονότονο ρολόι μέσω `Time::HiRes::clock_gettime(CLOCK_MONOTONIC)`.

- **Σε αριθμητική.** Το `time + 3600` είναι μία ώρα από τώρα· το `time - 86400` είναι 24 ώρες πριν. Τροφοδοτήστε το αποτέλεσμα στην `localtime` ή `gmtime` για να το αποδώσετε.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `gmtime` - αναλύει μια τιμή της `time` σε ημερολογιακά πεδία UTC (ή μια μορφοποιημένη συμβολοσειρά σε βαθμωτό περιβάλλον)
- `localtime` - το ίδιο, αλλά στην τοπική ζώνη ώρας· ο πιο συνηθισμένος συνεργάτης της `time`
- `sleep` - παύση για ακέραιο αριθμό δευτερολέπτων· συνδυάζεται φυσικά με την `time` για αδρό χρονοπρογραμματισμό
- `alarm` - προγραμματίζει ένα `SIGALRM` δεδομένα δευτερόλεπτα στο μέλλον
- `Time::HiRes::time` - δευτερόλεπτα κινητής υποδιαστολής με ανάλυση εκατομμυριοστών όταν η ακρίβεια ακέραιου δευτερολέπτου δεν επαρκεί

Διεργασίες · Χρόνος

times

Αναφέρει τον χρόνο CPU που κατανάλωσε αυτή η διεργασία και τα τερματισμένα παιδιά της.

Η `times` ρωτάει τον πυρήνα πόσα δευτερόλεπτα CPU σε λειτουργία χρήστη και σε λειτουργία πυρήνα έχουν χρεωθεί στην τρέχουσα διεργασία, και - χωριστά - σε παιδιά που έχουν ήδη συγκομιστεί. Η επιστροφή είναι τέσσερα δευτερόλεπτα κινητής υποδιαστολής: user, system, child-user, child-system. Απαντάει στο «πόση CPU έκαψα;», όχι στο «πόσος πραγματικός χρόνος πέρασε;» - για αυτό, δείτε την `time`.

Σύνοψη

```
my ($user, $system, $cuser, $csystem) = times;
my $user_only                         = times;    # scalar context
```

Τι επιστρέφεται

Σε περιβάλλον λίστας, τέσσερις τιμές:

- `$user` - χρόνος CPU που δαπανήθηκε σε κώδικα σε λειτουργία χρήστη αυτής της διεργασίας.
- `$system` - χρόνος CPU που δαπανήθηκε στον πυρήνα εκ μέρους αυτής της διεργασίας.
- `$cuser` - χρόνος CPU σε λειτουργία χρήστη **τερματισμένων** θυγατρικών διεργασιών.

- `$csystem` - χρόνος CPU σε λειτουργία πυρήνα **τερματισμένων** θυγατρικών διεργασιών.

Και τα τέσσερα είναι δευτερόλεπτα ως floats διπλής ακρίβειας· η ανάλυση είναι σε μικροδευτερόλεπτα σε Linux (προέρχεται από την `getrusage`).

Σε βαθμωτό περιβάλλον, η `times` επιστρέφει μόνο το `$user`.

Η `times` δεν δέχεται ορίσματα και δεν μπορεί να αποτύχει με τρόπο ορατό στον καλούντα.

Χρόνοι των παιδιών - ο κανόνας «τερματισμένα»

Τα `$cuser` και `$csystem` αθροίζουν CPU μόνο από παιδιά που έχουν **συγκομιστεί** - δηλαδή μια `wait` ή `waitpid` έχει επιτυχώς συλλέξει την κατάσταση εξόδου τους, ή συγκομίστηκαν αυτόματα επειδή το `$SIG{CHLD}` είχε τεθεί σε `'IGNORE'`. Παιδί που εξακολουθεί να εκτελείται, ή έχει εξέλθει αλλά δεν έχει ακόμη συγκομιστεί (zombie), δεν συνεισφέρει τίποτα.

Αυτό έχει σημασία όταν μετράτε μια pipeline που δημιουργεί 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
```

Μέχρι να επιστρέψει η `waitpid`, η CPU του παιδιού δεν εμφανίζεται στους χρόνους `times` του γονέα.

Παραδείγματα

Μετρήστε το κόστος CPU ενός μπλοκ εργασιών, εξαιρώντας τους ύπνους πραγματικού χρόνου ή τις αναμονές I/O:

```
my ($u0, $s0) = times;
compute_heavy();
my ($u1, $s1) = times;
printf "compute used %.3f s user + %.3f s system\n",
       $u1 - $u0, $s1 - $s0;
```

Συγκρίνετε τον χρόνο CPU με τον πραγματικό χρόνο - χαμηλή αναλογία σημαίνει ότι η διεργασία ήταν κυρίως σε αναμονή αντί να υπολογίζει:

```
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;
```

Αποδώστε έναν παράλληλο φόρτο εργασίας στα παιδιά του. Συγκομίστε κάθε παιδί πριν διαβάσετε τους χρόνους `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;
```

Βαθμωτό περιβάλλον όταν μόνο ο user time έχει σημασία - π.χ. μια γρήγορη εκτύπωση αυτο-προφίλ:

```
printf STDERR "[%.2f] reached checkpoint\n", scalar times;
```

Οριακές περιπτώσεις

- **Χωρίς ορίσματα, χωρίς πρωτότυπο για να ανησυχήσετε.** Η `times` είναι ενσωματωμένη χωρίς ορίσματα· τα `times()` και `times` αναλύονται ταυτόσημα. **Δεν** είναι ο τελεστής μεταγραμματισμού - αυτός είναι η `tr///` (ψευδώνυμο `y///`), που είναι ξεχωριστό διακριτικό.
- **Το βαθμωτό περιβάλλον έναντι περιβάλλοντος λίστας είναι κρίσιμο.** Το να γράψετε `my $t = times;` συνδέει το `$user` με το `$t`, απορρίπτοντας τις άλλες τρεις τιμές σιωπηρά. Χρησιμοποιήστε `my ($u,$s,$cu,$cs) = times;` όταν τις θέλετε όλες.
- **Τα μη συγκομισμένα παιδιά δεν μετράνε.** Δείτε τον κανόνα «τερματισμένα» παραπάνω. Η ανάγνωση των `times` πριν την `wait` δίνει `$cuser == $csystem == 0` ακόμη και όταν το παιδί καίει ορατά CPU στο `top`.
- **Τα threads μετράνε.** Σε Linux, εργασία σε threads πυρήνα στην τρέχουσα διεργασία εμφανίζεται στα `$user` / `$system` όπως η εργασία ενός μόνο thread. Τα threads σε χωριστή διεργασία δεν εμφανίζονται, μέχρι αυτή η διεργασία να εξέλθει και να συγκομιστεί.
- **Το ρολόι είναι μονότονο εντός διεργασίας, όχι πραγματικού χρόνου.** Η `times` δεν πηγαίνει ποτέ προς τα πίσω στη διάρκεια ζωής μιας διεργασίας, αλλά διαδοχικές κλήσεις `times` δεν είναι άμεσα συγκρίσιμες μεταξύ γονέα και παιδιού - η κάθε λογιστική `getrusage` είναι τοπική.
- **Ανάλυση.** Το Linux αναφέρει χτύπους ανάλυσης μικροδευτερολέπτου μέσω `getrusage`, οπότε σύντομες μετρήσεις (κάτω από millisecond) είναι θορυβώδεις αλλά όχι μηδέν. Για χρονομέτρηση πραγματικού χρόνου υψηλής ακρίβειας, χρησιμοποιήστε την `Time::HiRes::time`, όχι την `times`.
- **Υπερχείλιση.** Οι τιμές είναι floats· μπορούν καταρχήν να χάσουν ακρίβεια κάτω από το μικροδευτερόλεπτο μετά από πολύ μεγάλες εκτελέσεις (χρόνια CPU), αλλά αυτό δεν είναι ζήτημα για κανονικά προγράμματα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `time` - δευτερόλεπτα πραγματικού χρόνου από την epoch· χρησιμοποιήστε την όταν θέλετε χρόνο πραγματικού χρόνου που πέρασε, όχι χρόνο CPU
- `Time::HiRes::time` - πραγματικός χρόνος υψηλής ανάλυσης, ο συνηθισμένος εταίρος για λογιστική CPU
- `POSIX::times` - περιτύλιγμα POSIX που επιστρέφει επιπλέον τιμή πραγματικού χρόνου «χτύποι ρολογιού από την εκκίνηση»· χρησιμοποιήστε το όταν χρειάζεστε το πεδίο που πέρασε το οποίο η ενσωματωμένη παραλείπει
- `wait` - συγκομιδή παιδιού ώστε η CPU του να αρχίσει να μετράει στα `$cuser` / `$csystem`
- `waitpid` - συγκομιδή συγκεκριμένου παιδιού για τον ίδιο λόγο, χωρίς μπλοκάρισμα σε άσχετα
- `fork` - η λειτουργία της οποίας το κόστος εμφανίζεται στις στήλες παιδιού μόλις συγκομιστεί το παιδί

Βαθμωτά και συμβολοσειρές

tr///

Αντικατάσταση χαρακτήρα προς χαρακτήρα. Η `tr` σαρώνει μια συμβολοσειρά και αντικαθιστά κάθε εμφάνιση χαρακτήρα από το `SEARCHLIST` με τον αντίστοιχο θέση-προς-θέση χαρακτήρα από το `REPLACEMENTLIST`, επιστρέφοντας το πλήθος των χαρακτήρων που άγγιξε.

Δεν είναι regex. Το `SEARCHLIST` είναι σύνολο κυριολεκτικών χαρακτήρων (με συντομογραφία εύρους όπως `a-z`), όχι μοτίβο. Κανένας μετα-χαρακτήρας, κανένα `\d`, καμία παρεμβολή μεταβλητής. Ο πίνακας μετάφρασης κατασκευάζεται μία φορά, κατά τη μεταγλώττιση.

Σύνοψη

```
tr/SEARCHLIST/REPLACEMENTLIST/cdsr
y/SEARCHLIST/REPLACEMENTLIST/cdsr
$str =~ tr/abc/xyz/;
$count = ($str =~ tr/aeiou//);
```

Το `y///` είναι συνώνυμο συμβατότητας με sed για το `tr///` - πανομοιότυπη συμπεριφορά, πανομοιότυποι τροποποιητές. Δείτε `y///`.

Τι επιστρέφεται

Χωρίς τον τροποποιητή `/r`, η `tr` τροποποιεί επιτόπια τη συμβολοσειρά-στόχο και επιστρέφει τον **αριθμό χαρακτήρων που ταίριαξαν** - ή, υπό `/d`, ταιριάζοντες-ή-διεγραμμένους· υπό `/s`, ταιριάζοντες-πριν-το-squashing. Η τιμή επιστροφής είναι ακέραιος και συνήθως χρησιμοποιείται για να *μετρά* χαρακτήρες χωρίς να αλλάζει τίποτα (δείτε τον ιδιωματισμό `tr/*/*/` παρακάτω).

Με `/r`, η `tr` αφήνει την αρχική συμβολοσειρά ως έχει, κατασκευάζει μεταγραμματισμένο αντίγραφο, και επιστρέφει **αυτό το αντίγραφο**. Η επιστρεφόμενη συμβολοσειρά είναι πάντα απλή συμβολοσειρά - ακόμη και αν ο στόχος ήταν tied μεταβλητή ή blessed αντικείμενο.

Στόχος χωρίς =~: η `tr` λειτουργεί στο `$_`. Στόχος με =~: η αριστερή πλευρά πρέπει να είναι lvalue (βαθμωτό, στοιχείο πίνακα, στοιχείο hash) εκτός αν χρησιμοποιηθεί `/r`, οπότε λειτουργεί οποιαδήποτε έκφραση.

Τροποποιητές

Οι τέσσερις σημαίες στο τέλος αλλάζουν τι κάνει η `tr` με κάθε χαρακτήρα που βλέπει. Μπορούν να συνδυαστούν:

- **`c`** - συμπληρωματικό του SEARCHLIST. Το σύνολο των χαρακτήρων στους οποίους δρα γίνεται *κάθε χαρακτήρας που ΔΕΝ ανήκει στο* SEARCHLIST. Υπό `/c`, το SEARCHLIST ταξινομείται κατά codepoint μετά τη συμπλήρωση, και οποιοδήποτε REPLACEMENTLIST εφαρμόζεται σε αυτό το ταξινομημένο σύνολο - οπότε η σειρά των χαρακτήρων που γράψατε στο SEARCHLIST δεν έχει πλέον σημασία.
- **`d`** - διαγραφή. Χαρακτήρες που ταιριάζουν με το SEARCHLIST και δεν έχουν θέση στο REPLACEMENTLIST αφαιρούνται εντελώς από τον στόχο αντί να αντικατασταθούν με τον τελικό χαρακτήρα αντικατάστασης.
- **`s`** - squash. Μια ακολουθία χαρακτήρων που μεταφράζονται όλοι στον ίδιο χαρακτήρα εξόδου συμπτύσσεται σε μία μοναδική εμφάνιση αυτής της εξόδου.
- **`r`** - επιστροφή τροποποιημένου αντιγράφου. Μην αγγίζετε τον στόχο. Επιστρέψτε τη μεταγραμματισμένη συμβολοσειρά αντί για πλήθος.

Κανόνες μήκους λίστας αντικατάστασης

Η σχέση μεταξύ SEARCHLIST και REPLACEMENTLIST αλλάζει με τους τροποποιητές:

- Αν το REPLACEMENTLIST είναι **μικρότερο** από το SEARCHLIST και το `/d` **δεν** ισχύει, ο τελικός χαρακτήρας αντικατάστασης *αντιγράφεται* για να γεμίσει το κενό. Το `tr/abcd/AB/` είναι το ίδιο με το `tr/abcd/ABBB/`.
- Αν το REPLACEMENTLIST είναι μικρότερο από το SEARCHLIST **και** το `/d` ισχύει, οι περισσευούμενοι χαρακτήρες του SEARCHLIST διαγράφονται. Το `tr/abcd/AB/d` κάνει `tr/ab/AB/` συν `s/[cd]//g`.
- Αν το REPLACEMENTLIST είναι **κενό** και το `/d` δεν ισχύει, το REPLACEMENTLIST γίνεται αντίγραφο του SEARCHLIST. Αυτή είναι η μορφή μόνο μέτρησης: το `tr/abc//` μετρά a, b, c χωρίς να αλλάζει τίποτα.
- Αν το REPLACEMENTLIST είναι κενό **και** το `/d` ισχύει, κάθε χαρακτήρας στο SEARCHLIST διαγράφεται. Το `tr/abc//d` αφαιρεί όλα τα a, b, c από τον στόχο.

Εύρη χαρακτήρων

Ένα ενωτικό μεταξύ δύο χαρακτήρων ορίζει συμπεριληπτικό εύρος codepoint: το `tr/A-J/0-9/` είναι `tr/ABCDEFGHIJ/0123456789/`. Ένα ενωτικό στην αρχή, στο τέλος, ή προηγούμενο μιας ανάστροφης καθέτου λαμβάνεται κυριολεκτικά - τα `tr/-ab-/xyz/` και `tr/\-abc/wxyz/` αντιμετωπίζουν το ενωτικό ως τον κυριολεκτικό χαρακτήρα `-`.

Χρησιμοποιείτε μόνο εύρη ASCII-αλφαβήτου (ίδιας πεζότητας/κεφαλαιότητας) ή ψηφίων ASCII. Τα `a-z`, `A-Z`, `0-9` και καθαρά υποσύνολα όπως `h-k` ή `B-E` είναι φορητά και σημαίνουν ακριβώς ό,τι διαβάζονται. Εύρη ανάμικτης πεζότητας/κεφαλαιότητας, εύρη `\x`, ή εύρη που διασταυρώνουν το όριο αλφαβήτου/ψηφίων δείχνουν προφανή αλλά παράγουν διαφορετικά αποτελέσματα σε μη-ASCII πλατφόρμες - γράψτε τα αναλυτικά.

Τα φορητά εύρη Unicode χρησιμοποιούν άκρα `\N{U+...}`:

```
$str =~ tr/\N{U+20}-\N{U+7E}//d;    # delete all ASCII printables
```

Οριοθέτες

Οι τρεις οριοθέτες γύρω από τα `SEARCHLIST` και `REPLACEMENTLIST` μπορούν να είναι οποιοδήποτε ταιριαστό ζεύγος εκτυπώσιμων χαρακτήρων, όχι μόνο `/`. Τα ζεύγη αγκύλωσης (`{}`, `()`, `[]`, `<>`) απαιτούν δύο ανεξάρτητα ζεύγη, ένα για κάθε λίστα:

```
tr(aeiouy)(yuoiea);
tr[+\-*/]"ABCD";
```

Όταν ο οριοθέτης είναι μονό εισαγωγικό (`tr'...'...'`), οι λίστες αντιμετωπίζονται σχεδόν κυριολεκτικά - μόνο τα ζεύγη `\\` συμπτύσσονται. Τα ενωτικά μέσα σε `tr` με μονά εισαγωγικά **δεν** είναι προσδιοριστές εύρους· είναι κυριολεκτικά ενωτικά.

Καθολική κατάσταση που επηρεάζει

- `$_` - προεπιλεγμένος στόχος όταν δεν χρησιμοποιείται ούτε `=~` ούτε `!~`. Το `tr///` από μόνο του διαβάζει και (εκτός αν χρησιμοποιηθεί `/r`) τροποποιεί το `$_`.
- Καμία άλλη ειδική μεταβλητή. Η `tr` δεν ορίζει τα `$&`, `$1`, κ.λπ. - εκείνα ανήκουν στην αντιστοίχιση regex, και η `tr` δεν είναι regex.

Παραδείγματα

ASCII πεζά σε κεφαλαία:

```
my $name = "Perl";
$name =~ tr/a-z/A-Z/;          # $name is now "PERL"
```

Μέτρηση χωρίς αλλαγή - ο ιδιωματισμός `tr/X/X/`. Η αντικατάσταση είναι ο ίδιος χαρακτήρας, οπότε η συμβολοσειρά μένει αμετάβλητη· η τιμή επιστροφής είναι το πλήθος:

```
my $stars = "**hello**";
my $cnt   = ($stars =~ tr/*/*/);    # $cnt == 4
```

Κενό `REPLACEMENTLIST` χωρίς `/d` επίσης μετρά:

```
my $digits = ($text =~ tr/0-9//);   # count ASCII digits in $text
```

Διαγραφή χαρακτήρων με `/d`:

```
my $s = "(555) 123-4567";
$s =~ tr/0-9//cd;                # keep only digits: "5551234567"
```

Squash ακολουθιών με `/s`:

```
my $line = "hello    world";
$line =~ tr/ //s;                # "hello world"
```

Μη καταστροφικό `/r` - κατασκευή αντιγράφου, αφήνοντας ως έχει το πρωτότυπο:

```
my $host  = "example.com";
my $HOST  = $host =~ tr/a-z/A-Z/r;    # $host unchanged, $HOST
↪uppercased
```

Αλυσιδωτή σύνδεση με άλλους τελεστές `/r`:

```
my $tag = $host =~ tr/a-z/A-Z/r
               =~ s/\./_/gr;     # "EXAMPLE_COM"
```

`map` με `/r` - μετατροπή λίστας συμβολοσειρών σε μετασχηματισμένη λίστα:

```
my @upper = map tr/a-z/A-Z/r, @names;
```

Συμπίεση ολόκληρης λέξης - σύμπτυξη μη αλφαβητικών ακολουθιών σε ένα μόνο κενό:

```
$sentence =~ tr/a-zA-Z/ /cs;    # "foo, bar; BAZ!" -> "foo bar BAZ "
```

Αφαίρεση του πιο σημαντικού bit από κάθε byte:

```
$bytes =~ tr[\200-\377][\000-\177];
```

Οριακές περιπτώσεις

- **Δεν είναι regex.** Το `tr/\d/X/` ταιριάζει τους κυριολεκτικούς χαρακτήρες `\` και `d`, όχι «οποιοδήποτε ψηφίο». Οι κλάσεις χαρακτήρων, οι άγκυρες, οι ποσοδείκτες και οι οπισθαναφορές είναι όλα σιωπηρά κυριολεκτικά. Αν θέλετε regex, χρησιμοποιήστε `s///`.
- **Καμία παρεμβολή μεταβλητών.** Τα `$var` και `@var` μέσα στις λίστες είναι κυριολεκτικά `$` / `@` συν το όνομα που ακολουθεί. Ο πίνακας μεταγλωττίζεται μία φορά, όταν αναλύεται το πρόγραμμα. Για να κατασκευάσετε τις λίστες δυναμικά, τυλίξτε σε `eval`:

  ```
  eval "tr/$from/$to/";
  die $@ if $@;
  ```

- **Επαναλαμβανόμενοι χαρακτήρες στο `SEARCHLIST`** - μόνο η πρώτη αντιστοίχιση μετρά. Το `tr/AAA/XYZ/` μεταφράζει κάθε `A` σε `X`· οι θέσεις `Y` και `Z` είναι απρόσιτες.

- **Καμία επανασάρωση.** Η μεταγραμματισμένη έξοδος δεν τροφοδοτείται ξανά στον πίνακα, ακόμη και αν ο χαρακτήρας αντικατάστασης εμφανίζεται και στο SEARCHLIST. Το `tr/ox/xo/` μετατρέπει το `"oxxo"` σε `"xoox"`, όχι σε κάποιο σταθερό σημείο.
- **`!~` με `tr`** - επιστρέφει `0` αν άλλαξε οποιοσδήποτε χαρακτήρας, `1` διαφορετικά. Χρήσιμο μόνο για γρήγορους ελέγχους «δεν συνέβη τίποτα»:

```
$foo !~ tr/A/a/     # true iff no 'A' was in $foo
```

- **Απαίτηση lvalue.** Χωρίς `/r`, ο στόχος του `=~` πρέπει να είναι ανατεθέσιμος. Το `"constant" =~ tr/a/b/` είναι σφάλμα κατά τη μεταγλώττιση.
- **Το `/r` επιστρέφει πάντα απλή συμβολοσειρά.** Αν ο στόχος είναι tied βαθμωτό ή blessed αντικείμενο υπερφορτωμένο για συμβολοσειρές, το `/r` αφαιρεί τη μαγεία και σας δίνει μια φρέσκια απλή συμβολοσειρά.
- **Η `tr` δεν είναι η `tr(1)` του κελύφους.** Παρόμοια σύνταξη, επικαλυπτόμενη σημασιολογία, αλλά η πτύχωση πεζών-κεφαλαίων πέρα από το ASCII ανήκει στις `lc` / `uc` / `lcfirst` / `ucfirst` ή στο `s///` με `\U` / `\L` - όχι στην `tr`.
- **Το `/c` με `REPLACEMENTLIST` πολλαπλών χαρακτήρων** δεν είναι φορητό μεταξύ συνόλων χαρακτήρων. Υπό `/c` το σύνολο αναζήτησης ταξινομείται κατά codepoint, και αυτή η σειρά ταξινόμησης διαφέρει μεταξύ ASCII και EBCDIC. Μείνετε σε `REPLACEMENTLIST` ενός χαρακτήρα όταν συνδυάζετε με `/c`, ή χρησιμοποιήστε το `/c` μόνο με `/d` (όπου η αντικατάσταση δεν παίζει ρόλο).

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `y///` - συνώνυμο συμβατότητας με sed για το `tr///`· πανομοιότυπη συμπεριφορά, επιλέξτε όποιο διαβάζεται καλύτερα στα συμφραζόμενα
- `s///` - πλήρης αντικατάσταση regex· καταφύγετε σε αυτό όταν χρειάζεστε μοτίβα, συλλήψεις, ή παρεμβολή μεταβλητών στην πλευρά αναζήτησης
- `lc` - μετατροπή σε πεζά με γνώση Unicode· χρησιμοποιήστε αντί για `tr/A-Z/a-z/` όταν η είσοδος μπορεί να περιέχει μη-ASCII γράμματα
- `uc` - μετατροπή σε κεφαλαία με γνώση Unicode· συμπλήρωμα της `lc`
- `tr` *στο* `perlop` - η αναφορά του upstream για τη μορφή τελεστή τύπου εισαγωγικών, τους κανόνες οριοθετών, και τη φορητότητα εύρους
- `$_` - προεπιλεγμένος στόχος όταν η `tr` χρησιμοποιείται χωρίς `=~` ή `!~`

I/O

truncate

Συντομεύει (ή επεκτείνει) ένα αρχείο σε ακριβές μήκος bytes.

Η `truncate` ρυθμίζει το μέγεθος του αρχείου που είναι ανοιχτό στο `FILEHANDLE`, ή του αρχείου που κατονομάζεται από την `EXPR`, ώστε να είναι ακριβώς `LENGTH` bytes μακρύ. Τα bytes πέρα από το `LENGTH` απορρίπτονται. Η τρέχουσα θέση ανάγνωσης/εγγραφής του αρχείου **δεν** αλλάζει - ένα handle που βρίσκεται πέρα από το νέο τέλος αρχείου παραμένει εκεί, που σχεδόν ποτέ δεν είναι αυτό που θέλετε, οπότε σχεδιάστε μια `seek` πριν την επόμενη εγγραφή.

Σύνοψη

```
truncate FILEHANDLE, LENGTH
truncate EXPR, LENGTH
```

Τι επιστρέφεται

1 σε επιτυχία, `undef` σε αποτυχία με το `$!` τεθειμένο στο υποκείμενο `errno`. Μια απουσία υλοποίησης σε πλατφόρμα εγείρει εξαίρεση αντί να επιστρέψει `undef`· αυτή είναι ιδιότητα κατά την εγκατάσταση, όχι κάτι για διακλάδωση σε χρόνο εκτέλεσης.

```
truncate $fh, 0
    or die "truncate failed: $!";
```

Μορφή filehandle έναντι ονόματος αρχείου

Δύο μορφές, που επιλέγονται από αυτό που έχετε ήδη στα χέρια σας:

- **`truncate FILEHANDLE, LENGTH`** - το handle πρέπει να είναι ανοιχτό για εγγραφή (ή ανάγνωση/εγγραφή). Χρησιμοποιήστε αυτή τη μορφή όταν ο καλών κατέχει ήδη ανοιχτό handle εγγραφής· αποφεύγει έναν φρέσκο πλήρη κύκλο `open`/`close` και λειτουργεί ακόμη και μετά το unlink του αρχείου.
- **`truncate EXPR, LENGTH`** - η `EXPR` μετατρέπεται σε συμβολοσειρά ως διαδρομή. Το αρχείο ανοίγει, αποκόπτεται και κλείνει εσωτερικά. Χρησιμοποιήστε αυτή τη μορφή όταν έχετε μόνο διαδρομή και δεν χρειάζεστε το handle στη συνέχεια.

Και οι δύο μορφές απαιτούν δικαίωμα εγγραφής στον στόχο. Στη μορφή filehandle, αυτό είναι ιδιότητα της λειτουργίας ανοίγματος· στη μορφή διαδρομής, των δικαιωμάτων του filesystem.

Σημασιολογία LENGTH

- Το `LENGTH` είναι σε **bytes**, όχι χαρακτήρες. Τα στρώματα κωδικοποίησης στο filehandle είναι άσχετα - το όρισμα μήκους πάει απευθείας στο OS.
- Το `LENGTH` μικρότερο από το τρέχον μέγεθος του αρχείου απορρίπτει την ουρά.
- Το `LENGTH` ίσο με το τρέχον μέγεθος είναι no-op που εξακολουθεί να πετυχαίνει.
- Το `LENGTH` μεγαλύτερο από το τρέχον μέγεθος επεκτείνει το αρχείο με τρύπα μηδενικών bytes σε filesystems που υποστηρίζουν αραιά αρχεία, ή με κυριολεκτικά

μηδενικά bytes διαφορετικά. Το upstream `perlfunc` χαρακτηρίζει αυτή την περίπτωση απροσδιόριστη· στην πράξη συμπεριφέρεται όπως η POSIX `ftruncate(2)` σε Linux, που είναι αυτό που στοχεύει η pperl.

Καθολική κατάσταση που επηρεάζει

- Το `$!` τίθεται σε αποτυχία στο υποκείμενο `errno`.

Παραδείγματα

Αδειάστε ένα αρχείο καταγραφής χωρίς να κλείσετε το handle στο οποίο το υπόλοιπο πρόγραμμα ακόμη γράφει:

```
open my $log, "+<", "app.log" or die $!;
truncate $log, 0 or die "truncate: $!";
seek $log, 0, 0;                # rewind; see "Position is not reset
 ↪"
```

Αποκόψτε ένα αρχείο με όνομα, χωρίς να εμπλέκεται handle:

```
truncate "scratch.dat", 0
    or die "truncate scratch.dat: $!";
```

Περιστρέψτε ένα προεκχωρημένο αρχείο ενταμιευτή στο χρησιμοποιούμενο τμήμα:

```
my $used = tell $fh;            # bytes actually written
truncate $fh, $used
    or die "truncate: $!";
```

Επεκτείνετε ένα αρχείο σε σταθερό μέγεθος (αραιή τρύπα σε Linux):

```
open my $fh, ">", "image.raw" or die $!;
truncate $fh, 1024 * 1024 * 1024   # 1 GiB sparse file
    or die "truncate: $!";
```

Οριακές περιπτώσεις

- **Η θέση δεν επαναφέρεται.** Μετά την `truncate $fh, 0`, το handle εξακολουθεί να θυμάται όποιο offset είχε. Μια ακόλουθη `print $fh ...` χωρίς ενδιάμεση `seek` μπορεί να γράψει πέρα από το νέο τέλος αρχείου, αφήνοντας μηδενοσυμπληρωμένη τρύπα.
- **Handle μόνο για ανάγνωση**: η αποκοπή handle ανοιγμένου με `<` αποτυγχάνει με `EBADF` / `EINVAL` ανάλογα με την πλατφόρμα. Ανοίξτε με `+<` (ανάγνωση/εγγραφή) ή `>>` συν κατάλληλη λειτουργία αν χρειάζεται να αποκόψετε μέσω του ίδιου handle από το οποίο διαβάσατε.
- **Αρνητικό LENGTH**: ο πυρήνας το απορρίπτει και η `truncate` επιστρέφει `undef` με το `$!` τεθειμένο σε `EINVAL`.
- **Κατάλογος ως EXPR**: αποτυγχάνει με `EISDIR`. Η `truncate` είναι αυστηρά λειτουργία αρχείου.

- **Αρχείο με unlink αλλά ακόμη ανοιχτό**: η μορφή filehandle λειτουργεί καλά - το inode εξακολουθεί να υπάρχει εφόσον κάποιο handle το κρατά ανοιχτό.
- **LENGTH μεγαλύτερο από το τρέχον μέγεθος**: σε Linux αυτό επεκτείνει με αραιή τρύπα· η ανάγνωση της τρύπας αποδίδει bytes `"\0"`. Το upstream `perlfunc` το επισημαίνει ως απροσδιόριστο, οπότε φορητός κώδικας δεν θα έπρεπε να βασιστεί σε αυτό.
- **Πλατφόρμα χωρίς `ftruncate`/`truncate`**: εγείρει εξαίρεση (`The truncate function is unimplemented`). Όχι θέμα στην υποστηριζόμενη πλατφόρμα της pperl (Linux), αλλά έχει σημασία για φορητό κώδικα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `open` - παράγετε εγγράψιμο filehandle κατάλληλο για τη μορφή filehandle της `truncate`
- `seek` - επανατοποθετήστε το handle μετά την αποκοπή, πριν την εγγραφή ξανά
- `tell` - τρέχον offset, χρήσιμο ως όρισμα LENGTH όταν περικόπτετε σε «αυτό που έχω γράψει μέχρι τώρα»
- `sysopen` - χαμηλότερου επιπέδου άνοιγμα όταν χρειάζεστε ρητές σημαίες (η `O_TRUNC` αποκόπτει κατά τον χρόνο ανοίγματος· η `truncate` είναι το ισοδύναμο εκ των υστέρων)
- `unlink` - επιλέξτε αυτή αντί γι” αυτό όταν θέλετε το αρχείο να εξαφανιστεί, όχι απλώς να αδειάσει
- `$!` - `errno` σε αποτυχία

Ροή ελέγχου

try

Εκτελεί ένα μπλοκ και εκτρέπει οποιαδήποτε εξαίρεση εκπέμψει σε ένα μπλοκ `catch`, με προαιρετικό μπλοκ `finally` που εκτελείται πάντα κατά την έξοδο.

Η `try`/`catch` είναι η δομημένη σύνταξη χειρισμού εξαιρέσεων της Perl. Το μπλοκ `try` εκτελείται· αν οποιαδήποτε εντολή σε αυτό εκπέμψει (μέσω `die`, μη ληφθείσας εξαίρεσης από καλούμενο, ή σφάλματος κατά τον χρόνο εκτέλεσης), ο έλεγχος μεταφέρεται στο μπλοκ `catch` με την τιμή της εξαίρεσης δεσμευμένη στην εντός παρενθέσεων λεξιλογική μεταβλητή. Αν δεν εκπέμψει τίποτα, το μπλοκ `catch` παραλείπεται. Σε αντίθεση με την `eval`, ένα `return` μέσα σε `try` επιστρέφει από την περικλείουσα υπορουτίνα, όχι από το μπλοκ - η `try` είναι ροή ελέγχου, όχι πλαίσιο κλήσης.

Η σύνταξη πρέπει να ενεργοποιηθεί με φύλακα χαρακτηριστικού:

```
use feature 'try';
```

Σύνοψη

```
try BLOCK catch (VAR) BLOCK
try BLOCK catch (VAR) BLOCK finally BLOCK
```

Τι επιστρέφεται

Η `try` είναι εντολή, όχι έκφραση, αλλά αποδίδει την τελευταία αξιολογούμενη τιμή όποιου κλάδου εκτελέστηκε - του μπλοκ `try` σε επιτυχία, ή του μπλοκ `catch` σε αποτυχία. Όταν χρησιμοποιείται ως η τελευταία εντολή υπορουτίνας ή μπλοκ `do`, αυτή η τιμή γίνεται το αποτέλεσμα:

```
my $value = do {
    try {
        fetch_thing(@args);
    }
    catch ($e) {
        warn "fetch failed: $e";
        $DEFAULT;
    }
};
```

Μην γράφετε `return` μέσα σε `try` που χρησιμοποιείται με αυτόν τον τρόπο

- η `return` εξέρχεται από την περικλείουσα υπορουτίνα, παρακάμπτοντας την ανάθεση.

Η τελική έκφραση ενός μπλοκ `finally` απορρίπτεται· δεν μπορεί να συνεισφέρει στην τιμή.

Καθολική κατάσταση που επηρεάζει

- Το `$@` **δεν** είναι ο μηχανισμός εδώ. Η `try`/`catch` δεσμεύει την εξαίρεση στη λεξιλογική μεταβλητή που ονοματίζεται στο `catch (VAR)`, όχι στο `$@`. Ο κώδικας μέσα στο `catch` βλέπει ό,τι έτυχε να κρατά το `$@` πριν την `try`· βασιστείτε στη λεξιλογική.
- Τα `$_`, `$!`, και άλλες δυναμικές καθολικές δεν επηρεάζονται από την ίδια την κατασκευή - μόνο από όποιον κώδικα εκτελείται μέσα στα μπλοκ.

Η μεταβλητή catch

Το `catch` πρέπει να ακολουθείται αμέσως από δήλωση μεταβλητής σε παρενθέσεις. Το `my` είναι έμμεσο - γράψτε `catch ($e)`, όχι `catch (my $e)`. Η μεταβλητή είναι λεξιλογική στο μπλοκ `catch` και κρατά την εκπεμπόμενη τιμή ακριβώς όπως την παρέδωσε η `die`: συμβολοσειρά, blessed αναφορά, ή οποιοδήποτε βαθμωτό:

```
try {
    risky();
}
catch ($e) {
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    if (ref $e && $e->isa('My::Exception')) {
        $e->rethrow if $e->is_fatal;
        warn $e->message;
    }
    else {
        warn "plain die: $e";
    }
}
```

Δεν υπάρχει αποστολή σε πολλαπλά `catch` με βάση την κλάση εξαίρεσης - γράψτε εσείς την ταξινόμηση μέσα στο ένα μπλοκ `catch`.

`finally`

Ένα προαιρετικό τρίτο μπλοκ εκτελείται σε κάθε διαδρομή εξόδου από την κατασκευή `try`/`catch`: κανονική απρόσκοπτη ολοκλήρωση, εξαίρεση που πιάστηκε από το `catch`, εξαίρεση που ξανα-εκπέμφθηκε από το `catch`, ή μη τοπική μεταφορά (`return`, `last`, `next`, `redo`, `goto`) από οποιοδήποτε από τα δύο μπλοκ.

```
try {
    open_resource();
    use_resource();
}
catch ($e) {
    warn "failed: $e";
}
finally {
    close_resource();
}
```

Το `finally` είναι το σωστό μέρος για κώδικα απελευθέρωσης που πρέπει να εκτελεστεί όπως και αν εξέλθει το προστατευμένο τμήμα - file handles, κλειδώματα, επαναφορά καθολικής κατάστασης.

Ένα μπλοκ `finally` έχει περιορισμούς που τα άλλα δύο μπλοκ δεν έχουν:

- Κανένα `return`, κανένα `goto`, κανένα `last` / `next` / `redo`. Η απόπειρα κάποιου από αυτά αποτελεί σφάλμα κατά τη μεταγλώττιση ή τον χρόνο εκτέλεσης.
- Η τιμή της τελικής έκφρασης αγνοείται - το `finally` δεν μπορεί να συνεισφέρει στην αποδιδόμενη τιμή της κατασκευής.

`finally` remains **experimental** in Perl 5.44 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.

Αντιπαραβολή με `eval { ... }` / `$@`

Το παλαιότερο μοτίβο εξακολουθεί να υποστηρίζεται, αλλά η `try`/`catch` διορθώνει τρεις μακροχρόνιες παγίδες:

Παγίδα	`eval` / `$@`	`try` / `catch`
Επικάλυψη του `$@` μεταξύ της εξόδου από το `eval` και του ελέγχου `if ($@)`	πραγματικός κίνδυνος, χρειάζεται χορό `local $@`	η εξαίρεση δεσμεύεται σε φρέσκια λεξιλογική
`return` μέσα στο προστατευμένο μπλοκ	επιστρέφει από το `eval`, όχι από την υπορουτίνα	επιστρέφει από την περικλείουσα υπορουτίνα
Διάκριση μεταξύ «κανένα σφάλμα» και «το σφάλμα ήταν ψευδές»	το `$@` μπορεί να είναι ψευδές-αλλά-ορισμένο	το `catch` εκτελείται μόνο σε πραγματική εκπομπή

Χρησιμοποιήστε `try` για νέο κώδικα. Καταφύγετε σε `eval EXPR` μόνο όταν πραγματικά χρειάζεστε τη μεταγλώττιση κατά τον χρόνο εκτέλεσης του string-`eval`.

Παραδείγματα

Παγίδευση εξαίρεσης και καταγραφή της:

```
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";
```

Χρήση της `try` ως έκφρασης μέσα σε `do` για επιλογή εναλλακτικής λύσης:

```
use feature 'try';

my $config = do {
    try { load_config($path) }
    catch ($e) { warn "using defaults: $e"; default_config() }
};
```

Άνευ όρων απελευθέρωση πόρου με `finally`:

```
use feature 'try';
no warnings 'experimental::try';   # finally is still experimental

my $lock = acquire_lock();
try {
    do_critical_work();
}
catch ($e) {
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    warn "critical work failed: $e";
}
finally {
    release_lock($lock);
}
```

Ταξινόμηση εξαιρέσεων μέσα σε ένα μόνο μπλοκ `catch`:

```
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";
    }
}
```

Το `return` μέσα σε `try` επιστρέφει από την περικλείουσα υπορουτίνα - όχι μόνο από το μπλοκ:

```
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
}
```

Οριακές περιπτώσεις

- **Απαιτείται φύλακας χαρακτηριστικού.** Χωρίς `use feature 'try'` (ή ένα bundle `use v5.34` ή νεότερο που το ενεργοποιεί), το `try` είναι απλός προσδιοριστής και η κατασκευή είναι συντακτικό σφάλμα.
- **Το `catch` είναι υποχρεωτικό.** Το `try { ... }` από μόνο του δεν είναι νόμιμο - κάθε `try` χρειάζεται `catch`. Χρησιμοποιήστε `defer` ή `finally` αν θέλετε

καθαρισμό χωρίς χειρισμό εξαιρέσεων, σε συνδυασμό με ένα τετριμμένο `catch ($e) { die $e }` rethrow αν θέλετε επίσης να αφήσετε την εξαίρεση να διαδοθεί.

- **Επανεκπομπή.** Μέσα στο `catch`, το `die $e` ξανα-εκπέμπει. Δεν υπάρχει έμμεση επανεκπομπή αν απλώς πέσετε στο τέλος του `catch`· κάνοντάς το αυτό **καταπίνετε** την εξαίρεση. Αποφασίστε σκόπιμα.
- **Το $@ δεν εκκαθαρίζεται** με την είσοδο στο `try` και δεν ορίζεται από μια εξαίρεση που πιάστηκε. Μη διαβάζετε το `$@` μέσα στο `catch` περιμένοντας την εκπεμπόμενη τιμή - διαβάστε τη λεξιλογική.
- **Η `caller()` δεν βλέπει την `try`.** Όπως ο `while` ή ο `foreach`, η `try` δεν είναι πλαίσιο κλήσης. Η `caller` την παρακάμπτει, που είναι σωστό: αφού η `return` διαδίδεται μέσα από την `try`, η `try` δεν έχει ανεξάρτητη σημασιολογία επιστροφής που να αξίζει να εκτεθεί στην ενδοσκόπηση.
- **Οι έλεγχοι βρόχου περνούν διαπεραστικά.** Τα `last`, `next` και `redo` μέσα σε `try` ή `catch` δρουν στον περικλείοντα βρόχο, όχι στην κατασκευή `try`. Ένα μπλοκ `finally` θα εκτελεστεί ακόμη και πριν ολοκληρωθεί η μεταφορά.
- **Οι περιορισμοί του `finally` επιβάλλονται.** Τα `return`, `goto`, και οι έλεγχοι βρόχου μέσα σε `finally` είναι σφάλματα. Αν χρειάζεστε υπό συνθήκη καθαρισμό που μπορεί να ματαιωθεί, κάντε το στο `catch` ή έξω από την κατασκευή.
- **Κανένα συντακτικό σιρόπι έκφρασης `try` πέρα από την απόδοση τελευταίας εντολής.** Η `try` δεν είναι γενική έκφραση· μπορεί να παράγει τιμή μόνο όταν είναι η τελευταία εντολή μπλοκ που αποδίδει τιμή (`do`, υπορουτίνα). Το `my $x = try { ... } catch ($e) { ... };` είναι συντακτικό σφάλμα.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

- Τα `try` και `catch` δεν είναι πειραματικά (από την 5.40) και δεν εκπέμπουν προειδοποιήσεις.
- `finally` is still experimental in 5.44 and emits `experimental::try` warnings when used; silence with `no warnings 'experimental::try'`.

Δείτε επίσης

- `catch` - το υποχρεωτικό δεύτερο μπλοκ· λαμβάνει την εξαίρεση σε μια λεξιλογική μεταβλητή εντός παρενθέσεων
- `finally` - optional third block for unconditional cleanup; still experimental in 5.44
- `die` - η πλευρά εκπομπής της κατασκευής· οποιοδήποτε βαθμωτό (συμβολοσειρά, blessed αναφορά, αντικείμενο) μπορεί να είναι η εκπεμπόμενη τιμή
- `eval` - το παλαιότερο μοτίβο block-eval / `$@`, ακόμη απαραίτητο για string `eval`· προτιμήστε `try` για αμιγή χειρισμό εξαιρέσεων
- `defer` - άνευ όρων καθαρισμός χωρίς περιβάλλον `try`/`catch`· η σημασιολογική βάση πάνω στην οποία ορίζεται το `finally`
- `return` - μέσα σε `try` επιστρέφει από την περικλείουσα υπορουτίνα, όχι από το μπλοκ· μέσα σε `finally` είναι σφάλμα

Βαθμωτά και συμβολοσειρές

uc

Επιστροφή κεφαλαιοποιημένου αντιγράφου μιας συμβολοσειράς.

Η uc διασχίζει την EXPR χαρακτήρα-χαρακτήρα και επιστρέφει νέα συμβολοσειρά στην οποία κάθε χαρακτήρας με πεζά/κεφαλαία έχει αντικατασταθεί από το κεφαλαίο αντίστοιχό του. Χαρακτήρες που δεν έχουν αντιστοίχιση κεφαλαίου - ψηφία, σημεία στίξης, ήδη κεφαλαία γράμματα, σύμβολα - περνούν αμετάβλητοι. Αν η EXPR παραλειφθεί, η uc λειτουργεί στην $_. Η είσοδος δεν τροποποιείται ποτέ· η uc επιστρέφει πάντα νέα συμβολοσειρά.

Η uc είναι η εσωτερική συνάρτηση που υλοποιεί τη διαφυγή \U σε συμβολοσειρές με διπλά εισαγωγικά, οπότε τα "\Ufoo\E" και uc("foo") παράγουν το ίδιο αποτέλεσμα.

Σύνοψη

```
uc EXPR
uc
```

Τι επιστρέφεται

Μια νέα βαθμωτή συμβολοσειρά που περιέχει την κεφαλαιοποιημένη εκδοχή της EXPR. Το μήκος σε χαρακτήρες είναι αμετάβλητο για όλα τα γράμματα με πεζά/κεφαλαία στο Basic Multilingual Plane· λίγοι χαρακτήρες αναπτύσσονται σε μακρύτερη μορφή υπό τους κανόνες Unicode (δείτε *Οριακές περιπτώσεις*), οπότε το μήκος σε bytes και το μήκος σε χαρακτήρες του αποτελέσματος μπορούν να αυξηθούν αμφότερα.

```
my $s = uc("Perl is GREAT");   # "PERL IS GREAT"
```

Καθολική κατάσταση που επηρεάζει

- $_ - χρησιμοποιείται ως έμμεσο όρισμα όταν παραλείπεται η EXPR.
- Η τοπικιοποίηση LC_CTYPE - συμβουλεύεται όταν το use locale είναι ενεργό. Η uc αλλιώς αγνοεί το περιβάλλον.
- Τα use bytes / use feature 'unicode_strings' / use locale - τα λεξιλογικά pragmas στην εμβέλεια στο **σημείο κλήσης** επιλέγουν ποιο από τα παρακάτω σύνολα κανόνων πεζών/κεφαλαίων εφαρμόζεται.

Ποιοι κανόνες ισχύουν

Η uc επιλέγει ένα από τρία σύνολα κανόνων με βάση τα pragmas που ισχύουν και την εσωτερική αναπαράσταση της συμβολοσειράς. Η λογική ταιριάζει ακριβώς με την lc· ο πίνακας αντιστοίχισης είναι απλώς αντίστροφος.

- **Το use bytes ισχύει** - κανόνες ASCII. Αλλάζουν μόνο τα a-z, σε A-Z αντίστοιχα. Κάθε byte εκτός του 0x61..0x7A περνά αμετάβλητο, ανεξάρτητα από το τι θα σήμαινε ως Latin-1 ή UTF-8. Χρησιμοποιήστε το μόνο όταν έχετε σκόπιμα δηλώσει αποχώρηση από τον χειρισμό Unicode.

- **Το `use locale για το LC_CTYPE ισχύει`** - η τρέχουσα τοπικιοποίηση διέπει τα code points κάτω από 256· οι κανόνες Unicode διέπουν τα υπόλοιπα (τα τελευταία προσβάσιμα μόνο αν η συμβολοσειρά έχει ορισμένη τη σημαία UTF-8). Από την v5.20 και μετά, μια τοπικιοποίηση UTF-8 δίνει πλήρεις κανόνες Unicode για ολόκληρη τη συμβολοσειρά. Υπό μη-UTF-8 τοπικιοποιήσεις, αλλαγές πεζών/κεφαλαίων που διασχίζουν το όριο 255/256 δεν είναι καλά ορισμένες και, από την v5.22, πυροδοτούν προειδοποίηση τοπικιοποίησης. Δείτε το `perllocale`.
- **Η συμβολοσειρά έχει ορισμένη τη σημαία UTF-8** - κανόνες Unicode.
- **Το `use feature 'unicode_strings'` ή το `use locale ':not_characters'` ισχύει** - κανόνες Unicode, ακόμη και για συμβολοσειρές bytes.
- **Αλλιώς** - κανόνες ASCII. Οτιδήποτε εκτός του `a-z` περνά αμετάβλητο. Αυτή είναι η ιστορική προεπιλογή για συμβολοσειρές που δεν έχουν ούτε τη σημαία UTF-8 ούτε λεξιλογικό pragma για συμμετοχή στο Unicode.

Το πρακτικό συμπέρασμα: αν θέλετε προβλέψιμη κεφαλαιοποίηση Unicode αυθαίρετης εισόδου, βάλτε `use feature 'unicode_strings';` (ή μια σύγχρονη `use v5.12;` ή ανώτερη) στην κορυφή του αρχείου και πάψτε να ανησυχείτε για το ποια αναπαράσταση τυχαίνει να έχει η συμβολοσειρά.

Παραδείγματα

Βασικό ASCII:

```
my $s = uc("hello");            # "HELLO"
```

Όρισμα που παραλείπεται λειτουργεί στην `$_`:

```
for ("alpha", "beta") {
    print uc, "\n";             # "ALPHA\n", "BETA\n"
}
```

Το `\U` σε συμβολοσειρά με διπλά εισαγωγικά είναι η ίδια λειτουργία:

```
my $s = "Perl is \Ugreat\E";    # "Perl is GREAT"
```

Κεφαλαιοποίηση Unicode υπό `unicode_strings`:

```
use feature 'unicode_strings';
my $s = uc("straße");           # "STRASSE"
```

Το γερμανικό sharp s δεν έχει μονοχαρακτηρικό κεφαλαίο στο Unicode· αντιστοιχίζεται στη διχαρακτηρική ακολουθία SS. Η επιστρεφόμενη συμβολοσειρά είναι ως εκ τούτου ένας χαρακτήρας μακρύτερη από την είσοδο.

Τα ελληνικά πεζά γράμματα κεφαλαιοποιούνται σε κεφαλαία, με τους αμετάβλητους χαρακτήρες να περνούν:

```
use feature 'unicode_strings';
my $s = uc("Καλημέρα, world!"); # "ΚΑΛΗΜΈΡΑ, WORLD!"
```

Η λειτουργία byte περιορίζει τη λειτουργία στο ASCII, κάτι που περιστασιακά είναι το επιθυμητό για tokens πρωτοκόλλων:

```
use bytes;
my $s = uc("héllo");           # "HéLLO" - only the ASCII letters␣
↪change
```

Οριακές περιπτώσεις

- **undef**: η `uc(undef)` επιστρέφει `""` και πυροδοτεί προειδοποίηση `uninitialized` υπό `use warnings`. Προστατέψτε εισόδους που μπορεί να είναι undefined αν αυτή η προειδοποίηση έχει σημασία.
- **Κενή συμβολοσειρά**: η `uc("")` επιστρέφει `""`. Καμία προειδοποίηση.
- **Αντιστοιχίσεις ένα-προς-πολλά**: Μικρός αριθμός χαρακτήρων αναπτύσσεται σε πολλαπλούς χαρακτήρες σε κεφαλαία - η πιο γνωστή είναι U+00DF (ß) → SS, και οι ελληνικές αναπτύξεις U+0390 / U+03B0. Η συμβολοσειρά αποτελέσματος είναι μακρύτερη από την είσοδο σε αυτές τις περιπτώσεις. Μην υποθέτετε `length(uc($s)) == length($s)`.
- **Titlecase έναντι uppercase**: η uc **δεν** εφαρμόζει titlecase στο πρώτο γράμμα. Το titlecase είναι διακριτή κατηγορία Unicode· η `ucfirst` το εφαρμόζει στον πρώτο χαρακτήρα και αφήνει τα υπόλοιπα ως έχουν.
- **Χαρακτήρες χωρίς πεζά/κεφαλαία**: Ψηφία, σημεία στίξης, λευκοί χαρακτήρες, σύμβολα και ιδεογράμματα CJK επιστρέφονται αμετάβλητα. Η `uc("42!")` είναι `"42!"`.
- **Η επιτόπια ενημέρωση δεν υπάρχει**: η uc επιστρέφει νέα τιμή· δεν τροποποιεί ποτέ το όρισμά της. Για κεφαλαιοποίηση επιτόπου, εκχωρήστε πίσω: `$s = uc $s`.
- **Περιβάλλον**: η uc είναι πάντα βαθμωτή. Η κλήση της σε περιβάλλον λίστας παράγει παρ” όλα αυτά μία μόνο συμβολοσειρά.
- **Εκπλήξεις συμβολοσειράς bytes / σημαίας Unicode**: Συμβολοσειρά κατασκευασμένη από `read` χωρίς layer `:utf8` δεν έχει σημαία UTF-8, οπότε η uc χωρίς `use feature 'unicode_strings'` θα εφαρμόσει **κανόνες ASCII** ακόμη και αν τα bytes είναι έγκυρο UTF-8. Αποκωδικοποιήστε πρώτα, ή ενεργοποιήστε το pragma, για να πάρετε την κεφαλαιοποίηση που φαίνεται ότι θα έπρεπε να πάρουν τα bytes.
- **Το `use locale` και μη-UTF-8 τοπικιοποιήσεις (v5.22+)**: Αλλαγές πεζών/κεφαλαίων που θα διέσχιζαν το όριο 255/256 εκπέμπουν προειδοποίηση `Can't do uc("…") on non-UTF-8 locale` και επιστρέφουν τον χαρακτήρα εισόδου αμετάβλητο.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `lc` - η αντίστροφη λειτουργία· μετατρέπει σε πεζά κάθε χαρακτήρα με πεζά/κεφαλαία χρησιμοποιώντας την ίδια επιλογή κανόνων

- `ucfirst` - κεφαλαιοποιεί (titlecase) μόνο τον πρώτο χαρακτήρα της συμβολοσειράς· για ένα Unicode-aware μοτίβο `ucfirst . lc $rest` όταν κάνετε titlecase μια ολόκληρη λέξη
- `lcfirst` - μετατρέπει σε πεζό μόνο τον πρώτο χαρακτήρα
- `fc` - Unicode casefolding για σύγκριση χωρίς διάκριση πεζών-κεφαλαίων· χρησιμοποιήστε αυτή, όχι την `uc` ή την `lc`, όταν συγκρίνετε συμβολοσειρές για ισοδυναμία
- `sprintf` - η μετατροπή `%s` δεν κάνει casefold· συνδυάστε με την `uc` όταν ένα format χρειάζεται κεφαλαιοποιημένο πεδίο
- `tr///` - το ιδίωμα `tr/a-z/A-Z/` κεφαλαιοποιεί μόνο ASCII και είναι ταχύτερο από την `uc` για εγγυημένα-ASCII είσοδο

Βαθμωτά και συμβολοσειρές

ucfirst

Επιστρέφει αντίγραφο μιας συμβολοσειράς με τον πρώτο χαρακτήρα της σε titlecase.

Η `ucfirst` παίρνει μια συμβολοσειρά, μετατρέπει τον **πρώτο της χαρακτήρα** στη μορφή **titlecase** του Unicode, και επιστρέφει το αποτέλεσμα. Κάθε άλλος χαρακτήρας αφήνεται ανέγγιχτος. Το αρχικό δεν τροποποιείται ποτέ. Αν το όρισμα παραλειφθεί, η `ucfirst` λειτουργεί επί της `$_`. Το titlecase συνήθως συμπίπτει με το uppercase - το `"a"` γίνεται `"A"` - αλλά διαφέρει για μικρό αριθμό δίψηφων (digraphs) σε γραφές που διακρίνουν τα δύο (δείτε *Οριακές περιπτώσεις*). Οι ίδιοι κανόνες pragma, locale και σημαίας UTF-8 που διέπουν τις `lc` και `uc` διέπουν και την `ucfirst`.

Σύνοψη

```
ucfirst EXPR
ucfirst
```

Τι επιστρέφεται

Μια νέα συμβολοσειρά **ίδιου μήκους σε χαρακτήρες** με την είσοδο - αλλάζει μόνο ο πρώτος χαρακτήρας. Η τιμή επιστροφής είναι πάντοτε φρέσκο βαθμωτό· η τροποποίησή της δεν επηρεάζει το όρισμα, και η τροποποίηση του ορίσματος εκ των υστέρων δεν επηρεάζει την επιστρεφόμενη συμβολοσειρά. Το μήκος σε bytes μπορεί να αλλάξει κατά μερικά bytes αν η μορφή titlecase του πρώτου χαρακτήρα έχει διαφορετική κωδικοποίηση UTF-8 από την αρχική της μορφή.

```
my $str = ucfirst("hello world!");   # "Hello world!"
```

Καθολική κατάσταση που επηρεάζει

- Διαβάζει την `$_` όταν καλείται χωρίς όρισμα.
- Τηρεί το `use locale` για το `LC_CTYPE` - όταν ισχύει το locale, ο πίνακας uppercasing του τρέχοντος locale εφαρμόζεται σε πρώτο χαρακτήρα του οποίου το σημείο κώδικα είναι κάτω από 256.

- Τηρεί το `use bytes` - υπό `use bytes` αλλάζει μόνο ένα προπορευόμενο a-z, σε A-Z.
- Τηρεί το `use feature 'unicode_strings'` - επιβάλλει κανόνες Unicode ανεξάρτητα από τη σημαία UTF-8 στην είσοδο.

Ποιοι κανόνες πεζών-κεφαλαίων εφαρμόζονται

Η `ucfirst` επιλέγει ένα από πέντε σύνολα κανόνων για τον **πρώτο χαρακτήρα**, με σειρά προτεραιότητας. Νικά αυτό του οποίου η συνθήκη ισχύει πρώτη. Οι μετέπειτα χαρακτήρες περνούν πάντοτε αμετάβλητοι, ανεξάρτητα από το ποιο σύνολο κανόνων εφαρμόστηκε στον πρώτο.

1. **Σε ισχύ το `use bytes`** - κανόνες ASCII. Ένα προπορευόμενο a-z αντιστοιχίζεται σε A-Z· κάθε άλλο προπορευόμενο byte αφήνεται ως έχει, περιλαμβανομένων bytes στο εύρος 128-255 που διαφορετικά θα ήταν γράμματα Latin-1.
2. **Σε ισχύ το `use locale για το LC_CTYPE`** - οι πίνακες του τρέχοντος locale εφαρμόζονται σε πρώτο χαρακτήρα κάτω από το σημείο κώδικα 256· πρώτος χαρακτήρας στο 256 ή πάνω (προσβάσιμος μόνο όταν η συμβολοσειρά φέρει ήδη τη σημαία UTF-8) χρησιμοποιεί κανόνες Unicode. Από την Perl 5.20 και ύστερα, ένα UTF-8 locale χρησιμοποιεί πλήρως κανόνες Unicode καθ” όλη τη διαδρομή.
3. **Το όρισμα έχει τη σημαία UTF-8 ορισμένη** - οι κανόνες titlecase του Unicode εφαρμόζονται στον πρώτο χαρακτήρα.
4. **Σε ισχύ το `use feature 'unicode_strings'` ή το `use locale ':not_characters'`** - οι κανόνες titlecase του Unicode εφαρμόζονται στον πρώτο χαρακτήρα, ανεξάρτητα από τη σημαία UTF-8.
5. **Διαφορετικά** - κανόνες ASCII. Προπορευόμενος χαρακτήρας εκτός a-z επιστρέφεται αμετάβλητος, περιλαμβανομένων πεζών γραμμάτων Latin-1 όπως à-þ.

Το συμπέρασμα: αν θέλετε προβλέψιμη συμπεριφορά Unicode σε κάθε συμβολοσειρά ανεξάρτητα από τον τρόπο κατασκευής της, ενεργοποιήστε το `use feature 'unicode_strings'` (ή το `use v5.12` και άνω, που το ενεργοποιεί για εσάς).

Titlecase έναντι uppercase

Για τη συντριπτική πλειονότητα των χαρακτήρων, το titlecase και το uppercase είναι το ίδιο σημείο κώδικα - το `ucfirst("abc")` και το `uc(substr("abc", 0, 1)) . substr("abc", 1)` παράγουν ταυτόσημα αποτελέσματα. Διαφέρουν μόνο για μια χούφτα χαρακτήρες Unicode των οποίων η μορφή uppercase είναι ακολουθία δύο κεφαλαίων γραμμάτων αλλά η μορφή titlecase είναι ένα μοναδικό κεφαλαίο ακολουθούμενο από πεζό. Το τυπικό παράδειγμα είναι το λατινικό δίψηφο dz (U+01F3):

- το uppercase είναι DZ (U+01F1) - το `"DZ"` αποδιδόμενο ως ένα κεφαλαίο γλυφικό.
- το titlecase είναι Dz (U+01F2) - `"Dz"`, κεφαλαίο D ακολουθούμενο από μικρό z.

Η `ucfirst` επιστρέφει titlecase, που είναι η σωστή μορφή όταν ο χαρακτήρας στέκεται στην αρχή κεφαλαιοποιημένης λέξης. Η uc στον ίδιο πρώτο χαρακτήρα θα επέστρεφε τη μορφή δύο κεφαλαίων, η οποία διαβάζεται σαν να ήταν ολόκληρη η λέξη με κεφαλαία.

Παραδείγματα

Βασική μετατροπή πρώτου χαρακτήρα σε κεφαλαίο σε ASCII:

```
my $s = ucfirst("hello, world!");     # "Hello, world!"
```

Εξ ορισμού επί της $_:

```
for ("foo", "bar", "baz") {
    print ucfirst, "\n";              # Foo / Bar / Baz
}
```

Οι χαρακτήρες Unicode μετατρέπονται σε titlecase μόνο όταν το επιτρέπουν οι κανόνες:

```
use feature 'unicode_strings';
my $s = ucfirst("äpfel");             # "Äpfel"
```

Χωρίς το unicode_strings και χωρίς τη σημαία UTF-8 στην είσοδο, ένας μη-ASCII πρώτος χαρακτήρας περνά αμετάβλητος:

```
my $bytes = "\xE4pfel";               # ä as a single Latin-1 byte
my $cap   = ucfirst $bytes;           # unchanged: "\xE4pfel"
```

Η ucfirst είναι αυτό που υποστηρίζει τη διαφυγή \u μέσα σε συμβολοσειρές με διπλά εισαγωγικά:

```
my $str = "\uperl\E is great";        # "Perl is great"
```

Κεφαλαιοποιήστε κάθε λέξη - συνδυάστε με τις split, map και join:

```
my $title = join " ", map { ucfirst lc $_ } split / /, "HELLO world
 ↪";
# "Hello World"
```

Μετατροπή σε titlecase ενός δίψηφου που διακρίνει το titlecase από το uppercase:

```
use feature 'unicode_strings';
my $t = ucfirst("\x{01F3}xyz");        # "\x{01F2}xyz" - Dz, not DZ
```

Οριακές περιπτώσεις

- **Όρισμα undef** εγείρει προειδοποίηση uninitialized υπό use warnings και επιστρέφει την κενή συμβολοσειρά.
- **Κενή συμβολοσειρά** επιστρέφει την κενή συμβολοσειρά - δεν υπάρχει πρώτος χαρακτήρας για αλλαγή.
- **Συμβολοσειρά ενός χαρακτήρα** συμπεριφέρεται ακριβώς όπως θα συμπεριφερόταν η uc για εκείνον τον ένα χαρακτήρα υπό το ίδιο σύνολο κανόνων, εκτός από τις περιπτώσεις δίψηφων που περιγράφονται παραπάνω όπου titlecase και uppercase διαφέρουν.
- **Οι αριθμοί μετατρέπονται πρώτα σε συμβολοσειρά**: η ucfirst(42) επιστρέφει "42" - το προπορευόμενο "4" δεν είναι χαρακτήρας με πεζά/κεφαλαία, οπότε δεν

αλλάζει τίποτα.

- **Οι προπορευόμενοι λευκοί χαρακτήρες ή σημεία στίξης δεν** παραλείπονται. Η `ucfirst(" hello")` επιστρέφει `" hello"` - ο πρώτος χαρακτήρας είναι το κενό, που δεν έχει μορφή titlecase. Αν χρειάζεται να κεφαλαιοποιήσετε το πρώτο *γράμμα*, αφαιρέστε πρώτα τους προπορευόμενους μη-γράμματα χαρακτήρες ή χρησιμοποιήστε regex (`s/\b(\w)/\u$1/`).
- Το **`LATIN SMALL LETTER SHARP S` (U+00DF, ß)** μετατρέπεται σε titlecase σε `"Ss"` υπό πλήρεις κανόνες Unicode - αλλαγή μήκους που αντιφάσκει με τη συνηθισμένη εγγύηση «ίδιο μήκος σε χαρακτήρες». Υπό `use locale` σε μη-UTF-8 locale, η Perl αφήνει τον χαρακτήρα αμετάβλητο αντί να μαντέψει, και από την Perl 5.22 και ύστερα αυτό εγείρει προειδοποίηση locale.
- **Το locale και η σημαία UTF-8 αλληλεπιδρούν**: υπό `use locale` σε μη-UTF-8 locale, ένας πρώτος χαρακτήρας κάτω από το 256 ακολουθεί το locale ενώ ένας πρώτος χαρακτήρας 256 ή πάνω ακολουθεί το Unicode. Στην πράξη η `ucfirst` αγγίζει πάντοτε μόνο έναν χαρακτήρα, οπότε η αλληλεπίδραση είναι λεπτότερη από ό,τι στις `lc` / `uc`, αλλά εξακολουθεί να είναι παρατηρήσιμη σε συμβολοσειρές που ξεκινούν με χαρακτήρες συμπληρωματικού επιπέδου.
- **Οι μεταβλητές με tie** δέχονται την κλήση του `FETCH` τους μία φορά. Η `ucfirst` δεν τροποποιεί την τιμή με tie· επιστρέφει απλό (χωρίς tie) βαθμωτό.
- **Η `ucfirst` είναι μοναδιαίος ονομασμένος τελεστής**, όχι τελεστής λίστας. Η `ucfirst $a, $b` αναλύεται ως `(ucfirst $a), $b` - μετασχηματίζεται μόνο η `$a`, και ο τελεστής κόμμα απορρίπτει το αποτέλεσμα. Χρησιμοποιήστε παρενθέσεις ή `map` όταν εννοείτε να μετατρέψετε σε titlecase πολλαπλές συμβολοσειρές:

```
my @capped = map { ucfirst } @words;
```

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `lcfirst` - το αντίστοιχο για πεζά· μετατρέπει τον πρώτο χαρακτήρα στην πεζή του μορφή
- `uc` - μετατρέπει σε κεφαλαία κάθε χαρακτήρα της συμβολοσειράς· χρησιμοποιήστε όταν θέλετε όλη τη συμβολοσειρά κεφαλαιοποιημένη, όχι μόνο το πρώτο γράμμα
- `fc` - casefold του Unicode, η σωστή επιλογή για σύγκριση χωρίς διάκριση πεζών-κεφαλαίων όταν η είσοδος μπορεί να περιέχει μη-ASCII
- `$_` - το προεπιλεγμένο υποκείμενο που διαβάζει η `ucfirst` όταν καλείται χωρίς όρισμα
- `use locale` - ελέγχει αν εφαρμόζονται οι πίνακες locale ή οι κανόνες ASCII/Unicode στον πρώτο χαρακτήρα όταν αυτός είναι κάτω από το σημείο κώδικα 256
- `use feature 'unicode_strings'` - επιβάλλει πλήρη titlecasing Unicode ανεξάρτητα από τη σημαία UTF-8 στην είσοδο

Filehandles, αρχεία, κατάλογοι

umask

Θέτει ή διαβάζει τη μάσκα mode δημιουργίας αρχείων της διεργασίας.

Η `umask` ελέγχει ποια bits δικαιωμάτων **καθαρίζονται** από το όρισμα `MODE` κάθε κλήσης δημιουργίας αρχείου ή καταλόγου που κάνει αυτή η διεργασία από εδώ και στο εξής. Είναι ιδιότητα της εκτελούμενης διεργασίας, όχι κάποιου συγκεκριμένου filehandle, και κληρονομείται από τις θυγατρικές διεργασίες. Καλέστε `umask EXPR` για να εγκαταστήσετε νέα μάσκα και να λάβετε την προηγούμενη· καλέστε σκέτη `umask` για να διαβάσετε την τρέχουσα μάσκα χωρίς να την αλλάξετε.

Σύνοψη

```
umask EXPR
umask
```

Τι επιστρέφεται

Η **προηγούμενη** umask ως ακέραιος, ώστε να μπορείτε να την αποθηκεύσετε και να την επαναφέρετε αργότερα:

```
my $old = umask 0077;
# ... create private files ...
umask $old;
```

Η σκέτη `umask` επιστρέφει την τρέχουσα μάσκα χωρίς να την αλλάξει. Αν το σύστημα δεν διαθέτει κλήση `umask(2)` και ο καλών προσπαθεί να περιορίσει την πρόσβαση για τον εαυτό του (`(EXPR & 0700) > 0`), εγείρεται εξαίρεση· αν η κλήση δεν είναι υλοποιημένη και ο καλών δεν περιορίζει τη δική του πρόσβαση, επιστρέφεται `undef`. Σε Linux αυτή η περίπτωση δεν προκύπτει.

Πώς εφαρμόζεται η μάσκα

Η μάσκα είναι **αφαιρετική**. Κάθε bit που είναι ενεργό στη umask καθαρίζεται από το `MODE` που περνιέται στις `mkdir`, `sysopen`, `open` σε λειτουργίες δημιουργίας, και στις υποκείμενες κλήσεις συστήματος `creat(2)` / `mkdir(2)`. Τα πραγματικά δικαιώματα του νέου αρχείου είναι `MODE & ~umask`.

Δικαίωμα Unix όπως `rwxr-x---` είναι τρία σύνολα τριών bits, ή τρία οκταδικά ψηφία - `0750`. Το αρχικό `0` σηματοδοτεί την κυριολεκτική τιμή ως οκταδική και δεν είναι το ίδιο ψηφίο δικαιωμάτων. Η umask γράφεται με τον ίδιο τρόπο και κατονομάζει τα bits που θέλετε **απενεργοποιημένα** για νεοδημιουργημένα αρχεία.

Επεξεργασμένο παράδειγμα. Με `umask 0022`, ζητώντας από την `sysopen` το `0666` στην πραγματικότητα δημιουργείται αρχείο με mode `0644`:

```
  0666  &~  0022   =   0644
  rw-rw-rw-    ----w--w-      rw-r--r--
```

Με `umask 0027` (η ομάδα δεν μπορεί να γράψει, οι άλλοι δεν μπορούν να διαβάσουν, να γράψουν ή να εκτελέσουν), το ίδιο αίτημα `0666` αποδίδει `0640` (`0666 & ~0027 == 0640`).

Καθολική κατάσταση που επηρεάζει

Η ίδια η μάσκα είναι καθολική σε επίπεδο διεργασίας κατάσταση πυρήνα που τίθεται μέσω της `umask(2)`. Η `umask` δεν διαβάζει ούτε γράφει άμεσα καμία ειδική μεταβλητή της Perl, αλλά κάθε κλήση δημιουργίας στη διεργασία την παρατηρεί, και η `$!` τίθεται από τις υποκείμενες κλήσεις συστήματος που *καταναλώνουν* τη μάσκα (`mkdir`, `sysopen`, `open`), όχι από την ίδια την `umask`. Θυγατρικές διεργασίες που δημιουργούνται μέσω `fork`, `exec` ή `system` κληρονομούν την τρέχουσα τιμή.

Παραδείγματα

Εγκαταστήστε μάσκα που κρύβει το αρχείο από όλους εκτός του ιδιοκτήτη, διατηρώντας την προηγούμενη τιμή για επαναφορά:

```
my $old = umask 0077;
open my $fh, ">", "secret.txt" or die "open: $!";
# secret.txt is created mode 0666 & ~0077 == 0600
close $fh;
umask $old;
```

Διαβάστε την τρέχουσα μάσκα χωρίς να την αλλάξετε - χρήσιμο για καταγραφή ή διαγνωστικά:

```
printf "current umask: %04o\n", umask;      # e.g. "current umask:␣
↪0022"
```

Τυπικές τιμές που μπορείτε να περάσετε - αυτές είναι οι τρεις μάσκες που στην πραγματικότητα χρησιμοποιούνται στην πράξη:

```
umask 0022;   # group + other: read/execute, no write   (default on␣
↪most systems)
umask 0027;   # group: read/execute; other: nothing
umask 0077;   # owner-only; nothing for group or other
```

Αλλαγή με εμβέλεια γύρω από μπλοκ κώδικα που δημιουργεί αρχεία. Η `umask` δεν έχει δική της μαγεία `local` - αποθηκεύστε και επαναφέρετε ρητά:

```
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;
}
```

Οκταδική κυριολεκτική τιμή έναντι συμβολοσειράς - μια συνηθισμένη παγίδα. Η umask είναι **αριθμός**, όχι συμβολοσειρά οκταδικών ψηφίων. Η ανάγνωσή της από διαμόρφωση απαιτεί την `oct`:

```
my $from_config = "0027";                    # string
umask oct $from_config;                      # correct: 0027 as a␣
↪number
umask $from_config;                          # WRONG: decimal 27 ==␣
↪033 octal
```

Οριακές περιπτώσεις

- **Η σκέτη umask σε περιβάλλον λίστας** εξακολουθεί να επιστρέφει έναν μόνο ακέραιο. Η συνάρτηση έχει βαθμωτή τιμή.
- **Αρνητικές ή εκτός εύρους τιμές** μασκάρονται στα εννέα χαμηλής τάξης bits δικαιωμάτων από τον πυρήνα· επιβιώνουν μόνο τα bits `0777`. Το πέρασμα `0777` απενεργοποιεί όλα τα δικαιώματα σε κάθε νεοδημιουργημένο αρχείο, κάτι που σχεδόν ποτέ δεν είναι αυτό που θέλετε.
- **Η `umask` δεν επηρεάζει υπάρχοντα αρχεία.** Φιλτράρει μόνο το όρισμα `MODE` κλήσεων *δημιουργίας*. Για να αλλάξετε τα δικαιώματα ενός υπάρχοντος αρχείου χρησιμοποιήστε την `chmod`.
- **Η `umask` δεν επηρεάζει την `open` για υπάρχον αρχείο** - το άνοιγμα αρχείου που υπάρχει ήδη δεν εφαρμόζει ξανά τη μάσκα. Εφαρμόζεται μόνο όταν δημιουργείται αρχείο (`O_CREAT` στη `sysopen`, ή λειτουργίες `>`/`>>`/`+>` που δημιουργούν νέα διαδρομή στην `open`).
- **Κληρονομικότητα σε `fork`/`exec`.** Τα παιδιά ξεκινούν με την τρέχουσα umask του γονέα. Η ρύθμιση της μάσκας πριν από `system` ή `exec` αλλάζει το περιβάλλον που κληρονομεί το παιδί· επαναφέρετέ την μετά, αν ο γονέας εξακολουθεί να ενδιαφέρεται.
- **Threads.** Η umask είναι ιδιότητα ανά διεργασία στον πυρήνα, οπότε σε συστήματα όπου τα threads της Perl μοιράζονται διεργασία (η συνηθισμένη περίπτωση) κάθε thread βλέπει την ίδια μάσκα. Η αλλαγή της σε ένα thread την αλλάζει για όλα.
- **Η `umask(2)` δεν είναι υλοποιημένη.** Το POD τεκμηριώνει διαδρομή εφεδρείας για συστήματα χωρίς την κλήση συστήματος: εγείρει σε προσπάθειες περιορισμού της δικής του πρόσβασης του ιδιοκτήτη, διαφορετικά επιστρέφει `undef`. Κάθε πλατφόρμα που στοχεύει η pperl διαθέτει `umask(2)`, οπότε αυτή η διαδρομή είναι ουσιαστικά μη προσβάσιμη εδώ.
- **Οκταδική παρουσίαση**. Εκτυπώστε τη μάσκα με `printf "%04o"`· το προεπιλεγμένο `%d` εκτυπώνει δεκαδικά, που είναι δυσανάγνωστα ως bits δικαιωμάτων.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `chmod` - αλλάζει δικαιώματα σε **υπάρχον** αρχείο· ο αντίστροφος του φιλτραρίσματος κατά τη δημιουργία που εκτελεί η `umask`
- `mkdir` - κλήση δημιουργίας καταλόγου της οποίας το όρισμα `MODE` φιλτράρεται μέσω της τρέχουσας umask
- `sysopen` - δημιουργία αρχείου χαμηλού επιπέδου της οποίας το όρισμα `MODE` φιλτράρεται μέσω της τρέχουσας umask· το κανονικό σημείο όπου γίνεται αισθητή η μάσκα
- `open` - σε λειτουργίες δημιουργίας (`>`, `>>`, `+>`) η υποκείμενη `creat(2)` εφαρμόζει την umask στο προεπιλεγμένο `0666`
- `oct` - μετατρέπει συμβολοσειρά όπως `"0027"` σε ακέραια τιμή που αναμένει η `umask`
- `$!` - τίθεται από τις κλήσεις δημιουργίας που καταναλώνουν τη μάσκα, όχι από την ίδια την `umask`· ελέγξτε την μετά την `open` / `sysopen` / `mkdir` που αποτυγχάνει

Διάφορα

undef

Η απροσδιόριστη τιμή, και ο τελεστής που την παράγει.

Η `undef` φοράει δύο καπέλα. Η σκέτη `undef` είναι **τιμή** - το κανονικό απροσδιόριστο βαθμωτό, αυτό για το οποίο η `defined` επιστρέφει ψευδές. Η `undef EXPR` είναι **μοναδιαίος τελεστής** που απροσδιοριστοποιεί το όρισμά της επιτόπια: η μεταβλητή, πίνακας, πίνακας κατακερματισμού, υπορουτίνα ή typeglob που κατονομάζει η `EXPR` αδειάζεται και, όπου είναι δυνατόν, απελευθερώνεται η αποθήκευσή του.

Και οι δύο μορφές αποτιμώνται στην απροσδιόριστη τιμή.

Σύνοψη

```
undef                  # the undefined value
undef EXPR             # undefine the lvalue EXPR, return undef
```

Η `EXPR` πρέπει να είναι lvalue: βαθμωτό, πίνακας (με `@`), πίνακας κατακερματισμού (με `%`), υπορουτίνα (με `&`) ή typeglob (με `*`).

Τι επιστρέφεται

Πάντα η απροσδιόριστη τιμή. Η μορφή τελεστή επιστρέφει `undef` *μετά* το σβήσιμο του ορίσματός της· η μορφή τιμής απλώς σας δίνει `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
```

Η τιμή έναντι του τελεστή

Η σκέτη undef είναι το απροσδιόριστο βαθμωτό. Η ανάθεσή της σε μεταβλητή αφήνει την αποθήκευση της μεταβλητής στη θέση της και θέτει την τιμή της σε 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)
```

Σημειώστε την τελευταία γραμμή: η `@big = undef` είναι ανάθεση λίστας μίας λίστας ενός στοιχείου του οποίου το μοναδικό στοιχείο είναι undef, οπότε ο `@big` καταλήγει με μήκος 1. Σχεδόν ποτέ δεν είναι αυτό που θέλετε. Χρησιμοποιήστε `@big = ()` για άδειασμα, ή `undef @big` για άδειασμα *και* απελευθέρωση της υποκείμενης αποθήκευσης.

Η μορφή τελεστή κάνει περισσότερα από το να αναθέτει 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.
```

Η διάκριση έχει σημασία για τη μνήμη. Η `@array = ()` αφήνει την κατανεμημένη χωρητικότητα ανέπαφη ώστε το επόμενο push να είναι φθηνό· η `undef @array` την απελευθερώνει. Για μεγάλο, εφάπαξ πίνακα που θέλετε να αφήσετε πραγματικά, χρησιμοποιήστε undef. Για πίνακα εργασίας που θα γεμίσετε ξανά, χρησιμοποιήστε `= ()`.

Παραδείγματα

Επιστροφή «καμίας τιμής» από υπορουτίνα. Σε βαθμωτό περιβάλλον αυτός είναι ένας καθαρός τρόπος να σηματοδοτήσετε αποτυχία· σε περιβάλλον λίστας η σκέτη undef παράγει λίστα ενός στοιχείου που περιέχει undef, η οποία είναι αληθής:

```
sub find_user {
    my ($id) = @_;
    return undef unless $id;        # scalar-context caller: false
    ...
}
```

Για υπορουτίνες που προορίζονται να σηματοδοτούν αποτυχία και στα δύο περιβάλλοντα, επιστρέψτε αντ” αυτού κενή λίστα - η `return` χωρίς όρισμα κάνει το σωστό:

```
sub find_user {
    my ($id) = @_;
    return unless $id;              # () in list ctx, undef in␣
 ↪scalar ctx
    ...
}
```

Παράκαμψη θεσιακών τιμών σε ανάθεση λίστας. Η undef στην **αριστερή πλευρά** μιας ανάθεσης λίστας είναι σύμβολο κράτησης θέσης που πετάει την αντίστοιχη τιμή:

```
my ($x, undef, $z) = foo();         # discard the middle value
my (undef, @rest)  = @_;            # drop the first arg, keep the␣
 ↪rest
```

Κλασικό ιδίωμα select για ύπνο κάτω του δευτερολέπτου - τρία αγνοημένα σύνολα filehandle συν χρονικό όριο:

```
select undef, undef, undef, 0.25;   # sleep 250 ms
```

Επιστροφή διπλού περιβάλλοντος:

```
return wantarray ? (undef, $errmsg) : undef if $they_blew_it;
```

Απελευθέρωση μεγάλου συνόλου εργασίας:

```
my %cache = build_huge_cache();
use_cache(\%cache);
undef %cache;                       # free buckets and entries now
```

Απροσδιοριστοποίηση υπορουτίνας. Το όνομα στον πίνακα συμβόλων παραμένει, αλλά η κλήση της θα παραπονεθεί για απροσδιόριστη υπορουτίνα:

```
sub greet { print "hi\n" }
undef &greet;
greet();                            # Undefined subroutine &
 ↪main::greet
```

Η απροσδιοριστοποίηση typeglob εκμηδενίζει *κάθε* υποδοχή ταυτόχρονα:

```
our $xyz = 1;
our @xyz = (1, 2);
our %xyz = (a => 1);
sub xyz { 42 }

undef *xyz;                         # $xyz, @xyz, %xyz, &xyz all␣
 ↪gone
```

Οριακές περιπτώσεις

- **Η `undef $hash{$key}` δεν είναι `delete`**. Θέτει την τιμή σε εκείνο το κλειδί σε undef· το ίδιο το κλειδί εξακολουθεί να υπάρχει και η `exists` εξακολουθεί να επιστρέφει αληθές. Χρησιμοποιήστε `delete $hash{$key}` για να αφαιρέσετε το κλειδί. Το ίδιο ισχύει για τα στοιχεία πίνακα: η `undef $arr[3]` αφήνει το ευρετήριο 3 στη θέση του με τιμή undef, δεν συντομεύει τον πίνακα.
- **Σε tied μεταβλητές και τιμές DBM**, η `undef $tied{$key}` καλεί το STORE της tie, όχι το DELETE της. Η προειδοποίηση του perldoc («πιθανότατα δεν κάνει αυτό που περιμένετε στις περισσότερες προκαθορισμένες μεταβλητές ή τιμές λίστας DBM») αφορά συγκεκριμένα αυτό.
- **Η `undef` είναι μοναδιαίος τελεστής, όχι τελεστής λίστας**. Παίρνει ακριβώς ένα όρισμα, οπότε η `undef $a, $b` αναλύεται ως `(undef $a), $b` - μόνο η `$a` απροσδιοριστοποιείται. Για να απροσδιοριστοποιήσετε πολλές μεταβλητές, καλέστε την αρκετές φορές, ή χρησιμοποιήστε βρόχο:

```
undef $_ for $a, $b, $c;
```

- **Η `undef EXPR` απαιτεί lvalue**. Οι `undef 42` ή `undef func()` είναι σφάλμα μεταγλώττισης (*Can't modify ... in undef operator*).
- **Η `undef` στην αριστερή πλευρά σε ανάθεση λίστας είναι σύνταξη, όχι τιμή**. Η `(undef, $x) = @pair` αναλύεται ειδικά· δεν μπορείτε να γράψετε `my $slot = undef; ($slot, $x) = @pair` και να περιμένετε η πρώτη τιμή να απορριφθεί - αυτό κάνει ανάθεση στο `$slot`.
- **Η σταθερή `undef` σε λογικό περιβάλλον είναι ψευδής**, σε αριθμητικό περιβάλλον είναι `0`, σε περιβάλλον συμβολοσειράς είναι `""`. Υπό `use warnings` καθεμία από αυτές τις μετατροπές εκπέμπει προειδοποίηση `Use of uninitialized value` στο σημείο χρήσης, όχι στο σημείο που η μεταβλητή έγινε undef.
- **Η `undef &sub` δεν αφαιρεί το όνομα της υπορουτίνας** από το stash του πακέτου της - μόνο τον κώδικα. Για να αφαιρέσετε και το όνομα, απροσδιοριστοποιήστε ολόκληρη την υποδοχή glob: `delete $Package::{sub_name}` ή `undef *Package::sub_name` αν θέλετε επίσης να φύγουν οι αδελφές υποδοχές.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `defined` - το κατηγόρημα που ρωτάει «είναι αυτή η τιμή μη-undef;»· ο φυσικός αντίστοιχος της τιμής `undef`
- `delete` - αφαιρέστε κλειδί πίνακα κατακερματισμού ή στοιχείο πίνακα εξ ολοκλήρου· χρησιμοποιήστε αυτή, όχι την `undef`, όταν θέλετε η `exists` να γίνει ψευδής
- `exists` - ελέγξτε αν κλειδί πίνακα κατακερματισμού ή ευρετήριο πίνακα είναι παρόν, ανεξάρτητα αν η τιμή του είναι ορισμένη

- `local` - αντικαθιστά προσωρινά την τιμή μιας μεταβλητής πακέτου (συχνά με undef) για τη διάρκεια μιας εμβέλειας
- `wantarray` - συνδυάζεται με την undef στο ιδιωματικό `wantarray ? (undef, $err) : undef` για επιστροφή αποτυχίας διπλού περιβάλλοντος

Filehandles, αρχεία, κατάλογοι

unlink

Αφαιρεί μία ή περισσότερες εγγραφές καταλόγου που ονομάζουν αρχεία.

Η unlink ζητά από τον πυρήνα να καταργήσει τον σύνδεσμο μεταξύ κάθε ονόματος της LIST και του αρχείου στο οποίο αναφέρεται. Αν αυτός ο σύνδεσμος ήταν ο τελευταίος και καμία διεργασία δεν κρατά ακόμη το αρχείο ανοιχτό, ο αποθηκευτικός χώρος του αρχείου ανακτάται· διαφορετικά το αρχείο παραμένει στον δίσκο, χωρίς όνομα αλλά προσβάσιμο μέσω οποιουδήποτε ανοιχτού περιγραφέα, μέχρι να κλείσει το τελευταίο handle. Η unlink λειτουργεί σε **εγγραφές καταλόγου**, όχι στο περιεχόμενο αρχείων.

Σύνοψη

```
unlink LIST
unlink
```

Τι επιστρέφεται

Ο αριθμός των αρχείων που πραγματικά αφαίρεσε η unlink, ως ακέραιος. Σε ολική επιτυχία ισούται με `scalar @LIST`· σε μερική ή ολική αποτυχία είναι ο αριθμός των ονομάτων που αφαιρέθηκαν πριν και μετά εκείνα που απέτυχαν. Το μηδέν είναι έγκυρη επιστροφή «ψευδής» όταν τίποτα δεν μπόρεσε να αφαιρεθεί. Η `$!` τίθεται στο σφάλμα από την **τελευταία** αποτυχημένη κλήση - **δεν** σας λέει ποιο όνομα απέτυχε.

```
my $removed = unlink 'a', 'b', 'c';
unlink @goners;
unlink glob "*.bak";
```

Αν παραλειφθεί η LIST, η unlink λειτουργεί στο μεμονωμένο όνομα αρχείου στην `$_`.

Καθολική κατάσταση που επηρεάζει

- `$_` - χρησιμοποιείται ως τελούμενο όταν η unlink καλείται χωρίς ορίσματα.
- `$!` - τίθεται στην errno της τελευταίας unlink(2) που απέτυχε. Επειδή αντικατοπτρίζει μόνο την τελική αποτυχία στη λίστα, δεν είναι αξιόπιστος τρόπος να διαγνώσετε ποιο όνομα δεν μπόρεσε να αφαιρεθεί.

Εντοπισμός ποιο αρχείο απέτυχε

Η unlink επιστρέφει πλήθος, όχι κατάσταση ανά αρχείο, και επικαλύπτει την `$!` σε κάθε επανάληψη. Για να αποδώσετε σφάλματα σε μεμονωμένα ονόματα, καλέστε unlink μία φορά ανά αρχείο:

```
foreach my $file (@goners) {
    unlink $file
        or warn "could not unlink $file: $!";
}
```

Η μορφή ανά αρχείο επίσης σας επιτρέπει να συνεχίσετε μετά από αποτυχία, πράγμα που η μορφή λίστας κάνει έμμεσα αλλά σιωπηρά.

Κατάλογοι

Η unlink **δεν** αφαιρεί καταλόγους. Στο Linux επιστρέφει `0` και θέτει την `$!` σε EISDIR όταν της ζητηθεί. Το δυαδικό perl αναγνωρίζει μια επιλογή γραμμής εντολών `-U` που, σε συνδυασμό με εκτέλεση ως superuser, αίρει τον έλεγχο - αλλά ακόμη και τότε το unlink ενός καταλόγου παρακάμπτει τις εγγυήσεις ακεραιότητας καταλόγων του συστήματος αρχείων και μπορεί να αλλοιώσει το link count του γονικού καταλόγου. Χρησιμοποιήστε αντί αυτής την `rmdir`· καλεί την `rmdir(2)`, που είναι η σωστή κλήση συστήματος για την αφαίρεση κενής εγγραφής καταλόγου.

```
rmdir $dir  or die "rmdir $dir: $!";    # correct
unlink $dir;                            # fails with EISDIR
```

Σημασιολογία διαγραφής κατά POSIX

Η unlink αφαιρεί το **όνομα**, όχι το αρχείο. Όσο κάποια διεργασία κρατά το αρχείο ανοιχτό, τα δεδομένα του αρχείου και το inode παραμένουν έγκυρα και εκείνη η διεργασία μπορεί να συνεχίσει να διαβάζει και να γράφει μέσω του περιγραφέα της. Ο αποθηκευτικός χώρος ανακτάται μόνο όταν το τελευταίο όνομα έχει εξαφανιστεί **και** ο τελευταίος περιγραφέας έχει κλείσει. Αυτό είναι το ιδιωματικό μοτίβο για τη δημιουργία προσωρινού αρχείου που καθαρίζεται από μόνο του ακόμη και σε 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.
```

Είναι επίσης ο λόγος που ένα μακροπρόθεσμα τρέχον πρόγραμμα μπορεί να συνεχίσει να γράφει σε ένα αρχείο καταγραφής του οποίου το όνομα έχει αφαιρεθεί με unlink από τον διαχειριστή - οι εγγραφές πετυχαίνουν, αλλά τίποτα στον δίσκο δεν είναι ορατό κάτω από το παλιό όνομα. Η `df` θα αναφέρει τον χώρο ως χρησιμοποιούμενο μέχρι να κλείσει το handle.

Symbolic links

Αν ένα όνομα στη LIST είναι symbolic link, η unlink αφαιρεί το **ίδιο το link**, όχι το αρχείο στο οποίο δείχνει. Ο στόχος μένει ανέπαφος ακόμη και αν το link είναι κρεμασμένο ή δείχνει σε κατάλογο. Αυτό ταιριάζει με την κλήση συστήματος `unlink(2)` και είναι σχεδόν πάντα αυτό που θέλετε· χρησιμοποιήστε πρώτα την `readlink` αν χρειάζεστε συγκεκριμένα να ενεργήσετε στον στόχο.

Παραδείγματα

Αφαίρεση ενός αρχείου, έλεγχος του αποτελέσματος:

```
unlink $path
    or die "cannot remove $path: $!";
```

Αφαίρεση πολλών και αναφορά του απολογισμού:

```
my @files  = glob "*.tmp";
my $gone   = unlink @files;
warn "removed $gone of ", scalar @files, " files\n"
    if $gone != @files;
```

Διαγνωστικά ανά αρχείο (ο μόνος αξιόπιστος τρόπος για να ξέρετε ποιο όνομα απέτυχε):

```
my @failed;
for my $f (@goners) {
    unlink $f or push @failed, "$f: $!";
}
die "unlink failures:\n", join("\n", @failed), "\n" if @failed;
```

Στηριχθείτε στην $_ σε αλυσίδα ελέγχων αρχείων:

```
for (glob "*.bak") {
    next unless -f && -M _ > 30;    # older than 30 days
    unlink or warn "skip $_: $!";
}
```

Αυτο-καθαριζόμενο αρχείο εργασίας (ιδίωμα POSIX unlink-while-open):

```
open my $scratch, "+>", "/tmp/work.$$" or die $!;
unlink "/tmp/work.$$"                  or die $!;
# write through $scratch, no cleanup needed on exit
```

Οριακές περιπτώσεις

- **Κενή LIST**: η `unlink ()` επιστρέφει `0` και δεν αγγίζει την `$!`. Το πέρασμα κενού πίνακα συμπεριφέρεται με τον ίδιο τρόπο.
- **Ανύπαρκτο όνομα**: επιστρέφει `0` για εκείνη την εγγραφή και θέτει την `$!` σε `ENOENT`. Η υπόλοιπη λίστα εξακολουθεί να επεξεργάζεται.
- **Άρνηση δικαιωμάτων**: η `unlink(2)` ελέγχει το δικαίωμα εγγραφής στον **περιέχοντα κατάλογο**, όχι στο ίδιο το αρχείο. Η αφαίρεση αρχείου στο οποίο δεν μπορείτε να γράψετε επιτρέπεται όσο μπορείτε να γράψετε στον γονικό του κατάλογο· η αφαίρεση αρχείου που σας ανήκει σε κατάλογο όπου δεν μπορείτε να γράψετε όχι. Το `$!` θα είναι `EACCES` ή `EPERM`.
- **Sticky κατάλογοι** (`/tmp`, mode 1777): χρήστες χωρίς root μπορούν να αφαιρέσουν μόνο ονόματα που τους ανήκουν. Οι αποτυχίες εμφανίζονται ως `EPERM`.
- **Τιμή επιστροφής σε λογικό περιβάλλον**: η `unlink @files or die` ενεργοποιείται μόνο όταν απέτυχε **κάθε** όνομα (το πλήθος είναι `0`). Για να

τερματίσετε σε **οποιαδήποτε** αποτυχία, συγκρίνετε με `scalar @files` ή χρησιμοποιήστε τον βρόχο ανά αρχείο.

- **Απασχολημένο αρχείο σε Linux**: το Linux δεν αποτρέπει την `unlink` αρχείου που είναι mmap'ed, υπό εκτέλεση ή ανοιχτό - το όνομα αφαιρείται και ισχύει η σημασιολογία POSIX παραπάνω. Άλλα λειτουργικά συστήματα μπορεί να επιστρέψουν `ETXTBSY` ή `EBUSY`· αυτή η σελίδα καλύπτει το Linux.
- **Taint mode**: ονόματα προερχόμενα από tainted δεδομένα επιτρέπονται ως τελούμενα της `unlink`· δείτε το `perlsec` για τις συνήθεις προειδοποιήσεις σχετικά με την αποδοχή μη έμπιστων διαδρομών.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `rmdir` - αφαιρεί έναν **κενό** κατάλογο· το σωστό εργαλείο όταν το όνομα αναφέρεται σε εγγραφή καταλόγου τύπου `DIR`
- `rename` - μετακινεί ή ανασυνδέει ένα όνομα χωρίς να το απορρίψει, συχνά σε συνδυασμό με `unlink` κατά την ατομική αντικατάσταση αρχείου
- `link` - δημιουργεί επιπλέον σκληρό σύνδεσμο σε αρχείο· η `unlink` είναι το αντίστροφό της για ένα όνομα τη φορά
- `symlink` - δημιουργεί symbolic link· σημειώστε ότι η `unlink` αφαιρεί το ίδιο το symlink, όχι τον στόχο του
- `truncate` - συρρικνώνει αρχείο επιτόπια χωρίς να εκτελέσει unlink όταν θέλετε να ξαναχρησιμοποιήσετε το όνομα

Λίστες · Δεδομένα σταθερού μήκους

unpack

Εξάγει τυποποιημένες τιμές από δυαδική συμβολοσειρά ή συμβολοσειρά σταθερού πλάτους σύμφωνα με ένα πρότυπο.

Η `unpack` είναι το αντίστροφο της `pack`. Διασχίζει την `EXPR` από αριστερά προς τα δεξιά, καταναλώνοντας τα bytes που περιγράφει κάθε οδηγία στο `TEMPLATE` και μετατρέποντάς τα σε τιμές Perl. Το αποτέλεσμα είναι λίστα - μία τιμή ανά οδηγία, ή ανά επανάληψη όταν η οδηγία έχει μετρητή. Αν παραλειφθεί η `EXPR`, η `unpack` διαβάζει από την `$_`.

Αυτή η σελίδα είναι μια **αναφορά οδηγιών**. Για αφηγηματική εισαγωγή με λυμένα παραδείγματα πρωτοκόλλων και μορφών αρχείων, ξεκινήστε από τον *οδηγό pack/unpack*.

Σύνοψη

```
unpack TEMPLATE, EXPR
unpack TEMPLATE                                 # reads from $_
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
my @fields = unpack "A10 A10 A*", $line;
my ($ver, $len, $payload) = unpack "n N A*", $msg;
```

Τι επιστρέφεται

Μια λίστα τιμών, μία ανά οδηγία (ή ανά επανάληψη όταν η οδηγία έχει μετρητή). Σε **βαθμωτό περιβάλλον**, επιστρέφεται μόνο η **πρώτη** τιμή που παράχθηκε:

```
my @all   = unpack "A4 A4 A*", $rec;    # three values
my $first = unpack "A4 A4 A*", $rec;    # just the first - not a
↪count
```

Κάθε οδηγία καθορίζει τον τύπο Perl του αποτελέσματός της - δείτε τη στήλη «Τύπος αποτελέσματος» στον πίνακα οδηγιών παρακάτω.

Καθολική κατάσταση που επηρεάζει

Το unpack TEMPLATE χωρίς EXPR διαβάζει την $_. Καμία άλλη καθολική του διερμηνέα δεν συμβουλεύεται. Οι οδηγίες C0 / U0 αλλάζουν τοπικά εντός του προτύπου την ερμηνεία byte-έναντι-χαρακτήρα, αλλά δεν μεταβάλλουν κάποια εξωτερική κατάσταση.

Σύνταξη προτύπου

Πανομοιότυπη με αυτή της pack: ακολουθία γραμμάτων-οδηγιών, προαιρετικά ακολουθούμενη από μετρητές επανάληψης N / * / [...] και τροποποιητές ! / < / >. Οι παρενθέσεις σχηματίζουν ομάδες· το # εισάγει σχόλιο προτύπου. Δείτε την pack για τους πλήρεις κανόνες σύνταξης - αυτή η σελίδα καλύπτει μόνο τη σημασιολογία που είναι ειδική για την unpack.

Πίνακας οδηγιών

Όλες οι οδηγίες της unpack σε ένα μέρος. Το **W** είναι το πλάτος σε bytes που καταναλώνεται από την είσοδο ανά παραγόμενο βαθμωτό.

Οδηγία	W	Προσημασμ	Endian	Τύπος αποτελέσμ	Τροποποιητ	Σημειώσεις
a	μετρητής	-	-	συμβολοσει	-	Επιστρέφει τα bytes ανέγγιχτα
A	μετρητής	-	-	συμβολοσει	-	Αφαιρεί τους τελικούς λευκούς χαρακτήρες και τα NUL

συνέχεια στην επόμενη σελίδα

Πίνακας 2 – συνεχίζεται από την προηγούμενη σελίδα

Οδηγία	W	Προσημασμ	Endian	Τύπος αποτελέσμ	Τροποποιητ	Σημειώσεις
Z	μετρητής	-	-	συμβολοσει	-	Επιστρέφει τα πάντα μέχρι το πρώτο NUL
b	μετρητής/8	-	-	συμβολοσει bits	-	Χαρακτήρες «0»/»1», LSB κάθε byte πρώτο
B	μετρητής/8	-	-	συμβολοσει bits	-	Χαρακτήρες «0»/»1», MSB κάθε byte πρώτο
h	μετρητής/2	-	-	δεκαεξαδικ συμβολοσει	-	Χαρακτήρες [0-9a-f], κάτω nybble πρώτο
H	μετρητής/2	-	-	δεκαεξαδικ συμβολοσει	-	Χαρακτήρες [0-9a-f], άνω nybble πρώτο
c	1	ναι	-	ακέραιος	-	Προσημασμένο char
C	1	όχι	-	ακέραιος	-	Μη προσημασμένο char
W	1	όχι	-	ακέραιος	-	Μη προσημασμένο char· αποδίδει σημείο κώδικα σε κατάσταση U0
s	2	ναι	native	ακέραιος	! < >	s! = native short
S	2	όχι	native	ακέραιος	! < >	S! = native unsigned short

συνέχεια στην επόμενη σελίδα

Πίνακας 2 – συνεχίζεται από την προηγούμενη σελίδα

Οδηγία	W	Προσημασμ	Endian	Τύπος αποτελέσμ	Τροποποιητ	Σημειώσεις
`l`	4	ναι	native	ακέραιος	! < >	`l!` = native `long`
`L`	4	όχι	native	ακέραιος	! < >	`L!` = native `unsigned long`
`i`	native	ναι	native	ακέραιος	! < >	`sizeof(int)`
`I`	native	όχι	native	ακέραιος	! < >	`sizeof(unsigned int)`
`q`	8	ναι	native	ακέραιος	< >	Απαιτεί Perl με 64-bit ακεραίους
`Q`	8	όχι	native	ακέραιος	< >	Απαιτεί Perl με 64-bit ακεραίους
`n`	2	όχι	big	ακέραιος	!	Σειρά δικτύου· `n!` = προσημασμένο
`N`	4	όχι	big	ακέραιος	!	Σειρά δικτύου· `N!` = προσημασμένο
`v`	2	όχι	little	ακέραιος	!	Σειρά «VAX»· `v!` = προσημασμένο
`V`	4	όχι	little	ακέραιος	!	Σειρά «VAX»· `V!` = προσημασμένο
`j`	μέγεθος IV	ναι	native	ακέραιος	< >	Εσωτερική IV της Perl
`J`	μέγεθος UV	όχι	native	ακέραιος	< >	Εσωτερική UV της Perl
`f`	4	-	native	αριθμός	< >	IEEE 754 απλής ακρίβειας
`d`	8	-	native	αριθμός	< >	IEEE 754 διπλής ακρίβειας

συνέχεια στην επόμενη σελίδα

Πίνακας 2 – συνεχίζεται από την προηγούμενη σελίδα

Οδηγία	W	Προσημασι	Endian	Τύπος αποτελέσμ	Τροποποιητ	Σημειώσεις
F	μέγεθος NV	-	native	αριθμός	< >	Εσωτερικός float της Perl
D	ποικίλλει	-	native	αριθμός	< >	Long double
p	μέγεθος ptr	-	native	συμβολοσει	< >	Αποαναφέρει δείκτη σε συμβολοσειρά τερματισμένη με NUL
P	μέγεθος ptr	-	native	συμβολοσει	< >	Αποαναφέρει δείκτη· μετρητής = bytes προς ανάγνωση
u	ποικίλλει	-	-	συμβολοσει	-	Bytes αποκωδικοποιημένα με uudecode
U	ποικίλλει	-	-	ακέραιος	-	Αριθμός σημείου κώδικα Unicode
w	ποικίλλει	όχι	-	ακέραιος	-	Ακέραιος συμπιεσμένος κατά BER
x	1	-	-	(κανένας)	!	Παρακάμπτει ένα byte προς τα εμπρός· το `x!N` ευθυγραμμίζει προς τα εμπρός
X	−1	-	-	(κανένας)	!	Επιστροφή κατά ένα byte· το `X!N` ευθυγραμμίζει προς τα πίσω

συνέχεια στην επόμενη σελίδα

Πίνακας 2 – συνεχίζεται από την προηγούμενη σελίδα

Οδηγία	W	Προσημασι	Endian	Τύπος αποτελέσμ	Τροποποιητ	Σημειώσεις
@	απόλυτο	-	-	(κανένας)	!	Μετάβαση στη θέση N εντός της εσώτερης ομάδας
.	-	-	-	ακέραιος	!	Επιστρέφει την τρέχουσα θέση (σχετική με την αρχή της ομάδας / συμβολοσειράς)
(…)	-	-	-	-	< > !	Ομάδα: ο μετρητής επανάληψης και η endianness διαδίδονται προς τα μέσα
/	-	-	-	-	-	Δείτε «Φορτία με πρόθεμα μήκους» παρακάτω
%N	ποικίλλει	-	-	ακέραιος	-	**Πρόθεμα.** Άθροισμα ελέγχου N-bit των τιμών της επόμενης οδηγίας

Οι τροποποιητές έχουν την ίδια σημασία όπως στην pack: το ! επιλέγει native μεγέθη, σημασιολογία ευθυγράμμισης, προσήμανση για τα n / N / v / V, ή μετατοπίσεις σε bytes για τα @ / . · τα < / > εξαναγκάζουν endianness.

Διαφορές από την pack

- **Το a δεν αφαιρεί τίποτα** (επιστρέφει τα bytes ωμά)· **το A αφαιρεί τους τελικούς λευκούς χαρακτήρες και τα NUL· το Z σταματά στο πρώτο NUL**. Με την pack και τα τρία συμπληρώνουν· με την unpack και τα τρία αφαιρούν συμπλήρωση με διαφορετικό τρόπο.

- **Το x παρακάμπτει προς τα εμπρός κατά W, το X επιστρέφει κατά W.** Το X πριν την αρχή της συμβολοσειράς είναι μοιραίο σφάλμα.
- **Το / διαβάζει μετρητή από τα δεδομένα και τον εφαρμόζει στην επόμενη οδηγία**:

```
unpack("W/a", "\004Gurusamy")          # ("Guru")
unpack("a3/A A*", "007 Bond  J ")      # (" Bond", "J")
unpack("n/a*", "\x00\x0chello, world") # ("hello, world")
```

  Με την pack η μορφή είναι *length-item*/*item* και το μήκος υπολογίζεται από το μήκος της τιμής. Με την unpack η μορφή είναι /*item* (χωρίς length-item πριν) - η προηγούμενη ακέραια οδηγία παρέχει τον μετρητή.
- **Το %N είναι αποκλειστικό της unpack.** Αντικαθιστά την κανονική έξοδο της επόμενης οδηγίας με το άθροισμα N-bit των τιμών που θα είχε αλλιώς παραγάγει αυτή η οδηγία. Το %32W* είναι το άθροισμα ελέγχου sum του System V. Το %32b* μετρά τα ενεργοποιημένα bits.
- **Το . επιστρέφει την τρέχουσα θέση byte** αντί να συμπληρώνει με μηδενικά μέχρι μία. Χρήσιμο μετά από πλοήγηση x / X για να μάθετε πού βρίσκεστε.
- **Τα p και P αποαναφέρουν δείκτες που διαβάζονται από την είσοδο** - σχεδόν πάντα μη ασφαλές εκτός αν γνωρίζετε ότι η είσοδος παράχθηκε από pack "p" / pack "P" στην ίδια διεργασία.
- **Συμπεριφορά under-run / over-run:** αν το πρότυπο ζητά περισσότερα δεδομένα από όσα περιέχει η EXPR, το αποτέλεσμα δεν είναι καλά ορισμένο (μπορεί να αποδώσει κενές συμβολοσειρές, μηδενικά, μειωμένο μετρητή επανάληψης, ή εξαίρεση). Αν η EXPR είναι μακρύτερη από όσο καταναλώνει το πρότυπο, τα τελικά bytes αγνοούνται σιωπηλά.

Παραδείγματα

Ανάλυση μιας ελάχιστης κεφαλίδας δυαδικού πρωτοκόλλου - έκδοση 16-bit, μήκος 32-bit, στη συνέχεια φορτίο μεταβλητού μήκους, όλα σε σειρά bytes δικτύου:

```
my ($ver, $len, $payload) = unpack "n N A*", $msg;
```

Διαχωρισμός εγγραφής κειμένου σταθερού πλάτους - γρηγορότερος και πιο καθαρός από αλυσιδωμένες κλήσεις substr:

```
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"
```

Εξαγωγή αθροίσματος ελέγχου τύπου System V με το πρόθεμα % - το 32-bit άθροισμα κάθε byte, μασκαρισμένο σε 16 bits:

```
my $checksum = do {
    local $/;                          # slurp
    unpack "%32W*", readline $fh;
} % 65535;
```

Μέτρηση ενεργοποιημένων bits σε διάνυσμα bits - το %32b* διαβάζει ολόκληρη τη μάσκα ως bits και επιστρέφει το άθροισμά τους:

```
my $setbits = unpack "%32b*", $selectmask;
```

Ανάγνωση ακολουθίας little-endian 32-bit ακεραίων από ωμό ενταμιευτή:

```
my @ints = unpack "V*", $buf;
```

Χρησιμοποιήστε το x για να παρακάμψετε bytes πλήρωσης, το X για να επιστρέψετε μετά από peek:

```
# 2-byte tag, skip 2 reserved bytes, then 4 32-bit big-endian values
my ($tag, @vals) = unpack "n x2 N4", $frame;
```

Οριακές περιπτώσεις

- **Το βαθμωτό περιβάλλον επιστρέφει μόνο την πρώτη τιμή.** Η ανάθεση της unpack σε βαθμωτό σχεδόν ποτέ δεν είναι αυτό που θέλετε. Για να μετρήσετε πεδία, αναθέστε πρώτα σε λίστα, ή χρησιμοποιήστε `() = unpack ...` σε περιβάλλον λίστας:

  ```
  my $n = () = unpack "A10 A10 A*", $line;   # 3
  ```

- **Το * είναι άπληστο, όχι placeholder.** Το A* καταναλώνει όλα τα υπόλοιπα bytes ως μία συμβολοσειρά· δεν υπάρχει οπισθοχώρηση. Μια οδηγία * πρέπει να είναι η τελευταία στην ακολουθία της, διαφορετικά τίποτα μετά από αυτή δεν εκτελείται.
- **Τα πρότυπα δεν είναι κανονικές εκφράσεις.** Οι λευκοί χαρακτήρες αγνοούνται, αλλά δεν υπάρχει εναλλαγή, ούτε |, ούτε lookahead. Αν το σχήμα των δεδομένων εξαρτάται από προηγούμενα πεδία, κάντε unpack σε στάδια: αποκωδικοποιήστε μια κεφαλίδα, μετά καλέστε ξανά την unpack πάνω στο φορτίο χρησιμοποιώντας πρότυπο παραγόμενο από αυτό που μόλις διαβάσατε.
- **Σημασιολογία byte έναντι χαρακτήρα.** Η unpack τρέχει σε κατάσταση byte προεπιλεγμένα πάνω σε συμβολοσειρά bytes, και σε κατάσταση χαρακτήρα σε συμβολοσειρά με σημαία UTF-8 τεθειμένη. Το C0 εξαναγκάζει σε κατάσταση byte από εκείνο το σημείο· το U0 εξαναγκάζει σε κατάσταση UTF-8. Η ανάμειξη συμβολοσειράς σημαδεμένης ως UTF-8 με αριθμητικές οδηγίες όπως N διαβάζει σημεία κώδικα, όχι bytes - συνήθως όχι αυτό που θέλετε. Καλέστε πρώτα την utf8::encode, ή λειτουργήστε πάνω σε δεδομένα επιπέδου byte.
- **Το X πριν την αρχή της συμβολοσειράς είναι μοιραίο.**
- **Τα p / P σε αυθαίρετη είσοδο αποτελούν απροσδιόριστη συμπεριφορά.** Μην τα χρησιμοποιείτε ποτέ σε δεδομένα που δεν έχετε πακετάρει εσείς οι ίδιοι στην ίδια διεργασία.
- **Χωρίς όρισμα EXPR**: το unpack TEMPLATE διαβάζει την $_. Χρήσιμο μέσα σε βρόχους `while (<$fh>) { ... }` πάνω από εγγραφές σταθερού πλάτους.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `pack` - η αντίστροφη λειτουργία· ίδια γλώσσα προτύπων
- `substr` - εξαγωγή μίας σταθερής τομής· απλούστερη για ένα πεδίο, βραδύτερη για πολλά
- `sprintf` - πλήρης κύκλος μεταξύ δυαδικού και αναγνώσιμου από άνθρωπο
- `read` - διαβάζει ωμά bytes από filehandle σε ενταμιευτή για την `unpack`
- `vec` - προσπελάσιμη κατά δείκτη πρόσβαση σε διάνυσμα bits χωρίς πρότυπο
- *Οδηγός pack/unpack* - προσανατολισμένη σε εργασίες περιδιάβαση με λυμένα παραδείγματα πρωτοκόλλων και μορφών αρχείων

Πίνακες

unshift

Προπενθέτει μία ή περισσότερες τιμές στην αρχή ενός πίνακα και επιστρέφει το νέο μήκος του πίνακα.

Η `unshift` είναι το κάτοπτρο της `push`: όπου η `push` προσαρτά στην ουρά, η `unshift` εισάγει στην κεφαλή. Οι τιμές στη `LIST` προπενθέτονται ως ομάδα, με τη σειρά που δίνονται, οπότε το πρώτο όρισμα της `LIST` γίνεται το νέο στοιχείο 0, το δεύτερο γίνεται στοιχείο 1 και ούτω καθεξής, με τα αρχικά περιεχόμενα να ολισθαίνουν σε υψηλότερους δείκτες.

Σύνοψη

```
unshift @arr, $val
unshift @arr, @more
unshift @arr, $first, $second, $third
```

Τι επιστρέφεται

Το νέο πλήθος στοιχείων στον πίνακα, ως ακέραιος. Είναι η ίδια τιμή που θα παίρνατε από `scalar @arr` μετά την κλήση.

```
my @colors = ("red");
my $n = unshift @colors, "blue", "green";   # $n == 3
```

Οι περισσότερες κλήσεις απορρίπτουν την τιμή επιστροφής· είναι περιστασιακά χρήσιμη όταν θέλετε να προπενθέσετε και να διακλαδώσετε βάσει του μεγέθους που προκύπτει σε ένα βήμα.

Παραδείγματα

Προπένθεση ενός μεμονωμένου στοιχείου:

```
my @animals = ("cat");
unshift @animals, "mouse";
# @animals is now ("mouse", "cat")
```

Προπένθεση πολλών στοιχείων - **η σειρά διατηρείται**, δεν αντιστρέφεται. Αυτή είναι η πιο συχνή σύγχυση με την `unshift`:

```
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)
```

Αν πράγματι θέλετε τις προπενθεμένες τιμές αντεστραμμένες, αντιστρέψτε τες ρητά:

```
my @a = (10, 20, 30);
unshift @a, reverse(1, 2, 3);
# @a is now (3, 2, 1, 10, 20, 30)
```

Προπένθεση ολόκληρου πίνακα - τα στοιχεία ενσωματώνονται, όχι ο πίνακας ως μία μοναδική τιμή:

```
my @head = ("a", "b");
my @tail = ("c", "d");
unshift @tail, @head;
# @tail is now ("a", "b", "c", "d")
```

Εισαγωγή στην κεφαλή έναντι εισαγωγής στην ουρά. Αυτές οι δύο κλήσεις παράγουν κατοπτρικά αποτελέσματα στον ίδιο αρχικό πίνακα:

```
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
```

Δουλέψτε σε πίνακα μέσω αναφοράς. Η `unshift` δέχεται αποαναφορημένο πίνακα στην αριστερή πλευρά:

```
my $ref = [10, 20, 30];
unshift @$ref, 0;
# $$ref[0] is now 0, @$ref is (0, 10, 20, 30)
```

Η μορφή αποαναφοράς με βέλος λειτουργεί με τον ίδιο τρόπο· γράψτε την αποαναφορά ως ξεχωριστή έκφραση αντί να εκτείνετε αλυσίδα μεθόδων κατά μήκος του ορίσματος της `unshift`:

```
my @list = @$ref;
unshift @list, -1;
```

Οριακές περιπτώσεις

- **Το κόστος είναι O(n).** Κάθε υπάρχον στοιχείο μετατοπίζεται προς τα πάνω κατά `scalar(LIST)` θέσεις, οπότε η προπένθεση σε μεγάλο πίνακα είναι αναλογικά ακριβή. Αν χτίζετε λίστα με επαναλαμβανόμενη προπένθεση, χτίστε την με `push` και `reverse` στο τέλος - ή χτίστε τη αντίστροφα με δείκτες - αντί να καλείτε `unshift` σε βρόχο.
- **Η λίστα προπενθέτεται ολόκληρη, όχι αντεστραμμένη.** Η `unshift @a, 1, 2, 3` τοποθετεί το 1 στον δείκτη 0, το 2 στον δείκτη 1, το 3 στον δείκτη 2. Αναγνώστες που έρχονται από γλώσσες όπου το «push στην κεφαλή» φυσικά αντιστρέφει (επειδή κάθε στοιχείο γίνεται η νέα κεφαλή) συχνά περιμένουν το αντίθετο· η Perl δεν το κάνει.
- **Η κενή `LIST` είναι no-op.** Η `unshift @arr` χωρίς τίποτα μετά αφήνει τον πίνακα αμετάβλητο και επιστρέφει το τρέχον πλήθος στοιχείων.
- **Δεμένοι πίνακες**: η `unshift` σε δεμένο πίνακα καλεί τον χειριστή `UNSHIFT` αν το πακέτο παρέχει έναν· αλλιώς υποχωρεί σε μια ακολουθία κλήσεων `SPLICE` ή `STORE` / `STORESIZE`. Σε κλάση tie που υλοποιεί μόνο `FETCH` και `STORE` μπορεί ακόμη να γίνει `unshift`, αλλά κάθε κλήση ξαναγράφει κάθε δείκτη.
- **Ο αποαναφορημένος πίνακας δουλεύει**: η `unshift @$ref, $x` τροποποιεί επιτόπου τον πίνακα στον οποίο αναφέρεται το `$ref`. Η `unshift @{ $h{key} }, $x` αυτο-δημιουργεί το `$h{key}` σε κενή αναφορά πίνακα αν δεν υπήρχε προηγουμένως.
- **Βαθμωτή έκφραση στην αριστερή πλευρά είναι συντακτικό σφάλμα.** Ένα πείραμα της Perl 5.14 δέχτηκε για λίγο `unshift $scalar, ...` με σημασία «θεώρησε το `$scalar` ως αναφορά πίνακα και προπένθεσε»· αφαιρέθηκε στην Perl 5.24 και παραμένει αφαιρεμένο. Γράφετε `unshift @$scalar, ...`.
- **Η τιμή επιστροφής σε βαθμωτό έναντι περιβάλλοντος λίστας** είναι ο ίδιος ακέραιος. Σε αντίθεση με πολλές ενσωματωμένες, η `unshift` δεν συμπεριφέρεται διαφορετικά σε περιβάλλον λίστας - επιστρέφει πάντα το νέο μήκος.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `push` - η άλλη άκρη: προσαρτά στην ουρά του πίνακα και επιστρέφει το νέο μήκος
- `shift` - η αντίστροφη πράξη: αφαιρεί και επιστρέφει το πρώτο στοιχείο
- `pop` - συμμετρικός σύντροφος της `shift`, αφαιρεί από την ουρά
- `splice` - γενικής χρήσης εισαγωγή/αφαίρεση σε αυθαίρετη θέση· η `unshift @a, LIST` είναι ισοδύναμη με `splice(@a, 0, 0, LIST)`
- `reverse` - συνδυάστε με την `unshift` όταν θέλετε οι προπενθεμένες τιμές να καταλήξουν σε αντίστροφη σειρά
- `List::Util` - δεν διαθέτει ειδική prepend· για σημασιολογία εισαγωγής στην κεφαλή η `unshift` είναι το ιδιωματικό εργαλείο, και οι `List::Util::pairs` /

`List::Util::zip` συνθέτονται μαζί της όταν αναδιαρθρώνετε σχήματα λίστας αντί να απλώς προσθέτετε σε μία

Κλάσεις και αντικειμενοστρέφεια

untie

Σπάει τη σύνδεση μεταξύ μιας μεταβλητής και της tied κλάσης της.

Η `untie` αντιστρέφει μια προηγούμενη `tie`: μετά την κλήση, η `VARIABLE` επανέρχεται σε απλό βαθμωτό, πίνακα, hash, ή filehandle, και οι επόμενες αναγνώσεις και εγγραφές αγγίζουν τον ωμό χώρο αποθήκευσης αντί να δρομολογούνται μέσω των `FETCH`, `STORE`, και συγγενών του tied πακέτου. Αν η μεταβλητή δεν είναι tied, η `untie` είναι no-op.

Σύνοψη

```
untie VARIABLE
```

Τι επιστρέφεται

Επιστρέφει αληθή τιμή αν η μεταβλητή ήταν tied και η σύνδεση αφαιρέθηκε, ψευδή διαφορετικά. Η τιμή επιστροφής σπάνια ελέγχεται - τα σημεία κλήσης γνωρίζουν αν ξετυλίγουν tie που έστησαν τα ίδια.

Ως παρενέργεια, αν το tied πακέτο ορίζει μέθοδο `UNTIE`, αυτή καλείται στο tied αντικείμενο πριν εγκαταλειφθεί η σύνδεση. Η `UNTIE` λαμβάνει το αντικείμενο και έναν μετρητή τυχόν εκκρεμών εσωτερικών αναφορών που κρατούνται σε αυτό (δείτε *Οριακές περιπτώσεις* παρακάτω).

Τι κάνει

1. Εντοπίζει τη μαγεία στη `VARIABLE` που εγκατέστησε μια προηγούμενη `tie`.
2. Αν το tied πακέτο παρέχει `UNTIE`, την καλεί στο αντικείμενο.
3. Αφαιρεί τη μαγεία ώστε η μεταβλητή να επανέλθει σε συνηθισμένο χώρο αποθήκευσης.
4. Απελευθερώνει την αναφορά του διερμηνέα στο tied αντικείμενο. Το αντικείμενο καταστρέφεται (πυροδοτώντας τη `DESTROY` αν είναι ορισμένη) μόλις δεν παραμένουν άλλες αναφορές προς αυτό.

Μετά την `untie`, οποιαδήποτε τιμή είχε επιστρέψει προηγουμένως η `tied` για αυτή τη μεταβλητή δεν αντιστοιχεί πλέον σε ζωντανή σύνδεση.

Παραδείγματα

Tie ενός hash, χρήση του, και μετά απελευθέρωσή του:

```
use Tie::File;
tie my @lines, 'Tie::File', 'log.txt' or die $!;
push @lines, "new entry";
untie @lines;                       # flush and release
```

Πλήρης κύκλος μέσω της `tied` - πιάστε το υποκείμενο αντικείμενο ώστε να μπορέσετε να καλέσετε μια μη-tied μέθοδο σε αυτό, και μετά εγκαταλείψτε τη σύνδεση:

```
my $obj = tied %cache;
$obj->flush;
untie %cache;
```

Η `untie` σε μεταβλητή που δεν έγινε ποτέ tied είναι αβλαβής:

```
my %h;
untie %h;                               # no-op, no warning
```

Hook `UNTIE` που εκτελείται πριν εξαφανιστεί η σύνδεση:

```
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)
```

Οριακές περιπτώσεις

- **Εκκρεμείς εσωτερικές αναφορές**: αν κώδικας έχει αποθηκεύσει το αντικείμενο που επέστρεψε η `tie` ή η `tied` κάπου που ακόμα το κρατά τη στιγμή που εκτελείται η `untie`, η Perl εκπέμπει

  ```
  untie attempted while N inner references still exist
  ```

  υπό `use warnings`. Η σύνδεση αφαιρείται έτσι κι αλλιώς, αλλά αυτές οι αποθηκευμένες αναφορές συνεχίζουν να δείχνουν στο πλέον αποσπασμένο αντικείμενο. Εγκαταλείψτε αυτές τις αναφορές πρώτα, ή αποδεχθείτε ότι θα δουν τώρα το un-tied αντικείμενο μεμονωμένα.

- **Η κλήση της `untie` σε μη-tied μεταβλητή** είναι no-op. Καμία προειδοποίηση, κανένα σφάλμα, καμία κλήση μεθόδου.

- **Tie filehandle**: η `untie *FH` αφαιρεί την tie που εγκατέστησε η `tie *FH, $class, ...`. Το υποκείμενο glob και οποιοσδήποτε πραγματικός περιγραφέας αρχείου πίσω του δεν επηρεάζονται - η `untie` δεν κάνει `close`.

- **Η `UNTIE` είναι προαιρετική**: αν το tied πακέτο δεν ορίζει μία, η σύνδεση απλώς αφαιρείται. Μη βασίζεστε στο ότι η `UNTIE` θα εκτελεστεί· πολλές tied κλάσεις την παραλείπουν.

- **Σειρά καταστροφής**: η `untie` δεν καλεί η ίδια τη `DESTROY`. Το αντικείμενο καταστρέφεται όταν ο μετρητής αναφορών του φτάσει στο μηδέν, που μπορεί να είναι αμέσως μετά την `untie` (αν τίποτα άλλο δεν το κρατά) ή αργότερα (αν ο καλών κράτησε αντίγραφο από την `tied`).

- **Τοπικοποιημένες tied μεταβλητές**: μια tie εγκατεστημένη μέσα σε εμβέλεια σε `local`οποιημένη μεταβλητή αναιρείται αυτόματα όταν εξέρχεται η εμβέλεια· δεν χρειάζεστε ρητή `untie`, και η κλήση μίας είναι αβλαβής.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `tie` - εγκαθιστά τη σύνδεση που αφαιρεί η `untie`· δείχνει το πλήρες σύνολο ονομάτων μεθόδων tie-hook
- `tied` - ανακτά το αντικείμενο που είναι αυτή τη στιγμή συνδεδεμένο σε tied μεταβλητή, τυπικά ακριβώς πριν την κλήση `untie` σε αυτήν
- `bless` - η `tie` κάνει bless το αντικείμενο που της δίνεται· η `untie` δεν κάνει unbless, μόνο αποσυνδέει τη μεταβλητή από αυτό
- `ref` - εξετάστε τι είδος αναφοράς επέστρεψε η `tied` πριν αποφασίσετε να κάνετε `untie`
- `local` - εναλλακτική με όρια εμβέλειας όταν θέλετε η tie να εξαφανιστεί κατά την έξοδο μπλοκ χωρίς ρητή `untie`

Εμβέλεια · Αρθρώματα · Κλάσεις και αντικειμενοστρέφεια

use

Φορτώνει ένα άρθρωμα κατά τη μεταγλώττιση και εισάγει τα σύμβολά του στο τρέχον πακέτο.

Η `use` είναι ο καθημερινός μηχανισμός για τη φόρτωση κώδικα βιβλιοθήκης. Εκτελείται **κατά τη μεταγλώττιση**, όχι σε χρόνο εκτέλεσης: το κατονομαζόμενο άρθρωμα εντοπίζεται στο `@INC`, φορτώνεται (αν δεν είναι ήδη στο `%INC`), και του δίνεται η ευκαιρία να αλλοιώσει τον χώρο ονομάτων του καλούντος - τυπικά ψευδωνυμοποιώντας ονόματα υπορουτινών ή μεταβλητών μέσα σε αυτόν. Η ίδια σύνταξη οδηγεί επίσης τις **pragmas** (οδηγίες του μεταγλωττιστή όπως οι `strict` ή `warnings`)· η μόνη διαφορά είναι σύμβαση - οι pragmas γράφονται με πεζά και συνήθως έχουν λεκτικές επιδράσεις, όχι σε όλο το πακέτο.

Μια δήλωση `use` ορίζεται ως ακριβώς ισοδύναμη με:

```
BEGIN { require Module; Module->import(LIST); }
```

Το `BEGIN` εξαναγκάζει τη φόρτωση και την εισαγωγή να συμβούν ενώ το περιβάλλον αρχείο αναλύεται ακόμη, ώστε τα εισαγόμενα ονόματα να είναι ορατά στον μεταγλωττιστή για το υπόλοιπο του αρχείου.

Σύνοψη

```
use Module VERSION LIST
use Module VERSION
use Module LIST
use Module
use Module ()
use VERSION
```

Τι επιστρέφεται

Η `use` είναι δήλωση, όχι έκφραση - δεν έχει χρήσιμη τιμή επιστροφής και δεν μπορεί να εμφανιστεί στη δεξιά πλευρά κανενός. Η επίδρασή της είναι **παρενέργεια στην τρέχουσα μονάδα μεταγλώττισης**: το `%INC` αποκτά εγγραφή, το εισάγον πακέτο αποκτά όποια σύμβολα ψευδωνυμοποίησε η `import`, και - για την `use VERSION` - η τρέχουσα λεκτική εμβέλεια αποκτά ένα ενεργοποιημένο πακέτο χαρακτηριστικών.

Αν το άρθρωμα δεν μπορεί να φορτωθεί, ή αν η `import` πεθάνει, το μπλοκ `BEGIN` διαδίδει την εξαίρεση και η μεταγλώττιση ματαιώνεται.

Οι τέσσερις μορφές για άρθρωμα

use Module

Φορτώνει το άρθρωμα και καλεί τη μέθοδο `import` του **χωρίς ορίσματα**. Το άρθρωμα αποφασίζει τι σημαίνει «χωρίς ορίσματα»· για αρθρώματα βασισμένα στον `Exporter` τυπικά σημαίνει «εξάγαγε τις προεπιλογές `@EXPORT` σου»:

```
use File::Spec;                 # calls File::Spec->import()
```

use Module LIST

Φορτώνει το άρθρωμα και καλεί την `import` με `LIST`:

```
use List::Util qw(first sum max);
```

Αυτή είναι η συνηθισμένη μορφή «εισήγαγε ακριβώς αυτά τα ονόματα». Η `LIST` διαβιβάζεται αυτούσια στην `import`· το νόημά της είναι υπόθεση του αρθρώματος, όχι της γλώσσας.

use Module ()

Οι κενές παρενθέσεις απενεργοποιούν εντελώς την κλήση της import. Αυτό διαφέρει από την παράλειψη της λίστας - δεν καλείται καθόλου η μέθοδος `import`:

```
use POSIX ();                   # load, but do not import anything
POSIX::strftime(...);           # call by full name instead
```

Ισοδύναμο με:

```
BEGIN { require POSIX }
```

Χρησιμοποιήστε το όταν θέλετε το άρθρωμα φορτωμένο αλλά δεν θέλετε να αλλοιωθεί ο χώρος ονομάτων σας - τυπικά επειδή θα καλείτε τα πάντα με πλήρως κατονομασμένο όνομα, ή επειδή οι προεπιλεγμένες εξαγωγές του αρθρώματος θα αλλοίωναν κάτι που σας ενδιαφέρει.

use Module VERSION / use Module VERSION LIST

Όταν μια VERSION εμφανίζεται αμέσως μετά το Module (χωρίς κόμμα), η use καλεί τη μέθοδο VERSION του αρθρώματος με αυτή την έκδοση ως όρισμα **πριν** καλέσει την import:

```
use List::Util 1.45 qw(uniq);
```

είναι ισοδύναμο με:

```
BEGIN {
    require List::Util;
    List::Util->VERSION(1.45);
    List::Util->import(qw(uniq));
}
```

Η προεπιλεγμένη μέθοδος VERSION, κληρονομημένη από το UNIVERSAL, κράζει αν η ζητούμενη έκδοση είναι μεγαλύτερη από την $Module::VERSION.

Η VERSION πρέπει να μοιάζει με κυριολεκτικό έκδοσης - ψηφίο, ή ν ακολουθούμενο από ψηφίο. Αυθαίρετες εκφράσεις σε αυτή τη θέση αναλύονται ως αρχή της LIST, όχι ως έκδοση. Σημειώστε ότι **δεν υπάρχει κόμμα μετά τη VERSION**.

use VERSION - αίτημα για επίπεδο γλώσσας Perl

Η use VERSION (χωρίς όνομα αρθρώματος) δηλώνει την ελάχιστη έκδοση Perl που απαιτεί το αρχείο, και **ενεργοποιεί λεκτικά το αντίστοιχο πακέτο χαρακτηριστικών**:

```
use v5.36;                      # v-string form - preferred
use 5.036;                      # numeric form - equivalent
use 5.036_000;                  # older numeric form
```

Εγείρεται εξαίρεση κατά τη μεταγλώττιση αν η τρέχουσα Perl είναι παλαιότερη από τη VERSION· το υπόλοιπο του αρχείου δεν αναλύεται καν.

Πέρα από τον έλεγχο έκδοσης, η use VERSION ενεργοποιεί τα χαρακτηριστικά που συνοδεύουν αυτή την έκδοση:

- **use v5.12** ή ανώτερη - υπονοούμενο use strict.
- **use v5.35** ή ανώτερη - υπονοούμενο use warnings.
- **use v5.36** - η συνηθισμένη σύγχρονη βάση: προσθέτει τα say, signatures, isa, και τα χαρακτηριστικά των προηγούμενων πακέτων.
- **use v5.39** ή ανώτερη - εισάγει λεκτικά συναρτήσεις builtin για το αντίστοιχο πακέτο.
- **use v5.41** ή ανώτερη - ενεργοποιεί την source::encoding 'ascii'.

Η use VERSION **δεν** φορτώνει τα feature.pm, strict.pm, warnings.pm, ή builtin.pm - υλοποιεί την ισοδύναμη συμπεριφορά απευθείας. Τοποθετήστε την στην αρχή του αρχείου, πριν από οτιδήποτε εξαρτάται από την ενεργοποίηση αυτών των χαρακτηριστικών.

Από την Perl 5.39 και μετά, η έκδοση δεύτερης use VERSION ενώ μία είναι ήδη ενεργή είναι μοιραίο σφάλμα. Σε παλαιότερα πακέτα ήταν απλώς αποθαρρυμένη.

Αρθρώματα έναντι pragmas

Συντακτικά, οι pragmas και τα αρθρώματα είναι πανομοιότυπα - και τα δύο είναι απλώς πακέτα των οποίων τη μέθοδο `import` επικαλείστε μέσω `use`. Οι συμβατικές διαφορές:

- **Ονοματοδοσία.** Οι pragmas γράφονται με πεζά (`strict`, `warnings`, `feature`, `integer`)· τα αρθρώματα είναι `Studly::Caps`.
- **Εμβέλεια της επίδρασης.** Οι pragmas συνήθως αλλοιώνουν την **τρέχουσα λεκτική εμβέλεια** (ένα μπλοκ, όχι πακέτο) και ξετυλίγονται στο κλείσιμο της αγκύλης. Τα αρθρώματα συνήθως αλλοιώνουν το **τρέχον πακέτο** και παραμένουν μέχρι το τέλος του αρχείου.
- **no.** Η αντίστοιχη δήλωση `no` καλεί την `unimport` αντί της `import`. Οι pragmas τυπικά υλοποιούν την `unimport` για να αναιρούν την επίδρασή τους· τα περισσότερα συνηθισμένα αρθρώματα όχι.

```
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::
```

Υπό συνθήκη φόρτωση

Επειδή η `use` εκτελείται κατά τη μεταγλώττιση, **αγνοεί τη συνηθισμένη ροή ελέγχου χρόνου εκτέλεσης**. Μια `use` μέσα στην ψευδή διακλάδωση ενός `if` εκτελείται ούτως ή άλλως:

```
if (0) {
    use Heavy::Module;          # loaded anyway - compile time!
}
```

Για πραγματικά υπό συνθήκη φόρτωση, χρησιμοποιήστε είτε την `require` σε χρόνο εκτέλεσης, είτε αναθέστε την στην pragma `if`:

```
use if $] >= 5.036, 'experimental', 'builtin';
use if $^O eq 'MSWin32', 'Win32::API';
```

Παραδείγματα

Φόρτωση αρθρώματος με τις προεπιλεγμένες του εξαγωγές:

```
use Carp;                       # croak, carp, confess visible here
```

Εισαγωγή συγκεκριμένου υποσυνόλου:

```
use List::Util qw(first min max sum);
```

Φόρτωση χωρίς εισαγωγή - κλήση με πλήρες όνομα:

```
use Scalar::Util ();
my $ok = Scalar::Util::looks_like_number($x);
```

Απαίτηση ελάχιστης έκδοσης αρθρώματος:

```
use Moo 2.000000;
```

Δήλωση του επιπέδου γλώσσας Perl και απόκτηση του πακέτου χαρακτηριστικών του:

```
use v5.36;                      # strict, warnings, say, signatures.
↪..
sub greet ($name) {
    say "hello, $name";
}
```

Ενεργοποιήστε μια pragma για ένα αρχείο, στη συνέχεια απενεργοποιήστε την για ένα μπλοκ:

```
use warnings;

sub legacy_parser {
    no warnings 'uninitialized';
    return join ",", @_;        # undefs render as "" without␣
↪warnings
}
```

Υπό συνθήκη pragma μέσω του βοηθού `if`:

```
use if $] < 5.010, 'MRO::Compat';
```

Οριακές περιπτώσεις

- **Το `Module` πρέπει να είναι bareword.** Το `use $name;` είναι συντακτικό σφάλμα - η `use` δεν δέχεται έκφραση χρόνου εκτέλεσης για το όνομα αρθρώματος. Χρησιμοποιήστε την `require` γι" αυτό.
- **Κανένα κόμμα μεταξύ `Module` και `VERSION`.** Το `use List::Util 1.45` είναι έλεγχος έκδοσης· το `use List::Util, 1.45` είναι συντακτικό σφάλμα.
- **Ο κανόνας του κυριολεκτικού έκδοσης είναι αυστηρός.** Οτιδήποτε δεν ξεκινά με ψηφίο ή με `v` ακολουθούμενο από ψηφίο αναλύεται ως αρχή της `LIST`, όχι ως έκδοση. Το `use Module "1.45"` διαβιβάζει τη συμβολοσειρά `"1.45"` στην `import`· **δεν** ελέγχει έκδοση.
- **Κενή `LIST` έναντι παραλειπόμενης `LIST`.** Το `use Module;` καλεί την `Module->import()`. Το `use Module ();` δεν καλεί καθόλου την `import`. Η διάκριση έχει σημασία για αρθρώματα των οποίων η `import` έχει παρενέργειες ακόμη και χωρίς ορίσματα.
- **Λείπει η μέθοδος `import`.** Αν το άρθρωμα δεν ορίζει (και δεν κληρονομεί) μέθοδο `import`, η κλήση παρακάμπτεται σιωπηλά - κανένα σφάλμα, ακόμη και αν είναι ορισμένη η `AUTOLOAD`.

- **use μέσα σε eval.** Το BEGIN εκτελείται κατά τη μεταγλώττιση του κώδικα της eval, οπότε το eval "use Some::Module" καθυστερεί τη φόρτωση μέχρι να τρέξει η eval. Αυτός είναι ο συνηθισμένος τρόπος για ανίχνευση ενός προαιρετικού αρθρώματος σε χρόνο εκτέλεσης:

```
eval { require Some::Module; Some::Module->import };
my $have_it = !$@;
```

- **Σειρά της use VERSION και άλλων pragmas.** Τοποθετήστε την use VERSION πρώτη, ώστε το πακέτο χαρακτηριστικών να είναι σε ισχύ πριν τρέξουν επόμενες pragmas (που μπορεί να εξαρτώνται από αυτό). Ένα μεταγενέστερο no strict 'refs' εξακολουθεί να υπερισχύει των αυστηροτήτων που υπονοεί η use v5.12+, αλλά το να βασίζεται κανείς σε αυτό αποθαρρύνεται.
- **Pragmas που μοιάζουν με αρθρώματα.** Κάποιες pragmas διανέμονται με όνομα σε κεφαλαία (UNIVERSAL, Internals)· κάποια αρθρώματα διανέμονται με πεζά (if, lib, ok). Η σύμβαση ονοματοδοσίας είναι ένδειξη, όχι κανόνας - αυτό που έχει σημασία είναι το τι κάνει η import του αρθρώματος.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- require - το χρόνου εκτέλεσης μισό της use· φορτώνει άρθρωμα χωρίς να επικαλείται την import, και δέχεται έκφραση χρόνου εκτέλεσης για το όνομα του αρθρώματος
- no - η αντίστοιχη δήλωση· καλεί την unimport αντί της import για να αναιρέσει την επίδραση μιας pragma
- import - δεν είναι ενσωματωμένη· συνηθισμένη μέθοδος κλάσης που επικαλείται η use, συνήθως παρεχόμενη μέσω κληρονομικότητας από τον Exporter
- package - δηλώνει το πακέτο του οποίου τον χώρο ονομάτων θα αλλοιώσει η use
- @INC - η λίστα καταλόγων που ερευνώνται για το αρχείο του αρθρώματος
- %INC - καταγράφει κάθε αρχείο που φορτώθηκε από use ή require, με κλειδί τη σχετική διαδρομή

Filehandles, αρχεία, κατάλογοι

utime

Ορίζει χρόνους πρόσβασης και τροποποίησης σε μια λίστα αρχείων.

Η utime εφαρμόζει τον αριθμητικό χρόνο πρόσβασης ATIME και χρόνο τροποποίησης MTIME - και τους δύο σε δευτερόλεπτα Unix epoch - σε κάθε αρχείο που κατονομάζεται στη LIST. Είναι η διεπαφή σε επίπεδο Perl προς τις κλήσεις συστήματος utime(2) και utimes(2), και το εργαλείο πίσω από την εντολή Unix touch(1). Ο χρόνος αλλαγής inode (ctime) κάθε αρχείου τίθεται στον τρέχοντα χρόνο ως παρενέργεια - αυτή είναι αναλλοίωτη του πυρήνα, όχι επιλογή της Perl, και δεν μπορεί να κατασταλεί.

Σύνοψη

```
utime ATIME, MTIME, LIST
utime undef, undef, LIST      # set both to "now" (since 5.8.0)
```

Τι επιστρέφεται

Ο αριθμός των αρχείων των οποίων οι χρονικές σφραγίδες ενημερώθηκαν επιτυχώς. Όποιο αρχείο δεν μπόρεσε να αγγιχτεί παραλείπεται σιωπηρά από το πλήθος· η `$!` αντικατοπτρίζει το σφάλμα μόνο από το *τελευταίο* αρχείο που απέτυχε. Συγκρίνετε την τιμή επιστροφής με `scalar @files` όταν χρειάζεστε να εντοπίσετε μερική αποτυχία:

```
my @files = ("a", "b", "c");
my $n = utime $atime, $mtime, @files;
warn "only $n of ", scalar @files, " files updated: $!"
    if $n != @files;
```

Ορίσματα

- `ATIME` - νέος χρόνος πρόσβασης, σε δευτερόλεπτα από το Unix epoch. Πρέπει να είναι αριθμητικός· τα μη αριθμητικά βαθμωτά εξαναγκάζονται με τον συνήθη τρόπο και μια συμβολοσειρά όπως `"now"` γίνεται `0` (μεσάνυχτα 1970-01-01).
- `MTIME` - νέος χρόνος τροποποίησης, ίδια μορφή.
- `LIST` - ένα ή περισσότερα ονόματα αρχείων. Σε συστήματα που υποστηρίζουν την `futimes(2)` (όπως το Linux), filehandles μπορούν επίσης να εμφανίζονται στη `LIST`. Τα filehandles πρέπει να περνούν ως globs (`*FH`) ή ως αναφορές glob (`\*FH`)· ένα bareword σε εκείνη τη θέση αναλύεται ως όνομα αρχείου, όχι ως handle.

Η μορφή `undef, undef`

Από την Perl 5.8.0, το πέρασμα `undef` και για το `ATIME` και για το `MTIME` καλεί την `utime(2)` με null δεύτερο όρισμα. Ο πυρήνας τότε θέτει και τους δύο χρόνους στον τρέχοντα πραγματικό χρόνο, και - κρίσιμα - αυτή η μορφή πετυχαίνει σε αρχεία που δεν ανήκουν στον καλούντα, εφόσον ο καλών έχει **δικαίωμα εγγραφής** στο αρχείο. Η απαίτηση ιδιοκτήτη-ή-root ισχύει μόνο για τη μορφή με ρητές χρονικές σφραγίδες.

```
for my $file (@ARGV) {
    utime undef, undef, $file
        or warn "Couldn't touch $file: $!";
}
```

Αυτή είναι η μορφή που η πραγματική εντολή `touch(1)` χρησιμοποιεί εσωτερικά, και είναι η σωστή επιλογή για ένα γενικό σενάριο «touch».

Δικαιώματα

Δύο διακριτοί κανόνες, ανάλογα με τη μορφή που χρησιμοποιείτε:

Μορφή	Ποιος μπορεί να πετύχει
`utime $atime, $mtime, ...`	Ιδιοκτήτης αρχείου, ή root
`utime undef, undef, ...`	Οποιοσδήποτε έχει δικαίωμα εγγραφής στο αρχείο

Ένας μη-ιδιοκτήτης που επιχειρεί τη μορφή με ρητές χρονικές σφραγίδες λαμβάνει EPERM στην `$!` και το αρχείο μένει ανέπαφο (παρ” όλα αυτά μετράει ως αποτυχία, όχι ως σιωπηρό no-op).

Παραδείγματα

Σκέτο Unix-style `touch` σε λίστα αρχείων που σας ανήκουν:

```
my $now = time;
utime $now, $now, @ARGV;
```

Γενικό `touch` που λειτουργεί επίσης σε αρχεία στα οποία απλώς έχετε δικαίωμα εγγραφής:

```
for my $file (@ARGV) {
    utime undef, undef, $file
        or warn "Couldn't touch $file: $!";
}
```

Αντιγραφή χρονικών σφραγίδων από ένα αρχείο σε άλλο μέσω της `stat`:

```
my @st = stat $src or die "stat $src: $!";
utime $st[8], $st[9], $dst
    or die "utime $dst: $!";
```

Touch σε ήδη ανοιχτό filehandle σε Linux (χρειάζεται `futimes(2)`):

```
open my $fh, ">>", "log" or die $!;
utime undef, undef, *$fh;       # glob deref, not bareword
```

Ορισμός αρχείου μία ώρα στο παρελθόν, χρήσιμο για ελέγχους συστήματος build:

```
my $t = time - 3600;
utime $t, $t, "stale.tmp";
```

Οριακές περιπτώσεις

- **Ανάμικτο undef και αριθμός**: το πέρασμα undef για ένα μόνο από τα δύο πρώτα ορίσματα **δεν** ενεργοποιεί τη διαδρομή «και-τα-δύο-undef». Το undef εξαναγκάζεται σε `0` (μεσάνυχτα epoch 1970-01-01), η άλλη τιμή χρησιμοποιείται αυτούσια, και ενεργοποιείται προειδοποίηση `uninitialized` υπό `use warnings`. Αυτό σχεδόν ποτέ δεν είναι αυτό που θέλετε.

```
use warnings;
utime undef, time, $file;       # atime = 1970-01-01, warning
```

- **Μη αριθμητικές χρονικές σφραγίδες**: τα `ATIME` και `MTIME` πρέπει να είναι αριθμητικά. Τιμές συμβολοσειράς εξαναγκάζονται - το `"1700000000"` λειτουργεί· το `"yesterday"` γίνεται σιωπηρά `0`.
- **Ο `ctime` πάντα μετακινείται**: δεν υπάρχει τρόπος να διατηρηθεί ο χρόνος αλλαγής inode ενός αρχείου σε μια κλήση `utime`. Αν το χρειάζεστε, χρειάζεστε εργαλεία σε επίπεδο συστήματος αρχείων ή στιγμιότυπο, όχι Perl.
- **Filehandle στη `LIST`**: πρέπει να είναι glob (`*FH`) ή αναφορά glob (`\*FH`). Ένα bareword όπως `FH` σε εκείνη τη θέση αναλύεται ως το όνομα αρχείου `"FH"`, και ένα βαθμωτό συμβολοσειράς που κρατά όνομα handle αντιμετωπίζεται ομοίως ως όνομα αρχείου.
- **Ανύπαρκτο αρχείο**: μετράει ως αποτυχία και δεν αυξάνει την τιμή επιστροφής· η `$!` τίθεται σε `ENOENT` (`No such file or directory`).
- **Μερική αποτυχία**: το πλήθος αντικατοπτρίζει μόνο επιτυχίες. Για να ξέρετε *ποιο* αρχείο απέτυχε, διατρέξτε τη λίστα ένα τη φορά.
- **Απόκλιση ρολογιού NFS**: το ρολόι του διακομιστή είναι το έγκυρο, όχι του πελάτη. Μια αξιοσημείωτη διαφορά μεταξύ των δύο εμφανίζεται ως χρονικές σφραγίδες που δεν ταιριάζουν με ό,τι επέστρεψε η `time` στον πελάτη.
- **Ακρίβεια κάτω του δευτερολέπτου**: η `utime` της Perl λειτουργεί σε ακέραια δευτερόλεπτα. Συστήματα αρχείων και πυρήνες που υποστηρίζουν χρονικές σφραγίδες nanosecond θα δουν το κλασματικό μέρος να μηδενίζεται. Καταφύγετε στην `Time::HiRes::utime` (XS) αν χρειάζεστε καλύτερη ανάλυση.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `stat` - διαβάζει τις χρονικές σφραγίδες που γράφει η `utime`· στοιχεία 8 (`atime`), 9 (`mtime`) και 10 (`ctime`) της επιστρεφόμενης λίστας
- `time` - τρέχοντα δευτερόλεπτα epoch, η συνήθης πηγή και για το `ATIME` και για το `MTIME`
- `open` και `close` - filehandles που μπορεί να αγγίξει η `utime` σε συστήματα με `futimes(2)`
- `$!` - σφάλμα συστήματος από το τελευταίο αρχείο που απέτυχε όταν το πλήθος επιστροφής είναι μικρότερο από `scalar @files`
- `utime(2)`, `utimes(2)`, `futimes(2)` - οι υποκείμενες κλήσεις συστήματος

Πίνακες · Hashes

values

Απαριθμεί κάθε τιμή ενός hash (ή κάθε στοιχείο ενός πίνακα).

Η `values` είναι ο συνεργάτης από την πλευρά των τιμών για την `keys`. Δοθέντος ενός hash, επιστρέφει όλες τις τιμές· δοθέντος ενός πίνακα, επιστρέφει όλα τα στοιχεία. Τα στοιχεία επιστρέφονται ως **ψευδώνυμα** μέσα στον υποκείμενο περιέκτη, οπότε η ανάθεση σε αυτά από βρόχο τροποποιεί τον περιέκτη επιτόπου. Ως παρενέργεια, η κλήση της `values` επαναφέρει τον επαναλήπτη που μοιράζονται οι `each`, `keys` και `values` για αυτό το hash ή τον πίνακα.

Σύνοψη

```
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
```

Τι επιστρέφεται

Σε περιβάλλον λίστας: η πλήρης λίστα τιμών από hash, ή η πλήρης λίστα στοιχείων από πίνακα. Οι εγγραφές πίνακα επιστρέφονται σε σειρά θέσεων (μικρότερη θέση πρώτη). Οι εγγραφές hash επιστρέφονται σε ορισμένη από την υλοποίηση, τυχαία ανά hash σειρά - δείτε *Οριακές περιπτώσεις*.

Σε βαθμωτό περιβάλλον: το πλήθος εγγραφών, πανομοιότυπο με το `scalar keys %h` για hashes και `scalar @a` για πίνακες. Η `values` σε κενό περιβάλλον δεν κάνει τίποτα παρατηρήσιμο **εκτός** από το να επαναφέρει τον επαναλήπτη - αυτός είναι ο ιδιωματικός τρόπος βίαιης επαναφοράς επαναλήπτη στη μέση διάσχισης:

```
values %h;                       # void context: iterator reset only
```

Η επιστρεφόμενη λίστα δεν είναι αντίγραφο. Κάθε στοιχείο είναι ψευδώνυμο προς τη ζωντανή τιμή στον περιέκτη. Η ψευδωνυμοποίηση είναι τεκμηριωμένη συμπεριφορά και αποτελεί τη βάση για το ιδίωμα επιτόπιας τροποποίησης στην ενότητα *Παραδείγματα*.

Καθολική κατάσταση που επηρεάζει

Η `values` επαναφέρει την κατάσταση επαναλήπτη ανά περιέκτη που διαβάζει και προωθεί η `each`. Οποιοσδήποτε βρόχος `while (my ($k, $v) = each %h) { ... }` σε εξέλιξη στο ίδιο hash θα ξεκινήσει από την αρχή την επόμενη φορά που θα κληθεί η `each`. Η `keys` έχει το ίδιο αποτέλεσμα. Ο επαναλήπτης είναι ιδιότητα του περιέκτη, όχι κανενός συγκεκριμένου σημείου κλήσης - οπότε ένας καλών σε μία sub μπορεί να ενοχλήσει βρόχο `each` που τρέχει σε άλλη sub αν λειτουργούν στο ίδιο hash.

Παραδείγματα

Επανάληψη σε hash σε ταξινομημένη ως προς το κλειδί σειρά, κάτι που η `values` από μόνη της δεν σας δίνει:

```
my %score = (alice => 90, bob => 75, carol => 82);
for my $k (sort keys %score) {
    print "$k: $score{$k}\n";
}
```

Άθροισμα κάθε τιμής ενός hash:

```
use List::Util qw(sum0);
my %count = (apples => 3, pears => 5, plums => 2);
my $total = sum0 values %count;     # 10
```

Τροποποίηση κάθε τιμής επιτόπου μέσω του ψευδωνύμου:

```
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)
```

Το ίδιο ιδίωμα λειτουργεί σε πίνακα, αν και για πίνακες ένας σκέτος βρόχος `@array` είναι πιο συμβατικός:

```
my @nums = (1, 2, 3, 4);
$_ **= 2 for values @nums;          # @nums is now (1, 4, 9, 16)
```

Βίαιη επαναφορά επαναλήπτη `each` σε εξέλιξη χωρίς κατανάλωση της εξόδου του:

```
while (my ($k, $v) = each %h) {
    last if $v > $threshold;
}
values %h;                          # void context: drop iterator
 ↪state
# next each %h starts from the beginning
```

Καταμέτρηση εγγραφών χωρίς δημιουργία της λίστας - το βαθμωτό περιβάλλον ποτέ δεν υλοποιεί τις τιμές:

```
if (scalar values %cache > 10_000) {
    prune(\%cache);
}
```

Οριακές περιπτώσεις

- **Η ψευδωνυμοποίηση είναι το χαρακτηριστικό και η παγίδα.** Το `for (values %h)` σας δίνει ψευδώνυμα, όχι αντίγραφα. Μια αθώα φαινομενικά `s///` μέσα σε τέτοιο βρόχο ξαναγράφει το hash. Αν θέλετε αντίγραφα, αναθέστε πρώτα: `my @v = values %h; for (@v) { ... }`.
- **Η σειρά hash είναι τυχαιοποιημένη.** Η Perl 5.18+ τυχαιοποιεί τη σειρά διάσχισης ανά hash. Δύο hashes χτισμένα από τα ίδια κλειδιά μπορούν να επιστρέφουν τιμές σε διαφορετική σειρά. Μη βασίζεστε ποτέ στη σειρά για οτιδήποτε φεύγει από τη διεργασία - σειριοποιήστε μέσω `sort keys` αν χρειάζεστε σταθερότητα.

- **Η σειρά είναι σταθερή μόνο όσο το hash παραμένει αμετάβλητο.** Εισαγωγές και διαγραφές ενδέχεται να μεταθέσουν τη σειρά. Η μόνη τεκμηριωμένη εξαίρεση είναι η διαγραφή του κλειδιού που επιστράφηκε πιο πρόσφατα από την `each` ή `keys`, η οποία αφήνει την υπόλοιπη σειρά ανέπαφη.
- **Οι `keys`, `values`, `each` συμφωνούν.** Για ένα hash που δεν έχει τροποποιηθεί μεταξύ κλήσεων, οι `keys %h` και `values %h` παράγουν αντίστοιχες λίστες στην ίδια σειρά - οι `(keys %h)[$i]` και `(values %h)[$i]` κατονομάζουν την ίδια εγγραφή.
- **Η μορφή για πίνακα συμπεριφέρεται όπως το `@array` συν επαναφορά.** Η `values @a` σε περιβάλλον λίστας είναι ισοδύναμη με το `@a`, με την παρενέργεια της επαναφοράς του επαναλήπτη του πίνακα. Για μόνη την επαναφορά επαναλήπτη, η `keys @a` σε κενό περιβάλλον είναι η συμβατική επιλογή.
- **Κενό hash ή πίνακας** επιστρέφει την κενή λίστα σε περιβάλλον λίστας και `0` σε βαθμωτό περιβάλλον. Καμία προειδοποίηση, κανένα σφάλμα.
- **Tied hashes.** Η συμπεριφορά ανατίθεται στις μεθόδους `FIRSTKEY` / `NEXTKEY` / `FETCH` της κλάσης tie. Η σειρά, η ψευδωνυμοποίηση και η σημασιολογία επαναφοράς επαναλήπτη ακολουθούν ό,τι παρέχει η υλοποίηση tie - οι εγγυήσεις του καθαρού hash παραπάνω δεν μεταφέρονται αυτόματα.
- **Αφαιρεμένο πείραμα: βαθμωτό όρισμα.** Η Perl 5.14–5.22 δεχόταν πειραματικά το `values $ref`· αυτό το πείραμα αφαιρέθηκε στην 5.24. Κάντε dereference ρητά: `values %$ref`, `values @$ref`.
- **Η υποστήριξη για πίνακες απαιτεί 5.12+.** Η εκτέλεση `values @a` σε παλαιότερη Perl εγείρει συντακτικό σφάλμα. Χρησιμοποιήστε `use v5.12;` στην κορυφή αρχείου που βασίζεται στη μορφή για πίνακες για να καταστήσετε ρητή την απαίτηση.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `keys` - ο συνεργάτης από την πλευρά των κλειδιών· ίδιοι κανόνες σειράς, ίδιος επαναλήπτης, βαθμωτό περιβάλλον επιστρέφει το πλήθος
- `each` - βηματική επανάληψη που επιστρέφει ζεύγη `($key, $value)`· μοιράζεται τον επαναλήπτη που η `values` επαναφέρει
- `exists` - έλεγχος αν ένα κλειδί είναι παρόν χωρίς πυροδότηση αυτο-δημιουργίας
- `delete` - αφαιρεί ένα ζεύγος κλειδιού/τιμής· η διαγραφή του τελευταία επιστραφέντος κλειδιού της `each` ή `keys` αφήνει την υπόλοιπη σειρά διάσχισης ανέπαφη
- `map` - μετασχηματισμός τιμών σε παράγωγη λίστα χωρίς τροποποίηση του hash (η εναλλακτική χωρίς ψευδωνυμοποίηση έναντι του `for (values %h)`)
- `sort` - συνδυάστε με `keys` όταν χρειάζεστε ντετερμινιστική σειρά διάσχισης

Δεδομένα σταθερού μήκους

vec

Διαβάζει ή γράφει μια θυρίδα σταθερού πλάτους μέσα σε μια συμβολοσειρά που αντιμετωπίζεται ως πακεταρισμένο διάνυσμα bits.

Η vec θεωρεί την EXPR ως πίνακα μη προσημασμένων ακεραίων, καθένας με πλάτος BITS, πακεταρισμένων ο ένας μετά τον άλλο από την αρχή της συμβολοσειράς. Το OFFSET ευρετηριάζει σε αυτόν τον πίνακα - όχι σε bytes, όχι σε bits, αλλά σε στοιχεία μεγέθους BITS. Η μορφή ανάγνωσης επιστρέφει τον ακέραιο σε αυτή τη θυρίδα· η μορφή lvalue γράφει έναν. Τα πλάτη είναι 1, 2, 4, 8, 16, 32, και σε builds 64-bit το 64.

Σύνοψη

```
my $n = vec($buf, $offset, $bits);
vec($buf, $offset, $bits) = $n;
```

Τι επιστρέφεται

Σε rvalue περιβάλλον, έναν μη προσημασμένο ακέραιο - τα περιεχόμενα της επιλεγμένης θυρίδας, επεκταμένα με μηδενικά σε αριθμό της Perl. Σε lvalue περιβάλλον, μια εκχωρήσιμη θυρίδα· η εκχώρηση αποκόπτει τη δεξιά τιμή σε BITS και τη γράφει στη συμβολοσειρά, επεκτείνοντας τη συμβολοσειρά με μηδενικά bytes αν το OFFSET βρίσκεται πέρα από το τρέχον τέλος.

Οι παρενθέσεις γύρω από το `vec(...)` στη μορφή lvalue είναι απαραίτητες - χωρίς αυτές, το `vec $buf, $o, $b = 3` αναλύει το = ως μέρος της λίστας ορισμάτων.

Πώς διατάσσονται τα bits

Η διάταξη εξαρτάται από τα BITS, και επιλέγεται ώστε ο κώδικας να είναι φορητός μεταξύ μηχανών big- και little-endian:

- **`BITS == 8`**: κάθε θυρίδα είναι ένα byte της συμβολοσειράς. Το `vec($s, $i, 8)` είναι η μη προσημασμένη τιμή του `substr($s, $i, 1)`.
- **`BITS == 16, 32, 64`**: τα bytes της συμβολοσειράς ομαδοποιούνται σε κομμάτια των BITS/8 και ερμηνεύονται σε σειρά **big-endian** - ισοδύναμο με την unpack με n, N, ή (σε builds 64-bit) Q> / το ηθικό ισοδύναμο. Το `vec($s, 0, 32)` διαβάζει τα πρώτα τέσσερα bytes ως big-endian uint32.
- **`BITS == 4, 2, 1`**: η συμβολοσειρά σπάει σε bytes, και κάθε byte χωρίζεται σε 8/BITS ομάδες, αριθμημένες **little-endian-ish** εντός του byte. Οι τιμές bit από χαμηλό σε υψηλό είναι `0x01`, `0x02`, `0x04`, `0x08`, `0x10`, `0x20`, `0x40`, `0x80`. Έτσι για το `chr(0x36)` (`0b00110110`):
  - Το BITS == 4 δίνει τα δύο nibbles `(0x6, 0x3)`.
  - Το BITS == 2 δίνει τις τέσσερις ομάδες των 2 bits `(0x2, 0x1, 0x3, 0x0)`.
  - Το BITS == 1 δίνει τα οκτώ bits `(0, 1, 1, 0, 1, 1, 0, 0)`.

Μια θυρίδα εντελώς εκτός του τέλους της συμβολοσειράς διαβάζεται ως 0. Η εγγραφή πέρα από το τέλος μεγαλώνει τη συμβολοσειρά με μηδενικά bytes για να φτάσει τη θυρίδα. Ένα αρνητικό OFFSET είναι μοιραίο σφάλμα.

Καθολική κατάσταση που επηρεάζει

Καμία. Η vec λειτουργεί καθαρά επί των ορισμάτων της.

Παραδείγματα

Διαβάστε ένα byte σε δοσμένο δείκτη:

```
my $s = "Perl";
print vec($s, 0, 8);            # 80   (== ord 'P')
print vec($s, 3, 8);            # 108  (== ord 'l')
```

Κατασκευάστε μια συμβολοσειρά γράφοντας λέξεις 32-bit big-endian:

```
my $buf = '';
vec($buf, 0, 32) = 0x5065726C;  # "Perl"
vec($buf, 1, 32) = 0x50657270;  # "PerlPerp"
print $buf;                     # PerlPerp
```

Χρησιμοποιήστε τη vec ως συμπαγή πίνακα λογικών τιμών - ένα bit ανά σημαία:

```
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)
```

Μετρήστε τα ενεργοποιημένα bits σε διάνυσμα bits χωρίς βρόχο bit-προς-bit - το ιδιωματικό μοτίβο χρησιμοποιεί την unpack:

```
my $ones = unpack("%32b*", $flags);   # population count
```

Μετατρέψτε ένα διάνυσμα bits σε συμβολοσειρά από 0 και 1 για εμφάνιση:

```
my $bits = unpack("b*", $flags);      # "00...010...010..."
```

Συνδυάστε δύο διανύσματα bits με τους bitwise τελεστές συμβολοσειρών - αυτοί αντιμετωπίζουν τους τελεστέους συμβολοσειρών ως διανύσματα bits του ίδιου σχήματος που διαβάζει και γράφει η vec:

```
my $union        = $flags_a | $flags_b;
my $intersection = $flags_a & $flags_b;
my $diff         = $flags_a ^ $flags_b;
```

Οριακές περιπτώσεις

- **Προτεραιότητα lvalue**: το `vec EXPR, O, B = N` είναι συντακτικό σφάλμα. Γράφετε πάντα `vec(EXPR, O, B) = N`.
- **Ανάγνωση εκτός του τέλους**: η `vec($short, 1_000_000, 8)` επιστρέφει 0, ποτέ δεν πεθαίνει, ποτέ δεν προειδοποιεί.

- **Εγγραφή εκτός του τέλους**: η συμβολοσειρά συμπληρώνεται με μηδενικά έως τη θυρίδα. Για `BITS == 1`, η εγγραφή του bit 94 μεγαλώνει τη συμβολοσειρά στα 12 bytes.
- **Αρνητικό OFFSET**: μοιραίο - `"Negative offset to vec in lvalue context"` ή το ισοδύναμο rvalue.
- **Τα BITS δεν είναι υποστηριζόμενη δύναμη του δύο**: μοιραίο με `"Illegal number of bits in vec"`. Έγκυρα πλάτη είναι 1, 2, 4, 8, 16, 32, και 64 σε builds 64-bit.
- **Συμβολοσειρές κωδικοποιημένες σε UTF-8**: η `vec` θέλει συμβολοσειρά bytes. Αν το βαθμωτό φέρει τη σημαία UTF-8, η Perl προσπαθεί πρώτα να το υποβιβάσει σε αναπαράσταση ενός byte ανά χαρακτήρα. Αν κάποιος χαρακτήρας έχει codepoint 256 ή υψηλότερο, αυτό αποτυγχάνει μοιραία με `"Wide character in vec"`. Καλέστε την `utf8::downgrade` σκόπιμα, ή πακετάρετε τα δεδομένα πρώτα με `pack "C*"`, πριν προστρέξετε στη `vec`.
- **Ανάγνωση σε undef**: υπό `use warnings`, ενεργοποιεί προειδοποίηση `uninitialized` στο όρισμα συμβολοσειράς· επιστρέφει `0`.
- **Τιμή εκχώρησης πλατύτερη από τα BITS**: η τιμή μασκάρεται στα κατώτερα `BITS` bits. Το `vec($s, 0, 4) = 0x1F` αποθηκεύει `0xF`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `pack` - κατασκευάστε δυαδικές δομές πολλαπλών πεδίων· η `vec` είναι το αντίστοιχο τυχαίας πρόσβασης όταν κάθε πεδίο έχει το ίδιο πλάτος
- `unpack` - εξαγάγετε πεδία από δυαδική συμβολοσειρά· χρησιμοποιήστε `unpack("b*", $v)` ή `unpack("%32b*", $v)` για να αποδώσετε ή να μετρήσετε popcount ενός διανύσματος bits της `vec`
- `substr` - τυχαία πρόσβαση σε επίπεδο byte όταν τα `BITS` θα ήταν `8` και θέλετε επίσης η lvalue να μεγαλώσει ή να συρρικνώσει τη συμβολοσειρά
- `sprintf` - μορφοποιήστε τον ακέραιο που επιστρέφει μια ανάγνωση `vec`, π.χ. `sprintf "%08b", vec($s, $i, 8)`
- `ord` - ισοδύναμο μίας εκτέλεσης της `vec($s, $i, 8)` όταν χρειάζεστε μόνο την τιμή byte και ποτέ δεν εκχωρείτε πίσω

Διεργασίες

wait

Μπλοκάρει μέχρι να τερματιστεί οποιαδήποτε διεργασία παιδί και τη συγκομίζει.

Η `wait` είναι το απλούστερο front end για την υποκείμενη κλήση συστήματος `wait(2)`. Αναστέλλει την τρέχουσα διεργασία μέχρι να τερματιστεί ένα από τα παιδιά της, συλλέγει την κατάσταση εξόδου αυτού του παιδιού στα `$?` και `${^CHILD_ERROR_NATIVE}`, και επιστρέφει το PID του συγκομισμένου παιδιού. Αν

δεν υπάρχουν παιδιά για αναμονή, επιστρέφει αμέσως `-1`. Είναι ακριβώς ισοδύναμη με την `waitpid(-1, 0)`.

Σύνοψη

```
my $pid = wait;
```

Τι επιστρέφεται

Το PID του παιδιού που συγκομίστηκε (θετικός ακέραιος), ή `-1` αν δεν έχουν απομείνει ασυγκόμιστα παιδιά σε αναμονή. Η κατάσταση του παιδιού πηγαίνει στο `$?`· η ακατέργαστη, εξειδικευμένη για το OS λέξη κατάστασης πηγαίνει στο `${^CHILD_ERROR_NATIVE}`.

Για να αποκωδικοποιήσετε το `$?` μετά από επιτυχή `wait`:

```
my $exit   = $? >> 8;       # normal exit code (0..255)
my $signal = $? & 0x7f;     # termination signal, if any
my $core   = $? & 0x80;     # core-dump flag
```

Η επιστροφή `-1` **δεν** σημαίνει πάντα «δεν είχα ποτέ παιδιά» - μπορεί επίσης να σημαίνει «κάτι τα συγκόμισε ήδη για μένα» (δείτε παρακάτω).

Καθολική κατάσταση που επηρεάζει

- `$?` - τίθεται στην 16-bit κατάσταση αναμονής του συγκομισμένου παιδιού (υψηλό byte: κωδικός εξόδου, χαμηλό byte: σήμα τερματισμού και σημαία core). Επίσης αναγνώσιμο ως `$CHILD_ERROR` υπό `use English`.
- `${^CHILD_ERROR_NATIVE}` - η ακατέργαστη λέξη κατάστασης της πλατφόρμας, για καλούντες που θέλουν να χρησιμοποιήσουν τις POSIX μακροεντολές (`WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG`) από το `POSIX`.
- `%SIG` - αν το `$SIG{CHLD}` είναι ρυθμισμένο σε `'IGNORE'`, ή χειριστής εντός του καλεί ο ίδιος `wait`, το αποτέλεσμα αυτής της κλήσης αλλάζει. Δείτε *Οριακές περιπτώσεις*.

Παραδείγματα

Δημιουργήστε ένα worker με fork, κάντε κάτι, και μετά συγκομίστε το:

```
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;
```

Συγκομιδή κάθε εκκρεμούς παιδιού σε βρόχο. Η `wait` επιστρέφει `-1` μόλις η διεργασία δεν έχει άλλο:

```
while ((my $kid = wait) > 0) {
    warn "reaped $kid, status " . ($? >> 8) . "\n";
}
```

Διάκριση μεταξύ φυσιολογικής εξόδου και τερματισμού από σήμα:

```
my $pid = wait;
if ($pid > 0) {
    if    ($? == 0)       { say "clean exit" }
    elsif ($? & 0x7f)     { say "killed by signal " . ($? & 0x7f) }
    else                  { say "exit " . ($? >> 8) }
}
```

Αποκωδικοποίηση της εγγενούς λέξης κατάστασης με μακροεντολές POSIX, που είναι πιο φορητό από χειροκίνητη ολίσθηση bits:

```
use POSIX ":sys_wait_h";
my $pid = wait;
if ($pid > 0 && WIFEXITED(${^CHILD_ERROR_NATIVE})) {
    say "exit code: ", WEXITSTATUS(${^CHILD_ERROR_NATIVE});
}
```

Συγκομιδή χωρίς μπλοκάρισμα κάθε εκκρεμούς zombie - χρησιμοποιήστε `waitpid` με `WNOHANG`, όχι `wait`:

```
use POSIX ":sys_wait_h";
1 while waitpid(-1, WNOHANG) > 0;
```

Οριακές περιπτώσεις

- **Κανένα παιδί για αναμονή**: επιστρέφει `-1` αμέσως. Η `wait` δεν μπλοκάρει σε αυτή την περίπτωση - μπλοκάρει μόνο όταν τουλάχιστον ένα παιδί είναι ακόμη ζωντανό και ασυγκόμιστο.
- **`$SIG{CHLD} = 'IGNORE'`**: σε συστήματα POSIX αυτό ζητά από τον πυρήνα να συγκομίζει αυτόματα τα παιδιά. Υπό αυτή τη ρύθμιση, η `wait` σχεδόν πάντα επιστρέφει `-1` επειδή τα παιδιά έχουν φύγει προτού ο κώδικας χρήστη μπορέσει να την καλέσει. Το `$?` δεν τίθεται. Αν θέλετε και «fire-and-forget» παιδιά **και** αναφορά κατάστασης, χρησιμοποιήστε πραγματικό χειριστή, όχι `'IGNORE'`.
- **Προσαρμοσμένος χειριστής `$SIG{CHLD}`**: αν ο χειριστής καλεί ο ίδιος `wait` (ή `waitpid`), μια μετέπειτα `wait` στην κύρια ροή δεν θα βρει τίποτα και θα επιστρέψει `-1`. Αποφασίστε εκ των προτέρων ποια διαδρομή κώδικα κάνει τη συγκομιδή.
- **Αλληλεπίδραση με `qx//` και `system`**: και τα δύο εσωτερικά κάνουν fork, exec, και αναμονή στο δικό τους παιδί. Ένας χειριστής `$SIG{CHLD}` που καλεί τυφλά την `wait` μπορεί κατά λάθος να συγκομίσει αυτό το εσωτερικό παιδί κάτω από τη μύτη της `qx//` ή της `system`, αφήνοντάς τες ανίκανες να συλλέξουν την κατάσταση και επιστρέφοντας τυπικά `-1` με το `$!` ρυθμισμένο σε `ECHILD`. Οι χειριστές θα πρέπει να επαναλαμβάνουν με `waitpid(-1, WNOHANG)` και να ενεργούν μόνο σε PIDs που αναγνωρίζουν.

- **Το $? γράφεται μόνο σε επιτυχημένη συγκομιδή**. Μετά από επιστροφή `-1`, το `$?` διατηρεί ό,τι είχε πριν την κλήση. Μην το επιθεωρείτε εκτός αν η `wait` επέστρεψε θετικό PID.
- **Διακοπή από σήμα**: αν παραδοθεί σήμα ενώ η `wait` είναι μπλοκαρισμένη και ο χειριστής επιστρέψει φυσιολογικά, η Perl επανεκκινεί την αναμονή - δεν βλέπετε `EINTR` σε επίπεδο Perl. Αν ο χειριστής κάνει `die` ή `exit`, ο έλεγχος εγκαταλείπει τη `wait` με τον συνηθισμένο τρόπο για αυτόν τον μηχανισμό.
- **Threaded builds**: η `wait` περιμένει παιδιά της τρέχουσας διεργασίας, όχι παιδιά συγκεκριμένου νήματος. Σε πρόγραμμα πολλαπλών νημάτων, το νήμα που κάνει τη συγκομιδή είναι όποιο κάλεσε πρώτο τη `wait`.
- **Η τιμή επιστροφής είναι PID, όχι λογική τιμή**: το `if (wait) { ... }` είναι αληθές τόσο για επιτυχημένες συγκομιδές όσο και για `-1`. Συγκρίνετε πάντα με `0` ή ελέγχετε ρητά `> 0`, όπως στη μορφή βρόχου παραπάνω.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `waitpid` - αναμονή συγκεκριμένου παιδιού, ή poll χωρίς μπλοκάρισμα με `WNOHANG`· η `wait` είναι η μορφή `waitpid(-1, 0)`
- `fork` - δημιουργεί τα παιδιά που συγκομίζει η `wait`
- `exec` - αυτό που τυπικά καλεί ένα forked παιδί για να γίνει διαφορετικό πρόγραμμα
- `system` - fork + exec + wait σε μία κλήση· χρησιμοποιεί τη δική της εσωτερική `wait` και ρυθμίζει το `$?` για εσάς
- `kill` - στέλνει σήμα σε παιδί που δεν θέλετε πλέον να περιμένετε
- `$?` - εκεί όπου τοποθετείται η κατάσταση του συγκομισμένου παιδιού, κωδικοποιημένη ως υψηλό byte κωδικού εξόδου συν χαμηλό byte σήματος/core

Διεργασίες

waitpid

Περιμένει να τερματίσει συγκεκριμένη διεργασία-παιδί και τη συλλέγει.

Η `waitpid` είναι το στοχευμένο αντίστοιχο της `wait`. Ενώ η `wait` συλλέγει *οποιοδήποτε* παιδί και μπλοκάρει μέχρι ένα να εξέλθει, η `waitpid` σας επιτρέπει να ονομάσετε ποιο παιδί (ή ποια ομάδα παιδιών) να περιμένετε, και σας επιτρέπει να δημοσκοπείτε αντί να μπλοκάρετε, περνώντας `WNOHANG` στα `FLAGS`. Η κατάσταση εξόδου του παιδιού που συλλέχθηκε προσγειώνεται στην `$?`, όπως ακριβώς και με την `wait`.

Σύνοψη

```
use POSIX ":sys_wait_h";

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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
```

Τι επιστρέφεται

- Το PID του παιδιού που συλλέχθηκε σε επιτυχία.
- `0` όταν είναι ορισμένο το `WNOHANG` και υπάρχει τουλάχιστον ένα παιδί που ταιριάζει αλλά κανένα δεν έχει τερματίσει ακόμη - δηλαδή «τρέχει ακόμη, δοκιμάστε αργότερα».
- `-1` όταν καμία διεργασία-παιδί δεν ταιριάζει με το `PID` (δεν υπήρξε ποτέ, έχει ήδη συλλεχθεί, ή - σε συστήματα που κάνουν αυτόματη συλλογή - συγκομίστηκε από τον πυρήνα χωρίς τη βοήθεια της Perl).

Η κατάσταση εξόδου του παιδιού γράφεται στην `$?` και η ωμή OS-native κατάσταση στην `${^CHILD_ERROR_NATIVE}` σε κάθε επιτυχημένη συλλογή. Αποκωδικοποιήστε την `$?` με τον ίδιο τρόπο όπως μετά από `wait` ή κλήση με backticks:

```
my $reaped = waitpid $pid, 0;
if ($reaped > 0) {
    my $exit   = $? >> 8;
    my $signal = $? & 127;
    my $core   = $? & 128;
}
```

Τι σημαίνει το PID

Το `PID` επιλέγει ποια παιδιά είναι πρόθυμη να συλλέξει η `waitpid`:

- **Θετικό PID** - περιμένει για το συγκεκριμένο παιδί. Επιστροφή `-1` σημαίνει ότι δεν υπάρχει τέτοιο παιδί (το έχετε ήδη συλλέξει, ή δεν ήταν ποτέ δικό σας).
- **`0`** - περιμένει για οποιοδήποτε παιδί στην *τρέχουσα* ομάδα διεργασιών. Χρήσιμο όταν το πρόγραμμά σας έχει ενταχθεί σε ομάδα διεργασιών με άλλες συνεργαζόμενες διεργασίες και θέλετε να συγκομίζετε μόνο τα δικά σας.
- **`-1`** - περιμένει για οποιοδήποτε παιδί, ίδια εμβέλεια με την `wait`.
- **Μικρότερο από `-1`** - περιμένει για οποιοδήποτε παιδί του οποίου το ID ομάδας διεργασιών ισούται με την απόλυτη τιμή του `PID`. Η `waitpid -$pgid, 0` περιμένει για οποιοδήποτε μέλος της ομάδας διεργασιών `$pgid`.

Τα `FLAGS` είναι bitmask. Η φορητή σημαία είναι το `WNOHANG`, που κάνει την κλήση μη μπλοκαριστική: αν κανένα παιδί που ταιριάζει δεν έχει εξέλθει, η `waitpid` επιστρέφει `0` αμέσως αντί να κοιμηθεί. `FLAGS` ίσο με `0` σημαίνει «μπλόκαρε μέχρι κάτι να είναι συλλέξιμο», και υλοποιείται σε κάθε πλατφόρμα όπου τρέχει η Perl - ακόμη και σε αυτές χωρίς εγγενή κλήση συστήματος `waitpid(2)`, όπου η Perl την εξομοιώνει θυμούμενη τις καταστάσεις εξόδου παιδιών που έχει ήδη δει.

Καθολική κατάσταση που επηρεάζει

- `$?` - τίθεται στην κατάσταση εξόδου του παιδιού που συλλέχθηκε (μετατοπισμένη: το υψηλό byte = κωδικός `exit`, τα χαμηλά 7 bits = σήμα, το bit `0x80` = σημαία core-dumped).
- `${^CHILD_ERROR_NATIVE}` - τίθεται στην ωμή OS-native λέξη κατάστασης, πριν η Perl την κανονικοποιήσει σε `$?`.

Καμία από τις δύο μεταβλητές δεν αγγίζεται όταν η `waitpid` επιστρέφει `0` (`WNOHANG` και τίποτα έτοιμο) ή `-1` (κανένα παιδί που να ταιριάζει).

Παραδείγματα

Μπλοκάρισμα μέχρι γνωστό παιδί να τελειώσει, και κατόπιν επιθεώρηση του κωδικού εξόδου του:

```
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 $?;
```

Δημοσκόπηση χωρίς μπλοκάρισμα - χρήσιμη μέσα σε βρόχο γεγονότων ή χειριστή `SIGCHLD`:

```
use POSIX ":sys_wait_h";

sub harvest_children {
    while ((my $kid = waitpid -1, WNOHANG) > 0) {
        warn "child $kid exited with ", $? >> 8, "\n";
    }
}
```

Το κλασικό ιδίωμα «συλλογή κάθε εκκρεμούς zombie, χωρίς μπλοκάρισμα»:

```
use POSIX ":sys_wait_h";
1 while waitpid(-1, WNOHANG) > 0;
```

Αναμονή μόνο για παιδιά σε συγκεκριμένη ομάδα διεργασιών. Πρώτα `setpgid`, και κατόπιν στοχεύστε την ομάδα κάνοντας αρνητικό το ID της:

```
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) {
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    log_exit($kid, $?);
}
```

Διάκριση των τριών τιμών επιστροφής σε ένα σημείο:

```
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?)" }
```

Οριακές περιπτώσεις

- **Το `WNOHANG` απαιτεί `use POSIX`**. Η σταθερά δεν είναι ενσωματωμένη στην Perl· προέρχεται από το άρθρωμα `POSIX`, τυπικά μέσω `use POSIX ":sys_wait_h";`. Το να γράφετε γυμνό ακέραιο (`waitpid $pid, 1`) λειτουργεί στο Linux αλλά δεν είναι φορητό και αποκρύπτει την πρόθεση.
- **Το `$SIG{CHLD} = 'IGNORE'` μπαίνει σε race με την `waitpid`**. Όταν λέτε στον πυρήνα να συλλέγει αυτόματα τα παιδιά, δεν μένει τίποτα για να βρει η `waitpid` - επιστρέφει `-1` με την `$!` ορισμένη σε `ECHILD`. Επιλέξτε μία στρατηγική συλλογής ανά πρόγραμμα, όχι και τις δύο.
- **PIDs που έχουν ήδη συλλεχθεί** επιστρέφουν `-1`. Το PID μεμονωμένου παιδιού μπορεί να συλλεχθεί μόνο μία φορά· μετά από αυτό ξεχνιέται. Αν δώσετε το ίδιο PID στην `waitpid` δύο φορές, η δεύτερη κλήση επιστρέφει `-1`.
- **Το αρνητικό PID είναι ομάδα διεργασιών, όχι «οποιοδήποτε εκτός από αυτό»**. Η `waitpid -1234, 0` περιμένει για μέλη της ομάδας διεργασιών 1234, όχι «κάθε παιδί εκτός από το PID 1234». Αναγνώστες ανεξοικείωτοι με τη σύμβαση POSIX συχνά το παρερμηνεύουν.
- **Η `WNOHANG` με `PID = -1`** είναι η πρότυπη μορφή «αποστράγγιση της ουράς zombies». Σε συνδυασμό με βρόχο `while` συλλέγει κάθε παιδί που έχει ήδη εξέλθει και σταματά μόλις ο πυρήνας δεν έχει τίποτα άλλο να δώσει πίσω.
- **Η τιμή επιστροφής `-1` μπορεί επίσης να σημαίνει αυτόματη συλλογή**. Σε συστήματα διαμορφωμένα να συλλέγουν αυτόματα τα παιδιά (π.χ. χειριστής `$SIG{CHLD} = 'IGNORE'` εγκατεστημένος στην έναρξη), η `waitpid` δεν βλέπει κανένα παιδί και επιστρέφει `-1` ακόμη και για PID που μόλις κάνατε fork. Δείτε το `perlipc` για τις σημειώσεις πλατφόρμας.
- **Η μπλοκαριστική αναμονή είναι πάντα διαθέσιμη**. Ακόμη και σε πλατφόρμες χωρίς εγγενή κλήση συστήματος `waitpid(2)` ή `wait4(2)`, η `waitpid PID, 0` λειτουργεί - η Perl την εξομοιώνει διατηρώντας τις καταστάσεις παιδιών που έχει ήδη συλλέξει.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `wait` - μπλοκάρει για *οποιοδήποτε* παιδί· ισοδύναμη με `waitpid -1, 0` σε συστήματα που την υποστηρίζουν
- `fork` - δημιουργεί τα παιδιά που αργότερα θα συλλέξετε με `waitpid`
- `kill` - στέλνει σήμα στα ίδια PIDs και ομάδες διεργασιών που δέχεται η `waitpid`
- `$?` - κατάσταση εξόδου του παιδιού που συλλέχθηκε· αποκωδικοποιήστε ως `$? >> 8` για τον κωδικό εξόδου, `$? & 127` για το σήμα
- `${^CHILD_ERROR_NATIVE}` - ωμή OS-native λέξη κατάστασης για το ίδιο παιδί
- `POSIX` - πηγή του `WNOHANG` και του υπόλοιπου `:sys_wait_h`

Ροή ελέγχου

wantarray

Αναφέρει το περιβάλλον κλήσης της υπορουτίνας που εκτελείται τη δεδομένη στιγμή.

Η `wantarray` επιτρέπει σε μια υπορουτίνα να ρωτήσει τον runtime *πώς* έγραψε την κλήση ο καλών της: ανατέθηκε το αποτέλεσμα σε λίστα, σε βαθμωτό, ή πετάχτηκε; Η απάντηση είναι μία από τρεις διακριτές τιμές, και μια υπορουτίνα μπορεί να τη χρησιμοποιήσει για να επιστρέψει διαφορετικά σχήματα δεδομένων, ή να παρακάμψει εντελώς δαπανηρή εργασία όταν ο καλών δεν ζήτησε τίποτα.

Σύνοψη

```
wantarray
```

Δεν λαμβάνει ορίσματα και δεν έχει μορφή με παρενθέσεις - είναι δεσμευμένη λέξη, όχι κλήση συνάρτησης.

Τι επιστρέφεται

Η `wantarray` επιστρέφει μία από ακριβώς τρεις τιμές, που αντιστοιχούν στα τρία περιβάλλοντα που διακρίνει η Perl:

Τιμή επιστροφής	Περιβάλλον του καλούντος	Τυπική σύνταξη καλούντος
αληθές (1)	λίστα	`my @x = f(); (f())[0]`
ψευδές (`""`)	βαθμωτό	`my $x = f(); if (f()) {…}`
`undef`	κενό	`f();` ως εντολή

Η τριμερής διάκριση είναι ο λόγος που η συνάρτηση «θα έπρεπε να είχε ονομαστεί `wantlist()`»: η φαινομενικά λογική τιμή επιστροφής είναι στην πραγματικότητα τριών καταστάσεων, και το `defined wantarray` είναι ο σωστός έλεγχος για το «θέλησε καθόλου τίποτα ο καλών».

Ο κανονικός ιδιωματισμός

Το μοτίβο για το οποίο υπάρχει η `wantarray` είναι υπορουτίνα που προσαρμόζει το σχήμα επιστροφής της στον καλούντα:

```
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
```

Συνδυάστε το με πρόωρη έξοδο όταν ο καλών δεν ζήτησε τίποτα:

```
sub records {
    return unless defined wantarray;    # caller wrote `records();`
    my @rows = heavy_query();           # skipped in void context
    return wantarray ? @rows : "@rows";
}
```

Το `defined wantarray` είναι ο φύλακας κενού περιβάλλοντος. Η `wantarray` μόνη της δεν είναι - είναι ψευδής τόσο σε βαθμωτό όσο και σε κενό περιβάλλον, και πρόωρο `return` σε ψευδή `wantarray` θα έσπαζε κάθε καλούντα βαθμωτού περιβάλλοντος.

Το περιβάλλον διαδίδεται, δεν επανερμηνεύεται

Η `wantarray` αναφέρει το περιβάλλον που *επέβαλε* ο καλών στην υπορουτίνα, όχι τι κάνει εσωτερικά η υπορουτίνα:

```
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
```

Παρατηρήστε τα δύο τελευταία: η `print` επιβάλλει περιβάλλον λίστας στα ορίσματά της, και η `scalar` επιβάλλει ρητά βαθμωτό περιβάλλον. Κανένα δεν αφορά το τι *κάνει* η `ctx` με το αποτέλεσμα.

Κλήσεις από κενό περιβάλλον

Μια υπορουτίνα που καλείται ως γυμνή εντολή εκτελείται σε κενό περιβάλλον, και η `wantarray` επιστρέφει `undef` *μέσα σε εκείνη την υπορουτίνα*. Αυτό δεν διαδίδεται προς τα κάτω: μια υπορουτίνα που καλείται μέσα από υπορουτίνα κενού περιβάλλοντος βλέπει όποιο περιβάλλον επέβαλε ο *δικός της* καλών.

```
sub inner { defined wantarray ? "wanted" : "void" }
sub outer { my $x = inner(); return $x }

outer();            # outer is void; inner is scalar → "wanted"
```

Κάθε σημείο κλήσης καθιερώνει το δικό του περιβάλλον ανεξάρτητα. Η wantarray αναφέρει πάντα το περιβάλλον της κοντινότερης περικλείουσας υπορουτίνας, όχι της δυναμικής αλυσίδας.

Λειτουργεί επίσης μέσα σε eval

Η wantarray αναφέρει επίσης το περιβάλλον μπλοκ eval ή eval EXPR, όχι μόνο ονοματισμένων υπορουτινών:

```
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
```

Είναι ο ίδιος μηχανισμός - το eval είναι πλαίσιο κλήσης με την ίδια έννοια που είναι μια υπορουτίνα.

Πού η wantarray δεν έχει χρήσιμη απάντηση

Το αποτέλεσμα της wantarray είναι **απροσδιόριστο** σε αυτά τα μέρη, και ο κώδικας δεν πρέπει να βασίζεται σε κάποια συγκεκριμένη τιμή:

- Στο ανώτατο επίπεδο ενός αρχείου (έξω από οποιαδήποτε υπορουτίνα).
- Μέσα σε μπλοκ BEGIN, UNITCHECK, CHECK, INIT, ή END.
- Μέσα σε μέθοδο DESTROY.

Αντιμετωπίστε την wantarray ως νοηματική μόνο μέσα σε συνηθισμένη υπορουτίνα ή eval. Οπουδήποτε αλλού, σχεδιάστε τον κώδικά σας ώστε η απάντηση να μην έχει σημασία.

Παραδείγματα

Ένας getter που επιστρέφει την πλήρη λίστα σε καλούντες λίστας, το πλήθος σε καλούντες βαθμωτού, και δεν κάνει τίποτα σε κενό περιβάλλον:

```
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
```

Μια υπορουτίνα που αρνείται να κληθεί σε κενό περιβάλλον επειδή η τιμή επιστροφής είναι όλο το νόημα:

```
sub must_use {
    defined wantarray
        or croak "must_use: return value must be used";
    return compute();
}
```

Μια υπορουτίνα που εκπέμπει ένα στοιχείο ανά κλήση σε βαθμωτό περιβάλλον και ολόκληρη τη δέσμη σε περιβάλλον λίστας - σχήμα που εκθέτουν ορισμένα APIs τύπου iterator:

```
sub next_batch {
    state @queue;
    @queue = refill() unless @queue;
    return wantarray ? splice(@queue) : shift @queue;
}
```

Οριακές περιπτώσεις

- **Καμία παρένθεση, κανένα όρισμα**. Το `wantarray()` αναλύεται αλλά οι κενές παρενθέσεις είναι θόρυβος· η δεσμευμένη λέξη δεν παίρνει τίποτα. Το `wantarray $x` είναι συντακτικό σφάλμα, όχι κλήση με όρισμα.
- **Όχι το περιβάλλον του καλούντος για τελεστές**. Η `wantarray` αναφέρει μόνο το περιβάλλον πλαισίου κλήσης. Μια έκφραση όπως `$sub->() + 1` βάζει την κλήση σε βαθμωτό περιβάλλον· η `wantarray` μέσα στο `$sub` βλέπει βαθμωτό, όπως αναμένεται. Δεν υπάρχει τρόπος να ρωτήσετε «κλήθηκα μέσα σε αριθμητική έκφραση;» - μόνο λίστα / βαθμωτό / κενό.
- **Η `return` τιμά το ίδιο περιβάλλον**. Το `return @list` σε βαθμωτό περιβάλλον αποδίδει το τελευταίο στοιχείο, όχι το πλήθος - επειδή η ίδια η `return` βλέπει το περιβάλλον του καλούντος. Χρησιμοποιήστε `return wantarray ? @list : scalar @list` όταν θέλετε το πλήθος για καλούντες βαθμωτού.
- **Το λογικό περιβάλλον είναι βαθμωτό περιβάλλον**. Το `if (f())` καλεί την `f` σε βαθμωτό περιβάλλον· η `wantarray` εκεί επιστρέφει ψευδές, όχι `undef`.
- **Η ανάθεση λίστας σε κενή λίστα παραμένει περιβάλλον λίστας**. Το `() = f();` είναι περιβάλλον λίστας παρόλο που το αποτέλεσμα απορρίπτεται. Η `wantarray` επιστρέφει αληθές. Αυτό είναι χρήσιμο για επιβολή περιβάλλοντος λίστας σε υπορουτίνα της οποίας οι παρενέργειες εξαρτώνται από αυτό.
- **Η `wantarray` δεν είναι διαθέσιμη σε καλούντες XS με τον ίδιο τρόπο**. Ένα XSUB ελέγχει το περιβάλλον μέσω του `GIMME_V`· μια υπορουτίνα Perl που καλείται *από* XSUB βλέπει το περιβάλλον που το XSUB καθιέρωσε για την κλήση.
- **Τα πρωτότυπα δεν αλλάζουν την `wantarray`**. Ένα πρωτότυπο `($)` επιβάλλει το περιβάλλον ενός ορίσματος, όχι το ίδιο το περιβάλλον κλήσης της υπορουτίνας.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `return` - τιμά το ίδιο περιβάλλον λίστας/βαθμωτού/κενού που αναφέρει η `wantarray`· τα δύο σχεδιάζονται για να χρησιμοποιούνται μαζί
- `caller` - ελέγχει την ίδια τη στοίβα κλήσεων (πακέτο, αρχείο, γραμμή, και, στη μορφή τριών ορισμάτων, περισσότερες λεπτομέρειες περιβάλλοντος συμπεριλαμβανομένου του αν η `wantarray` θα επέστρεφε αληθές εκεί)
- `scalar` - επιβάλλει βαθμωτό περιβάλλον σε μια έκφραση, που είναι αυτό που κάνει την `wantarray` να επιστρέφει ψευδές στον καλούμενο
- `eval` - καθιερώνει πλαίσιο κλήσης του οποίου το περιβάλλον αναφέρει επίσης η `wantarray`

I/O

warn

Εκπέμπει μια προειδοποίηση στο `STDERR`.

Η `warn` συμβολοσειροποιεί το `LIST` και τυπώνει το αποτέλεσμα στο `STDERR` - ο ίδιος μηχανισμός μορφοποίησης με τη `die`, μείον το ξετύλιγμα. Ο έλεγχος επιστρέφει στον καλούντα· τίποτα δεν εκπέμπεται, τίποτα δεν πιάνεται. Καταφύγετε σε αυτή όταν το πρόγραμμα πρέπει να συνεχίσει να εκτελείται αλλά ο χρήστης, ο διαχειριστής, ή το log αξίζει να ακούσει κάτι αναπάντεχο.

Σύνοψη

```
warn LIST
warn $message
warn                          # re-surface current $@ as a warning
```

Τι επιστρέφεται

Η `warn` επιστρέφει 1 υπό κανονικές συνθήκες. Η τιμή επιστροφής σχεδόν ποτέ δεν είναι χρήσιμη - το νόημα της κλήσης είναι η παρενέργεια στο `STDERR` (ή σε ό,τι κάνει το `$SIG{__WARN__}` με το μήνυμα).

Σε αντίθεση με τη `die`, η `warn` **δεν** ξετυλίγει. Η εκτέλεση συνεχίζει με την εντολή μετά την κλήση.

Καθολική κατάσταση που επηρεάζει

- `$@` - διαβάζεται από τη μορφή χωρίς ορίσματα. Αν το `$@` κρατά μη κενή τιμή, η `warn` την επανεμφανίζει (δείτε *Διάδοση* παρακάτω). Η `warn` **δεν** γράφει το `$@`.
- `$,`, `$\` - **δεν** χρησιμοποιούνται. Η `warn` δεν είναι `print`· οι διαχωριστές εξόδου και ο διαχωριστής εγγραφών δεν παίζουν κανέναν ρόλο. Το μήνυμα γράφεται ως μία μόνο συμβολοσειρά.

- `$.` - ο τρέχων αριθμός γραμμής εισόδου, προσαρτάται σε μηνύματα που δεν τελειώνουν σε νέα γραμμή.
- `$SIG{__WARN__}` - αν είναι ορισμένο, ο χειριστής εκτελείται αντί της εγγραφής στο STDERR. Δείτε *To hook* `$SIG{__WARN__}` παρακάτω.
- STDERR - ο προεπιλεγμένος προορισμός. Οποιαδήποτε ανακατεύθυνση του STDERR (στο επίπεδο κελύφους με 2>, `open STDERR, ...`, ή στρώμα PerlIO) επηρεάζει το πού προσγειώνεται η έξοδος της `warn`.

Ο κανόνας τελικής νέας γραμμής

Ισχύει ο ίδιος κανόνας με τη `die`, και για τον ίδιο λόγο:

- **Το μήνυμα τελειώνει σε `"\n"`** - χρησιμοποιείται αυτούσιο. Τίποτα δεν προσαρτάται. Χρησιμοποιήστε αυτό για προειδοποιήσεις που αποτελούν πλήρη διαγνωστικά για τον χρήστη.
- **Το μήνυμα δεν τελειώνει σε `"\n"`** - η Perl προσαρτά `" at FILE line N"`, και αν διαβάζεται αρχείο, `", <HANDLE> line M"`, και έπειτα ένα τελικό `"."` και νέα γραμμή. Χρησιμοποιήστε αυτό όταν ο εντοπισμός του σημείου κλήσης είναι πιο χρήσιμος από ένα ταξινομημένο μήνυμα.

```
warn "cache is stale\n";              # "cache is stale\n"
warn "cache is stale";                # "cache is stale at main.pl␣
 ↪line 42.\n"
```

Σε προειδοποιήσεις με τιμή αναφοράς δεν προσαρτάται ποτέ πληροφορία τοποθεσίας. Η συμβολοσειροποίηση ενός αντικειμένου για την προσθήκη αρχείου και γραμμής θα ακύρωνε το νόημα της μετάδοσής του.

Διάδοση: `warn` χωρίς όρισμα (ή με κενή συμβολοσειρά)

Όταν το LIST είναι κενό ή συμβολοσειροποιείται σε `""`, η `warn` δεν εκπέμπει φρέσκο μήνυμα - επανεμφανίζει ό,τι βρίσκεται στο `$@`:

- Αν το `$@` είναι **μη κενό**, η `warn` προσαρτά `"\t...caught"` σε αυτό και εκπέμπει το αποτέλεσμα. Αυτός είναι ο κανονικός τρόπος να αντιληφθείτε ότι μια εξαίρεση πιάστηκε χωρίς να την αφήσετε να πεθάνει σιωπηρά:

  ```
  eval { risky() };
  warn if $@;                         # "<whatever died>\n\t...caught␣
   ↪at ..."
  ```

- Αν και το `$@` είναι κενό, χρησιμοποιείται η συμβολοσειρά `"Warning: Something's wrong"`.

Αυτό είναι σκόπιμα ασύμμετρο με τη `die`: η `die` χωρίς όρισμα **ξανα-εκπέμπει** (ενεργοποιώντας τον μηχανισμό διάδοσης και το hook PROPAGATE για αντικείμενα)· η `warn` χωρίς όρισμα **αναφέρει** ό,τι βρίσκεται ήδη στο `$@`.

To hook `$SIG{__WARN__}`

Ο ορισμός του `$SIG{__WARN__}` εγκαθιστά χειριστή που εκτελείται **αντί** της προεπιλεγμένης εγγραφής στο STDERR. Ο χειριστής λαμβάνει την εξαίρεση που προκύπτει από το LIST ως μοναδικό όρισμα και είναι υπεύθυνος για τη διάθεσή της - καταγραφή, επανεγγραφή, κλιμάκωση σε `die`, ή απόρριψη στο πάτωμα:

```
local $SIG{__WARN__} = sub {
    my ($msg) = @_;
    return if $msg =~ /^Use of uninitialized/;  # silently drop
    log_warning($msg);
};
```

Τρεις ασυμμετρίες με το `$SIG{__DIE__}` αξίζει να εμπεδωθούν:

- **Το προεπιλεγμένο μήνυμα καταστέλλεται.** Το `$SIG{__DIE__}` εκτελείται *παράλληλα* με τον μηχανισμό εξαιρέσεων· το `$SIG{__WARN__}` εκτελείται *αντί* της προεπιλεγμένης εγγραφής στο STDERR. Αν ο χειριστής δεν κάνει τίποτα, η προειδοποίηση απορρίπτεται σιωπηρά.
- **Για να περάσετε διαπεραστικά την προειδοποίηση, καλέστε ξανά `warn`.** Ο χειριστής δεν εισέρχεται ξανά για τη δική του κλήση `warn`, οπότε δεν υπάρχει άπειρος βρόχος:

  ```
  local $SIG{__WARN__} = sub {
      log_warning($_[0]);
      warn $_[0];                  # re-emit to STDERR as well
  };
  ```

- **Ένας χειριστής `__WARN__` μπορεί να αποσιωπήσει *υποχρεωτικές προειδοποιήσεις*** - προειδοποιήσεις που το `no warnings` δεν μπορεί να απενεργοποιήσει. Αυτό κάνει το `$SIG{__WARN__}` βαριά σφύρα· προτιμήστε `no warnings '<category>'` για στοχευμένη καταστολή και κρατήστε τον χειριστή για δρομολόγηση ή κλιμάκωση.

```
# wipe out *all* compile-time warnings, then switch on at runtime
BEGIN { $SIG{'__WARN__'} = sub { warn $_[0] if $DOWARN } }
my $foo = 10;
my $foo = 20;                       # duplicate-my silenced
$DOWARN = 1;
warn "\$foo is alive and $foo";     # now shows up
```

Η ενσωματωμένη `warn` έναντι του `use warnings`

Αυτά είναι διαφορετικά πράγματα. Μη τα μπερδεύετε:

- **Η `warn`** είναι ενσωματωμένη συνάρτηση που εκπέμπει ένα μήνυμα τώρα. Την καλείτε εσείς.
- **Το `use warnings`** είναι pragma που ενεργοποιεί κατηγορίες διαγνωστικών μηνυμάτων κατά τη μεταγλώττιση και κατά τον χρόνο εκτέλεσης (μη αρχικοποιημένες τιμές, παρωχημένα χαρακτηριστικά, μη ανοιγμένα filehandles, και ούτω καθεξής). Ο μεταγλωττιστής και ο διερμηνέας εκπέμπουν αυτά

τα διαγνωστικά εκ μέρους σας, χρησιμοποιώντας τον ίδιο μηχανισμό που χρησιμοποιεί η warn.

Τα διαγνωστικά της pragma ρέουν μέσα από το $SIG{__WARN__} ακριβώς όπως οι ρητές κλήσεις warn. Το no warnings '<category>' καταστέλλει τα διαγνωστικά της pragma στη λεξιλογική εμβέλειά του αλλά δεν έχει επίδραση σε ρητή warn που γράψατε εσείς - αυτή η κλήση πάντα πυροδοτείται.

Παραδείγματα

Αναφορά ανακτήσιμου προβλήματος και συνέχιση της εκτέλεσης:

```
open my $fh, "<", $path
    or warn "skipping $path: $!\n" and next;
```

Διαγνωστικό λειτουργίας ανάπτυξης με προσαρτημένη τοποθεσία:

```
warn "unexpected record shape";
# unexpected record shape at parser.pl line 87.
```

Παρατήρηση μιας εξαίρεσης που πιάστηκε χωρίς να κατασταλεί:

```
eval { load_config() };
warn if $@;                          # surfaces "...caught at ..."
```

Δρομολόγηση κάθε προειδοποίησης μέσω ενός logger, με εκτύπωση και στο STDERR:

```
local $SIG{__WARN__} = sub {
    my ($msg) = @_;
    $logger->warn($msg);
    print STDERR $msg;               # preserve default visibility
};
```

Κλιμάκωση συγκεκριμένης προειδοποίησης σε μοιραίο σφάλμα:

```
local $SIG{__WARN__} = sub {
    die @_ if $_[0] =~ /corrupt database/;
    warn @_;                         # everything else: default path
};
```

Μετάδοση blessed αντικειμένου για δομημένες προειδοποιήσεις (ο παραλήπτης ενός hook $SIG{__WARN__} λαμβάνει την αναφορά, όχι συμβολοσειροποιημένη μορφή):

```
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);
```

Οριακές περιπτώσεις

- **Λίστα με δύο ή περισσότερα στοιχεία** συμβολοσειροποιείται και συνενώνεται: `warn "bad record ", $n, ": ", $raw`. Ο κανόνας νέας γραμμής εφαρμόζεται στο συνενωμένο αποτέλεσμα.
- **Κενή λίστα** (`warn;` ή `warn "";`) ενεργοποιεί τη διαδρομή διάδοσης, όχι φρέσκο μήνυμα.
- **Προειδοποίηση με τιμή αναφοράς μέσα σε χειριστή `$SIG{__WARN__}`**: ο χειριστής λαμβάνει την αναφορά αμετάβλητη. Αν δεν είναι εγκατεστημένος χειριστής, η αναφορά συμβολοσειροποιείται στο STDERR μέσω της υπερφόρτωσης `""` (ή της προεπιλογής της Perl `MyClass=HASH(0x...)`).
- **`warn` μέσα σε `$SIG{__WARN__}`**: **δεν** εισέρχεται ξανά στον χειριστή. Η κλήση γράφει στο STDERR σαν να μην ήταν εγκατεστημένος χειριστής. Αυτό κάνει ασφαλή τον ιδιωματισμό «pass through».
- **`warn` κατά την καθολική καταστροφή**: εξακολουθεί να λειτουργεί, αλλά το STDERR μπορεί ήδη να είναι κλειστό σε παθολογικές περιπτώσεις. Προειδοποιήσεις από χειριστές DESTROY που εκτελούνται κατά την έξοδο του διερμηνέα μπορεί να εξαφανιστούν.
- **STDERR κλειστό ή ανακατευθυνόμενο**: η `warn` γράφει σε ό,τι δείχνει αυτή τη στιγμή το STDERR. Αν το STDERR είναι κλειστό, η εγγραφή αποτυγχάνει σιωπηρά· δεν εγείρεται σφάλμα, επειδή η ίδια η `warn` δεν ελέγχει.
- **Καμία σχέση με τα `$\` / `$,`**: η προσθήκη `local $\ = "\n"` δεν κάνει το `warn "x"` να τυπώσει νέα γραμμή. Ο κανόνας τελικής νέας γραμμής είναι ο μόνος μηχανισμός προσθήκης νέας γραμμής για την `warn`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `die` - ίδιοι κανόνες μορφοποίησης (η τελική νέα γραμμή καταστέλλει την τοποθεσία, οι αναφορές περνούν αμετάβλητες), αλλά εγείρει εξαίρεση αντί να αναφέρει και να συνεχίσει
- `eval` - συνδυάζεται με τη μορφή χωρίς ορίσματα της `warn` για τον ιδιωματισμό «πιάσε, εξέτασε, επανεμφάνισε»
- `$@` - η πηγή από την οποία αναφέρει η `warn` χωρίς όρισμα
- `$SIG{__WARN__}` - το hook που αντικαθιστά την προεπιλεγμένη εγγραφή στο STDERR· χρησιμοποιήστε το για δρομολόγηση, φιλτράρισμα, ή κλιμάκωση
- `Carp` - `carp`, `cluck`, `croak`, `confess`: προειδοποιήσεις και εξαιρέσεις που αναφέρουν από την οπτική του καλούντος και όχι του σημείου κλήσης, που είναι σχεδόν πάντα αυτό που θέλει μια βιβλιοθήκη

I/O

write

Αποδίδει μία εγγραφή σε ένα filehandle μέσω της συσχετισμένης `format` του.

Η `write` είναι το εκτελεστικό μισό του μηχανισμού παραγωγής αναφορών της Perl: παίρνει την εικόνα που δηλώνει μια `format`, τοποθετεί τις τιμές από τις αντίστοιχες γραμμές ορισμάτων στα πεδία, και εκπέμπει τις προκύπτουσες γραμμές στο `FILEHANDLE`. Παρακολουθεί επίσης τη σελίδα: όταν δεν απομένουν αρκετές γραμμές, γράφει ένα form feed, εκτελεί τη μορφή κορυφής σελίδας, και κατόπιν γράφει την εγγραφή. Η `write` είναι ο **μόνος** τρόπος με τον οποίο μια δηλωμένη format παράγει έξοδο· μια format είναι αδρανής μέχρι να την αναφέρει η `write`.

Μη μπερδεύετε αυτό με τη `syswrite` ή με το αντίστροφο της `read`: η `write` είναι αυστηρά ο εκτελεστής εγγραφών εικόνας. Δεν έχει καμία σχέση με ωμή έξοδο bytes.

Σύνοψη

```
write FILEHANDLE
write EXPR
write
```

Τι επιστρέφεται

1 σε επιτυχία, `undef` σε αποτυχία (με το `$!` ορισμένο). Μια αποτυχία εδώ είναι τυπικά είτε κλειστό handle είτε σφάλμα I/O στην υποκείμενη εγγραφή - όχι «format not found», το οποίο εκπέμπει croak αντί να επιστρέψει `undef`.

Πώς η `write` επιλέγει τη format

Κάθε filehandle έχει **δύο** θέσεις format: τη format εγγραφής και τη format κορυφής σελίδας. Εξ ορισμού:

- Η **format εγγραφής** είναι η format της οποίας το όνομα ταιριάζει με το filehandle (format `STDOUT` για το `STDOUT`, format `REPORT` για handle που ανοίχτηκε ως `REPORT`).
- Η **format κορυφής σελίδας** είναι το όνομα του filehandle με προσαρτημένο `_TOP` (`STDOUT_TOP`, `REPORT_TOP`), με εφεδρική επιλογή το `top` στο τρέχον πακέτο.

Και τα δύο είναι ανά handle υπερκαλύψιμα μέσω των `$~` (όνομα τρέχουσας format) και `$^` (όνομα τρέχουσας format κορυφής σελίδας). Αυτές οι μεταβλητές δρουν στο **τρέχον επιλεγμένο** handle, οπότε η αλλαγή τους για οτιδήποτε εκτός του `STDOUT` απαιτεί έναν χορό `select`:

```
my $ofh = select REPORT;
$~ = "My_Format";
$^ = "My_Top";
select $ofh;
```

Αν το `FILEHANDLE` παραλειφθεί, η `write` στοχεύει το τρέχον επιλεγμένο handle (προεπιλογή `STDOUT`). Αν το όρισμα είναι `EXPR` αντί για bareword, η έκφραση αξιολογείται και η τιμή της συμβολοσειράς αναζητείται ως όνομα filehandle κατά τον χρόνο εκτέλεσης.

Καθολική κατάσταση που επηρεάζει

Η `write` διαβάζει και ενημερώνει μια συστάδα μεταβλητών format ανά handle. Όλες είναι ανά filehandle, προσπελάσιμες μέσω `select`:

- `$~` - όνομα της format εγγραφής για το επιλεγμένο handle. Διαβάζεται σε κάθε κλήση `write`.
- `$^` - όνομα της format κορυφής σελίδας. Λαμβάνεται υπόψη όταν αρχίζει νέα σελίδα.
- `$=` - μήκος σελίδας σε γραμμές. Προεπιλογή `60`.
- `$-` - γραμμές που απομένουν στην τρέχουσα σελίδα. Μειώνεται με κάθε `write`. Ορίστε σε `0` για να εξαναγκάσετε αλλαγή σελίδας στην επόμενη κλήση.
- `$%` - τρέχων αριθμός σελίδας. Αυξάνεται σε κάθε top-of-page.
- `$^L` - συμβολοσειρά που εκπέμπεται πριν από κάθε top-of-page εκτός της πρώτης. Προεπιλογή `"\f"` (form feed).
- `$:` - χαρακτήρες πάνω στους οποίους μπορούν να σπάνε τα πεδία λειτουργίας γεμίσματος (^).
- `$^A` - ο συσσωρευτής format, ο ενταμιευτής στον οποίο γράφει η `formline` και τον οποίο εκκενώνει η `write`. Σπάνια αγγίζεται απευθείας εκτός αν παρακάμπτετε τη `write` με `formline`.

Η `write` επίσης γράφει μέσω της στοίβας PerlIO του handle-στόχου, οπότε το `$|` (autoflush στο επιλεγμένο handle) εξακολουθεί να ισχύει.

Επεξεργασία top-of-form

Πριν εκπέμψει την εγγραφή, η `write` ελέγχει αν η εγγραφή χωράει στις γραμμές που απομένουν στη σελίδα (`$-`). Αν δεν χωράει:

1. Εκπέμπει `$^L` (form feed εξ ορισμού) - παραλείπεται στην πρώτη σελίδα.
2. Εκτελεί τη format κορυφής σελίδας που ονοματίζει το `$^` για την παραγωγή της κεφαλίδας σελίδας.
3. Επανατοποθετεί το `$-` σε `$=` και αυξάνει το `$%`.
4. Εκπέμπει την εγγραφή.

Στην πρώτη `write` σε ένα handle, το βήμα 1 παραλείπεται ώστε η έξοδος να μην ξεκινά με form feed.

Παραδείγματα

Βασική αναφορά με κεφαλίδα:

```
format STDOUT_TOP =
                        Passwd File
 Name                Login    Office   Uid   Gid Home
 ------------------------------------------------------------------
.
format STDOUT =
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
 @<<<<<<<<<<<<<<<<<< @|||||||| @<<<<<<@>>>> @>>>> @<<<<<<<<<<<<<<<<<<
 $name,              $login,  $office,$uid,$gid, $home
.

while (my @row = getpwent) {
    ($name, $login, $uid, $gid, $office, $home) = @row[0,0,2,3,5,7];
    write;
}
```

Εγγραφή σε ονοματισμένο filehandle. Το όνομα format εξ ορισμού είναι το όνομα του handle· δεν χρειάζεται `select` αν η προεπιλογή είναι αποδεκτή:

```
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;
```

Εναλλαγή format κατά τον χρόνο εκτέλεσης ορίζοντας το `$~` στο επιλεγμένο handle. Το `local` περιορίζει την αλλαγή στην τρέχουσα εμβέλεια:

```
sub print_summary {
    local $~ = "SUMMARY_FORMAT";
    write;
}
```

Εξαναγκασμός αλλαγής σελίδας μηδενίζοντας το `$-` πριν την επόμενη κλήση:

```
$- = 0;                 # next write starts a new page
write;
```

Η `write EXPR` επιλέγει το handle κατά τον χρόνο εκτέλεσης από συμβολοσειρά - χρήσιμο όταν ο στόχος καθορίζεται από δεδομένα:

```
my $handle_name = $verbose ? "LOG" : "STDOUT";
write $handle_name;
```

Εναλλαγή `print` με `write`: νόμιμη, αλλά το `$-` παρακολουθεί μόνο την έξοδο της `write`. Αν μια ριπή `print` καταναλώσει γραμμές σελίδας που σας ενδιαφέρουν, μειώστε χειροκίνητα το `$-`:

```
print "extra note: $msg\n";
$-  -= 1;                # keep the page counter honest
write;
```

Οριακές περιπτώσεις

- **Format δεν βρέθηκε**: αν η format που ονοματίζει το `$~` δεν υπάρχει, η `write` κάνει croak με `Undefined format "NAME" called`. Αυτό είναι μοιραίο σφάλμα, όχι αποτυχία τύπου επιστροφή-undef-με-`$!`.
- **Κλειστό filehandle**: επιστρέφει `undef` και ορίζει το `$!`. Υπό το `use warnings` εκπέμπεται προειδοποίηση `write() on closed filehandle`.
- **Εγγραφή ψηλότερη από σελίδα**: αν η ίδια η εγγραφή έχει περισσότερες γραμμές από το `$=`, η `write` εξακολουθεί να εκπέμπει όλες τις γραμμές της σε μία σελίδα - δεν χωρίζει μια μεμονωμένη εγγραφή ανάμεσα σε όρια σελίδας. Το υπερμέγεθος γίνεται ανεκτό σιωπηρά.
- **Εφεδρική επιλογή `_TOP`**: αν δεν υπάρχει format `<HANDLE>_TOP`, η `write` καταφεύγει στη format με όνομα `top` στο τρέχον πακέτο. Αυτό σπάνια είναι αυτό που θέλετε για αυτο-δημιουργημένα handles· ορίστε ρητά το `$^` όταν έχει σημασία.
- **Λεξιλογικές μεταβλητές σε formats**: οι μεταβλητές `my` που δηλώνονται έξω από μια format **δεν είναι ορατές** μέσα της εκτός αν η format δηλωθεί εντός της λεξιλογικής εμβέλειας αυτών των μεταβλητών. Αυτό τσιμπάει όποιον τυλίγει τη `write` μέσα σε υπορουτίνα και απορεί γιατί η αναφορά είναι κενή - μετακινήστε τη δήλωση `format` μέσα στην υπορουτίνα, ή χρησιμοποιήστε καθολικές πακέτου για τις εκφράσεις της γραμμής ορισμάτων.
- **Η top-of-page ενεργοποιείται μεταξύ εγγραφών, όχι μέσα σε σελίδα**: η `write` ελέγχει τον χώρο στη σελίδα μία φορά ανά εγγραφή. Μια εγγραφή που ξεκινά κοντά στο κάτω μέρος της σελίδας είτε θα χωρέσει πλήρως είτε θα ενεργοποιήσει top-of-page πριν γραφεί· δεν διασταυρώνει ποτέ.
- **`LC_NUMERIC` και αριθμητικά πεδία**: αν το `use locale` ισχύει όταν δηλώνεται η format, η υποδιαστολή στα αριθμητικά πεδία ακολουθεί το locale κατά τη στιγμή δήλωσης - όχι κατά τη στιγμή της `write`. Η αλλαγή locale μετά τη μεταγλώττιση της format δεν αλλάζει τον διαχωριστή.
- **Οι χαρακτήρες ελέγχου σε πεδία κειμένου** αντικαθίστανται με κενά ώστε να μη μπορούν να ανακατέψουν τη στοίχιση στηλών. Η μόνη εξαίρεση είναι ο `\r` σε πεδία λειτουργίας γεμίσματος, όπου εξαναγκάζει αλλαγή γραμμής.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `format` - δηλώνει την εικόνα που αποδίδει η `write`· τα δύο είναι ανούσια το ένα χωρίς το άλλο
- `formline` - εκτελεί μία γραμμή εικόνας έναντι λίστας ορισμάτων μέσα στο `$^A` χωρίς παρακολούθηση σελίδας· η χαμηλού επιπέδου πρωτογενής συνάρτηση πίσω από τη `write`
- `select` - αλλάζει ποιο handle στοχεύει η `write` όταν καλείται χωρίς όρισμα, και είναι προαπαιτούμενο για τον ορισμό οποιουδήποτε από τα `$~`, `$^`, `$=`, `$-` ανά handle

- `print` - έξοδος ελεύθερης μορφής όταν η αναφορά σε στήλες είναι υπερβολή· μπορεί να εναλλάσσεται με `write` στο ίδιο handle
- `$~` - υπερκαλύπτει το όνομα της format εγγραφής για το επιλεγμένο handle
- `$-` - γραμμές που απομένουν στην τρέχουσα σελίδα· ορίστε σε `0` για να εξαναγκάσετε νέα σελίδα πριν την επόμενη `write`
- `perlform.pod` - πλήρης γραμματική γραμμών εικόνας, κανόνες λειτουργίας γεμίσματος, και επεξεργασμένα παραδείγματα αναφορών

Βαθμωτά και συμβολοσειρές

y///

Μεταγραμματίζει χαρακτήρες σε μια συμβολοσειρά. Το `y///` είναι συνώνυμο του `tr///` - τα δύο είναι πανομοιότυπα από κάθε άποψη.

Η γραφή `y` παρέχεται για τους πιστούς του **sed**, η μυϊκή μνήμη των οποίων παράγει `y/abc/ABC/` αντί για `tr/abc/ABC/`. Ο αναλυτής της Perl δέχεται οποιοδήποτε από τα δύο· η op που προκύπτει είναι η ίδια. Χρησιμοποιήστε όποια γραφή διαβάζεται καλύτερα στον περιβάλλοντα κώδικα - αν το αρχείο ήδη παραθέτει ένα script sed, το `y///` συχνά ταιριάζει· σε νέο Perl, το `tr///` είναι πιο συνηθισμένο.

Σύνοψη

```
y/SEARCHLIST/REPLACEMENTLIST/cdsr
$var =~ y/SEARCHLIST/REPLACEMENTLIST/cdsr
y/SEARCHLIST/REPLACEMENTLIST/      # transliterates $_
```

Τι επιστρέφεται

Ίδιο με το `tr///`:

- Χωρίς `/r`: ο αριθμός των χαρακτήρων που αντιστοιχίστηκαν (και αντικαταστάθηκαν ή διαγράφηκαν), αφού ο στόχος τροποποιηθεί επιτόπου.
- Με `/r`: μια νέα συμβολοσειρά με τον μεταγραμματισμό εφαρμοσμένο· το πρωτότυπο μένει ανέγγιχτο.

Παραδείγματα

Κλασικός μεταγραμματισμός κεφαλαίων στυλ sed στο `$_`:

```
while (<>) {
    y/a-z/A-Z/;
    print;
}
```

Μέτρηση φωνηέντων χωρίς τροποποίηση της συμβολοσειράς, με χρήση του `/r` για απόρριψη του μεταγραμματισμένου αντιγράφου:

```
my $vowels = ($text =~ y/aeiouAEIOU//) ;   # count in place is fine␣
↪too
my $count  = () = $text =~ /[aeiouAEIOU]/g;
```

Αφαιρέστε τα πάντα εκτός από τα ψηφία από μια συμβολοσειρά (μη καταστροφικό αντίγραφο μέσω `/r`, complement + delete):

```
my $digits = $phone =~ y/0-9//cdr;          # "(415) 555-1212" →
↪"4155551212"
```

Rot13 - το σχολικό μονογραμμικό `y///`:

```
(my $coded = $plain) =~ y/A-Za-z/N-ZA-Mn-za-m/;
```

Οριακές περιπτώσεις

Δείτε `tr///` για την πλήρη λίστα. Δύο αξίζει να επαναληφθούν εδώ:

- **Καμία παρεμβολή μεταβλητών.** Το `y/$x/$y/` μεταγραμματίζει τους κυριολεκτικούς χαρακτήρες `$`, `x` έναντι των `$`, `y` - όχι το περιεχόμενο αυτών των μεταβλητών. Αυτό ταιριάζει με το `tr///` και είναι το πιο συνηθισμένο bug μετάφρασης από sed σε Perl.
- **Το `y'...'...'` απενεργοποιεί τις περιοχές.** Με μονοί-εισαγωγικοί οριοθέτες, το `-` είναι κυριολεκτικό: το `y'a-z'A-Z'` μεταγραμματίζει μόνο τα `a`, `-`, `z` έναντι των `A`, `-`, `Z`.

Διαφορές από το upstream

Fully compatible with upstream Perl 5.44.

Δείτε επίσης

- `tr///` - η κανονική γραφή· πανομοιότυπη σημασιολογία, ίδιες σημαίες (`/c`, `/d`, `/s`, `/r`), ίδιες οριακές περιπτώσεις
- `s///` - αντικατάσταση βασισμένη σε μοτίβο όταν η αντιστοίχιση χαρακτήρα-προς-χαρακτήρα δεν αρκεί
- `lc`, `uc` - αλλαγή πεζών/κεφαλαίων με επίγνωση locale και Unicode· προτιμήστε αυτά έναντι του `y/a-z/A-Z/` για οτιδήποτε πέρα από ASCII
- `$_` - ο προεπιλεγμένος στόχος όταν δεν δίνεται δέσμευση =~

5.1.3 Τελεστές

Το ρεπερτόριο τελεστών της Perl είναι μεγάλο αλλά η δομή του είναι κανονική. Οι τελεστές ομαδοποιούνται σε μια χούφτα οικογένειες - αριθμητικοί, σύγκρισης, λογικοί, bitwise, σύνδεσης regex, ανάθεσης - και μερικούς μεμονωμένους μηχανισμούς (εύρος, τριαδικός, κόμμα, βέλος, subscripts). Αυτή η αναφορά χωρίζεται σε μία σελίδα ανά οικογένεια.

Η PetaPerl υλοποιεί το πλήρες σύνολο τελεστών Perl 5 με πανομοιότυπη σημασιολογία: ίδια προτεραιότητα, ίδια προσεταιριστικότητα, ίδιοι κανόνες περιβάλλοντος. Οι διαφορές εντοπίζονται στις *σημειώσεις ειδικές για PetaPerl* στο κάτω μέρος κάθε σελίδας όπου εφαρμόζονται.

Επιλέξτε οικογένεια

Αριθμητικοί τελεστές

Το αριθμητικό σύνολο τεσσάρων πράξεων αριθμομηχανής, επιπλέον υπόλοιπο και ύψωση σε δύναμη, επιπλέον οι μοναδιαίοι τελεστές προσήμου.

Τελεστής	Πράξη	Πληθικότητα	Προσεταιριστικότητα
`**`	ύψωση σε δύναμη	δυαδικός	δεξιά
`*`	πολλαπλασιασμός	δυαδικός	αριστερά
`/`	διαίρεση	δυαδικός	αριστερά
`%`	υπόλοιπο	δυαδικός	αριστερά
`+`	πρόσθεση	δυαδικός	αριστερά
`-`	αφαίρεση	δυαδικός	αριστερά
`+`	μοναδιαίο συν	μοναδιαίος	δεξιά
`-`	μοναδιαία άρνηση	μοναδιαίος	δεξιά

All arithmetic operators evaluate their operands in numeric context

- Όλοι οι αριθμητικοί τελεστές αποτιμούν τους τελεστέους τους σε αριθμητικό περιβάλλον - οι συμβολοσειρές εξαναγκάζονται μέσω του κανόνα «κερδίζει το αρχικό αριθμητικό πρόθεμα, τα υπόλοιπα αγνοούνται (με προειδοποίηση `Argument "..." isn't numeric` υπό `use warnings`)».

```
my $x   = 5 + 3;          # 8
my $y   = 10 / 3;         # 3.3333333333333…
my $z   = 2 ** 10;        # 1024
my $rem = 17 % 5;         # 2
my $neg = -$x;            # -8
```

Προτεραιότητα και προσεταιριστικότητα

Ο `**` δένει πιο σφιχτά και είναι *δεξιά*-προσεταιριστικός. Οι `*`, `/`, `%` σχηματίζουν μία ομάδα· οι `+`, `-` σχηματίζουν μία άλλη. Οι μοναδιαίοι `+` και `-` δένουν μεταξύ `**` και `*` (υψηλή προτεραιότητα).

```
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 /)
```

Η περίπτωση `-2 ** 2` δαγκώνει τον καθένα ακριβώς μία φορά. Βάλτε σε παρενθέσεις το πρόσημο όταν η βάση μπορεί να είναι αρνητική.

Διαίρεση

Η / επιστρέφει πάντα αποτέλεσμα κινητής υποδιαστολής, ακόμα και για ακέραιους τελεστέους που διαιρούνται ακριβώς:

```
10 / 5           # 2     -- numerically equal, stored as int or␣
↪float
                 #          depending on the runtime; treat as␣
↪numeric
10 / 4           # 2.5
10 / 3           # 3.3333333333333335
```

Για ακέραια διαίρεση, χρησιμοποιήστε `int`:

```
int(10 / 4)      # 2
int(-7 / 2)      # -3   (truncation toward zero, not floor)
```

Η διαίρεση με μηδέν είναι μοιραίο σφάλμα (όχι προειδοποίηση, όχι πράξη που παράγει άπειρο):

```
my $bad = 1 / 0;   # dies: Illegal division by zero
```

Υπόλοιπο

Ο % επιστρέφει το ακέραιο υπόλοιπο. Οι δύο τελεστέοι μετατρέπονται πρώτα σε ακεραίους· το πρόσημο του αποτελέσματος ακολουθεί τον δεξιό τελεστέο (τύπου Perl «πρόσημο του διαιρέτη», ίδιο με Python, αντίθετο της 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
```

Για υπόλοιπο κινητής υποδιαστολής, χρησιμοποιήστε `POSIX::fmod` από *POSIX*.

Ύψωση σε δύναμη

Ο ** δέχεται οποιοδήποτε ζεύγος αριθμών συμπεριλαμβανομένων μη ακέραιων εκθετών:

```
2 ** 10          # 1024
2 ** 0.5         # 1.4142135623730951    -- square root
$x ** -1         # 1/$x                   -- reciprocal
```

Για `sqrt`, η αφιερωμένη συνάρτηση *sqrt* είναι πιο σαφής (και ονομασμένος μοναδιαίος τελεστής, οπότε αναλύεται με ένα όρισμα άπληστα - δείτε *precedence*).

Μοναδιαίο + και μοναδιαίο -

Το μοναδιαίο - αρνείται. Το μοναδιαίο + είναι no-op σε αριθμούς· η μόνη πρακτική χρήση του είναι να *επιβάλλει* αριθμητικό περιβάλλον, ή να αποσαφηνίσει μια συντακτική οριακή περίπτωση όπου η Perl αλλιώς θα αντιμετώπιζε το επόμενο διακριτικό ως όνομα συνάρτησης:

```
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"
```

Σπάνια θα το χρειαστείτε. Τεκμηριώνεται επειδή εμφανίζεται σε παλαιότερο κώδικα.

Σύνθετη ανάθεση

Κάθε δυαδικός αριθμητικός τελεστής έχει σύνθετη μορφή ανάθεσης (`+=`, `-=`, `*=`, `/=`, `%=`, `**=`). Δείτε *assignment*.

Δείτε επίσης

- *String operators* - `.` και `x`, τα ισοδύναμα συμβολοσειράς των `+` και `*`.
- *Numeric comparison* - `< <= == > >= != <=>`.
- *Precedence* - πλήρης πίνακας.
- `abs`, `sqrt`, `int` - μορφές perlfunc σχετικών πράξεων.

Τελεστές συμβολοσειράς

Δύο τελεστές σε αυτή την οικογένεια - συνένωση και επανάληψη. Παίζουν τον ίδιο ρόλο για συμβολοσειρές που παίζουν οι `+` και `*` για αριθμούς.

Τελεστής	Πράξη	Πληθικότητα	Προσεταιριστικότητα	Περιβάλλον
`.`	συνένωση	δυαδικός	αριστερά	συμβολοσειρά
`x`	επανάληψη	δυαδικός	αριστερά	μικτό

Και οι δύο εξαναγκάζουν τους τελεστέους τους σε περιβάλλον συμβολοσειράς (με τη συνήθη μετατροπή αριθμού-σε-συμβολοσειρά: `42` γίνεται `"42"`, `1.5` γίνεται `"1.5"`, κλπ.).

`.` - συνένωση

Ενώνει τους δύο τελεστέους. Το αποτέλεσμα είναι η συνένωση της αριστερής και δεξιάς συμβολοσειράς:

```
"Hello" . " " . "World"         # "Hello World"
"x = " . 42                     # "x = 42"            (numeric␣
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
 ↪coercion)
"err: " . $!                    # "err: <strerror>"
```

Η σύνθετη .= προσθέτει επιτόπου:

```
my $s = "a";
$s .= "b";       # "ab"
$s .= "c";       # "abc"
```

Η .= είναι ταχύτερη από $s = $s . "..." για επαναλαμβανόμενη προσθήκη - όχι με ασυμπτωτικό περιθώριο στη σύγχρονη Perl, αλλά αρκετά για να έχει σημασία σε σφιχτούς βρόχους σε μεγάλες συμβολοσειρές. Το runtime του pperl μεταλλάσσει τον εσωτερικό ενταμιευτή επιτόπου όταν το πλήθος αναφορών της συμβολοσειράς είναι 1 (δείτε *Σημειώσεις ειδικές για PetaPerl* παρακάτω).

Για μεγάλο αριθμό τμημάτων, η join είναι γενικά πιο σαφής από μια αλυσίδα .:

```
join(", ", @items)
"hello, world, !"               # hand-built; readable for small␣
 ↪arity

join(", ", $a, $b, $c, $d)      # cleaner than $a.", ".$b.", ".$c.",
 ↪".$d
```

x - επανάληψη

Επαναλαμβάνει τον αριστερό τελεστέο έναν αριθμό φορών δεδομένο από τον δεξιό τελεστέο. Η συμπεριφορά εξαρτάται από το *περιβάλλον* του αριστερού τελεστέου:

- **Βαθμωτό περιβάλλον** στα αριστερά → επανάληψη συμβολοσειράς.
- **Περιβάλλον λίστας** στα αριστερά (παρενθέσεις γύρω από τον αριστερό τελεστέο) → επανάληψη λίστας.

```
"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)
```

Η διάκριση με παρενθέσεις-στα-αριστερά μπερδεύει τους αρχάριους. Χωρίς παρενθέσεις, η αριστερή πλευρά είναι σε βαθμωτό περιβάλλον ανεξάρτητα από πώς φαίνεται:

```
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
```

Η σύνθετη x= αναπαράγει επιτόπου:

```
my $line = "=";
$line x= 40;          # "========================================"
```

Μη θετικό πλήθος επιστρέφει κενή συμβολοσειρά (ή κενή λίστα σε περιβάλλον λίστας), χωρίς προειδοποίηση:

```
"abc" x  0            # ""
"abc" x -3            # ""
("a", "b") x 0        # ()
```

Μη ακέραιο πλήθος αποκόπτεται προς το μηδέν:

```
"x" x 3.7             # "xxx"      (3.7 truncates to 3)
"x" x -0.5            # ""
```

Αριθμητικοί έναντι συμβολοσειριακών τελεστών

Η ανάμιξη αριθμητικών και συμβολοσειριακών τελεστών στον ίδιο τελεστέο παράγει προβλέψιμα αλλά μερικές φορές εκπληκτικά αποτελέσματα, επειδή κάθε τελεστής επιβάλλει το δικό του περιβάλλον:

```
"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)
```

Δεν υπάρχει τελεστής που «μαντεύει» - ο τελεστής επιλέγει το περιβάλλον, τελεία.

Δείτε επίσης

- *Arithmetic operators* - οι + και * είναι τα αριθμητικά ανάλογα των . και x.
- *String comparison* - σύγκριση συμβολοσειρών αντί για κατασκευή τους.
- `join`, `sprintf`, `substr` - μορφές perlfunc κοινών πράξεων κατασκευής συμβολοσειρών.

Τελεστές αριθμητικής σύγκρισης

Επτά τελεστές που συγκρίνουν δύο αριθμούς και επιστρέφουν είτε λογική τιμή (οι πρώτοι έξι) είτε τριμερές αποτέλεσμα (<=>).

Τελεστής	Ερώτηση	Επιστρέφει
<	μικρότερο από	αληθές / ψευδές
<=	μικρότερο ή ίσο	αληθές / ψευδές
==	ίσο	αληθές / ψευδές
>=	μεγαλύτερο ή ίσο	αληθές / ψευδές
>	μεγαλύτερο από	αληθές / ψευδές
!=	άνισο	αληθές / ψευδές
<=>	τριμερής (sort)	-1, 0 ή +1

Και οι επτά εξαναγκάζουν τους δύο τελεστέους σε αριθμούς πριν τη σύγκριση.

```
$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
```

Η == είναι για αριθμούς, η eq είναι για συμβολοσειρές

Αυτό είναι το πιο συνηθισμένο σφάλμα που αποστέλλουν νέοι προγραμματιστές Perl. Η == συγκρίνει **αριθμητικά** - οι δύο τελεστέοι περνούν από αριθμητικό εξαναγκασμό, και κάθε μη αριθμητική συμβολοσειρά γίνεται 0:

```
"foo" == "bar"        # TRUE  -- both coerce to 0
"42x" == "42y"        # TRUE  -- both coerce to 42
"1e2" == 100          # TRUE  -- "1e2" is 100 in scientific notation
```

Για ισότητα συμβολοσειρών, χρησιμοποιήστε eq:

```
"foo" eq "bar"        # FALSE  -- the actual question
"42x" eq "42y"        # FALSE
```

Αν η `use warnings` είναι σε ισχύ, η προειδοποίηση `Argument "..." isn't numeric` ενεργοποιείται κάθε φορά που η == (ή οποιοσδήποτε άλλος αριθμητικός τελεστής) βλέπει μη αριθμητική συμβολοσειρά. Αυτή η προειδοποίηση είναι φίλος σας.

<=> - ο τελεστής spaceship

Ο <=> επιστρέφει -1 αν το αριστερό είναι μικρότερο, 0 αν ίσα, +1 αν μεγαλύτερο. Είναι *ο* συγκριτής ταξινόμησης:

```
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;
```

Το μοτίβο `cmp1 || cmp2 || cmp3` λειτουργεί επειδή ο <=> επιστρέφει 0 σε ισοπαλία και ο || περνάει παρακάτω στο 0. Δείτε *logical*.

Ο <=> και ο `cmp` είναι οι μόνοι δύο τελεστές αυτής της οικογένειας που *δεν* είναι λογικοί - είναι ακεραιότιμοι και υπάρχουν για ταξινόμηση, όχι για συνθήκες `if`.

Ισότητα κινητής υποδιαστολής

Η == σε αριθμούς κινητής υποδιαστολής είναι ακριβής ισότητα bit-προς-bit. Για τιμές παραγόμενες από αριθμητική κινητής υποδιαστολής αυτό σχεδόν ποτέ δεν είναι αυτό που θέλετε:

```
0.1 + 0.2 == 0.3      # FALSE  -- 0.30000000000000004
```

Συγκρίνετε με ανοχή:

```
abs($a - $b) < 1e-9              # absolute tolerance
abs($a - $b) < 1e-9 * abs($a)    # relative tolerance, when $a ≠ 0
```

Η `abs` είναι ένας *ονομασμένος μοναδιαίος* τελεστής.

Η αλυσίδωση δεν είναι μαγική

Η Perl **δεν** υποστηρίζει μαθηματική αλυσίδωση συγκρίσεων. Η `$a < $b < $c` δεν ελέγχει «ο $b είναι μεταξύ $a και $c» - αναλύεται ως `($a < $b) < $c`, όπου η πρώτη σύγκριση δίνει λογική τιμή (`""` ή 1) που στη συνέχεια συγκρίνεται αριθμητικά με `$c`:

```
1 < 5 < 3        # parses as (1 < 5) < 3
                 #         = 1 < 3
                 #         = TRUE   -- which is *not* what you
↪meant.
```

Γράψτε τη σύζευξη ρητά:

```
$a < $b && $b < $c     # the actual "between" test
```

Προτεραιότητα

Οι τελεστές σύγκρισης βρίσκονται στη γραμμή 11 του πίνακα *precedence* - χαλαρότεροι από τους αριθμητικούς τελεστές (οπότε η `$a + $b == $c + $d` λειτουργεί), σφιχτότεροι από τους λογικούς τελεστές (οπότε η `$a < $b && $c > $d` λειτουργεί). Είναι **μη προσεταιριστικοί**: η `$a < $b < $c` δεν είναι μη έγκυρη, αλλά η προειδοποίηση μη προσεταιριστικότητας που εξηγήθηκε παραπάνω ισχύει.

Δείτε επίσης

- *String comparison* - `eq ne lt le gt ge cmp`, η οικογένεια λεξικογραφικής σειράς.
- *Logical operators* - συνδυασμός αποτελεσμάτων σύγκρισης.
- *<=> in sort* - η κανονική περίπτωση χρήσης.
- *Λογική Boole για προγραμματιστές της Perl*

  - η επισκόπηση «δύο γεύσεων σύγκρισης» που αιτιολογεί τον αριθμητικό/συμβολοσειριακό διαχωρισμό.

- *Numbers and strings* - πότε ένα βαθμωτό εξαναγκάζεται σε αριθμό, και η ιστορία ακρίβειας/Inf/NaN πίσω από την ==.

Τελεστές σύγκρισης συμβολοσειρών

Τα ισοδύναμα σε γεύση συμβολοσειράς της οικογένειας αριθμητικής σύγκρισης. Ίδια μορφή, ίδια γραμμή προτεραιότητας, αλλά λειτουργία λεξικογραφικά (σειρά σημείων κώδικα Unicode από προεπιλογή) αντί αριθμητικά.

Τελεστής	Ερώτηση	Επιστρέφει
`lt`	μικρότερο από	αληθές / ψευδές
`le`	μικρότερο ή ίσο	αληθές / ψευδές
`eq`	ίσο	αληθές / ψευδές
`ge`	μεγαλύτερο ή ίσο	αληθές / ψευδές
`gt`	μεγαλύτερο από	αληθές / ψευδές
`ne`	άνισο	αληθές / ψευδές
`cmp`	τριμερής (sort)	`-1`, `0` ή `+1`

Οι δύο τελεστέοι εξαναγκάζονται σε συμβολοσειρές πριν τη σύγκριση.

```
$name eq "John"                 # exact string match
$kind ne "guest"                # negated match
$word lt "m"                    # alphabetically before "m"
$a    cmp  $b                   # sort comparator
```

Λεξικογραφική σειρά

Οι συμβολοσειρές συγκρίνονται χαρακτήρα-χαρακτήρα, σημείο κώδικα-σημείο κώδικα. Ο πρώτος διαφορετικός χαρακτήρας αποφασίζει· αν μια συμβολοσειρά είναι πρόθεμα της άλλης, η μικρότερη κερδίζει.

```
"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!)
```

Το τελευταίο παράδειγμα είναι ο κανονικός λόγος ύπαρξης των `eq`/`lt`/`gt`: όταν θέλετε συμβολοσειρές ψηφίων ταξινομημένες *αριθμητικά*, πρέπει να χρησιμοποιήσετε `<=>` ή να κάνετε προ-εξαναγκασμό.

Unicode

Η σειρά σημείων κώδικα **δεν** είναι ίδια με τη «αλφαβητική» σειρά που λαμβάνει υπόψη τοπικές ρυθμίσεις:

- Η `"ä" gt "z"` είναι TRUE σε σειρά σημείων κώδικα επειδή το U+00E4 βρίσκεται πέρα από το U+007A.

- Σε σειρά DIN 5007-1 («λεξικό»), η "ä" πρέπει να ταξινομείται μαζί με "a" - πολύ πριν τη "z".

Για σωστή συνταξιοθεσία τοπικών ρυθμίσεων, χρησιμοποιήστε `Unicode::Collate` από *perlfunc* ή `use locale` με κατάλληλες τοπικές ρυθμίσεις. Οι σκέτοι `lt`/`gt`/`cmp` σας δίνουν διατεταγμένη, σταθερή, ανεξάρτητη γλώσσας σύγκριση - που είναι ακριβώς το σωστό για κλειδιά `sort`, κατακερματισμό hash, ντετερμινιστική έξοδο ελέγχων κ.ο.κ. Είναι το *λάθος* πράγμα για αλφαβητικές καταχωρήσεις προοριζόμενες για ανθρώπους σε οποιαδήποτε μη αγγλική γλώσσα.

`cmp` για ταξινόμηση

Ο `cmp` είναι ο spaceship σύγκρισης συμβολοσειρών. Επιστρέφει `-1`, `0` ή `+1` και αλυσιδώνεται με τον ίδιο τρόπο που αλυσιδώνεται ο `<=>`:

```
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;
```

Ανάμιξη γεύσεων: ένα αναλυμένο σφάλμα

Το ιδίωμα ταξινόμησης σύνθετου κλειδιού από *numeric comparison* έδειξε αλυσίδωση `||` των `<=>` και `cmp`. Το σφάλμα που πρέπει να αποφύγετε είναι η χρήση του *λάθος* τελεστή για τον *τύπο* του κλειδιού:

```
# 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
```

Η σωστή απάντηση για συμβολοσειρές εκδόσεων είναι ένας αναλυτής όπως ο `Sort::Versions` ή χειροκίνητη τμηματοποίηση `(\d+)`· ούτε ο `cmp` ούτε ο `<=>` το κάνει σωστά μόνος του.

Προτεραιότητα

Η σύγκριση συμβολοσειρών μοιράζεται τη γραμμή 11 του πίνακα *precedence* με την αριθμητική σύγκριση. Είναι μη προσεταιριστικοί - η προειδοποίηση αλυσίδωσης από *numeric comparison* ισχύει και εδώ:

```
"a" lt "b" lt "c"     # parses as ("a" lt "b") lt "c"
                      #         = 1            lt "c"
                      #         = TRUE  (because "1" < "c"␣
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
↪lexically)
                        # - accidentally right for the wrong reason.
```

Γράψτε τη σύζευξη ρητά με &&.

Δείτε επίσης

- *Numeric comparison* - η παράλληλη οικογένεια.
- `sort`, `reverse`, `lc`, `uc`, `fc` - εργαλεία perlfunc που συνδυάζονται με σύγκριση συμβολοσειρών.
  - Αριθμητική έναντι συμβολοσειριακής σύγκρισης
- *Unicode in Perl* - η διάκριση τοπικών ρυθμίσεων / σειράς σημείων κώδικα σε βάθος.

Λογικοί τελεστές

Η οικογένεια λογικών τιμών - βραχυκυκλωτικό AND, OR, defined-or· τα αδέλφια τους σε μορφή λέξης χαμηλής προτεραιότητας· οι αρνήσεις· και `xor`.

Τελεστή	Διαβάζεται ως	Βραχυκύκλω	Επιστρέφει	Προτεραιότητα
`!`	όχι	δ/ε	κανονικοποιημένο 1 ή ""	υψηλή
`&&`	και	ναι	αριστερό αν ψευδές, αλλιώς δεξιό (ο τελεστέος)	υψηλή
`||`	ή	ναι	αριστερό αν αληθές, αλλιώς δεξιό (ο τελεστέος)	υψηλή
`//`	defined-or	ναι	αριστερό αν ορισμένο, αλλιώς δεξιό (τελεστέος)	υψηλή
`xor`	αποκλειστικ ή	όχι	κανονικοποιημένο 1 ή ""	πολύ χαμηλή
`not`	όχι	δ/ε	κανονικοποιημένο 1 ή ""	πολύ χαμηλή
`and`	και	ναι	ίδιο με &&	πολύ χαμηλή
`or`	ή	ναι	ίδιο με ||	πολύ χαμηλή

Η πλήρης εννοιολογική αντιμετώπιση αυτών των τελεστών - τι σημαίνει η αλήθεια, γιατί η && επιστρέφει έναν από τους τελεστέους, πώς ταιριάζουν η συνεπαγωγή και η XOR, οι μετασχηματισμοί De Morgan που σας επιτρέπουν να απλοποιήσετε μπερδεμένες συνθήκες - βρίσκεται στον οδηγό *Boolean Logic for Perl Programmers*. Αυτή η σελίδα είναι ο σύντροφος αναφοράς τελεστών.

Βραχυκύκλωμα και επιστροφή τελεστέου

Τα && και || αποτιμώνται από αριστερά προς τα δεξιά και σταματούν μόλις προσδιοριστεί το αποτέλεσμα.

- `A && B` - αν το A είναι ψευδές, επιστρέφει το A χωρίς να αποτιμήσει το B. Αλλιώς επιστρέφει το B.
- `A || B` - αν το A είναι αληθές, επιστρέφει το A χωρίς να αποτιμήσει το B. Αλλιώς επιστρέφει το B.

Η τιμή επιστροφής είναι ο **ίδιος ο τελεστέος**, όχι ένα κανονικοποιημένο αληθές/ψευδές. Αυτό είναι που κάνει τα ιδιώματα προεπιλογής της Perl να λειτουργούν:

```
my $port = $config{port} || 8080;           # operand of ||, not␣
↪boolean
my $name = $user_input  || "anonymous";
my $val  = $cache{$key} ||= compute($key);  # ||= sets if currently␣
↪false
```

Για σύγκριση, οι `!`, `not` και `xor` *κανονικοποιούν*: επιστρέφουν πάντα είτε `1` είτε την κενή συμβολοσειρά `""`.

`//` - defined-or

Ο `//` βραχυκυκλώνει βάσει **ορισμένης τιμής** αντί για αλήθεια:

- `A // B` - αν το A είναι ορισμένο, επιστρέφει το A. Αλλιώς επιστρέφει το B.

Η διαφορά έχει σημασία όταν τα `0`, `""` ή `"0"` είναι νόμιμες τιμές αντί για υποκατάστατο «λείπει»:

```
my $port    = $config{port} || 8080;    # 0 means "use default" -␣
↪wrong
my $port    = $config{port} // 8080;    # 0 means "0", undef means␣
↪"default"
my $verbose = $opt{verbose} // 0;       # 0 is a real choice
```

Η `//=` είναι η σύνθετη μορφή:

```
$config{port} //= 8080;                 # set only if currently␣
↪undef
```

`xor`

Ο τελεστής αποκλειστικού-ή. Έχει μόνο τη μορφή λέξης πολύ χαμηλής προτεραιότητας (δεν υπάρχει ^^), και σε αντίθεση με τα and/or *δεν* βραχυκυκλώνει (δεν μπορεί - πρέπει να αποτιμηθούν και οι δύο πλευρές για να καθοριστεί το αποτέλεσμα):

```
if ($admin xor $guest) { ... }          # exactly one of the two
```

Το αποτέλεσμα είναι κανονικοποιημένο - `1` ή `""`, όχι τελεστέος. Για bitwise XOR σε ακεραίους, δείτε *bitwise*.

Συμβολικές έναντι μορφών λέξης - προτεραιότητα, όχι στυλ

Οι &&, ||, ! δένουν σφιχτά (γραμμές 15–16). Οι and, or, not δένουν πολύ χαλαρά (γραμμές 22–24). Αυτό έχει σημασία στο σύνορο με την ανάθεση:

```
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.
```

Εμπειρικός κανόνας:

- Μέσα σε μια έκφραση, όπου θέλετε η τιμή να ρέει σε μια μεταβλητή ή σε άλλον τελεστή: χρησιμοποιήστε &&, ||, !, //.
- Μετά από μια πρόταση, όπου θέλετε μια παρενέργεια (die, warn, return) να ενεργοποιηθεί υπό συνθήκη: χρησιμοποιήστε or, and, not.

! και not

Και οι δύο αρνούνται. Και οι δύο επιστρέφουν το κανονικό 1 ή "":

```
my $missing = ! $config{port};          # 1 if port is unset/0/""/"0
 ↪"
return if not @items;                   # cleaner reading than `if !
 ↪@items`
```

Ο ! δένει ψηλά (γραμμή 5)· ο not δένει πολύ χαμηλά (γραμμή 22).

Σύνθετες μορφές

Τρεις από τους λογικούς τελεστές έχουν σύνθετες μορφές ανάθεσης:

Σύνθετη	Σημαίνει	Χρήση
&&=	$x = $x && $y	«στένεψε αν αληθές τώρα»
||=	$x = $x || $y	«προεπιλογή αν ψευδές τώρα»
//=	$x = $x // $y	«προεπιλογή αν undef τώρα»

```
$cache{$k} ||= compute($k);             # lazy populate
$cfg{port} //= 8080;                    # default that respects 0
$check     &&= validate($check);        # only if currently truthy
```

Παραπομπή σε οδηγό

Τα εννοιολογικά θεμέλια αυτής της σελίδας καλύπτονται στον οδηγό λογικής τιμών:

- *Truthiness* - τι θεωρεί η Perl ψευδές.
- *Operators* - ο κανόνας επιστροφής τελεστέου, η συλλογιστική προτεραιότητας και ο τριαδικός `?:` (που βρίσκεται στη *δική του σελίδα* σε αυτή την αναφορά).
- *Truth tables* - και οι δεκαέξι δυαδικές συναρτήσεις και πώς γράφεται η κάθε μία στην Perl.
- *De Morgan* - αναγωγή μπερδεμένων `unless` σε καθαρά `if`.
- *Functional completeness* - NAND/NOR.
  - NAND

Δείτε επίσης

- *Bitwise* - οι ίδιοι λογικοί τελεστές εφαρμοσμένοι σε bits ακεραίων παράλληλα.
- *Ternary* - `?:`, λογική επιλογή ως έκφραση.
- *Numeric comparison* και *string comparison* - παράγουν τις λογικές εισόδους που συνδυάζουν οι τελεστές αυτής της σελίδας.
- *Error variables* - οι `$!` και `$@` είναι οι τιμές που τυπικά βάζετε στη δεξιά πλευρά του ιδιώματος `or die` για το οποίο φτιάχτηκαν οι τελεστές σε μορφή λέξης αυτής της σελίδας.

Τελεστές κατά bit

Οι λογικοί τελεστές εφαρμοσμένοι σε κάθε ζεύγος bits παράλληλα, επιπλέον οι δύο ολισθήσεις. Ένας ακέραιος 32 ή 64 bit είναι, από τη σκοπιά αυτών των τελεστών, ένα διάνυσμα ανεξάρτητων τιμών ενός bit.

Τελεστής	Πράξη	Κανόνας ανά bit
`&`	AND κατά bit	κάθε bit εξόδου = a ∧ b
`|`	OR κατά bit	κάθε bit εξόδου = a ∨ b
`^`	XOR κατά bit	κάθε bit εξόδου = a ⊕ b
`~`	NOT κατά bit	κάθε bit εξόδου = ¬a (μοναδιαίος)
`<<`	αριστερή ολίσθηση	ολίσθηση bit προς τα αριστερά, γέμισμα με μηδενικά από τα δεξιά
`>>`	δεξιά ολίσθηση	ολίσθηση bit προς τα δεξιά

Οι τελεστέοι εξαναγκάζονται σε ακεραίους (στρογγυλοποίηση προς το μηδέν, με προειδοποιήσεις αν παρέχεται μη ακέραια συμβολοσειρά υπό `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)
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
1  << 4               # 16
32 >> 2               # 8
```

Θέση, καθάρισμα, εναλλαγή, έλεγχος μιας σημαίας

Οι τέσσερις βασικές πράξεις σημαίας σε ένα μόνο 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
```

Καθεμία είναι εφαρμογή ενός bit ενός λογικού τελεστή: ενεργοποίηση είναι `bit ∨ flag`, απενεργοποίηση είναι `bit ∧ ¬flag`, εναλλαγή είναι `bit ⊕ flag`, έλεγχος είναι `bit ∧ flag`.

Σύνθετη ανάθεση

Και οι έξι δυαδικοί τελεστές κατά bit έχουν σύνθετες μορφές:

```
$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 -∞ for signed values)
```

Ολισθήσεις

Ο << ολισθαίνει αριστερά· ο >> ολισθαίνει δεξιά. Ο αριθμός θέσεων bit προς ολίσθηση είναι ο δεξιός τελεστέος. Η ολίσθηση κατά αρνητικό ποσό, ή κατά ποσό ίσο ή μεγαλύτερο του πλάτους bit του ακεραίου, είναι ορισμένη από την υλοποίηση - μη βασίζεστε σε αυτό.

Για μη προσημασμένη αριθμητική, η αριστερή ολίσθηση είναι πολλαπλασιασμός με `2 ** $n` (modulo υπερχείλιση):

```
$x << 1               # like $x * 2
$x << 8               # like $x * 256
```

Για μη προσημασμένη αριθμητική, η δεξιά ολίσθηση είναι ακέραια διαίρεση με `2 ** $n`:

```
$x >> 1               # like int($x / 2)   (for non-negative $x)
$x >> 8               # like int($x / 256)
```

Για *προσημασμένες* τιμές, ο >> είναι ορισμένος από την υλοποίηση - η Perl δεν εγγυάται αριθμητική έναντι λογικής ολίσθησης. Εφαρμόστε μάσκα ρητά όταν έχει σημασία το πρόσημο:

```
($x >> 8) & 0xFF      # extract a byte at offset 8, regardless of␣
↪sign
```

Αριθμητική έναντι bytewise λειτουργία

Οι τελεστές κατά bit της Perl δρουν σε **ακεραίους** από προεπιλογή. Με `use feature 'bitwise'` (ή `use v5.22+`), οι μορφές με επίθημα `&.`, `|.`, `^.`, `~.` δρουν σε **συμβολοσειρές** byte-προς-byte:

```
use feature 'bitwise';
"\x80" |. "\x01"      # "\x81"  -- single-byte string OR
"abc"  ^. "abc"       # "\0\0\0" -- bytewise XOR
```

Οι χωρίς διακόσμηση τελεστές `&`, `|`, `^`, `~` κάνουν πάντα ακέραια αριθμητική, και οι τελεστέοι συμβολοσειράς εξαναγκάζονται πρώτα σε αριθμούς. Οι μορφές με επίθημα `.` είναι ο τρόπος να ζητήσετε ρητά πράξεις σε συμβολοσειρές byte.

Ανταλλαγή XOR

Η XOR έχει δύο ιδιότητες - $a \oplus a = 0$ και $a \oplus b \oplus b = a$ - που συνδυάζονται στο γνωστό τέχνασμα ανταλλαγής χωρίς προσωρινή μεταβλητή:

```
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
```

Δεν είναι ταχύτερο από `($a, $b) = ($b, $a)` στην Perl. Αξίζει να γνωρίζετε επειδή εμφανίζεται σε φολκλόρ συνεντεύξεων και ενσωματωμένο κώδικα, και επειδή είναι καθαρή απεικόνιση της ταυτότητας XOR.

Συνηθισμένα τεχνάσματα bit

Μοτίβα που θα δείτε σε κώδικα ευαίσθητο στην απόδοση:

```
$x &  ($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
```

Καθένα είναι μια άσκηση στην παρακολούθηση τι κάνει κάθε bit - καθαρή λογική συλλογιστική εφαρμοσμένη 32 (ή 64) φορές παράλληλα.

Παραπομπή σε οδηγό

Ο οδηγός λογικής τιμών έχει εκτενέστερη αντιμετώπιση του μοντέλου bitwise-ως-παράλληλη-λογική και των παραπάνω μοτίβων εφαρμογής:

- *Boolean Logic - Applications* (η ενότητα *Regular expressions: logic on sets of strings*)
  - τελεστές κατά bit ως παράλληλη λογική, χειρισμός σημαιών, ανταλλαγή XOR, κοινά τεχνάσματα bit.
- *Boolean Logic - Operators* - τα λογικά θεμέλια που κληρονομούν οι τελεστές κατά bit.
  - the boolean foundations the bitwise operators inherit.

Δείτε επίσης

- *Logical* - οι ίδιοι λογικοί τελεστές σε ολόκληρες τιμές.
- *Precedence* - οι &, |, ^ βρίσκονται μεταξύ σύγκρισης και &&/||. Σχεδόν πάντα βάζετε παρενθέσεις γύρω τους.
- `pack`, `unpack`, `vec` - εργαλεία perlfunc για εργασία σε επίπεδο bit πέρα από αυτά που προσφέρουν οι ωμοί τελεστές.

Τελεστές εύρους και flip-flop

Δύο τελεστές με δύο εντελώς διαφορετικές εργασίες ανάλογα με το περιβάλλον: σε **περιβάλλον λίστας** δημιουργούν εύρη, και σε **βαθμωτό περιβάλλον** είναι εκφράσεις λογικής τιμής με κατάσταση τύπου «flip-flop» για αντιστοίχιση εύρους γραμμών σε επεξεργασία ροής.

Τελεστής	Εργασία σε περιβάλλον λίστας	Εργασία σε βαθμωτό περιβάλλον
`..`	περιεκτικό εύρος	flip-flop (ελεγμένη-ψευδής εισαγωγή)
`...`	περιεκτικό εύρος	flip-flop (ελεγμένη-αληθής εισαγωγή)

`..` σε περιβάλλον λίστας - εύρη

Σε περιβάλλον λίστας, ο `..` παράγει την περιεκτική ακολουθία μεταξύ των ακραίων σημείων:

```
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
```

Για αριθμητικά ακραία σημεία το αποτέλεσμα είναι η ακέραια ακολουθία (με αυστηρή σειρά - η `5..1` είναι κενή λίστα, ποτέ αντεστραμμένη). Για ακραία σημεία συμβολοσειρών χρησιμοποιείται η μαγική αύξηση από `++`, που σας δίνει τη φυσική αλφαβητική εξάπλωση (και από `'aa'` μέχρι `'zz'` τους αναμενόμενους 676 δυόψηφους συνδυασμούς).

Ένα κοινό ιδίωμα Perl:

```
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
```

Το εξ ολοκλήρου ακέραιο εύρος μπορεί να είναι πολύ μεγάλο - η `1..1_000_000_000` είναι λίστα ενός δισεκατομμυρίου στοιχείων. Η Perl 5 δημιουργεί νωχελικά τα εύρη βρόχου στη `for`, οπότε η `for my $i (1..1_000_000_000)` είναι ασφαλής, αλλά η `my @x = 1..1_000_000_000` δεσμεύει ολόκληρο τον πίνακα. Η PetaPerl ακολουθεί τον ίδιο κανόνα.

Αντεστραμμένα ακραία σημεία

Η `5..1` είναι κενή. Δεν υπάρχει μορφή «αντεστραμμένου εύρους»· αν θέλετε φθίνουσα σειρά, δημιουργήστε αύξουσα και κάντε `reverse`:

```
1..5                    # (1,2,3,4,5)
5..1                    # ()        - empty, no warning
reverse 1..5            # (5,4,3,2,1)
```

Ακραία σημεία μικτού τύπου

Αν κάποιο ακραίο σημείο *μοιάζει* αριθμητικό, ο `..` κάνει αριθμητικό εύρος:

```
'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"
```

Κάντε τον τύπο ρητό για αποφυγή σύγχυσης:

```
0+'1' .. 0+'5'          # numeric, no doubt
sprintf("%d", 1) .. sprintf("%d", 5)
```

Στην πράξη, γράψτε ακέραια εύρη με σκέτα κυριολεκτικά ακεραίων και εύρη χαρακτήρων με σκέτα γράμματα σε εισαγωγικά· η διάκριση μαγικής-έναντι-αριθμητικής ακολουθεί αυτόματα.

`..` και `...` σε βαθμωτό περιβάλλον - flip-flops

Μέσα σε λογικό περιβάλλον (ένα `if`, ένα `while`, ένα `?:`, τα δεξιά ενός `&&`/`||`/`//`), ο `..` γίνεται μια έκφραση *με κατάσταση* που θυμάται αν βρίσκεται αυτή τη στιγμή μεταξύ αντιστοιχίας αριστερής πλευράς και αντιστοιχίας δεξιάς πλευράς.

```
while (<$fh>) {
    if (/^BEGIN/ .. /^END/) {     # true from BEGIN line through END␣
↪line
        print;                     # print everything in between,␣
↪inclusive
    }
}
```

Η σημασιολογία:

1. Το flip-flop είναι αρχικά **ψευδές**.
2. Όταν η **αριστερή** συνθήκη (`/^BEGIN/`) γίνει αληθής, το flip-flop μεταβαίνει σε **αληθές**, και η δεξιά πλευρά αποτιμάται αμέσως στην ίδια επανάληψη.
3. Το flip-flop παραμένει αληθές στις επόμενες επαναλήψεις μέχρι η **δεξιά** συνθήκη (`/^END/`) να γίνει αληθής. Σε εκείνη την επανάληψη επιστρέφει αληθές *και στη συνέχεια* μεταβαίνει πάλι σε ψευδές.
4. Έτσι τόσο η γραμμή BEGIN όσο και η γραμμή END συμπεριλαμβάνονται.

Όταν το flip-flop είναι αληθές, επιστρέφει έναν *αύξοντα αριθμό* (1 στην επανάληψη εισόδου, 2 στην επόμενη, ...). Στην κλείνουσα επανάληψη επιστρέφει το πλήθος ακολουθούμενο από `E0` - οπότε η τιμή είναι αληθής (μη κενή συμβολοσειρά, όχι το ψευδές `"0"`) αλλά μετατρέπεται σε συμβολοσειρά ως ο αριθμός `0` για κοσμητική σαφήνεια. Συνήθως μπορείτε να το αντιμετωπίσετε ως απλή λογική τιμή.

Ο `...` διαφέρει από τον `..` σε μία λεπτομέρεια:

- `..` (δύο τελείες) - όταν η αριστερή πλευρά γίνει αληθής, η δεξιά πλευρά ελέγχεται **στην ίδια επανάληψη**. Έτσι η ίδια γραμμή μπορεί να ανοίξει και να κλείσει το εύρος.
- `...` (τρεις τελείες) - όταν η αριστερή πλευρά γίνει αληθής, η δεξιά πλευρά **δεν** ελέγχεται μέχρι την επόμενη επανάληψη. Πάντα εύρος πολλαπλών γραμμών.

```
while (<$fh>) {
    print if /<head>/ .. /<\/head>/;     # one-line <head>...</head>␣
↪works
    print if /<head>/ ... /<\/head>/;    # forces multi-line; same-
↪line ignored
}
```

Πότε θα χρησιμοποιούσατε flip-flop

Το flip-flop λάμπει στο φιλτράρισμα εύρους γραμμών τύπου awk - εξαγωγή τμημάτων από αρχείο καταγραφής, απομόνωση περιφραγμένου μπλοκ κώδικα από markdown, επεξεργασία μπλοκ ρυθμίσεων οριοθετημένου από λέξεις-κλειδιά. Είναι γνωστά *μη*-αγαπημένο από ελεγκτές κώδικα επειδή η κατάστασή του είναι σιωπηρή.

Δύο πρακτικές εναλλακτικές που ορισμένοι προτιμούν:

```
# 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 }
}
```

Το flip-flop κερδίζει σε συντομία· η ρητή μορφή κερδίζει σε σαφήνεια με κόστος τρεις παραπάνω γραμμές. Επιλέξτε ανάλογα με το κοινό.

Δείτε επίσης

- `reverse`, `grep`, `map` - τελεστές λίστας που συνήθως συντίθενται με εύρη.
- *Subscript* - οι τομές πίνακα `@arr[1..3]` βασίζονται στο εύρος σε περιβάλλον λίστας.
- *Precedence* - οι `..` και `...` βρίσκονται στη γραμμή 17, μεταξύ `||//` και `?:`. Σχεδόν πάντα βάζετε παρενθέσεις όταν συνδυάζετε με λογική.

Τελεστές σύνδεσης regex

Δύο τελεστές που συνδέουν μια συμβολοσειρά με μια πράξη κανονικής έκφρασης. Δεν εκτελούν οι ίδιοι την αντιστοιχία· απλώς λένε «η είσοδος αυτής της έκφρασης είναι αυτή η συμβολοσειρά».

Τελεστής	Διαβάζεται ως	Χρήση
`=~`	ταιριάζει	σύνδεση πράξης regex (`m//`, `s///`, `tr///`)
`!~`	δεν ταιριάζει	ίδιο, με αρνημένο το λογικό αποτέλεσμα

```
$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
$str !~ /pattern/                    # match negated: TRUE if NO
 ↪match
```

Χωρίς ρητή σύνδεση, οι πράξεις regex δρουν στην `$_`:

```
$_ = "hello";
print "match\n" if /h/;              # implicit $_ binding
```

Οι =~ και !~ είναι ο τρόπος που ανακατευθύνετε αυτή την είσοδο σε μια διαφορετική μεταβλητή.

Τι πραγματικά επιστρέφουν

Ο =~ επιστρέφει ό,τι θα επέστρεφε η πράξη regex στα δεξιά:

- `m//` - λογική τιμή (αληθής σε αντιστοιχία, ψευδής αν δεν υπάρχει) σε βαθμωτό περιβάλλον· οι ομάδες σύλληψης σε περιβάλλον λίστας.
- `s///` - ο αριθμός αντικαταστάσεων που εκτελέστηκαν (που είναι λογικά αληθής όταν ⩾ 1).
- `tr///` - ο αριθμός χαρακτήρων που επεξεργάστηκαν.

Ο !~ επιστρέφει τη λογική άρνηση, ανεξάρτητα από την υποκείμενη πράξη. Χρησιμοποιείται κυρίως με `m//`:

```
print "no digit"   if $str !~ /\d/;
print "$n changes" if $str =~ s/foo/bar/g;
my @hits           = $str =~ /(\w+)/g;       # list context:␣
↪captures
```

Τρεις εταίροι πράξεων

Ο =~ δέχεται τρεις πράξεις regex στα δεξιά του:

- `m//` - αντιστοιχία. Το `m` είναι προαιρετικό όταν οι οριοθέτες είναι κάθετοι: η `$s =~ /pattern/` και η `$s =~ m{pattern}` λειτουργούν και οι δύο.
- `s///` - αντικατάσταση. Τρία μέρη: μοτίβο, αντικατάσταση, σημαίες. Επιστρέφει το πλήθος αντικαταστάσεων.
- `tr///` (επίσης γράφεται `y///`) - μεταγραμματισμός. Αντικαθιστά χαρακτήρες ένα-προς-ένα μεταξύ δύο συνόλων χαρακτήρων. Επιστρέφει το πλήθος χαρακτήρων που επεξεργάστηκαν.

```
$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
```

`!~` μόνο με `m//`

Ο !~ έχει νόημα μόνο με την πράξη *αντιστοιχίας*, αφού η αντικατάσταση και ο μεταγραμματισμός επιστρέφουν *πλήθη* και η περίπτωση «δεν έγιναν αλλαγές» είναι ένα νόημο μηδέν, όχι λογική τιμή «δεν ταίριαξε». Η Perl θα σας αφήσει να γράψετε `$s !~ s/.../.../`, αλλά σχεδόν ποτέ δεν το θέλετε - το αποτέλεσμα `!s///` είναι «αληθές αν μηδέν αντικαταστάσεις» που διαβάζεται περίεργα. Μείνετε στο `!~ /.../`.

Lvalue έναντι rvalue

Ο =~ δεν αναθέτει ο ίδιος. Μόνο δρομολογεί την πράξη regex στα δεξιά του στη συμβολοσειρά στα αριστερά. Η πράξη *η ίδια* μπορεί στη συνέχεια να μεταλλάξει αυτή τη συμβολοσειρά - η `s///` και η `tr///` το κάνουν, η `m//` όχι - αλλά η μετάλλαξη προέρχεται από την πράξη, όχι από τον =~:

```
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
```

Η συμβολοσειρά στα αριστερά του =~ πρέπει να είναι τροποποιήσιμη για `s///` και `tr//`. Μια κυριολεκτική συμβολοσειρά ή η $1 (μεταβλητή σύλληψης regex) θα αποτύχει με «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
```

Αντιγράψτε πρώτα αν χρειάζεται να τροποποιήσετε πηγή μόνο-ανάγνωσης:

```
(my $copy = $1) =~ s/x/y/;     # idiom for "modify a copy of $1"
```

Προτεραιότητα

Οι =~ και !~ βρίσκονται στη γραμμή 6 του πίνακα *precedence* - αρκετά σφιχτά, μεταξύ μοναδιαίων και πολλαπλασιαστικών τελεστών. Γι" αυτό μπορείτε να γράψετε `$s =~ /foo/ && $t =~ /bar/` χωρίς παρενθέσεις γύρω από κάθε αντιστοιχία.

Παραπομπή σε οδηγό

Ο οδηγός λογικής τιμών καλύπτει τα regex ως άλγεβρα συνόλων στο κεφάλαιο εφαρμογών - εναλλαγή ως ένωση, διεκδικήσεις ως τομή και συμπληρωματικό:

- *Boolean Logic - Applications* (η ενότητα *Regular expressions: logic on sets of strings*)

Η πλήρης αναφορά γλώσσας regex βρίσκεται στον οδηγό regex:

- *Regular expressions guide*

Δείτε επίσης

- `m//`, `s///`, `tr///`, `qr//`, `split` - οι μορφές perlfunc που καταναλώνουν regex.
- `$_` - η προεπιλεγμένη μεταβλητή εισόδου όταν δεν δίνεται σύνδεση.
- *Regex match variables* - `$&`, `$1..$N`, `%+`, `@+`, `@-`: τι γεμίζει κάθε επιτυχημένη αντιστοιχία.

Τελεστές ανάθεσης

Απλή ανάθεση επιπλέον οι σύνθετες μορφές - κάθε δυαδικός τελεστής που παράγει τιμή έχει επίσης μια σύνθετη μορφή ανάθεσης που εφαρμόζει την πράξη επιτόπου.

Απλό =

```
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
```

Η ανάθεση είναι **δεξιά-προσεταιριστική** - οι αλυσίδες αποτιμούνται από δεξιά προς αριστερά:

```
$a = $b = $c = 0;                    # parses as $a = ($b = ($c =
↪0))
                                     # all three become 0
```

Ολόκληρη η έκφραση ανάθεσης έχει τιμή: σε βαθμωτό περιβάλλον είναι η τιμή που ανατέθηκε· σε περιβάλλον λίστας είναι η **λίστα στοιχείων στα οποία ανατέθηκε**, σε σειρά από αριστερά προς δεξιά:

```
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
```

Ανάθεση λίστας σε βαθμωτό περιβάλλον

Πιθανώς το πιο χρησιμοποιούμενο ιδίωμα Perl για το οποίο κανείς δεν μπορεί να αναφέρει τον κανόνα:

```
my $n = () = $string =~ /\d/g;       # count digits in $string
```

Αποκωδικοποιημένο:

1. Η `$string =~ /\d/g` σε περιβάλλον λίστας δίνει όλες τις συλλήψεις ψηφίων.
2. Η `() = ...` είναι ανάθεση λίστας στην κενή λίστα (δεν αναθέτει τίποτα, αλλά αποτιμά τη δεξιά πλευρά σε περιβάλλον λίστας).
3. Η `$n = (...)` βάζει την *ανάθεση λίστας* σε βαθμωτό περιβάλλον, που αποτιμάται στον **αριθμό στοιχείων** στα δεξιά.

Αποτέλεσμα: η `$n` είναι το πλήθος αντιστοιχιών. Δεν υπάρχει `scalar(grep ...)` ή ενσωματωμένη `count` γι" αυτό· το τέχνασμα κενής λίστας είναι το κανονικό ιδίωμα.

Σύνθετη ανάθεση

Κάθε δυαδικός τελεστής που ταιριάζει στο μοτίβο `OP=` υπάρχει. Η μηχανική είναι πανομοιότυπη με `$x = $x OP $y`, *εκτός* του ότι η `$x` αποτιμάται μόνο μία φορά στα αριστερά - κάτι που έχει σημασία όταν το lvalue έχει παρενέργεια (π.χ. αυτο-δημιουργία, κλήση συνάρτησης):

```
$h->{counter}++;                        # autoviv $h->{counter}, increment
$h->{counter} += 1;                     # same effect, but spell out the OP
```

Τελεστής	Ισοδύναμο	Χρήση
`+=`	`$x = $x + $y`	συσσώρευση
`-=`	`$x = $x - $y`	αφαίρεση επιτόπου
`*=`	`$x = $x * $y`	κλιμάκωση
`/=`	`$x = $x / $y`	διαίρεση επιτόπου
`%=`	`$x = $x % $y`	υπόλοιπο
`**=`	`$x = $x ** $y`	ύψωση σε δύναμη επιτόπου
`.=`	`$x = $x . $y`	προσθήκη (συμβολοσειρά)
`x=`	`$x = $x x $y`	επανάληψη (συμβολοσειρά), ή `(@) x= n`
`&=`	`$x = $x & $y`	bitmask AND-ίσον
`|=`	`$x = $x | $y`	bitmask OR-ίσον (ενεργοποίηση bits)
`^=`	`$x = $x ^ $y`	bitmask XOR-ίσον (εναλλαγή)
`<<=`	`$x = $x << $y`	αριστερή ολίσθηση επιτόπου
`>>=`	`$x = $x >> $y`	δεξιά ολίσθηση επιτόπου
`&&=`	`$x = $x && $y`	«στένεψε αν αληθές τώρα»
`||=`	`$x = $x || $y`	προεπιλογή αν ψευδές τώρα
`//=`	`$x = $x // $y`	προεπιλογή αν undef τώρα

```
$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 - τι μπορεί να εμφανιστεί στα αριστερά

Ο αριστερός τελεστέος της = (ή οποιασδήποτε σύνθετης μορφής) πρέπει να είναι **lvalue** - μια θέση στην οποία μπορεί να γίνει εγγραφή. Τα lvalues της Perl περιλαμβάνουν:

- Μια απλή βαθμωτή μεταβλητή: `$x`, `$obj->{field}`.
- Μια μεταβλητή πίνακα ή hash: `@arr`, `%h`.
- Ένα στοιχείο πίνακα ή τομή: `$arr[3]`, `@arr[1..3]`.
- Ένα στοιχείο hash ή τομή: `$h{key}`, `@h{qw(a b c)}`.

- Ένας αποαναφερόμενος περιέκτης: `$$ref`, `@$aref`, `%{$h}`.
- Η μορφή δηλωμένη με `local`/`my` οποιουδήποτε από τα παραπάνω.
- Μια *υποσυμβολοσειρά*: `substr($s, $offset, $len) = "..."`.
- Ένας *τριαδικός* του οποίου οι κλάδοι είναι lvalues (δείτε *ternary*).
- Οι ψευδο-τελεστές `pos`, `vec`, `keys` σε tied hashes, ορισμένες θέσεις `@_` και μια χούφτα εξειδικευμένων μορφών.

Τις περισσότερες φορές γράφετε ένα lvalue χωρίς να το σκεφτείτε. Οι περιπτώσεις που σταματάτε και ελέγχετε είναι εξωτικές - `substr` ως lvalue, τριαδικός ως lvalue, τομή πίνακα ως στόχος ανάθεσης λίστας.

Ανάθεση λίστας σε πολλαπλούς στόχους

Η δεξιά πλευρά αποτιμάται *πρώτα* σε περιβάλλον λίστας, στη συνέχεια κατανέμεται στα lvalues στα αριστερά. Αυτό κάνει την ανταλλαγή εύκολη:

```
($a, $b) = ($b, $a);                    # swap
($x, $y, $z) = ($y, $z, $x);            # rotate left
```

Αν οι λίστες διαφέρουν σε μήκος:

- Περισσότερες τιμές στα δεξιά από θέσεις στα αριστερά → τα περισσευούμενα απορρίπτονται σιωπηλά.
- Περισσότερες θέσεις στα αριστερά από τιμές στα δεξιά → οι ασυμπλήρωτες θέσεις γίνονται `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
```

Ένας τελικός πίνακας ή hash στα αριστερά απορροφά τα υπόλοιπα:

```
my ($head, @tail) = @items;             # $head = first element
                                        # @tail = the rest
my ($head, %tail) = @items;             # if @items has key/value pairs
                                        # after the first
```

Μόνο **ένας** πίνακας ή hash στην αριστερή πλευρά· αλλιώς δεν θα ήταν σαφές πού τελειώνει ο ένας και πού αρχίζει ο επόμενος.

Προτεραιότητα

Οι τελεστές ανάθεσης βρίσκονται στη γραμμή 19 - αρκετά χαμηλά, χαλαρότεροι από σύγκριση και `?:`, σφιχτότεροι μόνο από τους τελεστές κόμματος και τους λογικούς σε μορφή λέξης (`and`, `or`, `not`, `xor`). Γι" αυτό η `my $fh = open ... or die` λειτουργεί (ο `or` είναι ακόμα πιο χαλαρός από το `=`).

Δείτε επίσης

- *Logical* - οι `||=`, `//=`, `&&=` είναι τα workhorses των ιδιωμάτων «προεπιλογή αν λείπει».
- *Subscript* - οι περισσότερες μορφές lvalue περιλαμβάνουν subscript.
- *Comma* - η `=` αναλύεται ξεχωριστά από τη γύρω λίστα `,` μέσω προτεραιότητας· η κατανόηση γραμμής 19 έναντι 20 είναι αυτό που σας λέει πότε η `(a, b)=(c, d)` κατανέμεται έναντι να είναι μία ενιαία λίστα.

Τριαδικός τελεστής

Ένας τελεστής, τρεις τελεστέοι. Λογική επιλογή ως *έκφραση* - το αποτέλεσμα έχει τιμή, μπορεί να ανατεθεί, να επιστραφεί, να μεταβιβαστεί ως όρισμα.

Μορφή

```
COND ? THEN : ELSE
```

Αν η `COND` είναι αληθής, η τιμή είναι `THEN`· αλλιώς είναι `ELSE`. Μόνο ο επιλεγμένος κλάδος αποτιμάται:

```
my $label = $count == 1 ? "1 item" : "$count items";
return $ok ? \%result : undef;
my $abs   = $x < 0     ? -$x       : $x;
```

Είναι έκφραση, όχι δήλωση

Ο πιο συνηθισμένος λόγος να χρησιμοποιήσετε `?:` αντί για `if`/`else` είναι ότι θέλετε μια *τιμή* - για ανάθεση, για επιστροφή, για τοποθέτηση μέσα σε μια άλλη έκφραση:

```
# verbose
my $kind;
if ($n == 1) { $kind = 'singular' }
else         { $kind = 'plural'   }

# idiomatic
my $kind = $n == 1 ? 'singular' : 'plural';
```

Αλυσίδωση (δεξιά προσεταιριστική)

Η `A ? B : C ? D : E` αναλύεται ως `A ? B : (C ? D : E)` - η αλυσίδα κατεβαίνει μέσω του ψευδούς κλάδου:

```
my $sign = $x < 0  ? '-'
         : $x > 0  ? '+'
         :           '0';
```

Η διάταξη σε στήλες κάνει την αλυσίδα αναγνώσιμη. Μετά από δύο ή τρία επίπεδα, αλλάξτε σε `if`/`elsif`/`else` - η αλυσίδα παύει να είναι πιο σαφής από την εναλλακτική.

Τριαδικός ως lvalue

Και οι δύο κλάδοι πρέπει να είναι lvalues, και ο επιλεγμένος κλάδος είναι το lvalue στο οποίο γίνεται ανάθεση:

```
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";
```

Αυτό το μοτίβο είναι σπάνιο σε σύγχρονο κώδικα - ένα ρητό `if` με δύο ευθείες αναθέσεις διαβάζεται καλύτερα. Αξίζει να το γνωρίζετε για όταν το συναντήσετε.

Τι δεν γράφετε

Ο `?:` *δεν* είναι διαχωριστής δηλώσεων. Αυτά είναι λάθος:

```
$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
```

Όταν ο στόχος είναι μια παρενέργεια (αντί για τιμή), χρησιμοποιήστε `if`/`else` ή το μεταθεματικό `if`/`unless`:

```
$x > 0 ? do_yes() : do_no();             # value-return phrasing
do_yes() if $x > 0;                       # side-effect phrasing
do_no()  unless $x > 0;
```

Με λογικούς τελεστές βραχυκυκλώματος

Ο `?:` αλληλεπιδρά με `&&`, `||`, `//` με τρόπους που μπορούν να μπερδέψουν το μάτι. Σχεδόν πάντα βάζετε παρενθέσεις:

```
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
```

Προτεραιότητα

Ο `?:` βρίσκεται στη γραμμή 18 - χαλαρότερος από τους λογικούς τελεστές (`&&`/`||`/`//`), σφιχτότερος από την ανάθεση. Γι” αυτό η `my $x = COND ? A : B` λειτουργεί χωρίς παρενθέσεις γύρω από τη δεξιά πλευρά.

Δείτε επίσης

- *Logical* - βραχυκύκλωμα && και ||, η οικογένεια εναλλακτικής χωρίς `if`.
- *Λογική Boole - Τελεστές*
  - `?:` in the context of all the boolean operators.
- *Assignment* - ο `?:` ως lvalue.

Τελεστές subscript και τομής

Η οικογένεια αγκυλών και αγκίστρων που επιλέγει στοιχεία από πίνακες και hashes - και η οικογένεια αγκίστρων αποαναφοράς που κάνει το ίδιο μέσω αναφοράς.

Μορφή	Επιλέγει	Περιέκτης	Τύπος αποτελέσματος
`$arr[i]`	ένα στοιχείο	πίνακας	scalar
`$hash{key}`	ένα στοιχείο	hash	scalar
`@arr[i, j, k]`	πολλά στοιχεία (τομή)	πίνακας	λίστα
`@hash{k1, k2}`	πολλά στοιχεία (τομή)	hash	λίστα
`%arr[i, j, k]`	τομή κλειδιού/τιμής	πίνακας	λίστα (k,v...)
`%hash{k1, k2}`	τομή κλειδιού/τιμής	hash	λίστα (k,v...)

Λίγοι κανόνες καλύπτουν και τους δώδεκα συνδυασμούς.

Το sigil σηματοδοτεί το *αποτέλεσμα*, όχι τον περιέκτη

Η `@arr` είναι ο πίνακας· η `$arr[3]` είναι ένα από τα στοιχεία του. Το αρχικό sigil σας λέει τι *είδος τιμής* προσπελαύνετε - βαθμωτό ($), λίστα τιμών (@), λίστα κλειδιού/τιμής (%):

```
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
```

Οι μορφές «%-τομή» (`%arr[...]`, `%hash{...}`) είναι μια προσθήκη της Perl 5.20 που επιστρέφει ζεύγη κλειδιού/τιμής ως λίστα. Είναι το σωστό εργαλείο όταν θέλετε αντίγραφο *μέρους* ενός hash:

```
my %selected = %h{ qw(a c) };           # %selected = (a => 1, c =>␣
↪3)
```

Αρνητικοί δείκτες σε πίνακες

Οι αρνητικοί δείκτες πίνακα μετρούν από το τέλος:

```
$arr[-1]          # last element
$arr[-2]          # second-from-last
@arr[-3..-1]      # last three elements as a slice
```

Οι θετικοί δείκτες εκτός εύρους διαβάζονται ως undef. Οι αρνητικοί δείκτες εκτός εύρους (πιο πίσω από την αρχή) εγείρουν μοιραίο σφάλμα χρόνου εκτέλεσης:

```
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)
```

Τομή ως lvalue

Τόσο οι τομές πίνακα όσο και οι τομές hash είναι έγκυροι στόχοι ανάθεσης:

```
@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!
```

Μπορείτε να ανταλλάξετε στοιχεία μέσω ανάθεσης τομής χωρίς προσωρινή μεταβλητή:

```
@arr[$i, $j] = @arr[$j, $i];              # swap two array elements
@h{$key1, $key2} = @h{$key2, $key1};      # swap two hash values
```

Αυτόματη εισαγωγή κλειδιού hash σε εισαγωγικά

Μέσα σε {} για subscripts hash, **τα αναγνωριστικά bareword γίνονται αυτόματα εισαγωγικά**:

```
$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
```

Αυτή είναι μία από τις πιο αγαπημένες (και πιο προκαλούσες σφάλματα) ευκολίες της Perl. Η απόφαση bareword-ή-όχι γίνεται κατά τη χρονική στιγμή ανάλυσης βάσει του κυριολεκτικού διακριτικού· σταθερές και υπολογιζόμενες τιμές δεν επωφελούνται:

```
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
```

Η μορφή +KEY είναι ο πιο συμπαγής τρόπος να πείτε «αυτό *δεν* είναι bareword, αποτίμησέ το ως έκφραση».

Δείκτες πίνακα: εξαναγκασμένοι σε ακεραίους, χωρίς αυτόματα εισαγωγικά

Οι δείκτες [] αποτιμούνται ως εκφράσεις και εξαναγκάζονται σε ακεραίους (αποκοπτόμενοι προς το μηδέν):

```
$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)
```

Η σύνταξη $#arr μέσα σε subscripts είναι ιδιωματική για «τελευταίο»:

```
$arr[$#arr]       # last
@arr[0..$#arr]    # all of @arr (verbose; just write @arr)
@arr[1..$#arr]    # all except first
```

Ανάμιξη - πολυδιάστατη πρόσβαση

Η Perl 5 δεν έχει εγγενείς πολυδιάστατους πίνακες· τα εμφωλευμένα δεδομένα κατασκευάζονται από αναφορές:

```
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]
```

Το βέλος μεταξύ subscripts υπονοείται από τη γραμματική της Perl: όταν δύο subscripts ακολουθούν αμέσως το ένα μετά το άλλο σε αλυσίδα αποαναφοράς, το -> μπορεί να παραλειφθεί. Δείτε *arrow*.

Δείτε επίσης

- *Arrow* - ->[] και ->{} για πρόσβαση μέσω αναφοράς· ο κανόνας παράλειψης βέλους.
- *Assignment* - ανάθεση τομής· αποδόμηση πολλαπλών στόχων.
- exists, delete - μορφές perlfunc που δέχονται έκφραση subscript.
- keys, values - μορφές ολόκληρου περιέκτη· οι τελεστές τομής είναι οι μερικές μορφές.
- *Arrays*, *Hashes* - οι σελίδες τύπων δεδομένων εξηγούν τον κανόνα sigil-έναντι-αποτελέσματος που ορίζει ποια μορφή subscript να χρησιμοποιήσετε.

Τελεστής βέλους

Ο μεμονωμένα πιο συχνά χρησιμοποιούμενος τελεστής σε λειτουργική Perl, μετά το =. Κάνει τρία πράγματα σε τρεις συντακτικές θέσεις, και όλα ξεκινούν *αποαναφέροντας* κάτι στα αριστερά.

Μορφή	Σημαίνει	Παράδειγμα
`$ref->[i]`	στοιχείο αναφοράς πίνακα	`$aref->[0]`
`$ref->{k}`	στοιχείο αναφοράς hash	`$href->{name}`
`$ref->(...)`	κλήση coderef	`$cref->(@args)`
`$obj->meth`	κλήση μεθόδου σε αντικείμενο	`$obj->print`
`$obj->meth(@a)`	κλήση μεθόδου με ορίσματα	`$obj->set_x(42)`
`$class->meth`	κλήση μεθόδου κλάσης	`Foo::Bar->new`

Οι πρώτες τρεις μορφές είναι *αποαναφορά*. Οι τελευταίες τρεις είναι *αποστολή μεθόδου*. Και οι δύο γράφονται με `->`.

Αποαναφορά

Μια αναφορά είναι ένα βαθμωτό που δείχνει σε έναν περιέκτη ή ένα coderef. Δεν μπορείτε να κάνετε δεικτοδότηση σε μια αναφορά απευθείας με `[]` ή `{}`· πρέπει πρώτα να αποαναφέρετε, και η `->` είναι ο κανονικός τρόπος:

```
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
```

Ενεργοποίηση coderef

```
my $f = sub { $_[0] * 2 };

$f->(5)             # 10  - invoke the coderef with argument 5
&$f(5)              # same - older syntax, still works
```

Η μορφή `->()` προτιμάται στη σύγχρονη Perl. Κενή λίστα ορισμάτων είναι `$f->()`.

Παράλειψη βέλους μεταξύ subscripts

Όταν δύο subscripts ακολουθούν άμεσα το ένα το άλλο σε αλυσίδα αποαναφοράς, **το δεύτερο βέλος μπορεί να παραλειφθεί**. Η γραμματική το υπονοεί:

```
$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
```

Το πρώτο βέλος (μεταξύ βαθμωτής μεταβλητής και του *πρώτου* subscript) είναι **υποχρεωτικό**. Μόνο βέλη *μεταξύ διαδοχικών subscripts* μπορούν να παραλειφθούν.

```
$aref->[0]->[1]    # explicit
$aref->[0][1]      # elided second arrow - fine
$aref[0][1]        # WRONG - $aref[0] reads from @aref, not the
                   # arrayref in $aref. Different variable!
```

Ο κανόνας πρώτου βέλους είναι η πηγή ανεπαίσθητων σφαλμάτων κατά την αναδιοργάνωση `$aref->[0]` σε βαθύτερη έκφραση - κρατήστε το αρχικό βέλος.

Κλήσεις μεθόδων

Όταν η δεξιά πλευρά του `->` είναι ένα *αναγνωριστικό* (ή μια έκφραση ονόματος σε παρενθέσεις), ερμηνεύεται ως κλήση μεθόδου αντί για 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?
```

Οι κλήσεις μεθόδων αποστέλλονται μέσω του πακέτου στο οποίο είναι blessed το αντικείμενο. Η ακριβής διαδρομή επίλυσης ελέγχεται από τη `@ISA` και τον επιλεγμένο MRO· δείτε τον *OOP guide* για την πλήρη ιστορία.

Σύνταξη έμμεσου αντικειμένου - μην τη χρησιμοποιείτε

Η Perl 5 ιστορικά επέτρεπε `print $fh "foo"` και `new Foo @args` χωρίς βέλος. Αυτές οι μορφές εξακολουθούν να αναλύονται αλλά είναι ευαίσθητες στο περιβάλλον με τρόπους που εκπλήσσουν - η σύνταξη έμμεσου αντικειμένου μπορεί να επιλύσει τη new (ή ό,τι) τη λάθος στιγμή. Η σύγχρονη κατεύθυνση είναι να **χρησιμοποιείτε πάντα `->`** για κλήσεις μεθόδων:

```
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␣
↪this
                                               # for IO::File-style␣
↪handles
```

Προτεραιότητα

Η `->` είναι η γραμμή 2 του πίνακα *precedence* - δεύτερη μόνο μετά τους όρους και τελεστές λίστας. Δένει σφιχτότερα από κάθε άλλον τελεστή, κάτι που θέλετε: η `$h->{a} + $h->{b}` αναλύεται ως άθροισμα δύο αναζητήσεων hash, όχι ως αναζήτηση `{a + $h->{b}}`.

Δείτε επίσης

- *Subscript* - απευθείας (χωρίς αποαναφορά) πρόσβαση πίνακα/hash· ο κανόνας παράλειψης βέλους.
- *References tutorial* - πλήρης αντιμετώπιση αναφορών, συμπεριλαμβανομένου πώς η `\` τις κατασκευάζει και πώς λειτουργούν οι εναλλακτικές αποαναφοράς (`@{...}`, `%{...}`, `${...}`).
- *OOP guide* - `->method` και αποστολή κλάσης σε βάθος.
- `ref`, `bless` - εργαλεία perlfunc που ζευγαρώνουν με τους δύο ρόλους του τελεστή βέλους.
- *References* - τι παράγει η `\X` και πώς συνθέτονται οι μορφές αποαναφοράς (`@{...}`, `%{...}`, `${...}`).

Τελεστές κόμματος

Δύο τελεστές, και οι δύο γραμμένοι με κόμμα - ο ένας γραμμένος ως «fat comma» με `=>`. Είναι συντακτικά ίδιοι στις περισσότερες θέσεις αλλά διαφέρουν στον κανόνα bareword της αριστερής πλευράς.

Τελεστι	Διαβάζεται ως	Τι κάνει
`,`	κόμμα	διαχωριστής λίστας (σε περιβάλλον λίστας)· απόρριψη-και-επιστροφή-δεξιού (σε βαθμωτό περιβάλλον)
`=>`	fat comma	ίδιο με `,`, επιπλέον κάνει αυτόματα εισαγωγικά σε bareword στα αριστερά

Σε περιβάλλον λίστας - κατασκευή λιστών

Σε περιβάλλον λίστας το κόμμα είναι ο διαχωριστής στοιχείων λίστας:

```
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
```

Οι λίστες στην Perl είναι *επίπεδες* - δεν υπάρχει τιμή εμφωλευμένης λίστας. Μια λίστα ενσωματωμένη σε μια άλλη εξαφανίζεται: τα στοιχεία της γίνονται απλώς στοιχεία της εξωτερικής λίστας. Για εμφωλευμένη δομή χρειάζεστε αναφορές (δείτε *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
```

Σε βαθμωτό περιβάλλον - το κόμμα είναι «απόρριψη, στη συνέχεια επιστροφή δεξιού»

Όταν μια έκφραση κόμματος *εξαναγκάζεται* σε βαθμωτό περιβάλλον - με ανάθεση σε βαθμωτό, με `scalar(...)`, ως συνθήκη ενός `if` - αποτιμά κάθε τελεστέο από αριστερά προς τα δεξιά για παρενέργειες, στη συνέχεια δίνει την τιμή του δεξιότερου τελεστέου:

```
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
```

Αυτή η χρήση είναι σπάνια στη σύγχρονη Perl - το ιδίωμα `for(init, init; cond; step, step)` τύπου C είναι η κύρια θέση που το συναντάτε (και ακόμα εκεί, οι παρενθέσεις γύρω από την υπο-έκφραση κόμματος είναι συνετές).

=> - το fat comma

Ο => κάνει το ίδιο πράγμα με το , *επιπλέον* κάνει αυτόματα εισαγωγικά στο bareword με μορφή αναγνωριστικού στα αριστερά του:

```
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', '...',
);
```

Η αυτόματη εισαγωγή σε εισαγωγικά εφαρμόζεται μόνο όταν η αριστερή πλευρά είναι **ένα συντακτικά έγκυρο αναγνωριστικό bareword** - γράμματα, ψηφία, κάτω παύλες, αρχίζοντας με γράμμα ή κάτω παύλα. Άλλες μορφές δεν γίνονται αυτόματα εισαγωγικά:

```
( 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
```

Η αλληλεπίδραση με σταθερές είναι η πιο συνηθισμένη έκπληξη - δείτε επίσης τη σελίδα *subscript* σχετικά με την αυτόματη εισαγωγή κλειδιών hash σε εισαγωγικά, που έχει το ίδιο μοτίβο.

Κοινά ιδιώματα

```
# 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-
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
↪line literals
my @list = (
    "first",
    "second",
    "third",       # trailing comma is fine
);
```

Το τελικό κόμμα είναι σκόπιμο: σας επιτρέπει να προσθέσετε, αφαιρέσετε ή αναδιατάξετε στοιχεία λίστας χωρίς να αγγίξετε γειτονικές γραμμές, κάτι που εμφανίζεται ως καθαρότερα diffs στον έλεγχο εκδόσεων.

Προτεραιότητα

Τα , και => βρίσκονται στη γραμμή 20 - χαλαρότερα από την ανάθεση, σφιχτότερα μόνο από τους λογικούς τελεστές σε μορφή λέξης (and/or/not/xor) και τους τελεστές λίστας στη δεξιά τους θέση. Γι" αυτό μια λίστα αναθέσεων λειτουργεί:

```
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
```

Δείτε επίσης

- *Assignment* - η ανάθεση λίστας βασίζεται στον ρόλο του κόμματος σε περιβάλλον λίστας.
- *Subscript* - αυτόματη εισαγωγή `$h{name}` σε εισαγωγικά, η παράλληλη περίπτωση του `name => ...`
- `use constant` - η σύγκρουση bareword/σταθεράς είναι η κανονική παγίδα του fat comma.
- *Context* - η ιστορία λίστας-έναντι-βαθμωτού πίσω από το γιατί το , σημαίνει δύο διαφορετικά πράγματα.

Προτεραιότητα

Δύο μέρη, με αυτή τη σειρά: ο πλήρης πίνακας προτεραιότητας, και ο κανόνας ανάλυσης για *ονομασμένους μοναδιαίους τελεστές* - ενσωματωμένες συναρτήσεις των οποίων η σύνταξη δένει σε προτεραιότητα τελεστή αντί ως πλήρεις κλήσεις τελεστή λίστας.

Ο πλήρης πίνακας

Σφιχτότεροι τελεστές στην κορυφή, χαλαρότεροι στο κάτω μέρος. Ίδια γραμμή = ίδια προτεραιότητα· ο συνδυασμός δύο τελεστών ίδιας γραμμής χρησιμοποιεί προσεταιριστικότητα. none = μη προσεταιριστικός (ο συνδυασμός δύο χωρίς παρενθέσεις είναι σφάλμα ή προειδοποίηση κατά τη χρονική στιγμή ανάλυσης).

Γραμμή	Προσεταιριστικότητα	Τελεστές
1	αριστερά	όροι, τελεστές λίστας (αριστερά)
2	αριστερά	`->`
3	καμία	`++ --`
4	δεξιά	`**`
5	δεξιά	`! ~ \` μοναδιαίο `+` μοναδιαίο `-`
6	αριστερά	`=~ !~`
7	αριστερά	`* / % x`
8	αριστερά	`+ - .`
9	αριστερά	`<< >>`
10	καμία	ονομασμένοι μοναδιαίοι τελεστές
11	καμία	`< <= > >= lt le gt ge`
12	καμία	`== != <=> eq ne cmp`
13	αριστερά	`&`
14	αριστερά	`| ^`
15	αριστερά	`&&`
16	αριστερά	`|| //`
17	καμία	`.. ...`
18	δεξιά	`?:`
19	δεξιά	`= += -= *= /=` κλπ.
20	αριστερά	`, =>`
21	καμία	τελεστές λίστας (δεξιά)
22	δεξιά	`not`
23	αριστερά	`and`
24	αριστερά	`or xor`

Μερικοί πρακτικοί κανόνες:

- **Σύγκριση και λογική.** Η σύγκριση (γραμμές 11–12) είναι σφιχτότερη από `&&`/`||`/`//` (γραμμές 15–16), οπότε η `$a < $b && $c > $d` λειτουργεί χωρίς εσωτερικές παρενθέσεις.
- **Bitwise έναντι σύγκρισης.** Οι `&`, `|`, `^` (γραμμές 13–14) είναι *χαλαρότεροι* από τη σύγκριση. Η `$x & 0xFF == 0` αναλύεται ως `$x & (0xFF == 0)`. Σχεδόν πάντα βάζετε παρενθέσεις γύρω από bitwise.
- **Συμβολική έναντι λογικής λέξης.** Οι `&&`/`||`/`!` είναι σφιχτοί (γραμμές 5, 15, 16). Οι `and`/`or`/`not` είναι πολύ χαλαροί (γραμμές 22–24). Ίδια λογική σημασία· διαφορετικό δέσιμο. Το κεφάλαιο *logical* δείχνει πού ταιριάζει ο καθένας.
- **Το `=` είναι χαλαρό.** Γραμμή 19 - σχεδόν τα πάντα δένουν σφιχτότερα από την ανάθεση. Γι” αυτό η `my $fh = open ... or die` λειτουργεί: ο `or` (γραμμή 24) είναι ακόμα πιο χαλαρός από το `=`, οπότε η `open` ολοκληρώνει την ανάθεση πριν αποφασίσει ο `or`.

- **Οι τελεστές λίστας βρίσκονται στα δύο άκρα του πίνακα.** Γραμμή 1 κοιτώντας *αριστερά*, γραμμή 21 κοιτώντας *δεξιά*. Ένας τελεστής λίστας όπως `print` ή `sort` απορροφά τα πάντα στα δεξιά του, μέχρι τον πρώτο τελεστή που δένει πιο χαλαρά από τη γραμμή 21 - που είναι μόνο not/and/or/xor. Η ενότητα *named unary* παρακάτω αντιπαραβάλλει αυτό με την περίπτωση ενός ορίσματος γραμμής 10.

Σε περίπτωση αμφιβολίας, γράψτε παρενθέσεις. Δεν κοστίζουν τίποτα και τεκμηριώνουν την πρόθεση.

Μετα- και προ-αύξηση / μείωση

Η γραμμή 3 (++ και --) είναι μη προσεταιριστική. Οι μορφές προ- και μετα- διαφέρουν στο *πότε* συμβαίνει η παρενέργεια:

- **++$x** - αύξηση πρώτα, στη συνέχεια απόδοση της νέας τιμής.
- **$x++** - απόδοση της τρέχουσας τιμής, στη συνέχεια αύξηση.

```
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
```

Η ++ σε συμβολοσειρά εκτελεί τη *μαγική αύξηση* της Perl:

```
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
```

Η μαγική αύξηση είναι αυτό που κάνει τη `'a'..'zz'` να παράγει όλους τους 702 δυόψηφους συνδυασμούς στο *range*.

Ονομασμένοι μοναδιαίοι τελεστές

Η γραμμή 10 περιέχει μια κατηγορία που ονομάζεται *ονομασμένοι μοναδιαίοι τελεστές*. Αυτοί δεν είναι συμβολικοί τελεστές - είναι **ενσωματωμένες συναρτήσεις** που δέχονται ακριβώς ένα όρισμα και αναλύονται με προτεραιότητα τύπου τελεστή (μεταξύ ολισθήσεων bit και σύγκρισης) αντί ως πλήρεις τελεστές λίστας.

Η πρακτική συνέπεια είναι ένας κανόνας ανάλυσης:

> Ένας ονομασμένος μοναδιαίος τελεστής αρπάζει το **ένα** του όρισμα άπληστα, και οτιδήποτε θα χρειαζόταν περισσότερα από ένα ορίσματα δεν ανήκει σε αυτόν.

```
defined $x + 1             # parses as: ( defined($x) ) + 1
                           # - NOT  defined($x + 1)
length $s > 3              # parses as: ( length($s) ) > 3
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
-e $f && -r _               # both file tests are named-unary;
                            # `&&` combines two boolean results
```

Συγκρίνετε με έναν τελεστή λίστας (`print`, `sort`, `push`, ...), που βρίσκεται στη γραμμή 1 / 21 και απορροφά τα υπόλοιπα της έκφρασης:

```
print "n=", $n, "\n"       # print's argument list is "n=", $n, "\n"
                            # all three commas belong to print
```

Αν θέλετε μια ενσωματωμένη ονομασμένου μοναδιαίου τελεστή να δεχτεί πιο σύνθετο όρισμα, βάλτε παρενθέσεις:

```
defined($x + 1)             # explicit: argument is the sum $x+1
length($s . "x")            # length of $s with "x" appended
```

Η αδιαμφισβήτητη μορφή είναι πάντα διαθέσιμη. Σε περίπτωση αμφιβολίας, βάλτε παρενθέσεις.

Η λίστα ενσωματωμένων ονομασμένων μοναδιαίων τελεστών

Αυτές οι ενσωματωμένες αναλύονται σε προτεραιότητα γραμμής 10. Κάθε μία συνδέεται στη σελίδα λεπτομερειών `perlfunc` - ο κανόνας ανάλυσης παραπάνω είναι το μόνο που χρειάζεται να γνωρίζετε γι" αυτές ως *τελεστές*· η συμπεριφορά *συνάρτησης* τους βρίσκεται στην αναφορά perlfunc.

`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` (όταν δίνεται ακριβώς ένα όρισμα), καθώς και όλοι οι **τελεστές ελέγχου αρχείων** (`-e`, `-r`, `-w`, `-x`, `-f`, `-d`, `-l`, `-s` και οι υπόλοιποι - δείτε την ενότητα *τελεστές ελέγχου αρχείων* παρακάτω).

Η πλαισίωση ως «τελεστής» στην κλασική τεκμηρίωση Perl είναι ιστορική. Σε σύγχρονους όρους Perl, αυτές είναι *συναρτήσεις των οποίων η ανάλυση ορισμάτων τις τοποθετεί μεταξύ ολίσθησης bit και σύγκρισης στον πίνακα προτεραιότητας*. Η ονομασία είναι ό,τι είναι· η συμπεριφορά είναι αυτό που μετράει.

Τελεστές ελέγχου αρχείων

Οι έλεγχοι αρχείων είναι μοναδιαίοι τελεστές (προτεραιότητα ονομασμένου μοναδιαίου) που εξετάζουν μια ιδιότητα αρχείου ή filehandle. Αρχίζουν με παύλα και ένα γράμμα. Όλοι επιστρέφουν λογική τιμή εκτός από τον `-s` (μέγεθος αρχείου σε bytes), τους `-M`/`-A`/`-C` (ηλικία σε ημέρες, κλασματικά) και τους ελέγχους ανάγνωσης `-T`/`-B` (λογικοί αλλά ελαφρώς πιο αργοί επειδή διαβάζουν το αρχείο).

Τελ.	Έλεγχος	Επιστρέφει
`-e`	exists	λογική τιμή
`-r`	αναγνώσιμο από effective UID	λογική τιμή
`-w`	εγγράψιμο από effective UID	λογική τιμή
`-x`	εκτελέσιμο από effective UID	λογική τιμή
`-o`	ιδιοκτησία effective UID	λογική τιμή
`-R`	αναγνώσιμο από real UID	λογική τιμή
`-W`	εγγράψιμο από real UID	λογική τιμή
`-X`	εκτελέσιμο από real UID	λογική τιμή
`-O`	ιδιοκτησία real UID	λογική τιμή
`-z`	μηδενικό μέγεθος	λογική τιμή
`-s`	μη μηδενικό μέγεθος	μέγεθος σε bytes ή ψευδές
`-f`	κανονικό αρχείο	λογική τιμή
`-d`	κατάλογος	λογική τιμή
`-l`	symbolic link	λογική τιμή
`-p`	ονομαστικός σωλήνας (FIFO)	λογική τιμή
`-S`	socket	λογική τιμή
`-b`	ειδικό αρχείο block	λογική τιμή
`-c`	ειδικό αρχείο χαρακτήρων	λογική τιμή
`-t`	TTY (τερματικό)	λογική τιμή
`-u`	bit setuid	λογική τιμή
`-g`	bit setgid	λογική τιμή
`-k`	sticky bit	λογική τιμή
`-T`	αρχείο κειμένου (ευρετικά)	λογική τιμή
`-B`	δυαδικό αρχείο (ευρετικά)	λογική τιμή
`-M`	χρόνος τροποποίησης (ηλικία από εκκίνηση σεναρίου, ημέρες)	αριθμός
`-A`	χρόνος πρόσβασης (ημέρες)	αριθμός
`-C`	χρόνος αλλαγής inode (ημέρες)	αριθμός

```
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
```

Στοιβαγμένοι έλεγχοι αρχείων. Η `-f -w -r $file` είναι συντομογραφία για `(-f $file) && (-w $file) && (-r $file)` - η αλυσίδα επαναχρησιμοποιεί τον ίδιο στόχο. Το ψευδο-handle `_` (μεμονωμένη κάτω παύλα) επαναχρησιμοποιεί τη cache `stat` του *προηγούμενου* ελέγχου αρχείου για αποφυγή επιπλέον κλήσεων συστήματος `stat()`:

```
if (-f $file && -r _) { ... }        # one stat() call, two checks
```

Η τελική `_` είναι το τυπικό ιδίωμα για οποιαδήποτε αλυσίδα ελέγχων αρχείων στην ίδια διαδρομή.

Δείτε επίσης

- Κάθε άλλη σελίδα σε αυτή την αναφορά συνδέεται στις γραμμές αυτού του πίνακα από τη δική της ενότητα *Προτεραιότητα*. Η διασταυρούμενη αναφορά είναι σκόπιμη - όταν χρειαστεί να συνδυάσετε έναν τελεστή με έναν άλλον, αυτός ο πίνακας είναι η απάντηση.
- *perlfunc index* - κάθε όνομα στη λίστα ονομασμένων μοναδιαίων τελεστών τεκμηριώνεται εκεί.
- *Arithmetic* - `+ - * / % **` και ο μοναδιαίος +/-.
- *String* - `.` (συνένωση), `x` (επανάληψη).
- *Numeric comparison* - `< <= == >= > != <=>`.
- *String comparison* - `lt le eq ge gt ne cmp` (λεξικογραφική).
- *Logical* - `&& || // ! and or not xor` και ο κανόνας βραχυκυκλώματος / επιστροφής τελεστέου.
- *Bitwise* - `& | ^ ~ << >>`.
- *Range* - `..` και `...`, με τη διπλή σημασιολογία λίστας/βαθμωτού.
- *Binding* - `=~` και `!~` συνδέουν συμβολοσειρές με regex.
- *Assignment* - `=` και οι σύνθετες μορφές (`+= .= ||= //=` κλπ.).
- *Ternary* - `?:`.
- *Subscript* - `[]`, `{}`, τομές, αγκύλες αποαναφοράς.
- *Arrow* - `->` (αποαναφορά, κλήση μεθόδου).
- *Comma* - `,` και το fat comma `=>`.
- *Precedence* - ο πλήρης πίνακας προτεραιότητας και ο κανόνας ανάλυσης ονομασμένου μοναδιαίου τελεστή που μπερδεύει όλους την πρώτη φορά.

Διατομεακές έννοιες

Μερικές ιδέες εμφανίζονται σε κάθε κεφάλαιο. Διαβάστε τις μία φορά και θα τις αναγνωρίζετε σε όλα τα υπόλοιπα:

- **Περιβάλλον.** Οι τελεστές αποτιμούν τους τελεστέους τους σε συγκεκριμένο περιβάλλον (αριθμητικό, συμβολοσειρά, λίστα, βαθμωτό, λογικό). Πολλές από τις εκπλήξεις στην αριθμητική και τη σύγκριση της Perl προέρχονται από τον εξαναγκασμό περιβάλλοντος στα όρια τελεστέου, όχι από τον ίδιο τον τελεστή.
- **Βραχυκύκλωμα.** Οι `&&`, `||`, `//`, `and`, `or` αποτιμούν από αριστερά προς τα δεξιά και σταματούν μόλις το αποτέλεσμα καθοριστεί. Η τιμή ολόκληρης της έκφρασης είναι το τελευταίο που πραγματικά αποτιμήθηκε - που είναι ένας από τους τελεστέους, όχι ένα κανονικοποιημένο αληθές/ψευδές. Δείτε *logical* και το *boolean-logic tutorial*.
- **Προτεραιότητα.** Σας λέει πώς δένουν οι τελεστές όταν δεν υπάρχουν παρενθέσεις. Ο πλήρης πίνακας βρίσκεται στο *precedence* και κάθε σελίδα συνδέει τις γραμμές που την αφορούν.

- **Lvalue.** Οι περισσότεροι τελεστές παράγουν *rvalues* (τιμές που μπορείτε να διαβάσετε). Μερικοί - ανάθεση, αποαναφορά, τριαδικός σε ορισμένες μορφές - παράγουν *lvalues* (θέσεις στις οποίες μπορείτε να γράψετε).

Δείτε επίσης

- *Λογική Boole για προγραμματιστές της Perl*
  - ο εννοιολογικός σύντροφος των σελίδων *logical* και *bitwise*.
- *Regular expressions guide*
  - paired with *binding*.
- *perlfunc* - ενσωματωμένες συναρτήσεις, συμπεριλαμβανομένου κάθε ονόματος που αναφέρεται στην ενότητα *ονομασμένοι μοναδιαίοι τελεστές* της σελίδας *precedence*.

5.1.4 Τύποι δεδομένων

Η Perl έχει τρεις θεμελιώδεις τύπους δεδομένων - **βαθμωτά (scalars)**, **πίνακες (arrays)** και **hashes** - καθώς και ένα μικρό αριθμό υποστηρικτικών ειδών (αναφορές, typeglobs, κώδικας) που τεχνικά είναι βαθμωτά αλλά συμπεριφέρονται αρκετά διαφορετικά ώστε να χρήζουν ξεχωριστής αντιμετώπισης. Κάθε τιμή που χειρίζεται η γλώσσα ανήκει σε κάποιον από αυτούς.

Αυτή η αναφορά χωρίζεται σε μία σελίδα ανά θέμα. Θέματα που μοιράζονται περιβάλλον (μετατροπή αριθμητικών έναντι συμβολοσειρών, αναφορές έναντι typeglobs, περιβάλλον έναντι παρεμβολής) βρίσκονται στην ίδια σελίδα· θέματα που δεν μοιράζονται περιβάλλον αποκτούν δική τους σελίδα.

Επιλέξτε θέμα

Βαθμωτά

Ένα βαθμωτό είναι η μοναδιαία τιμή της Perl. Περιέχει ένα από τα εξής: έναν αριθμό (ακέραιο ή κινητής υποδιαστολής), μια συμβολοσειρά, μια αναφορά ή undef. Το sigil είναι $, σε κάθε θέση όπου η τιμή είναι ένα μεμονωμένο βαθμωτό - συμπεριλαμβανομένης της περίπτωσης που η τιμή είναι ένα στοιχείο πίνακα ή 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
```

Ένα βαθμωτό *δεν* είναι στατικά τυποποιημένο. Η ίδια μεταβλητή μπορεί να περιέχει έναν αριθμό τώρα και μια συμβολοσειρά λίγο αργότερα. Η Perl μετατρέπει μεταξύ των δύο κάθε φορά που ένας τελεστής απαιτεί τη μία μορφή ή την άλλη· δείτε *numbers and strings*.

Αλήθεια, ορισμένη τιμή και undef

Δύο σχετικές αλλά διακριτές ερωτήσεις για ένα βαθμωτό:

```
defined $x      # is there any value at all?
$x ? T : F      # is the value true?
```

Ένα βαθμωτό είναι **ψευδές** σε λογικό περιβάλλον αν είναι ένα από τα εξής:

```
undef               # the undefined value
0                   # numeric zero
0.0                 # also numeric zero
""                  # empty string
"0"                 # the literal string "0"
```

Όλα τα υπόλοιπα είναι αληθή. Σημειώστε ότι τα `"00"`, `"0.0"`, `"0 but true"` και `"0E0"` είναι όλα *αληθή* - είναι μη κενές συμβολοσειρές που δεν ταιριάζουν με το κυριολεκτικό `"0"`. Τα τελευταία δύο είναι κοινά ιδιώματα για «αυτή η συνάρτηση επέστρεψε επιτυχώς αλλά δεν υπήρχε τίποτα να μετρηθεί»:

```
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(...))`
}
```

Η `defined` είναι ο μόνος τρόπος να διακρίνετε την `undef` από τις ψευδείς-αλλά-ορισμένες τιμές. Ένα συνηθισμένο σφάλμα:

```
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
```

Χρησιμοποιήστε την `defined` όταν το μόνο που σας ενδιαφέρει είναι «υπάρχει εδώ τιμή;»· την `exists` για «υπάρχει αυτό το κλειδί/δείκτης στον περιέκτη;» (οι δύο διαφέρουν σε αραιούς πίνακες και σε κλειδιά που έχουν υποστεί `delete`).

Δυναμική τυποποίηση

Ένα βαθμωτό μπορεί να αλλάξει μορφή κατά την εκτέλεση:

```
my $x = 42;           # internally an integer
$x = "hello";         # now a string
$x = 3.14;            # now a float
$x = \@items;         # now a reference
```

Ο περισσότερος κώδικας δεν χρειάζεται ποτέ να σκεφτεί ποια εσωτερική μορφή έχει αυτή τη στιγμή ένα βαθμωτό. Οι εξαιρέσεις είναι βιβλιοθήκες σειριοποίησης JSON που ενδιαφέρονται αν το 1 ανατέθηκε ως αριθμός ή ως συμβολοσειρά, και οι βοηθοί ενδοσκόπησης του `Scalar::Util` (`looks_like_number`, `dualvar`).

«Μοιάζει με αριθμό»

Όταν η Perl μετατρέπει μια συμβολοσειρά σε αριθμό, σταματάει στον πρώτο χαρακτήρα που δεν ανήκει σε ένα έγκυρο αριθμητικό πρόθεμα:

```
"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
```

Η ασυμφωνία μεταξύ αριθμητικών *κυριολεκτικών* (που αναγνωρίζουν `_` και `0x`) και αριθμητικών *συμβολοσειρών* (που δεν το κάνουν) είναι μία από τις συχνότερα αναφερόμενες ερωτήσεις «γιατί» στην Perl. Χρησιμοποιήστε `hex` / `oct` όταν η είσοδος είναι συμβολοσειρά σε μη δεκαδική μορφή:

```
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
```

Με `use warnings`, μια αριθμητική πράξη σε μη αριθμητική συμβολοσειρά εκπέμπει *Argument «...» isn't numeric in addition (+)*. Η πράξη επιστρέφει 0 - η προειδοποίηση, όχι ένα μοιραίο σφάλμα, είναι το σήμα.

Διπλές τιμές

Ορισμένα βαθμωτά φέρουν ταυτόχρονα μορφή συμβολοσειράς και αριθμού, με επιλογή ανάλογα με το περιβάλλον. Το κλασικό παράδειγμα είναι η `$!`:

```
$!                # in string context: "Permission denied"
$! + 0            # in numeric context: 13
```

Μπορείτε να φτιάξετε τη δική σας με `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"
```

Τα βαθμωτά διπλής τιμής είναι ένα εξειδικευμένο εργαλείο. Η συνηθισμένη περίπτωση όπου έχουν σημασία είναι η `$!` (errno)· η γενική διέξοδος είναι ο βοηθός dualvar.

«0 but true»

Η κυριολεκτική συμβολοσειρά `"0 but true"` είναι το κανονικό ιδίωμα της Perl για «μια συνάρτηση της οποίας η κανονική επιστροφή είναι ένα πλήθος, αλλά το πλήθος είναι μηδέν και η κλήση πέτυχε». Είναι αληθής (μη κενή, όχι `"0"`) αλλά αριθμητικά ίση με 0:

```
my $hits = "0 but true";

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
if ($hits)        { print "succeeded\n"; }    # prints
print $hits + 0;                              # 0
print "got $hits results\n";                  # got 0 but true␣
↪results
```

Ο μεταγλωττιστής αναγνωρίζει επίσης ειδικά την `"0 but true"` και παραλείπει την προειδοποίηση *isn't numeric* που θα εξέπεμπε οποιαδήποτε άλλη συμβολοσειρά «0 ακολουθούμενο από σκουπίδια».

Δείτε επίσης

- *Numbers and strings* - πώς οι μορφές ακεραίου, κινητής υποδιαστολής και συμβολοσειράς μετατρέπονται μεταξύ τους και πού βρίσκονται οι γωνιακές περιπτώσεις.
- *References* - η `\X` παράγει ένα βαθμωτό που δείχνει σε μια άλλη τιμή.
- *Context* - πότε ένα βαθμωτό παίρνει μία τιμή έναντι πολλών.
- `defined`, `undef` - έλεγχος και παραγωγή της απροσδιόριστης τιμής.
- `scalar` - επιβολή βαθμωτού περιβάλλοντος σε λίστα ή πίνακα.
- `$_` - το σιωπηρό βαθμωτό για πολλές ενσωματωμένες συναρτήσεις.

Πίνακες

Ένας πίνακας είναι μια διατεταγμένη, αριθμημένη με ακεραίους ακολουθία βαθμωτών. Το sigil είναι `@` για ολόκληρο τον πίνακα και για τομές (που είναι λίστες)· `$` για ένα μεμονωμένο στοιχείο (που είναι ένα βαθμωτό). Το αρχικό sigil δηλώνει τον *τύπο αποτελέσματος*, όχι τον περιέκτη.

```
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
```

Οι δείκτες ξεκινούν από 0. Οι αρνητικοί δείκτες μετρούν από το τέλος (`$days[-1]` είναι `'Fri'`).

Μήκος: `$#arr` έναντι `scalar @arr`

Δύο σχετικά μεγέθη, που χρησιμοποιούνται για διαφορετικούς σκοπούς:

```
$#days             # last valid index            - 4
scalar @days       # number of elements          - 5
$#days + 1         # equivalent to scalar @days
```

Η ανάθεση στο `$#arr` αλλάζει το μέγεθος του πίνακα. Η συρρίκνωση καταστρέφει τα στοιχεία πέρα από το νέο τέλος· η αύξηση τα γεμίζει με `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
```

Η μορφή `@a = ()` είναι η ιδιωματική ανάθεση κενού πίνακα· η `$#a = -1` είναι ισοδύναμη και μερικές φορές πιο σαφής όταν η πρόθεση είναι «συρρίκνωση στο τίποτα χωρίς αλλαγή ταυτότητας της μεταβλητής».

Πρόσβαση στοιχείων

```
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
```

Η ανάγνωση πέρα από το τέλος επιστρέφει `undef` (χωρίς προειδοποίηση - η ανάγνωση εκτός εύρους δεν είναι γεγονός). Η εγγραφή πέρα από το τέλος *αυξάνει* τον πίνακα· οι ενδιάμεσες θέσεις γίνονται `undef`.

```
my @a;
$a[5] = 'fifth';
# @a is now (undef, undef, undef, undef, undef, 'fifth')
print scalar @a;                 # 6
```

Η ανάγνωση αρνητικού δείκτη πιο πίσω από την αρχή είναι *μοιραίο* σφάλμα:

```
my @a = (1, 2, 3);
$a[-10];                        # FATAL: Modification of non-
 ↪creatable
                                #        array value attempted
```

Λειτουργίες σε ολόκληρο τον πίνακα

```
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
```

Οι `shift` και `unshift` είναι O(n) επειδή αναριθμούν κάθε εναπομείνον στοιχείο· οι `push` και `pop` είναι O(1) αποσβεσμένα. Για φόρτους εργασίας τύπου ουράς όπου αυτό έχει

σημασία, προτιμήστε push/shift στο *σύντομο* άκρο ή χρησιμοποιήστε deque από τη List::Util (η οποία έχει αναγωγές, όχι deque).

Η splice είναι ο ελβετικός σουγιάς: αφαιρεί ένα τμήμα στοιχείων, εισάγει ένα τμήμα στη θέση τους, επιστρέφει αυτά που αφαιρέθηκαν:

```
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)
```

Αρνητικό $offset μετράει από το τέλος. Αρνητικό $len διατηρεί τόσα τελευταία στοιχεία:

```
splice @arr, -1;                  # remove just the last element
splice @arr, 0, -2;               # remove all but the last two
```

Τομές

Μια τομή εξάγει (ή αναθέτει σε) πολλαπλά στοιχεία ταυτόχρονα. Οι δείκτες προέρχονται από μια λίστα:

```
my @arr = ('a', 'b', 'c', 'd', 'e');

my @subset = @arr[0, 2, 4];          # ('a', 'c', 'e')
my @middle = @arr[1..3];             # ('b', 'c', 'd')
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)
```

Μια τομή *κλειδιού/τιμής* με % επιστρέφει ζεύγη δείκτη/τιμής (Perl 5.20+):

```
my @arr = ('a', 'b', 'c', 'd', 'e');
my %picked = %arr[1, 3];             # (1 => 'b', 3 => 'd')
```

Αυτή είναι η σωστή μορφή όταν χρειάζεστε τόσο τον δείκτη όσο και την τιμή επιλεγμένων στοιχείων. Δείτε *subscript* για τον πίνακα δώδεκα μορφών τομής.

Περιβάλλον λίστας έναντι βαθμωτού περιβάλλοντος

Ένας πίνακας σε περιβάλλον λίστας δίνει τα στοιχεία του· σε βαθμωτό περιβάλλον, δίνει το μήκος του:

```
my @arr = (10, 20, 30);

my @copy   = @arr;                # list context - three elements
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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"
```

Η διάκριση είναι η πιο συνηθισμένη παγίδα στην Perl. Δύο πρακτικοί κανόνες:

- *Μέσα σε συμβολοσειρές με διπλά εισαγωγικά*, οι πίνακες παρεμβάλλονται ως τα στοιχεία τους ενωμένα με `$"` (ένα κενό από προεπιλογή). Δείτε *interpolation*.
- *Σε περιβάλλον αριθμητικού ή συμβολοσειριακού τελεστή* (`+`, `.`, σύγκριση), ένας πίνακας δίνει το μήκος του.

Επιβάλετε βαθμωτό περιβάλλον με `scalar` όταν η θέση της έκφρασης είναι κατά τα άλλα αμφίσημη:

```
print "got " . scalar(@arr) . " items\n";    # "got 3 items"
print "got @arr items\n";                    # "got 10 20 30 items"␣
↪- different!
```

Ανάθεση λίστας και το ιδίωμα μέτρησης

Η ανάθεση μιας λίστας σε έναν πίνακα κάνει τον πίνακα ακριβώς τόσο μεγάλο:

```
my @arr = (1, 2, 3);
@arr = (10, 20);                    # @arr is now (10, 20)
@arr = ();                          # @arr is now empty
```

Η ανάθεση λίστας σε μια *λίστα* βαθμωτών αποσυσκευάζει (τα περισσευούμενα απορρίπτονται· τα ελλείποντα γίνονται `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
```

Μια ανάθεση λίστας σε βαθμωτό περιβάλλον επιστρέφει τον αριθμό στοιχείων στη *δεξιά* πλευρά. Αυτή είναι η πηγή του ιδιώματος μέτρησης:

```
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
```

Το `() = LIST` στη μέση είναι το τέχνασμα. Χωρίς αυτό, η `$count = $s =~ /\d+/g` δίνει λογική τιμή αντί για πλήθος.

Αναφορές πίνακα

Οι πίνακες δεν είναι τιμές πρώτης τάξης υπό την έννοια ότι μπορούν να μεταβιβαστούν κατ" όνομα μέσα από μια αλυσίδα κλήσεων συναρτήσεων χωρίς ισοπέδωση σε λίστα. Για να μεταβιβάσετε έναν πίνακα χωρίς ισοπέδωση, πάρτε μια **αναφορά** σε αυτόν:

```
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
```

Δείτε *references* για την πλήρη εικόνα.

Πραγματικό παράδειγμα: κατασκευή ιστογράμματος μηκών λέξεων

```
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
# 4-letter words: 2
# 5-letter words: 3
```

Σημειώστε ότι η `$hist[length $_]++` αυτο-δημιουργεί τη θέση όταν πρόκειται για την πρώτη λέξη αυτού του μήκους. Η ανάγνωση από `$hist[7]` αργότερα επιστρέφει `undef`, την οποία η `++` αυξάνει χωρίς πρόβλημα - αλλά η `next unless $hist[$len]` παραλείπει το κενό πριν την εκτύπωση.

Δείτε επίσης

- *Hashes* - όταν ο δείκτης είναι συμβολοσειρά, όχι ακέραιος.
- *References* - μεταβίβαση πινάκων χωρίς ισοπέδωση.
- *Context* - η ιστορία λίστας-έναντι-βαθμωτού αναλυτικά.
- *Interpolation* - πώς εμφανίζονται οι πίνακες μέσα σε `""`.
- *Subscript and slice operators* - η οικογένεια αγκυλών και αγκίστρων αναλυτικά.
- *Range operator* `..` - η πιο συνηθισμένη πηγή λιστών δεικτών για τομές.

- `push`, `pop`, `shift`, `unshift`, `splice` - λειτουργίες σε ολόκληρο τον πίνακα.
- `scalar` - επιβολή βαθμωτού περιβάλλοντος (μήκος).
- `reverse`, `sort` - μετασχηματισμοί ολόκληρου πίνακα.

Hashes

Ένα hash είναι μια μη διατεταγμένη συλλογή ζευγών κλειδιού-τιμής όπου τα κλειδιά είναι συμβολοσειρές και οι τιμές είναι βαθμωτά. Το sigil είναι `%` για ολόκληρο το hash, `@` για τομές τιμών και τομές κλειδιού/τιμής, και `$` για μία τιμή:

```
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
```

Τα κλειδιά hash εξαναγκάζονται πάντα σε συμβολοσειρές. Η αποθήκευση με ακέραιο κλειδί `42` και η ανάγνωση με κλειδί συμβολοσειράς `"42"` ανακτούν την ίδια θέση - είναι το ίδιο κλειδί.

Αρχικοποίηση: ζεύγη, fat comma και `%h = LIST`

Ένα hash αρχικοποιείται από μια λίστα ζυγού μήκους, σε σειρά κλειδιού/τιμής:

```
my %h = ('a', 1, 'b', 2);                    # legal but ugly
my %h = (a => 1, b => 2);                    # idiomatic
```

Το fat comma `=>` είναι ένα κόμμα που *επίσης* κάνει αυτόματα εισαγωγικά σε ένα bareword στα αριστερά που μοιάζει με αναγνωριστικό. Έτσι το `a => 1` είναι ακριβώς `'a', 1`. Η αυτόματη εισαγωγή σε εισαγωγικά απαιτεί το bareword να είναι απλό αναγνωριστικό - το `2.0 => 'x'` αναλύεται ως ο αριθμός 2, όχι η συμβολοσειρά `"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
```

Αν ένα κλειδί εμφανίζεται περισσότερες από μία φορές στη λίστα αρχικοποίησης, **κερδίζει η τελευταία εμφάνιση**. Το τυπικό ιδίωμα για «συγχώνευση με υπερισχύσεις» εκμεταλλεύεται αυτό:

```
my %config = (%defaults, %overrides);
# %config has every default key, with %overrides values where they␣
↪collide
```

Πρόσβαση, με τις τέσσερις μορφές sigil

Η ανάγνωση και εγγραφή μίας θέσης γίνεται με sigil $:

```
my $name = $user{name};                 # read
$user{city} = 'NYC';                    # write
$user{age}++;                           # arithmetic on a hash value
delete $user{email};                    # remove the slot entirely
```

Οι άλλες τρεις μορφές subscript αντιστοιχούν σε αυτό που θέλετε πίσω:

```
@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
```

Δείτε *subscript* για τον πλήρη πίνακα δώδεκα μορφών και τον κανόνα αυτόματης εισαγωγής bareword σε εισαγωγικά μέσα σε {}.

`exists` έναντι `defined` έναντι αλήθειας

Τρεις διακριτές ερωτήσεις για ένα κλειδί:

```
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")?
```

Αυτές διαφέρουν όταν:

```
$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
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
exists  $h{c}        # FALSE - never set, or deleted
defined $h{c}        # FALSE
        $h{c}        # FALSE
```

Επιλέξτε κατά πρόθεση: `exists` για *παρουσία*, `defined` για *τιμή-όχι-undef*, αλήθεια για *ουσιαστική τιμή*. Η χρήση `if ($h{key})` όταν εννοείτε `exists` είναι πηγή σφαλμάτων - τα `0` και `""` είναι έγκυρες τιμές που ο έλεγχος απορρίπτει.

Επανάληψη: 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";
}
```

Η `each` μεταφέρει κατάσταση επαναλήπτη πάνω στο ίδιο το hash· η κλήση `keys` (ή `values`) στο hash *επαναφέρει* αυτόν τον επαναλήπτη. Ο συνδυασμός των δύο παράγει σφάλματα βρόχου δύσκολα στην αποσφαλμάτωση:

```
while (my ($k, $v) = each %h) {
    if (some_condition($k)) {
        print "size: ", scalar keys %h, "\n";   # resets each()
↪iterator!
        # next iteration of while() starts over from the top
    }
}
```

Διόρθωση: είτε συσσωρεύστε τη λίστα με `keys` μία φορά στην αρχή, είτε αποφύγετε εντελώς την `each`. Η περισσότερη επανάληψη σε hash γράφεται καλύτερα με `keys`:

```
for my $k (sort keys %h) {            # bonus: defined ordering
    ...
}
```

Σειρά κλειδιών

Τα κλειδιά hash εξάγονται σε σειρά **σειράς-εισαγωγής-διαταραγμένης-από-τυχαιοποίηση-hash** - δηλαδή δεν υπάρχει εγγυημένη σειρά. Δύο εκτελέσεις του ίδιου προγράμματος με την ίδια είσοδο μπορεί να επαναλαμβάνουν τα κλειδιά σε διαφορετική σειρά. Αυτό είναι χαρακτηριστικό ασφαλείας (αποτρέπει επιθέσεις αλγοριθμικής πολυπλοκότητας εναντίον της συνάρτησης κατακερματισμού)· είναι επίσης μια επαναλαμβανόμενη πηγή εύθραυστων ελέγχων:

```
my %h = (a => 1, b => 2, c => 3);
print "$_ " for keys %h;        # output order is undefined
```

Αν χρειάζεστε σταθερή σειρά, ταξινομήστε:

```
print "$_ " for sort keys %h;          # alphabetical
print "$_ " for sort { $h{$a} <=> $h{$b} } keys %h;   # by value
```

Αναφορές hash

Ένα hash, όπως και ένας πίνακας, ισοπεδώνεται όταν περνάει μέσω λίστας. Για να μεταβιβάσετε ένα hash χωρίς ισοπέδωση, πάρτε μια **αναφορά**:

```
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
```

Δείτε *references* για την πλήρη εικόνα.

Πραγματικό παράδειγμα: μέτρηση και ομαδοποίηση

Ο μετρητής συχνότητας είναι το σχολικό μικρό παράδειγμα για hashes:

```
my @words = qw(apple banana apple cherry apple banana);
my %count;
$count{$_}++ for @words;
# %count = (apple => 3, banana => 2, cherry => 1)
```

Η ομαδοποίηση είναι το επόμενο βήμα - κατασκευή hash του οποίου οι τιμές είναι αναφορές πίνακα:

```
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'])
```

Η γραμμή `push @{ $by_dept{$_->{dept}} }, ...` αυτο-δημιουργεί: όταν το κλειδί `eng` δεν υπάρχει ακόμα, δημιουργείται η θέση στο hash και εισάγεται ένας ανώνυμος

πίνακας, στη συνέχεια η τιμή γίνεται push. Δείτε *references* για την ιστορία της αυτο-δημιουργίας.

Τομή hash ως πολλαπλή ανάκτηση/ανάθεση

```
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
```

Συνδυάστε με τομή %h{...} για «εξαγωγή εγγραφής»:

```
my %fields = %config{qw(host port)};
# %fields = (host => 'localhost', port => 80)
```

Αυτός είναι ο κανονικός τρόπος να προβάλλετε ένα hash σε υποσύνολο των κλειδιών του. Χωρίς %h{...} θα έπρεπε να το φτιάξετε χειροκίνητα με βρόχο ή με map.

Δείτε επίσης

- *Arrays* - όταν ο δείκτης είναι ακέραιος.
- *References* - αυτο-δημιουργία, hash-of-arrays, hash-of-hashes· η μορφή $h->{a}{b}.
- *Subscript and slice operators* - ο κανόνας αυτόματης εισαγωγής bareword σε εισαγωγικά και οι τέσσερις μορφές τομής.
- exists, defined, delete - η τριάδα παρουσίας/τιμής/αφαίρεσης.
- keys, values, each - επανάληψη ολόκληρου hash.
- sort - για ταξινομημένη επανάληψη.
- tie - δημιουργία hash υποστηριζόμενου από κάτι άλλο εκτός από τον πίνακα κατακερματισμού στη μνήμη (αρχείο DBM, νωχελική βάση δεδομένων, ...).

Περιβάλλον

Σχεδόν κάθε τελεστής στην Perl αποτιμά τους τελεστέους του σε ένα συγκεκριμένο **περιβάλλον** - λίστας, βαθμωτό, κενό ή λογικό - και πολλοί τελεστές επιστρέφουν *διαφορετικά πράγματα* ανάλογα με το περιβάλλον στο οποίο κλήθηκαν. Η ίδια έκφραση μπορεί να σημαίνει διαφορετικά πράγματα σε διαφορετικές θέσεις:

```
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
```

Το περιβάλλον είναι η μεγαλύτερη μεμονωμένη διαφορά μεταξύ Perl και των περισσότερων άλλων γλωσσών, και η πιο συνηθισμένη πηγή εκπλήξεων «γιατί δεν κάνει αυτό που περιμένω».

Τα τέσσερα περιβάλλοντα

Περιβάλλον	Τι βλέπει ο τελεστέος	Επιβάλλεται από
λίστα	«δώσε μου όλες τις τιμές σου»	`@a = ...`, `(...)`, ορίσματα συνάρτησης, `print`
βαθμωτό	«δώσε μου μία τιμή»	`$s = ...`, `if/while`, `.`, `+`, σύγκριση
κενό	«πέταξε το αποτέλεσμα»	δήλωση της οποίας η τιμή δεν χρησιμοποιείται
λογικό	«δώσε μου αληθές/ψευδές»	`if`, `unless`, `while`, `&&`, `||`, `!`

Το λογικό περιβάλλον είναι τεχνικά μια ειδική περίπτωση βαθμωτού περιβάλλοντος - ζητείται μία τιμή από τον τελεστέο, στη συνέχεια αυτή η τιμή ελέγχεται για αλήθεια. Τα δύο δεν είναι πάντα ίδια: μια υπορουτίνα που γνωρίζει `wantarray` μπορεί να ανιχνεύσει ξεχωριστά το λογικό περιβάλλον (δείτε παρακάτω).

Επιβολή περιβάλλοντος

Η αριστερή πλευρά μιας ανάθεσης επιβάλλει περιβάλλον στη δεξιά:

```
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
```

Η πιο συνηθισμένη παγίδα είναι η `my $x = func()` έναντι `my ($x) = func()`. Χωρίς παρενθέσεις, η ανάθεση είναι βαθμωτή· με αυτές, είναι ανάθεση λίστας 1 στοιχείου. Αν η `func()` επιστρέφει τρία πράγματα σε περιβάλλον λίστας και «το πλήθος» σε βαθμωτό περιβάλλον, η διαφορά είναι μεταξύ `'first-thing'` και `3`.

Οι τελεστές επιβάλλουν επίσης περιβάλλον. Οι αριθμητικοί και συμβολοσειριακοί τελεστές επιβάλλουν βαθμωτό περιβάλλον στους τελεστέους τους· η `print` επιβάλλει λίστα:

```
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"
```

Και οι λογικοί έλεγχοι επιβάλλουν βαθμωτό (λογικό) περιβάλλον:

```
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
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
                          #   special: it reads ONE line and assigns␣
↪to $_
```

Τι δίνει ένας πίνακας σε βαθμωτό περιβάλλον

Ένας **πίνακας** σε βαθμωτό περιβάλλον δίνει το **μήκος** του:

```
my @arr = (1, 2, 3);
my $n = @arr;              # 3
```

Μια **κυριολεκτική λίστα** σε βαθμωτό περιβάλλον δίνει το **τελευταίο στοιχείο** της (ο τελεστής κόμματος):

```
my $x = (1, 2, 3);         # 3   - comma operator, last element
my $x = ('a', 'b', 'c'); # 'c'
```

Αυτά μοιάζουν πανομοιότυπα αλλά συμπεριφέρονται διαφορετικά. Ο κανόνας είναι: οι πίνακες έχουν μήκος, οι κυριολεκτικές λίστες δεν έχουν. Μια σκέτη λίστα με κόμμα είναι ο τελεστής κόμματος της C και παράγει τον τελευταίο τελεστέο· μια μεταβλητή πίνακα σε βαθμωτή θέση ρωτάει «πόσα»:

```
my @a = ('x', 'y', 'z');
my $count   = @a;          # 3
my $last_el = $a[-1];      # 'z'
```

Ο τελεστής `scalar` επιβάλλει βαθμωτό περιβάλλον σε ό,τι ακολουθεί, κάτι που είναι ιδιαίτερα χρήσιμο όταν η θέση της έκφρασης είναι κατά τα άλλα αμφίσημη:

```
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 ""
```

Τι δίνει ένα hash σε βαθμωτό περιβάλλον

Ένα μη κενό hash σε βαθμωτό περιβάλλον επιστρέφει μια αληθή τιμή (το πλήθος κλειδιών, από Perl 5.25)· ένα κενό hash επιστρέφει την ψευδή τιμή 0. Η μορφή που έχει σημασία είναι η τιμή αλήθειας:

```
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 5.44) `scalar %h` returns the key count and is fine.

wantarray - σύνταξη υπορουτινών ευαίσθητων στο περιβάλλον

Μια υπορουτίνα μπορεί να ρωτήσει «σε ποιο περιβάλλον κλήθηκα;» με `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
```

Η τριμερής διάκριση είναι ο μόνος τρόπος ανίχνευσης *κενού* περιβάλλοντος μέσα από μια υπορουτίνα. Χρησιμοποιήστε την με φειδώ: οι περισσότερες υπορουτίνες επιστρέφουν μία μορφή και αφήνουν τον καλούντα να ασχοληθεί με τον εξαναγκασμό περιβάλλοντος. Οι περιπτώσεις όπου η `wantarray` αξίζει τον κόπο είναι:

- Συναρτήσεις που νόμιμα επιστρέφουν είτε «τη λίστα» είτε «το πλήθος» - π.χ. ρουτίνες αναζήτησης κειμένου.
- Βοηθοί διαγνωστικών που εκτυπώνουν προειδοποίηση όταν το αποτέλεσμά τους απορρίφθηκε.

Μια υπορουτίνα που επιστρέφει διαφορετικές *σημασίες* (όχι απλώς διαφορετικές μορφές) βάσει `wantarray` είναι κίνδυνος για τη συντήρηση. Δύο κατονομασμένες συναρτήσεις διαβάζονται καλύτερα από μία πολυμορφική.

Παγίδα: `scalar(@arr)` έναντι `@arr` σε διαφορετικές θέσεις

```
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

# 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
```

Η ασυμφωνία μεταξύ της περίπτωσης 2 (συνένωση → βαθμωτό) και της περίπτωσης 3

(μέσα σε "" → λίστα) είναι η πιο συχνά ερωτώμενη ερώτηση περιβάλλοντος στην Perl. Μέσα σε "", η `@arr` είναι παρεμβολή πίνακα· εκτός "", το περιβάλλον της περιβάλλουσας έκφρασης αποφασίζει.

Παγίδα: ισοπέδωση λίστας στα ορίσματα συναρτήσεων

Μια συνάρτηση λαμβάνει τα `@_` ως μία ενιαία επίπεδη λίστα όλων των ορισμάτων - δεν υπάρχει τρόπος η καλούμενη να ξέρει πού τελείωσε ο ένας πίνακας και πού ξεκίνησε ο επόμενος:

```
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)
```

Αυτός είναι *ο* λόγος που υπάρχουν οι αναφορές: η μεταβίβαση μιας μη επίπεδης μορφής μέσω της σύμβασης κλήσης που ισοπεδώνει λίστες. Οτιδήποτε θέλετε να διατηρήσει την ταυτότητά του μέσα στα `@_` πρέπει να φτάσει ως αναφορά.

Πραγματικό παράδειγμα: μια υπορουτίνα που επιστρέφει πλήθος ή λίστα

```
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
```

Για να κάνετε τη `digits_of` να επιστρέφει *πλήθος* σε βαθμωτό περιβάλλον, η δεξιά πλευρά πρέπει να εξαναγκάσει ρητά:

```
sub digits_of {
    my ($s) = @_;
    my @d = $s =~ /\d/g;
    return wantarray ? @d : scalar @d;
}

my @digits = digits_of("a1b2c3");      # ('1', '2', '3')
my $n      = digits_of("a1b2c3");      # 3
```

Αυτό είναι το κανονικό ιδίωμα «επιστροφή λίστας ή του μήκους της». Σημειώστε ότι χωρίς τον διακόπτη `wantarray`, η `return @d` θα έδινε το πλήθος σε βαθμωτό περιβάλλον ούτως ή άλλως (λόγω του κανόνα πίνακα-σε-βαθμωτό)

- αλλά είναι πιο σαφές να είναι ρητό.

Δείτε επίσης

- *Arrays* - η τριχοτομία πίνακα-λίστας-βαθμωτού αναλυτικά.
- *Hashes* - %h σε βαθμωτό περιβάλλον.
- *Comma* - ο τελεστής που κατασκευάζει κυριολεκτικές λίστες και δίνει τον τελευταίο τελεστέο σε βαθμωτό περιβάλλον.
- `wantarray` - ο τρόπος ερώτησης μέσα από υπορουτίνα.
- `scalar` - επιβολή βαθμωτού περιβάλλοντος.
- `reverse` - ευαίσθητη στο περιβάλλον: λίστα βαθμωτών σε περιβάλλον λίστας, μία αντεστραμμένη συμβολοσειρά σε βαθμωτό περιβάλλον.
- *References* - μεταβίβαση μιας μη επίπεδης μορφής μέσω της σύμβασης ορισμάτων σε περιβάλλον λίστας που ισοπεδώνει.

Αριθμοί και συμβολοσειρές

Ένα βαθμωτό μπορεί να περιέχει έναν αριθμό, μια συμβολοσειρά ή και τα δύο ταυτόχρονα. Η Perl μετατρέπει σιωπηλά μεταξύ των δύο κάθε φορά που ένας τελεστής απαιτεί μία μορφή - η + θέλει αριθμούς, η . θέλει συμβολοσειρές, οι τελεστές σύγκρισης έχουν ξεχωριστές αριθμητικές και συμβολοσειριακές μορφές. Η κατανόηση των κανόνων μετατροπής και του πού δαγκώνουν είναι ουσιαστική για τη συγγραφή προβλέψιμου κώδικα Perl.

Αριθμητικά κυριολεκτικά

Τα κυριολεκτικά ακεραίων εμφανίζονται σε πέντε βάσεις:

```
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+)
```

Τα κυριολεκτικά κινητής υποδιαστολής απαιτούν είτε τελεία είτε εκθέτη:

```
3.14
0.5                 # leading zero ok
.5                  # leading dot ok
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
1e3                 # 1000
1.5E-10             # exponent - uppercase or lowercase 'e'
```

Οι κάτω παύλες γίνονται αποδεκτές μεταξύ ψηφίων οπουδήποτε - `3.14_15_92`, `1_000_000`. Πολλαπλές διαδοχικές κάτω παύλες προκαλούν προειδοποίηση: η `23__500` προειδοποιεί, η `23_500` όχι.

Τα δεκαεξαδικά και οκταδικά κυριολεκτικά κινητής υποδιαστολής χρειάζονται τον εκθέτη p (δύναμη του 2):

```
0x1.999ap-4         # hex float
0b101.01p-1         # binary float
0o7.65p2            # octal float
```

Ο εκθέτης p είναι *υποχρεωτικός* για μη δεκαδικά κυριολεκτικά κινητής υποδιαστολής. Η σκέτη `0x1F.5` είναι σφάλμα ανάλυσης.

Συμβολοσειρά → αριθμός

Όταν η Perl μετατρέπει μια συμβολοσειρά σε αριθμό, καταναλώνει το μεγαλύτερο πρόθεμα που μοιάζει με αριθμό και αγνοεί τα υπόλοιπα:

```
"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
```

Η ασυμφωνία μεταξύ αριθμητικών *κυριολεκτικών* (που αναγνωρίζουν `_`, `0x`, `0b`, `0`) και αριθμητικών *συμβολοσειρών* (που δεν αναγνωρίζουν τίποτα από αυτά) είναι ο κανόνας μετατροπής όπου σκοντάφτει κανείς πιο συχνά. Για μετατροπή συμβολοσειράς σε μη δεκαδική μορφή, χρησιμοποιήστε `hex` ή `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
```

Με `use warnings`, μια αριθμητική πράξη σε συμβολοσειρά που δεν *αρχίζει* με αριθμητικό πρόθεμα εκπέμπει *Argument «…» isn't numeric in addition (+)*. Το αποτέλεσμα παραμένει 0· η προειδοποίηση είναι το σήμα. Η συγγραφή ανεκτικού κώδικα που δέχεται είσοδο χρήστη συχνά σημαίνει φιλτράρισμα με `Scalar::Util::looks_like_number` ή regex πρώτα.

Αριθμός → συμβολοσειρά

Η Perl μετατρέπει αριθμό σε συμβολοσειρά χρησιμοποιώντας την ίδια λογική με `sprintf("%g", $n)` για αριθμούς κινητής υποδιαστολής και την προφανή δεκαδική αναπαράσταση για ακεραίους:

```
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"
```

Η προεπιλεγμένη ακρίβεια είναι αρκετά ψηφία για πλήρη κύκλο *των περισσότερων* double. Για ακριβή έλεγχο, μορφοποιήστε ρητά με `sprintf`:

```
sprintf "%.2f", 3.14159          # "3.14"
sprintf "%05d", 42               # "00042"
sprintf "%e", 1234567            # "1.234567e+06"
```

Η συνένωση εξαναγκάζει σε συμβολοσειρές· η αριθμητική εξαναγκάζει σε αριθμούς

```
"42" + "3"           # 45    - both coerced to numbers
42 . 3               # "423" - both coerced to strings
"3" x "5"            # "33333" - left operand string, right operand
 ↪number
```

Ο τελεστής `.` είναι ο τελεστής ρητής μετατροπής σε συμβολοσειρά· η `+` είναι ο τελεστής ρητής μετατροπής σε αριθμό. Ένα βαθμωτό που περιέχει αριθμό τον οποίο θέλετε ως συμβολοσειρά μπορεί να συνενωθεί με `""`:

```
my $s = "" . $n;     # idiomatic "stringify"
my $s = "$n";        # same; usually clearer
```

Ένα βαθμωτό που περιέχει συμβολοσειρά την οποία θέλετε ως αριθμό προσθέτει 0:

```
my $n = $s + 0;      # idiomatic "numify"
```

Αριθμητική έναντι συμβολοσειριακής σύγκρισης

Δύο παράλληλες οικογένειες τελεστών σύγκρισης με **διαφορετική σημασιολογία**:

Αριθμητικός	Συμβολοσειρά	Σημασία
==	eq	ίσο
!=	ne	άνισο
<	lt	μικρότερο
<=	le	μικρότερο ή ίσο
>	gt	μεγαλύτερο
>=	ge	μεγαλύτερο ή ίσο
<=>	cmp	τριμερής σύγκριση

Η σειρά ταξινόμησης διαφέρει δραματικά σε τιμές που μοιάζουν με αριθμούς αλλά είναι μεγάλες:

```
"10" <  "9"          # FALSE - 10 < 9 numerically
"10" lt "9"          # TRUE  - '1' < '9' lexicographically
```

Ταξινομήστε αριθμούς αριθμητικά:

```
my @sorted = sort { $a <=> $b } @numbers;       # numeric
my @sorted = sort                @strings;       # default - string
my @sorted = sort { $a cmp $b }  @strings;       # explicit string
```

Η επιλογή λάθος τελεστή είναι η πηγή σφαλμάτων «η ταξινόμησή μου είναι λάθος». Δείτε *numeric comparison* και *string comparison*.

Κινητή υποδιαστολή: ακρίβεια, Inf, NaN

Οι περισσότερες Perl χρησιμοποιούν αριθμούς κινητής υποδιαστολής διπλής ακρίβειας IEEE 754 - περίπου 15–17 δεκαδικά ψηφία ακρίβειας. Η κλασική έκπληξη:

```
0.1 + 0.2 == 0.3      # FALSE - neither side is exactly␣
↪representable
                      # 0.1 + 0.2 = 0.30000000000000004
```

Για χρήματα ή οτιδήποτε *πρέπει* να είναι ακριβές, δουλέψτε σε ακεραίους (λεπτά αντί για ευρώ) ή χρησιμοποιήστε ένα άρθρωμα bignum:

```
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;
```

Οι Inf και NaN είναι πραγματικές τιμές:

```
my $inf = 9 ** 9 ** 9;       # Inf - overflow
my $nan = $inf - $inf;       # NaN - undefined arithmetic
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
my $nan = "NaN" + 0;          # NaN - accepted from string

$nan == $nan                   # FALSE - NaN never equals anything
$nan != $nan                   # TRUE  - including itself
```

Οι `Inf` και `NaN` δεν κάνουν διάκριση πεζών-κεφαλαίων στην είσοδο και έχουν αρκετές ορθογραφίες (`Infinity`, `1.#INF`, ...)· στην έξοδο η Perl κανονικοποιεί στις σύντομες μορφές.

Σημειώστε ότι η perl5 παράγει *μοιραία* σφάλματα για `1/0` και `sqrt(-1)`· αυτά δεν παράγουν `NaN`. Είναι εξαιρετικά, όχι αριθμητικά.

Ανάλυση αριθμών επηρεαζόμενη από τοπικές ρυθμίσεις

Με `use locale` και ενεργή τοπική ρύθμιση POSIX, η υποδιαστολή είναι ό,τι ορίζει η τοπική ρύθμιση. Αυτό επηρεάζει την ανάλυση και τη μετατροπή σε συμβολοσειρά των αριθμών κινητής υποδιαστολής:

```
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
```

Αυτό σπάνια είναι αυτό που θέλετε για ανταλλαγή δεδομένων. Ο περισσότερος κώδικας που διαβάζει ή γράφει δομημένα δεδομένα δεν πρέπει ρητά να βρίσκεται στην εμβέλεια `use locale`, ώστε η `.` να είναι πάντα η υποδιαστολή. Χρησιμοποιήστε μορφοποίηση τοπικών ρυθμίσεων μόνο στο όριο όπου η έξοδος προορίζεται για ανθρώπινο αναγνώστη.

«Μοιάζει με αριθμό»

Οι κανόνες για το αν μια συμβολοσειρά είναι «αρκετά αριθμητική» δεν είναι απλοί για ενσωματωμένο χειρισμό· χρησιμοποιήστε `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
```

Αυτή είναι η συνάρτηση επικύρωσης - αυστηρή, χωρίς σκουπίδια στην αρχή, χωρίς σκουπίδια στο τέλος, χωρίς προθέματα hex/oct. Ταιριάζει τις συνθήκες υπό τις οποίες η Perl θα χρησιμοποιήσει τη cached αριθμητική μορφή ενός αριθμού αντί να αναλύει εκ νέου σε κάθε αριθμητική πράξη.

Δείτε επίσης

- *Scalars* - η ιστορία διπλής μορφής / dualvar· τιμές αλήθειας.
- *Numeric comparison* - ==, <=>.
- *String comparison* - eq, cmp.
- *Arithmetic* - +, -, *, /, %, **.
- *String* - . (συνένωση), x (επανάληψη).
- `hex`, `oct` - μετατροπή μη δεκαδικών αριθμητικών συμβολοσειρών.
- `int`, `abs`, `sprintf` - ρητή αριθμητική μετατροπή και μορφοποίηση.
- `Scalar::Util` - `looks_like_number`, `dualvar`.

Αναφορές

Μια αναφορά είναι ένα βαθμωτό που *δείχνει σε* μια άλλη τιμή. Οι αναφορές είναι ο τρόπος που η Perl κατασκευάζει εμφωλευμένες δομές δεδομένων - πίνακες πινάκων, hash από hashes, hash από πίνακες από hashes - επειδή οι βασικοί τύποι περιέκτη (πίνακες, hashes) ισοπεδώνονται όταν εμφανίζονται σε λίστες, και μια αναφορά είναι ένα μεμονωμένο βαθμωτό που επιβιώνει της ισοπέδωσης.

```
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)
```

Μια αναφορά είναι η ίδια ένα βαθμωτό. Μπορεί να μεταβιβαστεί μέσω οποιασδήποτε σύμβασης κλήσης που ισοπεδώνει λίστες χωρίς να χάσει την ταυτότητά της, να αποθηκευτεί σε πίνακες και hashes, να επιστραφεί από υπορουτίνες, να ασθενοποιηθεί, να σειριοποιηθεί - οπουδήποτε μπορεί να πάει ένα βαθμωτό.

Πράγματα στα οποία μπορείτε να κάνετε αναφορά

Επτά πράγματα δέχονται αναφορά:

```
\$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)
```

Επιπλέον οι **ανώνυμοι** κατασκευαστές, που κατασκευάζουν την τιμή και επιστρέφουν αναφορά σε ένα βήμα:

```
my $aref = [1, 2, 3];                  # anonymous array
my $href = { name => 'Alice', age => 30 };  # anonymous hash
my $cref = sub { $_[0] * 2 };          # anonymous sub
```

Η διάκριση αγκυλών-παρενθέσεων είναι ουσιαστική: η `(1, 2, 3)` είναι μια **λίστα**, η `[1, 2, 3]` είναι μια **ανώνυμη αναφορά πίνακα**. Η πρώτη ισοπεδώνεται· η δεύτερη όχι.

Αποαναφορά

Τέσσερις ορθογραφίες, σε αύξουσα σειρά συχνότητας εμφάνισης σε σύγχρονο κώδικα:

```
${$sref}              # full sigil-deref form
$$sref                # short sigil-deref form (only if name is␣
↪simple)
$sref->{key}          # arrow form (postfix; for arrays and hashes)
$sref->[0]            # same, for arrays
```

Η μορφή βέλους `->` είναι η ιδιωματική ορθογραφία για εμφωλευμένη πρόσβαση:

```
$config->{database}{host}            # arrow elision between␣
↪subscripts
$config->{database}->{host}          # explicit; same meaning
${$config->{database}}{host}         # full form; the same again
```

Μεταξύ δύο διαδοχικών subscripts το `->` είναι **προαιρετικό** (ο κανόνας «παράλειψης βέλους»)· στην αρχή μιας αλυσίδας είναι υποχρεωτικό:

```
$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
```

Αποαναφορά ολόκληρου περιέκτη:

```
@$aref                       # the whole array
%$href                       # the whole hash, as flat key/value␣
↪list
&$cref(@args)                # call the sub
$cref->(@args)               # idiomatic call form
```

Χρησιμοποιήστε τη μορφή με αγκύλες `@{ EXPR }` / `%{ EXPR }` όταν ο στόχος αποαναφοράς είναι μια έκφραση πιο σύνθετη από μια σκέτη βαθμωτή μεταβλητή:

```
my @found = @{ $config->{users} };   # deref a chain - needs braces
```

Δείτε *arrow* για τον τελεστή και *subscript* για την οικογένεια αγκυλών.

Προτεραιότητα της αποαναφοράς με πρόθεμα έναντι της δεικτοδότησης

Τα πέντε sigils αποαναφοράς με πρόθεμα `$ @ * % &` δένουν **πιο σφιχτά** από τις μεταθεματικές αγκύλες δεικτοδότησης `[]` και `{}`. Έτσι αυτά τα δύο είναι ακριβώς η ίδια έκφραση:

```
$$aref[2][2]                    # confusing, but valid
$aref->[2][2]                   # clear; identical meaning
```

Το `$$aref[2][2]` αποαναφέρει το `$aref` *πρώτα*, και κατόπιν δεικτοδοτεί το αποτέλεσμα. **Δεν** είναι η ανάγνωση της C `*a[i]`, όπου η δεικτοδότηση θα εφαρμοζόταν πριν την αποαναφορά. Η Perl δεν έχει καμία κατασκευή που να δεικτοδοτεί πριν την αποαναφορά σε αυτή τη θέση· αν θέλετε «το πράγμα στο οποίο δείχνει το `$i`-οστό στοιχείο», γράψτε το αναλυτικά από αριστερά προς τα δεξιά με μια μεταθεματική αποαναφορά:

```
$aref->[$i]->$*                 # deref the i-th element explicitly
```

Η μορφή με βέλος είναι η αναγνώσιμη επιλογή για εμφωλευμένη πρόσβαση· η μορφή με προπορευόμενο sigil είναι παγίδα ακριβώς επειδή διαβάζεται ως η προτεραιότητα της C και σημαίνει το αντίθετο. Δείτε την *προτεραιότητα* για τον πλήρη πίνακα τελεστών.

Τι επιστρέφει η `ref`

Η `ref` αναφέρει το είδος πράγματος στο οποίο δείχνει μια αναφορά:

```
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             # ''
```

Για blessed αντικείμενα, η `ref` επιστρέφει την κλάση. Για να διακρίνετε μια blessed αναφορά hash από μια απλή αναφορά hash, χρησιμοποιήστε `Scalar::Util::reftype` (που επιστρέφει πάντα τη μορφή του υποκείμενου περιέκτη ανεξαρτήτως bless) ή `isa` (ο έλεγχος με γνώση κληρονομικότητας).

Μεταθεματική αποαναφορά (@*, %*, $*)

Η Perl 5.20+ πρόσθεσε μεταθεματική σύνταξη που αντικατοπτρίζει την αλυσίδα βέλους:

```
$ref->@*                # equivalent to @$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
```

Οι μεταθεματικές μορφές διαβάζονται από αριστερά προς τα δεξιά μαζί με την υπόλοιπη αλυσίδα βέλους, κάτι που είναι συχνά πιο σαφές όταν ο στόχος αποαναφοράς είναι ο ίδιος αλυσίδα:

```
$config->{users}->@*            # all users - chains naturally
@{$config->{users}}             # same; reads right-to-left
```

Επιλέξτε τη μορφή που διαβάζεται καλύτερα στο εκάστοτε πλαίσιο. Η μεταθεματική είναι η πιο σαφής επιλογή σε αλυσίδες· η μορφή με αρχικό sigil είναι πιο σύντομη για απλές βαθμωτές αναφορές.

Αυτο-δημιουργία

Η ανάγνωση ή εγγραφή μέσω αποαναφοράς σε θέση που δεν υπάρχει ακόμα αυτο-δημιουργεί την ενδιάμεση δομή:

```
my %h;
$h{a}{b}{c} = 1;
# %h is now (a => { b => { c => 1 } })  - three nested hashes built
```

Ο μηχανισμός: όταν ο στόχος ανάθεσης είναι `$h{a}{b}{c}`, η Perl βλέπει ότι η `$h{a}` είναι `undef`, το επόμενο subscript είναι `{b}` (subscript hash), οπότε κάνει τη `$h{a}` νέα αναφορά hash. Στη συνέχεια η `$h{a}{b}` είναι `undef`, το επόμενο subscript είναι `{c}` (επίσης hash), οπότε την κάνει νέα αναφορά hash. Τέλος αναθέτει 1 στο φύλλο.

Το ίδιο συμβαίνει για subscripts πίνακα:

```
my %by_dept;
push @{ $by_dept{eng} }, 'Alice';
# $by_dept{eng} did not exist; the push autoviv'd it as an arrayref
```

Η αυτο-δημιουργία στα **αριστερά** μιας ανάθεσης (ή ενός μεταλλάκτη push/pop/...) είναι αυτό που κάνει αυτό να λειτουργεί. Η αυτο-δημιουργία στα **δεξιά** μιας έκφρασης είναι πιο επικίνδυνη - η *ανάγνωση* `$h{a}{b}{c}` όταν η `$h{a}` είναι `undef` *επίσης* αυτο-δημιουργεί, γεμίζοντας το hash ως παρενέργεια της εξέτασης:

```
my %h;
exists $h{a}{b};          # accidentally creates $h{a} = {} as a␣
 ↪hash!
                           # %h is now (a => {})
```

Η λύση όταν πραγματικά θέλετε μόνο να ελέγξετε: αλυσίδωση `exists`:

```
exists $h{a} && exists $h{a}{b};        # short-circuits if $h{a}␣
↪absent
```

Αυτή είναι η παγίδα αυτο-δημιουργίας που δαγκώνει πιο συχνά. Οι έλεγχοι εμφωλευμένης ύπαρξης πρέπει να βραχυκυκλώνουν, αλλιώς αφήνουν πίσω θέσεις hash.

Hash πινάκων - το κανονικό παράδειγμα

```
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'],
# )

# Iterate:
for my $dept (sort keys %by_dept) {
    print "$dept: @{ $by_dept{$dept} }\n";
}
# eng: Alice Carol
# sales: Bob Dan
```

Η `@{ $by_dept{...} }` είναι η μορφή αποαναφοράς με αγκύλες. Είναι απαραίτητη επειδή ο στόχος αποαναφοράς είναι μια έκφραση (`$by_dept{$dept}`), όχι μια σκέτη μεταβλητή.

Η ίδια μορφή με μεταθεματική σύνταξη:

```
print "$dept: ", join(' ', $by_dept{$dept}->@*), "\n";
```

Επιλέξτε όποια βρίσκετε πιο σαφή· και οι δύο είναι ιδιωματικές στη σύγχρονη Perl.

Ασθενείς αναφορές και κυκλικές δομές

Μια αναφορά κρατάει τον αναφερόμενο ζωντανό (με αυξημένο refcount) μέχρι η ίδια η αναφορά να βγει εκτός εμβέλειας. Αυτό γίνεται πρόβλημα όταν δύο δομές δείχνουν η μία στην άλλη:

```
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.
```

Η λύση είναι μια **ασθενής αναφορά** - μια αναφορά που δεν αυξάνει το refcount. Η θέση `$child->{parent}` είναι αυτή που πρέπει να ασθενοποιηθεί (ο πίσω δείκτης):

```
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;
```

Μετά την ασθενοποίηση, η `$child->{parent}` διαβάζεται ως ο γονέας όσο υπάρχει, και γίνεται σιωπηλά `undef` όταν ο γονέας απελευθερωθεί. Η κανονική θέση για `weaken` είναι σε κάθε πίσω δείκτη σε δενδρική δομή όπου τα παιδιά δείχνουν πίσω στον γονέα τους.

Το κατηγόρημα `Scalar::Util::isweak` ελέγχει αν μια δεδομένη αναφορά είναι ασθενοποιημένη.

Ισότητα αναφορών

Δύο αναφορές είναι ίσες (αριθμητικά `==`, συμβολοσειριακά `eq`) αν και μόνο αν αναφέρονται στην **ίδια** υποκείμενη τιμή:

```
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
```

Η αριθμητική σύγκριση αναφορών συγκρίνει ουσιαστικά τη διεύθυνση μηχανής τους. Δύο αναφορές με το ίδιο περιεχόμενο αλλά διαφορετικές ταυτότητες συγκρίνονται ως άνισες. Η σε βάθος ισότητα χρειάζεται βοηθό από `Test::More` ή χειροκίνητο περιηγητή.

Μια αναφορά μετατρέπεται σε συμβολοσειρά σαν κάτι σαν `ARRAY(0x55ab12cd)` - όνομα κλάσης συν δεκαεξαδική διεύθυνση. Μη βασίζεστε στη μορφή· χρησιμοποιήστε την μόνο ως εκτύπωση αποσφαλμάτωσης.

Δείτε επίσης

- *Arrays*, *Hashes* - οι δύο περιέκτες στους οποίους δείχνουν οι περισσότερες αναφορές.
- *Typeglobs* - *name, η εγγραφή πίνακα συμβόλων που οι αναφορές σε globs σας επιτρέπουν να κάνετε ψευδώνυμο.
- *Arrow operator* - ->, η συντομογραφία αποαναφοράς.
- *Subscript* - [] και {} μέσω αναφοράς.
- `ref` - σε τι είδος πράγμα δείχνει αυτή.
- `bless` - μετατροπή αναφοράς σε αντικείμενο.
- `Scalar::Util` - `weaken`, `isweak`, `reftype`, `blessed`, `looks_like_number`.
- *References tutorial* - βήμα-προς-βήμα οδηγός-συνοδός αυτής της συνοπτικής αναφοράς.
- *Weak refs tutorial* - το πρόβλημα των κύκλων σε βάθος.

Typeglobs

Ένα typeglob είναι ένα ενιαίο όνομα που κατέχει *κάθε* τύπο μεταβλητής με αυτό το όνομα σε ένα πακέτο - το βαθμωτό $foo, ο πίνακας @foo, το hash %foo, η υπορουτίνα &foo, το filehandle foo και η μορφοποίηση foo μοιράζονται μία εγγραφή πίνακα συμβόλων, και η *foo είναι ο τρόπος αναφοράς σε αυτή την εγγραφή ως τιμή. Το sigil είναι *, σκόπιμα επιλεγμένο επειδή ο αστερίσκος διαβάζεται ως «όλα».

```
*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
```

Στη σύγχρονη Perl, τα typeglobs εμφανίζονται σπάνια. Οι αναφορές καλύπτουν τα περισσότερα από αυτά για τα οποία παλιά ήταν απαραίτητα. Οι εναπομείνουσες νόμιμες χρήσεις είναι η ψευδωνυμοποίηση πακέτων, η δημιουργία filehandle σε παλαιότερα ιδιώματα, και μερικά hooks ενδοσκόπησης.

Τι κάνει ένα typeglob

Η πιο χρήσιμη μεμονωμένη πρόταση για τα typeglobs:

> Η ανάθεση σε typeglob κάνει ένα όνομα να σημαίνει ένα άλλο όνομα.

```
*foo = *bar;        # $foo, @foo, %foo, &foo, FH foo all alias $bar,␣
↪@bar, %bar, &bar, FH bar
```

Μετά από αυτή την ανάθεση, η τροποποίηση της `$foo` τροποποιεί τη `$bar`, η κλήση `foo()` καλεί τη `bar()`, και ούτω καθεξής για κάθε τύπο. Τα δύο ονόματα είναι πλέον πραγματικά η ίδια αποθήκη - όχι αντίγραφο.

Η ανάθεση μιας *αναφοράς* σε typeglob δημιουργεί ψευδώνυμο μόνο για αυτόν τον ένα τύπο:

```
our @items;
*shortcut = \@items;     # only @shortcut aliases @items; $shortcut␣
↪and %shortcut unchanged
```

Με αυτόν τον τρόπο το άρθρωμα `Exporter` εγκαθιστά εισαγόμενες υπορουτίνες στο πακέτο του καλούντος: `*caller_package::name = \&exporter_package::name`. Κάθε εισαγόμενη συνάρτηση αποκτά μία υποδοχή ενός glob ανατεθειμένη, ώστε οι υπόλοιπες μεταβλητές του πακέτου προορισμού με αυτό το όνομα να μένουν ανέγγιχτες.

`*name = \&sub` - το τυπικό ιδίωμα εισαγωγής

```
package My::Util;
sub greet { "Hello, $_[0]!" }

package main;
*greet = \&My::Util::greet;       # import greet into main::
print greet("world");              # "Hello, world!"
```

Αυτός είναι ο μηχανισμός που η `use` ενεργοποιεί μέσω `Exporter::import` - το πακέτο εισαγωγής λαμβάνει ανάθεση glob για κάθε ζητούμενο όνομα. Η `Exporter` το κάνει· μπορείτε να το κάνετε χειροκίνητα για περιπτώσεις που η `Exporter` δεν ταιριάζει.

Τοπικά ψευδώνυμα glob

Η `local *name = \$thing` είναι ένα προσωρινό ψευδώνυμο που αυτο-αποκαθίσταται όταν η περικλείουσα εμβέλεια τελειώσει:

```
our $config = "default";

sub run_with_config {
    my ($alt_config, $code) = @_;
    local *config = \$alt_config;        # only inside this sub
    $code->();                            # any read of $config␣
↪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
```

Αυτός είναι ένας βαρύς μηχανισμός που έχει σε μεγάλο βαθμό αντικατασταθεί από απλό `local $config = $alt_config;` για βαθμωτά και ρητή μεταβίβαση παραμέτρων για

όλα τα υπόλοιπα. Η περίπτωση όπου εξακολουθεί να έχει σημασία είναι όταν χρειάζεται να κάνετε ψευδώνυμο ένα *filehandle* κατά μήκος κλήσης αυθαίρετου κώδικα χρήστη (δείτε παρακάτω).

Typeglobs filehandle

Πριν το ιδίωμα «open my $fh» της Perl 5.6, τα filehandles έπρεπε να ονομάζονται μέσω καθολικών πακέτου ή να μεταβιβάζονται ως 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
```

Το σύγχρονο στυλ προτιμάται παντού στον νέο κώδικα. Η μορφή typeglob-filehandle επιβιώνει σε τρία σημεία:

1. **Τα τυπικά handles** - τα STDIN, STDOUT, STDERR είναι bareword filehandles υποστηριζόμενα από τα typeglobs *STDIN, *STDOUT, *STDERR. Τα βλέπετε σε μηνύματα σφάλματος και κώδικα επαναφοράς:

   ```
   print STDERR "error\n";
   *STDOUT = *STDERR;          # redirect all STDOUT to STDERR for
   ↪the rest of the program
   ```

2. **Επιστροφή νεοδημιουργημένων filehandles** - πριν τα λεξιλογικά filehandles, ο τρόπος επιστροφής νέου handle από υπορουτίνα ήταν local *FH και επιστροφή *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
   }
   ```

   Σύγχρονο ισοδύναμο:

   ```
   sub new_handle {
       my $path = shift;
       open my $fh, $path or return;
   ```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    return $fh;
}
```

3. **Επιλογή διαφορετικού προεπιλεγμένου handle εξόδου** - η `select` επιστρέφει και δέχεται filehandle, σε μορφή glob-ή-συμβολοσειράς:

```
my $old = select(STDERR);    # subsequent print without
↪explicit fh writes to STDERR
print "errors-only output\n";
select($old);                # restore
```

*foo{THING} - οι μέθοδοι προσπέλασης υποδοχής

Η `*foo{THING}` επιστρέφει αναφορά σε μία υποδοχή του glob, ή `undef` αν η υποδοχή είναι κενή:

```
*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
```

Η μορφή `*foo{IO}` είναι το ένα σημείο όπου βλέπετε τακτικά σύνταξη `THING` σε σύγχρονο κώδικα. Είναι ο τρόπος που ρωτάτε «αυτό το βαθμωτό κρατάει filehandle;»:

```
sub is_handle {
    my $thing = shift;
    return defined ref($thing) && (
           ref($thing) eq 'GLOB'
        || (ref(\$thing) eq 'GLOB' && defined *$thing{IO})
    );
}
```

Ο τυπικός βοηθός ενδοσκόπησης που τυλίγει αυτό είναι η `Scalar::Util::openhandle`.

Η `*foo{SCALAR}` έχει μια ιδιορρυθμία: επιστρέφει πάντα *κάτι* - η υποδοχή `$foo` του πακέτου δημιουργείται κατ" απαίτηση. Οι υπόλοιπες μέθοδοι προσπέλασης επιστρέφουν `undef` όταν η υποδοχή είναι κενή.

Τι αντικαθιστά τα typeglobs

Για τα μοτίβα όπου τα typeglobs ήταν η μόνη απάντηση, οι σύγχρονες εναλλακτικές είναι:

Παλαιό μοτίβο typeglob	Σύγχρονη αντικατάσταση
Μεταβίβαση filehandle σε υπορουτίνα	Λεξιλογικό FH: `open my $fh, ...`
Επιστροφή filehandle από υπορουτίνα	Λεξιλογικό FH: `return $fh;`
Ψευδώνυμο `@there` ως `@here`	`our @here; *here = \@there;` (εξακολουθεί typeglob, χωρίς αντικατάσταση)
Μεταβίβαση πίνακα με αναφορά	`\@arr` - μεταβίβαση αναφοράς
Εγκατάσταση εισαγόμενης υπορουτίνας	`*name = \&Source::name` (εξακολουθεί typeglob) - ή χρήση `Exporter`
Τοπικοποιημένο καθολικό ψευδώνυμο	`local $var = ...` για απλά βαθμωτά

Το μοτίβο ψευδωνυμοποίησης πακέτου `*here = \@there` είναι αυτό που τα typeglobs εξακολουθούν να κατέχουν ολοκληρωτικά - δεν υπάρχει μηχανισμός «ψευδώνυμο μιας μεταβλητής πακέτου σε μια άλλη» που να μην περνάει από glob. Μέσα σε αρθρώματα που χρησιμοποιούν `Exporter` αυτό γίνεται ως επί το πλείστον αυτόματα.

Δείτε επίσης

- *References* - η σύγχρονη εναλλακτική στα globs για τα περισσότερα μοτίβα «έμμεσης πρόσβασης».
- *Scalars* - η ιστορία της δυναμικής τυποποίησης. Τα globs είναι ξεχωριστός τύπος: η `ref \*FH` επιστρέφει `'GLOB'`.
- `local` - ο ασφαλής τρόπος εγκατάστασης προσωρινού ψευδωνύμου glob.
- `select` - η συνάρτηση επιλεγμένου handle εξόδου που δέχεται ονόματα handle σε μορφή glob ή συμβολοσειράς.
- `Scalar::Util` - `openhandle`, `reftype`, `blessed` για ενδοσκόπηση χρόνου εκτέλεσης.
- `Symbol` - δημιουργία νέων ανώνυμων globs και πλήρης κατονομασία ονομάτων πακέτου. (Δεν υπάρχει ακόμα τοπική σελίδα αναφοράς.)

Παρεμβολή

Οι συμβολοσειρές σε διπλά εισαγωγικά, τα here-docs, τα μοτίβα regex, η αντικατάσταση `s/.../.../` και η `qq//` κάνουν όλα *παρεμβολή*: αναπτύσσουν μεταβλητές και ορισμένες ακολουθίες διαφυγής επιτόπου. Οι συμβολοσειρές σε μονά εισαγωγικά, η `q//` και το αριστερό μοτίβο της `tr/.../.../` δεν το κάνουν. Η γνώση ποιες κατασκευές κάνουν παρεμβολή και ακριβώς ποιες μορφές αναπτύσσονται είναι μία από τις πιο χρήσιμες γνώσεις Perl.

```
my $name = "Alice";
my @nums = (1, 2, 3);

"hello, $name"              # "hello, Alice"
'hello, $name'              # 'hello, $name' - single quotes don't␣
↪interpolate
"@nums in a string"         # "1 2 3 in a string"
"\$name is $name"           # "$name is Alice"
```

Τι παρεμβάλλεται

Μέσα σε "" (και τα άλλα περιβάλλοντα τύπου διπλών εισαγωγικών), η Perl αναπτύσσει ακριβώς δύο μορφές μεταβλητής:

```
$name               # scalar - the simple form
@list               # array  - joined by $" (one space by default)
```

Επιπλέον οι μορφές subscript με αγκύλες και άγκιστρα που συνθέτονται πάνω σε αυτές:

```
$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
```

Σημειώστε ότι **ολόκληρα hashes δεν παρεμβάλλονται**:

```
"hash is %h"        # "hash is %h" - literal '%' is fine, '%h' is NOT␣
↪magical
```

Αν θέλετε ένα hash εκτυπωμένο σε συμβολοσειρά, μορφοποιήστε το ρητά με `Data::Dumper` ή χειροκίνητο `join`.

Τι *δεν* παρεμβάλλεται

```
&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)
```

Ο πρακτικός κανόνας: **οι μεταβλητές και τα subscripts τους παρεμβάλλονται· οι κλήσεις συναρτήσεων/μεθόδων και οι αριθμητικές πράξεις πάνω σε αυτές δεν παρεμβάλλονται.** Όταν θέλετε παρεμβολή μιας έκφρασης, δείτε τις κατασκευές `@{[ ... ]}` και `${\ ... }` παρακάτω.

Οι αγκύλες ${name}

Οι αγκύλες γύρω από ένα όνομα μεταβλητής αποσαφηνίζουν πού τελειώνει το όνομα:

```
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
```

Χωρίς τις αγκύλες, η Perl είναι άπληστη: η $whospeak διαβάζεται ως ένα αναγνωριστικό, η $who : : sub διαβάζεται ως πλήρως κατονομασμένο όνομα. Οι αγκύλες αναγκάζουν τον αναλυτή να σταματήσει στο }.

Αυτή δεν είναι η ίδια μορφή αγκυλών με ένα subscript hash:

```
"$h{key}"              # hash element                  - what most␣
↪people mean
"${h}{key}"            # hash element in a different parse path -␣
↪same result
"${h{key}}"            # also the same
"${\(\$h{key})}"        # the absurd "interpolate an expression"␣
↪form
```

Οι αγκύλες ${name} χρησιμοποιούνται κυρίως στην περίπτωση αποσαφήνισης (όνομα ακολουθούμενο από αλφαριθμητικούς χαρακτήρες που αλλιώς θα καταναλώνονταν), όχι για πρόσβαση σε hash.

Οι μορφές αποαναφοράς με αγκύλες @{...} και ${...}

Όταν θέλετε να *αποαναφέρετε* κάτι μέσα σε συμβολοσειρά, η μορφή με αγκύλες επιτρέπει η έκφραση αποαναφοράς να είναι αυθαίρετη:

```
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
```

Η μορφή @{ EXPR } αποαναφέρει την EXPR ως πίνακα. Οι αγκύλες είναι **απαραίτητες** επειδή ο στόχος αποαναφοράς είναι μια έκφραση, όχι μια απλή μεταβλητή. Η @$config->{users} δεν αναλύεται όπως εννοείται

- @$config would deref $config as an array, and ->{users} is a hash-element access on what's left.

Το ίδιο ισχύει για ${ EXPR } (βαθμωτή αποαναφορά) και %{ EXPR } (αποαναφορά hash, που επιστρέφει επίπεδη λίστα, σπάνια χρήσιμη στην παρεμβολή).

`@{[ expr ]}` - παρεμβολή οποιασδήποτε έκφρασης λίστας

Για παρεμβολή του αποτελέσματος μιας αυθαίρετης έκφρασης, τυλίξτε την σε `@{[ ... ]}`:

```
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
```

Ο μηχανισμός: η `[ ... ]` κατασκευάζει μια ανώνυμη αναφορά πίνακα με περιεχόμενα την τιμή της έκφρασης σε περιβάλλον λίστας· η `@{ ... }` στη συνέχεια αποαναφέρει αυτή την αναφορά πίνακα σε λίστα, η οποία παρεμβάλλεται όπως μια λίστα (ενωμένη με `$"`).

Αυτός είναι ο κανονικός τρόπος αντιμετώπισης του «θέλω κλήση συνάρτησης μέσα σε συμβολοσειρά με διπλά εισαγωγικά». Η εναλλακτική είναι η συνένωση:

```
print "hello, " . uc($name) . "!\n";             # equivalent
```

Η μορφή `@{[ ... ]}` είναι πιο πυκνή όταν υπάρχουν πολλαπλές παρεμβολές· η συνένωση είναι πιο σαφής όταν υπάρχει μόνο μία. Σε πρότυπα με πολλά πεδία, η `printf` ή η `sprintf` είναι συνήθως ακόμα πιο σαφής:

```
printf "found %d matches\n", scalar @hits;
```

Χρησιμοποιήστε `@{[ ... ]}` όταν θέλετε πραγματικά μια έκφραση Perl στη μέση μιας συμβολοσειράς και οι εναλλακτικές είναι πιο άσχημες.

`${\ expr }` - παρεμβολή βαθμωτής έκφρασης

Το βαθμωτό αντίστοιχο της `@{[ ... ]}`:

```
print "answer: ${\ 6 * 7 }\n";                  # "answer: 42"
print "ts: ${\ scalar localtime }\n";            # "ts: Mon Apr 27␣
↪..."
```

Η `\ EXPR` δημιουργεί αναφορά στη βαθμωτή τιμή της `EXPR`· η `${ ... }` την αποαναφέρει. Το αποτέλεσμα είναι μία τιμή, παρεμβαλλόμενη ως βαθμωτό.

Στις περισσότερες περιπτώσεις η μορφή `@{[ ... ]}` είναι πιο ομοιόμορφη - λειτουργεί τόσο σε βαθμωτό όσο και σε περιβάλλον λίστας. Η `${\ ... }` είναι περιστασιακά πιο σαφής όταν θέλετε συγκεκριμένα βαθμωτό περιβάλλον:

```
"got @{[ items() ]} items"      # items() called in LIST context
"got ${\ items() } items"        # items() called in SCALAR context
```

Αμφισημία subscript hash σε μοτίβα

Μέσα σε μοτίβο regex (που είναι τύπου διπλών εισαγωγικών), ο αναλυτής πρέπει να μαντέψει αν η `$foo[bar]` σημαίνει «παρεμβολή `$foo[bar]` (ένα στοιχείο της `@foo`)» ή «παρεμβολή `$foo` και στη συνέχεια αντιστοίχιση κλάσης χαρακτήρων `[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
```

Η αποσαφήνιση γίνεται με τη μορφή αγκυλών:

```
$x =~ /${foo}[abc]/                # $foo + character class [abc]
$x =~ /$foo[5]/                    # element 5 of @foo
$x =~ /\Q$foo\E[abc]/              # $foo (literal) + character class
```

Ο `\Q...\E` απενεργοποιεί την ερμηνεία μετα-χαρακτήρων εντός της εμβέλειάς του, κάτι που είναι η πιο καθαρή λύση όταν η μεταβλητή είναι είσοδος χρήστη.

Heredocs

Τα heredocs είναι τύπου διπλών εισαγωγικών εκτός αν ο δείκτης είναι σε μονά εισαγωγικά:

```
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
```

Τα εσοχοποιημένα heredocs (Perl 5.26+) αγνοούν τα αρχικά κενά μέχρι τη μικρότερη ακολουθία σε οποιαδήποτε γραμμή:

```
my $msg = <<~EOM;
    Hello, $name!
    EOM
# "Hello, Alice!\n" - leading "    " stripped from every line
```

Η μορφή `~` προτιμάται σε σύγχρονο κώδικα· η μη εσοχοποιημένη μορφή είναι άσχημη μέσα σε βρόχους και σώματα μεθόδων.

Πραγματικό παράδειγμα: κατασκευή HTML με ασφάλεια

```
my @items = ('Apples', 'Bananas', 'Cherries');

my $html = <<~HTML;
    <ul>
    @{[ map { "  <li>$_</li>" } @items ]}
    </ul>
    HTML
```

Η `map` επιστρέφει μια λίστα συμβολοσειρών `<li>...</li>`· η `@{[ ... ]}` τις παρεμβάλλει· η σιωπηρή `$"` είναι ένα κενό, δίνοντας:

```
<ul>
  <li>Apples</li>   <li>Bananas</li>   <li>Cherries</li>
</ul>
```

Για αλλαγές γραμμής μεταξύ στοιχείων, ορίστε τοπικά τη `$"`:

```
my $html = do {
    local $" = "\n";
    <<~HTML;
        <ul>
        @{[ map { "  <li>$_</li>" } @items ]}
        </ul>
        HTML
};
```

Ή ενώστε ρητά:

```
my $li = join "\n", map { "  <li>$_</li>" } @items;
my $html = "<ul>\n$li\n</ul>\n";
```

Η ρητή `join` είναι συνήθως πιο σαφής για μη τετριμμένα πρότυπα. Η μορφή `@{[ ... ]}` κερδίζει τη θέση της όταν το πρότυπο είναι κυρίως κυριολεκτικό κείμενο με έναν ή δύο ενσωματωμένους υπολογισμούς.

Τι ελέγχει η ειδική μεταβλητή $"

Ο διαχωριστής λίστας στην παρεμβολή:

```
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"
```

Η `$"` είναι ο κανονικός τρόπος ελέγχου του πώς παρεμβάλλονται οι πίνακες χωρίς

αναγραφή της συμβολοσειράς. Δείτε *filehandle and printing state* για την οικογένεια μεταβλητών διαχωριστή.

Δείτε επίσης

- *Scalars*, *Arrays*, *Hashes* - οι μορφές που αναπτύσσει η παρεμβολή.
- *References* - η μορφή αποαναφοράς `@{ EXPR }` και τι κάνει η `\` στην `${\ ... }`.
- `$"` - ο διαχωριστής λίστας για παρεμβολή πίνακα.
- `$,`, `$\`, `$/` - οι σχετικές μεταβλητές διαχωριστή για `print` και είσοδο.
- `sprintf`, `printf`
  - η ρητή εναλλακτική συμβολοσειράς μορφής όταν το πρότυπο είναι κυρίως σημεία στίξης.
- `qq`, `q`, `qw` - οι τελεστές τύπου εισαγωγικών με προσαρμοσμένους οριοθέτες.
- *Scalars* - `$x`. Αριθμοί, συμβολοσειρές, αναφορές· `undef`· αλήθεια και ορισμένη τιμή· η διπλή αριθμητική/συμβολοσειριακή φύση των βαθμωτών.
- *Arrays* - `@arr`. Πρόσβαση στοιχείων, `$#arr`, αρνητικοί δείκτες, τομές, ανάθεση λίστας, ο κανόνας sigil `@arr` / `$arr[i]` / `@arr[i,j]`.
- *Hashes* - `%h`. Ζεύγη κλειδιού-τιμής, οι τέσσερις παραλλαγές subscript, `keys`/`values`/`each`, σειρά κλειδιών, τομές hash και τομές κλειδιού/τιμής.
- *Context* - λίστα, βαθμωτό, κενό, λογικό. Πώς κάθε τελεστής και ανάθεση επιβάλλει περιβάλλον, τι αναφέρει η `wantarray`, ο τελεστής `scalar()` και οι επαναλαμβανόμενες παγίδες με το `,` έναντι κατασκευής λίστας.
- *Numbers and strings* - αυτόματη μετατροπή και προς τις δύο κατευθύνσεις, ακέραιος έναντι κινητής υποδιαστολής, `Inf` και `NaN`, μορφές αριθμητικών κυριολεκτικών, «0 but true», `dualvar`, παγίδες τοπικών ρυθμίσεων και ακρίβειας.
- *References* - `\X`, τα επτά πράγματα στα οποία μπορείτε να κάνετε αναφορά, οι τέσσερις τρόποι αποαναφοράς, αυτο-δημιουργία, ασθενείς αναφορές, κυκλικές δομές, `ref` και `Scalar::Util`.
- *Typeglobs* - `*name`. Η εγγραφή πίνακα συμβόλων, ψευδωνυμοποίηση, `*foo{THING}`, πότε η σύγχρονη Perl τα χρειάζεται ακόμα και πότε οι αναφορές είναι η σωστή εναλλακτική.
- *Interpolation* - τι παρεμβάλλεται μέσα σε συμβολοσειρές με διπλά εισαγωγικά, here-docs και μοτίβα regex· οι αγκύλες `${name}`· οι περίπλοκες κατασκευές `@{[ expr ]}` και `${\ expr }`· αμφισημίες με `[` και `{` μετά από sigil.

Διατομεακές έννοιες

Μια χούφτα ιδεών εμφανίζεται σε κάθε σελίδα παρακάτω. Διαβάστε τις μία φορά και τα υπόλοιπα θα σας φανούν οικεία:

- **Τα sigils σημειώνουν το *αποτέλεσμα*, όχι τον περιέκτη.** Το `@arr[1, 3]` δεν σημαίνει «δύο πίνακες»· το `@` λέει «το αποτέλεσμα είναι μια λίστα». Η μεταβλητή με όνομα `arr` είναι η ίδια στα `@arr`, `$arr[0]`, `@arr[0..3]` και `%arr[0..3]`.

- **Το περιβάλλον εξαναγκάζει.** Ένα βαθμωτό που αποτιμάται σε περιβάλλον λίστας γίνεται λίστα ενός στοιχείου· μια λίστα που αποτιμάται σε βαθμωτό περιβάλλον δίνει το **τελευταίο** στοιχείο της (τελεστής κόμματος), αλλά ένας *πίνακας* σε βαθμωτό περιβάλλον δίνει το **μήκος** του. Οι δύο περιπτώσεις δεν είναι ίδιες· δείτε *context*.
- **Οι αναφορές είναι βαθμωτά.** Μια αναφορά σε οτιδήποτε χωράει σε μια βαθμωτή μεταβλητή, κάτι που καθιστά δυνατές τις εμφωλευμένες δομές δεδομένων. Η σύνταξη αποαναφοράς (`@$ref`, `$ref->[0]`, `${$ref}{key}`) είναι η αντίστροφη της `\`.
- **Η undef είναι τιμή, όχι απουσία τιμής.** Ένα βαθμωτό μπορεί να περιέχει την τιμή `undef`· ο τρόπος ελέγχου είναι η `defined $x`. Ένας πίνακας μπορεί να έχει στοιχεία `undef` χωρίς να είναι κενός.
- **Συμβολοσειρές και αριθμοί μετατρέπονται σιωπηλά.** Το `"42" + 1` δίνει 43, το `42 . "x"` δίνει `"42x"`. Η μετατροπή είναι πάντα ορισμένη· δείτε *numbers and strings* για το τι θεωρείται «μοιάζει με αριθμό» και πού βρίσκονται οι γωνιακές περιπτώσεις.

Δείτε επίσης

- `perlop` - οι τελεστές που διαβάζουν και παράγουν αυτές τις τιμές (αριθμητικοί, σύγκρισης, subscript, εύρους, βέλους, ανάθεσης).
- `perlvar` - οι ειδικές μεταβλητές που η Perl γεμίζει με κατάσταση από την τελευταία λειτουργία, συμπεριλαμβανομένων των `$_`, `@_`, των μεταβλητών αντιστοιχίας και των `$/`, `$\`, `$,`, `$"` που διαμορφώνουν το I/O.
- `perlfunc` - οι ενσωματωμένες συναρτήσεις που λειτουργούν πάνω σε αυτούς τους τύπους: `ref`, `scalar`, `wantarray`, `defined`, `keys`, `values`, `each`, `exists`, `delete`, `bless`.
- *References tutorial* - βήμα-προς-βήμα εισαγωγή στις μορφές αναφορών που καλύπτονται συνοπτικά στο *references*.

5.1.5 Υπορουτίνες

Μια υπορουτίνα είναι ένα κατονομασμένο (ή ανώνυμο) μπλοκ κώδικα που δέχεται μια λίστα ορισμάτων, εκτελείται σε μια νέα λεξιλογική εμβέλεια, και επιστρέφει μια λίστα τιμών. Οι υπορουτίνες είναι η μονάδα επαναχρησιμοποίησης στην Perl, η μονάδα αποστολής για αντικειμενοστραφή κώδικα, και - μέσω closures

- η μονάδα ενθυλάκωσης με κατάσταση. Σχεδόν τα πάντα στη γλώσσα είναι χτισμένα πάνω τους.

PetaPerl implements the full Perl 5.44 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`).

Αυτή η αναφορά είναι χωρισμένη σε μία σελίδα ανά θέμα. Ο κανόνας ψευδωνυμοποίησης του `@_` βρίσκεται στα ορίσματα, η παγίδα του closure-πάνω-σε-μεταβλητή-βρόχου βρίσκεται στην εμβέλεια, οι επιδράσεις των πρωτοτύπων σε επίπεδο αναλυτή βρίσκονται

στα πρωτότυπα - ώστε κάθε σελίδα να μπορεί να διαβαστεί και να επισκεφθεί ξανά χωρίς ευρύτερο πλαίσιο.

Επιλέξτε θέμα

Δήλωση

Κάθε sub είναι ένα από τρία πράγματα: μια **κατονομασμένη** sub δεσμευμένη σε πακέτο, μια **ανώνυμη** sub αποθηκευμένη σε αναφορά κώδικα, ή μια **λεξιλογική** sub δεσμευμένη σε μια θέση τύπου my στην τρέχουσα εμβέλεια. Η μορφή δήλωσης επιλέγει μία· όλα τα υπόλοιπα (πρωτότυπο, υπογραφή, γνωρίσματα, σώμα) τη διακοσμούν.

Κατονομασμένες subs

```
sub greet {
    my ($name) = @_;
    return "Hello, $name!";
}

greet("Alice");                 # "Hello, Alice!"
```

Το όνομα greet δεσμεύεται στον πίνακα συμβόλων του τρέχοντος πακέτου κάτω από την υποδοχή ampersand & - την ίδια υποδοχή που αναζητά ο χρόνος εκτέλεσης όταν καλείτε greet(...) ή \&greet. Οι κατονομασμένες subs είναι ορατές σε κάθε κώδικα που εκτελείται μετά τη μεταγλώττιση της δήλωσης sub· στην πράξη «μεταγλωττίστηκε» σημαίνει «το αρχείο αναλύθηκε μέχρι τη δήλωση», κάτι που σχεδόν πάντα είναι αυτό που θέλετε.

Μια κατονομασμένη sub μπορεί επίσης να δηλωθεί σε δύο μέρη - μια προδήλωση και ένας ορισμός:

```
sub later;                      # promise: a sub named 'later' will␣
↪exist
sub later {                     # fulfilment
    return 42;
}
```

Η προδήλωση είναι κυρίως χρήσιμη όταν τα πρωτότυπα πρέπει να τεθούν σε ισχύ για αναδρομικές κλήσεις (το πρωτότυπο πρέπει να είναι ορατό *πριν* αναλυθεί η αναδρομική κλήση) ή όταν θέλετε να καταστείλετε τις προειδοποιήσεις «Bareword not allowed» για κώδικα που καλεί τη sub πριν τον ορισμό της.

Κλήση sub με βάση το όνομα

```
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 @_
```

Η μορφή σκέτου & (`&greet` χωρίς παρενθέσεις) είναι ένα κατάλοιπο από τη perl4 με μία εναπομείνασα νόμιμη χρήση: πέρασμα του τρέχοντος `@_` κατευθείαν χωρίς αντιγραφή. Δείτε *αναδρομή* για το `goto &greet`, την παραλλαγή κλήσης ουράς.

Ανώνυμες subs

```
my $double = sub {
    my ($x) = @_;
    return $x * 2;
};

$double->(7);                    # 14
```

Η έκφραση `sub { ... }` αποτιμάται σε αναφορά κώδικα. Αποτυπώνει οποιεσδήποτε λεξιλογικές μεταβλητές (`my`/`state`) από την περικλείουσα εμβέλεια - δείτε *εμβέλεια* - και καλείται μέσω `->()` ή με τις παλαιές μορφές `&$double()` / `&{$double}()`.

Οι ανώνυμες subs είναι ο τρόπος λειτουργίας κάθε callback στην Perl:

```
my @sorted = sort { $a <=> $b } @list;
my @big    = grep { $_ > 100   } @list;
my @str    = map  { "<$_>"     } @list;
```

Είναι επίσης το δομικό στοιχείο για closures, πίνακες αποστολής, και αντικείμενα mock. Ένας πίνακας αποστολής:

```
my %op = (
    add => sub { $_[0] + $_[1] },
    sub => sub { $_[0] - $_[1] },
    mul => sub { $_[0] * $_[1] },
);

my $result = $op{$cmd}->($x, $y);
```

Αναφορές κώδικα

Μια αναφορά κώδικα είναι ένα βαθμωτό πρώτης τάξης που κρατά μια sub. Αποκτάτε μία από `sub { ... }`, από `\&named_sub`, ή από αναζήτηση μεθόδου με `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
```

Μέσα σε μια αναφορά κώδικα, η sub διατηρεί την ταυτότητά της: η `\&greet` είναι πάντα η ίδια αναφορά για όλη τη διάρκεια ζωής της `&greet` του πακέτου. Οι ανώνυμες subs είναι διαφορετικές αναφορές κάθε φορά που αποτιμάται η έκφραση `sub { ... }` - αυτό κάνει τα εργοστάσια closures να λειτουργούν.

Λεξιλογικές subs

Μια λεξιλογική sub δηλώνεται με `my sub` (και τα αντίστοιχα `state sub` και `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)
```

Η `my sub` είναι πιο χρήσιμη μέσα σε μεγάλες συναρτήσεις όπου θέλετε ένα ιδιωτικό βοηθητικό χωρίς να μολύνετε τον χώρο ονομάτων του πακέτου. Δεν χρησιμοποιείται ευρέως στην πράξη, αλλά είναι το κατάλληλο εργαλείο όταν θέλετε πραγματική ενθυλάκωση. Δείτε *εμβέλεια* για τις παραλλαγές `state` και `our`.

Προδήλωση

Μια προδήλωση είναι μια δήλωση `sub NAME;` (χωρίς σώμα, χωρίς αγκύλες). Τρεις λόγοι για να γράψετε μία:

```
# 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
```

Σε σύγχρονο κώδικα οι λόγοι (1) και (3) είναι κυρίως υφολογικοί. Ο λόγος (2) είναι πραγματική απαίτηση όταν χρησιμοποιείτε πρωτότυπα· δείτε *πρωτότυπα*.

Τι δεν μπορείτε να κάνετε

- Δεν μπορείτε να ξαναορίσετε μια ενσωματωμένη με `sub print { ... }` και να κάνετε την Perl σιωπηρά να χρησιμοποιεί τη δική σας. Η υπερκάλυψη ενσωματωμένων γίνεται μέσω `CORE::GLOBAL::print` και τεκμηριώνεται στα *γνωρίσματα* και τη σελίδα λεπτομερειών `prototype`.
- Δεν μπορείτε να έχετε δύο κατονομασμένες subs με το ίδιο όνομα στο ίδιο πακέτο χωρίς προειδοποίηση «Subroutine NAME redefined». Είτε μετονομάστε είτε

σκοπεύετε πραγματικά τον επαναορισμό (και αποσιωπήστε την προειδοποίηση με `no warnings 'redefine';`).

- Δεν μπορείτε να δώσετε μια προδήλωση και ένα σώμα με *διαφορετικά* πρωτότυπα. Οι δύο δηλώσεις πρέπει να συμφωνούν.

Δείτε επίσης

- *Ορίσματα και @_* - τι βλέπει το σώμα μόλις γίνει η κλήση.
- *Εμβέλεια* - τι κάνουν τα `my`, `our`, `local`, και `state` στις μεταβλητές μέσα στο σώμα.
- `sub` - η σελίδα perlfunc της δεσμευμένης λέξης.
- `prototype` - ενδοσκόπηση χρόνου εκτέλεσης του πρωτοτύπου μιας sub, και η γέφυρα για υπερκάλυψη ενσωματωμένων.
- *Τελεστής βέλους* - επίκληση αναφορών κώδικα μέσω `->()`.

Ορίσματα και @_

Όταν μια sub καλείται με τον κλασικό τρόπο (χωρίς υπογραφή), η Perl ισοπεδώνει ολόκληρη τη λίστα ορισμάτων σε έναν ενιαίο πίνακα και τον εκθέτει μέσα στο σώμα ως `@_`. Τα στοιχεία αυτού του πίνακα δεν είναι αντίγραφα - είναι **ψευδώνυμα** προς τις εκφράσεις του καλούντος. Αυτός ο ενιαίος κανόνας είναι η πηγή των περισσότερων εκπλήξεων της σύμβασης κλήσης της Perl και του μεγαλύτερου μέρους της ισχύος της.

Ο κανόνας ψευδωνυμοποίησης

```
sub bump {
    $_[0]++;                        # mutates the caller's variable
}

my $x = 10;
bump($x);
say $x;                             # 11
```

Η τροποποίηση του `$_[0]` γράφει μέσω του ψευδωνύμου σε ό,τι πέρασε ο καλών. Αυτό γίνεται σκοπίμως - είναι ο τρόπος με τον οποίο η `chomp(@lines)` αφαιρεί newlines από κάθε στοιχείο του `@lines` αντί από αντίγραφο του `@lines`.

Η ψευδωνυμοποίηση επεκτείνεται και στα κυριολεκτικά, με προβλέψιμες συνέπειες:

```
bump(10);                           # error: Modification of a read-
 ↪only
                                    # value attempted
```

Ένα κυριολεκτικό `10` είναι σταθερά. Το ψευδώνυμο δείχνει σε αυτή· η απόπειρα τροποποίησης αποτυγχάνει με die.

Το ιδίωμα αποσυσκευασίας

Επειδή η ψευδωνυμοποίηση σπάνια είναι αυτό που θέλει το σώμα, σχεδόν κάθε sub κλασικού τύπου ξεκινά με ένα από αυτά τα δύο μοτίβα:

```
# 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
    ...
}
```

Μόλις αποσυσκευαστούν, το σώμα δουλεύει με ιδιωτικά αντίγραφα και δεν μπορεί να τροποποιήσει κατά λάθος την κατάσταση του καλούντος. Η `shift` χωρίς όρισμα μέσα σε sub χρησιμοποιεί εξ ορισμού το `@_`, γι" αυτό η `my $self = shift;` υπάρχει παντού στην OO Perl.

Νόμιμες χρήσεις ψευδωνυμοποίησης

Η ψευδωνυμοποίηση είναι το σωστό εργαλείο όταν πραγματικά θέλετε να τροποποιήσετε την κατάσταση του καλούντος:

```
# in-place trim
sub trim_inplace {
    for (@_) {
        s/^\s+//;
        s/\s+$//;
    }
}

my @data = ("  hello  ", "  world\n");
trim_inplace(@data);
# @data is now ("hello", "world")
```

Ο βρόχος `for (@_)` ψευδωνυμεί το `$_` σε κάθε στοιχείο του `@_`, το οποίο είναι με τη σειρά του ψευδώνυμο προς κάθε στοιχείο του `@data`. Οι τροποποιήσεις `s///` περνούν μέσα από τα δύο βήματα ψευδωνύμων στις αρχικές συμβολοσειρές. Χωρίς την ψευδωνυμοποίηση, ο ίδιος κώδικας θα ήταν διπλάσια δουλειά.

Ο ίδιος μηχανισμός τροφοδοτεί τα `chomp(@lines)`, `chop(@buf)`, και οποιαδήποτε συνάρτηση «επεξεργασία λίστας και επιτόπια τροποποίηση» γράφετε εσείς. Δηλώστε την πρόθεση στο όνομα της συνάρτησης (`_inplace`, `mutate_`, παρόμοια) ώστε οι καλούντες να μην αιφνιδιαστούν.

Ονομαστικά ορίσματα

Η Perl δεν έχει ενσωματωμένη σύνταξη ονομαστικών παραμέτρων στην κλασική μορφή, αλλά ο => (fat comma) σε συνδυασμό με ανάθεση hash δίνει το ίδιο αποτέλεσμα:

```
sub configure {
    my %opt = @_;               # ('host' => 'localhost', 'port' =>
↪8080)
    $opt{host} //= 'localhost';
    $opt{port} //= 80;
    ...
}

configure( host => 'example.com', port => 8080 );
```

Ο fat comma είναι σημασιολογικά ίδιος με ένα κανονικό κόμμα αλλά περικλείει σε εισαγωγικά το bareword στα αριστερά του, οπότε `host => ...` είναι συντομογραφία για `'host' => ...`. Δείτε `,` για τον ίδιο τον τελεστή.

Για υποχρεωτικά-και-προαιρετικά ονομαστικά ορίσματα, η τυπική μορφή είναι:

```
sub render {
    my ($template, %opt) = @_;   # one positional, rest as named
    $opt{escape} //= 1;
    ...
}

render('greeting.tt', escape => 0, locale => 'de');
```

Για το σύγχρονο ιδίωμα ονομαστικών παραμέτρων βασισμένο σε υπογραφές, δείτε *υπογραφές*.

`shift`, `pop`, `unshift`, `push` στο `@_`

Το `@_` είναι ένας πραγματικός πίνακας. Μπορείτε να κάνετε `shift` από την αρχή, pop από το τέλος, τομή, τροποποίηση. Τίποτα από αυτά δεν αλλάζει την κατάσταση του καλούντος - μόνο η εγγραφή μέσω `$_[N]` το κάνει.

```
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);
}
```

Μια πρακτική συνέπεια: αν θέλετε να *ξανα-πακετάρετε* τα εναπομείναντα ορίσματα για προώθηση σε άλλη sub, απλώς γράψτε `@_` - ακριβώς αυτό κάνει η `goto &other` χωρίς ρητό πέρασμα. Δείτε *αναδρομή* για προώθηση κλήσης ουράς.

wantarray και το περιβάλλον κλήσης

Μέσα σε μια sub, η `wantarray` σας λέει το περιβάλλον στο οποίο έγινε η κλήση:

wantarray	Περιβάλλον	Τυπική μορφή
1 (αληθές)	λίστα	επιστροφή λίστας
0 (ψευδές)	scalar	επιστροφή ενός βαθμωτού
undef	κενό	πρόωρη επιστροφή· το αποτέλεσμα θα απορριφθεί

```
sub items {
    return unless defined wantarray;     # void: nothing to compute
    my @result = compute_items();
    return wantarray ? @result : scalar @result;
}
```

Το `@_` λέει *τι* ήρθε· η `wantarray` λέει *τι αναμένεται πίσω*. Είναι τα δύο μισά της εικόνας της σύμβασης κλήσης. Δείτε *lvalue και περιβάλλον* για πλήρη συζήτηση.

Συνήθεις παγίδες

- **`my ($x) = @_` έναντι `my $x = @_`.** Το πρώτο είναι περιβάλλον λίστας: η `$x` παίρνει το πρώτο όρισμα. Το δεύτερο είναι βαθμωτό περιβάλλον: η `$x` παίρνει τον *αριθμό* ορισμάτων. Οι παρενθέσεις στα αριστερά κάνουν τη διαφορά· τα λάθη εδώ είναι συχνά.

  ```
  sub one_arg {
      my ($x) = @_;              # $x = first arg
  }

  sub count_args {
      my $n = @_;                # $n = number of args
  }
  ```

- **Τροποποίηση `$_[N]` «κατά λάθος».** Συναρτήσεις όπως η `chomp`, `chop`, `tr///`, `s///` λειτουργούν στο όρισμά τους *επιτόπια*. Η `chomp($_[0])` τροποποιεί τη συμβολοσειρά του καλούντος. Αν δεν είναι αυτό που θέλετε, αντιγράψτε πρώτα.

- **Ισοπέδωση πινάκων και hashes.** Πίνακες και hashes που περνιούνται ως ορίσματα χάνουν την ταυτότητά τους στο σημείο κλήσης - γίνονται όλα στοιχεία του ενιαίου επίπεδου `@_`. Για να περάσετε πίνακα χωρίς ισοπέδωση, περάστε αναφορά:

  ```
  takes_array(\@arr);            # one ref argument
  sub takes_array {
      my ($aref) = @_;
      push @$aref, 'extra';
  }
  ```

- **`@_` μετά από `goto &sub`.** Το τρέχον `@_` περνιέται στον στόχο. Ό,τι έχετε ήδη αφαιρέσει με shift δεν υπάρχει πλέον από την πλευρά του στόχου. Αυτό είναι χαρακτηριστικό, όχι σφάλμα - δείτε *αναδρομή*.

Δείτε επίσης

- `@_` - η εγγραφή perlvar της μεταβλητής, με τον κανόνα ψευδωνυμοποίησης και τη συζήτηση περιβάλλοντος βρόχου.
- `shift`, `pop` - χρησιμοποιούν εξ ορισμού το `@_` μέσα σε sub.
- *Υπογραφές* - η σύγχρονη εναλλακτική για χειρισμό παραμέτρων κατά θέση και ονομαστικών.
- *Τιμές επιστροφής* - τι στέλνει πίσω το σώμα.
- *Τελεστής κόμματος* - => για κατασκευή ονομαστικών ορισμάτων.

Τιμές επιστροφής

Μια sub επιστρέφει είτε φτάνοντας στην τελευταία δήλωσή της, είτε συναντώντας ρητό `return`, είτε μέσω `goto` αλλού (δείτε *αναδρομή*). Η λίστα επιστροφής **ισοπεδώνεται** με τον ίδιο τρόπο όπως η λίστα ορισμάτων - πίνακες και hashes χάνουν την ταυτότητά τους εκτός αν επιστρέψετε αναφορές - και στη συνέχεια αποτιμάται στο περιβάλλον του καλούντος.

Σιωπηρή επιστροφή

Αν ο έλεγχος πέσει από το τέλος του σώματος, επιστρέφεται η τιμή της τελευταίας δήλωσης *έκφρασης*:

```
sub square {
    my ($x) = @_;
    $x * $x                     # implicit return
}

say square(5);                  # 25
```

Η σιωπηρή επιστροφή είναι ιδιωματική για σύντομες subs μίας έκφρασης. *Δεν* είναι ιδιωματική για subs πολλαπλών κλάδων, όπου οι αναγνώστες ωφελούνται από τη ρητή σήμανση κάθε εξόδου με `return`.

Αν η τελευταία δήλωση είναι κατασκευή ελέγχου ροής (ένα `for`, `while`, `if` χωρίς `else`, ...), η τιμή επιστροφής είναι απροσδιόριστη. Η κενή sub επιστρέφει κενή λίστα.

Ρητή επιστροφή

```
sub max {
    my ($a, $b) = @_;
    return $a if $a >= $b;
    return $b;
}
```

Η ρητή `return` εξέρχεται αμέσως από τη sub και παρέχει την τιμή επιστροφής. Οι πολλαπλές πρόωρες επιστροφές είναι η τυπική μορφή για κώδικα με φρουρούς:

```
sub fetch {
    my ($id) = @_;
    return       unless defined $id;     # void/empty
    return undef if      $id eq '';      # explicit undef in scalar
    ...
}
```

Σημειώστε τη διαφορά μεταξύ `return;` και `return undef;`:

- Η `return;` επιστρέφει κενή λίστα σε περιβάλλον λίστας, `undef` σε βαθμωτό περιβάλλον, και τίποτα σε κενό περιβάλλον. Είναι ο σωστός ανάλογα με το περιβάλλον τρόπος να πείτε «κανένα αποτέλεσμα».
- Η `return undef;` επιστρέφει τη *λίστα* `(undef)` σε περιβάλλον λίστας, η οποία έχει μήκος 1 - οπότε ένας καλών που γράφει `if (my @r = my_sub()) { ... }` θα δει λίστα ενός στοιχείου και το `if` θα ενεργοποιηθεί. Αυτό είναι σχεδόν πάντα λάθος.

Ο πρακτικός κανόνας: γράψτε `return;` για να δηλώσετε αποτυχία ή απουσία· γράψτε `return undef;` μόνο όταν ο καλών εγγυημένα βρίσκεται σε βαθμωτό περιβάλλον.

Ισοπέδωση λίστας κατά την επιστροφή

Η ίδια ισοπέδωση που εφαρμόζεται στα ορίσματα εφαρμόζεται και στις τιμές επιστροφής:

```
sub two_lists {
    my @a = (1, 2);
    my @b = (3, 4);
    return (@a, @b);            # returns (1, 2, 3, 4) - one flat␣
↪list
}

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 = ()
```

Η τελευταία γραμμή είναι η κλασική έκπληξη: ο `@p` απορροφά άπληστα ολόκληρη τη λίστα επειδή η ανάθεση λίστας σε επίπεδο πίνακα στην αριστερή πλευρά δεν έχει τρόπο να γνωρίζει πού τελειώνει μια λογική ομάδα και πού αρχίζει η επόμενη. Για να επιστρέψετε δύο πίνακες ξεχωριστά, επιστρέψτε αναφορές:

```
sub two_lists {
    return (\@a, \@b);
}

my ($pref, $qref) = two_lists();
```

Επιστροφή ευαίσθητη στο περιβάλλον

```
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
```

Ο κλάδος «πρόωρη επιστροφή σε κενό περιβάλλον» είναι ένα πραγματικό μοτίβο βελτιστοποίησης - ακριβοί υπολογισμοί δεν πρέπει να εκτελούνται όταν ο καλών πετάει το αποτέλεσμα.

Για πλήρη συζήτηση σχετικά με τη `wantarray` και πώς συνδυάζεται με `:lvalue`, δείτε *lvalue και περιβάλλον*.

Επιστροφή hashes και πινάκων

Η επιστροφή ενός `@array` ή `%hash` απευθείας λειτουργεί μόνο λόγω ισοπέδωσης:

```
sub config {
    return (host => 'localhost', port => 80);
}

my %c = config();                # %c is the hash
my @c = config();                # @c = ('host', 'localhost', 'port
 ↪', 80)
```

Για μεγαλύτερες δομές, επιστρέψτε αναφορές:

```
sub config {
    my %c = (host => 'localhost', port => 80);
    return \%c;                  # one ref, no flattening
}

my $c = config();
say $c->{host};
```

Οι αναφορές είναι επίσης ο μόνος τρόπος να επιστρέψετε *πολλαπλά* συγκεντρωτικά χωρίς ο καλών να χρειάζεται να γνωρίζει τα μεγέθη τους.

`return` μέσα σε `eval`

Η `return` μέσα σε μπλοκ `do {}` ή `eval {}` επιστρέφει από την *περικλείουσα sub*, όχι από το μπλοκ:

```
sub safe {
    my $result = eval {
        return 'inside';         # this returns from `safe`, not
↪from eval
    };
    return $result;              # never reached
}
```

Για έξοδο μόνο από την `eval`, χρησιμοποιήστε μια τιμή (`'inside'` ως τελευταία έκφραση του μπλοκ) - η `eval` επιστρέφει την τιμή της τελευταίας έκφρασής της όπως κάθε άλλο μπλοκ.

Επιστροφή από `BEGIN`/`END`

Τα μπλοκ `BEGIN`, `END`, `INIT`, `CHECK`, και `UNITCHECK` είναι subs με την τεχνική έννοια αλλά η τιμή επιστροφής τους απορρίπτεται. Μην μπαίνετε στον κόπο να γράψετε `return ...` μέσα σε αυτά.

Δείτε επίσης

- `return` - η σελίδα perlfunc της δεσμευμένης λέξης.
- `wantarray` - τι θέλει πίσω ο καλών.
- *Lvalue subs και περιβάλλον* - η πλήρης εικόνα για subs ευαίσθητες στο περιβάλλον.
- *Ορίσματα και `@_`* - η συμμετρική ιστορία για το τι εισέρχεται.
- `die`, `warn` - οι εναλλακτικές αντί για «επιστροφή αποτυχίας».

Εμβέλεια: `my`, `our`, `local`, `state`

Μέσα σε μια sub, τέσσεροι δηλωτές ελέγχουν πού ζει μια μεταβλητή, πόσο ζει, και ποιος μπορεί να τη δει. Διαλέξτε λάθος και η sub διαρρέει κατάσταση, αποτυγχάνει να αποτυπώσει σωστά σε closure, ή σιωπηρά αλλοιώνει κάτι στο οποίο βασιζόταν ο καλών.

Δηλωτ	Πεδίο	Διάρκεια ζωής	Τυπική χρήση
`my`	λεξιλογική (το μπλοκ)	μέχρι να τελειώσει το μπλοκ	η προεπιλογή· ιδιωτικές τοπικές
`our`	λεξιλογικό ψευδώνυμο προς πακέτο	όσο υπάρχει το πακέτο	κοινόχρηστες καθολικές πακέτου με σύντομο όνομα
`local`	δυναμική (το δέντρο κλήσεων)	μέχρι να τελειώσει το μπλοκ	προσωρινή αλλαγή ειδικής μεταβλητής
`state`	λεξιλογική (το μπλοκ)	διάρκεια ζωής προγράμματος	επίμονη κατάσταση ανά sub, αρχικοποίηση μία φορά

`my` - λεξιλογικές τοπικές

Η `my` είναι ο καθημερινός δηλωτής:

```
sub greet {
    my ($name) = @_;
    my $message = "Hello, $name!";
    return $message;
}
```

Οι μεταβλητές υπάρχουν για τη διάρκεια της κλήσης, είναι ιδιωτικές στο σώμα, και αποθηκεύονται στο *pad* της sub (μία υποδοχή ανά δηλωμένη λεξιλογική). **Δεν** είναι ορατές στους καλούντες, τους κληθέντες, ή κώδικα μέσω `eval` που δεν βρίσκεται κειμενικά ένθετος μέσα στη sub.

Οι μεταβλητές `my` είναι οι μόνες που αποτυπώνονται σωστά σε closures. Τα closures αποτυπώνουν *κελιά* στο περιβάλλον pad, και μόνο οι μεταβλητές `my` (και `state`) αποκτούν κελιά pad.

`our` - καθολικές πακέτου με λεξιλογικό ψευδώνυμο

```
package Counter;

sub increment {
    our $value;                 # alias to $Counter::value
    $value++;
    return $value;
}
```

Η `our $x` δηλώνει ένα *λεξιλογικό ψευδώνυμο* στην τρέχουσα εμβέλεια προς τη μεταβλητή πακέτου `$Pkg::x`. Η ίδια η μεταβλητή είναι καθολική. Αν δύο πακέτα δηλώσουν `our` το `$VERSION`, καθένα παίρνει τη δική του - η `our` προσθέτει ως πρόθεμα το τρέχον όνομα πακέτου.

Χρησιμοποιήστε `our` όταν:

- Θέλετε πραγματικά μια κοινόχρηστη μεταβλητή σε επίπεδο πακέτου.

- Θέλετε να αποσιωπήσετε τις διαμαρτυρίες `use strict 'vars';` για μεταβλητή που *πρέπει* να είναι καθολική πακέτου (`$VERSION`, `@ISA`, `@EXPORT`, ψευδώνυμα `%ENV`, ...).

Μη χρησιμοποιείτε `our` για να μοιραστείτε κατάσταση μεταξύ subs στο ίδιο αρχείο. Γι" αυτό χρησιμεύει η `my` σε εμβέλεια αρχείου:

```
package Counter;

my $value = 0;                  # private to this package's file
sub increment { ++$value }
sub get       { $value   }
```

`local` - δυναμικά οριοθετημένο προσωρινό

Η `local` *δεν* δηλώνει νέα μεταβλητή. Αποθηκεύει την τρέχουσα τιμή μιας υπάρχουσας (συνήθως καθολικής) μεταβλητής, θέτει νέα τιμή, και επαναφέρει την παλιά τιμή όταν εξέλθει το περικλείον μπλοκ - ακόμα και σε εξαίρεση:

```
sub slurp {
    my ($path) = @_;
    open my $fh, '<', $path or die "open $path: $!";
    local $/;                   # slurp mode for this sub only
    return <$fh>;
}
```

Χωρίς τη `local`, η `$/` του καλούντος θα αλλοιωνόταν μόνιμα. Με αυτή, η αλλαγή περιορίζεται στη δυναμική έκταση της `slurp` - συμπεριλαμβανομένων οποιωνδήποτε subs η `slurp` τυχαίνει να καλέσει.

Η `local` είναι το σωστό εργαλείο για σχεδόν κάθε παράκαμψη ειδικής μεταβλητής (`$/`, `$\`, `$,`, `$_`, `$@`, χειριστές σημάτων στο `%SIG`). Σπάνια είναι το σωστό εργαλείο για συνηθισμένες μεταβλητές χρήστη - εκείνες χρειάζονται `my`.

Για πλήρη κατάλογο, δείτε `local` και τις σελίδες μεταβλητών *κατάσταση σφάλματος* και *I/O*, που αναφέρουν ποιες ειδικές χρειάζονται `local` και σε ποια μοτίβα.

`state` - αρχικοποίηση μία φορά, διατήρηση για πάντα

```
use feature 'state';

sub counter {
    state $n = 0;               # initialised once, ever
    return ++$n;
}

counter();                      # 1
counter();                      # 2
counter();                      # 3
```

Οι μεταβλητές `state` είναι λεξιλογικές που επιβιώνουν μεταξύ κλήσεων. Ο αρχικοποιητής εκτελείται την πρώτη φορά που ο έλεγχος φτάνει στη δήλωση· οι

επόμενες κλήσεις τον παραλείπουν.

Η `state` είναι το σωστό εργαλείο για:

- Caches και μετρητές ανά sub όπου το κόστος κατασκευής δεδομένων κυριαρχεί και δεν θέλετε εργοστάσιο closures.
- Εφάπαξ ρύθμιση που εξαρτάται από τιμές μη διαθέσιμες κατά τη μεταγλώττιση:

```
sub compiled_re {
    my ($pattern) = @_;
    state %cache;
    $cache{$pattern} //= qr/$pattern/;
    return $cache{$pattern};
}
```

(Σημείωση: αυτό αποθηκεύει μόνο το *πρώτο* μοτίβο με κυριολεκτικό `state` ενός μόνο regex· η μορφή hash το γενικεύει.)

Closures: αποτύπωση περιβάλλουσας `my`

Μια ανώνυμη sub αποτυπώνει κάθε μεταβλητή `my` (ή `state`) από την περιβάλλουσα εμβέλεια στην οποία αναφέρεται κειμενικά:

```
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
```

Κάθε κλήση στη `make_adder` δημιουργεί ένα νέο `$n`, και το επιστρεφόμενο closure αναφέρεται στο δικό του `$n`. Δύο κλήσεις παράγουν δύο ανεξάρτητους μετρητές / αθροιστές / μηχανές κατάστασης.

Η έκπληξη του closure-πάνω-σε-μεταβλητή-βρόχου

```
my @subs;
for my $i (1, 2, 3) {
    push @subs, sub { $i };
}

print $_->(), "\n" for @subs;
# 1
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
# 2
# 3   - fresh $i each iteration: each closure sees its own
```

Αυτή είναι η *σωστή* συμπεριφορά με `for my $i`: κάθε επανάληψη δηλώνει ένα νέο `$i`, κάθε closure αποτυπώνει το δικό του.

Η σφαλμένη μορφή εμφανίζεται όταν η `$i` δηλώνεται *έξω* από τον βρόχο:

```
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
```

Δηλώνετε πάντα τις μεταβλητές βρόχου με `my`. Η αποτύπωση closure *ακολουθεί τη μεταβλητή*, όχι την τιμή, και «ίδια μεταβλητή» έναντι «διαφορετική μεταβλητή» αποφασίζεται στη θέση `my`.

Αποτύπωση our και καθολικών πακέτου

Τα ψευδώνυμα `our` ζουν για ολόκληρο το πακέτο, οπότε ένα closure που αποτυπώνει τη `$Counter::value` μοιράζεται κατάσταση με κάθε άλλο κομμάτι κώδικα που αγγίζει τη `$Counter::value`. Αυτό σχεδόν πάντα δεν είναι αυτό που θέλετε - closures πάνω σε καθολικές πακέτου σημαίνουν ότι η σχεδίαση χρειάζεται `my`.

Διάρκεια ζωής pad

Κάθε ενεργή κλήση sub έχει ένα **pad**, μία υποδοχή ανά δηλωμένη λεξιλογική. Το pad είναι αυτό στο οποίο γράφει η `my` και αυτό στο οποίο τα closures κρατούν αναφορά. Όσο ένα closure είναι ζωντανό, κάθε pad που αποτυπώνει είναι ζωντανό - ακόμα κι αν η αρχική κλήση έχει πολύ καιρό επιστρέψει. Έτσι η `$n` της `make_adder` επιβιώνει μετά την κλήση στην ίδια τη `make_adder`.

Δείτε επίσης

- `my`, `our`, `local`, `state` - σελίδες λεπτομερειών perlfunc για κάθε δηλωτή.
- `@_` - η μία μεταβλητή που *πάντα* υπάρχει σε μια sub ανεξάρτητα από δηλωτές.
- *Δήλωση* - ανώνυμες subs και η λεξιλογική μορφή `my sub`.
- *Αναδρομή* - trampolines βασισμένα σε closures και `__SUB__`.
- *Ειδικές μεταβλητές* - σχεδόν κάθε ειδική χρειάζεται `local`, όχι `my`.

Πρωτότυπα

Ένα πρωτότυπο είναι μεταδεδομένα συνδεδεμένα με μια sub που ελέγχουν **πώς ο αναλυτής αναλύει τις κλήσεις προς αυτή τη sub**. Δεν ελέγχει τον χρόνο εκτέλεσης - δεν ελέγχει πλήθος ορισμάτων κατά τον χρόνο εκτέλεσης, δεν επιβάλλει τύπους ορισμάτων, δεν επηρεάζει κλήσεις μεθόδων. Δουλειά του είναι να κάνει μια sub ορισμένη από τον χρήστη να *αναλύεται* σαν ενσωματωμένη.

Αυτός ο στενός σκοπός είναι αυτό που πρέπει να θυμάστε για τα πρωτότυπα. Όλα τα υπόλοιπα ακολουθούν.

Δήλωση

Δύο ισοδύναμες μορφές:

```
sub mypush (\@@) {                  # short form
    my $aref = shift;
    push @$aref, @_;
}

sub mypush :prototype(\@@) {        # attribute form
    ...
}
```

Η μορφή γνωρίσματος είναι υποχρεωτική όταν οι *υπογραφές* είναι ενεργοποιημένες στην ίδια εμβέλεια, επειδή η λίστα σε παρενθέσεις μετά το όνομα της sub κατέχεται τότε από την υπογραφή.

Τι κάνουν πραγματικά τα πρωτότυπα

Ένα πρωτότυπο ξαναγράφει τις προσδοκίες του αναλυτή στο σημείο κλήσης:

```
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
```

Το πρωτότυπο `(&@)` λέει στον αναλυτή: «το πρώτο όρισμα είναι ένα μπλοκ (δεν χρειάζεται η δεσμευμένη λέξη `sub`, ούτε κόμμα μετά από αυτό), τα υπόλοιπα είναι λίστα.» Ακριβώς έτσι αναλύονται οι ενσωματωμένες `grep`, `map`, `sort`.

Αυτό είναι το **μόνο** σημείο όπου τα πρωτότυπα είναι χρήσιμα: κατασκευή subs με μορφή DSL που μοιάζουν με ενσωματωμένες.

Χαρακτήρες πρωτοτύπου

Χαρ.	Σημασία
`$`	βαθμωτό - επιβάλλει βαθμωτό περιβάλλον στο όρισμα
`@`	λίστα - καταναλώνει όλα τα εναπομείναντα ορίσματα, περιβάλλον λίστας
`%`	hash - όπως `@` (λίστα ζυγού μήκους)
`&`	μπλοκ / αναφορά κώδικα - στην πρώτη θέση επιτρέπει παράλειψη `sub`
`*`	bareword, βαθμωτό, typeglob, ή globref
`\$`	αναφορά βαθμωτού - ο καλών πρέπει να περάσει μορφή `$var` (αυτόματη εφαρμογή `\`)
`\@`	αναφορά πίνακα - ο καλών πρέπει να περάσει μορφή `@arr`
`\%`	αναφορά hash - ο καλών πρέπει να περάσει μορφή `%h`
`\&`	αναφορά κώδικα - ο καλών πρέπει να περάσει μορφή `&fn`
`\[$@%&*]`	αναφορά προς οποιονδήποτε από τους τύπους σε αγκύλες
`+`	βαθμωτό Ή `\@`/`\%` (αυτόματη ανίχνευση κυριολεκτικού πίνακα/hash)
`;`	διαχωρίζει υποχρεωτικά από προαιρετικά
`_`	βαθμωτό· αν παραλειφθεί, χρησιμοποιείται εξ ορισμού η `$_`
`()`	κανένα όρισμα - και ενεργοποιεί τη βελτιστοποίηση σταθεροποίησης

Η διαφορά (`\@`) έναντι (`@`)

Αυτός είναι ολόκληρος ο λόγος ύπαρξης πρωτοτύπων με ανάστροφη κάθετο.

```
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
```

Το πρωτότυπο `(\@)` σας επιτρέπει να *γράψετε* την κλήση ως `takes_array(@arr)` - μοιάζοντας ακριβώς με `push @arr, ...` - ενώ ο χρόνος εκτέλεσης λαμβάνει πραγματικά αναφορά πίνακα. Ακριβώς έτσι μοιάζουν στον χρήστη οι ενσωματωμένες `push`, `unshift`, `splice`, και `pop`.

Το πρωτότυπο σταθεροποίησης ()

Μια sub με κενό πρωτότυπο `()` και σώμα μίας έκφρασης που ο βελτιστοποιητής μπορεί να σταθεροποιήσει γίνεται ενσωματώσιμη:

```
sub PI ()  { 3.14159265358979 }
sub MAX () { 100 }

my $area = PI * $r ** 2;         # PI is inlined as 3.14159...
```

Αυτό παράγει η `use constant`:

```
use constant {
    PI  => 3.14159265358979,
    MAX => 100,
};
```

Οι σταθεροποιημένες subs έχουν μηδενικό κόστος κατά τον χρόνο εκτέλεσης. Απαιτούν ωστόσο το πρωτότυπο να είναι ακριβώς `()` (δεν δέχονται ορίσματα), και το σώμα πρέπει να είναι μία μόνη έκφραση που ο βελτιστοποιητής μπορεί να ελαχιστοποιήσει. Η προσθήκη `return` αναιρεί την ενσωμάτωση:

```
sub PI () { return 3.14159; }    # NOT inlined: explicit return
```

Τι *δεν* κάνουν τα πρωτότυπα

- **Δεν** ελέγχουν πλήθος ορισμάτων κατά τον χρόνο εκτέλεσης. Η κλήση μιας `($)` sub με τρία ορίσματα είναι σφάλμα *κατά τη μεταγλώττιση*, αλλά αν παρακάμψετε τον αναλυτή (μέσω `&fn(...)`, αναφορών κώδικα, `goto &fn`), δεν υπάρχει έλεγχος χρόνου εκτέλεσης.
- **Δεν** επηρεάζουν κλήσεις μεθόδων. Η αποστολή μεθόδων επιλύεται κατά τον χρόνο εκτέλεσης μέσω `@ISA`, οπότε η ανάλυση έχει ολοκληρωθεί από καιρό. Τα πρωτότυπα σε μεθόδους αγνοούνται σιωπηρά.
- **Δεν** επηρεάζουν κλήσεις μέσω αναφορών κώδικα. Το σημείο κλήσης βλέπει βαθμωτό, όχι όνομα· χωρίς όνομα, χωρίς αναζήτηση πρωτοτύπου.
- **Δεν** αποτελούν μέρος της ταυτότητας της sub. Η `\&named_sub` χάνει τη γνώση πρωτοτύπου.

Πότε πρέπει να χρησιμοποιήσετε πρωτότυπα

Γράφετε μια sub που μοιάζει με ενσωματωμένη. Παραδείγματα:

- Μια συνάρτηση που καταναλώνει λίστα με όρισμα μπλοκ (`(&@)`): `mygrep`, `myfirst`, `try { ... } catch { ... }`.
- Ένα βοηθητικό «inplace» που τροποποιεί πίνακα επιτόπια (`(\@)` ή `(\@@)`).
- Μια sub τύπου σταθεράς (`()` για ενσωμάτωση).

Αυτή είναι η λίστα. Αν δεν κατασκευάζετε κάτι που μοιάζει με ενσωματωμένη, μην καταφεύγετε σε πρωτότυπα - χρησιμοποιήστε υπογραφή, ή απλώς αποσυσκευάστε το `@_`.

Πότε δεν πρέπει να χρησιμοποιήσετε πρωτότυπα

- Για «έλεγχο τύπων». Τα πρωτότυπα δεν είναι σύστημα τύπων. Η `sub f ($) { ... }` δεν λέει «δέχεται ένα βαθμωτό»· λέει «δέχεται μία έκφραση σε βαθμωτό περιβάλλον». Η κλήση `f(@arr)` περνάει τον *αριθμό* του `@arr`, όχι τον ίδιο τον πίνακα, χωρίς προειδοποίηση.
- Για επιβολή πλήθους ορισμάτων κατά τον χρόνο εκτέλεσης. Χρησιμοποιήστε υπογραφές (που *επιβάλλουν* πληθικότητα κατά τον χρόνο εκτέλεσης) ή γράψτε τον έλεγχο χειροκίνητα.
- Σε μεθόδους. Αγνοούνται σιωπηρά.

Προδηλώσεις και αναδρομή

Μια αναδρομική κλήση βλέπει το πρωτότυπο μόνο αν η sub είχε προδηλωθεί:

```
sub fact ($);                       # forward declaration with␣
↪prototype
sub fact ($) {
    my ($n) = @_;
    return $n <= 1 ? 1 : $n * fact($n - 1);    # parsed with ($) in␣
↪mind
}
```

Χωρίς την προδήλωση, η αναδρομική `fact($n - 1)` μέσα στο σώμα αναλύεται πριν οριστικοποιηθεί το πρωτότυπο `($)`, οπότε το πρωτότυπο δεν εφαρμόζεται σε αυτή την κλήση.

Εξέταση πρωτοτύπου κατά τον χρόνο εκτέλεσης

Η `prototype` επιστρέφει τη συμβολοσειρά πρωτοτύπου μιας sub (ή `undef` αν δεν υπάρχει):

```
prototype(\&push);                  # '\@@' - yes, even built-ins have␣
↪one
prototype('CORE::push');            # '\@@'
prototype(\&my_sub);                # whatever you declared
```

Δείτε επίσης

- `prototype` - ενδοσκόπηση χρόνου εκτέλεσης.
- *Υπογραφές* - η σύγχρονη εναλλακτική για χειρισμό πληθικότητας και ονομαστικών παραμέτρων. Συνυπάρχουν με τα πρωτότυπα αλλά έχουν διαφορετικούς στόχους.
- `sub` - η σελίδα perlfunc της δεσμευμένης λέξης.
- *Δήλωση* - πού προσαρτώνται τα πρωτότυπα στη σύνταξη δήλωσης.
- *Γνωρίσματα* - η `:prototype(...)` είναι η μορφή γνωρίσματος.

Υπογραφές

Μια υπογραφή είναι μια λίστα παραμέτρων σε παρενθέσεις που τοποθετείται αμέσως μετά το όνομα της sub (και μετά τα γνωρίσματα), στη θέση - ή επιπλέον - της χειροκίνητης αποσυσκευασίας `@_` του σώματος. Οι υπογραφές εισάγουν **λεξιλογικές** μεταβλητές παραμέτρων, εκτελούν **έλεγχο πληθικότητας** κατά τον χρόνο κλήσης, και υποστηρίζουν προεπιλεγμένες τιμές και slurpy παραμέτρους.

Signatures stabilised in Perl 5.36; in 5.44 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 5.44 signature syntax.

Βασική μορφή

```
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
```

Κάθε `$name` στην υπογραφή δηλώνει μια λεξιλογική `my` που λαμβάνει το αντίστοιχο όρισμα. Η πληθικότητα ελέγχεται κατά την κλήση: το σώμα δεν εκτελείται ποτέ αν ο αριθμός είναι λάθος.

Αυτή είναι η πιο συγκεκριμένη βελτίωση σε σχέση με την κλασική μορφή:

```
# 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
```

Προεπιλεγμένες τιμές

```
sub greet ($name, $greeting = 'Hello') {
    return "$greeting, $name!";
}
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
greet('Alice');                 # "Hello, Alice!"
greet('Alice', 'Hi');           # "Hi, Alice!"
```

Η προεπιλεγμένη τιμή είναι μια αυθαίρετη έκφραση που αποτιμάται κάθε φορά που η παράμετρος παραλείπεται (οπότε λειτουργούν τόσο `state` όσο και υπολογισμός ανά κλήση). Μπορεί να αναφέρεται σε προηγούμενες παραμέτρους:

```
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
```

Μια προεπιλογή `= undef` είναι ακριβώς ισοδύναμη με «χωρίς προεπιλογή» για την πληθικότητα (η παράμετρος εξακολουθεί να είναι προαιρετική) αλλά τεκμηριώνει την πρόθεση. Χρησιμοποιήστε φρουρούς τύπου `//=` μέσα στο σώμα για σημασιολογία «η `undef` σημαίνει χρήση της εφεδρικής τιμής μου».

Προαιρετικές έναντι υποχρεωτικών

Οι παράμετροι είναι υποχρεωτικές εξ ορισμού. Μια παράμετρος γίνεται προαιρετική:

- Δίνοντάς της μια προεπιλεγμένη τιμή (`$x = 42`).
- Τοποθετώντας την μετά από άλλη προαιρετική παράμετρο (μεταβατικά).
- Ακολουθούμενη από slurpy `@arr` ή `%hash`.

Δεν μπορείτε να έχετε υποχρεωτική παράμετρο μετά από προαιρετική (εκτός μέσω slurpy):

```
sub bad ($x = 1, $y) { ... }     # error: Mandatory parameter
↪follows optional
```

Παράμετροι slurpy

Η τελευταία παράμετρος μπορεί να είναι slurpy `@list` ή `%hash`. Καταναλώνει όλα τα εναπομείναντα ορίσματα:

```
sub join_all ($sep, @parts) {
    return join $sep, @parts;
}

join_all(', ', 'a', 'b', 'c');   # "a, b, c"

sub configure ($name, %opts) {
    $opts{verbose} //= 0;
    ...
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
}

configure('demo', verbose => 1, port => 8080);
```

Το slurpy hash αποτυγχάνει επίσης αν η εναπομείνασα λίστα έχει μονό μήκος:

```
configure('demo', 'verbose');    # error: Odd-sized list given
```

Αυτό είναι το πιο χρήσιμο μοτίβο που ξεκλειδώνουν οι υπογραφές: ένα API ονομαστικών παραμέτρων με μία κατά θέση και ένα slurp `%opts`.

Ανώνυμες παράμετροι

Ένα `$` (ή `@`, `%`) χωρίς όνομα είναι ένα placeholder κατά θέση:

```
sub second ($, $x) { return $x }

second('ignored', 'kept');       # "kept"
```

Αυτό είναι ελαφρώς χρήσιμο όταν προσαρμόζετε ένα callback σε μια διεπαφή που παρέχει περισσότερα ορίσματα από όσα σας ενδιαφέρουν.

Συνδυασμός προεπιλεγμένων τιμών και 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
```

Αυτή η τελευταία περίπτωση είναι η παγίδα: μια προαιρετική παράμετρος κατά θέση ακολουθούμενη από slurpy hash είναι αμφίσημη σε κλήσεις μόνο-ονομαστικών-ορισμάτων. Είτε αφαιρέστε την προαιρετική κατά θέση και βάλτε τα πάντα στο hash, είτε περνάτε πάντα την προαιρετική κατά θέση ρητά.

Μετάβαση από αποσυσκευασία `@_`

Η μηχανική μετάφραση:

```
# Before
sub make_user {
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    my ($name, $email, %extra) = @_;
    ...
}

# After
use feature 'signatures';
sub make_user ($name, $email, %extra) {
    ...
}
```

Αποκτάτε δωρεάν έλεγχο πληθικότητας και ονομαστικές λεξιλογικές. Το σώμα παραμένει αμετάβλητο.

Τα σημεία όπου δεν μπορείτε να μεταφράσετε μηχανικά:

- Το σώμα χρησιμοποιεί δυναμικά τον *αριθμό* ορισμάτων (`if (@_ >= 3)`) - οι υπογραφές καθορίζουν πληθικότητα κατά τον χρόνο ανάλυσης, οπότε αναδιαμορφώστε σε προαιρετική παράμετρο με προεπιλογή.
- Το σώμα τροποποιεί το `$_[0]` για το αποτέλεσμα ψευδωνυμοποίησης - κρατήστε την κλασική μορφή, καθώς οι υπογραφές εισάγουν *αντίγραφα*, σπάζοντας το ψευδώνυμο.
- The body uses `goto &other` and wants to forward the original `@_` - `@_` is still populated under signatures in 5.44, but this is implementation-defined and may change. Keep the classic form for tail-call forwarding.

Υπογραφές και πρωτότυπα είναι ορθογώνια

Οι υπογραφές και τα *πρωτότυπα* λύνουν διαφορετικά προβλήματα:

Μηχανισμός	Επηρεάζει	Πότε ελέγχεται	Χρήσιμο για
υπογραφή	η οπτική του σώματος	χρόνος εκτέλεσης	πληθικότητα, ονομαστικές παράμετροι
prototype	πώς αναλύει ο αναλυτής	κατά τη μεταγλώττιση	ομοιώματα DSL

Μια sub μπορεί να έχει και τα δύο, με το πρωτότυπο γραμμένο ως γνώρισμα:

```
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
```

Το πρωτότυπο διαμορφώνει το σημείο κλήσης· η υπογραφή διαμορφώνει το σώμα. Δεν αλληλοεπικαλύπτονται.

Ο σημαντικότερος λόγος μετάβασης

Ανίχνευση κλήσεων λάθος πληθικότητας τη στιγμή που συμβαίνουν - όχι αρκετές γραμμές μετά, όταν τελικά κάνετε `$y + 1` και προκαλείτε προειδοποίηση για τιμή undef. Οι υπογραφές μετατρέπουν μια κατηγορία λεπτών σφαλμάτων σε ξεκάθαρο μήνυμα σφάλματος.

Δείτε επίσης

- *Ορίσματα και @_* - ο κλασικός μηχανισμός με τον οποίο συνυπάρχουν οι υπογραφές.
- *Πρωτότυπα* - διαφορετικός μηχανισμός, που συχνά μπερδεύεται με τις υπογραφές.
- `sub` - η σελίδα perlfunc της δεσμευμένης λέξης.
- *Δήλωση* - πού βρίσκονται οι υπογραφές στη σύνταξη δήλωσης.
- *Εμβέλεια* - οι παράμετροι υπογραφής είναι λεξιλογικές τύπου `my`.

Αναδρομή

Οι subs μπορούν να καλούν τον εαυτό τους, η μία την άλλη, ή - με λίγη τελετουργία - να αντικαθιστούν το δικό τους πλαίσιο κλήσης και να περνούν τον έλεγχο αλλού χωρίς να μεγαλώνει η στοίβα. Αυτή η σελίδα καλύπτει τους τέσσερις μηχανισμούς: απλή αναδρομή, αμοιβαία αναδρομή, κλήσεις ουράς `goto &fn`, και `__SUB__` για ανώνυμη αυτο-αναφορά. Καλύπτει επίσης το όριο βάθους στο οποίο θα φτάσετε αν τα αγνοήσετε.

Απλή αναδρομή

```
sub factorial {
    my ($n) = @_;
    return 1 if $n <= 1;
    return $n * factorial($n - 1);
}

factorial(10);                  # 3628800
```

Κάθε αναδρομική κλήση προσθέτει ένα νέο πλαίσιο στη στοίβα κλήσεων της Perl. Η στοίβα κοστίζει μνήμη και ένα σταθερό κόστος ανά κλήση· για ρηχή αναδρομή αυτό είναι αμελητέο, για βαθιά αναδρομή είναι το σημείο συμφόρησης.

Η Perl εκπέμπει μια προειδοποίηση «Deep recursion on subroutine X» όταν μια sub αναδράμει πέρα από 100 επίπεδα βάθους (κατώφλι μεταγλώττισης, `PERL_SUB_DEPTH_WARN`). Η προειδοποίηση ανήκει στην κατηγορία `recursion` - η `no warnings 'recursion'` την αποσιωπά. Το μόνο σκληρό όριο είναι η στοίβα του OS, τυπικά 8 MiB εξ ορισμού, ρυθμιζόμενο με `ulimit -s`.

Για βαθιούς, δομικά αναδρομικούς φόρτους εργασίας, προτιμήστε ένα από τα:

- Μια ρητή ουρά εργασίας (επαναδιατύπωση σε επαναληπτική μορφή).
- Αναδρομή ουράς μέσω `goto &fn` (βλέπε παρακάτω).
- Ένα trampoline (βλέπε παρακάτω).

Αμοιβαία αναδρομή

Δύο subs που καλούν η μία την άλλη:

```
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
```

Η αμοιβαία αναδρομή δεν έχει ειδική μεταχείριση από τον χρόνο εκτέλεσης - είναι απλώς δύο subs που τυχαίνει να καλούν η μία την άλλη. Ισχύει το ίδιο όριο βάθους.

Αν η `is_even` ορίζεται πρώτη και αναφέρεται στην `is_odd` πριν υπάρξει η `is_odd`, θα δείτε «Bareword not allowed» ή «Called undefined subroutine» ανάλογα με τη μορφή κλήσης. Προδηλώστε για να το διορθώσετε:

```
sub is_odd;                     # forward declaration
sub is_even { ... is_odd($n-1) }
sub is_odd  { ... is_even($n-1) }
```

`goto &fn` - προώθηση κλήσης ουράς

Η μορφή `goto &SUBROUTINE` **αντικαθιστά το τρέχον πλαίσιο** με μια κλήση σε άλλη sub. Το τρέχον πλαίσιο εξαφανίζεται - δεν βρίσκεται στη στοίβα - οπότε η κλήση δεν μεγαλώνει τη στοίβα:

```
sub api_v2 {
    my @args = @_;
    push @args, deprecated => 1;
    @_ = @args;
    goto &api_v1;               # tail-call: api_v1 sees the␣
 ↪modified @_
}
```

Αυτό είναι το σωστό εργαλείο όταν:

- Μια sub-περιτύλιγμα χρειάζεται να προωθήσει σε πραγματική υλοποίηση χωρίς να αφήσει πλαίσιο στοίβας πίσω. Decorators, shims καταγραφής, AUTOLOAD dispatchers.
- Μια αναδρομική sub μπορεί να εκφράσει την αναδρομή της ως «κάνε κάποια δουλειά, μετά γίνε αυτή η άλλη κλήση». Κλασική αναδρομή ουράς:

```
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
```

Η μηχανική: το @_ γίνεται ο πίνακας ορισμάτων της καλούμενης sub, το τρέχον πλαίσιο ξετυλίγεται, ο έλεγχος πηδά. Ο καλών της *τρέχουσας* sub θα δει το αποτέλεσμα της sub στην οποία έγινε goto, ποτέ αυτής που κάλεσε.

Η `goto &fn` είναι η **μόνη** μορφή goto που εκτελεί κλήση ουράς. Η μορφή βασισμένη σε ετικέτα (`goto LABEL`) είναι άσχετη και πολύ πιο περιορισμένη.

Δείτε `goto` για πλήρη περιγραφή της δεσμευμένης λέξης και τους περιορισμούς μορφής.

__SUB__ - ανώνυμη αυτο-αναφορά

```
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
```

Η __SUB__ αποτιμάται σε αναφορά κώδικα προς την τρέχουσα εκτελούμενη sub. Είναι ο μόνος τρόπος να κάνετε αναδρομή σε ανώνυμη sub χωρίς να την αποθηκεύσετε πρώτα σε ονομαστική μεταβλητή (και χωρίς να διαρρέει ισχυρή αναφορά προς αυτή μέσα από τον εαυτό της, η κλασική μορφή διαρροής μνήμης).

Για ανώνυμες subs που δεν χρειάζεται να κάνουν αναδρομή, προτιμήστε την απλή ανώνυμη μορφή. Η __SUB__ υπάρχει ειδικά για την περίπτωση αναδρομής.

Trampolines

Όταν θέλετε αναδρομή με οριοθετημένη στοίβα αλλά η `goto &fn` είναι δύσχρηστη (επειδή η αναδρομή είναι αμοιβαία, ή διακλαδίζεται σε πολλές υπο-περιπτώσεις), το μοτίβο trampoline λειτουργεί:

```
sub trampoline {
    my $next = shift;
    while (ref $next eq 'CODE') {
        $next = $next->();
    }
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    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
```

Κάθε «αναδρομική» κλήση επιστρέφει ένα closure για να γίνει η επόμενη κλήση, αντί να καλεί τον εαυτό της απευθείας. Το trampoline οδηγεί τα closures σε βρόχο. Η στοίβα παραμένει σε βάθος 2 για πάντα.

Τα trampolines είναι βαρύτερα από τη `goto &fn` (μία δέσμευση closure ανά «κλήση»), αλλά γενικεύονται σε αυθαίρετη ροή ελέγχου - συμπεριλαμβανομένων continuations και κώδικα τύπου CPS. Για Perl παραγωγής, η `goto &fn` είναι η συνηθισμένη επιλογή· τα trampolines εμφανίζονται όταν η δομή της αναδρομής κάνει τη `goto` δυσεπίτακτη.

Το όριο βάθους, στην πράξη

```
sub recurse {
    my ($n) = @_;
    return $n if $n <= 0;
    return recurse($n - 1);
}

recurse(2000);                    # warning: Deep recursion on
↪subroutine
                                  # (default limit 1000)
```

Η προειδοποίηση «Deep recursion» είναι ενημερωτική, όχι μοιραία - εκπέμπεται μία φορά ανά sub. Θα εξαντλήσετε τον πραγματικό χώρο στοίβας πολύ πριν φτάσετε κάποιο πρακτικό όριο στις περισσότερες πλατφόρμες. Αν δείτε αυτή την προειδοποίηση, η απάντηση είναι σχεδόν πάντα αναδιαμόρφωση (επαναληπτικός βρόχος, `goto &fn`, ή trampoline), όχι αύξηση του ορίου.

Αν έχετε ήδη επαληθεύσει ότι η αναδρομή είναι οριοθετημένη και απλώς θέλετε να αποσιωπήσετε τον θόρυβο, περιορίστε την κατηγορία προειδοποίησης:

```
no warnings 'recursion';
```

Αυτό αποσιωπά εντελώς την προειδοποίηση· δεν αλλάζει το όριο στοίβας του OS, το οποίο είναι αυτό που τελικά τερματίζει μια ανεξέλεγκτη αναδρομή ανεξάρτητα από τις προειδοποιήσεις.

Δείτε επίσης

- `goto` - η δεσμευμένη λέξη πίσω από τη `goto &fn`.
- `caller` - λειτουργεί μέσα από πλαίσια μετά από goto με την ίδια αρίθμηση όπως τα κανονικά πλαίσια.
- *Ορίσματα και @_* - σημασιολογία `@_` υπό `goto &fn`.
- *Δήλωση* - ανώνυμες subs, το υπόστρωμα για `__SUB__` και trampolines.
- *Εμβέλεια* - closures, το υπόστρωμα για trampolines.

Lvalue subs και περιβάλλον

Δύο σχετικά θέματα που εμφανίζονται όπου μια sub χρειάζεται να συμπεριφέρεται διαφορετικά ανάλογα με τον τρόπο χρήσης της: subs **`:lvalue`** (αυτές που μπορούν να χρησιμοποιηθούν στην αριστερή πλευρά μιας ανάθεσης) και αποστολή βασισμένη στο **`wantarray`** ανάλογα με το περιβάλλον του καλούντος.

Γιατί το περιβάλλον έχει σημασία στις subs

Κάθε έκφραση Perl αποτιμάται σε ένα περιβάλλον - λίστα, βαθμωτό (με υπο-ποικιλίες: αριθμητικό, συμβολοσειράς, λογικό), ή κενό - που επιβάλλεται από αυτό που την περιβάλλει. Μια κλήση sub κληρονομεί το περιβάλλον της θέσης της:

```
my @r = my_sub();                   # list context
my $r = my_sub();                   # scalar context
my_sub();                           # void context
print my_sub();                     # list context (print's args are␣
↪LIST)
my $n = () = my_sub();              # scalar of (a list assignment) -␣
↪element count
```

Για τις περισσότερες subs, αυτό είναι αόρατο - επιστρέφουν τα ίδια δεδομένα ούτως ή άλλως και η αυτόματη μετατροπή περιβάλλοντος της Perl κάνει τη σωστή δουλειά. Για κάποιες subs η δουλειά είναι πραγματικά διαφορετική ανάλογα με το περιβάλλον, και αυτές οι subs ωφελούνται από ρητή εξέταση της `wantarray`.

`wantarray`: ο τριμερής διακόπτης

```
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␣
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    return
    }
}
```

Οι τρεις τιμές επιστροφής της `wantarray`:

Τιμή επιστροφής	Περιβάλλον του καλούντος	Πρακτική σημασία
1 (αληθές)	λίστα	επιστροφή όλων
`0` (ψευδές)	scalar	επιστροφή ενός πράγματος (π.χ. αριθμός)
`undef`	κενό	κανέναν δεν ενδιαφέρει· σκεφτείτε πρόωρη επιστροφή

Η `defined wantarray` είναι ο έλεγχος για «βαθμωτό ή λίστα» (οτιδήποτε μη κενό).

Η συντόμευση κενού περιβάλλοντος

Όταν η εργασία είναι ακριβή και ο καλών πέταξε το αποτέλεσμα, επιστρέψτε πρόωρα:

```
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
my @r = expensive_query();                 # list: full result
my $n = expensive_query();                 # scalar: count
```

Αυτό το μοτίβο είναι ο κορυφαίος λόγος να συμβουλευτείτε τη `wantarray`: μετατροπή μιας sub «κάνε πάντα τη δουλειά» σε sub «κάνε τη δουλειά μόνο αν ο καλών τη θέλει».

Πότε *δεν* πρέπει να χρησιμοποιήσετε `wantarray`

- Για υπερφόρτωση ευκολίας. Μια sub που επιστρέφει hashref σε βαθμωτό περιβάλλον και hash σε περιβάλλον λίστας είναι κίνδυνος συντήρησης - οι αναγνώστες δεν μπορούν να καταλάβουν από το σημείο κλήσης τι παίρνουν. Διαλέξτε ένα.
- Για subs που είναι προφανώς τύπου-λίστας ή τύπου-βαθμωτού. Μια sub με όνομα `count_users` πρέπει να επιστρέφει αριθμό· οι αναγνώστες δεν περιμένουν να συμπεριφέρεται διαφορετικά σε περιβάλλον λίστας.

Subs :lvalue

Μια sub :lvalue είναι αυτή της οποίας η τιμή επιστροφής μπορεί η ίδια να λάβει ανάθεση. Το σώμα πρέπει να τελειώνει σε μια έκφραση που έχει θέση μνήμης - τυπικά μια λεξιλογική my ή στοιχείο συγκεντρωτικού:

```
sub editable :lvalue {
    my $self = shift;
    $self->{counter};                       # last expression is an␣
↪lvalue
}

my $obj = { counter => 0 };
bless $obj, 'Counter';

$obj->editable = 42;                        # writes through to $obj-
↪>{counter}
```

Η τελευταία έκφραση του σώματος είναι αυτό στο οποίο γράφει η ανάθεση. Αν η έκφραση δεν είναι lvalue (return $x + 1, return scalar @list), η ανάθεση είναι παράνομη και θα λάβετε ξεκάθαρο σφάλμα.

Η :lvalue σπάνια είναι το σωστό εργαλείο. Τα μόνα σημεία όπου αποδίδει:

- Tied μεταβλητές και περιτυλίγματα τύπου tied, όπου η sub μεσολαβεί στην πρόσβαση σε πραγματικό lvalue από κάτω.
- DSL που πραγματικά θέλουν σύνταξη ανάθεσης για ρύθμιση (π.χ. $config->host = 'localhost';).

Για «θέσε μια ιδιότητα σε αντικείμενο», ένας απλός setter είναι πιο ξεκάθαρος:

```
sub set_counter {
    my ($self, $value) = @_;
    $self->{counter} = $value;
    return $self;                            # method chaining
}

$obj->set_counter(42);                      # explicit, no surprises
```

:lvalue και η δομή του σώματος

Επειδή η τελευταία έκφραση του σώματος είναι ειδική, οι κανόνες είναι περιοριστικοί:

```
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 {
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    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
}
```

Ο κανόνας «χωρίς ρητό `return`» είναι ο ίδιος που κάνει τα πρωτότυπα σταθεροποίησης να λειτουργούν· και τα δύο βασίζονται στο ότι το σώμα τελειώνει με μία μόνη έκφραση που παράγει τιμή.

Για τη σύγχρονη εναλλακτική - ένας setter που επιστρέφει `$self` για αλυσίδωση - δείτε τον *οδηγό αντικειμενοστραφούς προγραμματισμού*.

Συνδυασμός `:lvalue` με περιβάλλον

Μια sub `:lvalue` δεν μπορεί εύκολα να διακλαδιστεί βάσει `wantarray`, επειδή η lvalue εκδοχή της κλήσης δεν ταιριάζει ακριβώς στο μοντέλο «επιστρέφει μια τιμή». Αν θέλετε και τις δύο συμπεριφορές από μία sub, αποδεχτείτε ότι βρίσκεστε σε περιοχή DSL και τεκμηριώστε προσεκτικά τη σύμβαση - ή χωρίστε σε δύο subs.

Τελεστές που επηρεάζουν το περιβάλλον

Μερικοί τελεστές επιβάλλουν συγκεκριμένο περιβάλλον στο όρισμά τους· η sub θα δει αυτό το περιβάλλον ανεξάρτητα από το πού κατευθύνεται η περιβάλλουσα έκφραση:

```
my @r = scalar my_sub();            # scalar() forces scalar context on␣
↪my_sub
my $n = () = my_sub();              # the inner () = ... is list␣
↪context;
                                    # outer assignment to $n is scalar␣
↪of that
```

Το ιδίωμα `() = LIST` είναι ο τυπικός τρόπος για επιβολή περιβάλλοντος λίστας σε κλήση sub και στη συνέχεια ανάγνωση του πλήθους. Δείτε *ανάθεση* για τη συζήτηση σε επίπεδο τελεστή.

Δείτε επίσης

- `wantarray` - η δεσμευμένη λέξη.
- *Τιμές επιστροφής* - η υπόλοιπη εικόνα τιμών επιστροφής (χωρίς την ανατροπή lvalue).
- *Γνωρίσματα* - η `:lvalue` είναι ένα από πολλά ενσωματωμένα γνωρίσματα.
- *Τελεστής ανάθεσης* - τι κάνει πραγματικά η `=` του καλούντος σε κάθε πλευρά μιας κλήσης `:lvalue`.

- `scalar` - ρητός εξαναγκασμός περιβάλλοντος.

Γνωρίσματα

Ένα γνώρισμα sub είναι ένας σχολιασμός με πρόθεμα άνω-κάτω τελεία ανάμεσα στο όνομα (ή την υπογραφή) της sub και στο σώμα της. Τα γνωρίσματα σημειώνουν μια sub ως έχουσα κάποια ειδική ιδιότητα - ότι είναι μέθοδος, ότι είναι lvalue, ότι έχει πρωτότυπο, ότι είναι σταθερά, ότι υπόκειται σε κάποιον μηχανισμό τρίτων - πάνω στον οποίο μπορεί να δράσει ο χρόνος εκτέλεσης, ο αναλυτής, ή ένας χειριστής γνωρισμάτων γραμμένος από τον χρήστη.

```
sub greet :method ($name)              { ... }
sub editable :lvalue                   { ... }
sub mypush :prototype(\@@)             { ... }
sub PI :const                          { 3.14159 }
sub handler :Memoize                   { ... }   # third-party
 ↪attribute
```

Τα τέσσερα βήματα γνωρισμάτων

Για να παρακολουθήσετε πώς ένα γνώρισμα επηρεάζει μια sub, διακρίνετε τέσσερις φάσεις:

1. **Ανάλυση.** Ο μεταγλωττιστής διαβάζει `:foo(args)` και το χωρίζει σε όνομα (`foo`) και προαιρετική συμβολοσειρά παραμέτρων (`args`).
2. **Καταχώρηση.** Ο μεταγλωττιστής καλεί είτε έναν ενσωματωμένο χειριστή (για `:lvalue`, `:method`, `:prototype(...)`) είτε τη `MODIFY_CODE_ATTRIBUTES` από το σχετικό πακέτο (για γνωρίσματα ορισμένα από τον χρήστη).
3. **Αποθήκευση.** Τα ενσωματωμένα γνωρίσματα θέτουν μια σημαία στο CV (το μεταγλωττισμένο αντικείμενο sub). Τα γνωρίσματα χρήστη αποθηκεύονται όπου επιλέξει ο χειριστής.
4. **Ερώτημα.** Οποιοσδήποτε μπορεί αργότερα να ρωτήσει `attributes::get(\&sub)` για τη λίστα γνωρισμάτων. Τα ενσωματωμένα γνωρίσματα μπορούν επίσης να ερωτηθούν μέσω πιο άμεσων API (`prototype`, `is_lvalue`, ...).

Οι τέσσερις φάσεις εξηγούν γιατί τα γνωρίσματα είναι ετερογενή σε συμπεριφορά: τα ενσωματωμένα γνωρίσματα είναι έννοιες χρόνου εκτέλεσης· τα γνωρίσματα χρήστη είναι ό,τι αποφασίσει ο κώδικας στη `MODIFY_CODE_ATTRIBUTES` να κάνει μαζί τους.

Ενσωματωμένα γνωρίσματα

`:lvalue`

Σημαίνει τη sub ως χρησιμοποιήσιμη στην αριστερή πλευρά μιας ανάθεσης. Η τελευταία έκφραση του σώματος πρέπει να είναι lvalue. Δείτε *lvalue και περιβάλλον*.

```
sub editable :lvalue { $self->{x} }

$obj->editable = 42;            # writes through to $self->{x}
```

`:method`

Υπόδειξη στον αναλυτή και στην ενδοσκόπηση ότι αυτή η sub προορίζεται να κληθεί ως μέθοδος. Η σημαία είναι κυρίως ενημερωτική· καταστέλλει επίσης τα αποτελέσματα πρωτοτύπων (αφού οι μέθοδοι δεν τηρούν τα πρωτότυπα ούτως ή άλλως):

```
package Counter;

sub increment :method {
    my $self = shift;
    $self->{n}++;
}
```

Η `:method` **δεν** κάνει αυτόματα `use parent` ούτε θέτει `@ISA`· δεν αλλάζει την αποστολή. Είναι δείκτης τεκμηρίωσης / πρόθεσης που ο χρόνος εκτέλεσης χρησιμοποιεί επίσης για να παρακάμψει την επεξεργασία πρωτοτύπων.

`:prototype(...)`

Η μορφή γνωρίσματος ενός *πρωτοτύπου*, απαιτείται όταν οι *υπογραφές* είναι ενεργοποιημένες στην ίδια εμβέλεια:

```
use feature 'signatures';
sub mygrep :prototype(&@) ($code, @list) {
    grep { $code->($_) } @list;
}
```

Η λίστα παραμέτρων του σώματος είναι η υπογραφή· το γνώρισμα φέρει το πρωτότυπο. Χωρίς τη μορφή γνωρίσματος, ο αναλυτής δεν μπορεί να ξεχωρίσει ποια λίστα σε παρενθέσεις είναι ποια.

`:const`

Υπόσχεται στον χρόνο εκτέλεσης ότι το σώμα της sub παράγει σταθερό αποτέλεσμα, επιτρέποντας πιο επιθετική ενσωμάτωση από ό,τι δίνει το σκέτο πρωτότυπο `()`:

```
sub PI :const { 3.14159265358979 }
```

Η `:const` εισήχθη για χρήση από τη `use constant` και παρόμοια εργαλεία. Ο περισσότερος κώδικας πρέπει να χρησιμοποιεί τη `use constant` αντί να εφαρμόζει απευθείας `:const`.

Γνωρίσματα τρίτων

Οποιοδήποτε πακέτο μπορεί να καταχωρήσει χειριστές γνωρισμάτων για αναφορές κώδικα υλοποιώντας τη `MODIFY_CODE_ATTRIBUTES`. Το κλασικό παράδειγμα είναι η `Attribute::Handlers`, που παρέχει ένα πιο δηλωτικό API πάνω στον ωμό μηχανισμό. Ένα γνώρισμα χρήστη μοιάζει πανομοιότυπο με ενσωματωμένο στο σημείο κλήσης:

```
use Some::Memoize;              # provides ':Memoize' attribute

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
sub expensive :Memoize {
    ...
}
```

Το πακέτο που παρέχει το γνώρισμα αποφασίζει τι σημαίνει η `:Memoize`. Συνηθισμένα γνωρίσματα τρίτων:

- `:Memoize` - αυτόματη αποθήκευση τιμών επιστροφής σε cache (από τη `Memoize`).
- `:Logged` - περιτύλιγμα της sub με καταγραφή εισόδου/εξόδου (από διάφορα αρθρώματα τύπου AOP).
- `:Test`, `:Benchmark` - δείκτες πλαισίου δοκιμών.

Όταν βλέπετε ένα γνώρισμα-με-άνω-κάτω-τελεία που δεν βρίσκεται στην παραπάνω λίστα ενσωματωμένων, βρείτε το πακέτο που το παρέχει.

Πώς γράφονται τα γνωρίσματα

Πολλαπλά γνωρίσματα μπορούν να αλυσιδωθούν. Η σειρά στο δίσκο είναι η σειρά επεξεργασίας τους:

```
sub fancy :method :prototype(\@) :Memoize {
    ...
}
```

Τα γνωρίσματα προσαρτώνται *τόσο* στην προδήλωση όσο και στον ορισμό· αν γράψετε και τα δύο, οι λίστες γνωρισμάτων πρέπει να συμφωνούν:

```
sub greet :method;                # forward declaration with :method
sub greet :method ($name) {       # body, same attribute
    ...
}
```

Η υπογραφή, όταν υπάρχει, έρχεται μετά τα γνωρίσματα:

```
sub greet :method ($name) {       # OK
    ...
}

sub greet ($name) :method {       # error: attributes must precede
↪signature
    ...
}
```

Τι *δεν* κάνουν τα γνωρίσματα

- Δεν είναι σχολιασμοί τύπου. Η `:Int` δεν είναι ενσωματωμένη· προσπάθειες ανάγνωσής της με αυτόν τον τρόπο δεν θα κάνουν τίποτα εκτός αν κάποιο πακέτο έχει καταχωρήσει χειριστή που σημαίνει το ίδιο πράγμα.

- Δεν είναι μεταδεδομένα για το πλαίσιο δοκιμών εκτός αν το πλαίσιο δοκιμών κατέχωρησε χειριστή γι" αυτά. Το σκόρπισμα `:tested` σε subs δεν κάνει τίποτα χωρίς κώδικα στην άλλη πλευρά που να διαβάζει τον σχολιασμό.
- Δεν είναι οδηγίες βελτιστοποίησης. Η `:fast` δεν κάνει τίποτα.

Ερώτημα γνωρισμάτων μιας sub

```
use attributes;

sub demo :method :prototype($) :Memoize { ... }

my @attrs = attributes::get(\&demo);
# returns: ('method', 'prototype($)', 'Memoize')
```

Αυτό είναι το ισοδύναμο ενδοσκόπησης της `prototype` - απαντάει στο «ποιους σχολιασμούς φέρει αυτή η sub;» αντί στο «πώς μοιάζει αυτή η sub στον αναλυτή;».

Δείτε επίσης

- *Δήλωση* - πού ταιριάζει η σύνταξη `:attr` στη γραμματική δήλωσης.
- *Πρωτότυπα* - η `:prototype(...)` είναι ένα γνώρισμα μεταξύ πολλών.
- *Lvalue και περιβάλλον* - η σημασία της `:lvalue`.
- `prototype` - ενδοσκόπηση χρόνου εκτέλεσης ενός συγκεκριμένου ενσωματωμένου γνωρίσματος.
- *Αντικειμενοστραφής προγραμματισμός* - η `:method` και το ευρύτερο πλαίσιο OO.

Phasers: `BEGIN`, `UNITCHECK`, `CHECK`, `INIT`, `END`

Πέντε ειδικά ονομασμένα μπλοκ επιτρέπουν στον κώδικα να συνδεθεί με τις μεταβάσεις που κάνει ένα πρόγραμμα ανάμεσα στο να μεταγλωττίζεται και στο να εκτελείται. Μοιάζουν με υπορουτίνες αλλά δεν είναι: μπορείτε να γράψετε πολλά από το καθένα, κανένα δεν έχει καλέσιμο όνομα, και το καθένα πυροδοτείται αυτόματα στη δική του στιγμή.

```
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
```

Το πρόθεμα sub (`sub BEGIN { ... }`) γίνεται αποδεκτό αλλά δεν προσθέτει τίποτα και αποθαρρύνεται. Αυτά είναι μπλοκ, όχι ονομασμένες υπορουτίνες· δεν μπορείτε να τα καλέσετε, μόνο να τα αφήσετε να πυροδοτηθούν.

Πότε εκτελείται το καθένα

Ένα πρόγραμμα Perl ζει σε δύο φάσεις: μια **φάση μεταγλώττισης** που αναλύει τον πηγαίο κώδικα σε μια εσωτερική μορφή, και μια **φάση εκτέλεσης** που τον εκτελεί. Οι phasers σημαδεύουν το όριο.

- Το `BEGIN` εκτελείται τη στιγμή που αναλύεται η αγκύλη κλεισίματός του, προτού καν διαβαστεί το υπόλοιπο του αρχείου. Έτσι λειτουργεί η `use`: ένα `use` είναι μια `require` τυλιγμένη σε `BEGIN` συν `import`, οπότε τα εισαγόμενα ονόματα υπάρχουν για τον κώδικα που ακολουθεί. Μόλις εκτελεστεί ένα `BEGIN`, γίνεται απροσδιόριστο και η μνήμη του ανακτάται.
- Το `UNITCHECK` εκτελείται μόλις η **μονάδα μεταγλώττισης** που το περιέχει ολοκληρώσει τη μεταγλώττισή της. Μια μονάδα είναι το κύριο πρόγραμμα, ένα άρθρωμα που φορτώνεται με `require`, ένα string `eval`, ένα `do` FILE, ή κώδικας που μεταγλωττίζεται από την κατασκευή regex `(?{ })`. Επειδή οι μονάδες ολοκληρώνουν τη μεταγλώττισή τους σε διαφορετικές στιγμές, το `UNITCHECK` είναι ο μόνος phaser του οποίου ο χρονισμός παρακολουθεί το μεμονωμένο αρχείο και όχι το πρόγραμμα ως σύνολο.
- Το `CHECK` εκτελείται μία φορά, αφού τελειώσει η **αρχική** φάση μεταγλώττισης ολόκληρου του προγράμματος και πριν ξεκινήσει η φάση εκτέλεσης.
- Το `INIT` εκτελείται μία φορά, λίγο πριν αρχίσει η φάση εκτέλεσης, μετά από όλα τα μπλοκ `CHECK`.
- Το `END` εκτελείται μία φορά, όσο πιο αργά γίνεται: αφού έχει τελειώσει το πρόγραμμα (συμπεριλαμβανομένου του μετά από μια `die`), λίγο πριν εξέλθει ο διερμηνευτής.

Τα `BEGIN` και `UNITCHECK` είναι δεμένα με την *ανάλυση*, όχι με τη φάση του διερμηνευτή, οπότε μπορούν να πυροδοτηθούν κατά τη διάρκεια οποιασδήποτε φάσης: ένα string `eval` κατά τον χρόνο εκτέλεσης εξακολουθεί να μεταγλωττίζει το σώμα του, και κάθε `BEGIN` ή `UNITCHECK` μέσα του εκτελείται εκεί και τότε. Τα `CHECK` και `INIT`, αντιθέτως, σημαδεύουν το εφάπαξ όριο μεταγλώττισης/εκτέλεσης του κύριου προγράμματος.

Πολλαπλότητα και σειρά εκτέλεσης

Μπορείτε να γράψετε οποιονδήποτε αριθμό από το κάθε μπλοκ. Όλα εκτελούνται, με καθορισμένη σειρά:

- `BEGIN`: first in, first out (FIFO) όπως συναντώνται κατά τη μεταγλώττιση. Το πρώτο `BEGIN` στη σειρά του πηγαίου κώδικα εκτελείται πρώτο.
- `UNITCHECK`: last in, first out (LIFO) μέσα σε κάθε μονάδα, αφού μεταγλωττιστεί αυτή η μονάδα.
- `CHECK`: LIFO, μετά από όλη τη μεταγλώττιση.
- `INIT`: FIFO, λίγο πριν τον χρόνο εκτέλεσης.
- `END`: LIFO κατά την έξοδο. Το τελευταίο `END` που ορίζεται είναι το πρώτο που εκτελείται.

Η διδακτική περίπτωση είναι ένα αρχείο που αναμειγνύει συνηθισμένες εντολές με τα και τα πέντε είδη μπλοκ. Τα μπλοκ δεν εκτελούνται εκεί όπου είναι γραμμένα· εκτελούνται στη φάση τους, με τη σειρά τους ανά φάση, ενώ οι συνηθισμένες εντολές

εκτελούνται ενδιάμεσα κατά τον χρόνο εκτέλεσης. Εννοιολογικά, για ένα μεμονωμένο αρχείο:

```
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
```

Έτσι τα BEGIN και UNITCHECK εκτελούνται πριν από οποιοδήποτε CHECK· το CHECK εκτελείται πριν το INIT· το INIT εκτελείται πριν την πρώτη γραμμή συνηθισμένου κώδικα· και κάθε END εκτελείται μετά την τελευταία. Μέσα στα BEGIN και INIT, τα νωρίτερα γραμμένα μπλοκ εκτελούνται νωρίτερα· μέσα στα UNITCHECK, CHECK και END, τα αργότερα γραμμένα μπλοκ εκτελούνται νωρίτερα.

Τα μπλοκ END αναλυτικά

Το END είναι ο phaser καθαρισμού. Εκτελείται με LIFO ώστε η αποδόμηση να ξετυλίγεται με την αντίστροφη σειρά της εγκατάστασης, όπως το περιμένουν οι εμφωλευμένοι πόροι.

Μέσα σε ένα μπλοκ END, το `$?` κρατά την τιμή που το πρόγραμμα πρόκειται να περάσει στην `exit`. Μπορείτε να το διαβάσετε για να μάθετε πώς τελειώνει το πρόγραμμα, και μπορείτε να του αναθέσετε τιμή για να αλλάξετε τον κωδικό εξόδου:

```
END {
    $? = 0 if $? == 42;     # rewrite one exit code on the way out
}
```

Προσέξτε μην αλλοιώσετε το $? κατά λάθος. Η εκτέλεση οτιδήποτε μέσω της `system` μέσα σε ένα μπλοκ END αντικαθιστά το $? με την κατάσταση του παιδιού, που στη συνέχεια γίνεται ο κωδικός εξόδου του προγράμματος εκτός αν το αποθηκεύσετε και το επαναφέρετε.

Τα μπλοκ END **δεν** εκτελούνται σε αρκετές περιπτώσεις:

- Υπό τον διακόπτη `-c` (έλεγχος σύνταξης μόνο μεταγλώττισης). Ο κύριος κώδικας δεν εκτελείται, ούτε και τα μπλοκ END.
- Όταν η μεταγλώττιση αποτυγχάνει. Ένα πρόγραμμα που δεν μεταγλωττίζεται δεν φτάνει ποτέ στο σημείο όπου θα πυροδοτούνταν τα μπλοκ END.
- Όταν η διεργασία αντικαθιστά τον εαυτό της με `exec`. Η εικόνα έχει χαθεί· δεν απομένει τίποτα για να εκτελέσει το μπλοκ.
- Όταν η διεργασία τερματίζεται από ένα μοιραίο σήμα που δεν παγιδεύετε. Πιάστε το σήμα μόνοι σας (μέσω `%SIG`) αν χρειάζεται να εκτελεστεί ο καθαρισμός.

Ένα μπλοκ END που δημιουργείται μέσα σε ένα string `eval` **δεν** εκτελείται όταν τελειώνει αυτό το `eval`. Καταχωρείται όπως κάθε άλλο μπλοκ END του πακέτου του και εκτελείται με σειρά LIFO λίγο πριν εξέλθει ο διερμηνευτής.

Το ${^GLOBAL_PHASE} και οι φάσεις

Η μεταβλητή μόνο για ανάγνωση ${^GLOBAL_PHASE} ονομάζει την τρέχουσα φάση του διερμηνευτή. Οι τιμές της επιτρέπουν σε κώδικα που εκτελείται σε περισσότερες από μία φάσεις να βρει πού βρίσκεται. Κατά τη διάρκεια ενός μπλοκ CHECK διαβάζεται CHECK· κατά το INIT, INIT· κατά τον συνηθισμένο κώδικα χρόνου εκτέλεσης, RUN· κατά τη διάρκεια ενός μπλοκ END, END. Δείτε το ${^GLOBAL_PHASE} για την πλήρη λίστα τιμών και τη χρονογραμμή των φάσεων.

Μπλοκ defer

Ένα μπλοκ defer είναι ο μικρής κλίμακας ξάδελφος του END. Εκτελείται όταν εξέρχεται η περικλείουσα **εμβέλεια μπλοκ**, για οποιονδήποτε λόγο, αντί όταν εξέρχεται ολόκληρο το πρόγραμμα. Καταφύγετε στο defer για να ζευγαρώσετε την απόκτηση με την αποδέσμευση μέσα σε μια υπορουτίνα ή βρόχο· καταφύγετε στο END για αποδόμηση σε επίπεδο διεργασίας που πρέπει να γίνει μία φορά, στο τέλος τέλος.

Δείτε επίσης

- BEGIN *και* use - το use είναι μια φόρτωση-και-εισαγωγή τυλιγμένη σε BEGIN· ο πιο συνηθισμένος λόγος για να ενδιαφέρεστε για τον χρονισμό της φάσης μεταγλώττισης.
- ${^GLOBAL_PHASE} - διαβάστε την τρέχουσα φάση από μέσα σε οποιοδήποτε μπλοκ για να μάθετε πού βρίσκεστε.
- defer - καθαρισμός με εμβέλεια μπλοκ, το σωστό εργαλείο όταν η εμβέλεια του END σε όλο το πρόγραμμα είναι υπερβολικά ευρεία.
- $? - η κατάσταση εξόδου που ένα μπλοκ END μπορεί να διαβάσει και να ξαναγράψει.
- %SIG - παγιδεύστε τα σήματα που διαφορετικά θα παρέκαμπταν εντελώς τα μπλοκ END.
- exit - αυτό λίγο πριν το οποίο εκτελούνται τα μπλοκ END· η τιμή που φέρει είναι το $? που βλέπει ένα μπλοκ END.
- *Δήλωση* - sub NAME { ... }, ανώνυμες subs, προδηλώσεις, αναφορές κώδικα, το sigil & και τι σας προσφέρει η προδήλωση.
- *Ορίσματα και @_* - ο κανόνας ψευδωνυμοποίησης, το ιδίωμα αποσυσκευασίας, συμβάσεις ονομαστικών ορισμάτων, προεπιλογές shift, η νόμιμη χρήση ψευδωνυμοποίησης στο chomp(@lines), και η παγίδα $_[0] = ...
- *Τιμές επιστροφής* - ρητό return, σιωπηρή τελευταία-έκφραση, ισοπέδωση λίστας κατά την επιστροφή, συντομεύσεις κενού περιβάλλοντος, και η διαφορά μεταξύ return; και return undef;.
- *Εμβέλεια:* my, our, local, state - διάρκεια ζωής pad, closures, η έκπληξη του closure-πάνω-σε-μεταβλητή-βρόχου, πότε το local είναι το κατάλληλο εργαλείο, και τι σημαίνει πραγματικά το state.
- *Πρωτότυπα* - τι κάνουν (ανάλυση στο σημείο κλήσης), τι δεν κάνουν (έλεγχοι χρόνου εκτέλεσης), η διαφορά (\@) έναντι (@), το πρωτότυπο σταθεροποίησης (), και γιατί ο περισσότερος σύγχρονος κώδικας δεν θα πρέπει να τα χρησιμοποιεί.

- *Υπογραφές* - η σύγχρονη σύνταξη από την 5.20+ σταθερή στην 5.36, με παραμέτρους κατά θέση, προαιρετικές, και slurpy· προεπιλεγμένες τιμές· ονομαστικές παράμετροι μέσω slurpy hash· και πώς αλληλεπιδρούν οι υπογραφές με τα πρωτότυπα (χωρίς να τα αντικαθιστούν).
- *Αναδρομή* - όριο βάθους, αμοιβαία αναδρομή, `goto &fn` για κλήσεις ουράς, `__SUB__` για ανώνυμη αυτο-αναφορά, trampolines.
- *Lvalue subs και περιβάλλον* - σημασιολογία `:lvalue`, `wantarray`, ο τριμερής διακόπτης κενό-βαθμωτό-λίστα, και το τυπικό μοτίβο «πρόωρη επιστροφή σε κενό περιβάλλον».
- *Γνωρίσματα* - `:method`, `:lvalue`, `:prototype(...)`, `:const`, γνωρίσματα τρίτων, και πώς συνδυάζονται τα τέσσερα βήματα σχετικά με γνωρίσματα (ανάλυση, καταχώρηση, αποθήκευση, ερώτημα).
- *Phasers* - τα μπλοκ `BEGIN`, `UNITCHECK`, `CHECK`, `INIT` και `END`: πότε πυροδοτείται το καθένα σε σχέση με το όριο μεταγλώττισης/εκτέλεσης, η σειρά τους FIFO/LIFO, το `$?` και το `${^GLOBAL_PHASE}` μέσα στο `END`, και το μπλοκ `defer`.

Ορισμός sub έναντι κλήσης sub

Τα δύο μισά του συστήματος υπορουτινών είναι σε μεγάλο βαθμό ορθογώνια:

- **Ορισμός** είναι μια δήλωση συν ένα σώμα. Η δήλωση καθορίζει το όνομα (ή κάνει τη sub ανώνυμη), το προαιρετικό πρωτότυπο ή υπογραφή, τα προαιρετικά γνωρίσματα. Το σώμα είναι ένα μπλοκ δηλώσεων με δικό του pad.
- **Κλήση** μετατρέπει μια λίστα εκφράσεων στο `@_` του νέου pad (ή σε μεταβλητές υπογραφής), εκτελεί το σώμα, και συλλέγει ό,τι επέστρεψε το σώμα στο περιβάλλον του καλούντος.

Σχεδόν κάθε «λεπτή» ερώτηση σχετικά με υπορουτίνες αφορά *τη μία* πλευρά ή την άλλη. Τα πρωτότυπα επηρεάζουν μόνο τις κλήσεις, όχι τους ορισμούς· οι υπογραφές επηρεάζουν τι βλέπει το σώμα, όχι πώς ο καλών γράφει την κλήση· τα γνωρίσματα επηρεάζουν πώς αντιμετωπίζεται η *ορισμένη* sub από τον χρόνο εκτέλεσης.

Ο κλασικός έναντι του σύγχρονου διαχωρισμός

Η Perl διαθέτει δύο παράλληλους μηχανισμούς περάσματος ορισμάτων:

```
# 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 5.44).

Ο διαχωρισμός μεταξύ κλασικού και σύγχρονου είναι *η πιο χρήσιμη πυξίδα προσανατολισμού* όταν διαβάζετε κώδικα Perl άλλων. Ο περισσότερος παλαιότερος κώδικας χρησιμοποιεί την κλασική μορφή· ο περισσότερος νεότερος χρησιμοποιεί υπογραφές· και οι δύο θα βρίσκονται μπροστά σας σε κάθε μη τετριμμένη βάση κώδικα.

Δείτε επίσης

- `perlop` - τελεστές που αλληλεπιδρούν με subs: `->` για αποστολή μεθόδου και επίκληση αναφοράς κώδικα, `,` (και το fat comma) για κατασκευή ονομαστικών ορισμάτων, `=` για την ανάθεση λίστας που χρησιμοποιείται στο ιδίωμα `my (...) = @_`.
- `perlvar` - ειδικές μεταβλητές που μια sub θα διαβάσει ή θα γράψει: `@_`, `$_`, `$@`, `$AUTOLOAD`.
- `perlfunc` - τα ρήματα του συστήματος υπορουτινών: `sub`, `return`, `caller`, `wantarray`, `prototype`, `goto`, `my`, `our`, `state`, `local`.
- *Αντικειμενοστραφής προγραμματισμός* - subs ως μέθοδοι, ο ρόλος του `bless`, αποστολή μέσω `@ISA`, η σύγχρονη σύνταξη `use experimental 'class'`.

5.1.6 Διαχείριση locale

Ένα *locale* είναι το σύνολο των πολιτισμικών συμβάσεων που ακολουθεί ένα πρόγραμμα όταν ταξινομεί κείμενο, αλλάζει πεζά/κεφαλαία, μορφοποιεί αριθμούς και νομισματικά ποσά, ονομάζει τις ημέρες της εβδομάδας ή γράφει ένα μήνυμα σφάλματος. Ο ίδιος πίνακας λέξεων ταξινομείται με έναν τρόπο υπό ένα σουηδικό locale και με άλλον υπό ένα γερμανικό· ο αριθμός δωδεκάμισι εκτυπώνεται ως `12.5` σε μία χώρα και ως `12,5` σε μια άλλη. Η διαχείριση locale είναι ο τρόπος με τον οποίο η Perl επιτρέπει σε ένα μόνο πρόγραμμα να προσαρμόζεται σε αυτές τις συμβάσεις κατά τον χρόνο εκτέλεσης, αντί να ενσωματώνει σταθερά τους κανόνες μιας χώρας.

Εξ ορισμού η PetaPerl αγνοεί εντελώς το locale: κάθε ενσωματωμένη συνάρτηση συμπεριφέρεται με τον ίδιο τρόπο ανεξάρτητα από το περιβάλλον του χρήστη, χρησιμοποιώντας τις συμβάσεις C/POSIX (συνταξιοθεσία ASCII, `.` ως υποδιαστολή, αγγλικά ονόματα μηνών). Ενεργοποιείτε την επίγνωση locale, μία λεξιλογική εμβέλεια τη φορά, με την pragma `use locale`. Αυτή η σελίδα καλύπτει τι ενεργοποιεί αυτή η pragma, ποιες λειτουργίες επηρεάζει, πώς να ορίσετε και να επιθεωρήσετε το ενεργό locale, και πού αλληλεπιδρούν το locale και το Unicode.

Τι είναι ένα locale

Η βιβλιοθήκη C του συστήματος χωρίζει τις πολιτισμικές συμβάσεις σε ανεξάρτητες *κατηγορίες*, καθεμία ονομασμένη `LC_` ακολουθούμενο από την περιοχή που διέπει. Η ταξινόμηση (`LC_COLLATE`) είναι ξεχωριστή από τους κανόνες πεζών/κεφαλαίων (`LC_CTYPE`), που είναι ξεχωριστή από τη μορφοποίηση αριθμών (`LC_NUMERIC`), και ούτω καθεξής. Ένα locale είναι μια ονομασμένη δέσμη ρυθμίσεων - `de_DE.UTF-8`, `sv_SE.UTF-8`, `C` - και η μετάβαση σε αυτό μπορεί να αλλάξει όλες τις κατηγορίες ταυτόχρονα ή μόνο εκείνες που επιλέγετε. Οι κατηγορίες που τιμά η PetaPerl παρατίθενται στον πίνακα κατηγοριών παρακάτω.

Η pragma `use locale`

Η συμπεριφορά που εξαρτάται από το locale είναι **απενεργοποιημένη εξ ορισμού** και έχει **λεξιλογική εμβέλεια**. Το `use locale` την ενεργοποιεί από εκείνο το σημείο μέχρι το τέλος του περικλείοντος μπλοκ (ή αρχείου)· το `no locale` την απενεργοποιεί ξανά για το υπόλοιπο της δικής του εμβέλειας. Τίποτα εκτός της λεξιλογικής εμβέλειας δεν επηρεάζεται - ένα άρθρωμα που δεν δηλώνει `use locale` διατηρεί την προεπιλεγμένη συμπεριφορά του locale C ακόμη και όταν καλείται από κώδικα που το δήλωσε.

```
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;
}
```

Χωρίς την pragma, τα `sort`, `lc`, `uc`, `printf` και τα υπόλοιπα χρησιμοποιούν σταθερούς κανόνες του locale C ανεξάρτητα από το τι λέει το περιβάλλον. Αυτή είναι μια σκόπιμη προεπιλογή ασφαλείας: ένα πρόγραμμα παράγει την ίδια έξοδο σε κάθε μηχάνημα μέχρι να ζητήσει ρητά να ακολουθήσει τις τοπικές συμβάσεις.

Η pragma ορίζει μια υπόδειξη ανά εντολή, ακριβώς όπως κάνει η upstream Perl, ώστε ένα μόνο αρχείο να μπορεί να αναμειγνύει ελεύθερα περιοχές με επίγνωση locale και περιοχές που το αγνοούν. Η ρύθμιση ταξιδεύει με τη λεξιλογική εμβέλεια, όχι με τη στοίβα κλήσεων.

Κατηγορίες locale

Κάθε κατηγορία ελέγχει μια ξεχωριστή ομάδα λειτουργιών. Το `use locale` τις ενεργοποιεί όλες ταυτόχρονα· επιλέγετε *ποιο* locale είναι ενεργό ανά κατηγορία μέσω της `setlocale`. Η στήλη «επηρεαζόμενες λειτουργίες» συνδέει με τις σελίδες αναφοράς όπου η συμπεριφορά τεκμηριώνεται πλήρως - εκεί βρίσκονται οι κανόνες που εξαρτώνται από το locale για κάθε ενσωματωμένη συνάρτηση.

Κατηγο	Διέπει	Επηρεαζόμενες λειτουργίες
LC_COL	Σειρά ταξινόμησης και σύγκριση συμβολοσειρών	`sort`, οι συγκρίσεις συμβολοσειρών `lt le gt ge cmp`
LC_CTY	Ταξινόμηση χαρακτήρων και αντιστοίχιση πεζών/κεφαλαίων	`lc`, `uc`, `lcfirst`, `ucfirst`, `\w \d \s` και `\b` σε regex
LC_NUM	Υποδιαστολή και ομαδοποίηση ψηφίων	`printf`, `sprintf`, *μετατροπή αριθμού σε συμβολοσειρά*
LC_MON	Συμβάσεις μορφοποίησης νομίσματος	`localeconv` (μόνο επιθεώρηση· καμία ενσωματωμένη συνάρτηση δεν μορφοποιεί νομισματικά ποσά για εσάς)
LC_TIM	Ονόματα ημερών και μηνών, διάταξη ημερομηνίας	`strftime` μέσω `POSIX`, `langinfo`
LC_MES	Γλώσσα των συμβολοσειρών σφαλμάτων του συστήματος	κείμενο της `$!`, συμβολοσειρές μηνυμάτων της `langinfo`

Το `LC_ALL` δεν είναι μια ξεχωριστή κατηγορία αλλά μια συντομογραφία που ορίζει κάθε κατηγορία ταυτόχρονα.

Σημειώστε μια ασυμμετρία που μπερδεύει τον κόσμο: το `LC_NUMERIC` αλλάζει το πώς οι αριθμοί *μορφοποιούνται για έξοδο*, αλλά ο ίδιος ο πηγαίος κώδικας της Perl και η αριθμητική *ανάλυσή* της χρησιμοποιούν πάντα `.` ως διαχωριστή υποδιαστολής. Ένα script που διαβάζει `12,5` από ένα αρχείο γερμανικού locale χρειάζεται ακόμη να μεταφράσει το ίδιο το κόμμα προτού το χειριστεί ως αριθμό - το locale δεν κάνει το `"12,5" + 0` να αποδίδει `12.5`. Δείτε *αριθμοί και συμβολοσειρές*.

Ορισμός του locale

Το `use locale` λέει *ακολούθησε το locale*· δεν λέει *ποιο*. Αυτή η επιλογή προέρχεται εξ ορισμού από το περιβάλλον, και μπορείτε να την ερωτήσετε ή να την παρακάμψετε μέσω του αρθρώματος `POSIX`.

Η `setlocale` επιλέγει το ενεργό locale για μια κατηγορία. Η ιδιωματική κλήση τιμά το περιβάλλον του χρήστη (`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
```

Η `localeconv` επιστρέφει έναν hash που περιγράφει τις ενεργές συμβάσεις `LC_NUMERIC` και `LC_MONETARY` - την υποδιαστολή, τον διαχωριστή χιλιάδων, το σύμβολο νομίσματος και τα υπόλοιπα - ώστε ένα πρόγραμμα να μπορεί να μορφοποιεί τιμές μόνο του όταν καμία ενσωματωμένη συνάρτηση δεν το κάνει απευθείας.

Το `I18N::Langinfo` εκθέτει την `langinfo`, η οποία διαβάζει μεμονωμένα στοιχεία από το τρέχον locale: το τοπικοποιημένο όνομα μιας ημέρας της εβδομάδας ή ενός

μήνα (DAY_1, MON_1), τη συμβολοσειρά μορφής ημερομηνίας, τον χαρακτήρα βάσης αρίθμησης, το σύμβολο νομίσματος. Είναι ο στενός συνοδός της localeconv, ένα στοιχείο τη φορά.

Locale και Unicode

Η διαχείριση locale και το Unicode είναι δύο διαφορετικές απαντήσεις στο «τι είναι ένας χαρακτήρας», και αλληλεπιδρούν με τρόπους που αξίζει να κατανοήσετε προτού τα συνδυάσετε. Η πλήρης πραγμάτευση - πώς συντίθενται το use utf8, τα layers I/O UTF-8 και οι κανόνες LC_CTYPE του locale, και οι παγίδες που προκύπτουν όταν διαφωνούν - βρίσκεται στον οδηγό εκμάθησης Unicode. Ξεκινήστε με τις *συμβολοσειρές και κωδικοποιήσεις Unicode* και το *κεφάλαιο των παγίδων*· το *κεφάλαιο regex και ιδιοτήτων* καλύπτει το πώς συμπεριφέρονται το \w και οι κλάσεις χαρακτήρων υπό κάθε μοντέλο.

Διαρροές locale και ασφάλεια

Ένα locale είναι καθολική κατάσταση επιπέδου διεργασίας στην υποκείμενη βιβλιοθήκη C, οπότε μια κλήση setlocale που γίνεται οπουδήποτε αλλάζει το locale παντού μέσα στη διεργασία - πέρα από τα όρια των αρθρωμάτων, όχι μόνο στην εμβέλεια που την κάλεσε. Δύο συνέπειες προκύπτουν:

- **Το use locale είναι ασφαλές ως προς την εμβέλεια· το setlocale δεν είναι.** Η pragma είναι λεξιλογική και δεν διαρρέει τίποτα. Μια κλήση setlocale είναι μια καθολική παρενέργεια - αν μια βιβλιοθήκη ορίσει το LC_NUMERIC σε ένα locale με κόμμα ως υποδιαστολή και ξεχάσει να το επαναφέρει, κάθε μεταγενέστερο printf στο πρόγραμμα επηρεάζεται. Αποθηκεύστε και επαναφέρετε γύρω από μια προσωρινή αλλαγή:

```
my $saved = setlocale(LC_NUMERIC);
setlocale(LC_NUMERIC, "de_DE.UTF-8");
# ... locale-sensitive work ...
setlocale(LC_NUMERIC, $saved);
```

- **Τα μη έμπιστα ονόματα locale είναι μη έμπιστη είσοδος.** Το να περνάτε μια συμβολοσειρά που παρέχεται από το περιβάλλον κατευθείαν στην setlocale επιτρέπει στο περιβάλλον να κατευθύνει τη μορφοποίηση του προγράμματός σας. Σε ένα ευαίσθητο ως προς την ασφάλεια περιβάλλον, επικυρώστε το όνομα ή καθηλώστε ρητά το locale αντί να καλείτε setlocale(LC_ALL, "").

Παράλληλη εκτέλεση και locale

Η PetaPerl παραλληλοποιεί αυτόματα ορισμένους αριθμητικούς βρόχους σε νήματα εργασίας. Αυτό δεν επηρεάζει ποτέ κώδικα που εξαρτάται από το locale, εκ κατασκευής. Οι λειτουργίες των οποίων τα αποτελέσματα εξαρτώνται από το τρέχον locale - sort με την προεπιλεγμένη συνταξιοθεσία, σύγκριση συμβολοσειρών LC_COLLATE, αναδίπλωση πεζών/κεφαλαίων lc/uc/fc και μορφοποίηση αριθμών LC_NUMERIC - δεν μεταγλωττίζονται ποτέ με JIT ούτε παραλληλοποιούνται αυτόματα· εκτελούνται πάντα σειριακά στον διερμηνευτή έναντι του μοναδικού locale που ισχύει σε όλη τη διεργασία. Η αυτόματη παραλληλοποίηση εκτελεί πάντα μόνο σώματα αριθμητικών βρόχων, οπότε καμία εξαρτώμενη από το locale υπολογιστική εργασία δεν αποστέλλεται σε

νήματα εργασίας. Δεν χρειάζεται να προστατεύετε την κατάσταση locale όταν γράφετε παραλληλοποιήσιμο κώδικα: η παράλληλη διαδρομή απλώς δεν την αγγίζει ποτέ.

Μεταβλητές περιβάλλοντος

Όταν ένα πρόγραμμα καλεί `setlocale(LC_ALL, "")`, η βιβλιοθήκη C διαβάζει το locale από το περιβάλλον με μια σταθερή σειρά προτεραιότητας: το `LC_ALL` παρακάμπτει κάθε μεμονωμένη μεταβλητή `LC_*`, κάθε μεταβλητή `LC_*` ορίζει τη δική της κατηγορία, και το `LANG` παρέχει την προεπιλογή για κάθε κατηγορία που δεν ονομάζεται διαφορετικά. Αυτές οι μεταβλητές βρίσκονται στο `%ENV` όπως κάθε άλλη τιμή περιβάλλοντος, ώστε ένα πρόγραμμα να μπορεί να τις επιθεωρήσει ή να τις προσαρμόσει πριν από την κλήση `setlocale` που τις καταναλώνει.

Μεταβλητή	Αποτέλεσμα
`LC_ALL`	Παρακάμπτει όλες τις κατηγορίες
`LC_COLLATE`, `LC_CTYPE`, …	Ορίζει μία ονομασμένη κατηγορία
`LANG`	Προεπιλογή για κάθε κατηγορία που δεν ορίζεται παραπάνω

Διαφορές από το upstream

- **Μόνο Linux, εγγενώς UTF-8.** Η PetaPerl τρέχει μόνο σε Linux έναντι των locale της βιβλιοθήκης C του συστήματος. Ο μηχανισμός locale ειδικός ανά πλατφόρμα που κουβαλά η upstream Perl για VMS, Windows και Android - τα shims locale του `Win32::`, οι ιδιοτροπίες κατηγοριών του VMS, οι εφεδρείες «τα locale δεν είναι διαθέσιμα σε αυτή την πλατφόρμα» - δεν έχει αντίστοιχο εδώ και απλώς απουσιάζει. Υπάρχει ένα μοντέλο locale: εκείνο του POSIX που παρέχει η `glibc` σας.
- **`LC_NUMERIC` και ανάλυση πηγαίου κώδικα.** Όπως στην upstream, ο διαχωριστής υποδιαστολής που χρησιμοποιείται όταν η Perl *αναλύει* αριθμητικά κυριολεκτικά και μετατροπές συμβολοσειράς σε αριθμό είναι πάντα `.`, ανεξάρτητα από το locale· μόνο η *μορφοποίηση* της εξόδου σέβεται το `LC_NUMERIC`. Η PetaPerl δεν κάνει καμία εξαίρεση σε αυτό.
- **Η αυτόματη παραλληλοποίηση δεν αγγίζει ποτέ κώδικα που εξαρτάται από το locale.** Η PetaPerl παραλληλοποιεί αυτόματα ορισμένους αριθμητικούς βρόχους σε νήματα εργασίας, μια δυνατότητα που η upstream Perl δεν διαθέτει. Επειδή οι λειτουργίες που εξαρτώνται από το locale (συνταξιοθεσία, αναδίπλωση πεζών/κεφαλαίων, μορφοποίηση αριθμών) δεν μεταγλωττίζονται ποτέ με JIT ούτε παραλληλοποιούνται, εκτελούνται πάντα σειριακά έναντι του μοναδικού locale που ισχύει σε όλη τη διεργασία - οπότε η δυνατότητα δεν εισάγει κανένα δικό της ζήτημα ασφάλειας locale. Δείτε *Παράλληλη εκτέλεση και locale* παραπάνω.

Δείτε επίσης

- `perlfunc` - οι ενσωματωμένες συναρτήσεις των οποίων τη συμπεριφορά αλλάζει το locale: `sort`, `lc`, `uc`, `lcfirst`, `ucfirst`, `sprintf`, `printf`.
- `perlop` - οι τελεστές σύγκρισης συμβολοσειρών (`lt`, `cmp`, …) που ταξινομούν υπό το `LC_COLLATE`.

- `perlvar` - το `%ENV` περιέχει τις μεταβλητές `LANG`/`LC_*`· το `$!` φέρει κείμενο σφάλματος μεταφρασμένο σύμφωνα με το locale.
- `POSIX` - οι `setlocale` και `localeconv`, η διεπαφή για την επιλογή και την επιθεώρηση του ενεργού locale.
- `I18N::Langinfo` - η `langinfo`, για την ανάγνωση ενός τοπικοποιημένου στοιχείου (όνομα μήνα, χαρακτήρα βάσης αρίθμησης) τη φορά.
- *Συμβολοσειρές και κωδικοποιήσεις Unicode*
  - πώς σχετίζεται το μοντέλο χαρακτήρων Unicode με το `LC_CTYPE` του locale.

5.1.7 Διαγνωστικά μηνύματα

Όταν ένα πρόγραμμα πάει στραβά, η Perl σας ενημερώνει γι" αυτό. Ορισμένα διαγνωστικά σταματούν το πρόγραμμα (μοιραία σφάλματα τύπου `die`)· άλλα εκτυπώνουν μια γραμμή στο `STDERR` και το αφήνουν να συνεχίσει (προειδοποιήσεις). Αυτή η αναφορά παραθέτει τα διαγνωστικά που εκπέμπει η PetaPerl, ομαδοποιημένα ανάλογα με το από πού προέρχονται, μαζί με την αιτία και τη διόρθωση για το καθένα.

Κάθε καταχώριση δείχνει το **μήνυμα αυτολεξεί** μέσα σε ένα code span - το ακριβές κείμενο που εκτυπώνει ο διερμηνέας, με τα σύμβολα κράτησης θέσης (`%s`, `%d`, μια ονομασμένη μεταβλητή) να εμφανίζονται όπως φαίνονται στην έξοδο. Αναζητήστε σε αυτή τη σελίδα τις αυτούσιες λέξεις που είδατε στο `STDERR` και θα καταλήξετε στην καταχώριση.

Ανάγνωση μιας καταχώρισης

Κάθε καταχώριση ανοίγει με το κείμενο του μηνύματος, έπειτα με ένα σύντομο σήμα ταξινόμησης σε παρενθέσεις, και κατόπιν με την αιτία και τη διόρθωση:

> `` `Illegal division by zero` `` **(F)** Διαιρέσατε με το μηδέν. Προστατεύστε τον διαιρέτη.

Το σήμα ακολουθεί τον ίδιο γραμματικό κωδικό που χρησιμοποιεί ανέκαθεν η Perl:

Κωδι	Σημασία
W	Μια προειδοποίηση - εκτυπώνεται μόνο όταν είναι ενεργοποιημένες οι προειδοποιήσεις. Προαιρετικά περιορισμένη σε μια κατηγορία (εμφανίζεται ως `(W category)`).
D	Μια απαξίωση - μια προειδοποίηση ότι ένα χαρακτηριστικό βαίνει προς κατάργηση.
S	Μια σοβαρή προειδοποίηση - ενεργή από προεπιλογή, αλλά παρ" όλα αυτά μόνο προειδοποίηση.
F	Ένα μοιραίο σφάλμα - το πρόγραμμα τερματίζεται εκτός αν το σφάλμα παγιδευτεί (`eval`, `try`).
P	Ένα εσωτερικό σφάλμα (ένα panic) - δεν θα έπρεπε ποτέ να φτάσει σε χρήστη. Αν δείτε κάποιο, είναι σφάλμα στο pperl.
X	Ένα πολύ μοιραίο σφάλμα - το πρόγραμμα τερματίζει αμέσως.
A	Ένα ξένο σφάλμα - παράγεται από κάτι άλλο εκτός της ίδιας της Perl.

Ενεργοποίηση και σίγαση προειδοποιήσεων

Οι προειδοποιήσεις (οι κλάσεις **W**, **D**, και **S**) ελέγχονται από την pragma `warnings`. Χωρίς αυτήν, εκτυπώνονται μόνο οι σοβαρές (**S**) προειδοποιήσεις:

```
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
```

Η κατηγορία που ονομάζεται σε ένα σήμα (`W category`) είναι η συμβολοσειρά που περνάτε στο `no warnings` για να σιγάσετε μόνο αυτό το διαγνωστικό. Ένας μόνο χειριστής `$SIG{__WARN__}` μπορεί να υποκλέψει κάθε προειδοποίηση κατά τον χρόνο εκτέλεσης· το `$SIG{__DIE__}` κάνει το ίδιο για τα μοιραία σφάλματα. Δείτε `%SIG`.

Μια σημείωση για τη φορητότητα

Η PetaPerl τρέχει μόνο σε Linux και δεν διαθέτει στρώμα XS. Οι οικογένειες διαγνωστικών πλατφόρμας που κουβαλά μια φορητή perl - κωδικοί κατάστασης VMS, κείμενο `GetLastError` του Win32, ειδοποιήσεις «unimplemented on this architecture», αποτυχίες του XS-loader - δεν προκύπτουν επομένως εδώ. Όπου η upstream Perl τεκμηριώνει ένα διαγνωστικό που ενεργοποιείται μόνο σε άλλο λειτουργικό σύστημα ή μέσω XS, η PetaPerl απλώς δεν το εκπέμπει ποτέ.

Τα διαγνωστικά

Γενικά μοιραία σφάλματα

`die` του χρόνου εκτέλεσης που δεν είναι ούτε σφάλματα μεταγλώττισης regex ούτε συντακτικά σφάλματα μεταγλώττισης: επίλυση μεθόδων, κληρονομικότητα, κλήσεις υπορουτινών, γνωρίσματα, και σφάλματα χρήσης σε επίπεδο αρθρώματος. Όλα είναι μοιραία (**F**) και παγιδεύσιμα με `eval` ή `try`.

Κλήσεις υπορουτινών

`Undefined subroutine &%s called`

(F) Μια κλήση ονομάτισε μια υπορουτίνα που δεν υπάρχει κατά τη στιγμή της κλήσης. Εμφανίζεται το πλήρως προσδιορισμένο όνομα. Αιτίες: ένα τυπογραφικό λάθος στο όνομα, ένα απόν `use` του αρθρώματος που την ορίζει, ή κλήση πριν δηλωθεί η `sub` σε ένα περιβάλλον όπου αυτό έχει σημασία. Η μορφή χρόνου εκτέλεσης `` `Undefined subroutine &NAME called at FILE line N.` `` προσθέτει το σημείο κλήσης.

Επίλυση μεθόδων και κληρονομικότητα

`Can't use anonymous symbol table for method lookup`

(F) Η αποστολή μεθόδου έφτασε σε ένα πακέτο χωρίς ονομασμένο πίνακα συμβόλων (ένα ανώνυμο stash). Μια blessed αναφορά πρέπει να δείχνει σε ένα πραγματικό, ονομασμένο πακέτο για να επιλύονται οι κλήσεις μεθόδων.

`Recursive inheritance detected in package '%s'`

(F) Η αλυσίδα `@ISA` μιας κλάσης οδηγεί πίσω στον εαυτό της - ένας κύκλος. Η επίλυση μεθόδων δεν μπορεί να προχωρήσει. Σπάστε τον βρόχο στον γράφο κληρονομικότητας.

`Inconsistent hierarchy during C3 merge of class '%s'`

(F) Υπό τη σειρά επίλυσης μεθόδων C3, η γραμμικοποίηση των γονέων αυτής της κλάσης δεν έχει συνεπή διάταξη. Δύο γονείς διαφωνούν ως προς τη σειρά ενός κοινού προγόνου. Αναδιαρθρώστε την κληρονομικότητα ώστε οι διατάξεις των γονέων να είναι συμβατές, ή επαναφέρετε την κλάση στην προεπιλεγμένη MRO κατά βάθος.

`Invalid mro name: '%s'`

(F) Ένα `use mro '...'` ονομάτισε μια σειρά που δεν υπάρχει. Τα δύο έγκυρα ονόματα είναι `dfs` (η προεπιλεγμένη σειρά κατά βάθος) και `c3`.

Γνωρίσματα

`Unterminated attribute parameter in attribute list`

(F) Ένα γνώρισμα με παράμετρο σε παρενθέσεις - `:Foo(bar` - δεν έκλεισε. Ισορροπήστε τις παρενθέσεις στη λίστα γνωρισμάτων.

Σφάλματα χρήσης αρθρωμάτων

Αυτά προέρχονται από native αρθρώματα όταν καλούνται λανθασμένα. Είναι μοιραία επειδή η κλήση δεν μπορεί να προχωρήσει.

`Usage: POSIX::sprintf(pattern, args...)`

(F) Μια συνάρτηση `POSIX` κλήθηκε με λανθασμένα ορίσματα. Το μήνυμα ονομάζει τη σωστή μορφή κλήσης.

`IO::Socket::%s`

(F) Μια λειτουργία `IO::Socket` απέτυχε· το μήνυμα ονομάζει τη συγκεκριμένη κλήση.

Δείτε επίσης

- `@ISA` - ο πίνακας κληρονομικότητας στο περιεχόμενο του οποίου αναφέρονται τα σφάλματα MRO.
- `mro` - επιλογή και επιθεώρηση της σειράς επίλυσης μεθόδων.
- `bless` - συσχετίζει μια αναφορά με το πακέτο στις μεθόδους του οποίου αναφέρεται το σφάλμα ανώνυμου πίνακα συμβόλων.
- *Σφάλματα τιμών και τύπων* - το «Not a subroutine reference» και τα υπόλοιπα διαγνωστικά για το τι είδους τιμή είναι κάτι.

Σφάλματα μεταγλώττισης regex

Κάθε διαγνωστικό σε αυτή τη σελίδα προέρχεται από τον μεταγλωττιστή μοτίβων. Μια κανονική έκφραση μεταγλωττίζεται πριν εκτελεστεί, και ένα κακοσχηματισμένο μοτίβο τερματίζεται σε εκείνο το σημείο - πριν καν επιχειρηθεί η αντιστοίχιση. Τα σφάλματα είναι όλα μοιραία (**F**)· παγιδεύστε τα με `eval` αν χτίζετε μοτίβα από μη έμπιστη είσοδο:

```
my $re = eval { qr/$user_pattern/ };
die "bad pattern: $@" unless $re;
```

Η upstream Perl προσαρτά ένα επίθημα `marked by <-- HERE in m/.../` που δείχνει την προβληματική θέση. Η PetaPerl αναφέρει τα ίδια σφάλματα· η ακριβής μορφοποίηση του δείκτη θέσης μπορεί να διαφέρει.

Συνηθισμένα δομικά σφάλματα

`Unmatched ) in regex`

(F) Ένα `)` στο μοτίβο δεν έχει αντίστοιχο `(`. Συνήθως ένα τυπογραφικό λάθος ή μια κυριολεκτική παρένθεση χωρίς διαφυγή. Για να ταιριάξετε ένα κυριολεκτικό `)`, διαφύγετέ το ως `\)` ή βάλτε το σε μια κλάση χαρακτήρων `[)]`.

`Quantifier follows nothing in regex`

(F) Ένας ποσοδείκτης (`*`, `+`, `?`, `{n,m}`) εμφανίστηκε χωρίς τίποτα μπροστά του για να επαναλάβει - για παράδειγμα `/*abc/` ή `/(?:)+/` σε μια κενή ομάδα. Ποσοδεικτήστε ένα κυριολεκτικό `*` διαφεύγοντάς το: `/\*/`.

`Unterminated group`

(F) Ένα `(` άνοιξε μια ομάδα που το μοτίβο δεν κλείνει ποτέ. Ισορροπήστε τις παρενθέσεις.

`Unterminated character class`

(F) Ένα `[` άνοιξε μια κλάση χαρακτήρων χωρίς κλείσιμο `]`. Για να ταιριάξετε ένα κυριολεκτικό `]`, τοποθετήστε το πρώτο στην κλάση (`[]x]`) ή διαφύγετέ το (`[\]x]`).

`Unterminated comment group`

(F) Μια ομάδα σχολίου `(?#...)` άνοιξε αλλά δεν έκλεισε με `)`.

`Unterminated conditional pattern`

(F) Μια κατασκευή `(?(condition)yes|no)` έμεινε ανοιχτή.

`Unterminated group name`

(F) Μια κατασκευή ονομασμένης ομάδας (`(?<name>...)`, `\k<name>`) έφτασε στο τέλος του μοτίβου χωρίς να κλείσει τον οριοθέτη του ονόματός της.

`Unrecognized group type`

(F) Οι χαρακτήρες μετά το `(?` δεν σχηματίζουν έναν τύπο ομάδας που γνωρίζει η PetaPerl. Ελέγξτε τη γραφή `(?...)` σε σχέση με την αναφορά regex.

`Unexpected end of character class`

(F) Το μοτίβο τελείωσε στη μέση μιας κλάσης `[...]`.

`Unexpected end of escape sequence`

(F) Μια διαφυγή με ανάστροφη κάθετο διακόπηκε από το τέλος του μοτίβου.

Σφάλματα κλάσης χαρακτήρων και ευρών

`Invalid [] range "%s" in regex`

(F) Ένα εύρος χαρακτήρων μέσα στο `[...]` τρέχει αντίστροφα (από υψηλό σε χαμηλό), όπως στο `[z-a]`. Διατάξτε τα ακραία σημεία από χαμηλό σε υψηλό, ή διαφύγετε το `-` για να το ταιριάξετε κυριολεκτικά.

`Invalid range endpoint in character class`

(F) Το ένα άκρο ενός εύρους `-` δεν είναι ένας μοναδικός χαρακτήρας (για παράδειγμα μια διαφυγή πολλών χαρακτήρων που χρησιμοποιείται ως ακραίο σημείο).

`POSIX class [:%s:] unknown`

(F) Μια κλάση POSIX `[:name:]` ονομάζει μια κλάση που δεν υπάρχει. Τα έγκυρα ονόματα είναι `alpha`, `digit`, `alnum`, `space`, `upper`, `lower`, `punct`, `print`, `graph`, `cntrl`, `xdigit`, `blank`, `word`, και `ascii`. Ένα σκέτο `[:alpha:]` εκτός κάποιου άλλου `[...]` είναι επίσης σφάλμα - οι κλάσεις POSIX λειτουργούν μόνο *μέσα* σε μια κλάση χαρακτήρων: `[[:alpha:]]`.

`Empty POSIX character class name`

(F) Ένα `[::]` χωρίς τίποτα ανάμεσα στις άνω-κάτω τελείες.

Διαφυγές χαρακτήρων ελέγχου και αριθμητικές διαφυγές

`Missing control char`

(F) Μια διαφυγή `\c` δεν ακολουθήθηκε από τον χαρακτήρα που έπρεπε να ελέγξει, όπως σε ένα μοτίβο που τελειώνει σε `\c`.

`Missing control character after \c`

(F) Ίδια αιτία με την παραπάνω, από διαφορετικό σημείο στον μεταγλωττιστή: το `\c` χρειάζεται έναν χαρακτήρα να ακολουθεί.

`Invalid control character '\c%s'`

(F) Ο χαρακτήρας μετά το `\c` δεν είναι ένας που αποδέχεται το `\c`.

`Invalid octal escape`

(F) Μια οκταδική διαφυγή `\0`/`\o` είναι κακοσχηματισμένη.

`Invalid octal digit in \o{}`

(F) Ένα ψηφίο μέσα στο `\o{...}` δεν είναι 0–7.

`Expected '{' after \o`

(F) Το `\o` πρέπει να ακολουθείται από `{`, όπως στο `\o{17}`.

`Expected '}' in \o{}` and `Invalid Unicode code point in \o{}`

(F) Μια διαφυγή `\o{...}` δεν τερματίζεται ή ονομάζει ένα σημείο κώδικα εκτός του έγκυρου εύρους.

`Invalid hex escape` / `Invalid hex digit` / `Expected '}' in hex escape`

(F) Μια διαφυγή `\x` είναι κακοσχηματισμένη: ένα μη δεκαεξαδικό ψηφίο, ή ένα μη τερματισμένο `\x{...}`.

`\g`, `\k`, και αναφορές ομάδων

`\g0 is not allowed` and `\g{0} is not allowed`

(F) Η ομάδα 0 είναι ολόκληρη η αντιστοιχία και δεν μπορεί να γίνει οπισθαναφορά σε αυτήν. Αναφερθείτε σε μια πραγματική ομάδα σύλληψης (`\g1`, `\g{name}`).

`Invalid group number in \g` / `Invalid group number in \g{}`

(F) Ο αριθμός μετά το `\g` δεν είναι έγκυρη αναφορά ομάδας.

`Expected digit or '{' after \g` / `Expected '}' after \g{`

(F) Το `\g` πρέπει να ακολουθείται από ένα ψηφίο, ένα πρόσημο, ή `{name}`.

`Reference to nonexistent group number %d`

(F) Μια οπισθαναφορά τύπου `\g`/`\1` ονομάζει έναν αριθμό ομάδας που δεν περιέχει το μοτίβο.

`Reference to nonexistent group (relative offset %d)`

(F) Μια σχετική οπισθαναφορά (`\g{-2}`) δείχνει πέρα από τις ομάδες που υπάρχουν.

`Reference to nonexistent named group '%s'`

(F) Ένα `\k<name>` ή `\g{name}` αναφέρεται σε μια ονομασμένη ομάδα που δεν ορίστηκε ποτέ.

`Reference to nonexistent or invalid group`

(F) Μια αναφορά ομάδας δεν μπόρεσε να επιλυθεί σε καμία ομάδα.

`Expected '<' or '\'' after \k` / `Expected '}' after \k{`

(F) Μια ονομαστική οπισθαναφορά `\k` πρέπει να γράφεται `\k<name>`, `\k'name'`, ή `\k{name}`.

Ονομασμένες ομάδες

`Empty group name`

(F) Μια ονομασμένη ομάδα (`(?<>...)`) ή αναφορά έχει κενές αγκύλες ονόματος.

`Group name must not start with a digit`

(F) Τα ονόματα ομάδων ακολουθούν τους κανόνες των αναγνωριστικών· δεν μπορούν να ξεκινούν με ψηφίο.

`\N` και διαφυγές Unicode

`\N in character class requires braces`

(F) Μέσα στο `[...]`, το `\N` πρέπει να γράφεται `\N{...}`· η σκέτη μορφή «οποιοσδήποτε χαρακτήρας εκτός newline» δεν επιτρέπεται σε μια κλάση.

`\N{NAME} not supported in character class`

(F) Μια διαφυγή ονομασμένου χαρακτήρα `\N{NAME}` δεν επιτρέπεται μέσα σε μια κλάση χαρακτήρων.

`Expected '}' after \N{` / `Empty \N{U+} escape`

(F) Μια διαφυγή `\N{...}` δεν τερματίζεται ή το `\N{U+}` δεν έχει δεκαεξαδικά ψηφία.

`Invalid hex in \N{U+...}` / `Invalid Unicode code point in \N{U+...}`

(F) Το δεκαεξαδικό μέσα στο `\N{U+...}` είναι κακοσχηματισμένο ή ονομάζει ένα σημείο κώδικα εκτός του εύρους Unicode.

`Invalid Unicode code point`

(F) Μια διαφυγή σημείου κώδικα ονομάζει μια τιμή πάνω από το μέγιστο του Unicode (`0x10FFFF`).

`Empty Unicode property name` / `Unknown Unicode property name`

(F) Μια διαφυγή ιδιότητας `\p{...}` / `\P{...}` είναι κενή ή ονομάζει μια ιδιότητα που δεν αναγνωρίζει η PetaPerl.

Εκτεταμένες κατασκευές `(?...)`

`Invalid flag after (?^` / `Invalid flag character`

(F) Μια ομάδα ορισμού σημαιών `(?flags)` ή `(?^flags:...)` περιέχει έναν χαρακτήρα που δεν είναι έγκυρος τροποποιητής μοτίβου.

`Expected '{' after (??`

(F) Μια κατασκευή αναβεβλημένου υπομοτίβου `(??{ code })` είναι κακοσχηματισμένη.

`Expected '<', '=', or '>' after (?P`

(F) Η κατασκευή τύπου Python `(?P<name>...)` / `(?P=name)` / `(?P>name)` είναι κακοσχηματισμένη.

`Expected DEFINE after (?(` / `Expected ')' after DEFINE`

(F) Ένα μπλοκ `(?(DEFINE)...)` είναι κακοσχηματισμένο.

`Switch condition not recognized in regex` / `Invalid lookaround condition`

(F) Η συνθήκη σε ένα υπό συνθήκη `(?(...)...)` δεν είναι αναγνωρισμένη μορφή (ένας αριθμός ομάδας, μια διεκδίκηση περιβλέπουσας, `R`, ή `DEFINE`).

Οικογένεια `Expected ')' after condition`

(F) Η συνθήκη μιας υπό συνθήκη κατασκευής δεν έκλεισε με `)`. Οι παραλλαγές - `Expected ')' after (?(R...)`, `Expected ')' after code block condition`, `Expected ')' after lookahead condition`, `Expected ')' after lookbehind condition`, `Expected '=' or '!' after (?(<` - αναφέρουν όλες το ίδιο σχήμα σφάλματος σε διαφορετικούς τύπους συνθηκών.

Άγκιστρα ποσοδείκτη

`Expected '}' after quantifier` / `Expected '}' or ',' in quantifier`

(F) Ένας ποσοδείκτης `{n,m}` δεν τερματίζεται ή περιέχει έναν μη αναμενόμενο χαρακτήρα.

Οπίσθια διεκδίκηση

`Variable length lookbehind not implemented`

(F) Μια οπίσθια διεκδίκηση `(?<=...)` / `(?<!...)` της οποίας το περιεχόμενο μπορεί να ταιριάξει μεταβλητό αριθμό χαρακτήρων. Η PetaPerl, όπως και η upstream Perl, απαιτεί οπίσθια διεκδίκηση σταθερού μήκους. Ξαναγράψτε την οπίσθια διεκδίκηση σε σταθερό πλάτος, ή χρησιμοποιήστε `\K` για να σημειώσετε ένα σημείο διατήρησης.

Ρήματα

`Unterminated verb`

(F) Ένα ρήμα ελέγχου `(*VERB)` / `(*VERB:ARG)` δεν έκλεισε.

Γενικής χρήσης

`Regex compilation error at position %d: %s` / `Regular expression error`

(F) Ένα σφάλμα μεταγλώττισης με ετικέτα θέσης για περιπτώσεις που δεν καλύπτουν τα πιο συγκεκριμένα μηνύματα παραπάνω. Το κείμενο που ακολουθεί ονομάζει το υποκείμενο πρόβλημα.

`Unterminated string in %s`

(F) Μια οριοθετημένη κατασκευή μέσα στο μοτίβο έφτασε στο τέλος της εισόδου χωρίς τον τερματικό οριοθέτη της.

Δείτε επίσης

- `qr` - προμεταγλωττίστε ένα μοτίβο ώστε ένα κακοσχηματισμένο να αποτυγχάνει μία φορά, κατά τη μεταγλώττιση, αντί σε κάθε αντιστοίχιση.
- `m` - ο τελεστής αντιστοίχισης· ο ίδιος μεταγλωττιστής απορρίπτει ένα κακοσχηματισμένο μοτίβο εδώ.
- `s` - ο τελεστής αντικατάστασης, του οποίου το μισό μοτίβο μεταγλωττίζεται από την ίδια μηχανή.
- *Προειδοποιήσεις* - τα μη μοιραία διαγνωστικά regex (`Useless use of \E`, `Invalid character in group name`).

Συντακτικά σφάλματα κατά τη μεταγλώττιση

Σφάλματα που εγείρει ο αναλυτής πριν εκτελεστεί το πρόγραμμα. Ένα σφάλμα κατά τη μεταγλώττιση σταματά ολόκληρο το πρόγραμμα: τίποτα δεν εκτελείται, επειδή ο πηγαίος κώδικας δεν μπόρεσε να μετατραπεί σε εκτελέσιμα ops. Σε αντίθεση με τα `die` του χρόνου εκτέλεσης, αυτά δεν παγιδεύονται με `eval BLOCK` - αλλά ένα `eval` συμβολοσειράς πηγαίου κώδικα που αποτυγχάνει να μεταγλωττιστεί τα αναφέρει μέσω της `$@`:

```
my $code = '1 +';
eval $code;
warn "compile failed: $@" if $@;
```

Επίλυση υπορουτίνας κατά τη μεταγλώττιση

`Undefined subroutine &%s`

(F) Ο αναλυτής επέλυσε μια κλήση προς μια ονομασμένη υπορουτίνα που δεν υπάρχει κατά τη μεταγλώττιση. Αυτό είναι το αντίστοιχο κατά τη μεταγλώττιση του σφάλματος χρόνου εκτέλεσης `Undefined subroutine &%s called`: η διαφορά είναι *πότε* εντοπίζεται η απούσα sub. Βεβαιωθείτε ότι έχει γίνει `use` του αρθρώματος που την ορίζει, και ότι το όνομα είναι γραμμένο σωστά και πλήρως προσδιορισμένο όπου το πακέτο έχει σημασία.

Δείτε επίσης

- *Γενικά μοιραία σφάλματα* - η μορφή χρόνου εκτέλεσης του σφάλματος μη ορισμένης υπορουτίνας.
- *Υπορουτίνες* - δήλωση, πρωτότυπα, και οι κανόνες που καθορίζουν πότε επιλύεται μια κλήση.
- `use` - φέρνει το άρθρωμα που ορίζει την υπορουτίνα, κατά τη μεταγλώττιση.

Προειδοποιήσεις

Μη μοιραία διαγνωστικά. Μια προειδοποίηση εκτυπώνει μία γραμμή στο STDERR και το πρόγραμμα συνεχίζει. Οι σοβαρές (**S**) προειδοποιήσεις εκτυπώνονται ακόμη και χωρίς την pragma `warnings`· οι απλές **W** προειδοποιήσεις εκτυπώνονται μόνο όταν είναι ενεργοποιημένες οι προειδοποιήσεις για την κατηγορία τους:

```
use warnings;                    # all categories on
no warnings 'recursion';         # silence just deep-recursion
use warnings FATAL => 'utf8';    # make utf8 warnings die instead
```

Η κατηγορία σε ένα σήμα `(W category)` είναι ακριβώς η συμβολοσειρά που περνάτε στο `no warnings`.

Βαθιά αναδρομή

`Deep recursion on subroutine "%s"`

(W recursion) Μια υπορουτίνα έχει καλέσει τον εαυτό της (απευθείας ή μέσω αλυσίδας) σε βάθος μεγαλύτερο από εκατό επίπεδα. Αυτό συνήθως είναι μια ανεξέλεγκτη αναδρομή - μια βασική περίπτωση που δεν ενεργοποιείται ποτέ - παρά πραγματικά βαθιά εργασία. Το μήνυμα ονομάζει την υπορουτίνα.

```
use warnings;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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.
```

Διορθώστε τη βασική περίπτωση. Αν η αναδρομή πράγματι προορίζεται να τρέξει σε βάθος, σιγάστε την κατηγορία για αυτή την εμβέλεια:

```
no warnings 'recursion';
```

Ευρείς χαρακτήρες

`Wide character in %s`

(S) Μια συμβολοσειρά που περιέχει έναν χαρακτήρα πάνω από το σημείο κώδικα 255 στάλθηκε σε μια λειτουργία προσανατολισμένη στα bytes - συχνότερα `print` σε ένα filehandle χωρίς στρώμα `:encoding`. Η Perl εκπέμπει τα bytes της εσωτερικής μορφής UTF-8 και προειδοποιεί, επειδή στο ρεύμα εξόδου δεν δηλώθηκε ποτέ ότι θα μετέφερε Unicode.

```
use warnings;
print "café\x{2603}\n";          # Wide character in print at ...␣
↪line 2.
```

Η διόρθωση είναι να δηλώσετε στο ρεύμα την κωδικοποίησή του:

```
binmode STDOUT, ':encoding(UTF-8)';
print "café\x{2603}\n";          # no warning; correct UTF-8 bytes
```

Αυτή είναι μια σοβαρή **(S)** προειδοποίηση: εκτυπώνεται ακόμη και χωρίς `use warnings`, επειδή η σιωπηρή εκπομπή ακατέργαστων εσωτερικών bytes είναι σχεδόν πάντα σφάλμα.

Άχρηστες κατασκευές

`Useless use of \E`

(W regexp) Ένα `\E` σε ένα μοτίβο ή συμβολοσειρά κλείνει ένα εύρος πεζών-κεφαλαίων/εισαγωγικών `\U`, `\L`, `\Q` που δεν άνοιξε ποτέ. Το `\E` δεν έχει τίποτα να τερματίσει. Αφαιρέστε το, ή προσθέστε το ανοίγον `\U`/`\L`/`\Q` που υποτίθεται ότι θα έκλεινε.

Ονόματα ομάδων

`Invalid character in group name`

(W) Μια ονομασμένη ομάδα regex (`(?<name>...)`) περιέχει έναν χαρακτήρα που δεν επιτρέπεται σε όνομα ομάδας. Τα ονόματα ομάδων ακολουθούν τους κανόνες των

αναγνωριστικών - γράμματα, ψηφία, και κάτω παύλα, χωρίς να ξεκινούν με ψηφίο. Το μοτίβο μεταγλωττίζεται και πάλι, αντιμετωπίζοντας την κατασκευή επιεικώς, αλλά το όνομα είναι ύποπτο.

Κωδικοποίηση

`Cannot find encoding "%s"`

(S) Ένα στρώμα `:encoding(...)` ή μια κλήση Encode ονόματισε μια κωδικοποίηση που δεν είναι διαθέσιμη. Ελέγξτε τη γραφή σε σχέση με τις κωδικοποιήσεις που υποστηρίζει το Encode (`UTF-8`, `Latin-1`, `ASCII`, και τις υπόλοιπες).

Γενικής χρήσης

`Warning: something's wrong`

(W) Μια γενική προειδοποίηση που εγείρει η παραγωγή κώδικα για μια συνθήκη που δεν έχει πιο συγκεκριμένο μήνυμα. Το γύρω περιβάλλον ονομάζει το τι την προκάλεσε.

Δείτε επίσης

- `warnings` - η pragma που ενεργοποιεί, οριοθετεί, και κατηγοριοποιεί κάθε **W** προειδοποίηση.
- `$SIG{__WARN__}` - υποκλέψτε προειδοποιήσεις κατά τον χρόνο εκτέλεσης αντί να τις αφήνετε να φτάσουν στο STDERR.
- *Σφάλματα μεταγλώττισης regex* - τα μοιραία διαγνωστικά regex, σε αντιδιαστολή με αυτά εδώ που είναι σε επίπεδο προειδοποίησης.
- Encode - τα ονόματα κωδικοποιήσεων πίσω από την προειδοποίηση «Cannot find encoding».

Σφάλματα τιμών και τύπων

Αυτά τα διαγνωστικά ενεργοποιούνται όταν μια τιμή δεν είναι το είδος που χρειάζεται η λειτουργία: ένα απλό βαθμωτό εκεί όπου αναμενόταν αναφορά, ένα μηδέν εκεί όπου αναμενόταν διαιρέτης, μια τιμή μόνο για ανάγνωση εκεί όπου αναμενόταν εγγράψιμη. Όλα είναι μοιραία (**F**) εκτός αν σημειώνεται διαφορετικά.

Αριθμητική

`Illegal division by zero`

(F) Το δεξί σκέλος του `/` ή του `%` αποτιμήθηκε σε μηδέν. Η αριθμητική διαίρεση και το υπόλοιπο δεν έχουν αποτέλεσμα στο μηδέν, οπότε η Perl τερματίζεται αντί να επιστρέψει άπειρο ή NaN. Προστατεύστε τον διαιρέτη:

```
my $rate = $count ? $total / $count : 0;
```

`Illegal modulus zero`

(F) Ο δεξιός τελεστέος του % ήταν μηδέν. Ίδια αιτία και ίδια διόρθωση με τη διαίρεση με το μηδέν, για τον τελεστή υπολοίπου.

Αναφορές

`Not a reference`

(F) Επιχειρήθηκε αποαναφορά (`@$x`, `%$x`, `$$x`, `$x->[0]`, `$x->{k}`) σε μια τιμή που δεν είναι αναφορά. Συχνότερα το `$x` είναι `undef` ή μια απλή συμβολοσειρά. Ελέγξτε το πρώτα:

```
die "expected an arrayref" unless ref $x eq 'ARRAY';
```

`Not a subroutine reference`

(F) Μια τιμή που χρησιμοποιήθηκε εκεί όπου απαιτούνταν αναφορά κώδικα (για παράδειγμα το δεύτερο όρισμα σε μια συνάρτηση ανώτερης τάξης) δεν είναι αναφορά `CODE`. Τα `` `set_subname: not a code reference` `` και `` `set_prototype: not a code reference` `` είναι το ίδιο σφάλμα όπως αναφέρεται από το `Sub::Util`, και το `` `PadWalker: cv is not a code reference` `` είναι η μορφή του `PadWalker`.

Περιορισμένα hashes

Ένα hash κλειδωμένο με το `lock_keys` του `Hash::Util` απορρίπτει την πρόσβαση σε κλειδιά εκτός του επιτρεπόμενου συνόλου του.

`Attempt to access disallowed key '%s' in a restricted hash`

(F) Διαβάσατε ή γράψατε ένα κλειδί που δεν επιτρέπει το περιορισμένο hash. Είτε προσθέστε το κλειδί στο επιτρεπόμενο σύνολο, είτε ξεκλειδώστε το hash.

`Attempt to delete readonly key '%s' from a restricted hash`

(F) Επιχειρήσατε να κάνετε `delete` ένα κλειδί που το hash έχει κλειδώσει έναντι διαγραφής.

Τιμές μόνο για ανάγνωση

`Modification of a read-only value attempted`

(F) Μια ανάθεση ή μια επιτόπια λειτουργία στόχευσε κάτι που δεν μπορεί να αλλάξει: ένα κυριολεκτικό, μια σταθερά, μια τιμή που πέρασε ο καλών και έγινε μόνο για ανάγνωση, ή μια τιμή `Readonly`/`const`. Μια συνηθισμένη έκπληξη είναι η τροποποίηση του `$_` μέσα σε ένα μπλοκ `map`/`grep` όταν τα στοιχεία της λίστας προέλευσης είναι μόνο για ανάγνωση.

Filehandles

`Bad filehandle: %s`

(F) Σε μια λειτουργία I/O δόθηκε κάτι που δεν είναι χρησιμοποιήσιμο filehandle - μια μη ορισμένη τιμή, ένα κλειστό handle, ή ένα bareword που δεν ονομάζει κανένα ανοιχτό handle. Ανοίξτε το handle (και ελέγξτε το αποτέλεσμα) πριν το χρησιμοποιήσετε:

```
open my $fh, '<', $path or die "open $path: $!";
```

Μνήμη

`Out of memory`

(F) Μια δέσμευση μνήμης απέτυχε. Συνήθως το πρόγραμμα ζήτησε μια απίθανα μεγάλη συμβολοσειρά, πίνακα, ή hash - ελέγξτε το μέγεθος που ζητάτε πριν υποθέσετε ότι φταίει το μηχάνημα.

Δείτε επίσης

- *Αναφορές* - τι είναι μια αναφορά και πώς λειτουργεί η αποαναφορά· το υπόβαθρο για τα σφάλματα «Not a reference».
- `ref` - ελέγξτε τι είδους αναφορά κρατά μια τιμή πριν την αποαναφέρετε.
- `Hash::Util` - τα `lock_keys`, `unlock_keys`, και ο υπόλοιπος μηχανισμός των περιορισμένων hashes.
- *Γενικά μοιραία σφάλματα* - σφάλματα επίλυσης μεθόδων και κληρονομικότητας που αφορούν επίσης το τι είναι μια τιμή.

Εσωτερικά σφάλματα

Ένα διαγνωστικό **P** είναι panic: ένας έλεγχος που δεν θα έπρεπε ποτέ να αποτύχει σε σωστή λειτουργία απέτυχε. Αυτά τα μηνύματα δεν είναι κάτι που μπορεί να προκαλέσει ο κώδικάς σας σε Perl μέσω συνηθισμένων λαθών - σηματοδοτούν ένα σφάλμα στην ίδια την PetaPerl. Αν δείτε κάποιο, αξίζει μια αναφορά σφάλματος. Δείτε *Αποποίηση ευθύνης & αναφορές σφαλμάτων*.

Δεν υπάρχει τίποτα να διορθωθεί από την πλευρά της Perl. Καταγράψτε το μήνυμα, την είσοδο που το προκάλεσε, και την έκδοση του `pperl`, και υποβάλετέ τα.

`Unknown error`

(P) Ένα εφεδρικό μήνυμα για μια διαδρομή σφάλματος που δεν παρήγαγε κανένα συγκεκριμένο διαγνωστικό - εμφανίζεται συχνότερα από ένα native άρθρωμα (όπως το GMP bignum backend) όταν μια υποκείμενη βιβλιοθήκη επέστρεψε έναν κωδικό αποτυχίας χωρίς κείμενο. Αντιμετωπίστε το ως εσωτερικό σφάλμα και αναφέρετέ το μαζί με τη λειτουργία που το προκάλεσε.

Δείτε επίσης

- *Αποποίηση ευθύνης & αναφορές σφαλμάτων* - πώς να υποβάλετε ένα σφάλμα του pperl, και τι να συμπεριλάβετε.
- *Διαγνωστικά ειδικά για το pperl* - άλλα μηνύματα χωρίς αντίστοιχο στην upstream Perl.

Διαγνωστικά ειδικά για το pperl

Μηνύματα που εκπέμπει η PetaPerl και τα οποία δεν έχουν αντίστοιχο στην upstream Perl. Τα περισσότερα αφορούν τη λειτουργία daemon - το χαρακτηριστικό μόνιμης διεργασίας του pperl, που δεν έχει ανάλογο σε μια τυπική perl - και μερικά native αρθρώματα με συμπεριφορά ειδική για το pperl.

Λειτουργία daemon

Το `pperl` μπορεί να τρέξει ένα σενάριο ως μόνιμο daemon και να δρομολογεί αιτήματα προς αυτό, κάτι που αποφεύγει το κόστος εκκίνησης του διερμηνέα σε κάθε κλήση. Τα διαγνωστικά του daemon αναφέρουν προβλήματα με αυτό το κανάλι.

`Daemon already running for this script (socket: %s)`

(F)

Ζητήθηκε εκκίνηση `--daemon` για ένα σενάριο που έχει ήδη έναν daemon σε εκτέλεση. Εμφανίζεται η διαδρομή της υποδοχής. Σταματήστε πρώτα τον υπάρχοντα daemon, ή συνδεθείτε σε αυτόν αντί να εκκινήσετε νέο.

`Cannot connect to daemon (%s): %s`

(F)

Ένας πελάτης δεν μπόρεσε να φτάσει την υποδοχή του daemon. Το μήνυμα συνεχίζει με μία από δύο υποδείξεις ανάλογα με το περιβάλλον - `Start with: pperl --daemon <script>` όταν δεν φαίνεται να εκτελείται κανένας daemon, ή `Is the daemon running?` όταν η υποδοχή υπάρχει αλλά η σύνδεση απέτυχε. Εκκινήστε τον daemon, ή ελέγξτε ότι η διαδρομή της υποδοχής ταιριάζει.

Native αρθρώματα

`ERRNO hash is read only!`

(F)

Κώδικας επιχείρησε να αναθέσει στο `%!` (το hash συμβολικών σφαλμάτων `Errno`) ή σε ένα από τα κλειδιά του. Αυτό το hash συμπληρώνεται από τον χρόνο εκτέλεσης και αντικατοπτρίζει το τρέχον `errno`· δεν είναι εγγράψιμο. Διαβάστε το (`$!{ENOENT}`) αντί να αναθέτετε σε αυτό.

`Hash has key '%s' which is not in the new key set`

(F)

Από το `Hash::Util`: μια κλήση `lock_keys` παρείχε ένα ρητό σύνολο κλειδιών που δεν περιλαμβάνει ένα κλειδί που περιέχει ήδη το hash. Το κλείδωμα δεν μπορεί να προχωρήσει επειδή αυτό το υπάρχον κλειδί θα γινόταν απρόσιτο. Είτε συμπεριλάβετε το υπάρχον κλειδί στο νέο σύνολο, είτε διαγράψτε το πριν το κλείδωμα.

`Time::HiRes::usleep(%s): negative time not invented yet`

(F)

Σε μια συνάρτηση αναμονής του `Time::HiRes` δόθηκε αρνητική διάρκεια. Ο χρόνος κυλά προς τα εμπρός· περάστε έναν μη αρνητικό αριθμό μικροδευτερολέπτων (ή νανοδευτερολέπτων για το `nanosleep`). Η μορφή `nanosleep` αναφέρει το ίδιο με το δικό της όνομα.

`%s version %s required--this is only version %s`

(F)

Ένα `use Module VERSION` (ή ένας ρητός έλεγχος έκδοσης) απαίτησε νεότερη έκδοση ενός αρθρώματος από αυτή που είναι εγκατεστημένη. Το μήνυμα ονομάζει το άρθρωμα, την έκδοση που ζητήθηκε, και την έκδοση που υπάρχει. Εγκαταστήστε ή φορτώστε την απαιτούμενη έκδοση.

Δείτε επίσης

- *Εσωτερικά σφάλματα* - τα panics που σημαίνουν σφάλμα του pperl αντί για σφάλμα χρήσης.
- *Αποποίηση ευθύνης & αναφορές σφαλμάτων* - πώς αποκλίνει το pperl από την τυπική perl, και πού να αναφέρετε προβλήματα.
- `Errno` - το άρθρωμα πίσω από το `%!`, το read-only hash συμβολικών errno.
- `Hash::Util` - το `lock_keys` και ο μηχανισμός των περιορισμένων hashes.
- *Γενικά μοιραία σφάλματα* - επίλυση μεθόδων, αναφορές, αριθμητική, περιορισμένα hashes, filehandles, και τα υπόλοιπα `die` του χρόνου εκτέλεσης που δεν αφορούν ειδικά regex ή τον αναλυτή.
- *Σφάλματα μεταγλώττισης regex* - οτιδήποτε απορρίπτει ο μεταγλωττιστής μοτίβων, από το `Unmatched )` μέχρι τις κακοσχηματισμένες κατασκευές `\g`, `\N`, `\o`, και `(?...)`.
- *Συντακτικά σφάλματα κατά τη μεταγλώττιση* - σφάλματα που εγείρουν ο αναλυτής και η παραγωγή κώδικα πριν εκτελεστεί το πρόγραμμα.
- *Προειδοποιήσεις* - τα μη μοιραία διαγνωστικά: βαθιά αναδρομή, ευρείς χαρακτήρες, άχρηστες κατασκευές.
- *Σφάλματα τιμών και τύπων* - διαγνωστικά για το τι είναι και τι δεν είναι μια τιμή: όχι αναφορά, όχι αναφορά κώδικα, διαίρεση με το μηδέν, πρόσβαση σε περιορισμένο hash.

- *Εσωτερικά σφάλματα* - τα panics που σηματοδοτούν ένα σφάλμα στην ίδια την pperl.
- *Διαγνωστικά ειδικά για το pperl* - μηνύματα που εκπέμπει η PetaPerl και τα οποία δεν έχουν αντίστοιχο στην upstream Perl.

5.1.8 Ασφάλεια και taint mode

Η ασφάλεια σε ένα πρόγραμμα Perl αφορά κυρίως το να μην εμπιστεύεσαι δεδομένα που προήλθαν από έξω από το πρόγραμμα: ορίσματα γραμμής εντολών, το περιβάλλον, περιεχόμενα αρχείων, είσοδο από το δίκτυο. Το επικίνδυνο βήμα δεν είναι η ανάγνωση αυτών των δεδομένων αλλά η *ενέργεια* βάσει αυτών. Ένα όνομα αρχείου που φτάνει από μια φόρμα web είναι αβλαβές μέχρι να φτάσει στην `open`· μια μεταβλητή περιβάλλοντος είναι αβλαβής μέχρι να κατευθύνει μια κλήση `system`. Αυτή η σελίδα περιγράφει τον μηχανισμό που σας δίνει η PetaPerl για να κάνετε αυτό το όριο ρητό: το taint mode. Εξηγεί τι είναι το taint mode, πότε ενεργοποιείται, τι σημειώνει ως μη έμπιστο, πώς καθαρίζονται τα δεδομένα και πού διαφέρουν οι δύο runtimes.

Taint mode

PetaPerl runs the perl5 5.44 core, so taint mode is the real perl5 taint machinery, not an approximation. Every check perl5 performs, PetaPerl performs.

Το taint mode ενεργοποιείται αυτόματα όταν η PetaPerl ανιχνεύσει ότι εκτελείται με διαφορετικά πραγματικά και ενεργά IDs χρήστη ή ομάδας, που είναι ο τρόπος με τον οποίο εκτελείται ένα setuid (Unix mode `04000`) ή setgid (mode `02000`) σενάριο. Μπορείτε επίσης να το ενεργοποιήσετε ρητά με τον διακόπτη γραμμής εντολών `-T`. Το `-T` συνιστάται έντονα για κάθε πρόγραμμα που εκτελείται για λογαριασμό κάποιου άλλου, με κλασική περίπτωση ένα σενάριο CGI. Μόλις ενεργοποιηθεί το taint mode, παραμένει ενεργό για το υπόλοιπο του προγράμματος· δεν μπορεί να απενεργοποιηθεί από μέσα.

Ο διακόπτης πρέπει να εμφανίζεται αρκετά νωρίς ώστε ο διερμηνευτής να τον δει πριν αρχίσει να αναλύει το πρόγραμμά σας. Στη γραμμή `#!` ενός σεναρίου αυτό γίνεται αυτόματα. Στη γραμμή εντολών, βάλτε τον ανάμεσα στους πρώτους διακόπτες. Ο σχετικός διακόπτης `-t` ενεργοποιεί τους ίδιους ελέγχους αλλά υποβαθμίζει τα μοιραία σφάλματα σε προειδοποιήσεις, κάτι που είναι χρήσιμο όταν προσθέτετε εκ των υστέρων το taint mode σε ένα υπάρχον πρόγραμμα: βλέπετε κάθε σημείο που *θα* αποτύγχανε χωρίς το πρόγραμμα να πεθαίνει στο πρώτο.

Η μεταβλητή μόνο για ανάγνωση `${^TAINT}` αναφέρει την τρέχουσα κατάσταση:

`${^TAINT}`	Σημασία
`1`	Taint mode ενεργό, μοιραίο (`-T`)
`-1`	Έλεγχοι taint ενεργοί, μόνο προειδοποίηση (`-t`)
`0`	Taint mode ανενεργό

Τι σημειώνεται ως tainted

Όταν το taint mode είναι ενεργό, κάθε τιμή που προήλθε από έξω από το πρόγραμμα σημειώνεται ως tainted, και αυτή η σήμανση διαδίδεται μέσα από κάθε έκφραση που αγγίζει η τιμή. Οι tainted πηγές είναι:

- τα ορίσματα γραμμής εντολών (@ARGV) και το περιβάλλον (%ENV)·
- όλη η είσοδος από αρχεία·
- τα αποτελέσματα των readdir και readlink, η μεταβλητή που διαβάζει η shmread, τα μηνύματα από τη msgrcv, και τα πεδία password, gcos και shell που επιστρέφουν οι κλήσεις getpw*·
- τα δεδομένα locale (δείτε perllocale).

Τα tainted δεδομένα δεν επιτρέπεται να χρησιμοποιηθούν, άμεσα ή έμμεσα, σε καμία λειτουργία που ξεκινά ένα sub-shell ή που τροποποιεί ένα αρχείο, κατάλογο ή διεργασία. Η χρήση τους σε μια τέτοια λειτουργία είναι μοιραία, με μήνυμα όπως Insecure dependency in %s ή Insecure $ENV{PATH}. Υπάρχουν τρεις εξαιρέσεις:

- τα ορίσματα προς την print και τη syswrite **δεν** ελέγχονται·
- οι συμβολικές κλήσεις μεθόδων ($obj->$method(@args)) και οι συμβολικές αναφορές υπορουτινών ($foo->(@args)) **δεν** ελέγχονται, οπότε επικυρώστε αυτά τα ονόματα μόνοι σας αν μπορούν να προέλθουν από έξω·
- τα κλειδιά hash **δεν** είναι ποτέ tainted.

Για λόγους αποδοτικότητας, το taint παρακολουθείται συντηρητικά: αν οποιοδήποτε μέρος μιας έκφρασης είναι tainted, ολόκληρη η έκφραση αντιμετωπίζεται ως tainted, ακόμη και όταν η tainted τιμή δεν επηρεάζει στην πραγματικότητα το αποτέλεσμα. Το taint προσκολλάται σε μεμονωμένες βαθμωτές τιμές, οπότε κάποια στοιχεία ενός πίνακα ή hash μπορεί να είναι tainted ενώ άλλα όχι.

```
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)
```

Η μία δομική εξαίρεση στο taint ολόκληρης της έκφρασης είναι ο τριαδικός τελεστής ?:. Επειδή

```
my $result = $tainted ? "untainted" : "also untainted";
```

είναι απλώς μια υπό συνθήκη επιλογή ανάμεσα σε δύο σταθερές, η $result δεν είναι tainted παρότι η συνθήκη είναι.

Ξέπλυμα και ανίχνευση tainted δεδομένων

Ο **μόνος** τρόπος για να καθαρίσετε το taint από μια τιμή είναι να εξαγάγετε μια υποσυμβολοσειρά της μέσω μιας σύλληψης κανονικής έκφρασης. Η PetaPerl υποθέτει ότι αν μπείτε στον κόπο να γράψετε ένα μοτίβο και να βγάλετε το $1, έχετε σκεφτεί τι είστε διατεθειμένοι να αποδεχτείτε. Καθαρίστε το taint επικυρώνοντας έναντι μιας λευκής λίστας αποδεκτών χαρακτήρων, ποτέ προσπαθώντας να αφαιρέσετε τους κακούς:

```
if ($data =~ /^([-\@\w.]+)$/) {
    $data = $1;             # $data is now untainted
} else {
    die "rejected suspicious input";
}
```

Αυτό είναι εύλογα ασφαλές επειδή το \w, μια τελεία, μια παύλα και ένα at δεν φέρουν ειδική σημασία για ένα shell. Ένα μοτίβο όπως /(.+)/ θα «ξέπλενε» την τιμή ενώ θα αποδεχόταν οτιδήποτε, κάτι που αναιρεί τον σκοπό. Να είστε σκόπιμοι ως προς το μοτίβο.

Υπάρχει μια επιφύλαξη ως προς το locale. Υπό use locale το σύνολο των χαρακτήρων που ταιριάζει το \w αντλείται το ίδιο από το locale, που είναι εξωτερικά δεδομένα τα οποία η PetaPerl δεν εμπιστεύεται, οπότε μια αντιστοιχία \w υπό use locale **δεν** καθαρίζει το taint. Αν χρειάζεται να ξεπλύνετε με \w σε ένα πρόγραμμα με επίγνωση locale, βάλτε no locale πριν την αντιστοιχία στο ίδιο μπλοκ.

Για να ελέγξετε αν μια τιμή είναι tainted, χρησιμοποιήστε την tainted από το άρθρωμα Scalar::Util. Υπό -T στο p5 runtime αναφέρει σωστά την κατάσταση taint του ορίσματός της.

Καθαρισμός του %ENV και του $ENV{PATH}

Ένα tainted PATH είναι η πιο συνηθισμένη έκπληξη. Επειδή η PetaPerl δεν μπορεί να γνωρίζει ότι ένα πρόγραμμα που εκτελεί δεν θα συμβουλευτεί το ίδιο το PATH, αρνείται να ξεκινήσει μια υποδιεργασία ενόσω το $ENV{PATH} είναι tainted, με Insecure $ENV{PATH}. Κάθε κατάλογος στο path πρέπει να είναι απόλυτος και εγγράψιμος μόνο από τον ιδιοκτήτη και την ομάδα του· ένας εγγράψιμος κατάλογος δίνει Insecure directory in %s. Σημειώστε ότι ένα κενό συστατικό του path αντιμετωπίζεται ως . (ο τρέχων κατάλογος) και επίσης πυροδοτεί το μήνυμα.

Το PATH δεν είναι ο μόνος ένοχος. Επειδή ένα shell μπορεί να συμβουλευτεί τα IFS, CDPATH, ENV και BASH_ENV, η PetaPerl απαιτεί αυτά να είναι κενά ή μη tainted πριν ξεκινήσει μια υποδιεργασία, αναφέροντας Insecure $ENV{%s} διαφορετικά. Ορίστε ένα γνωστά-καλό PATH και διαγράψτε τις μεταβλητές του shell νωρίς:

```
$ENV{PATH} = '/bin:/usr/bin';
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};
```

Όταν πρέπει να ανοίξετε ένα αρχείο ή σωλήνα από ένα προνομιούχο (setuid ή setgid) πρόγραμμα, το ασφαλές μοτίβο είναι να κάνετε fork ένα παιδί, να αφήσετε το παιδί να ρίξει τα προνόμιά του πίσω στον πραγματικό χρήστη και ομάδα, και να αφήσετε το μη προνομιούχο παιδί να κάνει τη δουλειά και να περάσει το αποτέλεσμα πίσω μέσω ενός σωλήνα:

```
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
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    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: $!";
}
```

Σημειώστε ότι η `exec` καλείται με μια λίστα, ποτέ με μια μεμονωμένη συμβολοσειρά, οπότε δεν εμπλέκεται κανένα shell για να επανεπεκτείνει τα ορίσματα. Αυτός είναι ο ασφαλέστερος τρόπος για να επικαλεστείτε μια εξωτερική εντολή. Η ίδια στρατηγική fork-and-drop λειτουργεί για τη `glob` και τα backticks, που (σε αντίθεση με τη `system` και την `exec`) δεν προσφέρουν δική τους σύμβαση κλήσης σε μορφή λίστας.

Taint mode και `@INC`

Υπό `-T`, η PetaPerl αγνοεί τις μεταβλητές περιβάλλοντος `PERL5LIB`, `PERLLIB` και `PERL_USE_UNSAFE_INC`. Αγνοούνται επειδή είναι εύκολο να οριστούν χωρίς να το παρατηρήσει ο χρήστης, κάτι που θα επέτρεπε στο περιβάλλον να εισάγει σιωπηλά καταλόγους βιβλιοθηκών. Μπορείτε ακόμη να ρυθμίσετε το path αναζήτησης αρθρωμάτων με τον ορατό διακόπτη `-I` ή την pragma `lib` (`-Mlib=/path`), επειδή αυτά εμφανίζονται καθαρά στη γραμμή εντολών.

Ο τρέχων κατάλογος `.` **δεν** βρίσκεται στο προεπιλεγμένο `@INC`, είτε με taint mode είτε χωρίς. (Το `PERL_USE_UNSAFE_INC=1` θα τον ξαναπρόσθετε, αλλά αυτή η μεταβλητή αγνοείται η ίδια υπό `-T`, οπότε το `.` μένει εκτός `@INC` για κάθε tainted πρόγραμμα.)

Σενάρια set-id και ο αγώνας του shebang

Η PetaPerl ανιχνεύει τη συνθήκη setuid/setgid (το πραγματικό ID χρήστη ή ομάδας διαφέρει από το ενεργό) και ενεργοποιεί το taint mode αναλόγως· η διατύπωση «while running setuid» / «while running setgid» στα μηνύματα ανασφάλειας προέρχεται από αυτήν την ανίχνευση. Η PetaPerl **δεν** διανέμει, ωστόσο, μια setuid εγκατάσταση `pperl` ούτε κανέναν βοηθό set-id. Αν χρειάζεστε ένα προνομιούχο πρόγραμμα Perl, η υποστηριζόμενη προσέγγιση στο Linux είναι ένα μικρό αποκλειστικό setuid wrapper που εξυγιαίνει το περιβάλλον και κατόπιν κάνει exec τον διερμηνευτή σε ένα σταθερό, μη εγγράψιμο path σεναρίου, αντί να στηρίζεται απευθείας σε ένα setuid σενάριο. Ο κλασικός αγώνας του shebang, όπου ο πυρήνας και ο διερμηνευτής ανοίγουν το αρχείο του σεναρίου σε δύο διαφορετικές στιγμές και ένας επιτιθέμενος ανταλλάσσει το αρχείο ενδιάμεσα, είναι ιδιότητα του χειρισμού set-id-σεναρίων από τον πυρήνα, όχι του διερμηνευτή, και η προσέγγιση του wrapper το παρακάμπτει.

Επιθέσεις αλγοριθμικής πολυπλοκότητας

Κάποιοι εσωτερικοί αλγόριθμοι μπορούν να οδηγηθούν να καταναλώσουν δυσανάλογο χρόνο ή μνήμη μέσω προσεκτικά επιλεγμένης εισόδου, κάτι που είναι ένας φορέας άρνησης υπηρεσίας ανεξάρτητος από το taint mode. Οι κατηγορίες απειλής:

- **Συγκρούσεις hash.** Ένας επιτιθέμενος που μπορεί να προβλέψει το hash seed μπορεί να κατασκευάσει κλειδιά που προσγειώνονται όλα σε έναν κάδο, υποβαθμίζοντας τις λειτουργίες hash. Το p5 runtime τυχαιοποιεί το hash seed στην εκκίνηση της διεργασίας και τυχαιοποιεί την ορατή σειρά των `keys`, `values` και `each` ανά hash. Δύο μεταβλητές περιβάλλοντος παρακάμπτουν αυτές τις προεπιλογές για αποσφαλμάτωση: το `PERL_HASH_SEED` καρφώνει το seed (που ελέγχει την αποθήκευση, όχι την παρουσιαζόμενη σειρά) και το `PERL_PERTURB_KEYS` ελέγχει τη διατάραξη της σειράς διάσχισης· η συνάρτηση `hash_traversal_mask` του `Hash::Util` κάνει το ίδιο ανά hash. Αυτές οι παρακάμψεις αποδυναμώνουν την προστασία και δεν συνιστώνται στην παραγωγή. Η PetaPerl δεν εγγυήθηκε ποτέ καμία σειρά κλειδιών hash· μην βασίζεστε σε αυτήν, και μην χρησιμοποιείτε την ψευδοτυχαία σειρά ως ανακάτεμα.
- **Καταστροφική οπισθοχώρηση regex.** Η μηχανή κανονικών εκφράσεων είναι ένα NFA και μπορεί να καταναλώσει χρόνο ή χώρο που αυξάνεται απότομα όταν ένα μοτίβο μπορεί να ταιριάξει μια είσοδο με πολλούς τρόπους. Μια εχθρική είσοδος που αντιστοιχίζεται έναντι ενός ευάλωτου μοτίβου μπορεί να εξαντλήσει τη μνήμη ή τη CPU. Περιορίστε το μέγεθος της εισόδου και προτιμήστε μοτίβα που δεν μπορούν να οπισθοχωρήσουν παθολογικά.
- **Sort.** Η `sort` χρησιμοποιεί mergesort, του οποίου η χειρότερη περίπτωση είναι φραγμένη, οπότε δεν μπορεί να οδηγηθεί σε τετραγωνική συμπεριφορά από εχθρική είσοδο.

Τα tied hashes μπορεί να φέρουν τη δική τους σειρά και τα δικά τους χαρακτηριστικά πολυπλοκότητας.

Διαφορές από το upstream

- **Μόνο Linux.** Η PetaPerl εκτελείται μόνο σε Linux. Η συζήτηση για set-id και ασφάλεια σε VMS, Windows και άλλες πλατφόρμες που κουβαλά η upstream Perl δεν έχει αντίστοιχο εδώ και απλώς απουσιάζει.
- **Κανένα εργαλείο «προστασίας».** Η PetaPerl δεν έχει μεταγλωττιστή bytecode ούτε μηχανισμό συσκότισης με source-filter, οπότε η upstream συμβουλή περί διανομής μεταγλωττισμένου ή φιλτραρισμένου κώδικα δεν ισχύει. Η συσκότιση του πηγαίου κώδικα δεν ήταν ποτέ μέτρο ασφαλείας έτσι κι αλλιώς.
- **Το διαμέρισμα `Safe` δεν είναι ακόμη διαθέσιμο.** Το άρθρωμα `Safe` (και το άρθρωμα `Opcode` στο οποίο στηρίζεται) δεν είναι ακόμη διαθέσιμο στην PetaPerl· η φόρτωσή του σήμερα αποτυγχάνει με `dynamic loading not available in this perl`. Μέχρι να γίνει, μην βασίζεστε στην προσέγγιση «εκτέλεσε μη έμπιστο κώδικα σε ένα περιορισμένο διαμέρισμα»: αντιμετωπίστε τον μη έμπιστο *κώδικα* (σε αντίθεση με τα μη έμπιστα δεδομένα) ως κάτι που εκτελείται σε ξεχωριστό sandbox σε επίπεδο OS ή σε διεργασία με ριγμένα προνόμια, όχι μέσα στον διερμηνευτή.

Δείτε επίσης

- perlvar - το ${^TAINT}, το %ENV, και οι μεταβλητές πραγματικού/ενεργού ID $< $> $($) που αποφασίζουν αν το taint mode αυτοενεργοποιείται.
- perllocale - γιατί τα δεδομένα locale είναι μη έμπιστα και πώς το use locale αλληλεπιδρά με το ξέπλυμα μέσω \w.
- perlop - ο τελεστής ?: και η εξαίρεσή του στο taint ολόκληρης της έκφρασης.
- *Διακόπτες γραμμής εντολών* - οι διακόπτες -T, -t και -I στο πλαίσιό τους.
- *Επιλογές γραμμής εντολών* - πώς ο pperl αναλύει τους διακόπτες του, συμπεριλαμβανομένου του επιλογέα runtime.
- Scalar::Util - η tainted, για τη διερεύνηση της κατάστασης taint μιας τιμής.
- Hash::Util - η hash_traversal_mask, για έλεγχο ανά hash της τυχαιοποίησης της σειράς διάσχισης.

5.2 Αρθρώματα

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 pperl 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,
- Compress :: [Raw]
- 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.

- IPC :: [SysV]
- List :: [Util]
- MIME :: [Base64, QuotedPrint]
- Math :: [GMP]
- Opcode - Name, group and mask the interpreter's opcodes - the machinery Safe
- 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, XS]
- 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_-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).

minus_c

Request that compilation stop after the compile phase (`-c`).

save_begins

Request that `BEGIN` blocks be saved rather than freed (`save_BEGINs`).

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 `0xNNNNNNNN`.

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_svtype

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

rhe_hash

Return the integer value of the wrapped SV, stringifying if necessary.

iv_ivx

Return the IV slot of the wrapped SV without coercion.

iv_uvx

Return the UV slot of the wrapped SV without coercion.

iv_iv

Return the SV's IV slot without coercion (`B::IV::IV`).

B::NV methods

nv_svnv

Return the floating-point value of the wrapped SV, coercing if needed.

nv_nvx

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`).

pv_rv

Return the referent of a reference SV as a B:: object (`B::PV::RV`).

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_xsub

Return the CV's C function pointer as an integer, or `0` for a Perl sub.

Synopsis

```
my $addr = $cv->XSUB;   # non-zero iff the sub is an XSUB
```

cv_xsubany

Return `CvXSUBANY` - per-XSUB private data.

Synopsis

```
my $any = $cv->XSUBANY;          # any_iv; 0 for Perl subs
my $sv  = $const_cv->XSUBANY;    # B object of the constant for␣
↪CvCONST subs
```

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.

cv_cvconst

Return true if the CV is a constant (inlinable) sub (`CvCONST`).

cv_const_sv

Return the SV a constant sub yields, as a B:: object (`const_sv`).

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_form

Return the format slot of the GV as a `B::CV` (stash walking via `O=Deparse` visits every GP slot).

gv_io

Return the IO slot of the GV as a `B::IO`.

gv_cvgen

Return the GV's method-cache generation number.

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.

hv_keys

Επιστρέφει τον αριθμό των κλειδιών στο hash (HvKEYS).

hv_array

Return the hash's contents as a key/B-object list (stash walking: B::Deparse's stash_subs via O=Deparse is the main consumer).

hv_max

Return the hash's bucket-array size (HvMAX).

hv_fill

Return the number of used buckets (HvFILL).

hv_riter

Return the hash's iterator position (HvRITER).

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.

Συνάρτηση

unop_aux_string

Return a string representation of op_aux where possible.

unop_aux_aux_list

Return the contents of the op_aux array as a list of IV/GV/etc.

op_next

Return the next op in execution order (op_next). @category B::OP methods

op_sibling

Return the next syntactic sibling. @category B::OP methods

op_targ

Return the op's pad target index (op_targ). @category B::OP methods

op_flags

Return the op's shared flag byte (op_flags). @category B::OP methods

op_private

Return the op's private flag byte (op_private). @category B::OP methods

op_first

Return the first child of a UNOP (op_first). @category B::OP methods

op_last

Return the last child of a BINOP/LISTOP (op_last). @category B::OP methods

op_other

Return a LOGOP's alternate branch target (op_other). @category B::OP methods

op_redoop

Return a LOOP's redo target (op_redoop). @category B::OP methods

op_nextop

Return a LOOP's next target (op_nextop). @category B::OP methods

op_lastop

Return a LOOP's last target (op_lastop). @category B::OP methods

op_pmflags

Return a PMOP's compiled-pattern flags (op_pmflags). @category B::OP methods

op_sv

Return an SVOP's SV payload (op_sv). @category B::OP methods

op_gv

Return an SVOP's GV payload (op_sv, GV-typed). @category B::OP methods

op_padix

Return a PADOP's pad index (op_padix). @category B::OP methods

op_cop_seq

Return a COP's compile sequence number (cop_seq). @category B::OP methods

op_line

Return a COP's source line number (cop_line). @category B::OP methods

op_hints

Return a COP's hint bits (cop_hints). @category B::OP methods

op_file

Return a COP's source filename (CopFILE). @category B::OP methods

op_stash

Return a COP's package stash as a B::HV (cop_stash). @category B::OP methods

op_stashpv

Return a COP's package name (CopSTASHPV). @category B::OP methods

op_name

Return the short op name (e.g. «print»). @category B::OP methods

op_desc

Return the human-readable op description. @category B::OP methods

op_ppaddr

Return a C-source reference to the op's pp function. @category B::OP methods

op_type

Return the numeric op-type code (`op_type`). @category B::OP methods

op_opt

Return the `op_opt` bit (peephole-optimised flag). @category B::OP methods

op_children

Επιστρέφει τον αριθμό των χαρακτήρων μιας συμβολοσειράς.

op_pmreplroot

Return a PMOP's replacement-optree root (`pmreplroot`). @category B::OP methods

op_precomp

Return a PMOP's precompiled pattern source (`RX_PRECOMP`). @category B::OP methods

op_label

Return a COP's statement label, or undef (`CopLABEL`). @category B::OP methods

op_moresib

Return the `op_moresib` bit. @category B::OP methods

op_parent

Return the syntactic parent of the op. @category B::OP methods

op_methop_first

Return a METHOP's first child for dynamic method calls. @category B::OP methods

op_meth_sv

Return a METHOP's method-name SV (`meth_sv`). @category B::OP methods

op_pmregexp

Return a PMOP's compiled regexp as a B::REGEXP. @category B::OP methods

op_rclass

Return a METHOP's redirect class SV (`rclass`). @category B::OP methods

av_fill

Return the logical last index of the array (`AvFILL`, i.e. $#a).

av_max

Return the allocated capacity of the array (`AvMAX`).

av_array

Return the array's elements as a list of B:: objects (`AvARRAY`).

av_arrayelt

Return the Nth element as a B:: object (`AvARRAYelt`).

padlist_max

Return the highest index in the padlist (`PadlistMAX`).

padlist_names

Return the padlist's PADNAMELIST (`PadlistNAMES`).

padlist_array

Return (NAMES, pad, pad, ...) as a list (`PadlistARRAY`).

padlist_arrayelt

Return the Nth padlist element (`PadlistARRAYelt`).

padnamelist_max

Return the highest index in the padname list (`PadnamelistMAX`).

padnamelist_array

Return the padnames as a list of B::PADNAME objects (PadnamelistARRAY).

padnamelist_arrayelt

Return the Nth padname as a B::PADNAME (PadnamelistARRAYelt).

padname_pv

Return the pad name PV with its sigil (e.g. «$x», «@arr») (PadnamePV).

padname_len

Return the pad name length in bytes (PadnameLEN).

padname_flags

Return the pad name flags (PadnameFLAGS), with SVf_FAKE OR'd for OUTER.

padname_type

Return the padname's type stash as a B:: object (PadnameTYPE).

padname_ourstash

Return the "our" stash for the padname (PadnameOURSTASH).

svref_2object

Wrap a reference's referent as a `B::*` object.

Σύνοψη

```
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.

Σύνοψη

```
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.

Σύνοψη

```
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.

Σύνοψη

```
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.

Σύνοψη

```
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.

Σύνοψη

```
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.

Σύνοψη

```
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`.

Σύνοψη

```
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.

Σύνοψη

```
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.

Σύνοψη

```
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.

Σύνοψη

```
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`.

Σύνοψη

```
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.

Σύνοψη

```
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.

Σύνοψη

```
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.

Σύνοψη

```
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`.

Σύνοψη

```
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`.

Σύνοψη

```
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.

Σύνοψη

```
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.

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).

Σύνοψη

```
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.

Σύνοψη

```
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)
```

Τι επιστρέφεται

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.

Παραδείγματα

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
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
my $copy = clone($a);
print $copy->{peer}{peer}{name};   # A  (cycle reproduced)
```

Οριακές περιπτώσεις

- `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.

Διαφορές από το upstream

Fully compatible with upstream `Clone` 0.50.

Δείτε επίσης

- `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 Compress

Namespace containing the following modules:

- `Compress::Raw`

Compress::Raw

Namespace containing the following modules:

- `Compress::Raw::Zlib` - Low-level interface to the `zlib` compression library.

Compress::Raw::Zlib

Low-level interface to the `zlib` compression library.

`Compress::Raw::Zlib` is the thin binding layer underneath `Compress::Zlib` and the whole `IO::Compress::*` family. It exposes zlib's streaming `deflate`/`inflate` state machines directly, plus the `crc32`/`adler32` checksums and the zlib manifest constants.

```
use Compress::Raw::Zlib;

my ($d, $status) = Compress::Raw::Zlib::Deflate->new();
$d->deflate($input, my $output);
$d->flush($output);

my ($i, $istatus) = Compress::Raw::Zlib::Inflate->new();
$i->inflate($output, my $roundtrip);
```

This is a 1:1 transliteration of `cpan/Compress-Raw-Zlib/Zlib.xs` (upstream 2.222). The Perl half of `Compress/Raw/Zlib.pm` - the `Parameters` option parser and the `Deflate`/`Inflate`/`InflateScan` constructors - is embedded verbatim and injected with `eval_pv_boot`, the same faithful-mirror technique `B.rs` uses for `B.pm`.

zlib itself is reached through `dlopen` (`libz.so.1`), exactly the library the upstream XS links against. No DEFLATE implementation lives here.

Functions

Other Functions

ret_iv

typemap `T_IV`: `sv_setiv(TARG, (IV)RETVAL); ST(0) = TARG;`

ret_uv

typemap `T_UV`: xsubpp emits `TARGu((IV)RETVAL, 1); ST(0) = TARG;`

ret_pv

typemap `T_PV`: `sv_setpv((SV*)TARG, RETVAL); ST(0) = TARG;` A NULL RETVAL sets undef - which is what `msg()` wants.

ret_bool

typemap `T_BOOL`: `ST(0) = boolSV(RETVAL);`

ret_dual

typemap `T_DUAL` (DualType): `SV * RETVALSV = sv_newmortal(); setDUALstatus(RETVALSV, RETVAL); ST(0) = RETVALSV;`

5.2.4 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.

Ενδοσκόπηση

`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.

Σύνοψη

```
use Config;
print Config::myconfig();
```

Τι επιστρέφεται

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.

Παραδείγματα

```
use Config;
my $summary = Config::myconfig();
print $summary if $summary =~ /osname=linux/;
```

Οριακές περιπτώσεις

- Returns a fresh scalar on every call; cache it if used repeatedly in a hot path.

Διαφορές από το 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.

Σύνοψη

```
use Config;
my $sh = Config::config_sh();
```

Τι επιστρέφεται

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.

Παραδείγματα

```
use Config;
for my $line (split /^/, Config::config_sh()) {
    print $line if $line =~ /^archname=/;
}
```

Οριακές περιπτώσεις

- Entries whose value is undef render as `key=''`, matching how the tied hash reports them via `EXISTS` (true) and `FETCH` (undef).

Διαφορές από το upstream

Fully compatible with upstream.

config_vars

Print selected configuration entries to `STDOUT`.

Σύνοψη

```
use Config;
Config::config_vars(qw(archname osname ivsize));
```

Τι επιστρέφεται

Nothing. Each named key is written to `STDOUT` on its own line as `key='value';`. Unknown keys render as `key='UNKNOWN';`, keys whose recorded value is undef as `key='undef';`. This is the backend of the `-V:key` command-line switch (see perlrun), including its query syntax: a leading `:` suppresses the `key=` tag, a trailing `:` ends the line with a space instead of `;\n`, and a query containing non-word characters is treated as a regex and answered from `config_re`.

Παραδείγματα

```
use Config;
Config::config_vars('archname');           # archname='x86_64-linux
 ↪';
Config::config_vars('archname', 'osname'); # two lines
Config::config_vars('ivsize|nvsize');      # regex: both entries
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
Config::config_vars(':archname');          # 'x86_64-linux'; (no␣
↪tag)
```

Οριακές περιπτώσεις

- Unknown keys print `'UNKNOWN'`; undef values print `'undef'`.
- Regex queries with no match print `pattern: not found`.
- Output goes straight to the OS `STDOUT`; redirecting with `local *STDOUT` in Perl does not affect it.

Διαφορές από το upstream

- Regex queries share `config_re`'s pattern approximation (see `config_re` «Differences from upstream»).

fetch

Tied-hash read: return the value stored under a key.

Σύνοψη

```
my $arch = $Config{archname};   # calls FETCH
```

Τι επιστρέφεται

The string value for the given key, or `undef` if the key is either absent or recorded as undef (for example `d_pseudofork`).

Παραδείγματα

```
use Config;
print $Config{osname};          # linux
print $Config{archname};        # x86_64-linux
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream.

exists

Tied-hash membership test: does a key exist in `%Config`?

Σύνοψη

```
if (exists $Config{useithreads}) { ... }
```

Τι επιστρέφεται

A true value when the key is part of the configuration (even if the recorded value is undef), false otherwise.

Παραδείγματα

```
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)
```

Οριακές περιπτώσεις

- Pair `EXISTS` with `FETCH` to distinguish «key is absent» from «key is present but undef».

Διαφορές από το upstream

Fully compatible with upstream.

store

Tied-hash write: always croaks; `%Config` is read-only.

Σύνοψη

```
$Config{archname} = 'foo';   # dies
```

Τι επιστρέφεται

Nothing. The call raises `Config is read-only` and unwinds to the nearest `eval`.

Παραδείγματα

```
eval { $Config{archname} = 'bogus' };
print $@;                      # Config is read-only
```

Οριακές περιπτώσεις

- The croak fires regardless of the key or value; there is no per-key override.

Διαφορές από το upstream

Fully compatible with upstream.

firstkey

Tied-hash iterator: reset and return the first key.

Σύνοψη

```
my @keys = keys %Config;            # triggers FIRSTKEY + NEXTKEY
while (my ($k, $v) = each %Config) { ... }
```

Τι επιστρέφεται

The first key in the internal iteration order, or `undef` if the hash is empty.

Οριακές περιπτώσεις

- Iteration order is unspecified; never rely on it.
- Calling `FIRSTKEY` resets a `NEXTKEY` walk in progress.

Διαφορές από το upstream

Fully compatible with upstream.

nextkey

Tied-hash iterator: return the next key after `FIRSTKEY`.

Σύνοψη

```
while (my ($k, $v) = each %Config) { ... }
```

Τι επιστρέφεται

The next key in the current iteration, or `undef` when the walk is exhausted.

Οριακές περιπτώσεις

- Calling `NEXTKEY` without a prior `FIRSTKEY` starts from wherever the hash's internal iterator happens to be.

Διαφορές από το upstream

Fully compatible with upstream.

config_re

Grep configuration entries whose key matches a pattern.

Σύνοψη

```
use Config;
my @hits = Config::config_re('^cc');
```

Τι επιστρέφεται

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.

Παραδείγματα

```
use Config;
my @arch = Config::config_re('arch');   # archname, archlibexp, ...
my @d    = Config::config_re('^d_');    # every d_* feature flag
```

Οριακές περιπτώσεις

- An empty pattern returns no results.
- Non-UTF-8 pattern bytes return no results.

Διαφορές από το 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.

Σύνοψη

```
use Config;
print Config::compile_date(), "\n";
```

Τι επιστρέφεται

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.

Διαφορές από το 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.

Σύνοψη

```
use Config;
my @patches = Config::local_patches();
```

Τι επιστρέφεται

A list of strings, one per local patch. The list is empty when the interpreter was built from pristine sources.

Παραδείγματα

```
use Config;
print "vanilla build\n" unless Config::local_patches();
```

Διαφορές από το 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.

Σύνοψη

```
use Config;
my @opts = Config::bincompat_options();
```

Τι επιστρέφεται

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.

Παραδείγματα

```
use Config;
print "64-bit ints\n"
    if grep { $_ eq 'USE_64_BIT_INT' } Config::bincompat_options();
```

Διαφορές από το 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.

Σύνοψη

```
use Config;
my @opts = Config::non_bincompat_options();
```

Τι επιστρέφεται

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.

Διαφορές από το upstream

- Reports only the switches that meaningfully apply to pperl's build.

header_files

Return the canonical list of Perl C header filenames.

Σύνοψη

```
use Config;
my @hdrs = Config::header_files();
```

Τι επιστρέφεται

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.

Οριακές περιπτώσεις

- The filenames name headers from the upstream perl C sources. pperl does not ship them as separate files; callers that read the headers from disk will fail to find them.

Διαφορές από το upstream

- pperl 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`.

Σύνοψη

```
use Config;
Config::_V();
```

Τι επιστρέφεται

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`.

Παραδείγματα

```
## Equivalent of the command-line `pperl -V`:

use Config;
Config::_V();
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το 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.

5.2.5 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.

Συνάρτηση

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.

Σύνοψη

```
use Cwd;
my $dir = getcwd();
my $dir = cwd();
my $dir = fastcwd();
my $dir = fastgetcwd();
```

Τι επιστρέφεται

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 `$!`.

Παραδείγματα

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 `chdir`s:

```
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
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το 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.

Δείτε επίσης

- `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.

Σύνοψη

```
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
```

Τι επιστρέφεται

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.

Παραδείγματα

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
```

Οριακές περιπτώσεις

- 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`.

Διαφορές από το 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.

Δείτε επίσης

- `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.

Σύνοψη

```
use Cwd qw(fast_abs_path);
my $abs = fast_abs_path($path);
my $abs = fast_abs_path();      # defaults to current directory
```

Τι επιστρέφεται

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.

Παραδείγματα

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()
```

Οριακές περιπτώσεις

- 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`.

Διαφορές από το 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.

Δείτε επίσης

- `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.

Σύνοψη

```
use Cwd qw(chdir);
chdir($path) or die $!;
```

Τι επιστρέφεται

1 on success, `0` on failure. `$!` is set when the call fails so you can distinguish errors.

Παραδείγματα

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 $!;
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

- Upstream's `Cwd::chdir` normalises slashes in the argument before calling the builtin. pperl 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.

Δείτε επίσης

- `getcwd` - absolute path of the current directory
- `abs_path` - canonicalise a path without changing directory

5.2.6 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`.

This file is a 1:1 transliteration of perl5-upstream/dist/Data-Dumper/Dumper.xs. dd_dump mirrors DD_dump branch-for-branch (one long function, as the C comment demands); xs_dumpxs mirrors Data_Dumper_Dumpxs; xs_vstring mirrors Data_Dumper__vstring. The output buffer is an SV (as in the C), the seen table is an HV of 3-slot AVs (as in the C), and the postamble is a real AV (postav).

Functions

Core API

dumpxs

Serialize one or more values into valid Perl code, fast.

Ενδοσκόπηση

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.

Σύνοψη

```
use Data::Dumper;
print Data::Dumper->new([\%h, \@a], ['hash', 'array'])->Dumpxs;
print Data::Dumper::Dumpxs('Data::Dumper', [\%h], ['hash']);
```

Τι επιστρέφεται

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.

Παραδείγματα

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;
```

Οριακές περιπτώσεις

- 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 the quoted stringification of the ref (e.g. `'ARRAY(0x...)'`) at the cut-off point rather than croaking.
- CODE references: emitted as `sub { "DUMMY" }` unless `Deparse(1)` is set, in which case the deparsed source is emitted.
- Regexp references: emitted as `qr/.../flags`.
- `Sortkeys` as a coderef: called once per hash to obtain the key order; a truthy non-coderef sorts alphabetically.

Δείτε επίσης

- `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.

5.2.7 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.

Σύνοψη

```
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 `CV`s. 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 `CV`s.

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 pperl, 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 pperl 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.

5.2.8 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.

Σύνοψη

```
my $ctx = Digest::MD5->new;   # class method - new empty context
$ctx->new;                    # instance method - reset in place
```

Τι επιστρέφεται

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.

Παραδείγματα

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;
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `Digest::MD5` 2.58.

Δείτε επίσης

- `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.

Σύνοψη

```
my $copy = $ctx->clone;
```

Τι επιστρέφεται

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.

Παραδείγματα

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;
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `Digest::MD5` 2.58.

Δείτε επίσης

- `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.

Σύνοψη

```
## Called automatically - you do not invoke DESTROY yourself.

undef $ctx;                 # triggers DESTROY on the last reference
```

Τι επιστρέφεται

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.

Παραδείγματα

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
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `Digest::MD5` 2.58.

Δείτε επίσης

- `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.

Σύνοψη

```
$ctx->reset;                    # discard accumulated data
$ctx->reset->add($data);        # reset and keep going in one line
```

Τι επιστρέφεται

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.

Παραδείγματα

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;
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `Digest::MD5` 2.58.

Δείτε επίσης

- `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.

Σύνοψη

```
$ctx->add_bits($bitstring);         # "01101000..." form
$ctx->add_bits($data, $nbits);      # byte-string + bit count
```

Τι επιστρέφεται

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.

Παραδείγματα

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;
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream Digest::MD5 2.58.

Δείτε επίσης

- 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.

Σύνοψη

```
$ctx->add($data);
$ctx->add($a, $b, $c);               # same as add($a.$b.$c)
$ctx->add("a")->add("b")->add("c");  # chained equivalent
```

Τι επιστρέφεται

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.

Παραδείγματα

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");
```

Οριακές περιπτώσεις

- **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.

Διαφορές από το upstream

Fully compatible with upstream `Digest::MD5` 2.58.

Δείτε επίσης

- `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.

Σύνοψη

```
open(my $fh, '<:raw', $path) or die $!;
$ctx->addfile($fh);
```

Τι επιστρέφεται

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.

Παραδείγματα

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;
```

Οριακές περιπτώσεις

- 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`.

Διαφορές από το upstream

Fully compatible with upstream `Digest::MD5` 2.58.

Δείτε επίσης

- `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.

Σύνοψη

```
my $bin = $ctx->digest;       # exactly 16 bytes
```

Τι επιστρέφεται

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.

Παραδείγματα

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);
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `Digest::MD5` 2.58.

Δείτε επίσης

- `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.

Σύνοψη

```
my $hex = $ctx->hexdigest;    # 32 chars, [0-9a-f]
```

Τι επιστρέφεται

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.

Παραδείγματα

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;
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `Digest::MD5` 2.58.

Δείτε επίσης

- `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.

Σύνοψη

```
my $b64 = $ctx->b64digest;    # 22 chars, no padding
```

Τι επιστρέφεται

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.

Παραδείγματα

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;
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `Digest::MD5` 2.58.

Δείτε επίσης

- `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.

Σύνοψη

```
my @state = $ctx->context;                          # get
$ctx->context($blocks, $state_bytes, $partial);  # restore
```

Τι επιστρέφεται

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.

Παραδείγματα

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
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
## ...

my $ctx = Digest::MD5->new;
$ctx->context(@{ thaw($blob) });
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `Digest::MD5` 2.58.

Δείτε επίσης

- `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.

Σύνοψη

```
use Digest::MD5 qw(md5);
my $bin = md5($data);              # 16 bytes
my $bin = md5($a, $b, $c);         # md5($a.$b.$c)
```

Τι επιστρέφεται

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.

Παραδείγματα

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));
```

Οριακές περιπτώσεις

- `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.

Διαφορές από το upstream

Fully compatible with upstream `Digest::MD5` 2.58.

Δείτε επίσης

- `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.

Σύνοψη

```
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)
```

Τι επιστρέφεται

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.

Παραδείγματα

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
```

Οριακές περιπτώσεις

- `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.

Διαφορές από το upstream

Fully compatible with upstream `Digest::MD5` 2.58.

Δείτε επίσης

- `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.

Σύνοψη

```
use Digest::MD5 qw(md5_base64);
my $b64 = md5_base64($data);       # 22 chars, no padding
```

Τι επιστρέφεται

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.

Παραδείγματα

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;
```

Οριακές περιπτώσεις

- `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.

Διαφορές από το upstream

Fully compatible with upstream `Digest::MD5` 2.58.

Δείτε επίσης

- `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

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`.

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 × 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.

Σύνοψη

```
use Digest::SHA qw(hmac_sha256_hex hmac_sha1_base64);
my $tag = hmac_sha256_hex($message, $key);
my $b64 = hmac_sha1_base64(@chunks, $key);
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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
```

Οριακές περιπτώσεις

- 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`.

Διαφορές από το upstream

Fully compatible with upstream `Digest::SHA` 6.04.

Δείτε επίσης

- `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.

Σύνοψη

```
my $sha = Digest::SHA::newSHA("Digest::SHA", 256);
```

Τι επιστρέφεται

A blessed `Digest::SHA` object, ready to accept `add`, `addfile`, or `digest` calls. Returns `undef` on an unknown algorithm code.

Παραδείγματα

```
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
```

Οριακές περιπτώσεις

- Missing algorithm or missing class name croaks with a usage error.
- Unknown algorithm code returns `undef`; check the return value before using it.

Διαφορές από το upstream

Fully compatible with upstream `Digest::SHA` 6.04.

Δείτε επίσης

- `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`).

Σύνοψη

```
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
```

Τι επιστρέφεται

A blessed `Digest::SHA` object, or `undef` when the algorithm argument cannot be parsed as a supported algorithm.

Παραδείγματα

```
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
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `Digest::SHA` 6.04.

Δείτε επίσης

- `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.

Σύνοψη

```
my $base = Digest::SHA->new(256);
$base->add($prefix);
my $copy = $base->clone;
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `Digest::SHA` 6.04.

Δείτε επίσης

- `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.

Σύνοψη

```
{
    my $sha = Digest::SHA->new(256);
    # ... use $sha ...
}   # DESTROY runs here when $sha leaves scope
```

Τι επιστρέφεται

Nothing. DESTROY returns to Perl with no value.

Παραδείγματα

```
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
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream Digest::SHA 6.04.

Δείτε επίσης

- 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").

Σύνοψη

```
$sha->reset;             # rewind, keep algorithm
$sha->reset(512);        # rewind and switch to SHA-512
$sha->reset("SHA-384");  # string form
```

Τι επιστρέφεται

The receiver on success (chainable), or `undef` if an algorithm argument was supplied that could not be parsed.

Παραδείγματα

```
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
```

Οριακές περιπτώσεις

- Called on a non-`Digest::SHA` object, returns `undef`.
- After the built-in auto-reset inside `digest`, calling `reset` explicitly is harmless.

Διαφορές από το upstream

Fully compatible with upstream `Digest::SHA` 6.04.

Δείτε επίσης

- `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.

Σύνοψη

```
$sha->add($data);
$sha->add(@chunks);
$sha->add($a)->add($b)->add($c);
```

Τι επιστρέφεται

The receiver, so calls chain.

Παραδείγματα

```
my $sha = Digest::SHA->new(256);
$sha->add("abc");
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")
```

Οριακές περιπτώσεις

- 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`.

Διαφορές από το upstream

Fully compatible with upstream `Digest::SHA` 6.04.

Δείτε επίσης

- `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.

Σύνοψη

```
$sha->add_bits("11001010");          # 8 bits from ASCII 0/1 string
$sha->add_bits($packed, 13);         # first 13 bits of $packed
```

Τι επιστρέφεται

The receiver, for chaining.

Παραδείγματα

```
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
```

Οριακές περιπτώσεις

- 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`.

Διαφορές από το upstream

Fully compatible with upstream `Digest::SHA` 6.04.

Δείτε επίσης

- `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.

Σύνοψη

```
open my $fh, "<:raw", "archive.tar.gz" or die $!;
$sha->_addfilebin($fh);
```

Τι επιστρέφεται

The receiver, for chaining.

Παραδείγματα

```
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
```

Οριακές περιπτώσεις

- 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`.

Διαφορές από το upstream

Fully compatible with upstream `Digest::SHA` 6.04.

Δείτε επίσης

- `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.

Σύνοψη

```
open my $fh, "<", "notes.txt" or die $!;
$sha->_addfileuniv($fh);
```

Τι επιστρέφεται

The receiver, for chaining.

Παραδείγματα

```
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");
```

Οριακές περιπτώσεις

- 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`.

Διαφορές από το upstream

Fully compatible with upstream `Digest::SHA` 6.04.

Δείτε επίσης

- `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.

Σύνοψη

```
$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
```

Τι επιστρέφεται

The receiver, for chaining.

Παραδείγματα

```
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);
```

Οριακές περιπτώσεις

- 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`.

Διαφορές από το upstream

Fully compatible with upstream `Digest::SHA` 6.04.

Δείτε επίσης

- `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.

Σύνοψη

```
my $bytes = $sha->digest;
```

Τι επιστρέφεται

A scalar holding the raw digest bytes. `length` equals `hashsize / 8`.

Παραδείγματα

```
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;
```

Οριακές περιπτώσεις

- 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`.

Διαφορές από το upstream

Fully compatible with upstream `Digest::SHA` 6.04.

Δείτε επίσης

- `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.

Σύνοψη

```
my $hex = $sha->hexdigest;
```

Τι επιστρέφεται

A string of `[0-9a-f]` characters of length `hashsize / 4`.

Παραδείγματα

```
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";
```

Οριακές περιπτώσεις

- 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`.

Διαφορές από το upstream

Fully compatible with upstream `Digest::SHA` 6.04.

Δείτε επίσης

- `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.

Σύνοψη

```
my $b64 = $sha->b64digest;
```

Τι επιστρέφεται

A Base64 string using the standard alphabet, with no = padding.

Παραδείγματα

```
my $sha = Digest::SHA->new(256);
$sha->add("abc");
print $sha->b64digest;

## ungWv48Bz+pBQUDeXa4iI7ADYaOWF3qctBD/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";
```

Οριακές περιπτώσεις

- 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`.

Διαφορές από το upstream

Fully compatible with upstream `Digest::SHA` 6.04.

Δείτε επίσης

- `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.

Σύνοψη

```
my $code = $sha->algorithm;
```

Τι επιστρέφεται

An integer in the set `{1, 224, 256, 384, 512, 512224, 512256}`.

Παραδείγματα

```
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";
```

Οριακές περιπτώσεις

- Called on a non-`Digest::SHA` object, returns `undef`.

Διαφορές από το upstream

Fully compatible with upstream `Digest::SHA` 6.04.

Δείτε επίσης

- `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`.

Σύνοψη

```
my $bits = $sha->hashsize;
```

Τι επιστρέφεται

An integer, always a multiple of 8.

Παραδείγματα

```
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);
```

Οριακές περιπτώσεις

- Called on a non-`Digest::SHA` object, returns `undef`.

Διαφορές από το upstream

Fully compatible with upstream `Digest::SHA` 6.04.

Δείτε επίσης

- `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.

Σύνοψη

```
my $blob = $sha->_getstate;
```

Τι επιστρέφεται

A binary string of fixed length: 116 bytes for SHA-1/224/256, 212 bytes for SHA-384/512/512-224/512-256.

Παραδείγματα

```
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;
```

Οριακές περιπτώσεις

- The output is binary and includes null bytes; never interpret it as text.
- Called on a non-`Digest::SHA` object, returns `undef`.

Διαφορές από το upstream

Fully compatible with upstream `Digest::SHA` 6.04.

Δείτε επίσης

- `_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.

Σύνοψη

```
$sha->_putstate($blob);
```

Τι επιστρέφεται

The receiver on success, `undef` on failure (invalid length or bad object).

Παραδείγματα

```
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 $!;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
binmode $in;
my $blob = do { local $/; <$in> };
my $sha = Digest::SHA->new(256);
$sha->_putstate($blob);
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream Digest::SHA 6.04.

Δείτε επίσης

- _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.

Σύνοψη

```
my $text = $sha->getstate;
```

Τι επιστρέφεται

A newline-terminated ASCII string.

Παραδείγματα

```
my $sha = Digest::SHA->new(256);
$sha->add("hello");
print $sha->getstate;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
## 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);
```

Οριακές περιπτώσεις

- The text format is stable across `Digest::SHA` versions and endianness.
- Called on a non-`Digest::SHA` object, returns `undef`.

Διαφορές από το upstream

Fully compatible with upstream `Digest::SHA` 6.04.

Δείτε επίσης

- `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.

Σύνοψη

```
my $sha = Digest::SHA->putstate($text);
$sha->putstate($text);
```

Τι επιστρέφεται

A `Digest::SHA` object on success, `undef` if the string could not be parsed.

Παραδείγματα

```
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
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `Digest::SHA` 6.04.

Δείτε επίσης

- `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.

Σύνοψη

```
$sha->dump("hash.state");
$sha->dump;                    # to STDOUT
```

Τι επιστρέφεται

1 on success (truthy), `undef` on I/O failure.

Παραδείγματα

```
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";
```

Οριακές περιπτώσεις

- An unwritable path returns `undef` rather than croaking.
- Called on a non-`Digest::SHA` object, returns `undef`.

Διαφορές από το upstream

Fully compatible with upstream `Digest::SHA` 6.04.

Δείτε επίσης

- `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).

Σύνοψη

```
my $sha = Digest::SHA->load("hash.state");
```

Τι επιστρέφεται

A `Digest::SHA` object on success, `undef` if the file could not be read or the contents could not be parsed.

Παραδείγματα

```
my $sha = Digest::SHA->new(256);
$sha->add("one");
$sha->dump("hash.state");

my $resumed = Digest::SHA->load("hash.state");
$resumed->add("two");
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");
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `Digest::SHA` 6.04.

Δείτε επίσης

- `dump` - writes the file `load` reads.
- `putstate` - parses the text state directly from a string.
- `_putstate` - binary-state counterpart.

5.2.9 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.

Conversion

native_from_to

Reencode a byte string in place from one encoding to another.

native_encode

Turn a character string into a byte string in the named encoding.

Σύνοψη

```
my $bytes = encode($encoding, $string);
my $bytes = encode($encoding, $string, $check);
```

Τι επιστρέφεται

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;`.

Παραδείγματα

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
```

Οριακές περιπτώσεις

- `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.

Διαφορές από το 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`.

Δείτε επίσης

- `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.

Σύνοψη

```
my $string = decode($encoding, $bytes);
my $string = decode($encoding, $bytes, $check);
```

Τι επιστρέφεται

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.

Παραδείγματα

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
```

Οριακές περιπτώσεις

- 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 reallocating.

Διαφορές από το 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.

Δείτε επίσης

- 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.

Σύνοψη

```
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.

Τι επιστρέφεται

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.

Παραδείγματα

```
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"
```

Οριακές περιπτώσεις

- `undef` returns an empty byte string.
- Idempotent only when applied to a character string; applying it twice mojibakes the result.

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `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.

Σύνοψη

```
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.

Τι επιστρέφεται

A character string with the `SVf_UTF8` flag set. `length` on the result counts characters, not bytes.

Παραδείγματα

```
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
```

Οριακές περιπτώσεις

- `undef` returns an empty character string.
- Input that is already valid UTF-8 is copied and flagged; no byte-level conversion happens.

Διαφορές από το upstream

Fully compatible with upstream. Covered by `t/81-xs-native/Encode/010-basic.t`.

Δείτε επίσης

- `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.

Σύνοψη

```
my $enc = find_encoding($name);
my $bytes  = $enc->encode($string);
my $string = $enc->decode($bytes);
my $canon  = $enc->name;
```

Τι επιστρέφεται

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.

Παραδείγματα

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
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το 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`.

Δείτε επίσης

- `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.

Σύνοψη

```
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.

Διαφορές από το 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.

Σύνοψη

```
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.

Σύνοψη

```
from_to($octets, $from, $to);
from_to($octets, $from, $to, $check);
```

Τι επιστρέφεται

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).

Παραδείγματα

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.

Οριακές περιπτώσεις

- undef input returns undef and leaves $octets untouched.
- $check is forwarded to both the decode and encode halves.

Διαφορές από το upstream

Fully compatible with upstream for the registered encodings.

native_obj_encode

Method form of encode on an encoding object.

Σύνοψη

```
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.

Σύνοψη

```
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.

5.2.10 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.

Σύνοψη

```
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 errnos 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.

5.2.11 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).

Απόδοση

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.

Σύνοψη

```
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.

Παραδείγματα

```
use Fcntl qw(:mode);
my $mode = (stat $path)[2];
print "is a directory\n" if S_IFMT($mode) == S_IFDIR;
```

s_imode

Extract the permission bits from a stat-mode value.

Σύνοψη

```
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`.

Παραδείγματα

```
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.

Σύνοψη

```
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.

Σύνοψη

```
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.

Σύνοψη

```
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.

Σύνοψη

```
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).

Σύνοψη

```
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.

Σύνοψη

```
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.

Σύνοψη

```
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`.

5.2.12 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.

csh_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.

Σύνοψη

```
my @list    = bsd_glob('*.[ch]');
my @homes   = bsd_glob('~gnat', GLOB_TILDE | GLOB_ERR);
my @latest  = bsd_glob('src/*.pl', GLOB_NOSORT);
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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);
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το 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 pperl returns the match count. Covered by `t/81-xs-native/File/Glob/*.t`.
- Filenames are not tainted. pperl does not implement taint mode.

Δείτε επίσης

- `csh_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.

csh_glob

Expand a pattern using the default flag set - the engine behind Perl's built-in `glob`.

Σύνοψη

```
use File::Glob qw(csh_glob);
my @files = csh_glob('*.pl');
```

Τι επιστρέφεται

A list of matching filenames. Flags always come from `$File::Glob::DEFAULT_FLAGS` (initially `GLOB_CSH`); `csh_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. `csh_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.»

Παραδείγματα

```
use File::Glob qw(csh_glob);
my @matches = csh_glob('*.txt');        # uses DEFAULT_FLAGS
```

```
## Flip case sensitivity globally, then call

use File::Glob qw(csh_glob :nocase);
my @any = csh_glob('ReadMe*');          # matches README, readme,␣
↪ReadMe, ...
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

- No whitespace-splitting of patterns. Upstream `csh_glob` tokenises `"a* b*"` into two patterns; pperl 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`.

Δείτε επίσης

- `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.

Σύνοψη

```
my @files = bsd_glob('~nobody', GLOB_TILDE | GLOB_ERR);
if (GLOB_ERROR) {
    warn "glob failed: $!";
}
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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
```

Οριακές περιπτώσεις

- Reading `GLOB_ERROR` without a preceding `bsd_glob` call returns `0`.
- Reading it after a successful call returns `0`.

Διαφορές από το upstream

- Always returns `0`. pperl 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`.

Δείτε επίσης

- `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.

Σύνοψη

```
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"`, `"willneed"`, ...).
- `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.

Σύνοψη

```
map_file my $map, $filename;           # read-only
map_file my $map, $filename, '+<';     # read/write
map_file my $map, $filename, '<', $offset, $length;
```

Ορίσματα

- `$lvalue` - scalar to receive the mapping. Any previous value is discarded. The variable must not already be mapped.
- `$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.

Παραδείγματα

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;
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `File::Map` 0.71.

Δείτε επίσης

- `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.

Σύνοψη

```
open my $fh, '<:raw', $filename;
map_handle my $map, $fh;
map_handle my $map, $fh, '+<', $offset, $length;
```

Ορίσματα

- `$lvalue` - scalar to receive the mapping.
- `$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`.

Παραδείγματα

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;
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `File::Map` 0.71.

Δείτε επίσης

- `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.

Σύνοψη

```
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
```

Ορίσματα

- `$lvalue` - scalar to receive the mapping.
- `$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'`.

Παραδείγματα

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";
```

Οριακές περιπτώσεις

- Zero-length anonymous maps are rejected.
- Any `$type` other than `'shared'` or `'private'` raises an exception.

Διαφορές από το upstream

Fully compatible with upstream `File::Map` 0.71.

Δείτε επίσης

- `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.

Σύνοψη

```
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;
```

Ορίσματα

- `$lvalue` - scalar to receive the mapping.
- `$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.

Διαφορές από το upstream

Fully compatible with upstream `File::Map` 0.71.

Δείτε επίσης

- `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.

Σύνοψη

```
sync $map;          # synchronous - wait for the kernel to finish
sync $map, 0;       # asynchronous - queue the flush and return
```

Ορίσματα

- `$lvalue` - a currently mapped scalar.
- `$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.

Οριακές περιπτώσεις

- Calling `sync` on an unmapped scalar raises an exception.
- Calling `sync` on an empty mapping is a no-op.

Διαφορές από το upstream

Fully compatible with upstream `File::Map` 0.71.

Δείτε επίσης

- `unmap` - release the mapping (also triggers a flush).
- `advise` - influence the kernel's write-back strategy.

remap

Resize an existing mapping in place.

Σύνοψη

```
remap $map, $new_size;
```

Ορίσματα

- `$lvalue` - a currently mapped scalar.
- `$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.

Διαφορές από το upstream

Fully compatible with upstream `File::Map` 0.71.

Δείτε επίσης

- `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.

Σύνοψη

```
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.

Οριακές περιπτώσεις

- Calling `unmap` on a scalar that is not mapped raises an exception.

Διαφορές από το upstream

Fully compatible with upstream `File::Map` 0.71.

Δείτε επίσης

- `sync` - flush writes without releasing the mapping.

advise

Tell the kernel how the mapped region will be accessed.

Σύνοψη

```
advise $map, 'sequential';
advise $map, 'willneed';
```

Ορίσματα

- `$lvalue` - a currently mapped scalar.
- `$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'` - prefault 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.

Παραδείγματα

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) { ... }
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `File::Map` 0.71.

Δείτε επίσης

- `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.

Σύνοψη

```
protect $map, '<';              # read-only
protect $map, '+<';             # read/write
protect $map, PROT_READ;        # constants also work
```

Ορίσματα

- `$lvalue` - a currently mapped scalar.
- `$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.

Παραδείγματα

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
```

Οριακές περιπτώσεις

- Protecting an unmapped scalar raises an exception.
- Protecting an empty mapping updates the scalar's read-only flag but performs no kernel call.

Διαφορές από το upstream

Fully compatible with upstream `File::Map` 0.71.

Δείτε επίσης

- `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.

`mkstemps`

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 `X`s 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.

Σύνοψη

```
my ($fh, $filename) = tempfile();
my ($fh, $filename) = tempfile($template, DIR => $dir, SUFFIX => '.
↪dat');
my $fh              = tempfile();               # scalar context,␣
↪file auto-unlinked
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `File::Temp` 0.2312.

Δείτε επίσης

- `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.

Σύνοψη

```
my $dir = tempdir();
my $dir = tempdir($template, DIR => $parent, CLEANUP => 1);
my $dir = tempdir(CLEANUP => 1);
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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)
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `File::Temp` 0.2312.

Δείτε επίσης

- `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.

Σύνοψη

```
my $tmp = File::Temp->new;
my $tmp = File::Temp->new(TEMPLATE => 'dataXXXXXX', DIR => '/var/tmp
 ↪', SUFFIX => '.dat');
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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
```

Οριακές περιπτώσεις

- `OPEN => 0` is silently ignored; the file is always opened.
- Object destruction while `$File::Temp::KEEP_ALL` is true skips unlink.

Διαφορές από το upstream

Fully compatible with upstream `File::Temp` 0.2312.

Δείτε επίσης

- `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.

Σύνοψη

```
my $dir = File::Temp->newdir;
my $dir = File::Temp->newdir('buildXXXXXX', DIR => '/var/tmp',␣
↪CLEANUP => 0);
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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
```

Οριακές περιπτώσεις

- Forked children do not clean up the parent's directory.
- `$File::Temp::KEEP_ALL = 1` suppresses removal on destruction.

Διαφορές από το upstream

Fully compatible with upstream `File::Temp` 0.2312.

Δείτε επίσης

- `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.

Σύνοψη

```
my $path = $tmp->filename;
```

Τι επιστρέφεται

The filename string the object was opened against, or `undef` if the argument is not a recognised `File::Temp` handle.

Παραδείγματα

```
my $tmp = File::Temp->new;
print "writing to ", $tmp->filename, "\n";
```

Διαφορές από το upstream

Fully compatible with upstream `File::Temp` 0.2312.

Δείτε επίσης

- `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.

Σύνοψη

```
my $tmp = File::Temp->new;
print "wrote to $tmp\n";                    # calls STRINGIFY
```

Τι επιστρέφεται

The object's pathname as a string, or the empty string if the filename is not known.

Διαφορές από το upstream

Fully compatible with upstream `File::Temp` 0.2312.

Δείτε επίσης

- `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.

Σύνοψη

```
my $tmp = File::Temp->new;
print 0 + $tmp, "\n";                      # the refaddr
$tmp == $other_handle                       # compares by identity
```

Τι επιστρέφεται

The integer refaddr of the underlying reference - the same value `Scalar::Util::refaddr` would return.

Διαφορές από το upstream

Fully compatible with upstream `File::Temp` 0.2312.

Δείτε επίσης

- `STRINGIFY` - companion overload used for string contexts.

unlink_on_destroy

Toggle or query whether a `File::Temp` object unlinks its file when destroyed.

Σύνοψη

```
my $flag = $tmp->unlink_on_destroy;          # getter
$tmp->unlink_on_destroy(0);                  # keep the file
$tmp->unlink_on_destroy(1);                  # remove on destruction
```

Τι επιστρέφεται

The new (or current) flag value as an integer: `1` if the file will be removed when the object is destroyed, `0` otherwise.

Παραδείγματα

```
my $tmp = File::Temp->new;
$tmp->unlink_on_destroy(0);

## $tmp->filename survives after $tmp goes out of scope
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το 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`.

Δείτε επίσης

- `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.

Σύνοψη

```
{
    my $tmp = File::Temp->new;  # UNLINK => 1 by default
}  # DESTROY runs here; file is unlinked
```

Τι επιστρέφεται

Nothing - this is a destructor invoked by the runtime when the last reference drops. It is rare to call it directly.

Οριακές περιπτώσεις

- `$File::Temp::KEEP_ALL = 1` suppresses all file removal.
- Files not registered with cleanup tracking are left on disk.

Διαφορές από το upstream

Fully compatible with upstream `File::Temp` 0.2312.

Δείτε επίσης

- `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.

Σύνοψη

```
my $name          = tmpnam();                # scalar: just a name
my ($fh, $name)   = tmpnam();                # list: creates the file
```

Τι επιστρέφεται

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.

Παραδείγματα

```
use File::Temp qw/ tmpnam /;
my ($fh, $name) = tmpnam();                  # race-free file
 ↪creation
```

Οριακές περιπτώσεις

- Scalar context does not create a file - callers must open the name themselves and accept the race.

Διαφορές από το upstream

Fully compatible with upstream `File::Temp` 0.2312.

Δείτε επίσης

- `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.

Σύνοψη

```
my $fh = tmpfile();
print $fh "scratch\n";
seek $fh, 0, 0;
```

Τι επιστρέφεται

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.

Παραδείγματα

```
my $fh = tmpfile();
print $fh "line 1\n";
seek $fh, 0, 0;
my $line = <$fh>;                          # "line 1\n"
```

Οριακές περιπτώσεις

- On filesystems that refuse to unlink open files the file stays on disk until `$fh` is closed.

Διαφορές από το upstream

Fully compatible with upstream `File::Temp` 0.2312.

Δείτε επίσης

- `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.

Σύνοψη

```
my $name = tempnam('/var/tmp', 'cache_');
```

Τι επιστρέφεται

A full path string `"$dir/${prefix}XXXXXXXX"` with the `X`s replaced by random characters. The caller is responsible for opening the file, and there is a race between name selection and open.

Παραδείγματα

```
use File::Temp qw/ tempnam /;
my $name = tempnam('/tmp', 'sess-');
open my $fh, '>', $name or die;
```

Οριακές περιπτώσεις

- Wrong arg count: croaks with a usage message.
- Like `mktemp`, unsafe against attackers - prefer `tempfile`.

Διαφορές από το upstream

Fully compatible with upstream `File::Temp` 0.2312.

Δείτε επίσης

- `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.

Σύνοψη

```
my $name = mktemp('/tmp/dataXXXXXX');
```

Τι επιστρέφεται

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.

Παραδείγματα

```
use File::Temp qw/ mktemp /;
my $name = mktemp('/var/tmp/snapXXXXXXXX');
```

Οριακές περιπτώσεις

- Template with fewer than four trailing Xs: croaks.
- Wrong arg count: croaks with a usage message.
- Unsafe against racing attackers - prefer `tempfile`.

Διαφορές από το upstream

Fully compatible with upstream `File::Temp` 0.2312.

Δείτε επίσης

- `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.

Σύνοψη

```
my ($fh, $path) = mkstemp('/tmp/dataXXXXXX');
my  $fh         = mkstemp('/tmp/dataXXXXXX');   # scalar context
```

Τι επιστρέφεται

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.

Παραδείγματα

```
use File::Temp qw/ mkstemp /;
my ($fh, $path) = mkstemp('/var/tmp/workXXXXXX');
print $fh "payload\n";
```

Οριακές περιπτώσεις

- Template with fewer than four trailing `X`s: croaks.
- Wrong arg count: croaks with a usage message.

Διαφορές από το upstream

Fully compatible with upstream `File::Temp` 0.2312.

Δείτε επίσης

- `tempfile` - higher-level variant accepting `DIR`, `SUFFIX`, `UNLINK`.
- `mkstemps` - like `mkstemp` but keeps a literal suffix on the name.
- `mkdtemp` - directory counterpart.

mkstemps

Create a file from a template with a literal suffix preserved at the end of the name.

Σύνοψη

```
my ($fh, $path) = mkstemps('/tmp/dataXXXXXX', '.log');

## $path looks like /tmp/dataAb3c9F.log
```

Τι επιστρέφεται

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.

Παραδείγματα

```
use File::Temp qw/ mkstemps /;
my ($fh, $path) = mkstemps('/var/tmp/rpt-XXXXXX', '.json');
```

Οριακές περιπτώσεις

- The suffix is a literal string, not a length count.
- Template with fewer than four trailing `X`s before the suffix: croaks.
- Wrong arg count: croaks with a usage message.

Διαφορές από το upstream

Fully compatible with upstream `File::Temp` 0.2312.

Δείτε επίσης

- `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.

Σύνοψη

```
my $dir = mkdtemp('/tmp/buildXXXXXX');
```

Τι επιστρέφεται

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.

Παραδείγματα

```
use File::Temp qw/ mkdtemp /;
my $dir = mkdtemp('/var/tmp/stageXXXXXX');

## caller owns $dir; remove with rmtree / File::Path
```

Οριακές περιπτώσεις

- Template with fewer than four trailing `X`s: croaks.
- Wrong arg count: croaks with a usage message.

Διαφορές από το upstream

Fully compatible with upstream `File::Temp` 0.2312.

Δείτε επίσης

- `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.

Σύνοψη

```
File::Temp::cleanup();
```

Τι επιστρέφεται

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.

Παραδείγματα

```
use File::Temp;
my $dir = File::Temp::tempdir(CLEANUP => 1);

## ...

File::Temp::cleanup();                          # remove $dir now
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `File::Temp` 0.2312.

Δείτε επίσης

- `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.

Σύνοψη

```
unlink0($fh, $path) or die "unlink0: $!";
```

Τι επιστρέφεται

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.

Παραδείγματα

```
use File::Temp qw/ tempfile unlink0 /;
my ($fh, $path) = tempfile();
unlink0($fh, $path);

## $path is gone, but $fh is still usable
```

Οριακές περιπτώσεις

- If `$File::Temp::KEEP_ALL` is true the function returns 1 without unlinking.
- Fewer than two arguments: no-op, returns 1.

Διαφορές από το 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`.

Δείτε επίσης

- `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.

Σύνοψη

```
unlink1($fh, $path) or die "unlink1: $!";
```

Τι επιστρέφεται

Integer 1 on success.

Παραδείγματα

```
use File::Temp qw/ tempfile unlink1 /;
my ($fh, $path) = tempfile();

## ... use $fh ...

unlink1($fh, $path);
```

Οριακές περιπτώσεις

- `$File::Temp::KEEP_ALL = 1` suppresses removal.
- Fewer than two arguments: no-op, returns 1.

Διαφορές από το 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`.

Δείτε επίσης

- `unlink0` - unlink-while-open variant.
- `cleanup` - bulk removal of registered entries.

safe_level

Report the current safety level used by `File::Temp` checks.

Σύνοψη

```
my $level = File::Temp::safe_level();
File::Temp->safe_level(File::Temp::STANDARD);
```

Τι επιστρέφεται

The integer `0`, which corresponds to `STANDARD`. The setter form is accepted but has no effect.

Διαφορές από το 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`.

Δείτε επίσης

- `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.

Σύνοψη

```
my $uid = File::Temp::top_system_uid();
```

Τι επιστρέφεται

The integer `0` (root).

Διαφορές από το 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`.

Δείτε επίσης

- `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.

Σύνοψη

```
cmpstat($fh, $path) or die "fh/path diverged";
```

Τι επιστρέφεται

A truthy value if the check passes, `0` otherwise. A passing result is a necessary precondition for safe `unlink0`-style operations.

Διαφορές από το 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`.

Δείτε επίσης

- `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.

Σύνοψη

```
my $path = $dir->dirname;
```

Τι επιστρέφεται

The directory pathname as a string, or `undef` if called on something that is not a `File::Temp::Dir`.

Παραδείγματα

```
my $dir = File::Temp->newdir;
my $p   = $dir->dirname;                    # same as "$dir"
```

Διαφορές από το upstream

Fully compatible with upstream `File::Temp` 0.2312.

Δείτε επίσης

- `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.

Σύνοψη

```
my $dir = File::Temp->newdir;
print "dir is $dir\n";                    # calls STRINGIFY
```

Τι επιστρέφεται

The directory pathname, identical to `$dir->dirname`.

Διαφορές από το upstream

Fully compatible with upstream `File::Temp` 0.2312.

Δείτε επίσης

- `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.

Σύνοψη

```
my $flag = $dir->unlink_on_destroy;          # getter
$dir->unlink_on_destroy(0);                  # keep the tree
```

Τι επιστρέφεται

The new (or current) cleanup flag as an integer.

Παραδείγματα

```
my $dir = File::Temp->newdir;
$dir->unlink_on_destroy(0);

## $dir's path survives after $dir goes out of scope
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `File::Temp` 0.2312.

Δείτε επίσης

- `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.

Σύνοψη

```
{
    my $dir = File::Temp->newdir;   # CLEANUP => 1 by default
}  # DESTROY runs here; tree is removed
```

Τι επιστρέφεται

Nothing - invoked by the runtime when the last reference drops.

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `File::Temp` 0.2312.

Δείτε επίσης

- `unlink_on_destroy` - per-object opt-out for the removal behavior.
- `cleanup` - explicit cleanup of all registered temp entries.

5.2.13 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 `$_`.

Σύνοψη

```
my $status = filter_read();         # line mode
my $status = filter_read($size);    # block mode, up to $size bytes
```

Τι επιστρέφεται

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.

Παραδείγματα

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;
}
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `Filter::Util::Call` 1.65.

Δείτε επίσης

- `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.

Σύνοψη

```
my $status = filter_read_exact($size);
```

Τι επιστρέφεται

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).

Παραδείγματα

Read 16-byte records until end of file:

```
sub filter {
    my $status = filter_read_exact(16);
    decode_record($_) if $status > 0;
    $status;
}
```

Οριακές περιπτώσεις

- `$size` must be strictly positive. Calling `filter_read_exact(0)` or with a negative argument croaks with `"filter_read_exact: size parameter must be > 0"`.
- On EOF with a partially-filled `$_`, the returned status is 1, not `0` - this lets the filter process the tail of the stream before the next call reports true EOF.

Διαφορές από το upstream

Fully compatible with upstream `Filter::Util::Call` 1.65.

Δείτε επίσης

- `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).

Σύνοψη

```
filter_del();
```

Τι επιστρέφεται

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.

Παραδείγματα

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;
}
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `Filter::Util::Call` 1.65.

Δείτε επίσης

- `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.

Σύνοψη

```
Filter::Util::Call::real_import($object_or_coderef, $package, $is_
↪coderef);
```

Τι επιστρέφεται

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).

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `Filter::Util::Call` 1.65.

Δείτε επίσης

- `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`.

Σύνοψη

```
filter_add($object);     # method filter: $object is blessed state
filter_add(sub { ... }); # closure filter: anonymous sub
```

Τι επιστρέφεται

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.

Παραδείγματα

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;
}
```

Οριακές περιπτώσεις

- `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.

Διαφορές από το upstream

Fully compatible with upstream `Filter::Util::Call` 1.65.

Δείτε επίσης

- `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.

5.2.14 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.

Σύνοψη

```
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
$h{foo} = 10;                       # ok
eval { $h{baz} = 99 };              # croaks - attempt to access␣
 ↪disallowed key

lock_value(%h, 'foo');              # make $h{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`.

`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.

hash_unlocked

Report whether the hash's keyset is *not* currently locked.

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.

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.

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).

`used_buckets`

Return the count of buckets that currently hold at least one key.

`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 barcharts.

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.

Σύνοψη

```
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.

Σύνοψη

```
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.

Σύνοψη

```
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.

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.

Σύνοψη

```
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.

Σύνοψη

```
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`.

Σύνοψη

```
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.

Returns `undef` for a hash that has never been given a mask - the mask lives in the HV's aux struct, which is only allocated on demand, and upstream reports its absence rather than inventing a 0.

bucket_stats

Return a rich statistical summary of the hash's bucket layout.

Σύνοψη

```
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 barcharts.

Σύνοψη

```
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 barcharts 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.

Σύνοψη

```
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`.

Σύνοψη

```
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`.

Σύνοψη

```
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.

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.

Σύνοψη

```
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.

Σύνοψη

```
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.

Σύνοψη

```
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.

Σύνοψη

```
lock_hash_recurse(%hash);
```

Recurses 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.

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.

hidden_keys

List the «hidden» placeholder keys of a restricted hash - legal keys that currently have no value assigned.

Σύνοψη

```
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.

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.

all_keys

Split a restricted hash's keys into visible and hidden buckets, writing the results into the caller's arrays.

Σύνοψη

```
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.

Σύνοψη

```
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`.

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.

5.2.15 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().

5.2.16 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)

sockatmark

IO.xs:567-602 sockatmark(sock) Uses SIOCATMARK ioctl on Linux (HAS_SOCKATMARK path uses sockatmark(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.

5.2.17 IPC

Namespace containing the following modules:

- `IPC::SysV` - System V IPC constants and the system calls Perl's builtins do not cover.

IPC::SysV

System V IPC constants and the system calls Perl's builtins do not cover.

Perl's core already provides `msgget/msgctl/msgsnd/msgrcv`, `semget/semctl/semop` and `shmget/shmctl/shmread/shmwrite` as builtins (pp_sys.c). What the `IPC::SysV` XS adds on top is the constant table (`IPC_CREAT`, `SEM_UNDO`, `SHM_RDONLY`, ...), the `ftok` key generator, shared-memory attach/detach, and the `pack/unpack` pair each of `IPC::Msg::stat`, `IPC::Semaphore::stat` and `IPC::SharedMem::stat` uses to move a `struct msqid_ds` / `semid_ds` / `shmid_ds` across the Perl boundary.

This is the XS half only. `IPC/SysV.pm` still loads from disk (`xs_only: true`) and supplies `@EXPORT_OK`, `%EXPORT_TAGS` and the `AUTOLOAD` that turns a constant name into a call to `_constant`. `IPC::Msg`, `IPC::Semaphore` and `IPC::SharedMem` are pure Perl on top of it and need no native half at all.

Functions

Other Functions

ftok

`ftok(path, id = undef)` - SysV.xs line 307.

memread

`memread(addr, sv, pos, size)` - SysV.xs line 345.

memwrite

`memwrite(addr, sv, pos, size)` - SysV.xs line 370.

shmat

`shmat(id, addr, flag)` - SysV.xs line 388. Returns the attach address as a PV holding the raw pointer bytes, matching sv2addr's expectation.

shmdt

`shmdt(addr)` - SysV.xs line 409.

constant

`_constant($name)` - const-xs.inc.

5.2.18 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.

uniqstr

Remove subsequent duplicates by string equality, treating `undef` as the empty string.

uniqnum

Remove subsequent duplicates by numerical equality, preserving order.

uniqint

Remove subsequent duplicates by integer equality, preserving order.

Διάφορα

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.

Σύνοψη

```
my $total = sum @values;
my $total = sum 1..10;           # 55
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `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.

Σύνοψη

```
my $total = sum0 @values;
my $total = sum0 ();              # 0, not undef
```

Τι επιστρέφεται

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.

Παραδείγματα

```
my $n = sum0 1..10;               # 55
my $n = sum0 ();                  # 0
my $n = sum0 grep { $_ > 0 } @x; # total of positives, 0 if none
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `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.

Σύνοψη

```
my $lo = min @values;
my $lo = min 3, 9, 12;            # 3
```

Τι επιστρέφεται

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`.

Παραδείγματα

```
my $n = min 1..10;                  # 1
my $n = min @bar, @baz;             # smallest across concatenated
 ↪lists
my $n = min();                      # undef
```

Οριακές περιπτώσεις

- Empty list returns `undef`.
- Strings are compared numerically (use `minstr` for lexical order).

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `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.

Σύνοψη

```
my $hi = max @values;
my $hi = max 3, 9, 12;              # 12
```

Τι επιστρέφεται

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`.

Παραδείγματα

```
my $n = max 1..10;                  # 10
my $n = max @bar, @baz;             # largest across concatenated lists
my $n = max();                      # undef
```

Οριακές περιπτώσεις

- Empty list returns `undef`.
- Strings are compared numerically (use `maxstr` for lexical order).

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `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.

Σύνοψη

```
my $p = product @values;
my $p = product 1..10;          # 3628800
```

Τι επιστρέφεται

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).

Παραδείγματα

```
my $n = product 3, 9, 12;       # 324
my $n = product();              # 1
my $n = product map { $_->weight } @items;
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `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.

Σύνοψη

```
my @first = head $size, @list;
my @first = head 2, qw( foo bar baz );   # ('foo', 'bar')
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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
```

Οριακές περιπτώσεις

- `$size >= @list` returns the whole list.
- `$size == 0` or `-$size >= @list` returns the empty list.

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `tail` - mirror operation, taking from the end.
- `sample` - random selection without positional bias.

tail

Return the last `$size` elements of a list.

Σύνοψη

```
my @last = tail $size, @list;
my @last = tail 2, qw( foo bar baz );    # ('bar', 'baz')
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `head` - mirror operation, taking from the start.
- `sample` - random selection without positional bias.

shuffle

Return the input list in a uniformly random order.

Σύνοψη

```
my @mixed = shuffle @values;
my @deck  = shuffle 0..51;       # a shuffled deck
```

Τι επιστρέφεται

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.

Παραδείγματα

```
my @picked = shuffle @candidates;
my ($winner) = shuffle @entries;       # pick one at random
local $List::Util::RAND = sub { 0.5 }; # deterministic for tests
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `sample` - take a fixed count without replacement.
- `head` - deterministic prefix selection.

sample

Pick `$count` distinct elements from the list at random.

Σύνοψη

```
my @picks = sample $count, @values;
my @three = sample 3, 1..100;
```

Τι επιστρέφεται

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()`.

Παραδείγματα

```
my @preview = sample 5, @articles;
my ($winner) = sample 1, @entries;
my @all     = sample 999, qw( a b c );   # ('c','a','b') say
```

Οριακές περιπτώσεις

- `$count <= 0` returns the empty list.
- Fewer inputs than `$count` returns all inputs, shuffled.

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `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.

Σύνοψη

```
my @subset = uniq    @values;
my @subset = uniqstr @values;
my $count  = uniq    @values;    # scalar context - count only
```

Τι επιστρέφεται

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`.

Παραδείγματα

```
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
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `uniqnum` - compare as numbers rather than as strings.
- `uniqint` - compare as integers after truncation.

uniqstr

Remove subsequent duplicates by string equality, treating `undef` as the empty string.

Synopsis

```
my @subset = uniqstr @values;
my $count  = uniqstr @values;    # scalar context - count only
```

What you get back

In list context, the unique elements in first-seen order; in scalar context, their count. Unlike `uniq`, an `undef` element is indistinguishable from `""` and comes back as `""`.

See also

- `uniq` - identical, except `undef` is kept as `undef`.
- `uniqnum` / `uniqint` - numerical comparisons.

uniqnum

Remove subsequent duplicates by numerical equality, preserving order.

Σύνοψη

```
my @subset = uniqnum @values;
my $count  = uniqnum @values;    # scalar context - count only
```

Τι επιστρέφεται

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`.

Παραδείγματα

```
my @u = uniqnum  1, 1.0, '1', 2;   # (1, 2)
my @u = uniqnum  0, -0.0;          # (0)
my $n = uniqnum  1, 2, 3, 2, 1;    # 3
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `uniq` - string-equality deduplication.
- `uniqint` - integer-equality deduplication (ignores fractions).

uniqint

Remove subsequent duplicates by integer equality, preserving order.

Σύνοψη

```
my @subset = uniqint @values;
my $count  = uniqint @values;    # scalar context - count only
```

Τι επιστρέφεται

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`.

Παραδείγματα

```
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
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `uniqnum` - full numerical (float) equality.
- `uniq` - string-equality deduplication.

minstr

Return the lexically smallest value in the list.

Σύνοψη

```
my $s = minstr @values;
my $s = minstr 'A'..'Z';          # 'A'
```

Τι επιστρέφεται

The first (original) list element whose stringification compares less than every other element's stringification under `lt` (byte-wise comparison). Empty list yields `undef`.

Παραδείγματα

```
my $s = minstr 'hello', 'world';    # 'hello'
my $s = minstr qw( foo bar baz );   # 'bar'
my $s = minstr ();                  # undef
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `maxstr` - string-wise maximum.
- `min` - numerical minimum.

maxstr

Return the lexically largest value in the list.

Σύνοψη

```
my $s = maxstr @values;
my $s = maxstr 'A'..'Z';          # 'Z'
```

Τι επιστρέφεται

The first (original) list element whose stringification compares greater than every other element's stringification under `gt` (byte-wise comparison). Empty list yields `undef`.

Παραδείγματα

```
my $s = maxstr 'hello', 'world';    # 'world'
my $s = maxstr qw( foo bar baz );   # 'foo'
my $s = maxstr ();                  # undef
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `minstr` - string-wise minimum.
- `max` - numerical maximum.

pairs

Group an even-sized list into key/value pair objects.

Σύνοψη

```
my @pairs = pairs @kvlist;
for my $p ( pairs %hash ) {
    my ($k, $v) = @$p;            # or $p->key / $p->value
}
```

Τι επιστρέφεται

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`.

Παραδείγματα

```
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;
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `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.

Σύνοψη

```
my @kvlist = unpairs @pairs;
my %hash   = unpairs @pairs;
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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;
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `pairs` - the inverse direction.

pairkeys

Extract just the keys from an even-sized key/value list.

Σύνοψη

```
my @keys = pairkeys @kvlist;
my @keys = pairkeys %hash;
```

Τι επιστρέφεται

A list of every even-indexed element (indices `0`, 2, 4, ...). Equivalent to but more efficient than

```
pairmap { $a } @kvlist
```

Παραδείγματα

```
my @k = pairkeys one => 1, two => 2;    # ('one', 'two')
my @k = pairkeys %ENV;                  # names of every env var
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `pairvalues` - the value-side counterpart.
- `pairs` - keep keys and values bound together.

pairvalues

Extract just the values from an even-sized key/value list.

Σύνοψη

```
my @values = pairvalues @kvlist;
my @values = pairvalues %hash;
```

Τι επιστρέφεται

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
```

Παραδείγματα

```
my @v = pairvalues one => 1, two => 2;   # (1, 2)
my @v = pairvalues %ENV;                 # values of every env var
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `pairkeys` - the key-side counterpart.
- `pairs` - keep keys and values bound together.

pairmap

Run a block for each pair and concatenate the results.

Σύνοψη

```
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.

Τι επιστρέφεται

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.

Παραδείγματα

```
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`
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `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.

Σύνοψη

```
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.

Τι επιστρέφεται

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.

Παραδείγματα

```
my %big     = pairgrep { $b > 100 } %totals;
my @uppers  = pairgrep { $a =~ /^[A-Z]/ } %h;
my $n_big   = pairgrep { $b > 100 } %totals;   # scalar: pair count
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `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.

Σύνοψη

```
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.

Τι επιστρέφεται

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.

Παραδείγματα

```
my ($k, $v) = pairfirst { $a =~ /^upper_/ } %config;
if ( pairfirst { $b > 1_000 } %totals ) {
    warn "at least one big entry";
}
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `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`.

Σύνοψη

```
my @out = mesh         \@a, \@b, \@c;
my @out = mesh_longest \@a, \@b, \@c;
my %h   = mesh \@keys, \@values;
```

Τι επιστρέφεται

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`.

Παραδείγματα

```
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
```

Οριακές περιπτώσεις

- An input that is not an arrayref contributes only `undef`s.
- Empty input list returns an empty list.

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `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.

Σύνοψη

```
my @out = mesh_shortest \@a, \@b, \@c;
```

Τι επιστρέφεται

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.

Παραδείγματα

```
my @r = mesh_shortest [1..3], ['a','b'];   # (1,'a', 2,'b')
my @r = mesh_shortest [1..3], [];          # ()
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `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`.

Σύνοψη

```
my @rows = zip         \@a, \@b, \@c;
my @rows = zip_longest \@a, \@b, \@c;
foreach ( zip \@xs, \@ys ) {
    my ($x, $y) = @$_;
}
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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])
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `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.

Σύνοψη

```
my @rows = zip_shortest \@a, \@b, \@c;
```

Τι επιστρέφεται

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.

Παραδείγματα

```
my @r = zip_shortest [1..3], ['a','b'];   # ([1,'a'], [2,'b'])
my @r = zip_shortest [1..3], [];          # ()
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `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.

Σύνοψη

```
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.

Τι επιστρέφεται

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.

Παραδείγματα

```
my $concat = reduce { "$a,$b" } 'a'..'c';    # 'a,b,c'
my $best   = reduce { $a->score > $b->score ? $a : $b } @players;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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;
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `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.

Σύνοψη

```
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.

Τι επιστρέφεται

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.

Παραδείγματα

```
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;
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `reduce` - when only the final result is needed.

first

Return the first list element for which a block returns true.

Σύνοψη

```
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.

Τι επιστρέφεται

The matching element (the original value, not the block's return). If no element matches - or the list is empty - returns `undef`.

Παραδείγματα

```
my $first_defined = first { defined $_ } @list;
my $first_big     = first { $_ > $threshold } @measurements;
my $first_admin   = first { $_->is_admin } @users;
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- 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.

Σύνοψη

```
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.

Τι επιστρέφεται

1 if any element satisfied the block, `0` otherwise.

Παραδείγματα

```
warn "has empty"   if any { $_ eq '' } @values;
die  "has admin"   if any { $_->is_admin } @users;
my $ok = any { /^--help$/ } @ARGV;
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `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).

Σύνοψη

```
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»).

Τι επιστρέφεται

1 if every element satisfied the block, `0` otherwise.

Παραδείγματα

```
die "found empty string" unless all { length } @strings;
my $ready = all { $_->is_ready } @workers;
my $yes   = all { $_ > 0 } ();           # 1 - vacuous truth
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- any - require at least one match.
- `notall` - inverse: at least one failure.
- none - inverse: zero matches.

καμία

True if the block returns true for no element (including empty list).

Σύνοψη

```
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.

Τι επιστρέφεται

1 if every element failed the block, `0` otherwise.

Παραδείγματα

```
die "negatives present" unless none { $_ < 0 } @measurements;
my $clean = none { /ERROR/ } @log_lines;
my $none  = none { 1 } ();               # 1 - nothing to fail
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- 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.

Σύνοψη

```
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.

Τι επιστρέφεται

1 if at least one element failed the block, `0` otherwise.

Παραδείγματα

```
my $incomplete = notall { defined } @row_cells;
warn "stale" if notall { $_->is_fresh } @entries;
my $false = notall { 1 } ();              # 0 - nothing to fail
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `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.

Σύνοψη

```
use Sub::Util qw(subname);
my $name = subname \&some_sub;          # 'Pkg::some_sub'
my $name = subname sub { 1 };           # 'main::__ANON__'
```

Τι επιστρέφεται

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`.

Παραδείγματα

```
package My::Pkg;
sub hello { 1 }
print Sub::Util::subname(\&hello);    # 'My::Pkg::hello'
```

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `set_subname` - rename a code reference (useful for stack traces).

sub_util_set_subname

Assign a name to a subroutine reference in place.

Σύνοψη

```
use Sub::Util qw(set_subname);
my $code = set_subname 'My::Pkg::name', sub { ... };
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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 { ... };
}
```

Οριακές περιπτώσεις

- Croaks with `Not a subroutine reference` for non-CODE arguments.
- Croaks with `set_subname: not enough arguments` on missing args.

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `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.

Σύνοψη

```
use Sub::Util qw(set_prototype);
my $code = set_prototype '$$', \&adder;
my $code = set_prototype undef, \&adder;   # clear the prototype
```

Τι επιστρέφεται

The same code reference, with its prototype replaced by `$proto` (a string like `'$$'`, `'&@'`, `'\@'`) or cleared when `$proto` is `undef`.

Παραδείγματα

```
set_prototype '$$', \&my_add;     # force two scalar args
set_prototype '&@',  \&wrapper;    # block followed by a list
set_prototype undef, \&free_form;  # no prototype
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `set_subname` - rename a code reference.
- `prototype` - perl built-in that reads a prototype.

5.2.19 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.

Σύνοψη

```
use MIME::Base64;
my $encoded = encode_base64($bytes);          # wrapped at 76, "\n"-
 ↪terminated
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
my $oneline = encode_base64($bytes, "");     # no wrapping, no␣
↪trailing EOL
my $crlf    = encode_base64($bytes, "\r\n"); # MIME-style CRLF␣
↪wrapping
```

Τι επιστρέφεται

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 `=`.

Παραδείγματα

```
encode_base64("Aladdin:open sesame");

## "QWxhZGRpbjpvcGVuIHNlc2FtZQ==\n"

encode_base64("Aladdin:open sesame", "");

## "QWxhZGRpbjpvcGVuIHNlc2FtZQ=="

encode_base64("\x00\x00\x00");

## "AAAA\n"
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `MIME::Base64` 3.16.

Δείτε επίσης

- `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.

Σύνοψη

```
use MIME::Base64;
my $bytes = decode_base64($encoded);
```

Τι επιστρέφεται

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.

Παραδείγματα

```
decode_base64("QWxhZGRpbjpvcGVuIHNlc2FtZQ==");

## "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"
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `MIME::Base64` 3.16.

Δείτε επίσης

- `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.

Σύνοψη

```
use MIME::Base64 qw(encode_base64url);
my $token = encode_base64url($bytes);
```

Τι επιστρέφεται

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.

Παραδείγματα

```
encode_base64url("Aladdin:open sesame");

## "QWxhZGRpbjpvcGVuIHNlc2FtZQ"

encode_base64url("\xFB\xFF");

## "-_8"                (note the '-' and '_' where '+' and '/'␣
↪would appear)

encode_base64url("");

## ""
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `MIME::Base64` 3.16.

Δείτε επίσης

- `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.

Σύνοψη

```
use MIME::Base64 qw(decode_base64url);
my $bytes = decode_base64url($token);
```

Τι επιστρέφεται

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.

Παραδείγματα

```
decode_base64url("QWxhZGRpbjpvcGVuIHNlc2FtZQ");

## "Aladdin:open sesame"

decode_base64url("-_8");

## "\xFB\xFF"

decode_base64url("QWxhZGRpbjpvcGVuIHNlc2FtZQ==");   # padding also␣
↪OK

## "Aladdin:open sesame"
```

Οριακές περιπτώσεις

- 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`.

Διαφορές από το upstream

Fully compatible with upstream `MIME::Base64` 3.16.

Δείτε επίσης

- `encode_base64url` - the inverse operation.
- `decode_base64` - standard variant.

encoded_base64_length

Return the length `encode_base64` would produce, without doing the encoding.

Σύνοψη

```
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
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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")
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `MIME::Base64` 3.16.

Δείτε επίσης

- `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.

Σύνοψη

```
use MIME::Base64 qw(decoded_base64_length);
my $n = decoded_base64_length($encoded);
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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)
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `MIME::Base64` 3.16.

Δείτε επίσης

- `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.

Σύνοψη

```
$encoded = encode_qp($str);
$encoded = encode_qp($str, $eol);
$encoded = encode_qp($str, $eol, $binmode);
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream `MIME::QuotedPrint` 3.16.

Δείτε επίσης

- `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.

Σύνοψη

```
$decoded = decode_qp($encoded);
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream MIME::QuotedPrint 3.16.

Δείτε επίσης

- encode_qp - produce the encoded form.
- MIME::Base64::decode_base64 - the counterpart for binary payloads.

5.2.20 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.

Αριθμητική

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 << $e` on a `Math::GMP` value; the operator form goes through `blshift`. `$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` (`$x mod 2**$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 `$x` mod `$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 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.

Σύνοψη

```
my $x = Math::GMP->new(123);
my $y = Math::GMP->new('12345678901234567890123456789O');
my $z = Math::GMP->new('abcd', 36);          # base 36
```

Τι επιστρέφεται

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.

Οριακές περιπτώσεις

- 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.

Σύνοψη

```
my $v = Math::GMP::_gmp_lib_version();
print "running libgmp $v\n";   # e.g. "running libgmp 6.3.0"
```

Τι επιστρέφεται

A dotted-decimal string like `"6.3.0"`. Feed it to `version->parse` if you need numeric comparison.

Διαφορές από το 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.

Σύνοψη

```
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.

Σύνοψη

```
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.

Σύνοψη

```
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.

Σύνοψη

```
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.

Σύνοψη

```
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`.

Σύνοψη

```
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, 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.

Σύνοψη

```
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.

Οριακές περιπτώσεις

- `$float == 0.0` → a `Math::GMP` of `0`.
- Negative `$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.

Σύνοψη

```
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 `$x` mod `$y`.

Σύνοψη

```
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, $y) != 1`), returns `0`. Behaviour is undefined when `$y` is `0`.

Οριακές περιπτώσεις

- `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`.

Σύνοψη

```
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`.

Σύνοψη

```
my $x = Math::GMP->new(7);
my $v = $x->probab_prime(10);   # 2 - definitely prime
```

Τι επιστρέφεται

- `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`.

Σύνοψη

```
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`.

Σύνοψη

```
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.

Σύνοψη

```
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.

Σύνοψη

```
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.

Σύνοψη

```
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`.

Σύνοψη

```
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).

Σύνοψη

```
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.

Σύνοψη

```
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.

Σύνοψη

```
my $x = Math::GMP->new(100);
my ($root, $rem) = $x->brootrem(3);    # 4, 36

## because 4**3 + 36 == 100
```

The returned list satisfies $root ** $n + $rem == $x exactly. Use when you need both pieces in one pass - calling broot and computing the remainder yourself doubles the work.

Διαφορές από το 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.

Σύνοψη

```
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`.

Σύνοψη

```
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.

5.2.21 Opcode

Name, group and mask the interpreter's opcodes - the machinery `Safe` compartments are built from.

An *opset* is a bit string with one bit per opcode, `(MAXO + 7) / 8` bytes wide. Operators are named individually (`"print"`, `"backtick"`) or by tag (`":base_core"`, `":subprocess"`); `Opcode` converts between names, tags and opsets, and installs an opset into `PL_op_mask`, which the compiler consults to refuse masked ops.

This is the XS half only. `Opcode.pm` still loads from disk (`xs_only: true`) and supplies the exported wrappers (`opset_to_hex`, `opdump`, `ops_to_opset`) plus the standard tag definitions it builds from its own `__DATA__` section via `define_optag`.

Functions

Other Functions

safe_pkg_prep

Opcode.xs `_safe_pkg_prep($Package)`: make the compartment stash believe it is `main`, and wire its _ to the global *_ so $_ still works.

safe_call_sv

Opcode.xs `_safe_call_sv($Package, $mask, $codesv)`: run $codesv with PL_op_mask localised to $mask and the compartment stash installed as PL_defstash/PL_curstash, so compiled code inside it sees the compartment as `main`.

verify_opset

`verify_opset($opset, $fatal = 0)` - Opcode.xs line 337.

invert_opset

`invert_opset($opset)` - Opcode.xs line 347. Clears the bits past MAXO in the last byte so the result never names a nonexistent opcode.

opset_to_ops

`opset_to_ops($opset, $desc = 0)` - Opcode.xs line 367.

opset

`opset(@names_or_opsets)` - Opcode.xs line 391. A leading `!` on a name turns the bit off instead of on.

opdesc

`opdesc(@names)` - Opcode.xs line 461.

define_optag

`define_optag($tag, $mask)` - Opcode.xs line 504.

empty_opset

`empty_opset()` - Opcode.xs line 517.

full_opset

`full_opset()` - Opcode.xs line 524.

opmask_add

`opmask_add($opset)` - Opcode.xs line 532. The PREINIT allocates PL_op_mask on first use; opmask_add itself refuses a null mask.

opcodes

`opcodes()` - Opcode.xs line 541. List context is unimplemented upstream too, and croaks.

opmask

`opmask()` - Opcode.xs line 552: the current PL_op_mask as an opset.

5.2.22 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.

Αρθρώματα

- `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.

Σύνοψη

```
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.

setinftobad

`setinftobad($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 setinftobad 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.

Σύνοψη

```
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.

Παραδείγματα

```
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␣
↪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.

Σύνοψη

```
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_dataref

`get_dataref($pdl)` - return a SCALAR ref to a byte buffer aliasing the pdl's data. Mirrors `PDL::Core::get_dataref` from `Core.xs:1141-1156`.

upd_data

`upd_data($pdl, $keep_datasv=0)` - Perl wrote into the dataref; 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?, @pdls)` - 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.

Παραδείγματα

```
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 → 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` → `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.

Σύνοψη

```
use PDL;
use PDL::FFT;
my $re = sequence(8);
my $im = zeroes(8);
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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.).

Σύνοψη

```
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.).

Σύνοψη

```
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.

Σύνοψη

```
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`.

Παραδείγματα

```
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.

Σύνοψη

```
use PDL;
my $a = sequence(10);
my $s = $a->slice("2:5");
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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)
```

uniqvec

```
uniqvec($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:

Προσδ.	Σημασία
`""` / `:` / `X`	keep axis as-is
`n`	a single index (keeps the axis, length 1)
`(n)`	a single index, *squish* (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`.

Παραδείγματα

```
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]

## 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
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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.).

Σύνοψη

```
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` stringification: 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 (*`pdl.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.

Σύνοψη

```
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: rank = 1 + p*(n-1); interp between sorted[floor(rank)-1] and sorted[ceil(rank)-1]. The c2rust version produces rounding errors on the boundary (17.0000000007 != 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 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).

Τι επιστρέφεται

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	Τύπος αποτελέσματος
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.

Παραδείγματα

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);    # 3×4 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"`.

Οριακές περιπτώσεις

- **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.

Δείτε επίσης

- `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

5.2.23 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 `$x` as an integer, `trunc(log2(|$x|))`.

Edge cases

- `ilogb(0)` returns `FP_ILOGB0` (typically `INT_MIN`).
- `ilogb(NaN)` returns `FP_ILOGBNAN`.
- `ilogb(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.

strxfrm

Locale-transformed form of a string, for repeated collation.

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`.

Χρόνος

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.

Σύνοψη

```
my $n = POSIX::abs(-7);    # 7
```

Τι επιστρέφεται

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.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

frexp

Split a floating-point number into mantissa and exponent.

Σύνοψη

```
my ($frac, $exp) = POSIX::frexp($x);

## $x == $frac * 2**$exp, with 0.5 <= |$frac| < 1 (or $frac == 0).
```

Τι επιστρέφεται

A two-element list: the fractional part as a double and the exponent as an integer.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

Δείτε επίσης

- `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.

Σύνοψη

```
my ($frac, $int) = POSIX::modf(3.75);   # (0.75, 3.0)
```

Τι επιστρέφεται

A two-element list `($frac, $int)`. Both parts carry the sign of the input.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

Δείτε επίσης

- `frexp` - split into mantissa and exponent instead.

fma

Fused multiply-add: `$x * $y + $z` with a single rounding.

Σύνοψη

```
my $r = POSIX::fma($x, $y, $z);
```

Τι επιστρέφεται

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))`.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

nan

Build a quiet NaN, optionally with a payload string.

Σύνοψη

```
my $q = POSIX::nan();         # quiet NaN
my $p = POSIX::nan("0x42");   # NaN with payload where supported
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

remquo

Floating-point remainder with partial quotient.

Σύνοψη

```
my ($rem, $quo) = POSIX::remquo($x, $y);
```

Τι επιστρέφεται

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).

Διαφορές από το upstream

Fully compatible with upstream POSIX.

jn

Bessel function of the first kind, integer order `$n`.

Σύνοψη

```
my $y = POSIX::jn($n, $x);
```

Διαφορές από το upstream

Fully compatible with upstream POSIX.

Δείτε επίσης

- `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`.

Σύνοψη

```
my $y = POSIX::yn($n, $x);
```

Οριακές περιπτώσεις

- `yn($n, 0)` returns negative infinity (the function has a singularity at zero).

Διαφορές από το upstream

Fully compatible with upstream POSIX.

Δείτε επίσης

- `y0`, `y1` - fixed-order variants.
- `jn` - Bessel function of the first kind.

fpclassify

Classify a floating-point number.

Σύνοψη

```
use POSIX qw(fpclassify FP_NAN FP_INFINITE FP_ZERO FP_SUBNORMAL FP_
 ↪NORMAL);
my $k = fpclassify($x);
```

Τι επιστρέφεται

An integer matching one of the `FP_*` constants. Compare against the symbolic names, not a hardcoded number - the values are libc implementation details.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

Δείτε επίσης

- `isnan`, `isinf`, `isfinite`, `isnormal` - narrower single-class tests.

fesetround

Change the floating-point rounding direction.

Σύνοψη

```
use POSIX qw(fesetround FE_DOWNWARD);
fesetround(FE_DOWNWARD);
```

Τι επιστρέφεται

`0` on success, non-zero if the requested mode is not supported.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

wexitstatus

Exit code of a child that terminated normally.

Σύνοψη

```
system($cmd);
my $code = POSIX::WEXITSTATUS($?);
```

Τι επιστρέφεται

The low 8 bits of the child's `exit()` argument. Only meaningful when `WIFEXITED($status)` is true.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

strtod

Parse a double from the start of a string, locale-aware.

Σύνοψη

```
use POSIX qw(strtod);
my ($num, $unparsed) = strtod("3.14abc");   # (3.14, 3)
```

Τι επιστρέφεται

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.

Οριακές περιπτώσεις

- Input that doesn't start with a number: returns `(0, length)`.
- Overflow: returns `HUGE_VAL` and sets `$!` to `ERANGE`.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

Δείτε επίσης

- `strtol`, `strtoul` - integer variants.

strtol

Parse a signed integer from the start of a string, in a given base.

Σύνοψη

```
use POSIX qw(strtol);
my ($n, $unparsed) = strtol("0x2a rest", 0);   # (42, 5)
```

Τι επιστρέφεται

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.

Οριακές περιπτώσεις

- Invalid base: returns `(undef, undef)` and sets `$!` to `EINVAL`.
- Overflow: returns `LONG_MIN` / `LONG_MAX` and sets `$!` to `ERANGE`.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

Δείτε επίσης

- `strtoul` - unsigned variant.
- `strtod` - floating-point variant.

strtoul

Parse an unsigned integer from the start of a string, in a given base.

Σύνοψη

```
use POSIX qw(strtoul);
my ($n, $unparsed) = strtoul("255 end", 10);
```

Οριακές περιπτώσεις

- Invalid base: returns `(undef, undef)` and sets `$!` to `EINVAL`.
- Negative input: parsed and two's-complement-cast to unsigned, matching C `strtoul`.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

Δείτε επίσης

- `strtol` - signed variant.

strxfrm

Locale-transformed form of a string, for repeated collation.

Synopsis

```
use POSIX qw(strxfrm);
my @sorted = map { $_->[1] }
             sort { $a->[0] cmp $b->[0] }
             map { [ strxfrm($_), $_ ] } @words;
```

What you get back

A byte string whose plain `cmp` ordering equals `strcoll`'s ordering of the originals under the current `LC_COLLATE` locale. Worth it when the same string is compared many times, as in a sort.

Differences from upstream

Fully compatible with upstream POSIX.

strerror

Message string for an `errno` value.

Σύνοψη

```
use POSIX qw(strerror);
print strerror($!);
```

Τι επιστρέφεται

A byte string in the current locale, like `"No such file or directory"`. For unknown error numbers, libc returns an implementation-defined placeholder.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

sprintf

Format a string POSIX-style, identical to built-in `sprintf`.

Σύνοψη

```
use POSIX qw(sprintf);
my $line = sprintf("%-10s %6.2f", $name, $price);
```

Τι επιστρέφεται

A single scalar holding the formatted string. Conversion specifiers, flags, width, precision, and length modifiers are whatever your built-in `sprintf` supports.

Οριακές περιπτώσεις

- 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`.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

setpgid

Put process `$pid` into process group `$pgid`.

Σύνοψη

```
POSIX::setpgid($pid, $pgid);
```

Τι επιστρέφεται

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.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

uname

Kernel and machine identification as a five-element list.

Σύνοψη

```
use POSIX qw(uname);
my ($sysname, $nodename, $release, $version, $machine) = uname();
```

Τι επιστρέφεται

Five strings in order: sysname (e.g. `"Linux"`), nodename (hostname), release (kernel release), version (kernel version string), and machine (architecture, e.g. `"x86_64"`).

Οριακές περιπτώσεις

- On uname failure, returns an empty list.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

sysconf

Runtime system configuration value for the given _SC_* name.

Σύνοψη

```
use POSIX qw(sysconf _SC_OPEN_MAX _SC_PAGESIZE);
my $max_fd = sysconf(_SC_OPEN_MAX);
```

Τι επιστρέφεται

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).

Διαφορές από το upstream

Fully compatible with upstream POSIX.

access

Check the caller's permissions on a path.

Σύνοψη

```
use POSIX qw(access R_OK W_OK);
print "readable\n" if access($path, R_OK);
```

Τι επιστρέφεται

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`.

Οριακές περιπτώσεις

- Check uses real (not effective) UID/GID, so is vulnerable to TOCTOU races; prefer trying the operation and handling failure.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

open

Open a file by path, returning a raw file descriptor.

Σύνοψη

```
use POSIX qw(open O_RDONLY O_CREAT O_WRONLY);
my $fd = open($path, O_RDONLY)
    or die "open: $!";
```

Τι επιστρέφεται

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`.

Διαφορές από το upstream

Fully compatible with upstream POSIX. Third argument `$mode` defaults to `0666` when opening with `O_CREAT` and no mode given.

Δείτε επίσης

- `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.

Σύνοψη

```
POSIX::dup2($fd, 1);   # redirect stdout to $fd
```

Τι επιστρέφεται

The target fd on success, `undef` on failure. Either argument being negative is rejected with `$!` set to `EBADF`.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

lseek

Reposition the read/write offset of an open fd.

Σύνοψη

```
use POSIX qw(lseek SEEK_SET SEEK_CUR SEEK_END);
my $pos = lseek($fd, 0, SEEK_CUR);    # current offset
```

Τι επιστρέφεται

The new absolute offset on success, `-1` on failure (with `$!` set).

Διαφορές από το upstream

Fully compatible with upstream POSIX.

pipe

Create an anonymous pipe and return its two fds.

Σύνοψη

```
use POSIX qw(pipe);
my ($rd, $wr) = pipe();
```

Τι επιστρέφεται

A two-element list `($read_fd, $write_fd)` on success. On failure, a single `undef` is returned and `$!` is set.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

ανάγνωση

Read up to `$nbytes` bytes from `$fd` into `$buffer`.

Σύνοψη

```
use POSIX qw(read);
my $buf = "";
my $n = read($fd, $buf, 4096);
```

Τι επιστρέφεται

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.

Οριακές περιπτώσεις

- Partial reads are normal on pipes, sockets, and terminals; always check the return length.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

write

Write bytes from `$buffer` to `$fd`.

Σύνοψη

```
POSIX::write($fd, $data, length($data));
```

Τι επιστρέφεται

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.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

strftime

Format a broken-down time as a string using a format template.

Σύνοψη

```
use POSIX qw(strftime);
my @tm = localtime;
my $str = strftime("%Y-%m-%d %H:%M:%S", @tm);
```

Τι επιστρέφεται

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`.

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

Δείτε επίσης

- `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.

Σύνοψη

```
use POSIX qw(mktime);
my $t = mktime($sec, $min, $hour, $mday, $mon, $year - 1900);
```

Τι επιστρέφεται

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).

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

tzname

Short names of the current standard and daylight timezones.

Σύνοψη

```
use POSIX qw(tzname);
my ($std, $dst) = tzname();
```

Τι επιστρέφεται

A two-element list. Both strings are derived from `TZ` - the call also implicitly refreshes the timezone cache.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

times

Raw wall + CPU times of the process and its children, as a five-element list.

Σύνοψη

```
my ($real, $user, $sys, $cuser, $csys) = POSIX::times();
```

Τι επιστρέφεται

Five integer tick counts, in `sysconf(_SC_CLK_TCK)` units - raw, not divided by the ticks-per-second constant. Divide if you want seconds.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

asctime

Canonical English timestamp string from a broken-down time.

Σύνοψη

```
use POSIX qw(asctime);
my @tm = localtime;
print asctime(@tm);   # "Wed Apr 22 14:30:00 2026\n"
```

Τι επιστρέφεται

A 25-byte string **including the trailing newline**, or `undef` if nine arguments are not supplied.

Διαφορές από το 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.

Σύνοψη

```
use POSIX qw(ctime);
print ctime(time);   # "Wed Apr 22 14:30:00 2026\n"
```

Τι επιστρέφεται

A 25-byte string including the trailing newline, or `undef` on conversion failure.

Διαφορές από το upstream

Fully compatible with upstream POSIX. The trailing newline is preserved.

Δείτε επίσης

- `asctime` - same formatting, but from a broken-down time list.

nice

Adjust the current process's scheduling niceness by `$incr`.

Σύνοψη

```
POSIX::nice(5);   # lower priority
```

Τι επιστρέφεται

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 `$!`.

Οριακές περιπτώσεις

- Positive increments lower priority; only root can pass negative values to raise priority.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

setlocale

Query or change the program's locale setting.

Σύνοψη

```
use POSIX qw(setlocale LC_ALL LC_NUMERIC);
setlocale(LC_ALL, "");              # honour environment
my $cur = setlocale(LC_NUMERIC);    # query current
```

Τι επιστρέφεται

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.

Οριακές περιπτώσεις

- Changing a category can affect formatting of numbers, collation, date strings, and error messages throughout the process.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

Δείτε επίσης

- `localeconv` - inspect the active locale's numeric and monetary conventions.

localeconv

Numeric and monetary formatting rules for the active locale.

Σύνοψη

```
use POSIX qw(localeconv);
my $lc = localeconv();
print "Decimal: $lc->{decimal_point}\n";
```

Τι επιστρέφεται

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`).

Οριακές περιπτώσεις

- 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).

Διαφορές από το upstream

Fully compatible with upstream POSIX.

Δείτε επίσης

- `setlocale` - pick which locale those rules come from.

fpathconf

Runtime limit for a configuration variable on an open fd.

Σύνοψη

```
use POSIX qw(fpathconf _PC_NAME_MAX);
my $max = fpathconf($fd, _PC_NAME_MAX);
```

Τι επιστρέφεται

The limit value, or `-1` if the variable is unsupported or unlimited (check `$!` to distinguish).

Διαφορές από το upstream

Fully compatible with upstream POSIX.

constant

Look up a POSIX constant by name.

Σύνοψη

```
my $enoent = POSIX::constant("ENOENT");
```

Τι επιστρέφεται

The integer or floating-point value of the named constant, or `undef` if the name is not known.

Διαφορές από το 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`.

Σύνοψη

```
use POSIX qw(tcflow TCOOFF TCOON);
tcflow($fd, TCOOFF);   # pause output
```

Τι επιστρέφεται

`"0 but true"` on success, `undef` on failure. Negative `$action` sets `$!` to `EINVAL`.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

tcflush

Discard buffered input, output, or both on terminal `$fd`.

Σύνοψη

```
use POSIX qw(tcflush TCIFLUSH TCOFLUSH TCIOFLUSH);
tcflush($fd, TCIOFLUSH);   # drop pending input and output
```

Τι επιστρέφεται

`"0 but true"` on success, `undef` on failure. Negative `$queue_selector` sets `$!` to `EINVAL`.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

termios_new

Create a fresh `POSIX::Termios` object, all fields zeroed.

Σύνοψη

```
use POSIX;
my $t = POSIX::Termios->new;
$t->getattr(fileno(STDIN));
```

Τι επιστρέφεται

A blessed `POSIX::Termios` reference. To fill it with the current terminal settings, follow up with `$t->getattr($fd)`.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

Δείτε επίσης

- `getattr`, `setattr` - populate and apply terminal settings.

termios_getattr

Fill the `POSIX::Termios` object with the current settings of `$fd`.

Σύνοψη

```
$t->getattr(fileno(STDIN));
```

Τι επιστρέφεται

`"0 but true"` on success, `undef` on failure. `$fd` defaults to `0` (stdin).

Διαφορές από το upstream

Fully compatible with upstream POSIX.

termios_setattr

Apply the `POSIX::Termios` settings to `$fd`.

Σύνοψη

```
use POSIX qw(TCSANOW TCSADRAIN);
$t->setattr(fileno(STDIN), TCSANOW);
```

Τι επιστρέφεται

`"0 but true"` on success, `undef` on failure. `$fd` defaults to `0`, `$optional_actions` defaults to `TCSANOW` (apply immediately).

Οριακές περιπτώσεις

- Negative `$optional_actions` sets `$!` to `EINVAL`.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

termios_getcc

Value of a special control character in the termios structure.

Σύνοψη

```
use POSIX qw(VINTR VEOF);
my $intr = $t->getcc(VINTR);   # Ctrl-C byte
```

Οριακές περιπτώσεις

- Indices beyond the `NCCS` bound croak with `Bad getcc subscript`.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

sigset_new

Create a `POSIX::SigSet` object, optionally pre-populated with the given signal numbers.

Σύνοψη

```
use POSIX qw(SIGINT SIGTERM);
my $set = POSIX::SigSet->new(SIGINT, SIGTERM);
```

Τι επιστρέφεται

A blessed `POSIX::SigSet` reference, with `sigemptyset` applied first and then each argument added via `sigaddset`.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

Δείτε επίσης

- `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.

Σύνοψη

```
use POSIX qw(sigprocmask SIG_BLOCK SIG_UNBLOCK SIG_SETMASK);
my $mask = POSIX::SigSet->new(SIGINT);
sigprocmask(SIG_BLOCK, $mask);
```

Τι επιστρέφεται

`"0 but true"` on success, `undef` on failure. When `$oldsigset` is supplied, it is filled with the mask in effect before the call.

Οριακές περιπτώσεις

- 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`.

Διαφορές από το upstream

Fully compatible with upstream POSIX.

Δείτε επίσης

- `sigpending`, `sigsuspend`, `sigaction` - the rest of the POSIX signal API.

sigaction

Install or inspect a signal handler for signal `$sig`.

Σύνοψη

```
use POSIX qw(sigaction SIGINT);
my $action = POSIX::SigAction->new("IGNORE");
my $old = POSIX::SigAction->new;
sigaction(SIGINT, $action, $old);
```

Τι επιστρέφεται

`"0 but true"` on success, `undef` on failure. When `$oldaction` is supplied, it is populated with the previous `HANDLER`, `MASK`, `FLAGS`, and `SAFE` fields.

Οριακές περιπτώσεις

- 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`.

Διαφορές από το 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.

Δείτε επίσης

- `sigprocmask` - change the blocked-signal mask.
- `POSIX::SigSet` - build the `MASK` field of an action.

5.2.24 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.

Σύνοψη

```
use PadWalker qw(peek_my);
my $h = peek_my($level);
```

Τι επιστρέφεται

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.

Παραδείγματα

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.

Οριακές περιπτώσεις

- `$level` defaults to `0` if omitted.
- `$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.

Διαφορές από το upstream

Fully compatible with upstream PadWalker 2.5.

Δείτε επίσης

- `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.

Σύνοψη

```
use PadWalker qw(peek_our);
my $h = peek_our($level);
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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;
}
```

Οριακές περιπτώσεις

- `$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`.

Διαφορές από το upstream

Fully compatible with upstream PadWalker 2.5.

Δείτε επίσης

- `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.

Σύνοψη

```
use PadWalker qw(peek_sub);
my $h = peek_sub(\&some_sub);
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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
```

Οριακές περιπτώσεις

- 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`.

Διαφορές από το upstream

Fully compatible with upstream PadWalker 2.5.

Δείτε επίσης

- `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.

Σύνοψη

```
use PadWalker qw(closed_over);
my $h = closed_over(\&some_sub);
my ($h, $targs) = closed_over(\&some_sub);
```

Τι επιστρέφεται

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.

Παραδείγματα

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";
}
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream PadWalker 2.5.

Δείτε επίσης

- `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.

Σύνοψη

```
use PadWalker qw(set_closed_over);
set_closed_over(\&some_sub, \%replacements);
```

Τι επιστρέφεται

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}`.

Παραδείγματα

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);
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream PadWalker 2.5.

Δείτε επίσης

- `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.

Σύνοψη

```
use PadWalker qw(var_name);
my $name = var_name($level, \$var);
my $name = var_name(\&sub,  \$var);
```

Τι επιστρέφεται

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.

Παραδείγματα

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
```

Οριακές περιπτώσεις

- 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)`.

Διαφορές από το upstream

Fully compatible with upstream PadWalker 2.5.

Δείτε επίσης

- `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.

5.2.25 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 inEncodeCall - 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 Fileno = 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*.

5.2.26 Peta

Namespace containing the following modules:

- `Peta::FFI` - Call C library functions from Perl without writing XS or a binding crate.
- `Peta::XS` - Peta::XS - the interpreter's capability layer for pure Perl.

Peta::FFI

Call C library functions from Perl without writing XS or a binding crate.

`Peta::FFI` is the dynamic FFI layer pperl 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:

Κωδικός	C type	Perl side
v	void	return only; 1 is returned
i	int	ακέραιος
l	long	ακέραιος
L	unsigned long / size_t	ακέραιος
d	double	float
f	float	float
p	const char *	input string
P	mutable buffer	output buffer, auto-allocated
o	opaque pointer	raw address as integer

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.

`o` is the handle type for C APIs built on opaque structs (`PGconn*`, `sqlite3*`, `MYSQL*`): the raw address travels as a Perl integer, is never dereferenced by the runtime, and `undef`/`0` passes `NULL`. Never use `p` for such pointers - `p` returns copy-until-NUL string semantics, which is wrong and unsafe for non-string data.

Memory primitives

For C APIs that traffic in caller-owned memory (out-parameters, pointer arrays, length-counted buffers) the module provides the pointer toolbox: `alloc`/`free`, `peek`/`peek_cstr`, `poke`, and `pack_ptr`/`unpack_ptr`. Addresses are ordinary Perl integers as with `o`. A pointer array (e.g. `char *argv[]`) is built portably as `join "", map pack_ptr($_), @addrs` poked into an `alloc`ed block, or passed directly as a `p` string when the callee only reads it during the call.

Σύνοψη

```
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
```

Αρθρώματα

- `Peta::FFI::Cairo` - Native reimplementation 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.

Other Functions

`ffi_peek`

Read `$len` raw bytes at address `$ptr` into a Perl byte string.

`ffi_peek_cstr`

Read the NUL-terminated C string at address `$ptr`.

`ffi_poke`

Copy a Perl byte string into memory at address `$ptr`.

`ffi_alloc`

Allocate `$len` zeroed bytes of C memory; returns the address.

`ffi_free`

Release memory obtained from `alloc`.

`ffi_pack_ptr`

Pack an address into a native pointer-sized byte string.

Synopsis

```
**Build a char*[] for PQexecParams-style APIs:**

my $vec = join "", map { Peta::FFI::pack_ptr($_) } @addrs;
```

What you get back

The address as native-endian bytes, sized to the platform pointer width - the in-memory representation a C pointer slot holds.

`ffi_unpack_ptr`

Unpack a native pointer-sized byte string into an address.

Peta::FFI::Cairo

Native reimplementation 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))] = &[ /* ... */ ];

// @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 `size_of::<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 (newSVCairoPattern_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_
(>=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(size_of::<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 `size_of::<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 x 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 (pperl 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 `int`s `{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_attribs, $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`.

Σύνοψη

```
my $lib = Peta::FFI::dlopen("libm.so.6");
my $lib = Peta::FFI::dlopen("/usr/lib/libuuid.so.1");   # absolute␣
↪path
```

Τι επιστρέφεται

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.

Οριακές περιπτώσεις

- **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.

Σύνοψη

```
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
```

Ορίσματα

- `$lib` - handle returned by `dlopen`.
- `$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.

Τι επιστρέφεται

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.

Οριακές περιπτώσεις

- **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`.

Σύνοψη

```
Peta::FFI::dlclose($lib);
```

Τι επιστρέφεται

Always 1. The return value exists so the call composes cleanly in expressions; inspecting it tells you nothing about success.

Οριακές περιπτώσεις

- **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.

Σύνοψη

```
my $libs = Peta::FFI::scan;
print $libs->{"libm.so.6"}, "\n";          # /usr/lib/libm.so.6
my @names = grep { /^libssl/ } keys %$libs;
```

Τι επιστρέφεται

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.

Οριακές περιπτώσεις

- **`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.

ffi_peek

Read `$len` raw bytes at address `$ptr` into a Perl byte string.

Synopsis

```
my $bytes = Peta::FFI::peek($ptr, 16);
```

What you get back

A byte string of exactly `$len` bytes copied from `$ptr`; `undef` when `$ptr` is `0`/`undef`. `$len` of `0` returns the empty string.

Edge cases

- **Fewer than 2 arguments** - croaks.
- **Address or length wrong** - reading memory the process does not own is undefined behaviour; the length must come from the same C API that produced the pointer (e.g. `PQgetlength`).

ffi_peek_cstr

Read the NUL-terminated C string at address `$ptr`.

Synopsis

```
my $str = Peta::FFI::peek_cstr($ptr);
```

What you get back

The bytes at `$ptr` up to (excluding) the first NUL; `undef` when `$ptr` is `0`/`undef`. This is the reader for `o`-typed returns that point at C strings (`PQgetvalue`, `mysql_error`, ...).

ffi_poke

Copy a Perl byte string into memory at address `$ptr`.

Synopsis

```
my $buf = Peta::FFI::alloc(8);
Peta::FFI::poke($buf, pack("l", 42));
```

What you get back

The number of bytes written (the string's byte length).

Edge cases

- **Fewer than 2 arguments or NULL pointer** - croaks; poking through NULL is never meaningful.
- **Buffer too small** - writing past an allocation is undefined behaviour; the destination must be at least as large as the string, which the caller controls via `alloc`.

ffi_alloc

Allocate `$len` zeroed bytes of C memory; returns the address.

Synopsis

```
my $out = Peta::FFI::alloc(8);       # e.g. a sqlite3** out-param
↪cell
Peta::FFI::call($lib, "sqlite3_open", "(po)i", ":memory:", $out);
my $db = Peta::FFI::unpack_ptr(Peta::FFI::peek($out, 8));
Peta::FFI::free($out);
```

What you get back

The address of a zeroed `calloc` block, as an integer for o arguments. The caller owns it: every `alloc` needs a matching `free`. A `$len` of `0` still returns a valid 1-byte allocation.

ffi_free

Release memory obtained from `alloc`.

Synopsis

```
Peta::FFI::free($ptr);
```

What you get back

Always 1.

Edge cases

- **Zero or undef pointer** - silently ignored, like `free(NULL)`.
- **Double free / foreign pointers** - undefined behaviour; only pass addresses obtained from `alloc`, exactly once. Memory a C library allocated is released by that library's own function (`PQclear`, `mysql_free_result`, ...), never by `free`.

ffi_unpack_ptr

Unpack a native pointer-sized byte string into an address.

Synopsis

```
my $db = Peta::FFI::unpack_ptr(Peta::FFI::peek($out, 8));
```

What you get back

The address stored in the string's first pointer-sized bytes, as an integer for o arguments. Croaks when the string is shorter than the platform pointer width.

Peta::XS

Peta::XS - the interpreter's capability layer for pure Perl.

Design: `docs/internal/design/peta-xs.md`; rationale and tier model: `docs/internal/analysis/xs.md`. Unlike every other module in src/native/ this is NOT a CPAN phenotype - there is no upstream .xs to mirror. It is pperl's own versioned, semantic API that lets third-party pure Perl do what internals-XS modules do on perl5.

CONTRACT (v1.0)

Capabilities: `facts`, `magic`. Evolution is additive within a major version; a capability is never silently redefined.

1. Facts and verbs, never layout. Every exposed datum is defined by observable semantics (never flag bits, offsets, addresses or struct names) and survives any internal refactor.
2. Opaque handles. Anything referring to an interpreter entity (a wizard, a magic attachment) is a runtime-owned opaque value - no arithmetic, no raw pointers as IVs.
3. The boundary speaks Perl. Callbacks are coderefs; data crosses as ordinary Perl values. Consumers are pure Perl and cannot be binary-broken by any pperl release. The loudest possible failure is a compile-time version/capability refusal in `import`.

JIT interaction (contract text): a variable carrying Peta::XS magic is never JIT-compiled - the p5jit entry guards decline magical SVs, so compiled-loop exactness holds by construction.

The implementation attaches PERL_MAGIC_ext ("~") magic through the pperl core (`sv.c Perl_sv_magicext`, `mg.c` call discipline), so callback timing is exactly perl5's tied-variable timing. No Rust-side registry exists: all state rides on the magic attachment's mg_obj, the way XS magic modules own state.

Functions

Other Functions

`refcount`

Reference count of the thing `$ref` points at.

`value_slots`

Which value representations the runtime currently considers valid for the scalar `$ref` points at.

`magic`

Inventory of the magic attached to the variable `$ref` points at.

`wizard`

Create a wizard: an opaque bundle of magic callbacks.

`cast`

Attach a wizard to the variable `$ref` points at.

`dispell`

Detach a wizard from the variable `$ref` points at.

`getdata`

Retrieve the private data this wizard's attachment carries.

refcount

Reference count of the thing `$ref` points at.

Synopsis

```
use Peta::XS ':facts';
my $x;
my $n = refcount(\$x);   # 2 - see the observer effect below
```

What you get back

An integer: how many references the runtime currently holds to the referent, at call time. Works on any referent - scalars, arrays, hashes, subs, aggregate elements. Croaks if the argument is not a reference; never dies otherwise.

Observer effect

The reference you pass in contributes one count of its own for the duration of the call, exactly as with `Devel::Refcount::refcount`. It is deliberately NOT subtracted: a lexical with no other references reports 2 (its pad slot plus your temporary reference).

Examples

```
my $x = 42;
say refcount(\$x);        # 2: pad slot + the \$x you just made

my $r = \$x;
say refcount($r);         # 2: pad slot + $r (no extra temporary)
my $r2 = \$x;
say refcount($r);         # 3: pad slot + $r + $r2

my @queue = ($r);
say refcount($r);         # 4: the array element counts too

## leak hunting: a cache entry still pinning an object

my $obj = SomeClass->new;
$cache{key} = $obj;
say refcount(\$obj);      # 3 - the cache holds one; forget it and
delete $cache{key};       # the count drops back to 2
```

See also

- `value_slots` - which representations the value holds.
- `magic` - what is attached to the variable.
- `Scalar::Util::refaddr` / `builtin::refaddr` - identity, not count.

value_slots

Which value representations the runtime currently considers valid for the scalar `$ref` points at.

Synopsis

```
use Peta::XS ':facts';
my @slots = value_slots(\42);      # ('int')
my @slots = value_slots(\3.14);    # ('num')
my @slots = value_slots(\'hi');    # ('str')
my @slots = value_slots(\\1);      # ('ref')
my @slots = value_slots(\undef);   # ()
```

What you get back

In list context, a subset of `qw(int num str ref)` - the representations the runtime currently considers valid for this value. In scalar context, how many of them there are. A scalar can hold several at once: after a numeric string is used in numeric context it carries both its string and its numeric form, and a `Scalar::Util::dualvar` carries `int` and `str` by construction.

Contract

This is an observation of optimization state, not of meaning: the same program may legitimately report different slot sets across runtimes and versions. Branch on it for diagnostics and teaching, never for program semantics.

Examples

```
## watch a scalar accumulate representations

my $port = '8080';
say "@{[ value_slots(\$port) ]}";   # str
my $next = $port + 1;               # numeric use caches the number
say "@{[ value_slots(\$port) ]}";   # int str

## dualvars hold two truths at once ($! is the classic case)

use Scalar::Util qw(dualvar);
my $dv = dualvar(5, 'five');
say "@{[ value_slots(\$dv) ]}";     # int str

## a classroom one-liner: show why "10" == 10 costs a conversion

## only the first time
```

See also

- `refcount` - how many references point here.
- `magic` - what is attached to the variable.

magic

Inventory of the magic attached to the variable `$ref` points at.

Synopsis

```
use Peta::XS ':facts';
tie my %h, 'Some::Tie';
for my $m (magic(\%h)) {
    printf "type=%s get=%d set=%d\n", $m->{type}, $m->{get}, $m->
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
 ↪{set};
}
```

What you get back

One hashref per attachment, in chain order (most recently attached first, mirroring perl's prepend discipline): `type` is perl's public one-character magic type (the `perlguts` vocabulary - "P" tied array/hash, "q" tied scalar, "~" extension magic as used by `cast`), `get`/`set` are booleans saying whether the attachment hooks reads/writes. In scalar context: the number of attachments. An unmagical variable yields an empty list / 0.

Examples

```
my $plain = 1;
say scalar magic(\$plain);          # 0

## is this hash really tied?

tie my %cfg, 'Config::Tied';
my ($m) = magic(\%cfg);
say "tied!" if $m && $m->{type} eq 'P';

## see your own wizards (and their hook shape) from the outside

use Peta::XS ':magic';
my $w = wizard(get => sub { });
my $x = 1;
cast(\$x, $w);
my @mg = magic(\$x);                # ({ type => '~', get => 1, set␣
 ↪=> 1 })

## debugging aid: why is every read of $slow slow?

printf "%s(get=%d,set=%d)\n", @{$_}{qw(type get set)}
    for magic(\$slow);
```

See also

- `cast` / `dispell` - attach and detach "~" magic.
- `refcount`, `value_slots` - the other facts.
- `perldoc perlguts` - the magic type vocabulary.

wizard

Create a wizard: an opaque bundle of magic callbacks.

Synopsis

```
use Peta::XS ':magic';
my $w = wizard(
    get  => sub { my ($ref, $data) = @_; ... },
    set  => sub { my ($ref, $data) = @_; ... },
    free => sub { my ($ref, $data) = @_; ... },
    data => sub { my ($ref, @args) = @_; return { count => 0 } },
);
```

What you get back

An opaque handle. Hold it, pass it to `cast`, `dispell` and `getdata` - nothing else. One wizard can be cast on any number of variables; each attachment gets its own private data.

Contract

All keys are optional but at least one of `get`/`set`/`free` is required; every value must be a code reference (anything else croaks, as does an unknown key). `get` is called before the variable's value is used, `set` after it changed, `free` while it is being destroyed; each receives `(\$var, $data)` and its return value is ignored (no value substitution in v1). `data` is called once per `cast` to build that attachment's private data. A callback that dies propagates exactly like a dying tied-variable FETCH/STORE. A variable carrying wizard magic is never JIT-compiled, so callbacks fire on every single access, compiled neighbors notwithstanding.

Examples

```
## a watchpoint: who is touching $config?

my $watch = wizard(
    get => sub { warn "read at @{[ join ':', caller ]}\n" },
    set => sub { warn "now: ${$_[0]}\n" },
);

## lazy initialization: get may write the variable before the

## pending read uses it (magic is suspended inside callbacks, so

## this does not recurse)

my $lazy = wizard(get => sub { ${$_[0]} //= expensive() });
my $val;
cast(\$val, $lazy);
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
say $val;                  # expensive() runs here, exactly once
say $val;                  # cached now

## per-attachment counters via a data callback

my $counted = wizard(
    data => sub { { reads => 0, writes => 0 } },
    get  => sub { $_[1]{reads}++ },
    set  => sub { $_[1]{writes}++ },
);
```

See also

- `cast` - attach the wizard to a variable.
- `getdata` - read an attachment's private data back.
- `magic` - the facts-side view of an attachment.

cast

Attach a wizard to the variable `$ref` points at.

Synopsis

```
use Peta::XS ':magic';
my $x;
cast(\$x, $w);              # true
cast(\$x, $w);              # false - same wizard twice is a no-op
cast(\$x, $other, 1, 2);    # @args reach $other's data callback
```

What you get back

True when the wizard was attached, false when this wizard was already on the variable (casting twice is a no-op and calls no callbacks). The cast itself fires nothing - the first `get`/`set` comes from the first read/write after it.

Contract

If the wizard has a `data` callback it is called once, here, as `$data_cb->(\$referent, @args)`, and its return value becomes the attachment's private data (see `getdata`). Multiple distinct wizards may be cast on one variable; they fire most recently cast first. Readonly variables can be cast (attaching magic does not write the value - writes still croak before `set` would fire). The attachment lives exactly as long as the variable or until `dispell`ed.

Examples

```
## any scalar works, including aggregate elements

my @row = (1, 2, 3);
cast(\$row[1], $watch);     # watch one array element
my %cfg = (mode => 'dev');
cast(\$cfg{mode}, $watch);  # watch one hash value

## cast args parameterize the attachment via the data callback

my $tagged = wizard(
    data => sub { my ($ref, $tag) = @_; $tag },
    set  => sub { warn "[$_[1]] changed\n" },
);
cast(\my $a, $tagged, 'alpha');
cast(\my $b, $tagged, 'beta');
$a = 1;                     # warns "[alpha] changed"
$b = 2;                     # warns "[beta] changed"
```

See also

- `wizard` - build the handle and its callbacks.
- `dispell` - the inverse operation.
- `getdata` - fetch what the data callback returned.

dispell

Detach a wizard from the variable `$ref` points at.

Synopsis

```
use Peta::XS ':magic';
dispell(\$x, $w);   # true if $w was attached, false otherwise
```

What you get back

True if this wizard's attachment was found and removed, false if it was not attached. Idempotent: a second dispell of the same wizard returns false and changes nothing.

Contract

Removes exactly this wizard's attachment - other wizards on the same variable stay, and so does unrelated magic (a tie, for example). The attachment's private data is released. The `free` callback does NOT run: dispelling is detachment, not destruction; `free` fires only when the variable itself dies.

Examples

```
## scoped watching: attach, observe, detach

cast(\$config, $watch);
run_suspicious_code();
dispell(\$config, $watch);   # $config is an ordinary scalar again

## wizards detach independently

cast(\$x, $w1);
cast(\$x, $w2);
dispell(\$x, $w1);           # $w2 keeps firing
say scalar magic(\$x);       # 1
```

See also

- `cast` - the inverse operation.
- `magic` - check what is (still) attached.

getdata

Retrieve the private data this wizard's attachment carries.

Synopsis

```
use Peta::XS ':magic';
my $data = getdata(\$x, $w);   # whatever $w's data callback
 ↪returned
```

What you get back

The attachment's private data - the value the wizard's `data` callback returned at `cast` time - or undef when this wizard has no attachment on the variable or the wizard has no `data` callback. It is the same value the `get`/`set`/`free` callbacks receive as their second argument: if it is a reference, mutations made inside callbacks are visible here.

Examples

```
## per-variable access statistics, read back after the fact

my $counted = wizard(
    data => sub { { reads => 0, writes => 0 } },
    get  => sub { $_[1]{reads}++ },
    set  => sub { $_[1]{writes}++ },
);
my $x = 1;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
cast(\$x, $counted);
my $v = $x; $v = $x;          # two reads
$x = 2;                       # one write
my $stats = getdata(\$x, $counted);
say "$stats->{reads}/$stats->{writes}";   # 2/1

## distinguishing "no attachment" from "no data"

say defined getdata(\$x, $counted) ? 'attached' : 'not here';
```

See also

- `cast` - where the data callback runs.
- `wizard` - declaring the data callback.

5.2.27 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.

Σύνοψη

```
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: `(tied %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.

5.2.28 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

ListUtil.xs tainted(sv) - true iff the argument carries PERL_MAGIC_taint (set by perl5's taint-source machinery under -T; ${^TAINT} is live in pperl and %ENV/@ARGV/$0/file reads propagate SvTAINTED normally). `int` RETVAL returns via the entersub TARG (the typemap's dXSTARG/PUSHi) - see xs_readonly for why.

readonly

ListUtil.xs readonly(sv) - int RETVAL = SvREADONLY(sv) after get-magic. RETVAL is the raw flag word (SVf_READONLY|SVf_PROTECT), not a normalized bool - perl5 returns e.g. 134283264 and callers can observe it. Returning through the TARG matters beyond

faithfulness: pushing &PL_sv_yes/no makes the caller's lvalue COW-static, so every later assignment to it pays a sv_force_normal/uncow round-trip (measured 0.66x perl5).

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.

5.2.29 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 $!;
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.

Σύνοψη

```
my $packed = inet_aton("127.0.0.1");
my $packed = inet_aton("localhost");
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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()
```

Οριακές περιπτώσεις

- 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, ...)`.

Διαφορές από το upstream

Fully compatible with upstream Socket.

Δείτε επίσης

- `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.

Σύνοψη

```
my $str = inet_ntoa($packed_ipv4);
```

Τι επιστρέφεται

A string like `"192.0.2.1"`, or `undef` if the input is shorter than 4 bytes.

Παραδείγματα

```
print inet_ntoa(INADDR_LOOPBACK);    # 127.0.0.1
```

```
my ($port, $ip) = unpack_sockaddr_in($peer);
printf "%s:%d\n", inet_ntoa($ip), $port;
```

Οριακές περιπτώσεις

- Input shorter than 4 bytes returns `undef`.
- Input longer than 4 bytes: the extra bytes are ignored.

Διαφορές από το upstream

Fully compatible with upstream Socket.

Δείτε επίσης

- `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.

Σύνοψη

```
my $v4 = inet_pton(AF_INET,  "192.0.2.1");
my $v6 = inet_pton(AF_INET6, "2001:db8::1");
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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";
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream Socket.

Δείτε επίσης

- `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.

Σύνοψη

```
my $s = inet_ntop(AF_INET,  $packed4);
my $s = inet_ntop(AF_INET6, $packed6);
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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;
```

Οριακές περιπτώσεις

- Input shorter than the family's address length returns `undef`.
- Any family other than `AF_INET` or `AF_INET6` returns `undef`.

Διαφορές από το upstream

Fully compatible with upstream Socket.

Δείτε επίσης

- `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`.

Σύνοψη

```
my $sin = pack_sockaddr_in($port, $packed_ipv4);
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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 $!;
```

Οριακές περιπτώσεις

- 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_*`.

Διαφορές από το upstream

Fully compatible with upstream Socket.

Δείτε επίσης

- `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.

Σύνοψη

```
my ($port, $packed_ipv4) = unpack_sockaddr_in($sockaddr);
```

Τι επιστρέφεται

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`.

Παραδείγματα

```
my $peer = getpeername($sock);
my ($port, $ip) = unpack_sockaddr_in($peer);
printf "connected from %s:%d\n", inet_ntoa($ip), $port;
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream Socket.

Δείτε επίσης

- `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.

Σύνοψη

```
my $sin6 = pack_sockaddr_in6($port, $packed_ipv6);
my $sin6 = pack_sockaddr_in6($port, $packed_ipv6, $scope_id,
 ↪$flowinfo);
```

Τι επιστρέφεται

A fixed-size byte string matching the platform's `struct sockaddr_in6`, ready for `bind` or `connect`.

Παραδείγματα

```
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);
```

Οριακές περιπτώσεις

- `$scope_id` and `$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.

Διαφορές από το upstream

Fully compatible with upstream Socket.

Δείτε επίσης

- `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.

Σύνοψη

```
my ($port, $addr, $scope_id, $flowinfo) = unpack_sockaddr_in6(
↪$sockaddr);
```

Τι επιστρέφεται

A four-element list. When the input is shorter than a `sockaddr_in6`, all four elements come back as `undef`.

Παραδείγματα

```
my ($port, $addr, $scope, $flow) = unpack_sockaddr_in6(getpeername(
↪$sock));
printf "[%s%%%d]:%d\n", inet_ntop(AF_INET6, $addr), $scope, $port;
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream Socket.

Δείτε επίσης

- `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.

Σύνοψη

```
my $sun = pack_sockaddr_un($path);
```

Τι επιστρέφεται

A fixed-size byte string matching the platform's `struct sockaddr_un`, ready for `bind` or `connect` against an `AF_UNIX` / `AF_LOCAL` socket.

Παραδείγματα

```
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");
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream Socket.

Δείτε επίσης

- `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`.

Σύνοψη

```
my ($path) = unpack_sockaddr_un($sockaddr);
```

Τι επιστρέφεται

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`.

Παραδείγματα

```
my ($path) = unpack_sockaddr_un(getsockname($sock));
print "listening on $path\n";
```

Οριακές περιπτώσεις

- 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`.

Διαφορές από το upstream

Fully compatible with upstream Socket.

Δείτε επίσης

- `pack_sockaddr_un` - reverse direction.

sockaddr_family

Read the address-family byte from any packed sockaddr without committing to one unpacking shape.

Σύνοψη

```
my $family = sockaddr_family($sockaddr);
```

Τι επιστρέφεται

An integer equal to one of `AF_INET`, `AF_INET6`, `AF_UNIX`, etc., or `undef` if the input is shorter than 2 bytes.

Παραδείγματα

```
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);
    ...
}
```

Οριακές περιπτώσεις

- The family byte is read in native byte order - this matches how every `pack_sockaddr_*` writes it.
- Input shorter than 2 bytes returns `undef`.

Διαφορές από το upstream

Fully compatible with upstream Socket.

Δείτε επίσης

- 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.

Σύνοψη

```
my $sin             = sockaddr_in($port, $packed_ipv4);
my ($port, $addr)   = sockaddr_in($sockaddr);
```

Τι επιστρέφεται

The same values as the underlying pack_sockaddr_in or unpack_sockaddr_in, depending on argument count.

Παραδείγματα

```
## Packing

connect($sock, sockaddr_in(80, inet_aton("127.0.0.1"))) or die $!;
```

```
## Unpacking

my ($port, $ip) = sockaddr_in(getpeername($sock));
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream Socket.

Δείτε επίσης

- 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.

Σύνοψη

```
my $sin6                              = sockaddr_in6($port, $addr);
my $sin6                              = sockaddr_in6($port, $addr,
 ↪$scope, $flow);
my ($port, $addr, $scope, $flow)  = sockaddr_in6($sockaddr);
```

Τι επιστρέφεται

The same values as the underlying pack_sockaddr_in6 or unpack_sockaddr_in6, depending on argument count.

Παραδείγματα

```
my $sin6 = sockaddr_in6(443, inet_pton(AF_INET6, "::1"));
```

```
my ($port, $addr) = sockaddr_in6(getpeername($sock));
```

Οριακές περιπτώσεις

- The dispatch is on argument count - a single-element list still takes the unpack branch.
- Prefer the explicit pack/unpack forms in new code.

Διαφορές από το upstream

Fully compatible with upstream Socket.

Δείτε επίσης

- 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.

Σύνοψη

```
my ($err, @res) = getaddrinfo($host, $service);
my ($err, @res) = getaddrinfo($host, $service, { family => AF_INET6
 ↪});
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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 });
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream Socket.

Δείτε επίσης

- `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.

Σύνοψη

```
my ($err, $host, $service) = getnameinfo($sockaddr);
my ($err, $host, $service) = getnameinfo($sockaddr, $flags);
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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);
```

Οριακές περιπτώσεις

- Input shorter than 2 bytes returns the error string `"Invalid sockaddr"` and two `undef`s.
- Internally the packed sockaddr is capped at 128 bytes; this is enough for every standard family.

Διαφορές από το upstream

Fully compatible with upstream Socket.

Δείτε επίσης

- `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.

5.2.30 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.

`nfreeze`

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 `nfreeze`.

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.

Σύνοψη

```
use Storable qw(freeze thaw);
my $bytes = freeze(\%config);
my $copy  = thaw($bytes);
```

Τι επιστρέφεται

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`.

Παραδείγματα

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
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το 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`.

Δείτε επίσης

- `thaw` - the matching reader.
- `nfreeze` - portable (network-order) variant.
- `store` - write directly to a named file.
- `dclone` - freeze/thaw round-trip as a deep-copy shortcut.

nfreeze

Serialise a referenced Perl structure to bytes in network (big-endian) byte order for portable transport.

Σύνοψη

```
use Storable qw(nfreeze thaw);
my $bytes = nfreeze(\@rows);
my $back  = thaw($bytes);     # thaw auto-detects byte order
```

Τι επιστρέφεται

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.

Παραδείγματα

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;
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

- Byte layout is not wire-compatible with upstream Storable's network format. `nfreeze` output round-trips only through pperl's `thaw`. Pinned by `t/81-xs-native/Storable/270-nfreeze-byteorder.t`.

Δείτε επίσης

- `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`.

Σύνοψη

```
use Storable qw(freeze thaw);
my $data = thaw($bytes);
```

Τι επιστρέφεται

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.

Παραδείγματα

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);
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το 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`.

Δείτε επίσης

- `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.

Σύνοψη

```
use Storable qw(dclone);
my $copy = dclone($original);
```

Τι επιστρέφεται

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.

Παραδείγματα

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
```

Οριακές περιπτώσεις

- 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).

Διαφορές από το 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`.

Δείτε επίσης

- `freeze` + `thaw` - the long-form equivalent of `dclone`.
- `nfreeze` - 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.

Σύνοψη

```
use Storable qw(store retrieve);
store(\%data, "cache.bin") or die "write failed";
my $data = retrieve("cache.bin");
```

Τι επιστρέφεται

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.

Παραδείγματα

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";
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το 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`.

Δείτε επίσης

- `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.

Σύνοψη

```
use Storable qw(nstore retrieve);
nstore(\%data, "portable.bin") or die "write failed";
my $data = retrieve("portable.bin");    # works on any host
```

Τι επιστρέφεται

Returns 1 on success. The file holds the same portable byte string `nfreeze` would produce, prefixed with the network-order marker. `retrieve` auto-detects that marker, so the reader side does not need a different function.

Παραδείγματα

Write a configuration cache that any client can read:

```
nstore(\%config, "/srv/shared/config.bin");
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

- Byte layout is not wire-compatible with upstream Storable's network-order format. Files round-trip only through pperl's `retrieve`. Pinned by `t/81-xs-native/Storable/250-nstore.t` and `t/81-xs-native/Storable/270-nfreeze-byteorder.t`.

Δείτε επίσης

- `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.

Σύνοψη

```
use Storable qw(store retrieve);
my $data = retrieve("cache.bin");
```

Τι επιστρέφεται

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.

Παραδείγματα

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");
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το 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`.

Δείτε επίσης

- `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.

Σύνοψη

```
use Storable qw(store_fd fd_retrieve);
open my $fh, ">", "data.bin" or die $!;
store_fd(\%data, $fh);
close $fh;
```

Τι επιστρέφεται

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.

Παραδείγματα

Stream multiple records through one filehandle:

```
open my $fh, ">", "log.bin" or die $!;
store_fd($_, $fh) for @events;
close $fh;

## reader side:

open my $r, "<", "log.bin" or die $!;
while (my $ev = fd_retrieve($r)) { ... }
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το 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`.

Δείτε επίσης

- `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.

Σύνοψη

```
use Storable qw(nstore_fd fd_retrieve);
open my $fh, ">", "portable.bin" or die $!;
nstore_fd(\%data, $fh);
```

Τι επιστρέφεται

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.

Παραδείγματα

Ship a record to a remote host over a pipe:

```
open my $out, ">&", \*STDOUT or die $!;
nstore_fd(\%payload, $out);
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το 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`.

Δείτε επίσης

- `fd_retrieve` - matching reader.
- `store_fd` - faster same-architecture variant.
- `nstore` - write portable bytes to a named file.
- `nfreeze` - return portable bytes instead of writing.

fd_retrieve

Read one frozen structure from an open filehandle and reconstruct it.

Σύνοψη

```
use Storable qw(store_fd fd_retrieve);
open my $fh, "<", "data.bin" or die $!;
my $data = fd_retrieve($fh);
```

Τι επιστρέφεται

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.

Παραδείγματα

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);
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το 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`.

Δείτε επίσης

- `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.

Σύνοψη

```
use Storable qw(file_magic);
my $info = file_magic("data.bin");
die "not Storable" unless $info;
```

Τι επιστρέφεται

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.

Παραδείγματα

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};
}
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το 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`.

Δείτε επίσης

- `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.

Σύνοψη

```
use Storable qw(read_magic);
my $info = read_magic($bytes);
die "not Storable" unless $info;
```

Τι επιστρέφεται

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`.

Παραδείγματα

Validate incoming bytes before committing to a thaw:

```
my $info = read_magic($received);
die "bad input" unless $info;
my $data = thaw($received);
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

- See `file_magic` - the same key subset is returned. Pinned by `t/81-xs-native/Storable/200-file-magic.t`.

Δείτε επίσης

- `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.

Σύνοψη

```
use Storable;
my $d = Storable::stack_depth();
```

Τι επιστρέφεται

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.

Παραδείγματα

Raise the limit for a particularly deep structure:

```
local $Storable::recursion_limit = 4096;
my $bytes = freeze($deep_tree);
```

Οριακές περιπτώσεις

- Zero or negative values of `$Storable::recursion_limit` are treated as the default 512 when serialising.

Διαφορές από το upstream

Fully compatible with upstream. Pinned by `t/81-xs-native/Storable/240-stack-depth.t`.

Δείτε επίσης

- `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.

Σύνοψη

```
use Storable;
my $d = Storable::stack_depth_hash();
```

Τι επιστρέφεται

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.

Οριακές περιπτώσεις

- Independent of the scalar limit returned by `stack_depth`. Raising one does not raise the other.

Διαφορές από το upstream

Fully compatible with upstream. Pinned by `t/81-xs-native/Storable/240-stack-depth.t`.

Δείτε επίσης

- `stack_depth` - scalar recursion limit.
- `freeze` / `thaw` - the consumers of this limit.

5.2.31 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.

5.2.32 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.

Ιστορικό

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.

Σύνοψη

```
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).

Σύνοψη

```
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.

Σύνοψη

```
$term->add_history($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.

Σύνοψη

```
$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`, ...).

Σύνοψη

```
$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.

> **Σημείωση**
>
> 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.

5.2.33 Χρόνος

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.

5.2.34 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.

Σύνοψη

```
my @unknown = attributes::_modify_attrs(\&foo, 'lvalue', 'Bent');

## 'lvalue' consumed internally; 'Bent' returned for package␣
 ↪dispatch.
```

Τι επιστρέφεται

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.

Παραδείγματα

```
## 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')
```

Οριακές περιπτώσεις

- 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`.

Διαφορές από το 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.

Δείτε επίσης

- `_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.

Σύνοψη

```
my @attrs = attributes::_fetch_attrs(\&foo);
```

Τι επιστρέφεται

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.

Παραδείγματα

```
sub s :lvalue :method { $x }
my @a = attributes::_fetch_attrs(\&s);    # ('lvalue', 'method')
```

```
my $h = {};
my @a = attributes::_fetch_attrs($h);     # ()
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `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.

Σύνοψη

```
use attributes 'get';
my @attrs = get(\&foo);
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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')
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το 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.

Δείτε επίσης

- `_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`.

Σύνοψη

```
my $pkg = attributes::_guess_stash(\&Some::Module::foo);   #
 ↪'Some::Module'
```

Τι επιστρέφεται

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`.

Παραδείγματα

```
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
```

Οριακές περιπτώσεις

- 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`.

Διαφορές από το upstream

Fully compatible with upstream.

Δείτε επίσης

- `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.

Σύνοψη

```
use attributes 'reftype';
my $t = reftype(\@a);            # 'ARRAY'
my $t = reftype(bless {}, 'X');  # 'HASH'
```

Τι επιστρέφεται

One of `SCALAR`, `ARRAY`, `HASH`, `CODE`, `GLOB`, `IO`, `FORMAT`, `LVALUE`, or `REGEXP`. Returns `undef` when the argument is not a reference.

Παραδείγματα

```
reftype(\1);                      # 'SCALAR'
reftype([1, 2, 3]);               # 'ARRAY'
reftype(sub { });                 # 'CODE'
reftype(bless [], 'Foo');         # 'ARRAY' - blessing is ignored
reftype(42);                      # undef
```

Οριακές περιπτώσεις

- Not a reference, including undef: returns undef rather than croaking.
- Called with the wrong number of args: croaks with Usage: attributes::reftype($reference).

Διαφορές από το 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.

Δείτε επίσης

- get - uses reftype to build FETCH_${type}_ATTRIBUTES.
- _guess_stash - complementary helper for the «which package?» half of the same lookup.

5.2.35 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.

Σύνοψη

```
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"
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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`.

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

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.

Σύνοψη

```
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"`.

Σύνοψη

```
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_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`.

Σύνοψη

```
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.

Σύνοψη

```
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`.

Σύνοψη

```
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)`.

invalidate_all_method_caches

Invalidate the method cache for every package in the interpreter.

Σύνοψη

```
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`.

Σύνοψη

```
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.

Σύνοψη

```
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.

Σύνοψη

```
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.

Σύνοψη

```
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.

5.2.36 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`.

Σύνοψη

```
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)/;
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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.

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.

Σύνοψη

```
my $handle = re::install();   # pragma plumbing; not for application
↪code
```

Τι επιστρέφεται

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.

Διαφορές από το 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.

Σύνοψη

```
use re 'regmust';
my ($anchored, $floating) = regmust(qr/here .* there/x);

## $anchored = 'here', $floating = 'there'
```

Τι επιστρέφεται

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.

Παραδείγματα

```
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//
```

Οριακές περιπτώσεις

- 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.

Διαφορές από το 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.

Σύνοψη

```
use re 'optimization';
my $info = optimization(qr/^\d+foo/);
print $info->{anchored};            # 'foo'
print $info->{'anchor SBOL'};       # 1
print $info->{minlen};              # minimum match length
```

Τι επιστρέφεται

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`.

Οριακές περιπτώσεις

- Argument is not a `qr//` - returns `undef`.
- Pattern has no extracted substrings - the substring-related keys are `undef` or `0` and `checking` is `"none"`.

Διαφορές από το 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.

5.2.37 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`.

Σύνοψη

```
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.

Τελεστές

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

Ενδοσκόπηση

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.

is_lax

Test whether a string is acceptable to `parse` / `new` as a version.

Σύνοψη

```
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`.

Τι επιστρέφεται

A boolean: 1 if the string parses as any valid version, the empty string otherwise.

Παραδείγματα

```
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");         # ""
```

Διαφορές από το upstream

Fully compatible with upstream version 0.9929.

Δείτε επίσης

- `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.

Σύνοψη

```
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.

Τι επιστρέφεται

A boolean: 1 if the string matches the strict grammar, the empty string otherwise.

Παραδείγματα

```
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
```

Διαφορές από το upstream

Fully compatible with upstream version 0.9929.

Δείτε επίσης

- `is_lax` - the permissive counterpart
- `declare` - the recommended way to produce a strict dotted-integer

ΚΕΦΑΛΑΙΟ 6

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. **pperl 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 Όριο μήκους αναγνωριστικών (`Identifier too long`)

6.1.1 Διατύπωση του upstream

Το `perl5-upstream/pod/perlvar.pod` αναφέρει, στην ενότητα **Syntax of Variable Names**:

> 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 wrong on three counts: the number, the unit, and the suggestion that the limit is soft.

6.1.2 Διόρθωση

The correct statement, as of the **Perl 5.44** stable series, is:

> Identifiers may be up to **1019 bytes** long. Names exceeding this length cause a parse-time error: `Identifier too long`.

Three things to take from this:

1. **1019, not 251.** Perl 5.44 sizes the parser's token buffer from `PERL_IDENTIFIER_LENGTH` (`parser.h`), defined as `256 * MAX_UNICODE_UTF8_BYTES` = 1024 bytes. `S_parse_ident` reserves the sigil byte, the trailing `NUL` and a margin, leaving 1019 bytes of name proper.

   The 251-character figure in the pod describes Perl 5.42 and earlier, where the same buffer was a flat `tokenbuf[256]`. The buffer was widened by commit `8785c114b5` («parser.h Allow up to 256 characters in a token», 2025-09-28) and given its name by `9bc8cdede1` («Add #define for the maximum Perl identifier length», 2025-10-11). Both are in `v5.44.0`. The pod text was not updated and is now stale.

2. **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, and a name is rejected as soon as a character's *end* crosses the bound:

Πλάτος χαρακτήρα	Τελευταίο αποδεκτό	Πρώτο απορριπτέο
1 byte (ASCII)	1019 chars = 1019 B	1020 chars = 1020 B
2 bytes (π.χ. é)	509 chars = 1018 B	510 chars = 1020 B
3 bytes (e.g. ל)	339 chars = 1017 B	340 chars = 1020 B
4 bytes (π.χ. 𐌻 - Γοτθικό)	254 chars = 1016 B	255 chars = 1020 B

   The «characters» in the upstream text is correct only for the ASCII case, where bytes and characters coincide.

3. **Σκληρό σφάλμα, όχι «εσωτερικό όριο».** Η διατύπωση «up to an internal limit» υπαινίσσεται κάτι ήπιο. Δεν είναι - ο αναλυτής του perl5 εγείρει μοιραία εξαίρεση `Identifier too long`. Το μήνυμα είναι το μόνο που βλέπει ο χρήστης· δεν υπάρχει αποκοπή, προειδοποίηση ή εφεδρική συμπεριφορά.

6.1.3 pperl behavior

pperl matches upstream Perl 5.44 exactly at every boundary in the table above, for ASCII and for 2-, 3- and 4-byte UTF-8 identifiers alike. There is no divergence to report.

> **Σημείωση**
>
> An earlier revision of this page claimed pperl diverged here by accepting names between 252 and 1019 bytes. That claim was an artifact of measuring against a Perl **5.42** binary while labelling it 5.44. Both perl 5.44 and pperl accept those names. The comparison baseline, not pperl, was wrong.

6.1.4 Τεστ

Τα τεστ συμμόρφωσης που καλύπτουν αυτό το errata βρίσκονται στα:

- `t/01-parsing/090-identifier-length-ascii.t`
- `t/01-parsing/091-identifier-length-utf8.t`

They probe the boundary at 1019/1020 bytes for ASCII and for each UTF-8 character width, plus a byte-versus-character discriminator (600 two-byte characters: only 600 characters, but 1200 bytes, so rejected) and lengths far past the cap.

ΠΑΡΑΡΤΗΜΑ Α'

Αποποίηση ευθύνης & αναφορές σφαλμάτων

Α'.1 Το PetaPerl δεν είναι το upstream Perl

Το PetaPerl είναι ένα ανεξάρτητο, από-το-μηδέν runtime Perl 5 γραμμένο σε Rust. **Δεν** είναι ένα build του διερμηνευτή perl αναφοράς από το CPAN. Πολλά αρθρώματα που έχετε συνηθίσει από το CPAN - `PDL`, `Scalar::Util`, `Storable`, `POSIX` και δεκάδες άλλα - συνοδεύουν το PetaPerl ως **εγγενείς υλοποιήσεις σε Rust** ενσωματωμένες στον διερμηνευτή. Χωρίς XS, χωρίς απαίτηση μεταγλωττιστή C, χωρίς βήμα εγκατάστασης από το CPAN.

Αυτό έχει μία συνέπεια που αφορά κάθε χρήστη:

Σημαντικό

Κάθε σφάλμα, χαρακτηριστικό που λείπει ή διαφορά συμπεριφοράς που παρατηρείτε όταν χρησιμοποιείτε ένα όνομα αρθρώματος που υπάρχει στο CPAN (`PDL::Ufunc`, `Scalar::Util`, `IO::File`, ...) ενώ εκτελείτε υπό το PetaPerl είναι **δικό μας σφάλμα, όχι του upstream.**

Μην υποβάλλετε αυτά τα ζητήματα στο upstream έργο του CPAN. Οι συντηρητές του upstream είναι εθελοντές που εργάζονται σε έναν κώδικα που δεν είναι αυτός που εκτελείτε.

Α'.2 Αναφορές σφαλμάτων

Το PetaPerl είναι μια **ερευνητική προεπισκόπηση υπό ταχύτατη αλλαγή**. Δεν υπάρχει ακόμη δημόσιος μηχανισμός αναφοράς σφαλμάτων.

Σημείωση

Τι σας ζητάμε να κάνετε σήμερα:

- **Μην υποβάλλετε ζητήματα σε upstream έργα του CPAN** για συμπεριφορά που παρατηρείτε στις εγγενείς υλοποιήσεις του pperl. Οι συντηρητές του upstream εργάζονται σε διαφορετικό κώδικα και δεν μπορούν να διορθώσουν τον δικό μας. Αυτό είναι το ένα μήνυμα που υπάρχει για να μεταδώσει αυτή η σελίδα.
- **Κρατήστε τα ευρήματά σας.** Ένα ελάχιστο σενάριο που αναπαράγει το πρόβλημα, η έξοδος του `pperl -V` και μια σύντομη περιγραφή της αναμενόμενης έναντι της πραγματικής συμπεριφοράς είναι τα τρία πράγματα που θα ζητήσει οποιοδήποτε μελλοντικό σύστημα παρακολούθησης. Αν τα αποθηκεύσετε τώρα, θα έχετε λιγότερη δουλειά αργότερα.
- **Ελέγξτε αν είναι ήδη γνωστό.** Κάθε σελίδα αρθρώματος περιλαμβάνει μια ενότητα *Διαφορές από το upstream* που απαριθμεί σκόπιμες αποκλίσεις. Αν η παρατήρησή σας ταιριάζει με κάποιο σημείο εκεί, δεν είναι σφάλμα - είναι τεκμηριωμένη συμπεριφορά.

Όταν ανοίξει δημόσιο σύστημα παρακολούθησης, αυτή η σελίδα θα ενημερωθεί με τη διεύθυνση υποβολής.

A’.3 Πρόθεση συμβατότητας

PetaPerl aims at ~100 % behavioral compatibility with **Perl 5.44** and with the specific versions of the CPAN modules we ship (see `pperl -V`, `PPERL_MODS`). Where we diverge, we diverge deliberately and document it.

Παλαιότερες μορφές της γλώσσας (π.χ. χαρακτηριστικά του `use v5.18;` που διέφεραν από τις μεταγενέστερες εκδόσεις) δεν αποτελούν στόχο συμβατότητας. Το PetaPerl ακολουθεί το τρέχον upstream Perl, όχι το ιστορικό upstream Perl.

A’.4 Συνεισφορές του upstream

Τα αρθρώματα CPAN των οποίων τα ονόματα φέρουμε αντιπροσωπεύουν χρόνια εργασίας εθελοντών συντηρητών, που αξίζουν αναγνώριση για τον σχεδιασμό της διεπαφής και την αρχική υλοποίηση, ακόμη κι όταν ο κώδικάς μας δεν μοιράζεται καμία γραμμή με τον δικό τους. Τους ευχαριστούμε και δεν θέλουμε τα γραμματοκιβώτιά τους να πλημμυρίζουν με ζητήματα που παράγει ένα runtime το οποίο δεν έγραψαν αυτοί. Αυτός είναι ο σημαντικότερος λόγος ύπαρξης αυτής της σελίδας.

A’.5 Άδεια χρήσης

Δείτε τη σελίδα *Άδεια χρήσης* για τους πλήρεις όρους της PetaMem Research Preview License που διέπει τη χρήση αυτού του Λογισμικού.

ΠΑΡΑΡΤΗΜΑ Β′

Άδεια χρήσης

Β′.1 PetaMem Research Preview License

Β′.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.

Β′.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.

Β′.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 «pperl» 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.

ΠΑΡΑΡΤΗΜΑ Γ′

Γλωσσάρι

Όροι, ακρωνύμια και λεξιλόγιο ειδικό του έργου.

pperl
: Το runtime PetaPerl - μια υλοποίηση της Perl 5 σε Rust, με όνομα εκτελέσιμου `pperl`. Διαβάζει τον ίδιο πηγαίο κώδικα Perl με τον τυπικό διερμηνευτή `perl`. Ένας συμπεριφορικός καθρέφτης της `perl5`: σημασιολογία, οριακές περιπτώσεις, magic. Όπου μια δοκιμή διαφωνεί με το upstream perl5, η pperl είναι αυτή που έχει άδικο. Προσθέτει επιπλέον έναν JIT και αυτόματη παραλληλοποίηση.

SV
: Scalar Value. Ένα βαθμωτό της Perl όπως αναπαρίσταται στο runtime του pperl - το enum βρίσκεται στο `src/runtime/sv.rs` και αντικατοπτρίζει τη διάταξη SV του `perl5` όπου η σημασιολογία το απαιτεί.

JIT
: Μεταγλώττιση Just-In-Time. Το pperl μεταγλωττίζει θερμούς βρόχους σε κώδικα μηχανής μέσω του Cranelift· ενεργοποιείται αυτόματα σε επιλέξιμα σώματα `for`/`while`.

ονοματισμένος μοναδιαίος τελεστής
: Μια ενσωματωμένη συνάρτηση της Perl που δέχεται **ακριβώς ένα όρισμα** σε ένα συγκεκριμένο επίπεδο προτεραιότητας - αυστηρότερο από τους τελεστές σύγκρισης, χαλαρότερο από τους αριθμητικούς και τους shift τελεστές. Σε αντίθεση με έναν **τελεστή λίστας** (που καταβροχθίζει άπληστα την υπόλοιπη έκφραση), ένας ονοματισμένος μοναδιαίος παίρνει το ένα του όρισμα και σταματά εκεί. Παραδείγματα: `defined`, `exists`, `ref`, `scalar`, `length`, `uc`/`lc`, `chr`/`ord`, `int`, `abs`, `sqrt`, `sin`/`cos`, `log`, `exp`, `-e`/`-r`/`-f`/... τα τεστ αρχείων. Δείτε τη γραμμή 10 προτεραιότητας του `perlop`.

τελεστής λίστας
: Μια ενσωματωμένη συνάρτηση της Perl που δέχεται μια **λίστα ορισμάτων** και, στη δεξιά πλευρά του τελεστή, καταναλώνει άπληστα την υπόλοιπη έκφραση. Παραδείγματα: `print`, `sort`, `push`, `unshift`, `join`, `split`. Αντιπαραβάλατε με τον **ονοματισμένο μοναδιαίο τελεστή**.

Αυτό είναι ένα αρχικό σπόρος. Όροι που αναφέρονται με ρόλους `{term}` αλλού στην τεκμηρίωση πρέπει να προστίθενται εδώ όταν εμφανίζονται.

μη-αλφαβιτικά

O

T