Perl

new features

v5.10 to v5.36

SAMPLE

brian d foy

{Perl School}

Preface

The Effective Perler

A long time ago, Joseph Hall published *Effective Perl Programming*, which I consider to be one of the best Perl books ever published. Much later, Joshua McAdams and I updated that book to its second edition around the time that Perl v5.10. The joy of that book was its short, independent chapters. There wasn't a narrative working toward something, and readers could drop into any chapter they cared to read without missing anything. This freed each chapter from the tyranny of explaining every feature as I was accustomed to do in my other books.

After we finished the book, we started a website for it and kept going. Eventually running out of general advice on existing features, I turned my attention to exploring new features. Now, the site is almost exclusively devoted to that.

This books collects some of the articles on new features into one place. I've updated many of them, especially since later versions of Perl adjust and unbug some of the features. All of these articles appear on the website, which you can read for free. Those free articles are slightly less polished than what you read here, but eventually I'll take the editing I've done for this book and port them back to the website.

Changes

- **March 2025** - Updated for v5.38. Perl v5.12 gets a new article about `readdir` inside a `while`.
- **June 2023** - Updated for v5.36. v5.34's item on `isa` is updated for its fix. v5.32 adds an item on `indirect`, which connects to its discussion in v5.36. v5.28 is updated for the switch in prototype and attribute order. Added a note about `refaliasing` not handling closures properly.
- **March 2022** - Updated for v5.34.1, including a note that `try-catch` issue is now fixed. v5.24 notes a fix in hex floating point values, and fixes other v5.22 . v5.18 gets item on the stringification of `%ENV` values. Added a v5.10 item for `$^V` as a version object.
- **June 2021** - Updated for v5.34, including a note that `isa` is now fixed. Note that `\g{}` works for named backreferences in v5.10 chapters.
- **January 2021** - Initial release

About the publisher

The Perl School brand has its roots in a series of low-cost Perl training courses that Dave Cross ran in 2012. By running low-cost training at the weekend, he hoped to encourage more programmers to keep their Perl knowledge up to date. These courses were run regularly for about a year before the idea was put on hold for a while.

Dave always knew that he would want to return to the Perl School brand at some point and late in 2017 he realised what the obvious next step was low-cost Perl books. He had already developed a pipeline for creating e-books from Markdown files so it was a short step to republishing some of his training materials as books.

The first Perl School book, Perl Taster was published at the end of 2017 (just in time for the London Perl Workshop). The second was Selenium and Perl,

and there have been many more since then.

If you are interested in writing a book for the Perl School range, then please get in touch. We are @perl_school on Twitter.

Acknowledgments

Josh McAdams made the original deal with Pearson to update *Effective Perl Programming* to its current second edition, and he's the person who roped me into helping him. He also setup the original companion website, *The Effective Perler*. He's since moved on to other things but without him none of this would have likely happened.

Dave Cross started Perl School as a way to quickly publish eBooks. He did quite a bit of the initial research and testing, and I highly customized that for my current process. His support made it possible for this to be my fourth book through his imprint. If you are interested in publishing your own book, I'm happy to share how I've done it.

And, over the years, I've had numerous readers of the website who offered valuable corrections and insights. I've been able to incorporate that feedback in real time on the website and used it to further shape this book.

Other Books By brian

- *Learning Perl* (O'Reilly Media)
- *Intermediate Perl* (O'Reilly Media)
- *Mastering Perl* (O'Reilly Media)
- *Programming Perl*, 4th Edition (O'Reilly Media)
- *Effective Perl Programming*, 2nd Edition (Addison-Wesley)

- *Mojolicious Web Clients* (Perl School)
- *Learning Perl Exercises* (Perl School)
- *Learning Perl 6* (O'Reilly Media)

Chapter 1

Introduction

I'm one of the co-authors of *Learning Perl*, and infrequently updated book that can't keep up with the new Perl features. This is largely due to the old publishing model where it often takes over a year to get a book onto the shelves. That's not a huge problem because almost everything in a book like that is still relevant.

However, I wanted a way to communicate changes faster, and I think I have that in this book. You can easily catch up with new features no matter which *Learning Perl* you have.

I intend to keep updating this book, whether that means expanding to later releases of Perl or expanding the sections with features I hadn't yet covered. Since this is mostly a LeanPub book, you will get all of the updates for free.

And, since I can update this book, I can release it earlier than I would otherwise like to. When people find problems, it's easy for me to fix the book and re-upload it. Let me know where I messed up by filing an issue or mailing me at *briandfoy@pobox.com*.

New Perl

I started with Perl back during the Perl 4 days, while the language was nearing the pinnacle of its popularity. Perl 5 was a major rewrite that introduced references, object orientation, and many other things. After that, Perl 5 progressed slightly, improved its Unicode support (now best in class), and eventually reached v5.6.

Many people thought v5.6 would be the final version of Perl 5. Development had somewhat stalled and nobody quite knew which direction the language should go. At the Open Source Conference in 2000, Larry Wall announced what he intended to be the next major version of Perl: Perl 6. I won't go into that history here (although I do in *Learning Perl 6*), but that's not quite what happened. That language took on a life of its own and is now known as Raku.

But, the Perl 5 Porters were invigorated to push forward to v5.8. That version of Perl is often used as the standard for compatible programming even today, nearly two decades after its release. It's the minimum version I support in most of my writing. I try to show how to do the same thing in v5.8 even if I'm showing off a new features. I think it may be time to break that habit, but it's still a habit.

It's v5.10 that really shook things up. Jesse Vincent took over the "pumpking" role to manage the development and he started developing processes to improve Perl. Perl committed to monthly development versions leading to a annual release, started a support policy, and came up with a way to add new syntax without breaking old programs.

Many people learned Perl a long time ago, and whatever they learned then is what they know now. It's probably like what I know about **awk**, **sed**, C, and other things I don't use that much.

But, that's why books like this exist. I take the time to play with a feature, find it's edges, push its envelope, and bend it until it breaks. I can then tell you everything that I've learned. Your 15 minutes reading something is my week playing with it. I just saved you a bunch of time. Your path to safe

usage is much more direct.

Perl versions

I usually refer to each version using the "v-string" notation and without "Perl", as I've already done. So, Perl 5.10 is simply v5.10. There's more about in one of the first articles in the "Perl 5.10" part (chapter 2).

Perl versions have two tracks. There's what we call the "devel" versions (those with an odd number in the second position) and the stable releases (with an even number in the second position). That's why this book mostly sticks the the even numbers (v5.10, v5.12, ...). Although all of these features were developed in the develop versions, for this purposes of this book I assign those features to the first stable release in which they appear.

The Perl developers release on a roughly annual cycle, with 12 development releases leading up to that. Anything not ready at that point is moved to the next annual cycle. This has worked out well, giving Perl a predictable schedule. In a few large long term features, such as signatures (chapter 51), take much more effort to land, but that hasn't been a typical case.

Perl's support policy, perlpolicy, commits to supporting the last two stable versions. At the moment, that's v5.30 or v5.32. This doesn't mean they won't support earlier versions, but they haven't committed to that. If you are using the **perl** that comes with your system, you may have a much older version since those systems stabilized their default packages while those versions were current. Or, some systems don't update at all.

In some items, I show how to do the same thing in earlier versions, but that's mostly to justify the existence of the feature. Often these additions to the language address pain points in the practice of programming Perl, so I think you need to understand that pain.

Perl's changes are in perldelta

Each version of **perl**, devel and stable, comes with a perldelta, although the filename is a bit weird.

At the start of a development cycle, the current release manager creates a stub file named *pod/perldelta.pod*. That file is always for the latest version.

For the next version, that *pod/perldelta.pod* is renamed such that it includes its version. For example, when v5.32.1 is released, the *pod/perldelta.pod* becomes *pod/perl5321delta.pod*. I find this a bit awkward because there's a bunch of *pod/perl5*delta.pod* files clumped together, then a lonely island of a file *pod/perldelta.pod*.

Typically, the *pod/perldelta.pod* for a stable release has everything you need to know about that release. Subsequent releases are fixes and adjustments to those features. At the end of most items, I link to the *perldelta* for that version.

Experimental features

Some of the features I highlight are not stable yet—they are stable-ish. Perl has a way to make these experimental features available in stable releases. I'll write more about that throughout the book, but (chapter 2, chapter 41, and chapter 42).

If a feature moves out of the experimental category and becomes a stable feature, I include it in the Perl version where that is true. For example, postfix dereferencing (chapter 59) shows up in v5.20 as an experimental feature, but is stable in v5.24. As such, I include it in the Perl v5.24 section.

Some experimental features are still experimental. The Perl process requires them to go through two stable versions (there are exceptions) before they are graduated out of the experimental category. Some don't make it out that quickly. For example, subroutine signatures (chapter 51) show up in v5.20 but are still experimental for various reasons. Those features are still

interesting and available, so I include them with the features for the Perl version where they first appear.

Going the other way, there are some features that have been removed from Perl. In most cases, I cover those in the Perl version where they disappear. That way, you don't read about them in an earlier version and start using them. For example, v5.10 introduced the lexical topic feature, but v5.24 removed it and that's the section that discusses it.

Everywhere reasonable, I make note of these situations in the introductions to each version.

The perlexperiment documentation has a table that tracks the appearance, promotion, or removal of major experimental features. Likewise, perldeprecation has the list of removed features and the features scheduled to be removed.

Try features in isolation

Before you shoehorn these new features into your production code, try them in small programs so you can understand how they work, what they actually can do for you, and what their limitations are. This way you know that what you observe comes from the isolated feature and not some other thing going on in your code. I write about this more on the website in use Test::More to experiment with new ideas.

Part I

Perl v5.10

Perl v5.10, released in December 2007. Full details are in perl5100delta.

This release started a new era by introducing a way to add new features and new keywords to the language without disturbing legacy code. The details are in perl5100delta, and I've pulled out most of the interesting features for this part.

Perl v5.10 showed up over five years after v5.8, which many people thought would be the last version of Perl before we moved to Perl 6 (which ended up not being the next Perl and became Raku instead). When people talk about Perl being dead, they're probably talking about that time between about 2000 and 2007 where no one knew how or when a new version might appear. It wasn't even apparent that v5.10 would change that single it was a single release.

Since then, Perl has released on a roughly annual schedule, along with monthly builds. Instead of waiting for a punch list of features, whatever isn't finished gets pushed off to the next cycle. Predictability, not feature count, is key. That pattern has long been established.

New syntax

Some of the new features are operators and keywords to make some common things easy. Although I haven't actually measured this, I think the defined-or, `//`, and state are my most-used features. Some people might think it should be say, but I'm too used to print. Here are the main new syntax and keywords:

- `use VERSION` turns on new features - chapter 2
- The `-E` command-line switch - chapter 3
- The state environment variable - chapter 5
- Stacked file tests - chapter 8
- say gives you newlines for free - chapter 4
- The defined-or operator - chapter 6
- The `UNITCHECK` subroutine - chapter 10
- Architecture-dependent pack formats - chapter 9
- Special method resolution orders - chapter 7

New features for regular expressions

Perl v5.10 steals some regex features from other languages and adds more Unicode support. There's more regex goodies to come in future versions too:

- The branch reset operator `(?|…)` - chapter 11
- Ignore part of a substitution's match with `\K` - chapter 16
- Relative back references - chapter 15

- Named captures - chapter 14
- Grammars in regular expressions - chapter 13
- Match the Unicode generic line ending - chapter 12

Future changes

Some of v5.10's features show up in later chapters in this book:

- Perl v5.24 removes the lexical `$_` - (chapter 53)
- state can initialize arrays or hashes starting with v5.28 - chapter 69
- The feature addition concept matures a bit in v5.18 - chapter 38 and chapter 39

Chapter 2

Turns on new features implicitly

You can declare the minimum version of Perl you'll allow your script to use. Merely specifying a Perl version prior to 5.10 does nothing other than check the version you specify against the interpreter version. If the version you specify is equal to or greater than the interpreter version, your program continues. If not, it dies:

```
use 5.008;  # needs perl5.008000 or later
```

This works with require too:

```
require 5.008;  # needs perl5.008000 or later
```

The v-string version works too, and that's the notation I prefer:

```
require v5.8;
```

With v5.10, you get three side effects with `use v5.10`. Starting with that version, use-ing the version also pulls in the new features for that version. This keeps programs designed for earlier versions breaking as newer `perl`s add keywords.

Perl 5.10 introduces the keywords say (chapter 4), state (chapter 5), and `given-when`. You implicitly import these when you specify a minimum version of Perl that is at least v5.10. You should use at least v5.10.1 though because that fixes a few problems with v5.10.0:

```
use v5.10.1;
say 'I can use Switch!';  # imported say()

given ($ARGV[0]) {        # imported given()
    when( defined ) { some_sub() }
    };

sub some_sub {
    state $n = 0;         # imported state()
    say "$n: got a defined argument";
    }
```

If you only want some of the new features, you can unimport the ones that you don't want. Perhaps that list is shorter than what you want to keep. It also denotes what you don't want:

```
use v5.10.1;
no feature qw(say);   # leaves state() and given()

sub say {
    # something that you want to do
    }
```

To insist on v5.10.1 but not use its new features, perhaps because that's the installation with the necessary modules, you can unimport the side effects immediately with the new `feature` pragma:

```
use v5.10.1;   # implicit imports
no feature;    # take it right back again

# your own version of say()
sub say {
    # something that you want to do
    }
```

But, a require doesn't import anything, so you might do that instead:

```
BEGIN { require v5.10.1 };
```

A workaround to restrict `perl` versions

There's a way around all of this, and it's to not restrict the version with use if that's the only thing that you want to do. Instead, you can check the value of the `$]` variable:

```
BEGIN {
    die "Unsupported version"
        unless $] >= 5.010 and $] < 5.011
    }
```

This has the added benefit of restricting the upper acceptable `perl` version too.

From the command line

The `-E` switch (chapter 3) allows you to enable new features from the command line:

```
% perl -E "..."
```

From Effective Perl Programming

- Item 83: Limit your distributions to the right platforms

From the Perl documentation

- The use entry in perlfunc
- `feature`

Chapter 3

Enable features on the command line

Perl v5.10 adds new features and you can enable those through the `feature` pragma or by specifying the minimum version with use (chapter 2). That's not so handy on the command line.

The `-M` switch lets you load a module from the command line, which means it's really doing a use for you:

```
% perl -Mfeature=say -e "..."

% perl -Mv5.10 -e "..."
```

Alternately, you could push that into the text for `-e`, but that's perhaps even more unattractive:

```
% perl -e "use feature qw(say); ..."

% perl -e "use v5.10; ..."
```

Instead of doing one of those, you can use the new `-E` switch:

```
% perl -E "..."
```

This acts just like `-e`, which takes the program from the command line, but `-E` also enables all optional features. It does this in the "main compilation unit", which is the fancy way of saying "in the string you give `-e`". It's not going to reach into other modules to enable features.

If you want to leave off one of the optional features, you're stuck with one of the unattractive options. Fortunately, you're not likely to care about the extra features you don't use.

From the Perl documentation

- perlrun

Chapter 4

Newlines for free

The new say is similar to print but it adds a newline for you. These two output statements are the same:

```
use v5.10;

print "Hello World\n";
say "Hello World";
```

Since you don't need to interpolate anything in the say, you might as well use single quotes:

```
say 'Hello World';
```

Note that unlike print, say specifically adds a newline and not the output record separator—the special variable `$\`.

```
$\ = 'XYZ';
print "Hello World";  # outputs "Hello WorldXYZ"
say "Hello World";    # outputs "Hello World\n"
```

The say is actually a shortcut for this block:

```
{ local $\ = "\n"; print ... }
```

Some tricks

One annoying Perl task is to output a transformed list but add a newline at the end. Since you want the newline in the output but not the processed list, you need to separate it from the list for map. Here's one way to do that:

```
print join( ' ', map { ... } @items ), "\n";
```

Two statements would work too, and as would many other things. It's much more convenient to let say do it for you without the parentheses:

```
say join ' ', map { ... } @items;
```

A similar trick allows you to indent everything but the first item when you join on a newline and tab. There's no newline-tab before the first item:

```
my @cats = qw(Buster Mimi Ginger);
say join "\n\t", 'Cats', @cats;
```

The say adds the newline at the end so it all works out:

```
Cats
    Buster
    Mimi
    Ginger
```

From the Perl documentation

- The perlfunc entry for say

Chapter 5

Lexically-persistent variables

Usually you want to limit variables to the smallest scope you can get away with. If your subroutine and only yoru subroutine needs to know something, you can define that variable inside the subroutine:

```
sub get_some_path {
    my $application_root = '/path/to/my/app';
    ...
    catfile( $application_root, $file );
    }
```

With my Perl has to recreate the data on every call to your subroutine. If that data won't change, you're doing a lot of extra work. Maybe you can't measure it, but it's ugly and useless. Once you know about persistent variables, you probably find this offensive.

Move that `$application_root` outside of the subroutine, but limit its scope by wrapping a block around it and the subroutine. A `BEGIN` block works nicely because the variable is defined and has its value before you define the subroutine:

```
BEGIN {
    my $application_root = '/path/to/my/app';
```

```
    sub get_some_path {
        ...
        catfile( $application_root, $file );
        }
    }
```

At the end of the `BEGIN` block, the named `$application_root` goes out of scope. That's a bit messy, but it works. As an aside, it's much nicer when you share the variable with multiple subroutines (see *Intermediate Perl* or my `File::Find::Closures` demonstration):

```
BEGIN {
    my $application_root = '/path/to/my/app';
    sub set_root {
        $application_root = $_[0];
        }
    sub get_some_path {
        ...
        catfile( $application_root, $file );
        }
    }
```

The state keyword

Perl v5.10 adds a state variable. It's like my in that is declares a lexical variable, but unlike it because it only executes on the first call to the subroutine. The next time you call the subroutine, the variable is still there and has the value you left in it during the previous subroutine call. Now you don't constantly redefine `$application_root`:

```
use v5.10;

sub get_some_path {
    state $application_root = '/path/to/my/app';
    ...
    catfile( $application_root, $file );
```

```
    }
```

The state variable is more useful for maintaining a variable's value between calls to the subroutine. This one increments the value of a letter each time:

```
use v5.10;

sub show_letter {
    state $letter = 'a';
    print "Letter is ", $letter++, "\n";
    }

foreach ( 0 .. 5 ) {
    show_letter();
    }
```

The output shows the progression of `$letter`:

```
Letter is a
Letter is b
Letter is c
Letter is d
Letter is e
Letter is f
```

Initializing aggregates

The state allows you to declare aggregates (arrays and hashes) but doesn't let you initialize them (but this is fixed in v5.28, so wait a minute). This is an error:

```
state @letters = qw(a b c d e f);  # Nope
```

But you can initialize scalar variables, which allows you to use references instead:

```
use v5.10;

foreach( 0 .. 9 ) {
    say next_letter();
    }

sub next_letter {
    state $n = 0;
    state $letters = [ qw(a b c d e f) ];
    $letters->[ $n++ % @$letters ];
    }
```

Starting with v5.28, you can initialize aggregates and don't need the references (chapter 69):

```
use v5.28;

foreach( 0 .. 9 ) {
    say next_letter();
    }

sub next_letter {
    state $n = 0;
    state @letters = qw(a b c d e f);
    $letters[ $n++ % @letters ];
    }
```

From Effective Perl Programming

- Item 49: Create closures to lock in data.

From the Perl documentation

- The perlfunc entry for state
- perlsub

Chapter 6

Defined-or

Prior to v5.10, you had to be a bit careful checking a Perl variable before you set a default value. An uninitialized value and a defined but false value acted the same in the logical `||` short-circuit operator. The Perl idiom to set a default value wouldn't know the difference:

```
$value ||= 'some default';
```

That's the binary-assignment equivalent of an expression where you type out the variable twice:

```
$value = $value || 'some default';
```

However, some of the false values might be perfectly valid and meaningful. If there are no rabbits in the hutch, you don't want to replace an actual value, which just happens to be false, with the default. If the command-line argument is zero, don't replace it:

```
# @ARGV = ( 0 );
my $rabbits = $ARGV[0] || 1; # not what you want
```

The undef value, however, is actually the absence of meaningful value. If

the argument list is empty, replacing that with a default is reasonable:

```
# @ARGV = ();
my $rabbits = $ARGV[0] || 1; # maybe what you want
```

To get around this overlap in valid and undefined values, don't use the short-circuit operator. The conditional operator can work, but you specify the variable name three times:

```
$rabbits = defined $ARGV[0] ? $ARGV[0] : 1;
```

As an aside, I'd probably check the number of elements in `@ARGV`, but then I don't get to some `||` or `//`:

```
my $rabbits = @ARGV ? $ARGV[0] : 1;
```

This is handy when you're processing the command-line options or subroutine arguments:

```
sub set_values {
    my( $arg1, $arg2 ) = @_;

    $arg1 = defined $arg1 ? $arg1 : 'some default';
    $arg2 = defined $arg2 ? $arg2 : 'some default';
    }
```

That's a bit messy, so v5.10 introduces the defined-or operator, `//`. Instead of testing for truth, it tests for defined-ness. If its lefthand value is defined, even if it's false (i.e. `0`, `'0'`, or the empty string), that's what it returns, short-circuiting the righthand side. This is much more compact:

```
sub set_values {
    my( $arg1, $arg2 ) = @_;

    $arg1 //= 'some default';
    $arg2 //= 'some default';
    }
```

From Effective Perl Programming

- Item 2: Enable new Perl features when you need them
- Item 83. Limit your distributions to the right platforms
- This article on the website

From the Perl documentation

- perlop