Modern Perl

4th edition

chromatic

Modern Perl
4th edition

Editor: Michael Swaine
Logo design: Devin Muldoon
Cover design: Allison Randal, chromatic, and Jeffrey Martin

ISBN-10: 1680500880
ISBN-13: 978-1680500882

Published by Onyx Neon Press, http://www.onyxneon.com/. The Onyx Neon logo is a trademark of Onyx Neon, Inc.

Onyx Neon typesets books with free software, especially Ubuntu GNU/Linux, Perl, PseudoPod, and LaTeX. Many thanks to the contributors who make these and other projects possible.

2010 - 2011 Edition October 2010
2011 - 2012 Edition January 2012
2014 - 2015 Edition January 2014
2015 - 2016 Edition October 2014

The new homepage of this book is https://pragprog.com/book/swperl/modern-perl-fourth-edition. Electronic versions of this book are also available from http://onyxneon.com/books/modern_perl/, and the companion website is http://modernperlbooks.com/. Please share with your friends and colleagues.

Thanks for reading!

Contents

Preface

Larry Wall released the first version of Perl in 1987. The language grew from its niche as a tool for system administrators who needed something more powerful than shell scripting and easier to use than C programming into a general-purpose programming language. Perl has a solid history of pragmatism and, in recent years, a disciplined approach to enhancement and backwards compatibility.

Over Perl's long history–Perl 5 has been continually refined over the past twenty years–our understanding of what makes great Perl programs has changed. While you can write productive programs which never take advantage of all the language has to offer, the global Perl community has invented, borrowed, enhanced, and polished ideas and made them available to anyone willing to learn them.

Modern Perl is a mindset. It's an approach to writing great software with the Perl programming language. It's how effective Perl programmers write powerful, maintainable, scalable, concise, and excellent code. It takes advantage of Perl's extensive library of free software (the CPAN) and language features designed to multiply your productivity.

You'll benefit most from this book if you have some experience with Perl or another programming language already. If you're comfortable writing and executing programs (and happy to consult the documentation when it's mentioned), you'll get the most from this book.

Running Modern Perl

The `Modern::Perl` module from the CPAN (The CPAN, pp. 9) allows Perl to warn you of typos and other potential problems. It also enables new features introduced in modern Perl releases. Unless otherwise mentioned, all of the code snippets in this book assume you've started with this basic program skeleton:

```
#!/usr/bin/env perl

use Modern::Perl '2015';
use autodie;
```

If you don't have `Modern::Perl` installed, you could write instead:

```
#!/usr/bin/env perl

use 5.016;      # implies "use strict;"
use warnings;
use autodie;
```

Some examples use testing functions such as `ok()`, `like()`, and `is()` (Testing, pp. 131). The skeleton for these examples is:

```
#!/usr/bin/env perl

use Modern::Perl;
use Test::More;
```

```
# example code here

done_testing();
```

At the time of writing, the current stable major Perl release is Perl 5.22. If you're using an older version of Perl, you may not be able to run all of the examples in this book unmodified. The examples in this book work best with Perl 5.16.0 or newer, though we recommend at least Perl 5.20. While the term "Modern Perl" has traditionally referred to any version of Perl from 5.10.1, the language has improved dramatically over the past several years.

Though Perl comes preinstalled on many operating systems, you may need to install a more modern version. Windows users, download Strawberry Perl from http://www.strawberryperl.com/ or ActivePerl from http://www.activestate.com/activeperl. Users of other operating systems with Perl already installed (and a C compiler and the other development tools), start by installing the CPAN module `App::perlbrew`[1].

`perlbrew` manages multiple Perl installations, so that you can switch between versions for testing and deployment. You can also install CPAN modules in your home directory without affecting the system installation. If you've ever had to beg a system administrator for permission to install software, you'll appreciate this.

Credits

This book would not have been possible without questions, comments, suggestions, advice, wisdom, and encouragement from many, many people. In particular, the author thanks this edition's tech reviewers Andy Lester, Sean Lindsay, and Mohsen Jokar as well as Michael Swaine, editor of this edition. Contributors to this and previous editions include:

John SJ Anderson, Peter Aronoff, Lee Aylward, Alex Balhatchet, Nitesh Bezzala, Ævar Arnfjörð Bjarmason, Matthias Bloch, John Bokma, Géraud CONTINSOUZAS, Vasily Chekalkin, Dmitry Chestnykh, E. Choroba, Tom Christiansen, Anneli Cuss, Paulo Custodio, Steve Dickinson, Kurt Edmiston, David Farrell, Felipe, Shlomi Fish, Jeremiah Foster, Mark Fowler, John Gabriele, Nathan Glenn, Kevin Granade, Andrew Grangaard, Bruce Gray, Ask Bjørn Hansen, Tim Heaney, Graeme Hewson, Robert Hicks, Michael Hicks, Michael Hind, Mark Hindess, Yary Hluchan, Daniel Holz, Mike Huffman, Gary H. Jones II, Curtis Jewell, Mohammed Arafat Kamaal, James E Keenan, Kirk Kimmel, Graham Knop, Yuval Kogman, Jan Krynicky, Michael Lang, Jeff Lavallee, Moritz Lenz, Andy Lester, Jean-Baptiste Mazon, Josh McAdams, Gareth McCaughan, John McNamara, Shawn M Moore, Alex Muntada, Carl Mäsak, Chris Niswander, Nelo Onyiah, Chas. Owens, ww from PerlMonks, Matt Pettis, Jess Robinson, Dave Rolsky, Gabrielle Roth, Grzegorz Rożniecki, Jean-Pierre Rupp, Eduardo Santiago, Andrew Savige, Lorne Schachter, Alex Schroeder, Steve Schulze, Dan Scott, Alex-ander Scott-Johns, Phillip Smith, Christopher E. Stith, Mark A. Stratman, Bryan Summersett, Audrey Tang, Scott Thomson, Ben Tilly, Ruud H. G. van Tol, Sam Vilain, Larry Wall, Lewis Wall, Paul Waring, Colin Wetherbee, Frank Wiegand, Doug Wilson, Sawyer X, David Yingling, Marko Zagozen, Ahmad M. Zawawi, harleypig, hbm, and sunnavy.

Any remaining errors are the fault of the stubborn author.

[1] http://search.cpan.org/perldoc?App::perlbrew

CHAPTER 1

The Perl Philosophy

Perl gets things done–it's flexible, forgiving, and malleable. Capable programmers use it every day for everything from one-liners and one-off automations to multi-year, multi-programmer projects.

Perl is pragmatic. You're in charge. You decide how to solve your problems and Perl will mold itself to do what you mean, with little frustration and no ceremony.

Perl will grow with you. In the next hour, you'll learn enough to write real, useful programs–and you'll understand *how* the language works and *why* it works as it does. Modern Perl takes advantage of this knowledge and the combined experience of the global Perl community to help you write working, maintainable code.

First, you need to know how to learn more.

Perldoc

Perl respects your time; Perl culture values documentation. The language ships with thousands of pages of core documentation. The `perldoc` utility is part of every complete Perl installation. Your OS may provide this as an additional package; install `perl-doc` on Debian or Ubuntu GNU/Linux, for example. `perldoc` can display the core docs as well as the documentation of every Perl module you have installed–whether a core module or one installed from the Comprehensive Perl Archive Network (CPAN).

CPAN Documentation

http://perldoc.perl.org/ hosts recent versions of the Perl documentation. CPAN indexes at http://search.cpan.org/ and http://metacpan.org/ provide documentation for all CPAN modules. Other distributions such as ActivePerl and Strawberry Perl provide local documentation in HTML formats.

Use `perldoc` to read the documentation for a module or part of the core documentation:

```
$ perldoc List::Util
$ perldoc perltoc
$ perldoc Moose::Manual
```

The first example displays the documentation of the `List::Util` module; these docs are in the module itself. The second example is the table of contents of the core docs. This file is purely documentation. The third example requires you to install the `Moose` (Moose, pp. 107) CPAN distribution; it displays the pure-documentation manual. `perldoc` hides these all of these details for you; there's no distinction between reading the documentation for a core library such as `Data::Dumper` or one installed from the CPAN. Perl culture values documentation so much that even external libraries follow the good example of the core language documentation.

The standard documentation template includes a description of the module, sample uses, and a detailed explanation of the module and its interface. While the amount of documentation varies by author, the form of the documentation is remarkably consistent.

How to Read the Documentation

Perl has lots of documentation. Where do you start?
`perldoc perltoc` displays the table of contents of the core documentation, and `perldoc perlfaq` is the table of contents for Frequently Asked Questions about Perl. `perldoc perlop` and `perldoc perlsyn` document Perl's symbolic operators and syntactic constructs. `perldoc perldiag` explains the meanings of Perl's warning messages. `perldoc perlvar` lists all of Perl's symbolic variables.
You don't have to memorize anything in these docs. Skim them for a great overview of the language and come back to them when you have questions.

The `perldoc` utility can do much, much more (see `perldoc perldoc`). Use the `-q` option with a keyword to search the Perl FAQ. For example, `perldoc -q sort` returns three questions: *How do I sort an array by (anything)?*, *How do I sort a hash (optionally by value instead of key)?*, and *How can I always keep my hash sorted?*.

The `-f` option shows the documentation for a builtin Perl function, such as `perldoc -f sort`. If you don't know the name of the function you want, browse the list of available builtins in `perldoc perlfunc`.

The `-v` option looks up a builtin variable. For example, `perldoc -v $PID` explains `$PID`, which is the variable containing the current program's process id. Depending on your shell, you may have to quote the variable appropriately.

The `-l` option shows the *path* to the file containing the documentation. (A module may have a separate *.pod* file in addition to its *.pm* file.)

The `-m` option displays the entire *contents* of the module, code and all, without any special formatting.

Perl uses a documentation format called *POD*, short for *Plain Old Documentation*. `perldoc perlpod` describes how POD works. Other POD tools include `podchecker`, which validates the structure of POD documents, and the `Pod::Webserver` CPAN module, which displays local POD as HTML through a minimal web server.

Expressivity

Before Larry Wall created Perl, he studied linguistics. Unlike other programming languages designed around a mathematical notion, Perl's design emulates how people communicate with people. This gives you the freedom to write programs depending on your current needs. You may write simple, straightforward code or combine many small pieces into larger programs. You may select from multiple design paradigms, and you may eschew or embrace advanced features.

Learning Perl is like learning any spoken language. You'll learn a few words, then string together sentences, and then enjoy simple conversations. Mastery comes from practice of both reading and writing code. You don't have to understand every detail of Perl to be productive, but the principles in this chapter are essential to your growth as a programmer.

Other languages may claim that there should be only one best way to solve any problem. Perl allows *you* to decide what's most readable, most useful, most appealing, or most fun.

Perl hackers call this *TIMTOWTDI*, pronounced “Tim Toady”, or “There's more than one way to do it!”

This expressivity allows master craftworkers to create amazing programs but also allows the unwary to make messes. You'll develop your own sense of good taste with experience. Express yourself, but be mindful of readability and maintainability, especially for those who come after you.

Perl novices often find certain syntactic constructs opaque. These idioms (Idioms, pp. 158) offer great (if subtle) power to experienced programmers, but it's okay to avoid them until you're comfortable with them.

As another design goal, Perl tries to avoid surprising experienced (Perl) programmers. For example, adding two variables (`$first_num + $second_num`) is obviously a numeric operation (Numeric Operators, pp. 64). You've expressed your intent to treat the values of those variables as numbers by using a numeric operator. Perl happily does so. No matter the contents of `$first_num` and `$second_num`, Perl will coerce them to numeric values (Numeric Coercion, pp. 50).

Perl adepts often call this principle *DWIM*, or *do what I mean*. You could just as well call this the *principle of least astonishment*. Given a cursory understanding of Perl (especially context; Context, pp. 3), it should be possible to understand the intent of an unfamiliar Perl expression. You will develop this skill as you learn Perl.

Perl's expressivity allows novices to write useful programs without having to understand the entire language. This is by design! Experienced developers often call the results *baby Perl* as a term of endearment. Everyone begins as a novice. Through practice and learning from more experienced programmers, you will understand and adopt more powerful idioms and techniques. It's okay for you to write simple code that you understand. Keep practicing and you'll become a native speaker.

A novice Perl hacker might triple a list of numbers with:

```
my @tripled;

for (my $i = 0; $i < scalar @numbers; $i++) {
    $tripled[$i] = $numbers[$i] * 3;
}
```

. . . and a Perl adept might write:

```
my @tripled;

for my $num (@numbers) {
    push @tripled, $num * 3;
}
```

. . . while an experienced Perl hacker could write:

```
my @tripled = map { $_ * 3 } @numbers;
```

Every program gets the same result. Each uses Perl in a different way.

As you get more comfortable with Perl, you can let the language do more for you. With experience, you can focus on *what* you want to do rather than *how* to do it. Perl doesn't care if you write baby or expert code. Design and refine your programs for clarity, expressivity, reuse, and maintainability, in part or in whole. Take advantage of this flexibility and pragmatism: it's far better to accomplish your task effectively now than to write a conceptually pure and beautiful program next year.

Context

In spoken languages, the meaning of a word or phrase depends on how you use it; the local *context* of other grammatical constructs helps clarify the intent. For example, the inappropriate pluralization of “Please give me one hamburgers!” sounds wrong (the pluralization of the noun differs from the amount), just as the incorrect gender of “la gato” (the article is feminine, but the noun is masculine) makes native speakers chuckle. Some words do double duty; one sheep is a sheep just as two sheep are also sheep and you program a program.

Perl uses context to express how to treat a piece of data. This governs the *amount* of data as well as the *kind* of data. For example, several Perl operations produce different behaviors when you expect zero, one, or many results. A specific construct in Perl may do something different if you write “Do this, but I don't care about any results” compared to “Do this and give me multiple results.” Other operations allow you to specify whether you expect to work with numeric, textual, or true or false data.

You must keep context in mind when you read Perl code. Every expression is part of a larger context. You may find yourself slapping your forehead after a long debugging session when you discover that your assumptions about context were incorrect. If instead you're aware of context, your code will be more correct–and cleaner, flexible, and more concise.

Void, Scalar, and List Context

Amount context governs *how many* items you expect an operation to produce. Think of subject-verb number agreement in English. Even without knowing the formal description of this principle, you probably understand the error in the sentence “Perl are a fun language.” (In terms of amount context, you could say that the verb “are” expects a plural noun or noun phrase.) In Perl, the number of items you request influences how many you receive.

Suppose the function (Declaring Functions, pp. 67) called `find_chores()` sorts your household todo list in order of priority. The number of chores you expect to read from your list influences what the function produces. If you expect nothing, you're

just pretending to be busy. If you expect one task, you have something to do for the next fifteen minutes. If you have a burst of energy on a free weekend, you could get all of your chores.

Why does context matter? A context-aware function can examine its calling context and decide how much work it must do. When you call a function and never use its return value, you've used *void context*:

```
find_chores();
```

Assigning the function's return value to a single item (Scalars, pp. 37) enforces *scalar context*:

```
my $single_result = find_chores();
```

Assigning the results of calling the function to an array (Arrays, pp. 39) or a list, or using it in a list, evaluates the function in *list context*:

```
my @all_results             = find_chores();
my ($single_element, @rest) = find_chores();

# list of results passed to a function
process_list_of_results( find_chores() );
```

The parentheses in the second line of the previous example group the two variable declarations (Lexical Scope, pp. 77) into a single unit so that assignment assigns to both of the variables. A single-item list is still a list, though. You could also correctly write:

```
my ($single_element)   = find_chores();
```

. . . . in which case the parentheses tell Perl parser that you intend list context for the single variable `$single_element`. This is subtle, but now that you know about it, the difference of amount context between these two statements should be obvious:

```
my $scalar_context = find_chores();
my ($list_context) = find_chores();
```

Lists propagate list context to the expressions they contain. This often confuses novices until they understand it. Both of these calls to `find_chores()` occur in list context:

```
process_list_of_results( find_chores() );

my %results = (
    cheap_operation     => $cheap_results,
    expensive_operation => find_chores(), # OOPS!
);
```

Yes, initializing a hash (Hashes, pp. 43) with a list of values imposes list context on `find_chores`. Use the `scalar` operator to impose scalar context:

```
my %results = (
    cheap_operation     => $cheap_results,
    expensive_operation => scalar find_chores(),
);
```

Again, context can help you determine how much work a function should do. In void context, `find_chores()` may legitimately do nothing. In scalar context, it can find only the most important task. In list context, it must sort and return the entire list.

Numeric, String, and Boolean Context

Perl's other context–*value context*–influences how Perl interprets a piece of data. Perl can figure out if you have a number or a string and convert data between the two types. In exchange for not having to declare explicitly what *type* of data a variable contains or a function produces, Perl's value contexts provide hints about how to treat that data.

Perl will coerce values to specific proper types (Coercion, pp. 49) depending on the operators you use. For example, the `eq` operator tests that two values contain equivalent string values:

```
say "Catastrophic crypto fail!" if $alice eq $bob;
```

You may have had a baffling experience where you *know* that the strings are different, but they still compare the same:

```
my $alice = 'alice';
say "Catastrophic crypto fail!" if $alice == 'Bob';
```

The `eq` operator treats its operands as strings by enforcing *string context* on them, but the `==` operator imposes *numeric context*. In numeric context, both strings evaluate to `0` (Numeric Coercion, pp. 50). Be sure to use the proper operator for your desired value context.

Boolean context occurs when you use a value in a conditional statement. In the previous examples, `if` evaluated the results of the `eq` and `==` operators in boolean context.

In rare circumstances, you may not be able to use the appropriate operator to enforce value context. To force a numeric context, add zero to a variable. To force a string context, concatenate a variable with the empty string. To force a boolean context, double up the negation operator:

```
my $numeric_x =  0 + $x;  # forces numeric context
my $stringy_x = '' . $x;  # forces string  context
my $boolean_x =    !!$x;  # forces boolean context
```

Value contexts are easier to identify than amount contexts. Once you know which operators provide which contexts (Operator Types, pp. 64), you'll rarely make mistakes.

Implicit Ideas

Perl code can seem dense at first, but it's full of linguistic shortcuts. These allow experienced programmers to glance at code and understand its important implications. Context is one shortcut. Another is default variables–the programming equivalent of pronouns.

The Default Scalar Variable

The *default scalar variable* (or *topic variable*), `$_`, is most notable in its *absence*: many of Perl's builtin operations work on the contents of `$_` in the absence of an explicit variable. You can still type `$_` if it makes your code clearer to you, but it's often unnecessary.

Many of Perl's scalar operators (including `chr`, `ord`, `lc`, `length`, `reverse`, and `uc`) work on the default scalar variable if you do not provide an alternative. For example, the `chomp` builtin removes any trailing newline sequence (technically the contents of `$/`; see `perldoc -f chomp`) from its operand:

```
my $uncle = "Bob\n";
chomp $uncle;
say "'$uncle'";
```

`$_` behaves the same way in Perl as the pronoun *it* does in English. Without an explicit variable, `chomp` removes the trailing newline sequence from `$_`. When you write "`chomp;`", Perl will always chomp *it*. These two lines of code are equivalent:

```
chomp $_;
chomp;
```

`say` and `print` also operate on `$_` in the absence of other arguments:

```
print;  # prints $_ to the current filehandle
say;    # prints $_ and a newline to the current filehandle
```

Perl's regular expression facilities (Regular Expressions and Matching, pp. 94) default to `$_` to match, substitute, and transliterate:

```
$_ = 'My name is Paquito';
say if /My name is/;

s/Paquito/Paquita/;

tr/A-Z/a-z/;
say;
```

Perl's looping directives (Looping Directives, pp. 29) default to using `$_` as the iteration variable, whether `for` iterating over a list:

```
say "#$_" for 1 .. 10;

for (1 .. 10) {
    say "#$_";
}
```

...or `while` waiting for an expression to evaluate to false:

```
while (<STDIN>) {
    chomp;
    say scalar reverse;
}
```

...or `map` transforming a list:

```
my @squares = map { $_ * $_ } 1 .. 10;
say for @squares; # note the postfix for
```

...or `grep` filtering a list:

```
say 'Brunch is possible!'
    if grep { /pancake mix/ } @pantry;
```

Just as English gets confusing when you have too many pronouns and antecedents, so does Perl when you mix explicit and implicit uses of `$_`. In general, there's only one `$_`. If you use it in multiple places, one operator's `$_` may override another's. For example, if one function uses `$_` and you call it from another function which uses `$_`, the callee may clobber the caller's value:

```
while (<STDIN>) {
    chomp;

    # BAD EXAMPLE
    my $munged = calculate_value( $_ );
    say "Original: $_";
    say "Munged  : $munged";
}
```

If `calculate_value()` or any other function changed `$_`, that change would persist through that iteration of the loop. Using a named lexical is safer and may be clearer:

```
while (my $line = <STDIN>) {
    ...
}
```

Use `$_` as you would the word "it" in formal writing: sparingly, in small and well-defined scopes.

The ... Operator

The triple-dot (`...`) operator is a placeholder for code you intend to fill in later. Perl will parse it as a complete statement, but will throw an exception that you're trying to run unimplemented code if you try to run it. See `perldoc perlop` for more details.

The Default Array Variables

Perl also provides two implicit array variables. Perl passes arguments to functions (Declaring Functions, pp. 67) in an array named `@_`. Array operations (Arrays, pp. 39) inside functions use this array by default. These two snippets of code are equivalent:

```
sub foo {
    my $arg = shift;
    ...
}

sub foo_explicit_args {
    my $arg = shift @_;
    ...
}
```

Just as `$_` corresponds to the pronoun *it*, `@_` corresponds to the pronouns *they* and *them*. *Unlike* `$_`, each function has a separate copy of `@_`. The builtins `shift` and `pop` operate on `@_`, if provided no explicit operands.

Outside of all functions, the default array variable `@ARGV` contains the command-line arguments provided to the program. Perl's array operations (including `shift` and `pop`) operate on `@ARGV` implicitly outside of functions. You cannot use `@_` when you mean `@ARGV`.

readline

Perl's `<$fh>` operator is the same as the `readline` builtin. `readline $fh` does the same thing as `<$fh>`. A bare `readline` behaves just like `<>`. For historic reasons, `<>` is still more common, but consider using `readline` as a more readable alternative. (What's more readable, `glob '*.html'` to `<*.html>`? The same idea applies.)

`ARGV` has one special case. If you read from the null filehandle `<>`, Perl will treat every element in `@ARGV` as the *name* of a file to open for reading. (If `@ARGV` is empty, Perl will read from standard input; see Input and Output, pp. 138.) This implicit `@ARGV` behavior is useful for writing short programs, such as a command-line filter which reverses its input:

```
while (<>) {
    chomp;
    say scalar reverse;
}
```

The Double Open Operator

Perl 5.22 made this expression a little safer with the `<<>>` operator. If a filename provided contains a special punctuation symbol like `|filename` or `filename|`, Perl would do something special with it. The double-diamond operator avoids this behavior.

Why `scalar`? `say` imposes list context on its operands. `reverse` passes its context on to its operands, treating them as a list in list context and a concatenated string in scalar context. If the behavior of `reverse` sounds confusing, your instincts are correct. Perl arguably should have separated "reverse a string" from "reverse a list".

If you run it with a list of files:

```
$ perl reverse_lines.pl encrypted/*.txt
```

. . . the result will be one long stream of output. Without any arguments, you can provide your own standard input by piping in from another program or typing directly. That's a lot of flexibility in a small program–and you're only getting started.

CHAPTER 2

Perl and Its Community

Perl's greatest accomplishment is the huge amount of reusable libraries it has available. Larry Wall explicitly encouraged the Perl community to create and maintain their own extensions to solve every problem imaginable without fragmenting the language into incompatible pidgins. It worked.

That technical accomplishment was almost as important as the growth of a community around Perl. *People* write libraries. *People* build on the work of other people. *People* make a community worth joining and preserving and expanding.

The Perl community welcomes willing participants at all levels, from novices to the developers of Perl itself. Take advantage of the knowledge and experience and code of countless other programmers, and you'll become a better programmer.

The CPAN

Perl is a pragmatic language. If you have a problem, chances are the global Perl community has already written–and shared–code to solve it.

Modern Perl programming relies on the CPAN (http://www.cpan.org/). The Comprehensive Perl Archive Network is an uploading and mirroring system for redistributable, reusable Perl code. It's one of the largest libraries of code in the world. You can find everything from database access to profiling tools to protocols for almost every network device ever created to sound and graphics libraries and wrappers for shared libraries on your system.

Modern Perl without the CPAN is just another language. Modern Perl with the CPAN is a powerful toolkit for solving problems.

CPAN hosts *distributions*, or collections of reusable Perl code. A single distribution can contain one or more *modules*: self-contained libraries of Perl code. Each distribution occupies its own CPAN namespace and provides unique metadata.

The CPAN is Big, Really Big

The CPAN *adds* hundreds of registered contributors and thousands of indexed modules in hundreds of distributions every month. Those numbers do not take into account updates. In May 2015, search.cpan.org reported 12207 uploaders, 150552 modules, and 31873 distributions (representing growth rates of 10.8%, 16.7%, and 9.6% since the previous edition of this book, respectively).

The CPAN itself is merely a mirroring service. Authors upload distributions to a central service (PAUSE) which replicates them to mirror sites from which CPAN clients download them. All of this relies on common behavior; community standards have evolved to identify the attributes and characteristics of well-formed CPAN distributions. These include:

- the behavior of automated CPAN installers
- metadata to describe what each distribution provides and expects
- machine-readable documentation and licensing

Additional CPAN services provide comprehensive automated testing and reporting across platforms and Perl versions. Every CPAN distribution has its own ticket queue on http://rt.cpan.org/ for reporting bugs and working with authors. CPAN sites also

link to previous distribution versions, module ratings, documentation annotations, and more. All of this is available from both http://search.cpan.org/ and http://metacpan.org/.

Modern Perl installations include a client to connect to, search, download, build, test, and install CPAN distributions; this is *CPAN.pm*. With a recent version (as of this writing, 2.10 is the latest stable release), module installation is reasonably easy. Start the client with:

```
$ cpan
```

To install a distribution within the client:

```
$ cpan
cpan[1]> install Modern::Perl
```

. . . or to install directly from the command line:

```
$ cpan Modern::Perl
```

Eric Wilhelm's tutorial on configuring CPAN.pm[1] includes a great troubleshooting section.

CPAN Requirements

Even though the CPAN client is a core module for the Perl distribution, you will need to install standard development tools such as a `make` utility and possibly a C compiler. Windows users, see Strawberry Perl[a] and Strawberry Perl Professional. Mac OS X users must install XCode. Unix and Unix-like users often have these tools available (though Debian and Ubuntu users should install `build-essential`).

[a] http://strawberryperl.com/

CPAN Management Tools

If your operating system provides its own Perl installation, it may be out of date or depend on specific versions of CPAN distributions. Serious Perl developers often construct virtual walls between the system Perl and their development Perl installations. Several projects help to make this possible.

The `App::cpanminus` CPAN client is fast and simple and needs no configuration. Install it with `cpan App::cpanminus`, or:

```
$ curl -LO http://xrl.us/cpanm
$ less cpanm # review the code before running
$ chmod +x cpanm
$ ./cpanm
```

`App::perlbrew` is a system to manage and to switch between your own installations of multiple versions and configurations of Perl. Installation is as easy as:

```
$ curl -LO http://xrl.us/perlbrew
$ less perlbrew # review the code before running
$ chmod +x perlbrew
$ ./perlbrew install
$ perldoc App::perlbrew
```

[1] http://learnperl.scratchcomputing.com/tutorials/configuration/

The `local::lib` CPAN distribution allows you to install and to manage multiple Perl installations. This is an effective way to maintain CPAN distributions for individual users or applications without affecting the system as a whole. See https://metacpan.org/pod/local::lib and https://metacpan.org/pod/App::local::lib::helper for more details.

All three projects tend to assume a Unix-like environment. Windows users, see the Padre all-in-one download (http://padre.perlide.org/download.html).

Community Sites

Perl's homepage at http://www.perl.org/ links to documentation, source code, tutorials, mailing lists, and several important community projects, such as the Perl.org Online library (https://www.perl.org/books/library.html). If you're new to Perl, the Perl beginners mailing list is a friendly place to ask novice questions and get accurate and helpful answers. See http://learn.perl.org/faq/beginners.html.

The home of Perl development is http://dev.perl.org/, which links to relevant resources for Perl's core development.

The CPAN's (The CPAN, pp. 9) central location is http://www.cpan.org/, though experienced users spend more time on http://search.cpan.org/ and https://metacpan.org/. Get used to browsing here for freely available libraries.

Several community sites offer news and commentary. http://blogs.perl.org/ is a free blog platform open to any Perl community member.

Other sites aggregate the musings of Perl hackers, including http://perlsphere.net/, http://PerlTricks.com/, and http://ironman.enlightenedperl.org/. The latter is part of an initiative from the Enlightened Perl Organization (http://enlightenedperl.org/) to increase the amount and improve the quality of Perl publishing on the web.

Perl Weekly (http://perlweekly.com/) offers a weekly take on news from the Perl world. @perlbuzz (https://twitter.com/perlbuzz) regularly tweets new Perl links.

Development Sites

Best Practical Solutions (http://bestpractical.com/) maintains an installation of RT, its popular request-tracking system, for Perl development. Perl's queue is http://rt.perl.org/. Every CPAN distribution has its own queue on http://rt.cpan.org/.

The Perl 5 Porters (or *p5p*) mailing list is the focal point of the development of Perl. See http://lists.cpan.org/showlist.cgi?name=perl5-porters.

The Perl Foundation (http://www.perlfoundation.org/) exists to support the development and promotion of Perl and its community.

Many Perl hackers use Github[2] to host their projects, including the sources of this book[3]. See especially Gitpan [4], which hosts Git repositories chronicling the complete history of every distribution on the CPAN.

A Local Git Mirror

GitPAN receives infrequent updates. As an alternative to hacking CPAN distributions from GitPAN, consider using Yanick Champoux's wonderful `Git::CPAN::Patch` module to create local Git repositories from CPAN distributions.

Events

The Perl community holds countless conferences, workshops, seminars, and meetings. In particular, the community-run YAPC–Yet Another Perl Conference–is a successful, local, low-cost conference model held on multiple continents. See http://yapc.org/.

[2] http://github.com/

[3] http://github.com/chromatic/modern_perl_book/

[4] http://github.com/gitpan/

Hundreds of local Perl Mongers groups get together frequently for technical talks and social interaction. See http://www.pm.org/.

IRC

When Perl mongers can't meet in person, many collaborate and chat online through the textual chat system known as IRC. The main server for Perl community is `irc://irc.perl.org/`. Be aware that the channel *#perl* is a general purpose channel for discussing whatever its participants want to discuss. Direct questions to *#perl-help* instead. Many of the most popular and useful Perl projects have their own IRC channels, such as *#moose* and *#catalyst*; you can find mention of these channels in project documentation.

CHAPTER 3

The Perl Language

The Perl language is a combination of several individual pieces. Although spoken languages use nuance and tone of voice and intuition to communicate across gaps in knowledge and understanding, computers and source code require precision. You can write effective Perl code without knowing every detail of every language feature, but you must understand how they work together to write Perl code well.

Names

Names (or *identifiers*) are everywhere in Perl programs: you choose them for variables, functions, packages, classes, and even filehandles. Valid Perl names all begin with a letter or an underscore and may optionally include any combination of letters, numbers, and underscores. When the `utf8` pragma (Unicode and Strings, pp. 19) is in effect, you may use any UTF-8 word characters in identifiers. These are valid Perl identifiers:

```
my $name;
my @_private_names;
my %Names_to_Addresses;
sub anAwkwardName3;

# with use utf8; enabled
package Ingy::Döt::Net;
```

These are invalid Perl identifiers:

```
my $invalid name; # space is invalid
my @3;            # cannot start with number
my %~flags;       # symbols invalid in name

package a-lisp-style-name;
```

Names exist primarily for your benefit as a programmer. These rules apply only to literal names which appear in your source code, such as `sub fetch_pie` or `my $waffleiron`.

Only Perl's parser enforces the rules about identifier names. You may also refer to entities with names generated at runtime or provided as input to a program. These *symbolic lookups* provide flexibility at the expense of safety. Invoking functions or methods indirectly or looking up symbols in a namespace lets you bypass Perl's parser. Symbolic lookups can produce confusing code. As Mark Jason Dominus recommends[1], prefer a hash (Hashes, pp. 43) or nested data structure (Nested Data Structures, pp. 58) over variables named, for example, `$recipe1`, `$recipe2`, and so on.

Variable Names and Sigils

Variable names always have a leading *sigil* (a symbol) which indicates the type of the variable's value. *Scalar variables* (Scalars, pp. 37) use the dollar sign (`$`). *Array variables* (Arrays, pp. 39) use the at sign (`@`). *Hash variables* (Hashes, pp. 43) use the percent sign (`%`):

[1] http://perl.plover.com/varvarname.html

```
my $scalar;
my @array;
my %hash;
```

Sigils separate variables into different namespaces. It's possible–though confusing–to declare multiple variables of the same name with different types:

```
my ($bad_name, @bad_name, %bad_name);
```

Perl won't get confused, though humans will.

The sigil of a variable changes depending on its use; this change is called *variant sigils*. As context determines how many items you expect from an operation or what type of data you expect to get, so the sigil governs how you manipulate the data of a variable. For example, use the scalar sigil (`$`) to access a single element of an array or a hash:

```
my $hash_element  = $hash{ $key };
my $array_element = $array[ $index ]

$hash{ $key }     = 'value';
$array[ $index ]  = 'item';
```

The parallel with amount context is important. Using a scalar element of an aggregate as an *lvalue* (the target of an assignment; on the *l*eft side of the = character) imposes scalar context (Context, pp. 3) on the *rvalue* (the value assigned; on the *r*ight side of the = character).

Similarly, accessing multiple elements of a hash or an array–an operation known as *slicing*–uses the at symbol (`@`) and imposes list context–even if the list itself has zero or one elements:

```
my @hash_elements  = @hash{ @keys };
my @array_elements = @array[ @indexes ];

my %hash;
@hash{ @keys }     = @values;
```

Given Perl's variant sigils, the most reliable way to determine the type of a variable–scalar, array, or hash–is to observe the operations performed on it. Arrays support indexed access through square brackets. Hashes support keyed access through curly brackets. Scalars have neither.

Namespaces

Perl allows you to collect similar functions and variables into their own unique named spaces–*namespaces* (Packages, pp. 51). A namespace is a collection of symbols grouped under a globally unique name. Perl allows multi-level namespaces, with names joined by double colons (`::`). `DessertShop::IceCream` refers to a logical collection of related variables and functions, such as `scoop()` and `pour_hot_fudge()`.

Within a namespace, you may use the short name of its members. Outside of the namespace, you must refer to a member by its *fully-qualified name*. Within `DessertShop::IceCream`, `add_sprinkles()` refers to the same function as does `DessertShop::IceCream::add_sprinkles()` outside of the namespace.

Standard identifier rules apply to package names. By convention, the Perl core reserves lowercase package names for core pragmas (Pragmas, pp. 129), such as `strict` and `warnings`. User-defined packages all start with uppercase letters. This is a policy enforced primarily by community guidelines.

All namespaces in Perl are globally visible. When Perl looks up a symbol in `DessertShop::IceCream::Freezer`, it looks in the `main::` symbol table for a symbol representing the `DessertShop::` namespace, in that namespace for the `IceCream::` namespace, and so on. Yet `Freezer::` is visible from outside of the `IceCream::` namespace. The nesting of the former within the latter is only a storage mechanism; it implies nothing about relationships between parent and child or sibling packages.

Only you as a programmer can make *logical* relationships between entities obvious–by choosing good names and organizing them well.

Variables

A *variable* in Perl is a storage location for a value (Values, pp. 16). While a trivial program may manipulate values directly, most programs work with variables. Think of this like algebra: you manipulate symbols to describe formulas. It's easier to explain the Pythagorean theorem in terms of the variables `a`, `b`, and `c` than by intuiting its principle by producing a long list of valid values.

Variable Scopes

Your ability to access a variable within your program depends on the variable's scope (Scope, pp. 77). Most variables in modern Perl programs have a lexical scope (Lexical Scope, pp. 77) governed by the syntax of the program as written. Most lexical scopes are either the contents of blocks delimited by curly braces (`{` and `}`) or entire files. *Files* themselves provide their own lexical scopes, such that a `package` declaration on its own does not create a new scope:

```
package Store::Toy;

my $discount = 0.10;

package Store::Music;

# $discount still visible
say "Our current discount is $discount!";
```

You may also provide a block to the `package` declaration. Because this introduces a new block, it also provides a new lexical scope:

```
package Store::Toy {
    my $discount = 0.10;
}

package Store::Music {
    # $discount not visible
}

package Store::BoardGame;

# $discount still not visible
```

Variable Sigils

The sigil of the variable in a declaration determines the type of the variable: scalar, array, or hash. The sigil used when *accessing* a variable varies depending on what you do to the variable. For example, you declare an array as `@values`. Access the first element–a single value–of the array with `$values[0]`. Access a list of values from the array with `@values[ @indices ]`. The sigil you use determines amount context in an lvalue situation:

```
# imposes lvalue context on some_function()
@values[ @indexes ] = some_function();
```

. . . or gets coerced in an rvalue situation:

```
# list evaluated to final element in scalar context
my $element = @values[ @indices ];
```

Anonymous Variables

Perl variables do not *require* names. Names exist to help you, the programmer, keep track of an `$apple`, `@barrels`, or `%cookie_recipes`. Variables created *without* literal names in your source code are *anonymous*. The only way to access anonymous variables is by reference (References, pp. 53).

Variables, Types, and Coercion

This relationship between variable types, sigils, and context is essential to your understanding of Perl.

A Perl variable represents both a value (a dollar cost, available pizza toppings, the names and numbers of guitar stores) and the container which stores that value. Perl's type system deals with *value types* and *container types*. While a variable's *container type*–scalar, array, or hash–cannot change, Perl is flexible about a variable's value type. You may store a string in a variable in one line, append to that variable a number on the next, and reassign a reference to a function (Function References, pp. 57) on the third, though this is a great way to confuse yourself.

Performing an operation on a variable which imposes a specific value type may cause coercion (Coercion, pp. 49) of the variable's existing value type.

For example, the documented way to determine the number of entries in an array is to evaluate that array in scalar context (Context, pp. 3). Because a scalar variable can only ever contain a scalar, assigning an array (the rvalue) to a scalar (the lvalue) imposes scalar context on the operation, and an array evaluated in scalar context produces the number of elements in the array:

```
my $count = @items;
```

Values

New programmers spend a lot of time thinking about *what* their programs must do. Mature programmers spend their time designing a model for the data their programs must understand.

Variables allow you to manipulate data in the abstract. The values held in variables make programs concrete and useful. These values are your aunt's name and address, the distance between your office and a golf course on the moon, or the sum of the masses of all of the cookies you've eaten in the past year. Within your program, the rules regarding the format of that data are often strict.

Effective programs need effective (simple, fast, efficient, easy) ways to represent their data.

Strings

A *string* is a piece of textual or binary data with no particular formatting or contents. It could be your name, an image read from disk, or the source code of the program itself. A string has meaning in the program only when you give it meaning.

A literal string appears in your program surrounded by a pair of quoting characters. The most common *string delimiters* are single and double quotes:

```
my $name    = 'Donner Odinson, Bringer of Despair';
my $address = "Room 539, Bilskirnir, Valhalla";
```

Characters in a *single-quoted string* are exactly and only ever what they appear to be, with two exceptions. To include a single quote inside a single-quoted string, you must escape it with a leading backslash:

```
my $reminder = 'Don\'t forget to escape '
             . 'the single quote!';
```

To include a backslash at the *end* of a string, escape it with another leading backslash. Otherwise Perl will think you're trying to escape the closing delimiter:

```
my $exception = 'This string ends with a '
              . 'backslash, not a quote: \\';
```

Any other backslash will be part of the string as it appears, unless you have two adjacent backslashes, in which case Perl will believe that you intended to escape the second:

```
use Test::More;

is 'Modern \ Perl', 'Modern \\ Perl',
   'single quotes backslash escaping';

done_testing();
```

Testing Examples

This example uses `Test::More` to prove the assertion that Perl considers these two lines equivalent. See Testing, pp. 131 for details on how that works.

A *double-quoted string* gives you more options, such as encoding otherwise invisible whitespace characters in the string:

```
my $tab       = "\t";
my $newline   = "\n";
my $carriage  = "\r";
my $formfeed  = "\f";
my $backspace = "\b";
```

You may have inferred from this that you can represent the same logical string in multiple ways. You can include a tab within a string by typing the `\t` escape sequence or by hitting the Tab key on your keyboard. Both strings look and behave the same to Perl, even though the representation of the string may differ in the source code.

A string declaration may cross (and include) newlines, so these two declarations are equivalent:

```
my $escaped = "two\nlines";
my $literal = "two
lines";
is $escaped, $literal, 'equivalent \n and newline';
```

...but the escape sequences are easier for humans to read.

Perl strings have variable–not fixed–lengths. Perl will change their sizes for you as you modify and manipulate them. Use the *concatenation* operator `.` to combine multiple strings together:

```
my $kitten = 'Choco' . ' ' . 'Spidermonkey';
```

...though concatenating three literal strings like this is ultimate the same to Perl as writing a single string.

When you *interpolate* the value of a scalar variable or the values of an array within a double-quoted string, the *current* contents of the variable become part of the string as if you'd concatenated them:

```
my $factoid = "$name lives at $address!";

# equivalent to
my $factoid = $name . ' lives at ' . $address . '!';
```

Include a literal double-quote inside a double-quoted string by *escaping* it with a leading backslash:

```
my $quote = "\"Ouch,\", he cried. \"That hurt!\"";
```

Repeated backslashing sometimes becomes unwieldy. A *quoting operator* allows you to choose an alternate string delimiter. The q operator indicates single quoting (no interpolation), while the qq operator provides double quoting behavior (interpolation). The character immediately following the operator determines the characters used as delimiters. If the character is the opening character of a balanced pair–such as opening and closing braces–the closing character will be the final delimiter. Otherwise, the character itself will be both the starting and ending delimiter.

```
my $quote     = qq{"Ouch", he said. "That hurt!"};
my $reminder  =  q^Don't escape the single quote!^;
my $complaint =  q{It's too early to be awake.};
```

Use the *heredoc* syntax to assign multiple lines to a string:

```
my $blurb =<<'END_BLURB';

He looked up. "Change is the constant on which they all
can agree.  We instead, born out of time, remain perfect
and perfectly self-aware. We only suffer change as we
pursue it. It is against our nature. We rebel against
that change. Shall we consider them greater for it?"
END_BLURB
```

This syntax has three parts. The double angle-brackets introduce the heredoc. The quotes determine whether the heredoc follows single- or double-quoted behavior; double-quoted behavior is the default. END_BLURB is an arbitrary identifier, chosen by the programmer, used as the ending delimiter.

Regardless of the indentation of the heredoc declaration itself, the ending delimiter must *start* at the beginning of the line:

```
sub some_function {
    my $ingredients =<<'END_INGREDIENTS';
    Two eggs
    One cup flour
    Two ounces butter
    One-quarter teaspoon salt
    One cup milk
    One drop vanilla
    Season to taste
END_INGREDIENTS
}
```

Heredocs and Indentation

If the identifier *begins* with whitespace, that same whitespace must be present before the ending delimiter–that is, <<' END_HEREDOC'>> needs a leading space before END_HEREDOC. Yet if you indent the identifier, Perl will *not* remove equivalent whitespace from the start of each line of the heredoc. Keep that design wart in mind; it'll eventually surprise you.

Using a string in a non-string context will induce coercion (Coercion, pp. 49).

Unicode and Strings

Unicode is a system used to represent the characters of the world's written languages. Most English text uses a character set of only 127 characters (which requires seven bits of storage and fits nicely into eight-bit bytes), but it's naïve to believe that you won't someday need an umlaut.

Perl strings can represent either of two separate but related data types:

Sequences of Unicode characters

: Each character has a *codepoint*, a unique number which identifies it in the Unicode character set.

Sequences of octets

: Binary data in a sequence of *octets*–8 bit numbers, each of which can represent a number between 0 and 255.

Words Matter

Why *octet* and not *byte*? An octet is unambiguously eight bits. A byte can be fewer or more bits, depending on esoteric hardware. Assuming that one character fits in one byte will cause you no end of Unicode grief. Separate the idea of memory storage from character representation. Forget that you ever heard of bytes.

Unicode strings and binary strings look superficially similar. Each has a `length()`. Each supports standard string operations such as concatenation, splicing, and regular expression processing (Regular Expressions and Matching, pp. 94). Any string which is not purely binary data is textual data, and thus should be a sequence of Unicode characters.

However, because of how your operating system represents data on disk or from users or over the network–as sequences of octets–Perl can't know if the data you read is an image file or a text document or anything else. By default, Perl treats all incoming data as sequences of octets. It's up to you to give that data meaning.

Character Encodings

A Unicode string is a sequence of octets which represents a sequence of characters. A *Unicode encoding* maps octet sequences to characters. Some encodings, such as UTF-8, can encode all of the characters in the Unicode character set. Other encodings represent only a subset of Unicode characters. For example, ASCII encodes plain English text (no accented characters allowed), while Latin-1 can represent text in most languages which use the Latin alphabet (umlauts, grave and circumflex accents, et cetera).

An Evolving Standard

Perl 5.16 supports the Unicode 6.1 standard, 5.18 the 6.2 standard, 5.20 the 6.3 standard, and 5.22 the 7.0 standard. See http://unicode.org/versions/.

To avoid most Unicode problems, always decode to and from the appropriate encoding at the inputs and outputs of your program. Read that sentence again. Memorize it. You'll be glad of it later.

Unicode in Your Filehandles

When you tell Perl that a specific filehandle (Files, pp. 138) should interpret data via specific Unicode encoding, Perl will use an *IO layer* to convert between octets and characters. The *mode* operand of the `open` builtin allows you to request an IO layer by name. For example, the `:utf8` layer decodes UTF-8 data:

```
open my $fh, '<:utf8', $textfile;

my $unicode_string = <$fh>;
```

Use `binmode` to apply an IO layer to an existing filehandle:

```
binmode $fh, ':utf8';
my $unicode_string = <$fh>;

binmode STDOUT, ':utf8';
say $unicode_string;
```

If you want to write Unicode to files, you must specify the desired encoding. Otherwise, Perl will warn you when you print Unicode characters that don't look like octets; this is what `Wide character in %s` means.

Enable UTF-8 Everywhere

Use the `utf8::all` module to add the UTF-8 IO layer to all filehandles throughout your program. The module also enables all sorts of other Unicode features. It's very handy, but it's a blunt instrument and no substitute for understanding what your program needs to do.

Unicode in Your Data

The core module `Encode`'s `decode()` function converts a sequence of octets to Perl's internal Unicode representation. The corresponding `encode()` function converts from Perl's internal encoding to the desired encoding:

```
my $from_utf8 = decode('utf8', $data);
my $to_latin1 = encode('iso-8859-1', $string);
```

To handle Unicode properly, you must always *decode* incoming data via a known encoding and *encode* outgoing data to a known encoding. Again, you must know what kind of data you expect to consume and to produce. Being specific will help you avoid all kinds of trouble.

Unicode in Your Programs

The easiest way to use Unicode characters in your source code us with the `utf8` pragma (Pragmas, pp. 129), which tells the Perl parser to decode the rest of the file as UTF-8 characters. This allows you to use Unicode characters in strings and identifiers:

```
use utf8;

sub £_to_¥ { ... }

my $yen = £_to_¥('1000£');
```

To *write* this code, your text editor must understand UTF-8 and you must save the file with the appropriate encoding. Again, any two programs which communicate with Unicode data must agree on the encoding of that data.

Within double-quoted strings, you may also use a Unicode escape sequence to represent character encodings. The syntax `\x{}` represents a single character; place the hex form of the character's Unicode number[2] within the curly brackets:

```
my $escaped_thorn = "\x{00FE}";
```

Some Unicode characters have names, which make them easier for other programmers to read. Use the `charnames` pragma to enable named characters via the `\N{}` escape syntax:

[2] http://unicode.org/charts/

```
use charnames ':full';
use Test::More tests => 1;

my $escaped_thorn = "\x{00FE}";
my $named_thorn   = "\N{LATIN SMALL LETTER THORN}";

is $escaped_thorn, $named_thorn,
    'Thorn equivalence check';
```

You may use the `\x{}` and `\N{}` forms within regular expressions as well as anywhere else you may legitimately use a string or a character.

Implicit Conversion

Most Unicode problems in Perl arise from the fact that a string could be *either* a sequence of octets *or* a sequence of characters. Perl allows you to combine these types through the use of implicit conversions. When these conversions are wrong, they're rarely *obviously* wrong but they're also often *spectacularly* wrong in ways that are difficult to debug.

When Perl concatenates a sequence of octets with a sequence of Unicode characters, it implicitly decodes the octet sequence using the Latin-1 encoding. The resulting string will contain Unicode characters. When you print Unicode characters, Perl will encode the string using UTF-8, because Latin-1 cannot represent the entire set of Unicode characters–because Latin-1 is a subset of UTF-8.

The asymmetry between encodings and octets can lead to Unicode strings encoded as UTF-8 for output and decoded as Latin-1 from input. Worse yet, when the text contains only English characters with no accents, the bug stays hidden, because both encodings use the same representation for every character.

You don't have to understand all of this right now; just know that this behavior happens and that it's not what you want.

```
my $hello    = "Hello, ";
my $greeting = $hello . $name;
```

If `$name` contains *Alice*, you will never notice any problem: because the Latin-1 representation is the same as the UTF-8 representation. If `$name` contains *José*, `$name` can contain several possible values:

- `$name` contains four Unicode characters.
- `$name` contains four Latin-1 octets representing four Unicode characters.
- `$name` contains *five* UTF-8 octets representing four Unicode characters.

The string literal has several possible scenarios:

- It is an ASCII string literal and contains octets: `my $hello = "Hello, ";`
- It is a Latin-1 string literal with no explicit encoding and contains octets: `my $hello = "¡Hola, ";`
- It is a non-ASCII string literal (the `utf8` or `encoding` pragma is in effect) and contains Unicode characters: `my $hello = "Kuirabá, ";`

If both `$hello` and `$name` are Unicode strings, the concatenation will produce another Unicode string.

If both strings are octet sequences, Perl will concatenate them into a new octet sequence. If both values are octets of the same encoding–both Latin-1, for example, the concatenation will work correctly. If the octets do not share an encoding–for example, a concatenation appending UTF-8 data to Latin-1 data–then the resulting sequence of octets makes sense in *neither* encoding. This could happen if the user entered a name as UTF-8 data and the greeting were a Latin-1 string literal, but the program decoded neither.

If only one of the values is a Unicode string, Perl will decode the other as Latin-1 data. If this is not the correct encoding, the resulting Unicode characters will be wrong. For example, if the user input were UTF-8 data and the string literal were a Unicode string, the name would be incorrectly decoded into five Unicode characters to form *JosÃ©* (*sic*) instead of *José* because the UTF-8 data means something else when decoded as Latin-1 data.

Again, you don't have to follow all of the details here if you remember this: always decode on input and encode on output.

See `perldoc perluniintro` for a far more detailed explanation of Unicode, encodings, and how to manage incoming and outgoing data in a Unicode world. For *far* more detail about managing Unicode effectively throughout your programs, see Tom Christiansen's answer to "Why does Modern Perl avoid UTF-8 by default?"[3] and his "Perl Unicode Cookbook" series[4].

Unicode Strings

If you work with Unicode in Perl, use at least Perl 5.18 (and ideally the latest version). See also the `feature` pragma for information on the `unicode_strings` feature.

Numbers

Perl supports numbers as both integers and floating-point values. You may represent them with scientific notation as well as in binary, octal, and hexadecimal forms:

```
my $integer   = 42;
my $float     = 0.007;
my $sci_float = 1.02e14;
my $binary    = 0b101010;
my $octal     = 052;
my $hex       = 0x20;

# only in Perl 5.22
my $hex_float = 0x1.0p-3;
```

The numeric prefixes `0b`, `0`, and `0x` specify binary, octal, and hex notation respectively. Be aware that a leading zero on an integer *always* indicates octal mode.

When 1.99 + 1.99 is 4

Even though you can write floating-point values explicitly with perfect accuracy, Perl–like most programming languages–represents them internally in a binary format. This representation is sometimes imprecise in specific ways; consult `perldoc perlnumber` for more details. Perl 5.22 allows you to use a hexadecimal representation of floating point values, so as to keep maximum precision. See "Scalar value constructors" in `perldoc perldata` for more information.

You may *not* use commas to separate thousands in numeric literals, as the parser will interpret them as the comma operator. Instead, use underscores. These three examples are equivalent, though the second might be the most readable:

```
my $billion = 1000000000;
my $billion = 1_000_000_000;
my $billion = 10_0_00_00_0_0_0;
```

[3] http://stackoverflow.com/questions/6162484/why-does-modern-perl-avoid-utf-8-by-default/6163129#6163129

[4] http://www.perl.com/pub/2012/04/perlunicook-standard-preamble.html

Because of coercion (Coercion, pp. 49), Perl programmers rarely have to worry about converting incoming data to numbers. Perl will treat anything which looks like a number *as* a number when evaluated in a numeric context. In the rare circumstances where *you* need to know if something looks like a number without evaluating it in a numeric context, use the `looks_like_number` function from the core module `Scalar::Util`. This function returns a true value if Perl will consider the given argument numeric.

The `Regexp::Common` module from the CPAN provides several well-tested regular expressions to identify more specific *types* of numeric values such as whole numbers, integers, and floating-point values.

Numeric Size Limits

What's the maximum size of a value you can represent in Perl? It depends; you're probably using a 64-bit build, so the largest integer is `(2**31) - 1` and the smallest is `-(2**31)`–though see `perldoc perlnumber` for more thorough details. Use `Math::BigInt` and `Math::BigFloat` to handle with larger or smaller or more precise numbers.

Undef

Perl's `undef` value represents an unassigned, undefined, and unknown value. Declared but undefined scalar variables contain `undef`:

```
my $name = undef;    # unnecessary assignment
my $rank;            # also contains undef
```

`undef` evaluates to false in boolean a context. Evaluating `undef` in a string context–such as interpolating it into a string:

```
my $undefined;
my $defined = $undefined . '... and so forth';
```

...produces an `uninitialized value` warning:

```
Use of uninitialized value $undefined in
concatenation (.) or string...
```

The `defined` builtin returns a true value if its operand evaluates to a defined value (anything other than `undef`):

```
my $status = 'suffering from a cold';

say defined $status;  # 1, which is a true value
say defined undef;    # empty string; a false value
```

The Empty List

When used on the right-hand side of an assignment, the `()` construct represents an empty list. In scalar context, this evaluates to `undef`. In list context, it is an empty list. When used on the left-hand side of an assignment, the `()` construct imposes list context. Hence this idiom (Idioms, pp. 158) to count the number of elements returned from an expression in list context without using a temporary variable:

```
my $count = () = get_clown_hats();
```

Because of the right associativity (Associativity, pp. 63) of the assignment operator, Perl first evaluates the second assignment by calling `get_clown_hats()` in list context. This produces a list.

Assignment to the empty list throws away all of the values of the list, but that assignment takes place in scalar context, which evaluates to the number of items on the right hand side of the assignment. As a result, `$count` contains the number of elements in the list returned from `get_clown_hats()`.

This idiom often confuses new programmers, but with practice, you'll understand how Perl's fundamental design features fit together.

Lists

A list is a comma-separated group of one or more expressions. Lists may occur verbatim in source code as values:

```
my @first_fibs = (1, 1, 2, 3, 5, 8, 13, 21);
```

. . . as targets of assignments:

```
my ($package, $filename, $line) = caller();
```

. . . or as lists of expressions:

```
say name(), ' => ', age();
```

Parentheses do not *create* lists. The comma operator creates lists. The parentheses in these examples merely group expressions to change their *precedence* (Precedence, pp. 63).

As an example of lists without parens, use the range operator to create lists of literals in a compact form:

```
my @chars = 'a' .. 'z';
my @count = 13 .. 27;
```

Use the `qw()` operator to split a literal string on whitespace to produce a list of strings. As this is a quoting operator, you may choose any delimiters you like:

```
my @stooges = qw! Larry Curly Moe Shemp Joey Kenny !;
```

No Comment Please

Perl will emit a warning if a `qw()` contains a comma or the comment character (`#`), because not only are such characters rare in a `qw()`, their presence is often a mistake.

Lists can (and often do) occur as the results of expressions, but these lists do not appear literally in source code.

Lists and arrays are not interchangeable in Perl. Lists are values. Arrays are containers. You may store a list in an array and you may coerce an array to a list, but they are separate entities. For example, indexing into a list always occurs in list context. Indexing into an array can occur in scalar context (for a single element) or list context (for a slice):

```
# don't worry about the details right now
sub context
{
    my $context = wantarray();

    say defined $context
```

```
          ? $context
              ? 'list'
              : 'scalar'
          : 'void';
    return 0;
}

my @list_slice  = (1, 2, 3)[context()];
my @array_slice = @list_slice[context()];
my $array_index = $array_slice[context()];

say context(); # list context
context();     # void context
```

Control Flow

Perl's basic *control flow* is straightforward. Program execution starts at the beginning (the first line of the file executed) and continues to the end:

```
say 'At start';
say 'In middle';
say 'At end';
```

Perl's *control flow directives* change the order of what happens next in the program.

Branching Directives

The `if` directive performs the associated action only when its conditional expression evaluates to a *true* value:

```
say 'Hello, Bob!' if $name eq 'Bob';
```

This postfix form is useful for simple expressions. Its block form groups multiple expressions into a unit which evaluates to a single boolean value:

```
if ($name eq 'Bob') {
    say 'Hello, Bob!';
    found_bob();
}
```

The conditional expression may consist of multiple subexpressions which will be coerced to a boolean value:

```
if ($name eq 'Bob' && not greeted_bob()) {
    say 'Hello, Bob!';
    found_bob();
}
```

The block form requires parentheses around its condition, but the postfix form does not. In the postfix form, adding parentheses can clarify the intent of the code at the expense of visual cleanliness:

```
greet_bob() if ($name eq 'Bob' && not greeted_bob());
```

The `unless` directive is the negated form of `if`. Perl will perform the action when the conditional expression evaluates to a *false* value:

```
say "You're not Bob!" unless $name eq 'Bob';
```

Like `if`, `unless` also has a block form, though many programmers avoid it due to its potential for confusion:

```
unless (is_leap_year() and is_full_moon()) {
    frolic();
    gambol();
}
```

`unless` works very well for postfix conditionals, especially parameter validation in functions (Postfix Parameter Validation, pp. 162):

```
sub frolic {
    # do nothing without parameters
    return unless @_;

    for my $chant (@_) { ... }
}
```

The block forms of `if` and `unless` both support the `else` directive, which provides a block to execute when the conditional expression does not evaluate to the appropriate value:

```
if ($name eq 'Bob') {
    say 'Hi, Bob!';
    greet_user();
}
else {
    say "I don't know you.";
    shun_user();
}
```

`else` blocks allow you to rewrite `if` and `unless` conditionals in terms of each other:

```
unless ($name eq 'Bob') {
    say "I don't know you.";
    shun_user();
}
else {
    say 'Hi, Bob!';
    greet_user();
}
```

However, the implied double negative of using `unless` with an `else` block can be confusing. This example may be the only place you ever see it.

Just as Perl provides both `if` and `unless` to allow you to phrase your conditionals in the most readable way, Perl has both positive and negative conditional operators:

```
if ($name ne 'Bob') {
    say "I don't know you.";
    shun_user();
}
else {
    say 'Hi, Bob!';
    greet_user();
}
```

. . . though the double negative implied by the presence of the `else` block may be difficult to read.

Use one or more `elsif` directives to check multiple and mutually exclusive conditions:

```
if ($name eq 'Robert') {
    say 'Hi, Bob!';
    greet_user();
}
elsif ($name eq 'James') {
    say 'Hi, Jim!';
    greet_user();
}
elsif ($name eq 'Armando') {
    say 'Hi, Mando!';
    greet_user();
}
else {
    say "You're not my uncle.";
    shun_user();
}
```

An `unless` chain may also use an `elsif` block, but good luck deciphering that.

Perl supports neither `elseunless` nor `else if`. Larry prefers `elsif` for aesthetic reasons, as well the prior art of the Ada programming language:

```
if ($name eq 'Rick') {
    say 'Hi, cousin!';
}

# warning; syntax error
else if ($name eq 'Kristen') {
    say 'Hi, cousin-in-law!';
}
```

The Ternary Conditional Operator

The *ternary conditional* operator evaluates a conditional expression and evaluates to one of two alternatives:

```
my $time_suffix = after_noon($time)
                ? 'afternoon'
                : 'morning';

# equivalent to
my $time_suffix;

if (after_noon(time)) {
    $time_suffix = 'afternoon';
}
else {
    $time_suffix = 'morning';
}
```

The conditional expression precedes the question mark character (`?`). The colon character (`:`) separates the alternatives. The alternatives are expressions of arbitrary complexity–including other ternary conditional expressions, though consider clarity over concision.

Lvalues and the Ternary Conditional

An interesting, though obscure, idiom uses the ternary conditional to select between alternative *variables*, not only values:

```
push @{ rand() > 0.5 ? \@red_team : \@blue_team },
    Player->new;
```

Short Circuiting

Perl exhibits *short-circuiting* behavior when it encounters complex conditional expressions. When Perl can determine that a complex expression would succeed or fail as a whole without evaluating every subexpression, it will not evaluate subsequent subexpressions. This is most obvious with an example:

```
say 'Both true!' if ok 1, 'subexpression one'
                 && ok 1, 'subexpression two';

done_testing();
```

The return value of `ok()` (Testing, pp. 131) is the boolean value produced by the first argument, so the example prints:

```
ok 1 - subexpression one
ok 2 - subexpression two
Both true!
```

When the first subexpression–the first call to `ok`–evaluates to a true value, Perl must evaluate the second subexpression. If the first subexpression had evaluated to a false value, there would be no need to check subsequent subexpressions, as the entire expression could not succeed:

```
say 'Both true!' if ok 0, 'subexpression one'
                 && ok 1, 'subexpression two';
```

This example prints:

```
not ok 1 - subexpression one
```

Even though the second subexpression would obviously succeed, Perl never evaluates it. The same short-circuiting behavior is evident for logical-or operations:

```
say 'Either true!' if ok 1, 'subexpression one'
                   || ok 1, 'subexpression two';
```

This example prints:

```
ok 1 - subexpression one
Either true!
```

Given the success of the first subexpression, Perl can avoid evaluating the second subexpression. If the first subexpression were false, the result of evaluating the second subexpression would dictate the result of evaluating the entire expression.

Besides allowing you to avoid potentially expensive computations, short circuiting can help you to avoid errors and warnings, as in the case where using an undefined value might raise a warning:

```
my $bbq;
if (defined $bbq and $bbq eq 'brisket') { ... }
```

Context for Conditional Directives

The conditional directives–`if`, `unless`, and the ternary conditional operator–all evaluate an expression in boolean context (Context, pp. 3). As comparison operators such as `eq`, `==`, `ne`, and `!=` all produce boolean results when evaluated, Perl coerces the results of other expressions–including variables and values–into boolean forms.

Perl has neither a single true value nor a single false value. Any number which evaluates to 0 is false. This includes `0`, `0.0`, `0e0`, `0x0`, and so on. The empty string (`''`) and `'0'` evaluate to a false value, but the strings `'0.0'`, `'0e0'`, and so on do not. The idiom `'0 but true'` evaluates to 0 in numeric context–but true in boolean context due to its string contents.

Both the empty list and `undef` evaluate to a false value. Empty arrays and hashes return the number 0 in scalar context, so they evaluate to a false value in boolean context. An array which contains a single element–even `undef`–evaluates to true in boolean context. A hash which contains any elements–even a key and a value of `undef`–evaluates to a true value in boolean context.

Greater Control Over Context

The `Want` module from the CPAN allows you to detect boolean context within your own functions. The core `overloading` pragma (Overloading, pp. 154) allows you to specify what your own data types produce when evaluated in various contexts.

Looping Directives

Perl provides several directives for looping and iteration. The *foreach*-style loop evaluates an expression which produces a list and executes a statement or block until it has exhausted that list:

```
# square the first ten positive integers
foreach (1 .. 10) {
    say "$_ * $_ = ", $_ * $_;
}
```

This example uses the range operator to produce a list of integers from one to ten inclusive. The `foreach` directive loops over them, setting the topic variable `$_` (The Default Scalar Variable, pp. 5) to each in turn. Perl executes the block for each integer and, as a result, prints the squares of the integers.

`foreach` versus `for`

Many Perl programmers refer to iteration as `foreach` loops, but Perl treats the names `foreach` and `for` interchangeably. The parenthesized expression determines the type and behavior of the loop; the keyword does not.

Like `if` and `unless`, this loop has a postfix form:

```
say "$_ * $_ = ", $_ * $_ for 1 .. 10;
```

A `for` loop may use a named variable instead of the topic:

```
for my $i (1 .. 10) {
    say "$i * $i = ", $i * $i;
}
```

When a `for` loop uses an iterator variable, the variable is scoped to the block *within* the loop. Perl will set this lexical to the value of each item in the iteration. Perl will not modify the topic variable (`$_`). If you have declared a lexical `$i` in an outer scope, its value will persist outside the loop:

```
my $i = 'cow';

for my $i (1 .. 10) {
    say "$i * $i = ", $i * $i;
}

is $i, 'cow', 'Value preserved in outer scope';
```

This localization occurs even if you do not redeclare the iteration variable as a lexical, but keep the habit of declaring iteration values as lexicals:

```
my $i = 'horse';

for $i (1 .. 10) {
    say "$i * $i = ", $i * $i;
}

is $i, 'horse', 'Value preserved in outer scope';
```

Iteration and Aliasing

The `for` loop *aliases* the iterator variable to the values in the iteration such that any modifications to the value of the iterator modifies the value in place:

```
my @nums = 1 .. 10;

$_ **= 2 for @nums;

is $nums[0], 1, '1 * 1 is 1';
is $nums[1], 4, '2 * 2 is 4';

...

is $nums[9], 100, '10 * 10 is 100';
```

This aliasing also works with the block style `for` loop:

```
for my $num (@nums) {
    $num **= 2;
}
```

. . . as well as iteration with the topic variable:

```
for (@nums) {
    $_ **= 2;
}
```

You cannot use aliasing to modify *constant* values, however. Perl will produce an exception about modification of read-only values.

```
$_++ and say for qw( Huex Dewex Louid );
```

You may occasionally see the use of `for` with a single scalar variable:

```
for ($user_input) {
    s/\A\s+//;      # trim leading whitespace
    s/\s+\z//;      # trim trailing whitespace

    $_ = quotemeta; # escape non-word characters
}
```

This idiom (Idioms, pp. 158) uses the iteration operator for its side effect of aliasing `$_`, though it's clearer to operate on the named variable itself.

Iteration and Scoping

The topic variable's iterator scoping has a subtle gotcha. Consider a function `topic_mangler()` which modifies `$_` on purpose. If code iterating over a list called `topic_mangler()` without protecting `$_`, you'd have to spend some time debugging the effects:

```
for (@values) {
    topic_mangler();
}

sub topic_mangler {
    s/foo/bar/;
}
```

The substitution in `topic_mangler()` will modify elements of `@values` in place. If you *must* use `$_` rather than a named variable, use the topic aliasing behavior of `for`:

```
sub topic_mangler {
    # was $_ = shift;
    for (shift)
    {
        s/foo/bar/;
        s/baz/quux/;
        return $_;
    }
}
```

Alternately, use a named iteration variable in the `for` loop. That's almost always the right advice.

The C-Style For Loop

The C-style *for loop* requires you to manage the conditions of iteration:

```
for (my $i = 0; $i <= 10; $i += 2) {
    say "$i * $i = ", $i * $i;
}
```

You must explicitly assign to an iteration variable in the looping construct, as this loop performs neither aliasing nor assignment to the topic variable. While any variable declared in the loop construct is scoped to the lexical block of the loop, Perl will not limit the lexical scope of a variable declared outside of the loop construct:

```
my $i = 'pig';

for ($i = 0; $i <= 10; $i += 2) {
```

```
    say "$i * $i = ", $i * $i;
}

isnt $i, 'pig', '$i overwritten with a number';
```

The looping construct may have three subexpressions. The first subexpression–the initialization section–executes only once, before the loop body executes. Perl evaluates the second subexpression–the conditional comparison–before each iteration of the loop body. When this evaluates to a true value, iteration proceeds. When it evaluates to a false value, iteration stops. The final subexpression executes after each iteration of the loop body.

```
for (
    # loop initialization subexpression
    say 'Initializing', my $i = 0;

    # conditional comparison subexpression
    say "Iteration: $i" and $i < 10;

    # iteration ending subexpression
    say 'Incrementing ' . $i++
) {
    say "$i * $i = ", $i * $i;
}
```

Note the lack of a semicolon after the final subexpression as well as the use of the comma operator and low-precedence `and`; this syntax is surprisingly finicky. When possible, prefer the `foreach`-style loop to the `for` loop.

All three subexpressions are optional. One infinite `for` loop is:

```
for (;;) { ... }
```

While and Until

A *while* loop continues until the loop conditional expression evaluates to a false value. An idiomatic infinite loop is:

```
while (1) { ... }
```

Unlike the iteration `foreach`-style loop, the `while` loop's condition has no side effects. If `@values` has one or more elements, this code is also an infinite loop, because every iteration will evaluate `@values` in scalar context to a non-zero value and iteration will continue:

```
while (@values) {
    say $values[0];
}
```

To prevent such an infinite `while` loop, use a *destructive update* of the `@values` array by modifying the array within each iteration:

```
while (@values) {
    my $value = shift @values;
    say $value;
}
```

Modifying `@values` inside of the `while` condition check also works, but it has some subtleties related to the truthiness of each value.

```
while (my $value = shift @values) {
    say $value;
}
```

This loop will exit as soon as *the assignment expression* used as the conditional expression evaluates to a false value. If that's what you intend, add a comment to the code.

The *until* loop reverses the sense of the test of the `while` loop. Iteration continues while the loop conditional expression evaluates to a false value:

```
until ($finished_running) {
    ...
}
```

The canonical use of the `while` loop is to iterate over input from a filehandle:

```
while (<$fh>) {
    # remove newlines
    chomp;
    ...
}
```

Perl interprets this `while` loop as if you had written:

```
while (defined($_ = <$fh>)) {
    # remove newlines
    chomp;
    ...
}
```

Without the implicit `defined`, any line read from the filehandle which evaluated to a false value in a scalar context–a blank line or a line which contained only the character `0`–would end the loop. The `readline` (`<>`) operator returns an undefined value only when it has reached the end of the file.

Both `while` and `until` have postfix forms, such as the infinite loop `1 while 1;`. Any single expression is suitable for a postfix `while` or `until`, including the classic "Hello, world!" example from 8-bit computers of the early 1980s:

```
print "Hello, world!  " while 1;
```

Infinite loops are more useful than they seem, especially for event loops in GUI programs, program interpreters, or network servers:

```
$server->dispatch_results until $should_shutdown;
```

Use a `do` block to group several expressions into a single unit:

```
do {
    say 'What is your name?';
    my $name = <>;
    chomp $name;
    say "Hello, $name!" if $name;
} until (eof);
```

A `do` block parses as a single expression which may contain several expressions. Unlike the `while` loop's block form, the `do` block with a postfix `while` or `until` will execute its body *at least* once. This construct is less common than the other loop forms, but very powerful.

Loops within Loops

You may nest loops within other loops:

```
for my $suit (@suits) {
    for my $values (@card_values) { ... }
}
```

Note the value of declaring iteration variables! The potential for confusion with the topic variable and its scope is too great otherwise.

Novices commonly exhaust filehandles accidentally while nesting `foreach` and `while` loops:

```
use autodie 'open';
open my $fh, '<', $some_file;

for my $prefix (@prefixes) {

    # DO NOT USE; buggy code
    while (<$fh>) {
        say $prefix, $_;
    }
}
```

Opening the filehandle outside of the `for` loop leaves the file position unchanged between each iteration of the `for` loop. On its second iteration, the `while` loop will have nothing to read (the `readline` will return a false value). You can solve this problem in many ways; re-open the file inside the `for` loop (wasteful but simple), slurp the entire file into memory (works best with small files), or `seek` the filehandle back to the beginning of the file for each iteration:

```
for my $prefix (@prefixes) {
    while (<$fh>) {
        say $prefix, $_;
    }

    seek $fh, 0, 0;
}
```

Loop Control

Sometimes you must break out of a loop before you have exhausted the iteration conditions. Perl's standard control mechanisms–exceptions and `return`–work, but you may also use *loop control* statements.

The *next* statement restarts the loop at its next iteration. Use it when you've done everything you need to in the current iteration. To loop over lines in a file and skip everything that starts with the comment character `#`:

```
while (<$fh>) {
    next if /\A#/;
    ...
}
```

Multiple Exits versus Nested Ifs

Compare the use of `next` with the alternative: wrapping the rest of the body of the block in an `if`. Now consider what happens if you have multiple conditions which could cause you to skip a line. Loop control modifiers with postfix conditionals can make your code much more readable.

The *last* statement ends the loop immediately. To finish processing a file once you've seen the ending token, write:

```
while (<$fh>) {
    next if /\A#/;
    last if /\A__END__/
    ...
}
```

The *redo* statement restarts the current iteration without evaluating the conditional again. This can be useful in those few cases where you want to modify the line you've read in place, then start processing over from the beginning without clobbering it with another line. To implement a silly file parser that joins lines which end with a backslash:

```
while (my $line = <$fh>) {
    chomp $line;

    # match backslash at the end of a line
    if ($line =~ s{\\$}{})
    {
        $line .= <$fh>;
        redo;
    }

    ...
}
```

Nested loops can be confusing, especially with loop control statements. If you cannot extract inner loops into named functions, use *loop labels* to clarify your intent:

```
LINE:
while (<$fh>) {
    chomp;

    PREFIX:
    for my $prefix (@prefixes) {
        next LINE unless $prefix;
        say "$prefix: $_";
        # next PREFIX is implicit here
    }
}
```

Continue

The `continue` construct behaves like the third subexpression of a `for` loop; Perl executes any continue block before subsequent iterations of a loop, whether due to normal loop repetition or premature re-iteration from `next`. You may use it with a `while`, `until`, `when`, or `for` loop. Examples of `continue` are rare, but it's useful any time you want to guarantee that something occurs with every iteration of the loop, regardless of how that iteration ends:

```
while ($i < 10 ) {
    next unless $i % 2;
    say $i;
}
continue {
    say 'Continuing...';
    $i++;
}
```

Be aware that a `continue` block does *not* execute when control flow leaves a loop due to `last` or `redo`.

Switch Statements

Perl 5.10 introduced a new construct named `given` as a Perlish `switch` statement. It didn't quite work out; `given` is still experimental, if less buggy in newer releases. Avoid it unless you know exactly what you're doing.

If you need a switch statement, use `for` to alias the topic variable (`$_`) and `when` to match it against simple expressions with smart match (Smart Matching, pp. 105) semantics. To write the Rock, Paper, Scissors game:

```
my @options  = ( \&rock, \&paper, \&scissors );
my $confused = "I don't understand your move.";

do {
    say "Rock, Paper, Scissors!  Pick one: ";
    chomp( my $user = <STDIN> );
    my $computer_match = $options[ rand @options ];
    $computer_match->( lc( $user ) );
} until (eof);

sub rock {
    print "I chose rock.  ";

    for (shift) {
        when (/paper/)    { say 'You win!' };
        when (/rock/)     { say 'We tie!'  };
        when (/scissors/) { say 'I win!'   };
        default           { say $confused  };
    }
}

sub paper {
    print "I chose paper.  ";

    for (shift) {
        when (/paper/)    { say 'We tie!'  };
        when (/rock/)     { say 'I win!'   };
        when (/scissors/) { say 'You win!' };
        default           { say $confused  };
    }
}

sub scissors {
    print "I chose scissors.  ";

    for (shift) {
        when (/paper/)    { say 'I win!'   };
        when (/rock/)     { say 'You win!' };
        when (/scissors/) { say 'We tie!'  };
        default           { say $confused  };
    }
}
```

Perl executes the `default` rule when none of the other conditions match. Adding Spock and Lizard is left as an exercise for the reader.

Tailcalls

A *tailcall* occurs when the last expression within a function is a call to another function. The outer function's return value becomes the inner function's return value:

```
sub log_and_greet_person {
    my $name = shift;
    log( "Greeting $name" );

    return greet_person( $name );
}
```

Returning from `greet_person()` directly to the caller of `log_and_greet_person()` is more efficient than returning *to* `log_and_greet_person()` and then *from* `log_and_greet_person()`. Returning directly *from* `greet_person()` to the caller of `log_and_greet_person()` is a *tailcall optimization*.

Heavily recursive code (Recursion, pp. 74)–especially mutually recursive code–can consume a lot of memory. Tailcalls reduce the memory needed for internal bookkeeping of control flow and can make expensive algorithms cheaper. Unfortunately, Perl does not automatically perform this optimization, so you have to do it yourself when it's necessary.

The builtin `goto` operator has a form which calls a function as if the current function were never called, essentially erasing the bookkeeping for the new function call. The ugly syntax confuses people who've heard "Never use `goto`", but it works:

```
sub log_and_greet_person {
    my ($name) = @_;
    log( "Greeting $name" );

    goto &greet_person;
}
```

This example has two important characteristics. First, `goto &function_name` or `goto &$function_reference` requires the use of the function sigil (`&`) so that the parser knows to perform a tailcall instead of jumping to a label. Second, this form of function call passes the contents of `@_` implicitly to the called function. You may modify `@_` to change the passed arguments if you desire.

This technique is most useful when you want to hijack control flow to get out of the way of other functions inspecting `caller` (such as when you're implementing special logging or some sort of debugging feature), or when using an algorithm which requires a lot of recursion. Remember it if you need it, but feel free not to use it.

Scalars

Perl's fundamental data type is the *scalar*: a single, discrete value. That value may be a string, an integer, a floating point value, a filehandle, or a reference–but it is always a single value. Scalars may be lexical, package, or global (Global Variables, pp. 163) variables. You may only declare lexical or package variables. The names of scalar variables must conform to standard variable naming guidelines (Names, pp. 13). Scalar variables always use the leading dollar-sign (`$`) sigil (Variable Sigils, pp. 15).

Variant Sigils and Context

Scalar values and scalar context have a deep connection; assigning to a scalar imposes scalar context. Using the scalar sigil with an aggregate variable accesses a single element of the hash or array in scalar context.

Scalars and Types

A scalar variable can contain any type of scalar value without special conversions, coercions, or casts. The type of value stored in a scalar variable, once assigned, can change arbitrarily:

```
my $value;
$value = 123.456;
$value = 77;
$value = "I am Chuck's big toe.";
$value = Store::IceCream->new;
```

Even though this code is *legal*, changing the type of data stored in a scalar is confusing.

This flexibility of type often leads to value coercion (Coercion, pp. 49). For example, you may treat the contents of a scalar as a string, even if you didn't explicitly assign it a string:

```
my $zip_code       = 97123;
my $city_state_zip = 'Hillsboro, Oregon' . ' ' . $zip_code;
```

You may also use mathematical operations on strings:

```
my $call_sign = 'KBMIU';

# update sign in place and return new value
my $next_sign = ++$call_sign;

# return old value, then update sign
my $curr_sign = $call_sign++;

# but does not work as:
my $new_sign  = $call_sign + 1;
```

One-Way Increment Magic

This magical string increment behavior has no corresponding magical decrement behavior. You can't restore the previous string value by writing `$call_sign--`.

This string increment operation turns `a` into `b` and `z` into `aa`, respecting character set and case. While `ZZ9` becomes `AAA0`, `ZZ09` becomes `ZZ10`–numbers wrap around while there are more significant places to increment, as on a vehicle odometer.

Evaluating a reference (References, pp. 53) in string context produces a string. Evaluating a reference in numeric context produces a number. Neither operation modifies the reference in place, but you cannot recreate the reference from either result:

```
my $authors     = [qw( Pratchett Vinge Conway )];
my $stringy_ref = '' . $authors;
my $numeric_ref =  0 + $authors;
```

`$authors` is still useful as a reference, but `$stringy_ref` is a string with no connection to the reference and `$numeric_ref` is a number with no connection to the reference.

To allow coercion without data loss, Perl scalars can contain both numeric and string components. The internal data structure which represents a scalar in Perl has a numeric slot and a string slot. Accessing a string in a numeric context produces a scalar with both string and numeric values.

Scalars do not contain a separate slot for boolean values. In boolean context, the empty strings (`''`) and `'0'` evaluate to false values. All other strings evaluate to true values. In boolean context, numbers which evaluate to zero (`0`, `0.0`, and `0e0`) evaluate to false values. All other numbers evaluate to true values.

What is Truth?

Be careful that the *strings* `'0.0'` and `'0e0'` evaluate to true values. This is one place where Perl makes a distinction between what *looks like* a number and what really is a number.

`undef` is always a false value.

Arrays

Perl's *array* data type is a language-supported aggregate which can store zero or more scalars. You can access individual members of the array by integer indexes, and you can add or remove elements at will. Arrays grow or shrink as you manipulate them.

The `@` sigil denotes an array. To declare an array:

```
my @items;
```

Array Elements

Use the scalar sigil to *access* an individual element of an array. `$cats[0]` refers unambiguously to the `@cats` array, because postfix (Fixity, pp. 64) square brackets (`[]`) always mean indexed access to an array.

The first element of an array is at the zeroth index:

```
# @cats contains a list of Cat objects
my $first_cat = $cats[0];
```

The last index of an array depends on the number of elements in the array. An array in scalar context (due to scalar assignment, string concatenation, addition, or boolean context) evaluates to the number of elements in the array:

```
# scalar assignment
my $num_cats = @cats;

# string concatenation
say 'I have ' . @cats . ' cats!';

# addition
my $num_animals = @cats + @dogs + @fish;

# boolean context
say 'Yep, a cat owner!' if @cats;
```

To get the *index* of the final element of an array, subtract one from the number of elements of the array (because array indexes start at 0) or use the unwieldy `$#cats` syntax:

```
my $first_index = 0;
my $last_index  = @cats - 1;
# or
# my $last_index = $#cats;

say   "My first cat has an index of $first_index, "
    . "and my last cat has an index of $last_index."
```

Most of the time you care more about the relative position of an array element. Use a negative array index to refer to elements from the end. The last element of an array is available at the index `-1`. The second to last element of the array is available at index `-2`, and so on:

```
my $last_cat           = $cats[-1];
my $second_to_last_cat = $cats[-2];
```

`$#` has another use: resize an array in place by *assigning* to `$#array`. Remember that Perl arrays are mutable. They expand or contract as necessary. When you shrink an array, Perl will discard values which do not fit in the resized array. When you expand an array, Perl will fill the expanded positions with `undef`.

Array Assignment

Assign to individual positions in an array directly by index:

```
my @cats;
$cats[3] = 'Jack';
$cats[2] = 'Tuxedo';
$cats[0] = 'Daisy';
$cats[1] = 'Petunia';
$cats[4] = 'Brad';
$cats[5] = 'Choco';
```

If you assign to an index beyond the array's current bounds, Perl will extend the array for you. As you might expect, all intermediary positions with then contain `undef`. After the first assignment, the array will contain `undef` at positions 0, 1, and 2 and `Jack` at position 3.

As an assignment shortcut, initialize an array from a list:

```
my @cats = ( 'Daisy', 'Petunia', 'Tuxedo', ... );
```

. . . but remember that these parentheses *do not* create a list. Without parentheses, this would assign `Daisy` as the first and only element of the array, due to operator precedence (Precedence, pp. 63). `Petunia`, `Tuxedo`, and all of the other cats would be evaluated in void context and Perl would complain. (So would all the other cats, especially Petunia.)

You may assign any expression which produces a list to an array:

```
my @cats     = get_cat_list();
my @timeinfo = localtime();
my @nums     = 1 .. 10;
```

Assigning to a scalar element of an array imposes scalar context, while assigning to the array as a whole imposes list context.

To clear an array, assign an empty list:

```
my @dates = ( 1969, 2001, 2010, 2051, 1787 );
...
@dates    = ();
```

This is one of the only cases where parentheses *do* indicate a list; without something to mark a list, Perl and readers of the code would get confused.

Arrays Start Empty

`my @items = ();` is a longer and noisier version of `my @items`. Freshly-declared arrays start out empty. Not "full of `undef`" empty. Really empty.

Array Operations

Sometimes an array is more convenient as an ordered, mutable collection of items than as a mapping of indices to values. Perl provides several operations to manipulate array elements.

The `push` and `pop` operators add and remove elements from the tail of an array, respectively:

```
my @meals;

# what is there to eat?
push @meals, qw( hamburgers pizza lasagna turnip );

# ... but your nephew hates vegetables
pop @meals;
```

You may `push` a list of values onto an array, but you may only `pop` one at a time. `push` returns the new number of elements in the array. `pop` returns the removed element.

Because `push` operates on a list, you can easily append the elements of one multiple arrays with:

```
push @meals, @breakfast, @lunch, @dinner;
```

Similarly, `unshift` and `shift` add elements to and remove an element from the start of an array, respectively:

```
# expand our culinary horizons
unshift @meals, qw( tofu spanakopita taquitos );

# rethink that whole soy idea
shift @meals;
```

`unshift` prepends a list of elements to the start of the array and returns the new number of elements in the array. `shift` removes and returns the first element of the array. Almost no one uses these return values.

The `splice` operator removes and replaces elements from an array given an offset, a length of a list slice, and replacement elements. Both replacing and removing are optional; you may omit either. The `perlfunc` description of `splice` demonstrates its equivalences with `push`, `pop`, `shift`, and `unshift`. One effective use is removal of two elements from an array:

```
my ($winner, $runnerup) = splice @finalists, 0, 2;

# or
my $winner              = shift @finalists;
my $runnerup            = shift @finalists;
```

The `each` operator allows you to iterate over an array by index and value:

```
while (my ($position, $title) = each @bookshelf) {
    say "#$position: $title";
}
```

Array Slices

An *array slice* allows you to access elements of an array in list context. Unlike scalar access of an array element, this indexing operation takes a list of zero or more indices and uses the array sigil (`@`):

```
my @youngest_cats = @cats[-1, -2];
my @oldest_cats   = @cats[0 .. 2];
my @selected_cats = @cats[ @indexes ];
```

Array slices are useful for assignment:

```
@users[ @replace_indices ] = @replace_users;
```

The only syntactic difference between an array slice of one element and the scalar access of an array element is the leading sigil. The *semantic* difference is greater: an array slice always imposes list context. An array slice evaluated in scalar context will produce a warning:

```
  Scalar value @cats[1] better written as $cats[1]...
```

An array slice imposes list context on the expression used as its index:

```
# function called in list context
my @hungry_cats = @cats[ get_cat_indices() ];
```

A slice can contain zero or more elements–including one:

```
# single-element array slice; list context
@cats[-1] = get_more_cats();

# single-element array access; scalar context
$cats[-1] = get_more_cats();
```

Arrays and Context

In list context, arrays flatten into lists. If you pass multiple arrays to a normal function, they will flatten into a single list:

```
my @cats = qw( Daisy Petunia Tuxedo Brad Jack Choco );
my @dogs = qw( Rodney Lucky Rosie );

take_pets_to_vet( @cats, @dogs );

sub take_pets_to_vet {
    # BUGGY: do not use!
    my (@cats, @dogs) = @_;
    ...
}
```

Within the function, `@_` will contain nine elements, not two, because list assignment to arrays is *greedy*. An array will consume as many elements from the list as possible. After the assignment, `@cats` will contain *every* argument passed to the function. `@dogs` will be empty, and woe to anyone who treats Rodney as a cat.

This flattening behavior sometimes confuses people who attempt to create nested arrays:

```
# creates a single array, not an array of arrays
my @numbers = (1 .. 10, (11 .. 20, (21 .. 30)));
```

...but this code is effectively the same as either:

```
# parentheses do not create lists
my @numbers = ( 1 .. 10, 11 .. 20, 21 .. 30 );

# creates a single array, not an array of arrays
my @numbers = 1 .. 30;
```

...because, again, parentheses merely group expressions. They do not *create* lists. To avoid this flattening behavior, use array references (Array References, pp. 54).

Array Interpolation

Arrays interpolate in strings as lists of the stringification of each item separated by the current value of the magic global `$"`. The default value of this variable is a single space. Its *English.pm* mnemonic is `$LIST_SEPARATOR`. Thus:

```
my @alphabet = 'a' .. 'z';
say "[@alphabet]";
[a b c d e f g h i j k l m
 n o p q r s t u v w x y z]
```

Per Mark Jason Dominus, localize `$"` with a delimiter to improve your debugging:

```
# what's in this array again?
local $" = ')(';
say "(@sweet_treats)";
(pie)(cake)(doughnuts)(cookies)(cinnamon roll)
```

Hashes

A *hash* is an aggregate data structure which associates string keys with scalar values. Just as the name of a variable corresponds to something which holds a value, so a hash key refers to something which contains a value. Think of a hash like a contact list: use the names of your friends to look up their birthdays. Other languages call hashes *tables*, *associative arrays*, *dictionaries*, and *maps*.

Hashes have two important properties: they store one scalar per unique key and they provide no specific ordering of keys. Keep that latter property in mind. Though it has always been true in Perl, it's very, very true in modern Perl.

Declaring Hashes

Hashes use the `%` sigil. Declare a lexical hash with:

```
my %favorite_flavors;
```

A hash starts out empty. You could write `my %favorite_flavors = ();`, but that's redundant.

Hashes use the scalar sigil `$` when accessing individual elements and curly braces `{ }` for keyed access:

```
my %favorite_flavors;
$favorite_flavors{Gabi}    = 'Dark chocolate raspberry';
$favorite_flavors{Annette} = 'French vanilla';
```

Assign a list of keys and values to a hash in a single expression:

```
my %favorite_flavors = (
    'Gabi',    'Dark chocolate raspberry',
    'Annette', 'French vanilla',
);
```

Hashes store pairs of keys and values. Perl will warn you if you assign an odd number of elements to a hash. Idiomatic Perl often uses the *fat comma* operator (`=>`) to associate values with keys, as it makes the pairing more visible:

```
my %favorite_flavors = (
    Gabi    => 'Dark chocolate raspberry',
    Annette => 'French vanilla',
);
```

The fat comma operator acts like the regular comma *and* also automatically quotes the previous bareword (Barewords, pp. 166). The `strict` pragma will not warn about such a bareword–and if you have a function with the same name as a hash key, the fat comma will *not* call the function:

```
sub name { 'Leonardo' }

my %address = (
    name => '1123 Fib Place'
);
```

The key of this hash will be `name` and not `Leonardo`. To call the function, make the function call explicit:

```
my %address = (
    name() => '1123 Fib Place'
);
```

You may occasionally see `undef %hash`, but that's a little ugly. Assign an empty list to empty a hash:

```
%favorite_flavors = ();
```

Hash Indexing

To access an individual hash value, use the *keyed access* syntax:

```
my $address = $addresses{$name};
```

In this example, `$name` contains a string which is also a key of the hash. As with accessing an individual element of an array, the hash's sigil has changed from `%` to `$` to indicate keyed access to a scalar value.

You may also use string literals as hash keys. Perl quotes barewords automatically according to the same rules as fat commas:

```
# auto-quoted
my $address = $addresses{Victor};

# needs quoting; not a valid bareword
my $address = $addresses{'Sue-Linn'};

# function call needs disambiguation
my $address = $addresses{get_name()};
```

Don't Quote Me

Novices often always quote string literal hash keys, but experienced developers elide the quotes whenever possible. If you code this way, you can use the rare presence of quotes to indicate that you're doing something special on purpose.

Even Perl builtins get the autoquoting treatment:

```
my %addresses = (
    Leonardo => '1123 Fib Place',
    Utako    => 'Cantor Hotel, Room 1',
```

```
);

sub get_address_from_name {
    return $addresses{+shift};
}
```

The unary plus (Unary Coercions, pp. 163) turns what would be a bareword (`shift`) subject to autoquoting rules into an expression. As this implies, you can use an arbitrary expression–not only a function call–as the key of a hash:

```
# don't actually do this though
my $address = $addresses{reverse 'odranoeL'};

# interpolation is fine
my $address = $addresses{"$first_name $last_name"};

# so are method calls
my $address = $addresses{ $user->name };
```

Hash keys can only be strings. Anything that evaluates to a string is an acceptable hash key. Perl will go so far as to coerce (Coercion, pp. 49) an expression into a string. For example, if you use an object as a hash key, you'll get the stringified version of that object instead of the object itself:

```
for my $isbn (@isbns) {
    my $book = Book->fetch_by_isbn( $isbn );

    # unlikely to do what you want
    $books{$book} = $book->price;
}
```

That stringified hash will look something like `Book=HASH(0x222d148)`. `Book` refers to the class name. `HASH` identifies the object as a blessed reference. `0x22d148` is a number used to identify the object (more precisely: it's the location of the data structure representing the hash in memory, so it's neither quite random nor unique).

Hash Key Existence

The `exists` operator returns a boolean value to indicate whether a hash contains the given key:

```
my %addresses = (
    Leonardo => '1123 Fib Place',
    Utako    => 'Cantor Hotel, Room 1',
);

say "Have Leonardo's address" if exists $addresses{Leonardo};
say "Have Warnie's address"   if exists $addresses{Warnie};
```

Using `exists` instead of accessing the hash key directly avoids two problems. First, it does not check the boolean nature of the hash *value*; a hash key may exist with a value even if that value evaluates to a boolean false (including `undef`):

```
my %false_key_value = ( 0 => '' );
ok %false_key_value,
   'hash containing false key & value should evaluate to a true value';
```

Second, `exists` avoids autovivification (Autovivification, pp. 60) within nested data structures (Nested Data Structures, pp. 58).

If a hash key exists, its value may be `undef`. Check that with `defined`:

```
$addresses{Leibniz} = undef;

say "Gottfried lives at $addresses{Leibniz}"
    if exists  $addresses{Leibniz}
    && defined $addresses{Leibniz};
```

Accessing Hash Keys and Values

Hashes are aggregate variables, but their pairwise nature is unique. Perl allows you to iterate over a hash's keys, its values, or pairs of its keys and values. The `keys` operator produces a list of hash keys:

```
for my $addressee (keys %addresses) {
    say "Found an address for $addressee!";
}
```

The `values` operator produces a list of hash values:

```
for my $address (values %addresses) {
    say "Someone lives at $address";
}
```

The `each` operator produces a list of two-element key/value lists:

```
while (my ($addressee, $address) = each %addresses) {
    say "$addressee lives at $address";
}
```

Unlike arrays, hash elements have no obvious ordering. The ordering depends on the internal implementation of the hash, the particular version of Perl you are using, the size of the hash, and a random factor. Even so, the order of hash items is consistent between `keys`, `values`, and `each`. Modifying the hash may change the order, but you can rely on that order if the hash remains the same. However, even if two hashes have the *same* keys and values, you cannot rely on the iteration order between those hashes being the same. They may have been constructed differently or have had elements removed. Since Perl 5.18, even if you build two hashes in the same way, you cannot depend on the same iteration order between them.

Read the previous paragraph again. You'll be glad you did.

Each hash has only a *single* iterator for the `each` operator. You cannot reliably iterate over a hash with `each` more than once; if you begin a new iteration while another is in progress, the former will end prematurely and the latter will begin partway through the iteration. Beware not to call any function which may itself try to iterate over the hash with `each`.

This is rarely a problem, but it's not fun to debug. Reset a hash's iterator with `keys` or `values` in void context:

```
# reset hash iterator
keys %addresses;

while (my ($addressee, $address) = each %addresses) {
    ...
}
```

Hash Slices

A *hash slice* is a list of keys or values of a hash indexed in a single operation. To initialize multiple elements of a hash at once:

```
my %cats;
@cats{qw( Jack Brad Mars Grumpy )} = (1) x 4;
```

This is equivalent to the initialization:

```
my %cats = map { $_ => 1 } qw( Jack Brad Mars Grumpy );
```

Note however that the hash slice assignment can also add to the existing contents of the hash.

Hash slices also allow you to retrieve multiple values from a hash in a single operation. As with array slices, the sigil of the hash changes to `@` to indicate list context. The use of the curly braces indicates keyed access and makes the fact that you're working with a hash unambiguous:

```
my @buyer_addresses = @addresses{ @buyers };
```

Hash slices make it easy to merge two hashes:

```
my %addresses        = ( ... );
my %canada_addresses = ( ... );

@addresses{ keys %canada_addresses } = values %canada_addresses;
```

This is equivalent to looping over the contents of `%canada_addresses` manually, but is much shorter. Note that this relies on the iteration order of the hash remaining consistent between `keys` and `values`. Perl guarantees this, but only because these operations occur on the same hash with no modifications to that hash between the `keys` and `values` operations.

What if the same key occurs in both hashes? The hash slice approach always *overwrites* existing key/value pairs in `%addresses`. If you want other behavior, looping is more appropriate.

The Empty Hash

An empty hash contains no keys or values. It evaluates to a false value in a boolean context. A hash which contains at least one key/value pair evaluates to a true value in boolean context even if all of the keys or all of the values or both would themselves evaluate to boolean false values.

```
use Test::More;

my %empty;
ok ! %empty, 'empty hash should evaluate false';

my %false_key = ( 0 => 'true value' );
ok %false_key, 'hash containing false key should evaluate to true';

my %false_value = ( 'true key' => 0 );
ok %false_value, 'hash containing false value should evaluate to true';

done_testing();
```

In scalar context, a hash evaluates to a string which represents the ratio of full buckets in the hash–internal details about the hash implementation that you can safely ignore. In a boolean scalar context, this ratio evaluates to a false value, so remember *that* instead of the ratio details.

In list context, a hash evaluates to a list of key/value pairs similar to the list produced by the `each` operator. However, you *cannot* iterate over this list the same way you can iterate over the list produced by `each`. This loop will never terminate:

```
# infinite loop for non-empty hashes
while (my ($key, $value) = %hash) {
    ...
}
```

You *can* loop over the list of keys and values with a `for` loop, but the iterator variable will get a key on one iteration and its value on the next, because Perl will flatten the hash into a single list of interleaved keys and values.

Hash Idioms

Because each key exists only once in a hash, assigning the same key to a hash multiple times stores only the most recent value associated with that key. This behavior has advantages! For example, to find unique elements of a list:

```
my %uniq;
undef @uniq{ @items };
my @uniques = keys %uniq;
```

Using `undef` with a hash slice sets the values of the hash to `undef`. This idiom is the cheapest way to perform set operations with a hash.

Hashes are useful for counting elements, such as IP addresses in a log file:

```
my %ip_addresses;

while (my $line = <$logfile>) {
    chomp $line;
    my ($ip, $resource) = analyze_line( $line );
    $ip_addresses{$ip}++;
    ...
}
```

The initial value of a hash value is `undef`. The postincrement operator (`++`) treats that as zero. This in-place modification of the value increments an existing value for that key. If no value exists for that key, Perl creates a value (`undef`) and immediately increments it to one, as the numification of `undef` produces the value 0.

This strategy provides a useful caching mechanism to store the result of an expensive operation with little overhead:

```
{
    my %user_cache;

    sub fetch_user {
        my $id = shift;
        $user_cache{$id} //= create_user($id);
        return $user_cache{$id};
    }
}
```

This *orcish maneuver* (or-cache) returns the value from the hash, if it exists. Otherwise, it calculates, caches, and returns the value. The defined-or assignment operator (`//=`) evaluates its left operand. If that operand is not defined, the operator assigns to the lvalue the value of its right operand. In other words, if there's no value in the hash for the given key, this function will call `create_user()` with the key and update the hash.

You may see older code which uses the boolean-or assignment operator (`||=`) for this purpose. Remember though that some valid values evaluate as false in a boolean context. The defined-or operator usually makes more sense, as it tests for definedness instead of truthiness.

If your function takes several arguments, use a slurpy hash (Slurping, pp. 71) to gather key/value pairs into a single hash as named function arguments:

```
sub make_sundae {
    my %parameters = @_;
    ...
}

make_sundae( flavor  => 'Lemon Burst',
             topping => 'cookie bits' );
```

This approach allows you to set default values:

```
sub make_sundae {
    my %parameters           = @_;
    $parameters{flavor}    //= 'Vanilla';
    $parameters{topping}   //= 'fudge';
    $parameters{sprinkles} //= 100;
    ...
}
```

...or include them in the hash initialization, as latter assignments take precedence over earlier assignments:

```
sub make_sundae {
    my %parameters = (
        flavor    => 'Vanilla',
        topping   => 'fudge',
        sprinkles => 100,
        @_,
    );
    ...
}
```

Locking Hashes

As hash keys are barewords, they offer little typo protection compared to the function and variable name protection offered by the `strict` pragma. The little-used core module `Hash::Util` can make hashes safer.

To prevent someone from accidentally adding a hash key you did not intend (whether as a typo or from untrusted user input), use the `lock_keys()` function to restrict the hash to its current set of keys. Any attempt to add a new key to the hash will raise an exception. Similarly you can lock or unlock the existing value for a given key in the hash (`lock_value()` and `unlock_value()`) and make or unmake the entire hash read-only with `lock_hash()` and `unlock_hash()`.

This is lax security; anyone can use the appropriate unlocking functions to work around the locking. Yet it does protect against typos and other unintended accidents.

Coercion

Throughout the lifetime of a Perl variable, it may contain values of different types–strings, integers, rational numbers, and more. Instead of attaching type information to variables, Perl relies on the context provided by operators (Numeric, String, and Boolean Context, pp. 5) to determine how to handle values. By design, Perl attempts to do what you mean–you may hear this referred to as *DWIM* for *do what I mean* or *dwimmery*.>–though you must be specific about your intentions. If you treat a value as a string, Perl will do its best to *coerce* that value into a string.

Boolean Coercion

Boolean coercion occurs when you test the *truthiness* of a value, such as in an `if` or `while` condition. Numeric 0, `undef`, the empty string, and the string `'0'` all evaluate as false values. All other values–including strings which may be *numerically* equal to zero (such as `'0.0'`, `'0e'`, and `'0 but true'`)–evaluate as true values.

When a scalar has *both* string and numeric components (Dualvars, pp. 51), Perl prefers to check the string component for boolean truth. `'0 but true'` evaluates to zero numerically, but it is not an empty string, so it evaluates to a true value in boolean context.

String Coercion

String coercion occurs when using string operators such as comparisons (`eq` and `cmp`), concatenation, `split`, `substr`, and regular expressions, as well as when using a value or an expression as a hash key. The undefined value stringifies to an empty string but produces a "use of uninitialized value" warning. Numbers *stringify* to strings containing their values; the value `10` stringifies to the string `10`. You can even `split` a number into individual digits with:

```
my @digits = split '', 1234567890;
```

Numeric Coercion

Numeric coercion occurs when using numeric comparison operators (such as == and <=>), when performing mathematic operations, and when using a value or expression as an array or list index. The undefined value *numifies* to zero and produces a "Use of uninitialized value" warning. Strings which do not begin with numeric portions numify to zero and produce an "Argument isn't numeric" warning. Strings which begin with characters allowed in numeric literals numify to those values and produce no warnings, such that `10 leptons leaping` numifies to `10` and `6.022e23 moles marauding` numifies to `6.022e23`.

The core module `Scalar::Util` contains a `looks_like_number()` function which uses the same rules as the Perl parser to extract a number from a string.

Mathematicians Rejoice

The strings `Inf` and `Infinity` represent the infinite value and behave as numbers. The string `NaN` represents the concept "not a number". Numifying them produces no "Argument isn't numeric" warning. Beware that Perl's ideas of infinity and not a number may not match your platform's ideas; these notions aren't always portable across operating systems. Perl is consistent even if the rest of the universe isn't.

Reference Coercion

Using a dereferencing operation on a non-reference turns that value *into* a reference. This process of autovivification (Autovivification, pp. 60) is handy when manipulating nested data structures (Nested Data Structures, pp. 58):

```
my %users;

$users{Brad}{id} = 228;
$users{Jack}{id} = 229;
```

Although the hash never contained values for `Brad` and `Jack`, Perl helpfully created hash references for them, then assigned each a key/value pair keyed on `id`.

Cached Coercions

Perl's internal representation of values stores both string and numeric values. Stringifying a numeric value does not *replace* the numeric value. Instead, it *adds* a stringified value to the internal representation, which then contains *both* components. Similarly, numifying a string value populates the numeric component while leaving the string component untouched.

Certain Perl operations prefer to use one component of a value over another–boolean checks prefer strings, for example. If a value has a cached representation in a form you do not expect, relying on an implicit conversion may produce surprising results. You almost never need to be explicit about what you expect. Your author can recall doing so twice in two decades. Even so, knowing that this caching occurs may someday help you diagnose an odd situation.

Dualvars

The multi-component nature of Perl values is available to users in the form of *dualvars*. The core module `Scalar::Util` provides a function `dualvar()` which allows you to bypass Perl coercion and manipulate the string and numeric components of a value separately:

```
use Scalar::Util 'dualvar';
my $false_name = dualvar 0, 'Sparkles & Blue';

say 'Boolean true!'  if        !! $false_name;
say 'Numeric false!' unless  0  + $false_name;
say 'String true!'   if     ''  . $false_name;
```

Packages

A Perl *namespace* associates and encapsulates various named entities. It's like your family name or a brand name. Unlike a real-world name, a namespace implies no direct relationship between entities. Such relationships may exist, but they are not required to.

A *package* in Perl is a collection of code in a single namespace. The distinction is subtle: the package represents the source code and the namespace represents the internal data structure Perl uses to collect and group that code.

The `package` builtin declares a package and a namespace:

```
package MyCode;

our @boxes;

sub add_box { ... }
```

All global variables and functions declared or referred to after the package declaration refer to symbols within the `MyCode` namespace. You can refer to the `@boxes` variable from the `main` namespace only by its *fully qualified* name of `@MyCode::boxes`. A fully qualified name includes a complete package name, so that you can call the `add_box()` function by `MyCode::add_box()`.

The scope of a package continues until the next `package` declaration or the end of the file, whichever comes first. You may also provide a block with `package` to delineate the scope of the declaration:

```
package Pinball::Wizard
{
    our $VERSION = 1969;
}
```

The default package is the `main` package. Without a package declaration, the current package is `main`. This rule applies to one-liners, standalone programs, and even *.pm* files.

Besides a name, a package has a version and three implicit methods, `import()` (Importing, pp. 72), `unimport()`, and `VERSION()`. `VERSION()` returns the package's version. This is a series of numbers contained in a package global named `$VERSION`. By rough convention, versions are a series of dot-separated integers such as `1.23` or `1.1.10`.

Perl includes a stricter syntax for version numbers, as documented in `perldoc version::Internals`. These version numbers must have a leading `v` character and at least three integer components separated by periods:

```
package MyCode v1.2.1;
```

Combined with the block form of a `package` declaration, you can write:

```
package Pinball::Wizard v1969.3.7 { ... }
```

You're more likely to see the older version of this code, written as:

```
package MyCode;

our $VERSION = 1.21;
```

Every package inherits a `VERSION()` method from the `UNIVERSAL` base class. This method returns the value of `$VERSION`:

```
my $version = Some::Plugin->VERSION;
```

If you provide a version number as an argument, this method will throw an exception unless the version of the module is equal to or greater than the argument:

```
# require at least 2.1
Some::Plugin->VERSION( 2.1 );

die "Your plugin $version is too old" unless $version > 2;
```

You may override `VERSION()`, though there are few reasons to do so.

Packages and Namespaces

Every `package` declaration creates a new namespace, if necessary. After Perl parses that declaration, it will store all subsequent package global symbols (global variables and functions) in that namespace.

Perl has *open namespaces*. You can add functions or variables to a namespace at any point, either with a new package declaration:

```
package Pack
{
    sub first_sub { ... }
}

Pack::first_sub();

package Pack
{
    sub second_sub { ... }
}

Pack::second_sub();
```

...or by declaring functions with fully qualified names:

```
# implicit
package main;

sub Pack::third_sub { ... }
```

You can add to a package at any point during compilation or runtime, regardless of the current file, though building up a package from multiple declarations in multiple files can make code difficult to spelunk.

Namespaces can have as many levels as your organizational scheme requires, though namespaces are not hierarchical. The only relationship between separate packages is semantic, not technical. Many projects and businesses create their own top-level namespaces. This reduces the possibility of global conflicts and helps to organize code on disk. For example:

- `StrangeMonkey` is the project name
- `StrangeMonkey::UI` organizes user interface code
- `StrangeMonkey::Persistence` organizes data management code
- `StrangeMonkey::Test` organizes testing code for the project

. . . and so on. This is a convention, but it's a useful one.

References

Perl usually does what you expect, even if what you expect is subtle. Consider what happens when you pass values to functions:

```
sub reverse_greeting {
    my $name = reverse shift;
    return "Hello, $name!";
}

my $name = 'Chuck';
say reverse_greeting( $name );
say $name;
```

Outside of the function, `$name` contains `Chuck`, even though the value passed into the function gets reversed into `kcuhC`. You probably expected that. The value of `$name` outside the function is separate from the `$name` inside the function. Modifying one has no effect on the other.

Consider the alternative. If you had to make copies of every value before anything could possibly change them out from under you, you'd have to write lots of extra defensive code.

Sometimes it's useful to modify values in place. If you want to pass a hash full of data to a function to modify it, creating and returning a new hash for each change could be tedious and inefficient, at least without some amazing compiler magic.

Perl provides a mechanism by which to refer to a value without making a copy. Any changes made to that *reference* will update the value in place, such that *all* references to that value will refer to the modified value. A reference is a first-class scalar data type which refers to another first-class data type.

Scalar References

The reference operator is the backslash (\). In scalar context, it creates a single reference which refers to another value. In list context, it creates a list of references. To take a reference to `$name`:

```
my $name     = 'Larry';
my $name_ref = \$name;
```

You must *dereference* a reference to evaluate the value to which it refers. Dereferencing requires you to add an extra sigil for each level of dereferencing:

```
sub reverse_in_place {
    my $name_ref = shift;
    $$name_ref   = reverse $$name_ref;
}

my $name = 'Blabby';
reverse_in_place( \$name );
say $name;
```

The double scalar sigil (`$$`) dereferences a scalar reference.

Parameters in `@_` behave as *aliases* to caller variables (Iteration and Aliasing, pp. 30), so you can modify them in place:

```
sub reverse_value_in_place {
    $_[0] = reverse $_[0];
}

my $name = 'allizocohC';
reverse_value_in_place( $name );
say $name;
```

You usually don't want to modify values this way–callers rarely expect it, for example. Assigning parameters to lexicals within your functions makes copies of the values in `@_` and avoids this aliasing behavior.

Saving Memory with References

Modifying a value in place or returning a reference to a scalar can save memory. Because Perl copies values on assignment, you could end up with multiple copies of a large string. Passing around references means that Perl will only copy the references–a far cheaper operation. Before you modify your code to pass only references, however, measure to see if this will make a difference.

Complex references may require a curly-brace block to disambiguate portions of the expression. You may *always* use this syntax, though sometimes it clarifies and other times it obscures:

```
sub reverse_in_place {
    my $name_ref   = shift;
    ${ $name_ref } = reverse ${ $name_ref };
}
```

If you forget to dereference a scalar reference, Perl will likely coerce the reference into a string value of the form `SCALAR(0x93339e8)` or a numeric value such as `0x93339e8`. This value indicates the type of reference (in this case, `SCALAR`) and the location in memory of the reference (because that's an unambiguous design choice, not because you can do anything with the memory location itself).

References Aren't Pointers

Perl does not offer native access to memory locations. The address of the reference is a value used as an identifier. Unlike pointers in a language such as C, you cannot modify the address of a reference or treat it as an address into memory. These addresses are *mostly* unique because Perl may reuse storage locations as it reclaims unused memory.

Array References

Array references are useful in several circumstances:

- To pass and return arrays from functions without list flattening
- To create multi-dimensional data structures
- To avoid unnecessary array copying

- To hold anonymous data structures

Use the reference operator to create a reference to a declared array:

```
my @cards     = qw( K Q J 10 9 8 7 6 5 4 3 2 A );
my $cards_ref = \@cards;
```

Any modifications made through `$cards_ref` will modify `@cards` and vice versa. You may access the entire array as a whole with the `@` sigil, whether to flatten the array into a list (list context) or count its elements (scalar context):

```
my $card_count = @$cards_ref;
my @card_copy  = @$cards_ref;
```

Access individual elements with the dereferencing arrow (`->`):

```
my $first_card = $cards_ref->[0];
my $last_card  = $cards_ref->[-1];
```

The arrow is necessary to distinguish between a scalar named `$cards_ref` and an array named `@cards_ref`. Note the use of the scalar sigil (Variable Sigils, pp. 15) to access a single element.

Doubling Sigils

An alternate syntax prepends another scalar sigil to the array reference. It's shorter but uglier to write `my $first_card = $$cards_ref[0];`.

Use the curly-brace dereferencing syntax to slice (Array Slices, pp. 41) an array reference:

```
my @high_cards = @{ $cards_ref }[0 .. 2, -1];
```

You *may* omit the curly braces, but their grouping often improves readability.

To create an anonymous array, surround a list-producing expression with square brackets:

```
my $suits_ref = [qw( Monkeys Robots Dinos Cheese )];
```

This array reference behaves the same as named array references, except that the anonymous array brackets *always* create a new reference. Taking a reference to a named array in its scope always refers to the *same* array. For example:

```
my @meals      = qw( soup sandwiches pizza );
my $sunday_ref = \@meals;
my $monday_ref = \@meals;

push @meals, 'ice cream sundae';
```

...both `$sunday_ref` and `$monday_ref` now contain a dessert, while:

```
my @meals      = qw( soup sandwiches pizza );
my $sunday_ref = [ @meals ];
my $monday_ref = [ @meals ];

push @meals, 'berry pie';
```

...neither `$sunday_ref` nor `$monday_ref` contains a dessert. Within the square braces used to create the anonymous array, list context flattens the `@meals` array into a list unconnected to `@meals`.

Hash References

Use the reference operator on a named hash to create a *hash reference*:

```
my %colors = (
    blue   => 'azul',
    gold   => 'dorado',
    red    => 'rojo',
    yellow => 'amarillo',
    purple => 'morado',
);

my $colors_ref = \%colors;
```

Access the keys or values of the hash by prepending the reference with the hash sigil %:

```
my @english_colors = keys   %$colors_ref;
my @spanish_colors = values %$colors_ref;
```

Access individual values of the hash (to store, delete, check the existence of, or retrieve) by using the dereferencing arrow or double scalar sigils:

```
sub translate_to_spanish {
    my $color = shift;
    return $colors_ref->{$color};
    # or return $$colors_ref{$color};
}
```

Use the array sigil (@) and disambiguation braces to slice a hash reference:

```
my @colors  = qw( red blue green );
my @colores = @{ $colors_ref }{@colors};
```

Create anonymous hashes in place with curly braces:

```
my $food_ref = {
    'birthday cake' => 'la torta de cumpleaños',
    candy           => 'dulces',
    cupcake         => 'bizcochito',
    'ice cream'     => 'helado',
};
```

As with anonymous arrays, anonymous hashes create a new anonymous hash on every execution.

Watch Those Braces!

The common novice error of assigning an anonymous hash to a standard hash produces a warning about an odd number of elements in the hash. Use parentheses for a named hash and curly brackets for an anonymous hash.

Function References

Perl supports *first-class functions*; a function is a data type just as is an array or hash. In other words, Perl supports *function references*. This enables many advanced features (Closures, pp. 84). Create a function reference by using the reference operator and the function sigil (`&`) on the name of a function:

```
sub bake_cake { say 'Baking a wonderful cake!' };

my $cake_ref = \&bake_cake;
```

Without the *function sigil* (`&`), you will take a reference to the function's return value or values.

Create anonymous functions with the bare `sub` keyword:

```
my $pie_ref = sub { say 'Making a delicious pie!' };
```

The use of the `sub` builtin *without* a name compiles the function but does not register it with the current namespace. The only way to access this function is via the reference returned from `sub`. Invoke the function reference with the dereferencing arrow:

```
$cake_ref->();
$pie_ref->();
```

Perl 4 Function Calls

An alternate invocation syntax for function references uses the function sigil (`&`) instead of the dereferencing arrow. Avoid this `&$cupcake_ref` syntax; it has subtle implications for parsing and argument passing.

Think of the empty parentheses as denoting an invocation dereferencing operation in the same way that square brackets indicate an indexed (array) lookup and curly brackets a keyed (hash) lookup. Pass arguments to the function within the parentheses:

```
$bake_something_ref->( 'cupcakes' );
```

You may also use function references as methods with objects (Moose, pp. 107). This is useful when you've already looked up the method (Reflection, pp. 121):

```
my $clean = $robot_maid->can( 'cleanup' );
$robot_maid->$clean( $kitchen );
```

Filehandle References

The lexical filehandle form of `open` and `opendir` operate on filehandle references. Internally, these filehandles are objects of the class `IO::File`. You can call methods on them directly:

```
use autodie 'open';

open my $out_fh, '>', 'output_file.txt';
$out_fh->say( 'Have some text!' );
```

Old code might `use IO::Handle;`. Older code may take references to typeglobs:

```
local *FH;
open FH, "> $file" or die "Can't write '$file': $!";
my $fh = \*FH;
```

This idiom predates the lexical filehandles introduced by Perl 5.6 in March 2000. You may still use the reference operator on typeglobs to take references to package-global filehandles such as `STDIN`, `STDOUT`, `STDERR`, or `DATA`–but these are all global names anyhow.

As lexical filehandles respect explicit scoping, they allow you to manage the lifespan of filehandles as a feature of Perl's memory management.

Reference Counts

Perl uses a memory management technique known as *reference counting*. Every Perl value has a counter attached to it, internally. Perl increases this counter every time something takes a reference to the value, whether implicitly or explicitly. Perl decreases that counter every time a reference goes away. When the counter reaches zero, Perl can safely recycle that value. Consider the filehandle opened in this inner scope:

```
say 'file not open';

{
    open my $fh, '>', 'inner_scope.txt';
    $fh->say( 'file open here' );
}

say 'file closed here';
```

Within the inner block in the example, there's one `$fh`. (Multiple lines in the source code mention it, but there's only one variable, the one named `$fh`.) `$fh` is only in scope in the block. Its value never leaves the block. When execution reaches the end of the block, Perl recycles the variable `$fh` and decreases the reference count of the filehandle referred to by `$fh`. The filehandle's reference count reaches zero, so Perl destroys the filehandle to reclaim memory. As part of the process, it calls `close()` on the filehandle implicitly.

You don't have to understand the details of how all of this works. You only need to understand that your actions in taking references and passing them around affect how Perl manages memory (see Circular References, pp. 61).

References and Functions

When you use references as arguments to functions, document your intent carefully. Modifying the values of a reference from within a function may surprise the calling code, which never expected anything else to modify its data. To modify the contents of a reference without affecting the reference itself, copy its values to a new variable:

```
my @new_array = @{ $array_ref };
my %new_hash  = %{ $hash_ref  };
```

This is only necessary in a few cases, but explicit cloning helps avoid nasty surprises for the calling code. If you use nested data structures or other complex references, consider the use of the core module `Storable` and its `dclone` (*deep cloning*) function.

Nested Data Structures

Perl's aggregate data types–arrays and hashes–store scalars indexed by integer or string keys. Note the word scalar. If you try to store an array in an array, Perl's automatic list flattening will make everything into a single array:

```
my @counts = qw( eenie miney moe      );
my @ducks  = qw( huey  dewey louie    );
my @game   = qw( duck  duck  grayduck );

my @famous_triplets = (
    @counts, @ducks, @game
);
```

Perl's solution to this is references (References, pp. 53), which are special scalars that can refer to other variables (scalars, arrays, and hashes). You Nested data structures in Perl, such as an array of arrays or a hash of hashes, are possible through the use of references. Unfortunately, the reference syntax can be a little bit ugly.

Use the reference operator, \, to produce a reference to a named variable:

```
my @famous_triplets = (
    \@counts, \@ducks, \@game
);
```

...or the anonymous reference declaration syntax to avoid the use of named variables:

```
my @famous_triplets = (
    [qw( eenie miney moe   )],
    [qw( huey  dewey louie )],
    [qw( duck  duck  goose )],
);

my %meals = (
    breakfast => { entree => 'eggs',
                   side   => 'hash browns'   },
    lunch     => { entree => 'panini',
                   side   => 'apple'         },
    dinner    => { entree => 'steak',
                   side   => 'avocado salad' },
);
```

Commas are Free

Perl allows an optional trailing comma after the last element of a list. This makes it easy to add more elements in the future.

Use Perl's reference syntax to access elements in nested data structures. The sigil denotes the amount of data to retrieve. The dereferencing arrow indicates that the value of one portion of the data structure is a reference:

```
my $last_nephew = $famous_triplets[1]->[2];
my $meal_side   = $meals{breakfast}->{side};
```

The only way to access elements in a nested data structure is through references, so the arrow in the previous examples is superfluous. You may omit it for clarity, except for invoking function references:

```
my $nephew = $famous_triplets[1][2];
my $meal   = $meals{breakfast}{side};

$actions{generous}{buy_food}->( $nephew, $meal );
```

Use disambiguation blocks to access components of nested data structures as if they were first-class arrays or hashes:

```
my $nephew_count   = @{ $famous_triplets[1] };
my $dinner_courses = keys %{ $meals{dinner} };
```

...or to slice a nested data structure:

```
my ($entree, $side) = @{ $meals{breakfast} }{ qw( entree side ) };
```

Whitespace helps, but does not entirely eliminate the noise of this construct. Sometimes a temporary variable provides more clarity:

```
my $meal_ref        = $meals{breakfast};
my ($entree, $side) = @$meal_ref{qw( entree side )};
```

...or use `for`'s implicit aliasing to avoid the use of an intermediate reference (though note the lack of `my`):

```
($entree, $side) = @{ $_ }{qw( entree side )} for $meals{breakfast};
```

`perldoc perldsc`, the data structures cookbook, gives copious examples of how to use Perl's various data structures.

Autovivification

When you attempt to write to a component of a nested data structure, Perl will create the path through the data structure to the destination as necessary:

```
my @aoaoaoa;
$aoaoaoa[0][0][0][0] = 'nested deeply';
```

After the second line of code, this array of arrays of arrays of arrays contains an array reference in an array reference in an array reference in an array reference. Each array reference contains one element.

Similarly, when you ask Perl to treat an undefined value as if it were a hash reference, Perl will turn that undefined value into a hash reference:

```
my %hohoh;
$hohoh{Robot}{Santa} = 'mostly harmful';
```

This behavior is *autovivification*. While it reduces the initialization code of nested data structures, it cannot distinguish between the honest intent to create missing elements in nested data structures or a typo.

You may wonder at the contradiction between taking advantage of autovivification while enabling `strict`ures. Is it more convenient to catch errors which change the behavior of your program at the expense of disabling error checks for a few well-encapsulated symbolic references? Is it more convenient to allow data structures to grow or safer to require a fixed size and an allowed set of keys?

The answers depend on your project. During early development, allow yourself the freedom to experiment. While testing and deploying, consider an increase of strictness to prevent unwanted side effects. The `autovivification` pragma (Pragmas, pp. 129) from the CPAN lets you disable autovivification in a lexical scope for specific types of operations. Combined with the `strict` pragma, you can enable these behaviors where and as necessary.

You *can* verify your expectations before dereferencing each level of a complex data structure, but the resulting code is often lengthy and tedious. It's better to avoid deeply nested data structures by revising your data model to provide better encapsulation.

Debugging Nested Data Structures

The complexity of Perl's dereferencing syntax combined with the potential for confusion with multiple levels of references can make debugging nested data structures difficult. The core module `Data::Dumper` converts values of arbitrary complexity into strings of Perl code, which helps visualize what you have:

```
use Data::Dumper;

my $complex_structure = {
    numbers => [ 1 .. 3 ];
```

```
    letters => [ 'a' .. 'c' ],
    objects => {
        breakfast => $continental,
        lunch     => $late_tea,
        dinner    => $banquet,
    },
};

print Dumper( $my_complex_structure );
```

. . . which might produce something like:

```
  $VAR1 = {
      'numbers' => [
                     1,
                     2,
                     3
                   ],
      'letters' => [
                     'a',
                     'b',
                     'c'
                   ],
      'meals' => {
          'dinner' => bless({...}, 'Dinner'),
          'lunch' => bless({...}, 'Lunch'),
          'breakfast' => bless({...}, 'Breakfast'),
      },
  };
```

Use this when you need to figure out what a data structure contains, what you should access, and what you accessed instead. As the elided example alludes, `Data::Dumper` can dump objects as well as function references (if you set `$Data::Dumper::Deparse` to a true value).

While `Data::Dumper` is a core module and prints Perl code, its output is verbose. Some developers prefer the use of the `YAML::XS` or `JSON` modules for debugging. They do not produce Perl code, but their outputs can be much clearer to read and to understand.

Circular References

Perl's memory management system of reference counting (Reference Counts, pp. 58) has one drawback. Two references which point to each other, whether directly or indirectly, form a *circular reference* that Perl cannot destroy on its own. Consider a biological model, where each entity has two parents and zero or more children:

```
my $alice  = { mother => '',     father => ''     };
my $robin  = { mother => '',     father => ''     };
my $cianne = { mother => $alice, father => $robin };

push @{ $alice->{children} }, $cianne;
push @{ $robin->{children} }, $cianne;
```

Both `$alice` and `$robin` contain an array reference which contains `$cianne`. Because `$cianne` is a hash reference which contains `$alice` and `$robin`, Perl will never decrease the reference count of any of these three people to zero. It doesn't recognize that these circular references exist, and it can't manage the lifespan of these entities.

Either break the reference count manually yourself (by clearing the children of `$alice` and `$robin` or the parents of `$cianne`), or use *weak references*. A weak reference does not increase the reference count of its referent. Use the core module `Scalar::Util`'s `weaken()` function to weaken a reference:

```
use Scalar::Util 'weaken';

my $alice  = { mother => '',      father => ''     };
my $robin  = { mother => '',      father => ''     };
my $cianne = { mother => $alice, father => $robin };

push @{ $alice->{children} }, $cianne;
push @{ $robin->{children} }, $cianne;

weaken( $cianne->{mother} );
weaken( $cianne->{father} );
```

`$cianne` will retain usable references to `$alice` and `$robin`, but those weak references do not count toward the number of remaining references to the parents. If the reference count of `$alice` reaches zero, Perl's garbage collector will reclaim her record, even though `$cianne` has a weak reference to `$alice`. When `$alice` gets reclaimed, `$cianne`'s reference to `$alice` will be set to `undef`.

Most data structures do not need weak references, but when they're necessary, they're invaluable.

Alternatives to Nested Data Structures

While Perl is content to process data structures nested as deeply as you can imagine, the human cost of understanding these data structures and their relationships–to say nothing of the complex syntax–is high. Beyond two or three levels of nesting, consider whether modeling various components of your system with classes and objects (Moose, pp. 107) will allow for clearer code.

CHAPTER 4

Operators

Some people call Perl an “operator-oriented language”. A Perl *operator* is a series of one or more symbols used as part of the syntax of a language. Each operator operates on zero or more *operands*. Think of an operator as a special sort of function the parser understands and its operands as arguments.

You've seen how Perl manages context through its operators. To understand Perl fully, you must understand how operators interact with their operands.

Operator Characteristics

Every operator possesses several important characteristics which govern its behavior: the number of operands on which it operates, its relationship to other operators, the contexts it enforces, and the syntax it provides.

`perldoc perlop` and `perldoc perlsyn` provide voluminous information about Perl's operators, but the docs assume you're already familiar with a few essential computer science concepts. Fortunately, you'll recognize these ideas from written language and elementary mathematics, even if you've never heard their complicated names before.

Precedence

The *precedence* of an operator governs when Perl should evaluate it in an expression. Perl evaluates the operator with the highest precedence first, then the next highest, all the way to the lowest precedence. Remember basic math? Multiply and divide before you add and subtract. That's precedence. Because the precedence of multiplication is higher than the precedence of addition, in Perl `7 + 7 * 10` evaluates to `77`, not `140`.

Use grouping parentheses to force the evaluation of some operators before others. In `(7 + 7) * 10`, grouping the addition into a single unit forces its evaluation before the multiplication–though Perl wants to perform the multiplication first, it has to evaluate the grouped subexpression into a single value as the multiplication operator's left operand. The result is `140`.

`perldoc perlop` contains a table of precedence. Skim it a few times, but don't bother memorizing it (almost no one does). Spend your time simplifying your code where you can. Then add parentheses where they clarify.

In cases where two operators have the same precedence, other factors such as associativity (Associativity, pp. 63) and fixity (Fixity, pp. 64) break the tie.

Associativity

The *associativity* of an operator governs whether it evaluates from left to right or right to left. Addition is left associative, such that `2 + 3 + 4` evaluates `2 + 3` first, then adds `4` to the result, not that order of evaluation matters. Exponentiation is right associative, such that `2 ** 3 ** 4` evaluates `3 ** 4` first, then raises `2` to the 81st power. Use parentheses if you write code like this.

If you memorize only the precedence and associativity of the common mathematical operators, you'll be fine. Simplify your code and you won't have to memorize other associativities. If you can't simplify your code (or if you're maintaining code and trying to understand it), use the core `B::Deparse` module to see exactly how Perl handles operator precedence and associativity.

Run `perl -MO=Deparse,-p` on a snippet of code The `-p` flag adds extra grouping parentheses which often clarify evaluation order. Beware that Perl's optimizer will simplify mathematical operations using constant values. If you really need to deparse a complex expression, use named variables instead of constant values, as in `$x ** $y ** $z`.

Arity

The *arity* of an operator is the number of operands on which it operates. A *nullary* operator operates on zero operands. A *unary* operator operates on one operand. A *binary* operator operates on two operands. A *trinary* operator operates on three operands. A *listary* operator operates on a list of zero or more operands.

Arithmetic operators are binary operators and are usually left associative. This has implications for tie-breaking evaluation order of operands with the same precedence. For example, `2 + 3 - 4` evaluates `2 + 3` first. Addition and subtraction have the same precedence, but they're left associative and binary, so the proper evaluation order applies the leftmost operator (`+`) to the leftmost two operands (`2` and `3`) with the leftmost operator (`+`), then applies the rightmost operator (`-`) to the result of the first operation and the rightmost operand (`4`).

Fixity

Perl novices often find confusion between the interaction of listary operators–especially function calls–and nested expressions. Where parentheses usually help, beware of the parsing complexity of:

```
 probably buggy code
ay ( 1 + 2 + 3 ) * 4;
```

...which prints the value `6` and (probably) evaluates as a whole to `4` (the return value of `say` multiplied by `4`). Perl's parser happily interprets the parentheses as postcircumfix operators denoting the arguments to `say`, not circumfix parentheses grouping an expression to change precedence.

An operator's *fixity* is its position relative to its operands:

- *Infix* operators appear between their operands. Most mathematical operators are infix operators, such as the multiplication operator in `$length * $width`.
- *Prefix* operators precede their operands. *Postfix* operators follow their operands. These operators tend to be unary, such as mathematic negation (`-$x`), boolean negation (`!$y`), and postfix increment (`$z++`).
- *Circumfix* operators surround their operands, as with the anonymous hash constructor (`{ ... }`) and quoting operators (`qq[ ... ]`).
- *Postcircumfix* operators follow certain operands and surround others, as seen in hash and array element access (`$hash{$x}` and `$array[$y]`).

Operator Types

Perl's operators provide value contexts (Numeric, String, and Boolean Context, pp. 5) to their operands. To choose the appropriate operator, you must know the values of the operands you provide as well as the value you expect to receive.

Numeric Operators

Numeric operators impose numeric contexts on their operands. These operators are the standard arithmetic operators of addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), exponentiation (`**`), and modulo (`%`), their in-place variants (`+=`, `-=`, `*=`, `/=`, `**=`, and `%=`), and both postfix and prefix auto-decrement (`--`).

The auto-increment operator has special string behavior (Special Operators, pp. 65).

Several comparison operators impose numeric contexts upon their operands. These are numeric equality (`==`), numeric inequality (`!=`), greater than (`>`), less than (`<`), greater than or equal to (`>=`), less than or equal to (`<=`), and the sort comparison operator (`<=>`).

String Operators

String operators impose string contexts on their operands. These operators are positive and negative regular expression binding (`=~` and `!~`, respectively), and concatenation (`.`).

Several comparison operators impose string contexts upon their operands. These are string equality (`eq`), string inequality (`ne`), greater than (`gt`), less than (`lt`), greater than or equal to (`ge`), less than or equal to (`le`), and the string sort comparison operator (`cmp`).

Logical Operators

Logical operators impose a boolean context on their operands. These operators are `&&`, `and`, `||`, and `or`. These infix operators all exhibit *short-circuiting* behavior (Short Circuiting, pp. 28). The word forms have lower precedence than the punctuation forms.

The defined-or operator, `//`, tests the *definedness* of its operand. Unlike `||` which tests the *truth* of its operand, `//` evaluates to a true value even if its operand evaluates to a numeric zero or the empty string. This is especially useful for setting default parameter values:

```
sub name_pet {
    my $name = shift // 'Fluffy';
    ...
}
```

The ternary conditional operator (`?:`) takes three operands. It evaluates the first in boolean context and evaluates to the second if the first is true and the third otherwise:

```
my $truthiness = $value ? 'true' : 'false';
```

The prefix `!` and `not` operators return the logical opposites of the boolean values of their operands. `not` is a lower precedence version of `!`.

The `xor` operator is an infix operator which evaluates to the exclusive-or of its operands.

Bitwise Operators

Bitwise operators treat their operands numerically at the bit level. These uncommon operators are left shift (`<<`), right shift (`>>`), bitwise and (`&`), bitwise or (`|`), and bitwise xor (`^`), as well as their in-place variants (`<<=`, `>>=`, `&=`, `|=`, and `^=`).

Special Operators

The auto-increment operator has special behavior. When used on a value with a numeric component (Cached Coercions, pp. 50), the operator increments that numeric component. If the value is obviously a string (if it has no numeric component), the operator increments the value's string component such that `a` becomes `b`, `zz` becomes `aaa`, and `a9` becomes `b0`.

```
my $num = 1;
my $str = 'a';

$num++;
$str++;
is $num,   2, 'numeric autoincrement';
is $str, 'b', 'string autoincrement';

no warnings 'numeric';
$num += $str;
$str++;

is $num, 2, 'numeric addition with $str';
is $str, 1, '... gives $str a numeric part';
```

The repetition operator (`x`) is an infix operator with complex behavior. When evaluated in list context *with a list as its first operand*, it evaluates to that list repeated the number of times specified by its second operand. When evaluated in list context *with a scalar as its first operand*, it produces a string consisting of the string value of its first operand concatenated to itself the number of times specified by its second operand.

In scalar context, the operator repeats and concatenates a string:

```
my @scheherazade = ('nights') x 1001;
my $calendar     =  'nights'  x 1001;
my $cal_length   =  length $calendar;

is @scheherazade, 1001, 'list repeated';
is $cal_length,   1001 * length 'nights', 'word repeated';

my @schenolist   =  'nights'  x 1001;
my $calscalar    = ('nights') x 1001;

is @schenolist, 1, 'no lvalue list';
is length $calscalar, 1001 * length 'nights', 'word still repeated';
```

The infix *range* operator (`..`) produces a list of items in list context:

```
my @cards = ( 2 .. 10, 'J', 'Q', 'K', 'A' );
```

It can *only* produce simple, incrementing ranges of integers or strings.

In boolean context, the range operator performs a *flip-flop* operation. This operator produces a false value until its left operand is true. That value stays true until the right operand is true, after which the value is false again until the left operand is true again. Imagine parsing the text of a formal letter with:

```
while (/Hello, $user/ .. /Sincerely,/) {
    say "> $_";
}
```

The *comma* operator (`,`) is an infix operator. In scalar context it evaluates its left operand then returns the value produced by evaluating its right operand. In list context, it evaluates both operands in left-to-right order.

The fat comma operator (`=>`) also quotes any bareword used as its left operand (Hashes, pp. 43).

The *triple-dot* or *whatever* operator stands in for a single statement. It is nullary and has neither precedence nor associativity. It parses, but when executed it throws an exception with the string `Unimplemented`. This makes a great placeholder in example code you don't expect anyone to execute:

```
sub some_example {
    # implement this yourself
    ...
}
```

CHAPTER 5

Functions

A *function* (or *subroutine*) in Perl is a discrete, encapsulated unit of behavior. A program is a collection of little black boxes where the interaction of these functions governs the control flow of the program. A function may have a name. It may consume incoming information. It may produce outgoing information.

Functions are a prime mechanism for organizing code into similar groups, identifying individual pieces by name, and providing reusable units of behavior.

Declaring Functions

Use the `sub` builtin to declare a function:

```
sub greet_me  { ... }
```

Now you can invoke `greet_me()` from anywhere within your program.

Just as you may *declare* a lexical variable but leave its value undefined, you may declare a function without defining it. A *forward declaration* tells Perl to record that a named function exists. You may define it later:

```
sub greet_sun;
```

Invoking Functions

Use postfix (Fixity, pp. 64) parentheses to invoke a named function. Any arguments to the function may go within the parentheses:

```
greet_me( 'Jack', 'Tuxie', 'Brad' );
greet_me( 'Snowy' );
greet_me();
```

While these parentheses are not strictly necessary for these examples–even with `strict` enabled–they provide clarity to human readers as well as Perl's parser. When in doubt, use them.

Function arguments can be arbitrary expressions–including variables and function calls:

```
greet_me( $name );
greet_me( @authors );
greet_me( %editors );
greet_me( get_readers() );
```

. . . though Perl's default parameter handling sometimes surprises novices.

Function Parameters

A function receives its parameters in a single array, `@_` (The Default Array Variables, pp. 7). When you invoke a function, Perl *flattens* all provided arguments into a single list. The function must either unpack its parameters into variables or operate on `@_` directly:

```
sub greet_one {
    my ($name) = @_;
    say "Hello, $name!";
}

sub greet_all {
    say "Hello, $_!" for @_;
}
```

`@_` behaves as a normal array. Most Perl functions `shift` off parameters or use list assignment, but you may also access individual elements by index:

```
sub greet_one_shift {
    my $name = shift;
    say "Hello, $name!";
}

sub greet_two_list_assignment {
    my ($hero, $sidekick) = @_;
    say "Well if it isn't $hero and $sidekick. Welcome!";
}

sub greet_one_indexed {
    my $name = $_[0];
    say "Hello, $name!";

    # or, less clear
    say "Hello, $_[0]!";
}
```

You may also `unshift`, `push`, `pop`, `splice`, and use slices of `@_`. Remember that the array builtins use `@_` as the default operand *within functions*, so that `my $name = shift;` works. Take advantage of this idiom.

To access a single scalar parameter from `@_`, use `shift`, an index of `@_`, or lvalue list context parentheses. Otherwise, Perl will happily evaluate `@_` in scalar context for you and assign the number of parameters passed:

```
sub bad_greet_one {
    my $name = @_;  # buggy
    say "Hello, $name; you look numeric today!"
}
```

List assignment of multiple parameters is often clearer than multiple lines of `shift`. Compare:

```
my $left_value  = shift;
my $operation   = shift;
my $right_value = shift;
```

...to:

```
my ($left_value, $operation, $right_value) = @_;
```

The latter is simpler to read. As a side benefit, it has better runtime performance, though you're unlikely to notice.

Occasionally you may see code which extracts parameters from `@_` and passes the rest to another function:

```
sub delegated_method {
    my $self = shift;
    say 'Calling delegated_method()'

    $self->delegate->delegated_method( @_ );
}
```

Use `shift` when your function needs only a single parameter. Use list assignment when accessing multiple parameters.

Real Function Signatures

Perl 5.20 added built-in function signatures as an experimental feature. "Experimental" means that they may change or even go away in future releases of Perl, so you need to enable them to signal that you accept the possibility of rewriting code.

```
use experimental 'signatures';
```

With that disclaimer in place, you can now write:

```
sub greet_one($name) {
    say "Hello, $name!";
}
```

. . . which is equivalent to writing:

```
sub greet_one {
    die "Too many arguments for subroutine" if @_ < 1;
    die "Too few arguments for subroutine" if @_ > 1;
    my $name = shift;
    say "Hello, $name!";
}
```

You can make `$name` an optional variable by assigning it a default value:

```
sub greet_one($name = 'Bruce') {
    say "Hello, $name!";
}
```

. . . in which case writing `greet_one( 'Bruce' )` and `greet_one()` will both ignore Batman's crime-fighting identity.

You may use aggregate arguments at the end of a signature:

```
sub greet_all($leader, @everyone) {
    say "Hello, $leader!";
    say "Hi also, $_." for @everyone;
}

sub make_nested_hash($name, %pairs) {
    return { $name => \%pairs };
}
```

. . . or indicate that a function expects *no* arguments:

```
sub no_gifts_please() {
    say 'I have too much stuff already.'
}
```

. . . which means that you'll get the `Too many arguments for subroutine` exception by calling that function with arguments.

These experimental signatures have more features than discussed here. As you get beyond basic positional parameters, the possibility of incompatible changes in future versions of Perl increases, however. See `perldoc perlsub`'s "Signatures" section for more details, especially in newer versions of Perl.

Signatures aren't your only options. Several CPAN distributions extend Perl's parameter handling with additional syntax and options. `Method::Signatures` works as far back as Perl 5.8. `Kavorka` works with Perl 5.14 and newer.

Should You Use Signatures?

Despite the experimental nature of function signatures–or the additional dependencies of the CPAN modules–all of these options can make your code a little shorter and a little clearer both to read and to write. By all means experiment with these options to find out what works best for you and your team. Even sticking with simple positional parameters can improve your work.

Flattening

List flattening into `@_` happens on the caller side of a function call. Passing a hash as an argument produces a list of key/value pairs:

```
my %pet_names_and_types = (
    Lucky   => 'dog',
    Rodney  => 'dog',
    Tuxedo  => 'cat',
    Petunia => 'cat',
    Rosie   => 'dog',
);

show_pets( %pet_names_and_types );

sub show_pets {
    my %pets = @_;

    while (my ($name, $type) = each %pets) {
        say "$name is a $type";
    }
}
```

When Perl flattens `%pet_names_and_types` into a list, the order of the key/value pairs from the hash will vary, but the list will always contain a key immediately followed by its value. Hash assignment inside `show_pets()` works the same way as the explicit assignment to `%pet_names_and_types`.

This flattening is often useful, but beware of mixing scalars with flattened aggregates in parameter lists. To write a `show_pets_of_type()` function, where one parameter is the type of pet to display, pass that type as the *first* parameter (or use `pop` to remove it from the end of `@_`, if you like to confuse people):

```
sub show_pets_by_type {
    my ($type, %pets) = @_;

    while (my ($name, $species) = each %pets) {
        next unless $species eq $type;
        say "$name is a $species";
    }
}

my %pet_names_and_types = (
    Lucky   => 'dog',
    Rodney  => 'dog',
    Tuxedo  => 'cat',
    Petunia => 'cat',
    Rosie   => 'dog',
);

show_pets_by_type( 'dog',   %pet_names_and_types );
show_pets_by_type( 'cat',   %pet_names_and_types );
show_pets_by_type( 'moose', %pet_names_and_types );
```

With experimental function signatures, you could write:

```
sub show_pets_by_type($type, %pets) {
    ...
}
```

Slurping

List assignment with an aggregate is always greedy, so assigning to `%pets` *slurps* all of the remaining values from `@_`. If the `$type` parameter came at the end of `@_`, Perl would warn about assigning an odd number of elements to the hash. You *could* work around that:

```
sub show_pets_by_type {
    my $type = pop;
    my %pets = @_;

    ...
}
```

. . . at the expense of clarity. The same principle applies when assigning to an array as a parameter. Use references (References, pp. 53) to avoid unwanted aggregate flattening.

Aliasing

`@_` contains a subtlety; it *aliases* function arguments. In other words, if you access `@_` directly, you can modify the arguments passed to the function:

```
sub modify_name {
    $_[0] = reverse $_[0];
}

my $name = 'Orange';
modify_name( $name );
```

```
say $name;

# prints egnar0
```

Modify an element of `@_` directly and you will modify the original argument. Be cautious and unpack `@_` rigorously–or document the modification carefully.

Functions and Namespaces

Every function has a containing namespace (Packages, pp. 51). Functions in an undeclared namespace–functions not declared within the scope of an explicit `package` statement–exist in the `main` namespace. You may also declare a function within another namespace by prefixing its name:

```
sub Extensions::Math::add { ... }
```

This will create the namespace as necessary and then declare the function within it. Remember that Perl packages are open for modification at any point–even while your program is running. Perl will issue a warning if you declare multiple functions with the same name in a single namespace.

Refer to other functions within the same namespace with their short names. Use a fully-qualified name to invoke a function in another namespace:

```
package main;

Extensions::Math::add( $scalar, $vector );
```

Remember, functions are *visible* outside of their own namespaces through their fully-qualified names. Alternately, you may import names from other namespaces.

Lexical Functions

Perl 5.18 added an experimental feature to declare functions lexically. They're visible only within lexical scopes after declaration. See the "Lexical Subroutines" section of `perldoc perlsub` for more details.

Importing

When loading a module with the `use` builtin (Modules, pp. 143), Perl automatically calls a method named `import()`. Modules can provide their own `import()` method which makes some or all defined symbols available to the calling package. Any arguments after the name of the module in the `use` statement get passed to the module's `import()` method. Thus:

```
use strict;
```

...loads the *strict.pm* module and calls `strict->import()` with no arguments, while:

```
use strict 'refs';
use strict qw( subs vars );
```

...loads the *strict.pm* module, calls `strict->import( 'refs' )`, then calls `strict->import( 'subs', vars' )`.

`use` has special behavior with regard to `import()`, but you may call `import()` directly. The `use` example is equivalent to:

```
BEGIN {
    require strict;
    strict->import( 'refs' );
    strict->import( qw( subs vars ) );
}
```

The `use` builtin adds an implicit `BEGIN` block around these statements so that the `import()` call happens *immediately* after the parser has compiled the entire `use` statement. This ensures that the parser knows about any symbols imported by `strict` before it compiles the rest of the program. Otherwise, any functions *imported* from other modules but not *declared* in the current file would look like barewords, and would violate `strict`, for example.

Of course, `strict` is a pragma (Pragmas, pp. 129), so it has other effects.

Reporting Errors

Use the `caller` builtin to inspect a function's calling context. When passed no arguments, `caller` returns a list containing the name of the calling package, the name of the file containing the call, and the line number of the file on which the call occurred:

```
package main;

main();

sub main {
    show_call_information();
}

sub show_call_information {
    my ($package, $file, $line) = caller();
    say "Called from $package in $file:$line";
}
```

The full call chain is available for inspection. Pass a single integer argument *n* to `caller()` to inspect the caller of the caller of the caller *n* times back. Within `show_call_information()`, `caller(0)` returns information about the call from `main()`. `caller(1)` returns information about the call from the start of the program.

This optional argument also tells `caller` to provide additional return values, including the name of the function and the context of the call:

```
sub show_call_information {
    my ($package, $file, $line, $func) = caller(0);
    say "Called $func from $package in $file:$line";
}
```

The standard `Carp` module uses `caller` to enhance error and warning messages. When used in place of `die` in library code, `croak()` throws an exception from the point of view of its caller. `carp()` reports a warning from the file and line number of its caller (Producing Warnings, pp. 135).

Use `caller` (or `Carp`) when validating parameters or preconditions of a function to indicate that whatever called the function did so erroneously.

Validating Arguments

While Perl does its best to do what you mean, it offers few native ways to test the validity of arguments provided to a function. Evaluate `@_` in scalar context to check that the *number* of parameters passed to a function is correct:

```
sub add_numbers {
    croak 'Expected two numbers, received: ' . @_
        unless @_ == 2;

    ...
}
```

This validation reports any parameter count error from the point of view of its caller, thanks to the use of `croak`.

Type checking is more difficult, because of Perl's operator-oriented type conversions (Context, pp. 3). If you want additional safety of function parameters, see CPAN modules such as `Params::Validate`.

Advanced Functions

Functions are the foundation of many advanced Perl features.

Context Awareness

Perl's builtins know whether you've invoked them in void, scalar, or list context. So too can your functions. The `wantarray` builtin returns `undef` to signify void context, a false value to signify scalar context, and a true value to signify list context. Yes, it's misnamed; see `perldoc -f wantarray` for proof.

```
sub context_sensitive {
    my $context = wantarray();

    return qw( List context )   if         $context;
    say    'Void context'   unless defined $context;
    return 'Scalar context' unless         $context;
}

context_sensitive();
say my $scalar = context_sensitive();
say context_sensitive();
```

This can be useful for functions which might produce expensive return values to avoid doing so in void context. Some idiomatic functions return a list in list context and the first element of the list or an array reference in scalar context. However, there exists no single best recommendation for the use of `wantarray`. Sometimes it's clearer to write separate and unambiguous functions, such as `get_all_toppings()` and `get_next_topping()`.

Putting it in Context

Robin Houston's `Want` and Damian Conway's `Contextual::Return` distributions from the CPAN offer many possibilities for writing powerful context-aware interfaces.

Recursion

Suppose you want to find an element in a sorted array. You *could* iterate through every element of the array individually, looking for the target, but on average, you'll have to examine half of the elements of the array. Another approach is to halve the array, pick the element at the midpoint, compare, then repeat with either the lower or upper half. Divide and conquer. When you run out of elements to inspect or find the element, stop.

An automated test for this technique could be:

```
use Test::More;

my @elements = ( 1, 5, 6, 19, 48, 77, 997, 1025, 7777, 8192, 9999 );

ok   elem_exists(     1, @elements ), 'found first element in array';
ok   elem_exists(  9999, @elements ), 'found last element in array';
ok ! elem_exists(   998, @elements ), 'did not find element not in array';
ok ! elem_exists(    -1, @elements ), 'did not find element not in array';
ok ! elem_exists( 10000, @elements ), 'did not find element not in array';

ok   elem_exists(    77, @elements ), 'found midpoint element';
ok   elem_exists(    48, @elements ), 'found end of lower half element';
ok   elem_exists(   997, @elements ), 'found start of upper half element';

done_testing();
```

Recursion is a deceptively simple concept. Every call to a function in Perl creates a new *call frame*, an data structure internal to Perl itself which represents the fact that you've called a function. This call frame includes the lexical environment of the function's current invocation–the values of all lexical variables within the function as invoked. Because the storage of the values of the lexical variables is separate from the function itself, you can have multiple calls to a function active at the same time. A function can even call itself, or *recur*.

To make the previous test pass, write the recursive function `elem_exists()`:

```
sub elem_exists {
    my ($item, @array) = @_;

    # break recursion with no elements to search
    return unless @array;

    # bias down with odd number of elements
    my $midpoint = int( (@array / 2) - 0.5 );
    my $miditem  = $array[ $midpoint ];

    # return true if found
    return 1 if $item  == $miditem;

    # return false with only one element
    return   if @array == 1;

    # split the array down and recurse
    return elem_exists(
        $item, @array[0 .. $midpoint]
    ) if $item < $miditem;

    # split the array and recurse
    return elem_exists(
         $item, @array[ $midpoint + 1 .. $#array ]
    );
}
```

Keep in mind that the arguments to the function will be *different* for every call, otherwise the function would always behave the same way (it would continue recursing until the program crashes). That's why the termination condition is so important.

Every recursive program can be written without recursion, but this divide-and-conquer approach is an effective way to manage many similar types of problems. For more information about recursion, iteration, and advanced function use in Perl the free book *Higher Order Perl*[1] is an excellent reference.

Lexicals

Every invocation of a function creates its own *instance* of a lexical scope represented internally by a call frame. Even though the declaration of `elem_exists()` creates a single scope for the lexicals `$item`, `@array`, `$midpoint`, and `$miditem`, every *call* to `elem_exists()`–even recursively–stores the values of those lexicals separately.

Not only can `elem_exists()` call itself, but the lexical variables of each invocation are safe and separate:

```
use Carp 'cluck';

sub elem_exists {
    my ($item, @array) = @_;

    cluck "[$item] (@array)";
    ...
}
```

Tail Calls

One *drawback* of recursion is that it's easy to write a function which calls itself infinitely. `elem_exists()` function has several `return` statements for this reason. Perl offers a helpful `Deep recursion on subroutine` warning when it suspects runaway recursion. The limit of 100 recursive calls is arbitrary, but often useful. Disable this warning with `no warnings 'recursion'`.

Because each call to a function requires a new call frame and lexical storage space, highly-recursive code can use more memory than iterative code. *Tail call elimination* can help.

A *tail call* is a call to a function which directly returns that function's results. These recursive calls to `elem_exists()`:

```
# split the array down and recurse
return elem_exists(
    $item, @array[0 .. $midpoint]
) if $item < $miditem;

# split the array and recurse
return elem_exists(
     $item, @array[ $midpoint + 1 .. $#array ]
);
```

. . . are candidates for tail call elimination. This optimization would avoid returning to the current call and then returning to the parent call. Instead, it returns to the parent call directly.

Perl does not eliminate tail calls automatically, but you can get the same effect by using a special form of the `goto` builtin. Unlike the form which often produces spaghetti code, the `goto` function form replaces the current function call with a call to another function. You may use a function by name or by reference. Manipulate `@_` to modify the arguments passed to the replacement function:

```
# split the array down and recurse
if ($item < $miditem) {
    @_ = ($item, @array[0 .. $midpoint]);
    goto &elem_exists;
```

[1] http://hop.perl.plover.com/

```
}

# split the array up and recurse
else {
    @_ = ($item, @array[$midpoint + 1 .. $#array] );
    goto &elem_exists;
}
```

Sometimes optimizations are ugly, but if the alternative is highly recursive code which runs out of memory, embrace the ugly and rejoice in the practical.

Pitfalls and Misfeatures

Perl still supports old-style invocations of functions, carried over from ancient versions of Perl. Previous versions of Perl required you to invoke functions with a leading ampersand (`&`) character:

```
# outdated style; avoid
my $result = &calculate_result( 52 );

# very outdated; truly avoid
my $result = do &calculate_result( 42 );
```

While the vestigial syntax is visual clutter, the leading ampersand form has other surprising behaviors. First, it disables any prototype checking. Second, it *implicitly* passes the contents of `@_` unmodified, unless you've explicitly passed arguments yourself. That unfortunate behavior can be confusing invisible action at a distance.

A final pitfall comes from leaving the parentheses off of function calls. The Perl parser uses several heuristics to resolve ambiguous barewords and the number of parameters passed to a function. Heuristics can be wrong:

```
# warning; contains a subtle bug
ok elem_exists 1, @elements, 'found first element';
```

The call to `elem_exists()` will gobble up the test description intended as the second argument to `ok()`. Because `elem_exists()` uses a slurpy second parameter, this may go unnoticed until Perl produces warnings about comparing a non-number (the test description, which it cannot convert into a number) with the element in the array.

While extraneous parentheses can hamper readability, thoughtful use of parentheses can clarify code to readers and to Perl itself.

Scope

Everything with a name in Perl (a variable, a function, a filehandle, a class) has a scope. This *scope* governs the lifespan and visibility of these entities. Scoping helps to enforce *encapsulation*–keeping related concepts together and preventing their details from leaking.

Lexical Scope

Lexical scope is the scope apparent to the readers of a program. Any block delimited by curly braces creates a new scope: a bare block, the block of a loop construct, the block of a `sub` declaration, an `eval` block, a `package` block, or any other non-quoting block. The Perl compiler resolves this scope during compilation.

Lexical scope describes the visibility of variables declared with `my`–*lexical* variables. A lexical variable declared in one scope is visible in that scope and any scopes nested within it, but is invisible to sibling or outer scopes:

```
# outer lexical scope
{
```

```
    package Robot::Butler

    # inner lexical scope
    my $battery_level;

    sub tidy_room {
        # further inner lexical scope
        my $timer;

        do {
            # innermost lexical scope
            my $dustpan;
            ...
        } while (@_);

        # sibling inner lexical scope
        for (@_) {
            # separate innermost scope
            my $polish_cloth;
            ...
        }
    }
}
```

...`$battery_level` is visible in all four scopes. `$timer` is visible in the method, the `do` block, and the `for` loop. `$dustpan` is visible only in the `do` block and `$polish_cloth` within the `for` loop.

Declaring a lexical in an inner scope with the same name as a lexical in an outer scope hides, or *shadows*, the outer lexical within the inner scope. For example:

```
my $name = 'Jacob';

{
    my $name = 'Edward';
    say $name;
}

say $name;
```

The silly lexical shadowing example program prints `Edward` and then `Jacob` (don't worry; they're family members, not vampires) because the lexical in the nested scope hides the lexical in the outer scope. Shadowing a lexical is a feature of encapsulation. Declaring multiple variables with the same name and type *in the same lexical scope* produces a warning message.

In real code with larger scopes, this shadowing behavior is often desirable–it's easier to understand code when a lexical is in scope only for a couple of dozen lines. Lexical shadowing *can* happen by accident, though. Limit the scope of variables and the nesting of scopes to lessen your risk.

Some lexical declarations have subtleties, such as a lexical variable used as the iterator variable of a `for` loop. Its declaration occurs *outside* of the block, but its scope is that *within* the loop block:

```
my $cat = 'Brad';

for my $cat (qw( Jack Daisy Petunia Tuxedo Choco )) {
    say "Inner cat is $cat";
}
```

```
say "Outer cat is $cat";
```

Functions–named and anonymous–provide lexical scoping to their bodies. This enables closures (Closures, pp. 84).

Our Scope

Within a scope you may declare an alias to a package variable with the `our` builtin. Like `my`, `our` enforces lexical scoping of the alias. The fully-qualified name is available everywhere, but the lexical alias is visible only within its scope.

`our` is most useful with package global variables such as `$VERSION` and `$AUTOLOAD`. You get a little bit of typo detection (declaring a package global with `our` satisfies the `strict` pragma's `vars` rule), but you still have to deal with a global variable.

Dynamic Scope

Dynamic scope resembles lexical scope in its visibility rules, but instead of looking outward in compile-time scopes, lookup traverses backwards through all of the function calls you've made to reach the current code. Dynamic scope applies only to global and package global variables (as lexicals aren't visible outside their scopes). While a package global variable may be *visible* within all scopes, its *value* may change depending on `local`ization and assignment:

```
our $scope;

sub inner {
    say $scope;
}

sub main {
    say $scope;
    local $scope = 'main() scope';
    middle();
}

sub middle {
    say $scope;
    inner();
}

$scope = 'outer scope';
main();
say $scope;
```

The program begins by declaring an `our` variable, `$scope`, as well as three functions. It ends by assigning to `$scope` and calling `main()`.

Within `main()`, the program prints `$scope`'s current value, `outer scope`, then `local`izes the variable. This changes the visibility of the symbol within the current lexical scope *as well as* in any functions called from the *current* lexical scope; that *as well as* condition is what dynamic scoping does. Thus, `$scope` contains `main() scope` within the body of both `middle()` and `inner()`. After `main()` returns, when control flow reaches the end of its block, Perl restores the original value of the `local`ized `$scope`. The final `say` prints `outer scope` once again.

Perl also uses different storage mechanisms for package variables and lexical variables. Every scope which contains lexical variables uses a data structure called a *lexical pad* or *lexpad* to store the values for its enclosed lexical variables. Every time control flow enters one of these scopes, Perl creates another lexpad to contain the values of the lexical variables for that particular call. This makes functions work correctly, especially recursive functions (Recursion, pp. 74).

Each package has a single *symbol table* which holds package variables and well as named functions. Importing (Importing, pp. 72) works by inspecting and manipulating this symbol table. So does `local`. This is why you may only `local`ize global and package global variables–never lexical variables.

`local` is most often useful with magic variables. For example, `$/`, the input record separator, governs how much data a `readline` operation will read from a filehandle. `$!`, the system error variable, contains error details for the most recent system call. `$@`, the Perl `eval` error variable, contains any error from the most recent `eval` operation. `$|`, the autoflush variable, governs whether Perl will flush the currently `select`ed filehandle after every write operation.

`local`izing these in the narrowest possible scope limits the effect of your changes. This can prevent strange behavior in other parts of your code.

State Scope

Perl's `state` keyword allows you to declare a lexical which has a one-time initialization as well as value persistence:

```
sub counter {
    state $count = 1;
    return $count++;
}

say counter();
say counter();
say counter();
```

On the first call to `counter`, Perl initializes `$count`. On subsequent calls, `$count` retains its previous value. This program prints `1`, `2`, and `3`. Change `state` to `my` and the program will print `1`, `1`, and `1`.

You may use an expression to set a `state` variable's initial value:

```
sub counter {
    state $count = shift;
    return $count++;
}

say counter(2);
say counter(4);
say counter(6);
```

Even though a simple reading of the code may suggest that the output should be `2`, `4`, and `6`, the output is actually `2`, `3`, and `4`. The first call to the sub `counter` sets the `$count` variable. Subsequent calls will not change its value.

`state` can be useful for establishing a default value or preparing a cache, but be sure to understand its initialization behavior if you use it:

```
sub counter {
    state $count = shift;
    say 'Second arg is: ', shift;
    return $count++;
}

say counter(2, 'two');
say counter(4, 'four');
say counter(6, 'six');
```

The counter for this program prints `2`, `3`, and `4` as expected, but the values of the intended second arguments to the `counter()` calls are `two`, `4`, and `6`–because the `shift` of the first argument only happens in the first call to `counter()`. Either change the API to prevent this mistake, or guard against it with:

```
sub counter {
    my ($initial_value, $text) = @_;

    state $count = $initial_value;
    say "Second arg is: $text";
    return $count++;
}

say counter(2, 'two');
say counter(4, 'four');
say counter(6, 'six');
```

Anonymous Functions

An *anonymous function* is a function without a name. It behaves exactly like a named function–you can invoke it, pass it arguments, return values from it, and take references to it. Yet you can only access an anonymous function by reference (Function References, pp. 57), not by name.

A Perl idiom known as a *dispatch table* uses hashes to associate input with behavior:

```
my %dispatch = (
    plus     => \&add_two_numbers,
    minus    => \&subtract_two_numbers,
    times    => \&multiply_two_numbers,
);

sub add_two_numbers      { $_[0] + $_[1] }
sub subtract_two_numbers { $_[0] - $_[1] }
sub multiply_two_numbers { $_[0] * $_[1] }

sub dispatch {
    my ($left, $op, $right) = @_;

    return unless exists $dispatch{ $op };

    return $dispatch{ $op }->( $left, $right );
}
```

The `dispatch()` function takes arguments of the form `(2, 'times', 2)`, evaluates the operation, and returns the result. A trivial calculator application could use `dispatch` to figure out which calculation to perform based on user input.

Declaring Anonymous Functions

The `sub` builtin used without a name creates and returns an anonymous function. Use this function reference where you'd use a reference to a named function, such as to declare the dispatch table's functions in place:

```
my %dispatch = (
    plus      => sub { $_[0]  + $_[1] },
    minus     => sub { $_[0]  - $_[1] },
    times     => sub { $_[0]  * $_[1] },
    dividedby => sub { $_[0]  / $_[1] },
    raisedto  => sub { $_[0] ** $_[1] },
);
```

Defensive Dispatch

Only those functions within this dispatch table are available for users to call. If your dispatch function used a user-provided string as the literal name of functions, a malicious user could call any function anywhere by passing a fully-qualified name such as `'Internal::Functions::malicious_function'`.

You may also see anonymous functions passed as function arguments:

```
sub invoke_anon_function {
    my $func = shift;
    return $func->( @_ );
}

sub named_func {
    say 'I am a named function!';
}

invoke_anon_function( \&named_func );
invoke_anon_function( sub { say 'Who am I?' } );
```

Anonymous Function Names

Use introspection to determine whether a function is named or anonymous, whether through `caller()` or the CPAN module `Sub::Identify`'s `sub_name()` function:

```
package ShowCaller;

sub show_caller {
    my ($package, $file, $line, $sub) = caller(1);
    say "Called from $sub in $package:$file:$line";
}

sub main {
    my $anon_sub = sub { show_caller() };
    show_caller();
    $anon_sub->();
}

main();
```

The result may be surprising:

```
  Called from ShowCaller::main
           in ShowCaller:anoncaller.pl:20
  Called from ShowCaller::__ANON__
           in ShowCaller:anoncaller.pl:17
```

The `__ANON__` in the second line of output demonstrates that the anonymous function has no name that Perl can identify. The CPAN module `Sub::Name`'s `subname()` function allows you to attach names to anonymous functions:

```
use Sub::Name;
use Sub::Identify 'sub_name';

my $anon  = sub {};
say sub_name( $anon );

my $named = subname( 'pseudo-anonymous', $anon );
say sub_name( $named );
say sub_name( $anon );

say sub_name( sub {} );
```

This program produces:

```
__ANON__
pseudo-anonymous
pseudo-anonymous
__ANON__
```

Be aware that both references refer to the same underlying anonymous function. Using `subname()` on one reference will change that underlying function; all other references to that function will see the new name.

Implicit Anonymous Functions

Perl allows you to declare anonymous functions as function arguments without using the `sub` keyword. Though this feature exists nominally to enable programmers to write their own syntax such as that for `map` and `eval` (Prototypes, pp. 170), you can use it for other things, such as to write *delayed* functions that don't look like functions.

Consider the CPAN module `Test::Fatal`, which takes an anonymous function as the first argument to its `exception()` function:

```
use Test::More;
use Test::Fatal;

my $croaker = exception { die 'I croak!' };
my $liver   = exception { 1 + 1 };

like $croaker, qr/I croak/, 'die() should croak';
is   $liver,   undef,       'addition should live';

done_testing();
```

You might rewrite this more verbosely as:

```
my $croaker = exception( sub { die 'I croak!' } );
my $liver   = exception( sub { 1 + 1 } );
```

. . . or to pass named functions by reference:

```
sub croaker { die 'I croak!' }
sub liver   { 1 + 1 }

my $croaker = exception \&croaker;
my $liver   = exception \&liver;
```

```
like $croaker, qr/I croak/, 'die() should die';
is   $liver,   undef,       'addition should live';
```

...but you may *not* pass them as scalar references:

```
my $croak_ref = \&croaker;
my $live_ref  = \&liver;

# BUGGY: does not work
my $croaker   = exception $croak_ref;
my $liver     = exception $live_ref;
```

...because the prototype changes the way the Perl parser interprets this code. It cannot determine with 100% clarity *what* `$croaker` and `$liver` will contain, and so will throw an exception.

```
    Type of arg 1 to Test::Fatal::exception
       must be block or sub {} (not private variable)
```

A function which takes an anonymous function as the first of multiple arguments *cannot* have a trailing comma after the function block:

```
use Test::More;
use Test::Fatal 'dies_ok';

dies_ok { die 'This is my boomstick!' } 'No movie references here';
```

This is an occasionally confusing wart on otherwise helpful syntax, courtesy of a quirk of the Perl parser. The syntactic clarity available by promoting bare blocks to anonymous functions can be helpful, but use it sparingly and document the API with care.

Closures

The computer science term *higher order functions* refers to functions which manipulate other functions. Every time control flow enters a function, that function gets a new environment representing that invocation's lexical scope (Scope, pp. 77). That applies equally well to anonymous functions (Anonymous Functions, pp. 81). The implication is powerful, and closures show off this power.

Creating Closures

A *closure* is a function that uses lexical variables from an outer scope. You've probably already created and used closures without realizing it:

```
use Modern::Perl '2015';

my $filename = shift @ARGV;

sub get_filename { return $filename }
```

If this code seems straightforward to you, good! *Of course* the `get_filename()` function can see the `$filename` lexical. That's how scope works!

Suppose you want to iterate over a list of items without managing the iterator yourself. You can create a function which returns a function that, when invoked, will return the next item in the iteration:

```
sub make_iterator {
    my @items = @_;
    my $count = 0;

    return sub {
        return if $count == @items;
        return $items[ $count++ ];
    }
}

my $cousins = make_iterator(qw(
    Rick Alex Kaycee Eric Corey Mandy Christine Alex
));

say $cousins->() for 1 .. 6;
```

Even though `make_iterator()` has returned, the anonymous function stored in `$cousins` has closed over the values of these variables *as they existed within* the invocation of `make_iterator()`–and their values persist (Reference Counts, pp. 58).

Because invoking `make_iterator()` creates a separate lexical environment, the anonymous sub it creates and returns closes over a unique lexical environment for each invocation:

```
my $aunts = make_iterator(qw(
    Carole Phyllis Wendy Sylvia Monica Lupe
));

say $cousins->();
say $aunts->();
```

Because `make_iterator()` does not return these lexicals by value or by reference, only the closure can access them. They're encapsulated as effectively as any other lexical is, although any code which shares a lexical environment can access these values. This idiom provides better encapsulation of what would otherwise be a file or package global variable:

```
{
    my $private_variable;

    sub set_private { $private_variable = shift }
    sub get_private { $private_variable }
}
```

Named functions have package global scope, thus you cannot *nest* named functions. Any lexical variables shared between nested functions will go unshared when the outer function destroys its first lexical environment. Perl will warn you when this happens.

Invasion of Privacy

The CPAN module `PadWalker` lets you violate lexical encapsulation, but anyone who uses it gets to fix any bugs that result.

Uses of Closures

Iterating over a fixed-sized list with a closure is interesting, but closures can do much more, such as iterating over a list which is too expensive to calculate or too large to maintain in memory all at once. Consider a function to create the Fibonacci series

as you need its elements (probably so you can check the output of your Haskell homework). Instead of recalculating the series recursively, use a cache and lazily create the elements you need:

```
sub gen_fib {
    my @fibs = (0, 1);

    return sub {
        my $item = shift;

        if ($item >= @fibs) {
            for my $calc (@fibs .. $item) {
                $fibs[$calc] = $fibs[$calc - 2]
                             + $fibs[$calc - 1];
            }
        }
        return $fibs[$item];
    }
}

# calculate 42nd Fibonacci number
my $fib = gen_fib();
say $fib->( 42 );
```

Every call to the function returned by `gen_fib()` takes one argument, the *n*th element of the Fibonacci series. The function generates and caches all preceding values in the series as necessary, and returns the requested element.

Here's where closures and first class functions get interesting. This code does two things; there's a pattern specific to caching intertwined with the numeric series. What happens if you extract the cache-specific code (initialize a cache, execute custom code to populate cache elements, and return the calculated or cached value) to a function `gen_caching_closure()`?

```
sub gen_caching_closure {
    my ($calc_element, @cache) = @_;

    return sub {
        my $item = shift;

        $calc_element->($item, \@cache) unless $item < @cache;

        return $cache[$item];
    };
}

sub gen_fib {
    my @fibs = (0, 1, 1);

    return gen_caching_closure( sub {
            my ($item, $fibs) = @_;

            for my $calc ((@$fibs - 1) .. $item) {
                $fibs->[$calc] = $fibs->[$calc - 2]
                               + $fibs->[$calc - 1];
            }
        }, @fibs
```

```
    );
}
```

The program behaves as it did before, but now function references and closures separate the cache initialization behavior from the calculation of the next number in the Fibonacci series. Customizing the behavior of code–in this case, `gen_caching_closure()`–by passing in a function allows tremendous flexibility and can clean up your code.

Fold, Apply, and Filter

The builtins `map`, `grep`, and `sort` are themselves higher-order functions.

Closures and Partial Application

Closures can also *remove* unwanted genericity. Consider the case of a function which takes several parameters:

```
sub make_sundae {
    my %args      = @_;

    my $ice_cream = get_ice_cream( $args{ice_cream} );
    my $banana    = get_banana(    $args{banana}    );
    my $syrup     = get_syrup(     $args{syrup}     );
    ...
}
```

Myriad customization possibilities might work very well in a full-sized ice cream store, but for an ice cream cart where you only serve French vanilla ice cream on Cavendish bananas, every call to `make_sundae()` passes arguments that never change.

Partial application allows you to bind *some* of the arguments to a function now so that you can provide the others later. Wrap the function you intend to call in a closure and pass the bound arguments. For your ice cream cart:

```
my $make_cart_sundae = sub {
    return make_sundae( @_,
        ice_cream => 'French Vanilla',
        banana    => 'Cavendish',
    );
};
```

Now whenever you process an order, invoke the function reference in `$make_cart_sundae` and pass only the interesting arguments. You'll never forget the invariants or pass them incorrectly. You can even use `Sub::Install` from the CPAN to import `$make_cart_sundae` function into another namespace.

This is only the start of what you can do with higher order functions. Mark Jason Dominus's *Higher Order Perl* is the canonical reference on first-class functions and closures in Perl. Read it online at http://hop.perl.plover.com/.

State versus Closures

Closures (Closures, pp. 84) use lexical scope (Scope, pp. 77) to control access to lexical variables–even with named functions:

```
{
    my $safety = 0;

    sub enable_safety  { $safety = 1 }
    sub disable_safety { $safety = 0 }
```

```
    sub do_something_awesome {
        return if $safety;
        ...
    }
}
```

All three functions encapsulate that shared state without exposing the lexical variable outside of their shared scope. This idiom works well for cases where multiple functions access that lexical, but it's clunky when only one function does. Suppose every hundredth ice cream parlor customer gets free sprinkles:

```
my $cust_count = 0;

sub serve_customer {
    $cust_count++;
    my $order = shift;

    add_sprinkles($order) if $cust_count % 100 == 0;
    ...
}
```

This approach *works*, but creating a new outer lexical scope for a single function is a little bit noisy. The `state` builtin allows you to declare a lexically scoped variable with a value that persists between invocations:

```
sub serve_customer {
    state $cust_count = 0;
    $cust_count++;

    my $order = shift;
    add_sprinkles($order)
        if ($cust_count % 100 == 0);

    ...
}
```

`state` also works within anonymous functions:

```
sub make_counter {
    return sub {
         state $count = 0;
         return $count++;
     }
}
```

. . . though there are few obvious benefits to this approach.

State versus Pseudo-State

In old versions of Perl, a named function could close over its previous lexical scope by abusing a quirk of implementation. Using a postfix conditional which evaluates to false with a `my` declaration avoided *reinitializing* a lexical variable to `undef` or its initialized value.

Now any use of a postfix conditional expression modifying a lexical variable declaration produces a deprecation warning. It's too easy to write inadvertently buggy code with this technique; use `state` instead where available, or a true closure otherwise. Rewrite this idiom when you encounter it:

```
sub inadvertent_state {
    # my $counter  = 1 if 0; # DEPRECATED; don't use
    state $counter = 1;      # prefer

    ...
}
```

You may only initialize a state variable with a scalar value. If you need to keep track of an aggregate, use a hash or array reference (References, pp. 53).

Attributes

Named entities in Perl–variables and functions–can have additional metadata attached in the form of *attributes*. These attributes are arbitrary names and values used with certain types of metaprogramming (Code Generation, pp. 150).

Attribute declaration syntax is awkward, and using attributes effectively is more art than science. Most programs never use them, but when used well they offer clarity and maintenance benefits.

A simple attribute is a colon-preceded identifier attached to a declaration:

```
my $fortress      :hidden;

sub erupt_volcano :ScienceProject { ... }
```

When Perl parses these declarations, it invokes attribute handlers named `hidden` and `ScienceProject`, if they exist for the appropriate types (scalars and functions, respectively). These handlers can do *anything*. If the appropriate handlers do not exist, Perl will throw a compile-time exception.

Attributes may include a list of parameters. Perl treats these parameters as lists of constant strings. The `Test::Class` module from the CPAN uses such parametric arguments to good effect:

```
sub setup_tests          :Test(setup)     { ... }
sub test_monkey_creation :Test(10)        { ... }
sub shutdown_tests       :Test(teardown)  { ... }
```

The `Test` attribute identifies methods which include test assertions and optionally identifies the number of assertions the method intends to run. While introspection (Reflection, pp. 121) of these classes could discover the appropriate test methods, given well-designed solid heuristics, the `:Test` attribute is unambiguous. `Test::Class` provides attribute handlers which keep track of these methods. When the class has finished parsing, `Test::Class` can loop through the list of test methods and run them.

The `setup` and `teardown` parameters allow test classes to define their own support methods without worrying about conflicts with other such methods in other classes. This separates the idea of what this class must do from how other classes do their work. Otherwise a test class might have only one method named `setup` and one named `teardown` and would have to do everything there, then call the parent methods, and so on.

Drawbacks of Attributes

Attributes have their drawbacks. The canonical pragma for working with attributes (the `attributes` pragma) has listed its interface as experimental for many years, and for good reason. Damian Conway's core module `Attribute::Handlers` is much easier to use, and Andrew Main's `Attribute::Lexical` is a newer approach. Prefer either to `attributes` whenever possible.

The worst feature of attributes is that they make it easy to warp the syntax of Perl in unpredictable ways. You may not be able to predict what code with attributes will do. Good documentation helps, but if an innocent-looking declaration on a lexical variable stores a reference to that variable somewhere, your expectations of its lifespan may be wrong. Likewise, a handler may wrap a function in another function and replace it in the symbol table without your knowledge–consider a `:memoize` attribute which automatically invokes the core `Memoize` module.

Attributes *can* help you to solve difficult problems or to make an API much easier to use. When used properly, they're powerful–but most programs never need them.

AUTOLOAD

Perl does not require you to declare every function before you call it. Perl will happily attempt to call a function even if it doesn't exist. Consider the program:

```
use Modern::Perl;

bake_pie( filling => 'apple' );
```

When you run it, Perl will throw an exception due to the call to the undefined function `bake_pie()`.

Now add a function called `AUTOLOAD()`:

```
sub AUTOLOAD {}
```

When you run the program now, nothing obvious will happen. Perl will call a function named `AUTOLOAD()` in a package–if it exists–whenever normal dispatch fails. Change the `AUTOLOAD()` to emit a message to demonstrate that it gets called:

```
sub AUTOLOAD { say 'In AUTOLOAD()!' }
```

The `AUTOLOAD()` function receives the arguments passed to the undefined function in `@_` and the fully-qualified *name* of the undefined function in the package global `$AUTOLOAD` (here, `main::bake_pie`):

```
sub AUTOLOAD {
    our $AUTOLOAD;

    # pretty-print the arguments
    local $" = ', ';
    say "In AUTOLOAD(@_) for $AUTOLOAD!"
}
```

Extract the method name with a regular expression (Regular Expressions and Matching, pp. 94):

```
sub AUTOLOAD {
    my ($name) = our $AUTOLOAD =~ /::(\w+)$/;

    # pretty-print the arguments
    local $" = ', ';
    say "In AUTOLOAD(@_) for $name!"
}
```

Whatever `AUTOLOAD()` returns, the original call receives:

```
say secret_tangent( -1 );

sub AUTOLOAD { return 'mu' }
```

So far, these examples have merely intercepted calls to undefined functions. You have other options.

Redispatching Methods in AUTOLOAD()

A common pattern in OO programming (Moose, pp. 107) is to *delegate* or *proxy* certain methods from one object to another. A logging proxy can help with debugging:

```
package Proxy::Log;

# constructor blesses reference to a scalar

sub AUTOLOAD {
    my ($name) = our $AUTOLOAD =~ /::(\w+)$/;
    Log::method_call( $name, @_ );

    my $self = shift;
    return $$self->$name( @_ );
}
```

This `AUTOLOAD()` extracts the name of the undefined method. Then it dereferences the proxied object from a blessed scalar reference, logs the method call, then invokes that method on the proxied object with the provided parameters.

Generating Code in AUTOLOAD()

This double dispatch is easy to write but inefficient. Every method call on the proxy must fail normal dispatch to end up in `AUTOLOAD()`. Pay that penalty only once by installing new methods into the proxy class as the program needs them:

```
sub AUTOLOAD {
    my ($name) = our $AUTOLOAD =~ /::(\w+)$/;
    my $method = sub { ... };

    no strict 'refs';
    *{ $AUTOLOAD } = $method;
    return $method->( @_ );
}
```

The body of the previous `AUTOLOAD()` has become a closure (Closures, pp. 84) bound over the *name* of the undefined method. Installing that closure in the appropriate symbol table allows all subsequent dispatch to that method to find the created closure (and avoid `AUTOLOAD()`). This code finally invokes the method directly and returns the result.

Though this approach is cleaner and almost always more transparent than handling the behavior directly in `AUTOLOAD()`, the code *called* by `AUTOLOAD()` may see `AUTOLOAD()` in its `caller()` list. While it may violate encapsulation to care that this occurs, leaking the details of *how* an object provides a method may also violate encapsulation.

Some code uses a tailcall (Tailcalls, pp. 37) to *replace* the current invocation of `AUTOLOAD()` with a call to the destination method:

```
sub AUTOLOAD {
    my ($name) = our $AUTOLOAD =~ /::(\w+)$/;
    my $method = sub { ... }

    no strict 'refs';
    *{ $AUTOLOAD } = $method;
    goto &$method;
}
```

This has the same effect as invoking `$method` directly, except that `AUTOLOAD()` will no longer appear in the list of calls available from `caller()`, so it looks like normal method dispatch occurred.

Drawbacks of AUTOLOAD

`AUTOLOAD()` can be useful, though it is difficult to use properly. The naïve approach to generating methods at runtime means that the `can()` method will not report the right information about the capabilities of objects and classes. The easiest solution is to predeclare all functions you plan to `AUTOLOAD()` with the `subs` pragma:

```
use subs qw( red green blue ochre teal );
```

Now You See Them

Forward declarations are useful only in the two rare cases of attributes (Attributes, pp. 89) and autoloading (AUTOLOAD, pp. 90).

That technique documents your intent well, but requires you to maintain a static list of functions or methods. Overriding `can()` (The UNIVERSAL Package, pp. 148) sometimes works better:

```
sub can {
    my ($self, $method) = @_;

    # use results of parent can()
    my $meth_ref = $self->SUPER::can( $method );
    return $meth_ref if $meth_ref;

    # add some filter here
    return unless $self->should_generate( $method );

    $meth_ref = sub { ... };
    no strict 'refs';
    return *{ $method } = $meth_ref;
}

sub AUTOLOAD {
    my ($self) = @_;
    my ($name) = our $AUTOLOAD =~ /::(\w+)$/;>

    return unless my $meth_ref = $self->can( $name );
    goto &$meth_ref;
}
```

`AUTOLOAD()` is a big hammer; it can catch functions and methods you had no intention of autoloading, such as `DESTROY()`, the destructor of objects. If you write a `DESTROY()` method with no implementation, Perl will happily dispatch to it instead of `AUTOLOAD()`:

```
# skip AUTOLOAD()
sub DESTROY {}
```

A Very Special Method

The special methods `import()`, `unimport()`, and `VERSION()` never go through `AUTOLOAD()`.

If you mix functions and methods in a single namespace which inherits from another package which provides its own `AUTOLOAD()`, you may see the strange error:

```
Use of inherited AUTOLOAD for non-method slam_door() is deprecated
```

If this happens to you, simplify your code; you've called a function which does not exist in a package which inherits from a class which contains its own `AUTOLOAD()`. The problem compounds in several ways: mixing functions and methods in a single namespace is often a design flaw, inheritance and `AUTOLOAD()` get complex very quickly, and reasoning about code when you don't know what methods objects provide is difficult.

`AUTOLOAD()` is useful for quick and dirty programming, but robust code avoids it.

CHAPTER 6

Regular Expressions and Matching

Perl's sometimes called the Practical Extraction and Reporting Language. You've seen how control flow, operators, and data structures make Perl practical and you can imagine how to create reports. What's the Extraction part mean? Perl's good at text processing, in part due to regular expressions.

A regular expression (also *regex* or *regexp*) is a *pattern* which describes characteristics of a piece of text–to extract an address, replace a misspelling, even to scrape stock prices off of a website to help you figure out what to do with your investment account. Perl's *regular expression engine* applies these patterns to match or to replace portions of text.

While mastering regular expressions is a daunting pursuit, a little knowledge will give you great power. You'll build up your knowledge over time, with practice, as you add more and more features to your toolkit.

While this chapter gives an overview of the most important regex features, it's not exhaustive. Perl's documentation includes a tutorial (`perldoc perlretut`), a reference guide (`perldoc perlreref`), and full documentation (`perldoc perlre`). If you're interested in the theory, Jeffrey Friedl's book *Mastering Regular Expressions* explains the computer science and the mechanics of how regular expressions work.

Literals

A regex can be as simple as a substring pattern:

```
my $name = 'Chatfield';
say 'Found a hat!' if $name =~ /hat/;
```

The match operator (`m//`, abbreviated `//`) identifies a regular expression–in this example, `hat`. This pattern is *not* a word. Instead it means "the `h` character, followed by the `a` character, followed by the `t` character." Each character in the pattern is an indivisible element (an *atom*). An atom either matches or it doesn't.

The regex binding operator (`=~`) is an infix operator (Fixity, pp. 64) which applies the regex of its second operand to a string provided as its first operand. When evaluated in scalar context, a match evaluates to a boolean value representing the success or failure of the match. The negated form of the binding operator (`!~`) evaluates to a true value *unless* the match succeeds.

> **Remember `index`!**
>
> The `index` builtin can also search for a literal substring within a string. Using a regex engine for that is like flying an autonomous combat drone to the corner store to buy cheese–but Perl lets you write code however you find it clear.

The substitution operator, `s///`, is in one sense a circumfix operator (Fixity, pp. 64) with two operands. Its first operand (the part between the first and second delimiters) is a regular expression. The second operand (the part between the second and third delimiters) is a substring used to replace the matched portion of the string operand used with the regex binding operator. For example, to cure pesky summer allergies:

```
my $status = 'I feel ill.';
$status    =~ s/ill/well/;
say $status;
```

The qr// Operator and Regex Combinations

The `qr//` operator creates first-class regexes you can store in variables. Use these regexes as operands to the match and substitution operators:

```
my $hat = qr/hat/;
say 'Found a hat!' if $name =~ /$hat/;
```

. . . or combine multiple regex objects into complex patterns:

```
my $hat   = qr/hat/;
my $field = qr/field/;

say 'Found a hat in a field!'
    if $name =~ /$hat$field/;

like $name, qr/$hat$field/, 'Found a hat in a field!';
```

Like `is`, with More `like`

`Test::More`'s `like` function tests that the first argument matches the regex provided as the second argument.

Quantifiers

Matching literal expressions is good, but *regex quantifiers* make regexes more powerful. These metacharacters govern how often a regex component may appear in a matching string. The simplest quantifier is the *zero or one quantifier*, or `?`:

```
my $cat_or_ct = qr/ca?t/;

like 'cat', $cat_or_ct, "'cat' matches /ca?t/";
like 'ct',  $cat_or_ct, "'ct' matches /ca?t/";
```

Any atom in a regular expression followed by the `?` character means "match zero or one instance(s) of this atom." This regular expression matches if zero or one `a` characters immediately follow a `c` character *and* immediately precede a `t` character. This regex matches both the literal substrings `cat` and `ct`.

The *one or more quantifier*, or `+`, matches at least one instance of its atom:

```
my $some_a = qr/ca+t/;

like 'cat',    $some_a, "'cat' matches /ca+t/";
like 'caat',   $some_a, "'caat' matches/";
like 'caaat',  $some_a, "'caaat' matches";
like 'caaaat', $some_a, "'caaaat' matches";

unlike 'ct',   $some_a, "'ct' does not match";
```

There is no theoretical limit to the maximum number of quantified atoms which can match.

The *zero or more quantifier*, `*`, matches zero or more instances of the quantified atom:

```
my $any_a = qr/ca*t/;

like 'cat',    $any_a, "'cat' matches /ca*t/";
like 'caat',   $any_a, "'caat' matches";
like 'caaat',  $any_a, "'caaat' matches";
like 'caaaat', $any_a, "'caaaat' matches";
like 'ct',     $any_a, "'ct' matches";
```

As silly as this seems, it allows you to specify optional components of a regex. Use it sparingly, though: it's a blunt and expensive tool. *Most* regular expressions benefit from using the `?` and `+` quantifiers far more than `*`. Be precise about your intent to clarify your code.

Numeric quantifiers express the number of times an atom may match. `{n}` means that a match must occur exactly *n* times.

```
# equivalent to qr/cat/;
my $only_one_a = qr/ca{1}t/;

like 'cat', $only_one_a, "'cat' matches /ca{1}t/";
```

`{n,}` matches an atom *at least n* times:

```
# equivalent to qr/ca+t/;
my $some_a = qr/ca{1,}t/;

like 'cat',    $some_a, "'cat' matches /ca{1,}t/";
like 'caat',   $some_a, "'caat' matches";
like 'caaat',  $some_a, "'caaat' matches";
like 'caaaat', $some_a, "'caaaat' matches";
```

`{n,m}` means that a match must occur at least *n* times and cannot occur more than *m* times:

```
my $few_a = qr/ca{1,3}t/;

like 'cat',    $few_a, "'cat' matches /ca{1,3}t/";
like 'caat',   $few_a, "'caat' matches";
like 'caaat',  $few_a, "'caaat' matches";

unlike 'caaaat', $few_a, "'caaaat' doesn't match";
```

You may express the symbolic quantifiers in terms of the numeric quantifiers, but the symbolic quantifiers are shorter and more common.

Greediness

The `+` and `*` quantifiers are *greedy*: they try to match as much of the input string as possible. This can be particularly pernicious. Consider a naïve use of the "zero or more non-newline characters" pattern of `.*`:

```
# a poor regex
my $hot_meal = qr/hot.*meal/;

say 'Found a hot meal!' if 'I have a hot meal' =~ $hot_meal;

say 'Found a hot meal!' if 'one-shot, piecemeal work!' =~ $hot_meal;
```

Greedy quantifiers start by matching *everything* at first. If that match does not succeed, the regex engine will back off one character at a time until it finds a match.

The `?` quantifier modifier turns a greedy-quantifier non-greedy:

```
my $minimal_greedy = qr/hot.*?meal/;
```

When given a non-greedy quantifier, the regular expression engine will prefer the *shortest* possible potential match. If that match fails, the engine will increase the number of characters identified by the `.*?` token combination one character at a time. Because `*` matches zero or more times, the minimal potential match for this token combination is zero characters:

```
say 'Found a hot meal' if 'ilikeahotmeal' =~ /$minimal_greedy/;
```

Use `+?` to match one or more items non-greedily:

```
my $minimal_greedy_plus = qr/hot.+?meal/;

unlike 'ilikeahotmeal',   $minimal_greedy_plus;

like 'i like a hot meal', $minimal_greedy_plus;
```

The `?` quantifier modifier applies to the `?` (zero or one matches) quantifier as well as the range quantifiers. It causes the regex to match as little of the input as possible.

Regexes are powerful, but they're not always the best way to solve a problem. This is doubly true for the greedy patterns `.+` and `.*`. A crossword puzzle fan who needs to fill in four boxes of 7 Down ("Rich soil") will find too many invalid candidates with the pattern:

```
my $seven_down = qr/l$letters_only*m/;
```

If you run this against all of the words in a dictionary, it'll match `Alabama`, `Belgium`, and `Bethlehem` long before it reaches `loam`, the real answer. Not only are those words too long, but the matched portions occur everywhere in the word, not just at the start.

Regex Anchors

It's important to know how the regex engine handles greedy matches–but it's equally as important to know what kind of matches you want. *Regex anchors* force the regex engine to start or end a match at a fixed position. The *start of string anchor* (`\A`) dictates that any match must start at the beginning of the string:

```
# also matches "lammed", "lawmaker", and "layman"
my $seven_down = qr/\Al${letters_only}{2}m/;
```

The *end of line string anchor* (`\z`) requires that a match end at the end of the string.

```
# also matches "loom", but an obvious improvement
my $seven_down = qr/\Al${letters_only}{2}m\z/;
```

You will often see the `^` and `$` assertions used to match the start and end of strings. `^` *does* match the start of the string, but in certain circumstances it can match the invisible point just after a newline within the string. Similarly, `$` *does* match the end of the string (just before a newline, if it exists), but it can match the invisible point just before a newline in the middle of the string. `\A` and `\z` are more specific and, thus, more useful.

The *word boundary anchor* (`\b`) matches only at the boundary between a word character (`\w`) and a non-word character (`\W`). That boundary isn't a character in and of itself; it has no width. It's invisible. Use an anchored regex to find `loam` while prohibiting `Belgium`:

```
my $seven_down = qr/\bl${letters_only}{2}m\b/;
```

This anchor has one flaw which may or may not trip you; it doesn't understand punctuation such as the apostrophe. Fortunately, Perl 5.22 added the *Unicode word boundary metacharacter* `\b{wb}`, which *does* understand contractions:

```
say "Panic"    if     "Don't Panic" =~ /Don\b/;
say "No Panic" unless "Don't Panic" =~ /Don\b{wb}/;
```

Metacharacters

Perl interprets several characters in regular expressions as *metacharacters*, characters represent something other than their literal interpretation. You've seen a few metacharacters already (`\b`, `.`, and `?`, for example). Metacharacters give regex wielders power far beyond mere substring matches. The regex engine treats all metacharacters as atoms. See `perldoc perlrebackslash` for far more detail about metacharacters.

The `.` metacharacter means "match any character except a newline". Many novices forget that nuance. A simple regex search–ignoring the obvious improvement of using anchors–for 7 Down might be `/l..m/`. Of course, there's always more than one way to get the right answer:

```
for my $word (@words) {
    next unless length( $word ) == 4;
    next unless $word =~ /l..m/;
    say "Possibility: $word";
}
```

If the potential matches in `@words` are more than the simplest English words, you will get false positives. `.` also matches punctuation characters, whitespace, and numbers. Be specific! The `\w` metacharacter represents all Unicode alphanumeric characters (Unicode and Strings, pp. 19) and the underscore:

```
    next unless $word =~ /l\w\wm/;
```

The `\d` metacharacter matches Unicode digits:

```
next unless $number =~ /\d{3}-\d{3}-\d{4}/;
say "I have your number: $number";
```

. . . though in this case, the `Regexp::English` module has a much better phone number regex already written for you.

Use the `\s` metacharacter to match whitespace. *Whitespace* means a literal space, a tab character, a carriage return, a form-feed, or a newline:

```
my $two_three_letter_words = qr/\w{3}\s\w{3}/;
```

Negated Metacharacters

These metacharacters have negated forms. Use `\W` to match any character *except* a word character. Use `\D` to match a non-digit character. Use `\S` to match anything but whitespace. Use `\B` to match anywhere except a word boundary and `\B{wb}` to match anywhere except a Unicode word boundary.

Character Classes

When none of those metacharacters is specific enough, group multiple characters into a *character class* by enclosing them in square brackets. A character class allows you to treat a group of alternatives as a single atom.

```
my $ascii_vowels = qr/[aeiou]/;
my $maybe_cat    = qr/c${ascii_vowels}t/;
```

Interpolation Happens

Without those curly braces, Perl's parser would interpret the variable name as `$ascii_vowelst`, which either causes a compile-time error about an unknown variable or interpolates the contents of an existing `$ascii_vowelst` into the regex.

The hyphen character (`-`) allows you to include a contiguous range of characters in a class, such as this `$ascii_letters_only` regex:

```
my $ascii_letters_only = qr/[a-zA-Z]/;
```

To include the hyphen as a member of the class, place it at the start or end of the class:

```
my $interesting_punctuation = qr/[-!?]/;
```

. . . or escape it:

```
my $line_characters = qr/[|=\-_]/;
```

Use the caret (`^`) as the first element of the character class to mean "anything *except* these characters":

```
my $not_an_ascii_vowel = qr/[^aeiou]/;
```

Use a caret anywhere but the first position to make it a member of the character class. To include a hyphen in a negated character class, place it after the caret or at the end of the class–or escape it.

Capturing

Regular expressions allow you to group and capture portions of the match for later use. To extract an American telephone number of the form `(202) 456-1111` from a string:

```
my $area_code    = qr/\(\d{3}\)/;
my $local_number = qr/\d{3}-?\d{4}/;
my $phone_number = qr/$area_code\s?$local_number/;
```

Note the escaped parentheses within `$area_code`. Parentheses are special in Perl regular expressions. They group atoms into larger units and capture portions of matching strings. To match literal parentheses, escape them with backslashes as seen in `$area_code`.

Named Captures

Named captures allow you to capture portions of matches from applying a regular expression and access them later. For example, when extracting a phone number from contact information:

```
if ($contact_info =~ /(?<phone>$phone_number)/) {
    say "Found a number $+{phone}";
}
```

Named capture syntax has the form:

```
(?<capture name> ... )
```

Parentheses enclose the capture. The `?< name >` construct immediately follows the opening parenthesis and provides a name for this particular capture. The remainder of the capture is a regular expression.

When a match against the enclosing pattern succeeds, Perl updates the magic variable `%+`. In this hash, the key is the name of the capture and the value is the portion of the string which matched the capture.

Numbered Captures

Perl also supports *numbered captures*:

```
if ($contact_info =~ /($phone_number)/) {
    say "Found a number $1";
}
```

This form of capture provides no identifying name and does nothing to `%+`. Instead, Perl stores the captured substring in a series of magic variables. The *first* matching capture goes into `$1`, the second into `$2`, and so on. Capture counts start at the *opening* parenthesis of the capture. Thus the first left parenthesis begins the capture into `$1`, the second into `$2`, and so on.

While the syntax for named captures is longer than for numbered captures, it provides additional clarity. Counting left parentheses is tedious, and combining regexes which each contain numbered captures is difficult. Named captures improve regex maintainability–though name collisions are possible, they're relatively infrequent. Minimize the risk by using named captures only in top-level regexes, rather than in smaller regexes composed into larger.

In list context, a regex match returns a list of captured substrings:

```
if (my ($number) = $contact_info =~ /($phone_number)/) {
    say "Found a number $number";
}
```

Numbered captures are also useful in simple substitutions, where named captures may be more verbose:

```
my $order = 'Vegan brownies!';

$order =~ s/Vegan (\w+)/Vegetarian $1/;
# or
$order =~ s/Vegan (?<food>\w+)/Vegetarian $+{food}/;
```

Grouping and Alternation

Previous examples have all applied quantifiers to simple atoms. You may apply them to any regex element:

```
my $pork  = qr/pork/;
my $beans = qr/beans/;

like 'pork and beans', qr/\A$pork?.*?$beans/,
     'maybe pork, definitely beans';
```

If you expand the regex manually, the results may surprise you:

```
my $pork_and_beans = qr/\Apork?.*beans/;

like 'pork and beans', qr/$pork_and_beans/,
     'maybe pork, definitely beans';
like 'por and beans', qr/$pork_and_beans/,
     'wait... no phylloquinone here!';
```

Sometimes specificity helps pattern accuracy:

```
my $pork  = qr/pork/;
my $and   = qr/and/;
my $beans = qr/beans/;

like 'pork and beans', qr/\A$pork? $and? $beans/,
     'maybe pork, maybe and, definitely beans';
```

Some regexes need to match either one thing or another. The *alternation* metacharacter (`|`) indicates that either possibility may match.

```
my $rice  = qr/rice/;
my $beans = qr/beans/;

like 'rice',  qr/$rice|$beans/, 'Found rice';
like 'beans', qr/$rice|$beans/, 'Found beans';
```

While it's easy to interpret `rice|beans` as meaning `ric`, followed by either `e` or `b`, followed by `eans`, alternations always include the *entire* fragment to the nearest regex delimiter, whether the start or end of the pattern, an enclosing parenthesis, another alternation character, or a square bracket.

Alternation has a lower precedence (Precedence, pp. 63) than even atoms:

```
like   'rice',  qr/rice|beans/, 'Found rice';
like   'beans', qr/rice|beans/, 'Found beans';
unlike 'ricb',  qr/rice|beans/, 'Found hybrid';
```

To reduce confusion, use named fragments in variables (`$rice|$beans`) or group alternation candidates in *non-capturing groups*:

```
my $starches = qr/(?:pasta|potatoes|rice)/;
```

The `(?:)` sequence groups a series of atoms without making a capture.

Non-Captured For Your Protection

A stringified regular expression includes an enclosing non-capturing group; `qr/rice|beans/` stringifies as `(?^u:rice|beans)`.

Other Escape Sequences

To match a *literal* instance of a metacharacter, *escape* it with a backslash (\). You've seen this before, where \(refers to a single left parenthesis and \] refers to a single right square bracket. \. refers to a literal period character instead of the "match anything but an explicit newline character" atom.

Remember to escape the alternation metacharacter (|) as well as the end of line metacharacter ($) and the quantifiers (+, ?, *) if you want to match their symbols literally.

The *metacharacter disabling characters* (\Q and \E) disable metacharacter interpretation within their boundaries. This is especially useful when you don't control the source of match text:

```
my ($text, $literal_text) = @_;

return $text =~ /\Q$literal_text\E/;
```

The $literal_text argument can contain anything–the string ** ALERT **, for example. Within the fragment bounded by \Q and \E, Perl will interpret the regex as ** ALERT ** and attempt to match literal asterisk characters instead of treating the asterisks as greedy quantifiers.

> **Regex Security**
>
> Be cautious when processing regular expressions from untrusted user input. A malicious regex master can craft a regular expression which may take *years* to match input strings, creating a denial-of-service attack against your program.

Assertions

Regex anchors such as \A, \b, \B, and \Z are *regex assertions*. These assertions do not match individual characters within the string. Instead they match specific conditions of the string. For example, no matter what the string contains, the regex qr/\A/ will *always* match.

Zero-width assertions match a *pattern*. Most importantly, they do not *consume* the portion of the pattern that they match. For example, to find a cat on its own, you might use a word boundary assertion:

```
my $just_a_cat = qr/cat\b/;

# or
my $just_a_cat = qr/cat\b{wb}/;
```

. . . but to find a non-disastrous feline, you could use a *zero-width negative look-ahead assertion*:

```
my $safe_feline = qr/cat(?!astrophe)/;
```

The construct (?!...) matches the phrase cat only if the phrase astrophe does not immediately follow. The *zero-width positive look-ahead assertion*:

```
my $disastrous_feline = qr/cat(?=astrophe)/;
```

. . . matches the phrase cat only if the phrase astrophe immediately follows. While a normal regular expression can accomplish the same thing, consider a regex to find all non-catastrophic words in the dictionary which start with cat:

```
my $disastrous_feline = qr/cat(?!astrophe)/;

while (<$words>) {
    chomp;
    next unless /\A(?<cat>$disastrous_feline.*)\Z/;
    say "Found a non-catastrophe '$+{cat}'";
}
```

The zero-width assertion consumes none of the source string, which leaves the anchored fragment `.*\Z` to match. Otherwise, the capture would only capture the `cat` portion of the source string.

To assert that your feline never occurs at the start of a line, use a *zero-width negative look-behind assertion*. These assertions must have fixed sizes, and thus you may not use quantifiers:

```
my $middle_cat = qr/(?<!\A)cat/;
```

The construct `(?<!...)` contains the fixed-width pattern. You could also express that the `cat` must always occur immediately after a space character with a *zero-width positive look-behind assertion*:

```
my $space_cat = qr/(?<=\s)cat/;
```

The construct `(?<=...)` contains the fixed-width pattern. This approach can be useful when combining a global regex match with the `\G` modifier.

Perl also includes the *keep* assertion `\K`. This zero-width positive look-behind assertion *can* have a variable length:

```
my $spacey_cat = qr/\s+\Kcat/;

like 'my cat has been to space', $spacey_cat;
like 'my  cat  has  been  to  doublespace', $spacey_cat;
```

`\K` is surprisingly useful for certain substitutions which remove the end of a pattern. It lets you match a pattern but remove only a portion of it:

```
my $exclamation = 'This is a catastrophe!';
$exclamation    =~ s/cat\K\w+!/./;

like $exclamation, qr/\bcat\./, "That wasn't so bad!";
```

Everything until the `\K` assertion matches, but only the portion of the match *after* the assertion will be substituted away.

Regex Modifiers

Several modifiers change the behavior of the regular expression operators. These modifiers appear at the end of the match, substitution, and `qr//` operators. For example, to enable case-insensitive matching:

```
my $pet = 'ELLie';

like $pet, qr/Ellie/,  'Nice puppy!';
like $pet, qr/Ellie/i, 'shift key br0ken';
```

The first `like()` will fail because the strings contain different letters. The second `like()` will pass, because the `/i` modifier causes the regex to ignore case distinctions. `L` and `l` are effectively equivalent in the second regex due to the modifier.

You may also embed regex modifiers within a pattern:

```
my $find_a_cat = qr/(?<feline>(?i)cat)/;
```

The `(?i)` syntax enables case-insensitive matching only for its enclosing group–in this case, the named capture. You may use multiple modifiers with this form. Disable specific modifiers by preceding them with the minus character (`-`):

```
my $find_a_rational = qr/(?<number>(?-i)Rat)/;
```

The multiline operator, `/m`, allows the `^` and `$` anchors to match at any newline embedded within the string.

The `/s` modifier treats the source string as a single line such that the `.` metacharacter matches the newline character. Damian Conway suggests the mnemonic that `/m` modifies the behavior of *multiple* regex metacharacters, while `/s` modifies the behavior of a *single* regex metacharacter.

The `/r` modifier causes a substitution operation to return the result of the substitution, leaving the original string unchanged. If the substitution succeeds, the result is a modified copy of the original. If the substitution fails (because the pattern does not match), the result is an unmodified copy of the original:

```
my $status     = 'I am hungry for pie.';

my $newstatus  = $status =~ s/pie/cake/r;
my $statuscopy = $status =~ s/liver and onions/bratwurst/r;

is $status, 'I am hungry for pie.',
   'original string should be unmodified';

like   $newstatus,  qr/cake/,      'cake wanted';
unlike $statuscopy, qr/bratwurst/, 'wurst not want not';
```

The `/x` modifier allows you to embed additional whitespace and comments within patterns. With this modifier in effect, the regex engine ignores whitespace and comments, so your code can be more readable:

```
my $attr_re = qr{
    \A                    # start of line

    (?:
      [;\n\s]*            # spaces and semicolons
      (?:/\*.*?\*/)?      # C comments
    )*

    ATTR

    \s+
    (   U?INTVAL
      | FLOATVAL
      | STRING\s+\*
    )
}x;
```

This regex isn't *simple*, but comments and whitespace improve its readability. Even if you compose regexes together from compiled fragments, the `/x` modifier can still improve your code.

The `/g` modifier matches a regex globally throughout a string. This makes sense when used with a substitution:

```
# appease the Mitchell estate
my $contents = slurp( $file );
$contents    =~ s/Scarlett O'Hara/Mauve Midway/g;
```

When used with a match–not a substitution–the \G metacharacter allows you to process a string within a loop one chunk at a time. \G matches at the position where the most recent match ended. To process a poorly-encoded file full of American telephone numbers in logical chunks, you might write:

```
while ($contents =~ /\G(\w{3})(\w{3})(\w{4})/g) {
    push @numbers, "($1) $2-$3";
}
```

Be aware that the \G anchor will begin at the *last* point in the string where the previous iteration of the match occurred. If the previous match ended with a greedy match such as .*, the next match will have less available string to match. Lookahead assertions can also help.

The /e modifier allows you to write arbitrary code on the right side of a substitution operation. If the match succeeds, the regex engine will use the return value of that code as the substitution value. The earlier global substitution example could be simpler with code like:

```
# appease the Mitchell estate
$sequel  =~ s{Scarlett( O'Hara)?}
            {
               'Mauve' . defined $1
                       ? ' Midway'
                       : ''
            }ge;
```

Each additional occurrence of the /e modifier will cause another evaluation of the result of the expression, though only Perl golfers use anything beyond /ee.

Smart Matching

The smart match operator, ~~, compares two operands and returns a true value if they match. The type of comparison depends on the type of both operands. given (Switch Statements, pp. 36) performs an implicit smart match.

This feature is experimental. The details of the current design are complex and unwieldy, and no proposal for simplifying things has gained enough popular support to warrant the feature's overhaul. The more complex your operands, the more likely you are to receive confusing results. Avoid comparing objects and stick to simple operations between two scalars or one scalar and one aggregate for the best results.

The smart match operator is an infix operator:

```
use experimental 'smartmatch';

say 'They match (somehow)' if $l_operand ~~ $r_operand;
```

The type of comparison *generally* depends first on the type of the right operand and then on the left operand. For example, if the right operand is a scalar with a numeric component, the comparison will use numeric equality. If the right operand is a regex, the comparison will use a grep or a pattern match. If the right operand is an array, the comparison will perform a grep or a recursive smart match. If the right operand is a hash, the comparison will check the existence of one or more keys. A large and intimidating chart in perldoc perlsyn gives far more details about all the comparisons smart match can perform.

These examples are deliberately simple, because smart match can be confusing:

```
use experimental 'smartmatch';

my ($x, $y) = (10, 20);
say 'Not equal numerically' unless $x ~~ $y;
```

```
my $z = '10 little endians';
say 'Equal numeric-ishally' if $x ~~ $z;

my $needle = qr/needle/;

say 'Pattern match'                    if 'needle'  ~~ $needle;
say 'Grep through array'               if @haystack ~~ $needle;
say 'Grep through hash keys'           if %hayhash  ~~ $needle;
say 'Grep through array'               if $needle   ~~ @haystack;

say 'Array elements exist as hash keys' if %hayhash  ~~ @haystack;
say 'Smart match elements'              if @straw    ~~ @haystack;
say 'Grep through hash keys'            if $needle   ~~ %hayhash;
say 'Array elements exist as hash keys' if @haystack ~~ %hayhash;

say 'Hash keys identical'               if %hayhash  ~~ %haymap;
```

Smart match works even if one operand is a *reference* to the given data type:

```
say 'Hash keys identical' if %hayhash ~~ \%hayhash;
```

It's difficult to recommend the use of smart match except in the simplest circumstances, but it can be useful when you have a literal string or number to match against a variable.

CHAPTER 7

Objects

Every large program has several levels of design. At the bottom, you have specific details about the problem you're solving. At the top levels, you have to organize the code so it makes sense. Our only hope to manage this complexity is to exploit *abstraction* (treating similar things similarly) and *encapsulation* (grouping related details together).

Functions alone are insufficient for large problems. Several techniques group functions into units of related behaviors; you've already seen higher-order functions. Another popular technique is *object orientation* (OO), or *object oriented programming* (OOP), where programs work with *objects*–discrete, unique entities with their own identities.

Moose

Perl's default object system is minimal but flexible. Its syntax is a little clunky, and it exposes *how* an object system works. You can build great things on top of it, but it doesn't give you what many other languages do by default.

Moose is a complete object system for Perl. It's a complete distribution available from the CPAN–not a part of the core language, but worth installing and using regardless. Moose offers both a simpler way to use an object system as well as advanced features of languages such as Smalltalk and Common Lisp.

Moose objects work with plain vanilla Perl. Within your programs, you can mix and match objects written with Perl's default object system and Moose.

Moose Documentation

See `Moose::Manual` on the CPAN for comprehensive Moose documentation.

Classes

A Moose object is a concrete instance of a *class*, which is a template describing data and behavior specific to the object. A class generally belongs to a package (Packages, pp. 51), which provides its name:

```
package Cat {
    use Moose;
}
```

This `Cat` class *appears* to do nothing, but that's all Moose needs to make a class. Create objects (or *instances*) of the `Cat` class with the syntax:

```
my $brad = Cat->new;
my $jack = Cat->new;
```

In the same way that this arrow operator dereferences a reference, it calls a method on `Cat`.

Methods

A *method* is a function associated with a class. In the same way that a function belongs to a namespace, a method belongs to a class.

When you call a method, you do so with an *invocant*. When you call `new()` on `Cat`, the name of the class, `Cat`, is `new()`'s invocant. Think of this as sending a message to a class: "do whatever `new()` does." In this case, calling the `new()` method–sending the `new` message–returns a new object of the `Cat` class.

When you call a method on an *object*, that object is the invocant:

```
my $choco = Cat->new;
$choco->sleep_on_keyboard;
```

A method's first argument is its invocant (`$self`, by convention). Suppose a `Cat` can `meow()`:

```
package Cat {
    use Moose;

    sub meow {
        my $self = shift;
        say 'Meow!';
    }
}
```

Now any `Cat` instance can wake you for its early morning feeding:

```
# the cat always meows three times at 6 am
my $fuzzy_alarm = Cat->new;
$fuzzy_alarm->meow for 1 .. 3;
```

Every object can have its own distinct data. Methods which read or write the data of their invocants are *instance methods*; they depend on the presence of an appropriate invocant to work correctly. Methods (such as `meow()`) which do not access instance data are *class methods*. You may invoke class methods on classes and class and instance methods on instances, but you cannot invoke instance methods on classes.

Class methods are effectively namespaced global functions. Without access to instance data, they have few advantages over namespaced functions. Most OO code uses instance methods to read and write instance data.

Constructors, which *create* instances, are class methods. When you declare a Moose class, Moose provides a default constructor named `new()`.

Attributes

Every Perl object is unique. Objects can contain private data associated with each unique object–often called *attributes*, *instance data*, or object *state*. Define an attribute by declaring it as part of the class:

```
package Cat {
    use Moose;

    has 'name', is => 'ro', isa => 'Str';
}
```

Moose exports the `has()` function for you to use to declare an attribute. In English, this code reads "`Cat` objects have a `name` attribute. It's read-only, and is a string." The first argument, `'name'`, is the attribute's name. The `is => 'ro'` pair of arguments declares that this attribute is `r`ead only, so you cannot modify the attribute's value after you've set it. Finally, the `isa => 'Str'` pair declares that the value of this attribute can only be a `Str`ing.

From this code Moose creates an *accessor* method named `name()` and allows you to pass a `name` parameter to `Cat`'s constructor:

```
for my $name (qw( Tuxie Petunia Daisy )) {
    my $cat = Cat->new( name => $name );
    say "Created a cat for ", $cat->name;
}
```

Moose's documentation uses parentheses to separate attribute names and characteristics:

```
has 'name' => ( is => 'ro', isa => 'Str' );
```

This is equivalent to:

```
has( 'name', 'is', 'ro', 'isa', 'Str' );
```

Moose's approach works nicely for complex declarations:

```
has 'name' => (
    is         => 'ro',
    isa        => 'Str',

    # advanced Moose options; perldoc Moose
    init_arg   => undef,
    lazy_build => 1,
);
```

...while this book prefers a low-punctuation approach for simple declarations. Choose the style which offers you the most clarity.

When an attribute declaration has a type, Moose will attempt to validate all values assigned to that attribute. Sometimes this strictness is invaluable. While Moose will complain if you try to set `name` to a value which isn't a string, attributes do not *require* types. In that case, anything goes:

```
package Cat {
    use Moose;

    has 'name', is => 'ro', isa => 'Str';
    has 'age',  is => 'ro';
}

my $invalid = Cat->new( name => 'bizarre', age  => 'purple' );
```

If you mark an attribute as readable *and* writable (with `is => rw`), Moose will create a *mutator* method which can change that attribute's value:

```
package Cat {
    use Moose;

    has 'name', is => 'ro', isa => 'Str';
    has 'age',  is => 'ro', isa => 'Int';
    has 'diet', is => 'rw';
}

my $fat = Cat->new( name => 'Fatty',
                    age  => 8,
```

```
                    diet => 'Sea Treats' );

say $fat->name, ' eats ', $fat->diet;

$fat->diet( 'Low Sodium Kitty Lo Mein' );
say $fat->name, ' now eats ', $fat->diet;
```

An `ro` accessor used as a mutator will throw the exception `Cannot assign a value to a read-only accessor at ...`.

Using `ro` or `rw` is a matter of design, convenience, and purity. Moose enforces no single philosophy here. Some people suggest making all instance data `ro` such that you must pass instance data into the constructor (Immutability, pp. 124). In the `Cat` example, `age()` might still be an accessor, but the constructor could take the *year* of the cat's birth and calculate the age itself based on the current year. This approach consolidates validation code and ensures that all objects have valid data after creation.

This illustrates a subtle but important principle of object orientation. An object contains related data and can perform behaviors with and on that data. A class describes that data and those behaviors. You can have multiple independent objects with separate instance data and treat all of those objects the same way; they will behave differently depending on their instance data.

Encapsulation

Moose allows you to declare *which* attributes class instances possess (a cat has a name) as well as the attributes of those attributes (you can name a cat once and thereafter its name cannot change). Moose itself decides how to *store* those attributes–you access them through accessors. This is *encapsulation*: hiding the internal details of an object from external users of that object.

Consider the aforementioned idea to change how `Cat`s manage their ages by passing in the year of the cat's birth and calculating the age as needed:

```
package Cat {
    use Moose;

    has 'name',        is => 'ro', isa => 'Str';
    has 'diet',        is => 'rw';
    has 'birth_year',  is => 'ro', isa => 'Int';

    sub age {
        my $self = shift;
        my $year = (localtime)[5] + 1900;

        return $year - $self->birth_year;
    }
}
```

While the syntax for *creating* `Cat` objects has changed, the syntax for *using* `Cat` objects has not. Outside of `Cat`, `age()` behaves as it always has. *How* it works is a detail hidden inside the `Cat` class.

Compatibility and APIs

Retain the old syntax for *creating* `Cat` objects by customizing the generated `Cat` constructor to allow passing an `age` parameter. Calculate `birth_year` from that. See `perldoc Moose::Manual::Attributes`.

This change offers another advantage; a *default attribute value* will let users construct a new `Cat` object *without* providing a birth year:

```
package Cat {
    use Moose;

    has 'name', is => 'ro', isa => 'Str';
    has 'diet', is => 'rw', isa => 'Str';

    has 'birth_year',
        is      => 'ro',
        isa     => 'Int',
        default => sub { (localtime)[5] + 1900 };
}
```

The `default` keyword on an attribute uses a function reference (or a literal string or number) which returns the default value for that attribute when constructing a new object. If the code creating an object passes no constructor value for that attribute, the object gets the default value:

```
my $kitten = Cat->new( name => 'Hugo' );
```

. . . and that kitten will have an age of `0` until next year.

Polymorphism

The real power of object orientation goes beyond classes and encapsulation. A well-designed OO program can manage many types of data. When well-designed classes encapsulate specific details of objects into the appropriate places, something curious happens: the code often becomes *less* specific.

Moving the details of what the program knows about individual `Cats` (the attributes) and what the program knows that `Cats` can do (the methods) into the `Cat` class means that code that deals with `Cat` instances can happily ignore *how* `Cat` does what it does.

Consider a function which displays details of an object:

```
sub show_vital_stats {
    my $object = shift;

    say 'My name is ', $object->name;
    say 'I am ',       $object->age;
    say 'I eat ',      $object->diet;
}
```

This function obviously works if you pass it a `Cat` object. It will also do the right thing for *any* object with the appropriate three accessors, no matter *how* that object provides those accessors and no matter *what kind* of object it is: `Cat`, `Caterpillar`, or `Catbird`, or even if the class uses Moose at all. `show_vital_stats()` cares that an invocant is valid only in that it supports three methods, `name()`, `age()`, and `diet()` which take no arguments and each return something which can concatenate in a string context. Your code may have a hundred different classes with no obvious relationship between them, but they will all work with this function if they support the behavior it expects.

This property is *polymorphism*: you can substitute an object of one class for an object of another class if they provide the same external interface.

Duck Typing

Some languages and environments require you to imply or declare a formal relationship between two classes before allowing a program to substitute instances for each other. Perl makes no such requirement. You may treat any two instances with methods of the same name as equivalent. Some people call this *duck typing*, arguing that any object which can `quack()` is sufficiently duck-like that you can treat it as a duck.

Without object polymorphism, enumerating a zoo's worth of animals would be tedious. Similarly, you may already start to see how calculating the age of an ocelot or octopus should be the same as calculating the age of a `Cat`. Hold that thought.

Of course, the mere existence of a method called `name()` or `age()` does not by itself imply the behavior of that object. A `Dog` object may have an `age()` which is an accessor such that you can discover `$rodney` is 13 but `$lucky` is 8. A `Cheese` object may have an `age()` method that lets you control how long to stow `$cheddar` to sharpen it. `age()` may be an accessor in one class but not in another:

```
# how old is the cat?
my $years = $zeppie->age;

# store the cheese in the warehouse for six months
$cheese->age;
```

Sometimes it's useful to know *what* an object does and what that *means*.

Roles

A *role* is a named collection of behavior and state. (Many of the ideas come from Smalltalk traits[1]) While a class organizes behaviors and state into a template for objects, a role organizes a named collection of behaviors and state. You can instantiate a class, but not a role. A role is something a class *does*.

Given an `Animal` which has an age and a `Cheese` which can age, one difference may be that `Animal` does the `LivingBeing` role, while the `Cheese` does the `Storable` role:

```
package LivingBeing {
    use Moose::Role;

    requires qw( name age diet );
}
```

The `requires` keyword provided by `Moose::Role` allows you to list methods that this role requires of its composing classes. Anything which does this role must supply the `name()`, `age()`, and `diet()` methods. The `Cat` class must declare that it performs the role:

```
package Cat {
    use Moose;

    has 'name', is => 'ro', isa => 'Str';
    has 'diet', is => 'rw', isa => 'Str';

    has 'birth_year',
        is      => 'ro',
        isa     => 'Int',
        default => sub { (localtime)[5] + 1900 };

    with 'LivingBeing';

    sub age { ... }
}
```

[1] http://scg.unibe.ch/research/traits.

The `with` line causes Moose to *compose* the `LivingBeing` role into the `Cat` class. Composition ensures all of the attributes and methods of the role are part of the class. `LivingBeing` requires any composing class to provide methods named `name()`, `age()`, and `diet()`. `Cat` satisfies these constraints. If `LivingBeing` were composed into a class which did not provide those methods, Moose would throw an exception.

Order Matters!

The `with` keyword used to apply roles to a class must occur *after* attribute declaration so that composition can identify any generated accessor methods. This is a side-effect of the implementation of Moose and not an intrinsic feature of roles.

Now all `Cat` instances will return a true value when queried if they provide the `LivingBeing` role. `Cheese` objects should not:

```
say 'Alive!' if $fluffy->DOES( 'LivingBeing' );
say 'Moldy!' if $cheese->DOES( 'LivingBeing' );
```

This design technique separates the *capabilities* of classes and objects from the *implementation* of those classes and objects. As implied earlier, the birth year calculation behavior of the `Cat` class could itself be a role:

```
package CalculateAge::From::BirthYear {
    use Moose::Role;

    has 'birth_year',
        is      => 'ro',
        isa     => 'Int',
        default => sub { (localtime)[5] + 1900 };

    sub age {
        my $self = shift;
        my $year = (localtime)[5] + 1900;

        return $year - $self->birth_year;
    }
}
```

Extracting this role from `Cat` makes the useful behavior available to other classes. Now `Cat` can compose both roles:

```
package Cat {
    use Moose;

    has 'name', is => 'ro', isa => 'Str';
    has 'diet', is => 'rw';

    with 'LivingBeing', 'CalculateAge::From::BirthYear';
}
```

The `age()` method of `CalculateAge::From::BirthYear` satisfies the requirement of the `LivingBeing` role. Extracting the `CalculateAge::From::BirthYear` role has only changed the details of *how* `Cat` calculates an age. It's still a `LivingBeing`. `Cat` can choose to implement its own age or get it from somewhere else. All that matters is that it provides an `age()` which satisfies the `LivingBeing` constraint.

While polymorphism means that you can treat multiple objects with the same behavior in the same way, *allomorphism* means that an object may implement the same behavior in multiple ways. Pervasive allomorphism can reduce the size of your classes and increase the amount of code shared between them. It also allows you to name specific and discrete collections of behaviors–very useful for testing for capabilities instead of implementations.

Roles and DOES()

When you compose a role into a class, the class and its instances will return a true value when you call `DOES()` on them:

```
say 'This Cat is alive!' if $kitten->DOES( 'LivingBeing' );
```

Inheritance

Perl's object system supports *inheritance*, which establishes a parent and child relationship between two classes such that a child specializes its parent. The child class behaves the same way as its parent–it has the same number and types of attributes and can use the same methods. It may have additional data and behavior, but you may substitute any instance of a child where code expects its parent. In one sense, a subclass provides the role implied by the existence of its parent class.

Roles versus Inheritance

Should you use roles or inheritance? Roles provide composition-time safety, better type checking, better factoring of code, and finer-grained control over names and behaviors, but inheritance is more familiar to experienced developers of other languages. Use inheritance when one class truly *extends* another. Use a role when a class needs additional behavior, especially when that behavior has a meaningful name.
Roles compare favorably to other design techniques such as mixins, multiple inheritance, and monkeypatching[a].

[a] http://www.modernperlbooks.com/mt/2009/04/the-why-of-perl-roles.html

Consider a `LightSource` class which provides two public attributes (`enabled` and `candle_power`) and two methods (`light` and `extinguish`):

```
package LightSource {
    use Moose;

    has 'candle_power', is      => 'ro',
                        isa     => 'Int',
                        default => 1;

    has 'enabled', is      => 'ro',
                   isa     => 'Bool',
                   default => 0,
                   writer  => '_set_enabled';

    sub light {
        my $self = shift;
        $self->_set_enabled( 1 );
    }

    sub extinguish {
        my $self = shift;
        $self->_set_enabled( 0 );
    }
}
```

Note that `enabled`'s `writer` option creates a private accessor usable within the class to set the value.

Inheritance and Attributes

A subclass of `LightSource` could define an industrial-strength super candle with a hundred times the luminance:

```
package SuperCandle {
    use Moose;

    extends 'LightSource';

    has '+candle_power', default => 100;
}
```

`extends` takes a list of class names to use as parents of the current class. If that were the only line in this class, `SuperCandle` objects would behave in the same ways as `LightSource` objects. A `SuperCandle` instance would have both the `candle_power` and `enabled` attributes as well as the `light()` and `extinguish()` methods.

The `+` at the start of an attribute name (such as `candle_power`) indicates that the current class does something special with that attribute. Here the super candle overrides the default value of the light source, so any new `SuperCandle` created has a light value of 100 regular candles.

When you invoke `light()` or `extinguish()` on a `SuperCandle` object, Perl will look in the `SuperCandle` class for the method. If there's no method by that name in the child class, Perl will look at the parent class, then grandparent, and so on. In this case, those methods are in the `LightSource` class.

Attribute inheritance works similarly (see `perldoc Class::MOP`).

Method Dispatch Order

Perl's *dispatch* strategy controls how Perl selects the appropriate method to run for a method call. As you may have gathered from roles and polymorphism, much of OO's power comes from method dispatch.

Method dispatch order (or *method resolution order* or *MRO*) is obvious for single-parent classes. Look in the object's class, then its parent, and so on until you find the method–or run out of parents. Classes which inherit from multiple parents (*multiple inheritance*), such as a `Hovercraft` which extends both `Boat` and `Car`, require trickier dispatch. Reasoning about multiple inheritance is complex, so avoid multiple inheritance when possible.

Perl uses a depth-first method resolution strategy. It searches the class of the *first* named parent and all of that parent's parents recursively before searching the classes of the current class's immediate parents. The `mro` pragma (Pragmas, pp. 129) provides alternate strategies, including the C3 MRO strategy which searches a given class's immediate parents before searching any of their parents.

See `perldoc mro` for more details.

Inheritance and Methods

As with attributes, subclasses may override methods. Imagine a light that you cannot extinguish:

```
package Glowstick {
    use Moose;

    extends 'LightSource';

    sub extinguish {}
}
```

Calling `extinguish()` on a glowstick does nothing, even though `LightSource`'s method does something. Method dispatch will find the subclass's method. You may not have meant to do this. When you do, use Moose's `override` to express your intention clearly.

Within an overridden method, Moose's `super()` allows you to call the overridden method:

```
package LightSource::Cranky {
    use Carp 'carp';
    use Moose;

    extends 'LightSource';

    override light => sub {
        my $self = shift;

        carp "Can't light a lit LightSource!" if $self->enabled;

        super();
    };

    override extinguish => sub {
        my $self = shift;

        carp "Can't extinguish unlit LightSource!" unless $self->enabled;

        super();
    };
}
```

This subclass adds a warning when trying to light or extinguish a light source that already has the current state. The `super()` function dispatches to the nearest parent's implementation of the current method, per the normal Perl method resolution order. (See `perldoc Moose::Manual::MethodModifiers` for more dispatch options.)

Inheritance and isa()

Perl's `isa()` method returns true if its invocant is or extends a named class. That invocant may be the name of a class or an instance of an object:

```
say 'Looks like a LightSource' if $sconce->isa( 'LightSource' );

say 'Hominidae do not glow' unless $chimpy->isa( 'LightSource' );
```

Moose and Perl OO

Moose provides many features beyond Perl's default OO system. While you *can* build everything you get with Moose yourself (Blessed References, pp. 118), or cobble it together with a series of CPAN distributions, Moose is worth using. It is a coherent whole, with documentation, a mature and attentive development community, and a history of successful use in important projects.

Moose provides constructors, destructors, accessors, and encapsulation. You must do the work of declaring what you want, and you get safe and useful code in return. Moose objects can extend and work with objects from the vanilla Perl system.

While Moose is not a part of the Perl core, its popularity ensures that it's available on many OS distributions. Perl distributions such as Strawberry Perl and ActivePerl also include it. Even though Moose is a CPAN module and not a core library, its cleanliness and simplicity make it essential to modern Perl programming.

Moose also allows *metaprogramming*–manipulating your objects through Moose itself. If you've ever wondered which methods are available on a class or an object or which attributes an object supports, this information is available:

```
my $metaclass = Monkey::Pants->meta;
```

```
say 'Monkey::Pants instances have the attributes:';

say $_->name for $metaclass->get_all_attributes;

say 'Monkey::Pants instances support the methods:';

say $_->fully_qualified_name for $metaclass->get_all_methods;
```

You can even see which classes extend a given class:

```
my $metaclass = Monkey->meta;

say 'Monkey is the superclass of:';

say $_ for $metaclass->subclasses;
```

See `perldoc Class::MOP::Class` for more information about metaclass operations and `perldoc Class::MOP` for Moose metaprogramming information.

Moose and its *meta-object protocol* (or MOP) offers the possibility of a better syntax for declaring and working with classes and objects in Perl. This is valid code:

```
use MooseX::Declare;

role LivingBeing { requires qw( name age diet ) }

role CalculateAge::From::BirthYear {
    has 'birth_year',
        is      => 'ro',
        isa     => 'Int',
        default => sub { (localtime)[5] + 1900 };

    method age {
        return (localtime)[5] + 1900 - $self->birth_year;
    }
}

class Cat with LivingBeing with CalculateAge::From::BirthYear {
    has 'name', is => 'ro', isa => 'Str';
    has 'diet', is => 'rw';
}
```

The `MooseX::Declare` CPAN distribution adds the `class`, `role`, and `method` keywords to reduce the amount of boilerplate necessary to write good object oriented code in Perl. Note specifically the declarative nature of this example, as well as the lack of `my $self = shift;` in `age()`.

Another good option is `Moops`, which allows you to write:

```
use Moops;

role LivingBeing {
    requires qw( name age diet );
}
```

```
role CalculateAge::From::BirthYear :ro {
    has 'birth_year',
        isa     => Int,
        default => sub { (localtime)[5] + 1900 };

    method age {
        return (localtime)[5] + 1900 - $self->birth_year;
    }
}

class Cat with LivingBeing with CalculateAge::From::BirthYear :ro {
    has 'name', isa => Str;
    has 'diet', is => 'rw';
}
```

The Svelte *Alces*

Moose isn't a small library, but it's powerful. The most popular alternative is `Moo`, a slimmer library that's almost completely compatible with Moose. Many projects migrate some or all code to Moo where speed or memory use is an issue. Start with Moose, then see if Moo makes sense for you.

Blessed References

Perl's core object system is deliberately minimal. It has only three rules:

- A class is a package.
- A method is a function.
- A (blessed) reference is an object.

You can build anything else out of those three rules. This minimalism can be impractical for larger projects–in particular, the possibilities for greater abstraction through metaprogramming (Code Generation, pp. 150) are awkward and limited. Moose (Moose, pp. 107) is a better choice for modern programs larger than a couple of hundred lines, although plenty of legacy code uses Perl's default OO.

You've seen the first two rules already. The `bless` builtin associates the name of a class with a reference. That reference is now a valid invocant. Perl will perform method dispatch on it.

A constructor is a method which creates and blesses a reference. By convention, constructors are named `new()`. Constructors are also almost always *class methods*.

`bless` takes two operands, a reference and a class name, and evaluates to the reference. The reference may be any valid reference, empty or not. The class does not have to exist yet. You may even use `bless` outside of a constructor or a class, but you violate encapsulation to expose the details of object construction outside of a constructor. A constructor can be as simple as:

```
sub new {
    my $class = shift;
    bless {}, $class;
}
```

By design, this constructor receives the class name as the method's invocant. You may also hard-code the name of a class at the expense of flexibility. A parametric constructor–one which relies on the invocant to determine the class name–allows reuse through inheritance, delegation, or exporting.

The type of reference used is relevant only to how the object stores its own *instance data*. It has no other effect on the resulting object. Hash references are most common, but you can bless any type of reference:

```
my $array_obj  = bless [],          $class;
my $scalar_obj = bless \$scalar,    $class;
my $func_obj   = bless \&some_func, $class;
```

Moose classes define object attributes declaratively, but Perl's default OO is lax. A class representing basketball players which stores jersey number and position might use a constructor like:

```
package Player {
    sub new {
        my ($class, %attrs) = @_;
        bless \%attrs, $class;
    }
}
```

...and create players with:

```
my $joel   = Player->new( number => 10, position => 'center' );
my $damian = Player->new( number  => 0, position => 'guard'  );
```

The class's methods can access object attributes as hash elements directly:

```
sub format {
    my $self = shift;
    return '#'       . $self->{number}
         . ' plays ' . $self->{position};
}
```

...but so can any other code, so any change to the object's internal representation may break other code. Accessor methods are safer:

```
sub number   { return shift->{number}   }
sub position { return shift->{position} }
```

...and now you're starting to write yourself what Moose gives you for free. Better yet, Moose encourages people to use accessors instead of direct attribute access by generating the accessors itself. You won't see them in your code. Goodbye, temptation.

Method Lookup and Inheritance

Given a blessed reference, a method call of the form:

```
my $number = $joel->number;
```

...looks up the name of the class associated with the blessed reference `$joel`–in this case, `Player`. Next, Perl looks for a function named `number()` in `Player`. (Remember that Perl makes no distinction between functions in a namespace and methods.) If no such function exists and if `Player` extends a parent class, Perl looks in the parent class (and so on and so on) until it finds a `number()`. If Perl finds `number()`, it calls that method with `$joel` as an invocant. You've seen this before with Moose; it works the same way here.

Keeping Namespaces Clean

The `namespace::autoclean` CPAN module can help avoid unintentional collisions between imported functions and methods.

Moose provides `extends` to track inheritance relationships, but Perl uses a package global variable named `@ISA`. The method dispatcher looks in each class's `@ISA` to find the names of its parent classes. If `InjuredPlayer` extends `Player`, you might write:

```
package InjuredPlayer {
    @InjuredPlayer::ISA = 'Player';
}
```

The `parent` pragma (Pragmas, pp. 129) is cleaner:

```
package InjuredPlayer {
    use parent 'Player';
}
```

Moose has its own metamodel which stores extended inheritance information. This allows Moose to provide additional metaprogramming opportunities.

You may inherit from multiple parent classes:

```
package InjuredPlayer; {
    use parent qw( Player Hospital::Patient );
}
```

. . . though the caveats about multiple inheritance and method dispatch complexity apply. Consider instead roles (Roles, pp. 112) or Moose method modifiers.

AUTOLOAD

If there is no applicable method in the invocant's class or any of its superclasses, Perl will next look for an `AUTOLOAD()` function (AUTOLOAD, pp. 90) in every applicable class according to the selected method resolution order. Perl will invoke any `AUTOLOAD()` it finds.

In the case of multiple inheritance, `AUTOLOAD()` can be very difficult to understand.

Method Overriding and SUPER

As with Moose, you may override methods in basic Perl OO. Unlike Moose, Perl provides no mechanism for indicating your *intent* to override a parent's method. Worse yet, any function you predeclare, declare, or import into the child class may silently override a method in the parent class. Even if you forget to use Moose's `override` system, at least it exists. Basic Perl OO offers no such protection.

To override a parent method in a child class, declare a method of the same name. Within an overridden method, call the parent method with the `SUPER::` dispatch hint:

```
sub overridden {
    my $self = shift;
    warn 'Called overridden() in child!';
    return $self->SUPER::overridden( @_ );
}
```

The SUPER:: prefix to the method name tells the method dispatcher to dispatch to an overridden method of the appropriate name. You can provide your own arguments to the overridden method, but most code reuses @_. Be careful to shift off the invocant if you do.

The Brokenness of SUPER::

SUPER:: has a confusing misfeature: it dispatches to the parent of the package into which the overridden method was *compiled*. If you've imported this method from another package, Perl will happily dispatch to the *wrong* parent. The desire for backwards compatibility has kept this misfeature in place. The SUPER module from the CPAN offers a workaround. Moose's super() does not suffer the same problem.

Strategies for Coping with Blessed References

Blessed references may seem simultaneously minimal and confusing. Moose is much easier to use, so use it whenever possible. If you do find yourself maintaining code which uses blessed references, or if you can't convince your team to use Moose in full yet, you can work around some of the problems of blessed references with a few rules of thumb:

- Do not mix functions and methods in the same class.
- Use a single *.pm* file for each class, unless the class is a small, self-contained helper used from a single place.
- Follow Perl standards, such as naming constructors new() and using $self as the invocant name.
- Use accessor methods pervasively, even within methods in your class. A module such as Class::Accessor helps to avoid repetitive boilerplate.
- Avoid AUTOLOAD() where possible. If you *must* use it, use function forward declarations (Declaring Functions, pp. 67) to avoid ambiguity.
- Expect that someone, somewhere will eventually need to subclass (or delegate to or reimplement the interface of) your classes. Make it easier for them by not assuming details of the internals of your code, by using the two-argument form of bless, and by breaking your classes into the smallest responsible units of code.
- Use helper modules such as Role::Tiny to allow better use and reuse.

Reflection

Reflection (or *introspection*) is the process of asking a program about itself as it runs. By treating code as data you can manage code in the same way that you manage data. That sounds like a truism, but it's an important insight into modern programming. It's also a principle behind code generation (Code Generation, pp. 150).

Moose's Class::MOP (Class::MOP, pp. 153) simplifies many reflection tasks for object systems. Several other Perl idioms help you inspect and manipulate running programs.

Checking that a Module Has Loaded

If you know the name of a module, you can check that Perl believes it has loaded that module by looking in the %INC hash. When Perl loads code with use or require, it stores an entry in %INC where the key is the file path of the module to load and the value is the full path on disk to that module. In other words, loading Modern::Perl effectively does:

```
$INC{'Modern/Perl.pm'} = '.../lib/site_perl/5.22.1/Modern/Perl.pm';
```

The details of the path will vary depending on your installation. To test that Perl has successfully loaded a module, convert the name of the module into the canonical file form and test for that key's existence within %INC:

```
sub module_loaded {
    (my $modname = shift) =~ s!::!/!g;
    return exists $INC{ $modname . '.pm' };
}
```

As with `@INC`, any code anywhere may manipulate `%INC`. Some modules (such as `Test::MockObject` or `Test::MockModule`) manipulate `%INC` for good reasons. Depending on your paranoia level, you may check the path and the expected contents of the package yourself.

The `Class::Load` CPAN module's `is_class_loaded()` function does all of this for you without making you manipulate `%INC`.

Checking that a Package Exists

To check that a package exists somewhere in your program–if some code somewhere has executed a `package` directive with a given name–check that the package inherits from `UNIVERSAL`. Anything which extends `UNIVERSAL` must somehow provide the `can()` method (whether by inheriting it from `UNIVERSAL` or overriding it). If no such package exists, Perl will throw an exception about an invalid invocant, so wrap this call in an `eval` block:

```
say "$pkg exists" if eval { $pkg->can( 'can' ) };
```

An alternate approach is to grovel through Perl's symbol tables. You're on your own here.

Checking that a Class Exists

Because Perl makes no strong distinction between packages and classes, the best you can do without Moose is to check that a package of the expected class name exists. You *can* check that the package `can()` provide `new()`, but there is no guarantee that any `new()` found is either a method or a constructor.

Checking a Module Version Number

Modules do not have to provide version numbers, but every package inherits the `VERSION()` method from the universal parent class `UNIVERSAL` (The UNIVERSAL Package, pp. 148):

```
my $version = $module->VERSION;
```

`VERSION()` returns the given module's version number, if defined. Otherwise it returns `undef`. If the module does not exist, the method will likewise return `undef`.

Checking that a Function Exists

To check whether a function exists in a package, call `can()` as a class method on the package name:

```
say "$func() exists" if $pkg->can( $func );
```

Perl will throw an exception unless `$pkg` is a valid invocant; wrap the method call in an `eval` block if you have any doubts about its validity. Beware that a function implemented in terms of `AUTOLOAD()` (AUTOLOAD, pp. 90) may report the wrong answer if the function's package has not predeclared the function or overridden `can()` correctly. This is a bug in the other package.

Use this technique to determine if a module's `import()` has imported a function into the current namespace:

```
say "$func() imported!" if __PACKAGE__->can( $func );
```

As with checking for the existence of a package, you *can* root around in symbol tables yourself, if you have the patience for it.

Checking that a Method Exists

There is no foolproof way for reflection to distinguish between a function or a method.

Rooting Around in Symbol Tables

A *symbol table* is a special type of hash where the keys are the names of package global symbols and the values are typeglobs. A *typeglob* is an internal data structure which can contain a scalar, an array, a hash, a filehandle, and a function–any or all at once.

Access a symbol table as a hash by appending double-colons to the name of the package. For example, the symbol table for the `MonkeyGrinder` package is available as `%MonkeyGrinder::`.

You *can* test the existence of specific symbol names within a symbol table with the `exists` operator (or manipulate the symbol table to *add* or *remove* symbols, if you like). Yet be aware that certain changes to the Perl core have modified the details of what typeglobs store and when and why.

See the "Symbol Tables" section in `perldoc perlmod` for more details, then consider the other techniques explained earlier instead. If you really need to manipulate symbol tables and typeglobs, use the `Package::Stash` CPAN module.

Advanced OO Perl

Creating and using objects in Perl with Moose (Moose, pp. 107) is easy. *Designing* good programs is not. It's as easy to overdesign a program as it is to underdesign it. Only practical experience can help you understand the most important design techniques, but several principles can guide you.

Favor Composition Over Inheritance

Novice OO designs often overuse inheritance to reuse code and to exploit polymorphism. The result is a deep class hierarchy with responsibilities scattered all over the place. Maintaining this code is difficult–who knows where to add or edit behavior? What happens when code in one place conflicts with code declared elsewhere?

Inheritance is only one of many tools for OO programmers. It's not always the right tool. It's often the wrong tool. A `Car` may extend `Vehicle::Wheeled` (an *is-a relationship*), but `Car` may better *contain* several `Wheel` objects as instance attributes (a *has-a relationship*).

Decomposing complex classes into smaller, focused entities improves encapsulation and reduces the possibility that any one class or role does too much. Smaller, simpler, and better encapsulated entities are easier to understand, test, and maintain.

Single Responsibility Principle

When you design your object system, consider the responsibilities of each entity. For example, an `Employee` object may represent specific information about a person's name, contact information, and other personal data, while a `Job` object may represent business responsibilities. Separating these entities in terms of their responsibilities allows the `Employee` class to consider only the problem of managing information specific to who the person is and the `Job` class to represent what the person does. (Two `Employees` may have a `Job`-sharing arrangement, for example, or one `Employee` may have the CFO and the COO `Jobs`.)

When each class has a single responsibility, you reduce coupling between classes and improve the encapsulation of class-specific data and behavior.

Don't Repeat Yourself

Complexity and duplication complicate development and maintenance. The *DRY* principle (Don't Repeat Yourself) is a reminder to seek out and to eliminate duplication within the system. Duplication exists in data as well as in code. Instead of repeating configuration information, user data, and other important artifacts of your system, create a single, canonical representation of that information from which you can generate the other artifacts.

This principle helps you to find the optimal representation of your system and its data and reduces the possibility that duplicate information will get out of sync.

Liskov Substitution Principle

The Liskov substitution principle suggests that you should be able to substitute a specialization of a class or a role for the original without violating the original's API. In other words, an object should be as or more general with regard to what it expects and at least as specific about what it produces as the object it replaces.

Imagine two classes, `Dessert` and its child class `PecanPie`. If the classes follow the Liskov substitution principle, you can replace every use of `Dessert` objects with `PecanPie` objects in the test suite, and everything should pass. See Reg Braithwaite's "IS-STRICTLY-EQUIVALENT-TO-A"[2] for more details.

Subtypes and Coercions

Moose allows you to declare and use types and extend them through subtypes to form ever more specialized descriptions of what your data represents and how it behaves. These type annotations help verify that the function and method parameters are correct–or can be coerced into the proper data types.

For example, you may wish to allow people to provide dates to a `Ledger` entry as strings while representing them as `DateTime` instances internally. You can do this by creating a Date type and adding a coercion from string types. See `Moose::Util::TypeConstraint` and `MooseX::Types` for more information.

Immutability

With a well-designed object, you tell it *what to do*, not *how to do it*. If you find yourself accessing object instance data (even through accessor methods) outside of the object itself, you may have too much access to an object's internals.

OO novices often treat objects as if they were bundles of records which use methods to get and set internal values. This simple technique leads to the unfortunate temptation to spread the object's responsibilities throughout the entire system.

You can prevent inappropriate access by making your objects immutable. Provide the necessary data to their constructors, then disallow any modifications of this information from outside the class. Expose no methods to mutate instance data–make all of your public accessors read-only and use internal attribute writers sparingly. Once you've constructed such an object, you know it's always in a valid state. You can never modify its data to put it in an invalid state.

This takes tremendous discipline, but the resulting systems are robust, testable, and maintainable. Some designs go as far as to prohibit the modification of instance data *within* the class itself.

[2] http://weblog.raganwald.com/2008/04/is-strictly-equivalent-to.html.

CHAPTER 8

Style and Efficacy

To program well, we must find the balance between getting the job done now and doing the job right. We must balance time, resources, and quality. Programs have bugs. Programs need maintenance and expansion. Programs have multiple programmers. A beautiful program that never delivers value is worthless, but an awful program that cannot be maintained is a risk waiting to happen.

Skilled programmers understand their constraints and write the right code.

To write Perl well, you must understand the language. You must also cultivate a sense of good taste for the language and the design of programs. The only way to do so is to practice–not just writing code, but maintaining and reading good code.

This path has no shortcuts, but it does have guideposts.

Writing Maintainable Perl

Maintainability is the nebulous measurement of how easy it is to understand and modify a program. Write some code. Come back to it in six months (or six days). How long does it take you to find and fix a bug or add a feature? That's maintainability.

Maintainability doesn't measure whether you have to look up the syntax for a builtin or a library function. It doesn't measure how someone who has never programmed before will or won't read your code. Assume you're talking to a competent programmer who understands the problem you're trying to solve. How much work does she have to put in to understand your code? What problems will she face in doing so?

To write maintainable software, you need experience solving real problems, an understanding of the idioms and techniques and style of your programming language, and good taste. You can develop all of these by concentrating on a few principles.

- *Remove duplication.* Bugs lurk in sections of repeated and similar code–when you fix a bug in one piece of code, did you fix it in others? When you updated one section, did you update the others?Well-designed systems have little duplication. They use functions, modules, objects, and roles to extract duplicate code into distinct components which accurately model the domain of the problem. The best designs sometimes allow you to add features by *removing* code.

- *Name entities well.* Your code tells a story. Every name you choose for a variable, function, module, class, and role allows you to clarify or obfuscate your intent. Choose your names carefully. If you're having trouble choosing good names, you may need to rethink your design or study your problem in more detail.

- *Avoid unnecessary cleverness.* Concise code is good, when it reveals the intention of the code. Clever code hides your intent behind flashy tricks. Perl allows you to write the right code at the right time. Choose the most obvious solution when possible. Experience and good taste will guide you.Some problems require clever solutions. When this happens, encapsulate this code behind a simple interface and document your cleverness.

- *Embrace simplicity.* If everything else is equal, a simpler program is easier to maintain than a complex program. Simplicity means knowing what's most important and doing just that.

Sometimes you need powerful, robust code. Sometimes you need a one-liner. Simplicity means building only what you need. This is no excuse to avoid error checking or modularity or validation or security. Simple code can use advanced features. Simple code can use CPAN modules, and many of them. Simple code may require work to understand. Yet simple code solves problems effectively, without *unnecessary* work.

Writing Idiomatic Perl

Perl borrows liberally from other languages. Perl lets you write the code you want to write. C programmers often write C-style Perl, just as Java programmers write Java-style Perl and Lisp programmers write Lispy Perl. Effective Perl programmers write Perlish Perl by embracing the language's idioms.

- *Understand community wisdom.* Perl programmers often debate techniques and idioms fiercely. Perl programmers also often share their work, and not just on the CPAN. Pay attention; there's not always one and only one best way to do things. The interesting discussions happen about the tradeoffs between various ideals and styles.
- *Follow community norms.* Perl is a community of toolsmiths who solve broad problems, including static code analysis (`Perl::Critic`), reformatting (`Perl::Tidy`), and private distribution systems (`CPAN::Mini`, `Carton`, `Pinto`). Take advantage of the CPAN infrastructure; follow the CPAN model of writing, documenting, packaging, testing, and distributing your code.
- *Read code.* Join a mailing list such as Perl Beginners (http://learn.perl.org/faq/beginners.html) and otherwise immerse yourself in the community[1]. Read code and try to answer questions–even if you never post your answers, writing code to solve one problem every work day will teach you an enormous amount very quickly.

CPAN developers, Perl Mongers, and mailing list participants have hard-won experience solving problems in myriad ways. Talk to them. Read their code. Ask questions. Learn from them and let them guide–and learn from–you.

Writing Effective Perl

Writing maintainable code means designing maintainable code. Good design comes from good habits.

- *Write testable code.* Writing an effective test suite (Testing, pp. 131) exercises the same design skills as writing effective code. Code is code. Good tests also give you the confidence to modify a program while keeping it running correctly.
- *Modularize.* Enforce encapsulation and abstraction boundaries. Find the right interfaces between components. Name things well and put them where they belong. Modularity forces you to think about similarities and differences and points of communication where your design fits together. Find the pieces that don't fit well. Revise your design until they do fit.
- *Follow sensible coding standards.* Effective guidelines discuss error handling, security, encapsulation, API design, project layout, and other facets of maintainable code. Excellent guidelines help developers communicate with each other with code. If you look at a new project and nothing surprises you, that's great! Your job is to solve problems with code. Let your code–and the infrastructure around it–speak clearly.
- *Exploit the CPAN.* Perl programmers solve problems, then share those solutions. The CPAN is a force multiplier; search it first for a solution or partial solution to your problem. Invest time in research to find full or partial solutions you can reuse. It will pay off.

If you find a bug, report it. Patch it, if possible. Submit a failing test case. Fix a typo. Ask for a feature. Say “Thank you!” Then, when you're ready, When you're ready–when you create something new or fix something old in a reusable way–share your code.

Exceptions

Good programmers anticipate the unexpected. Files that should exist won't. A huge disk that should never fill up will. The network that never goes down stops responding. The unbreakable database crashes and eats a table.

The unexpected happens.

Perl handles exceptional conditions through *exceptions*: a dynamically-scoped control flow mechanism designed to raise and handle errors. Robust software must handle them. If you can recover, great! If you can't, log the relevant information and retry.

[1] http://www.perl.org/community.html

Throwing Exceptions

Suppose you want to write a log file. If you can't open the file, something has gone wrong. Use `die` to throw an exception (or see The autodie Pragma, pp. 178):

```
sub open_log_file {
    my $name = shift;
    open my $fh, '>>', $name or die "Can't open log to '$name': $!";
    return $fh;
}
```

`die()` sets the global variable `$@` to its operand and immediately exits the current function *without returning anything*. This is known as throwing an exception. A thrown exception will continue up the call stack (Controlled Execution, pp. 161) until something catches it. If nothing catches the exception, the program will exit with an error.

Exception handling uses the same dynamic scope (Dynamic Scope, pp. 79) as `local` symbols.

Catching Exceptions

Sometimes allowing an exception to end the program is useful. A program run from a timed process might throw an exception when the error logs are full, causing an SMS to go out to administrators. Other exceptions might not be fatal–your program might be able to recover from one. Another might give you a chance to save the user's work and exit cleanly.

Use the block form of the `eval` operator to catch an exception:

```
# log file may not open
my $fh = eval { open_log_file( 'monkeytown.log' ) };
```

If the file open succeeds, `$fh` will contain the filehandle. If it fails, `$fh` will remain undefined and program flow will continue.

The block argument to `eval` introduces a new scope, both lexical and dynamic. If `open_log_file()` called other functions and something eventually threw an exception, this `eval` could catch it.

An exception handler is a blunt tool. It will catch all exceptions thrown in its dynamic scope. To check which exception you've caught (or if you've caught an exception at all), check the value of `$@`. Be sure to `localize` `$@` before you attempt to catch an exception, as `$@` is a global variable:

```
local $@;

# log file may not open
my $fh = eval { open_log_file( 'monkeytown.log' ) };

# caught exception
if (my $exception = $@) { ... }
```

Copy `$@` to a lexical variable immediately to avoid the possibility of subsequent code clobbering the global variable `$@`. You never know what else has used an `eval` block elsewhere and reset `$@`.

`$@` usually contains a string describing the exception. Inspect its contents to see whether you can handle the exception:

```
if (my $exception = $@) {
    die $exception unless $exception =~ /^Can't open logging/;
    $fh = log_to_syslog();
}
```

Rethrow an exception by calling `die()` again. Pass the existing exception or a new one as necessary.

Applying regular expressions to string exceptions can be fragile, because error messages may change over time. This includes the core exceptions that Perl itself throws. Instead of throwing an exception as a string, you may use a reference–even a blessed

reference–with `die`. This allows you to provide much more information in your exception: line numbers, files, and other debugging information. Retrieving information from a data structure is much easier than parsing data out of a string. Catch these exceptions as you would any other exception.

The CPAN distribution `Exception::Class` makes creating and using exception objects easy:

```
package Zoo::Exceptions {
    use Exception::Class
        'Zoo::AnimalEscaped',
        'Zoo::HandlerEscaped';
}

sub cage_open {
    my $self = shift;

    Zoo::AnimalEscaped->throw unless $self->contains_animal;
    ...
}

sub breakroom_open {
    my $self = shift;
    Zoo::HandlerEscaped->throw unless $self->contains_handler;
    ...
}
```

Another fine option is `Throwable::Error`.

Exception Caveats

Though throwing exceptions is simple, catching them is less so. Using `$@` correctly requires you to navigate several subtle risks:

- Unlocalized uses in the same or a nested dynamic scope may modify `$@`
- `$@` may contain an object which returns a false value in boolean context
- A signal handler (especially the `DIE` signal handler) may change `$@`
- The destruction of an object during scope exit may call `eval` and change `$@`

Modern Perl has fixed some of these issues. Though they rarely occur, they're difficult to diagnose. The `Try::Tiny` CPAN distribution improves the safety of exception handling *and* the syntax:

```
use Try::Tiny;

my $fh = try   { open_log_file( 'monkeytown.log' ) }
         catch { log_exception( $_ ) };
```

`try` replaces `eval`. The optional `catch` block executes only when `try` catches an exception. `catch` receives the caught exception as the topic variable `$_`.

Built-in Exceptions

Perl itself throws several exceptional conditions. `perldoc perldiag` lists several "trappable fatal errors". Some are syntax errors that Perl produces during failed compilations, but you can catch the others during runtime. The most interesting are:

- Using a disallowed key in a locked hash (Locking Hashes, pp. 49)

- Blessing a non-reference (Blessed References, pp. 118)
- Calling a method on an invalid invocant (Moose, pp. 107)
- Failing to find a method of the given name on the invocant
- Using a tainted value in an unsafe fashion (Taint, pp. 156)
- Modifying a read-only value
- Performing an invalid operation on a reference (References, pp. 53)

You can also catch exceptions produced by `autodie` (The autodie Pragma, pp. 178) and any lexical warnings promoted to exceptions (Registering Your Own Warnings, pp. 137).

Pragmas

Most Perl modules provide new functions or define classes (Moose, pp. 107). Others, such as `strict` or `warnings`, influence the behavior of the language itself. This second type of module is a *pragma*. By convention, pragma names are written in lower-case to differentiate them from other modules.

Pragmas and Scope

Pragmas work by exporting specific behavior or information into the lexical scopes of their callers. You've seen how declaring a lexical variable makes a symbol name available within a scope. Using a pragma makes its behavior effective within a scope as well:

```
{
    # $lexical not visible; strict not in effect
    {
        use strict;
        my $lexical = 'available here';
        # $lexical is visible; strict is in effect
    }
    # $lexical again invisible; strict not in effect
}
```

Just as lexical declarations affect inner scopes, pragmas maintain their effects within inner scopes:

```
# file scope
use strict;

{
    # inner scope, but strict still in effect
    my $inner = 'another lexical';
}
```

Using Pragmas

`use` a pragma as you would any other module. Pragmas may take arguments, such as a minimum version number to use or a list of arguments to change their behaviors:

```
# require variable declarations, prohibit barewords
use strict qw( subs vars );

# rely on the semantics of the 2014 book
use Modern::Perl '2014';
```

Sometimes you need to *disable* all or part of those effects within a further nested lexical scope. The `no` builtin performs an unimport (Importing, pp. 72), which reverses some or all effects of a well-behaved pragma. For example, to disable the protection of `strict` when you need to do something symbolic:

```
use Modern::Perl; # or use strict;

{
    no strict 'refs';
    # manipulate the symbol table here
}
```

Useful Pragmas

Perl includes several useful core pragmas.

- the `strict` pragma enables compiler checking of symbolic references, bareword use, and variable declaration.
- the `warnings` pragma enables optional warnings for deprecated, unintended, and awkward behaviors.
- the `utf8` pragma tells Perl's parser to understand the source code of the current file with the UTF-8 encoding.
- the `autodie` pragma enables automatic error checking of system calls and builtins.
- the `constant` pragma allows you to create compile-time constant values (though see the CPAN's `Const::Fast` for an alternative).
- the `vars` pragma allows you to declare package global variables, such as `$VERSION` or `@ISA` (Blessed References, pp. 118).
- the `feature` pragma allows you to enable and disable newer features of Perl individually. Where `use 5.18;` enables all of the Perl 5.18 features and the `strict` pragma, `use feature ':5.18';` does the same. This pragma is more useful to *disable* individual features in a lexical scope.
- the `experimental` pragma enables or disables experimental features such as signatures (Real Function Signatures, pp. 69) or postfix dereferencing.
- the `less` pragma demonstrates how to write a pragma.

As you might suspect from `less`, you can write your own lexical pragmas in pure Perl. `perldoc perlpragma` explains how to do so, while the explanation of `$^H` in `perldoc perlvar` explains how the feature works.

The CPAN has begun to gather non-core pragmas:

- `autovivification` disables autovivification (Autovivification, pp. 60)
- `indirect` prevents the use of indirect invocation (Indirect Objects, pp. 168)
- `autobox` enables object-like behavior for Perl's core types (scalars, references, arrays, and hashes).
- `perl5i` combines and enables many experimental language extensions into a coherent whole.

These tools are not widely used yet, but they have their champions. `autovivification` and `indirect` can help you write more correct code. `autobox` and `perl5i` are experiments with what Perl might one day become; they're worth playing with in small projects.

CHAPTER 9

Managing Real Programs

You can learn a lot of syntax from a book by writing small programs to solve the example problems. Writing good code to solve real problems takes more discipline and understanding. You must learn to *manage* code. How do you know that it works? How do you organize it? What makes it robust in the face of errors? What makes code clean? Clear? Maintainable?

Modern Perl helps you answer all those questions.

Testing

You've already tested your code.

If you've ever run it, noticed something wasn't quite right, made a change, and then ran it again, you've tested your code. *Testing* is the process of verifying that your software behaves as intended. Effective testing automates that process. Rather than relying on humans to perform repeated manual checks perfectly, let the computer do it.

Perl's tools help you write the right tests.

Test::More

The fundamental unit of testing is a test assertion. Every test *assertion* is a simple question with a yes or no answer: does this code behave as I intended? Any condition you can test in your program can (eventually) become one or more assertions. A complex program may have thousands of individual conditions. That's fine. That's testable. Isolating specific behaviors into individual assertions helps you debug errors of coding and errors of understanding, and it makes your code and tests easier to maintain.

Perl testing begins with the core module `Test::More` and its `ok()` function. `ok()` takes two parameters, a boolean value and a string which describes the test's purpose:

```
ok   1, 'the number one should be true';
ok   0, '... and zero should not';
ok  '', 'the empty string should be false';
ok '!', '... and a non-empty string should not';

done_testing();
```

The function `done_testing()` tells `Test::More` that the program has executed all of the assertions you expected to run. If the program exited unexpectedly (from an uncaught exception, a call to `exit`, or whatever), the test framework will notify you that something went wrong. Without a mechanism like `done_testing()`, how would you *know*? While this example code is too simple to fail, code that's too simple to fail fails far more often than you want.

`Test::More` allows an optional *test plan* to count the number of individual assertions you plan to run:

```
use Test::More tests => 4;

ok   1, 'the number one should be true';
ok   0, '... and zero should not';
ok  '', 'the empty string should be false';
ok '!', '... and a non-empty string should not';
```

The `tests` argument to `Test::More` sets the test plan for the program. This is a safety net. If fewer than four tests ran, something went wrong. If more than four tests ran, something went wrong. `done_testing()` is easier, but sometimes an exact count can be useful (when you want to control the number of assertions in a loop, for example).

Running Tests

This example test file is a complete Perl program which produces the output:

```
ok 1 - the number one should be true
not ok 2 - ... and zero should not
#   Failed test '... and zero should not'
#   at truth_values.t line 4.
not ok 3 - the empty string should be false
#   Failed test 'the empty string should be false'
#   at truth_values.t line 5.
ok 4 - ... and a non-empty string should not
1..4
# Looks like you failed 2 tests of 4.
```

This output uses a test output format called *TAP*, the *Test Anything Protocol*[1]. Failed TAP tests produce diagnostic messages for debugging purposes.

This is easy enough to read, but it's only four assertions. A real program may have thousands of assertions. In most cases, you want to know either that everything passed or the specifics of any failures. The core module The program `prove`–built on the core module `TAP::Harness`– runs tests, interprets TAP, and displays only the most pertinent information:

```
$ prove truth_values.t
truth_values.t .. 1/?
#   Failed test '... and zero should not'
#   at truth_values.t line 4.

#   Failed test 'the empty string should be false'
#   at truth_values.t line 5.
# Looks like you failed 2 tests of 4.
truth_values.t .. Dubious, test returned 2 (wstat 512, 0x200)
Failed 2/4 subtests

Test Summary Report
-------------------
truth_values.t (Wstat: 512 Tests: 4 Failed: 2)
  Failed tests:  2-3
```

That's a lot of output to display what is already obvious: the second and third tests fail because zero and the empty string evaluate to false. Fortunately, it's easy to fix those failing tests (Boolean Coercion, pp. 49):

```
ok   ! 0, '... and zero should not';
ok  ! '', 'the empty string should be false';
```

With those two changes, `prove` now displays:

```
$ prove truth_values.t
truth_values.t .. ok
All tests successful.
```

[1] http://testanything.org/

See `perldoc prove` for other test options, such as running tests in parallel (`-j`), automatically adding the relative directory *lib/* to Perl's include path (`-l`), recursively running all test files found under *t/* (`-r t`), and running slow tests first (`--state=slow,save`).

The Bash shell alias `proveall` combines many of these options:

```
alias proveall='prove -j9 --state=slow,save -lr t'
```

Better Comparisons

Even though the heart of all automated testing is the boolean condition "is this true or false?", reducing everything to that boolean condition is tedious and produces awkward diagnostics. `Test::More` provides several other convenient assertion functions.

The `is()` function compares two values using Perl's `eq` operator. If the values are equal, the test passes:

```
is         4, 2 + 2, 'addition should work';
is 'pancake',   100, 'pancakes are numeric';
```

The first test passes and the second fails with a diagnostic message:

```
t/is_tests.t .. 1/2
#   Failed test 'pancakes are numeric'
#   at t/is_tests.t line 8.
#          got: 'pancake'
#     expected: '100'
# Looks like you failed 1 test of 2.
```

Where `ok()` only provides the line number of the failing test, `is()` displays the expected and received values.

`is()` applies implicit scalar context to its values (Prototypes, pp. 170). This means, for example, that you can check the number of elements in an array without explicitly evaluating the array in scalar context, and it's why you can omit the parentheses:

```
my @cousins = qw( Rick Kristen Alex Kaycee Eric Corey );
is @cousins, 6, 'I should have only six cousins';
```

...though some people prefer to write `scalar @cousins` for the sake of clarity.

`Test::More`'s corresponding `isnt()` function compares two values using the `ne` operator and passes if they are not equal. It also provides scalar context to its operands.

Both `is()` and `isnt()` apply *string comparisons* with the `eq` and `ne` operators. This almost always does the right thing, but for strict numeric comparisons or complex values such as objects with overloading (Overloading, pp. 154) or dual vars (Dualvars, pp. 51), use the `cmp_ok()` function. This function takes the first value to compare, a comparison operator, and the second value to compare:

```
cmp_ok     100, '<=', $cur_balance, 'I should have at least $100';
cmp_ok $monkey, '==', $ape, 'Simian numifications should agree';
```

If you're concerned about string equality with numeric comparisons–a reasonable concern–then use `cmp_ok()` instead of `is()`.

Classes and objects provide their own interesting ways to interact with tests. Test that a class or object extends another class (Inheritance, pp. 114) with `isa_ok()`:

```
my $chimpzilla = RobotMonkey->new;

isa_ok $chimpzilla, 'Robot';
isa_ok $chimpzilla, 'Monkey';
```

`isa_ok()` provides its own diagnostic message on failure.

`can_ok()` verifies that a class or object can perform the requested method (or methods):

```
can_ok $chimpzilla, 'eat_banana';
can_ok $chimpzilla, 'transform', 'destroy_tokyo';
```

The `is_deeply()` function compares two references to ensure that their contents are equal:

```
use Clone;

my $numbers   = [ 4, 8, 15, 16, 23, 42 ];
my $clonenums = Clone::clone( $numbers );

is_deeply $numbers, $clonenums, 'clone() should produce identical items';
```

If the comparison fails, `Test::More` will do its best to provide a reasonable diagnostic indicating the position of the first inequality between the structures. See the CPAN modules `Test::Differences` and `Test::Deep` for more configurable tests.

`Test::More` has several other more specialized test functions.

Organizing Tests

CPAN distributions should include a *t/* directory containing one or more test files named with the *.t* suffix. When you build a distribution, the testing step runs all of the *t/*.t* files, summarizes their output, and succeeds or fails based on the results of the test suite as a whole. Two organization strategies are popular:

- Each *.t* file should correspond to a *.pm* file
- Each *.t* file should correspond to a logical feature

A hybrid approach is the most flexible; one test can verify that all of your modules compile, while other tests demonstrate that each module behaves as intended. As your project grows, the second approach is easier to manage. Keep your test files small and focused and they'll be easier to maintain.

Separate test files can also speed up development. If you're adding the ability to breathe fire to your `RobotMonkey`, you may want only to run the *t/robot_monkey/breathe_fire.t* test file. When you have the feature working to your satisfaction, run the entire test suite to verify that local changes have no unintended global effects.

Other Testing Modules

`Test::More` relies on a testing backend known as `Test::Builder` which manages the test plan and coordinates the test output into TAP. This design allows multiple test modules to share the same `Test::Builder` backend. Consequently, the CPAN has hundreds of test modules available–and they can all work together in the same program.

- `Test::Fatal` helps test that your code throws (and does not throw) exceptions appropriately. You may also encounter `Test::Exception`.
- `Test::MockObject` and `Test::MockModule` allow you to test difficult interfaces by *mocking* (emulating behavior to produce controlled results).
- `Test::WWW::Mechanize` helps test web applications, while `Plack::Test`, `Plack::Test::Agent`, and the subclass `Test::WWW::Mechanize::PSGI` can do so without using an external live web server.
- `Test::Database` provides functions to test the use and abuse of databases. `DBICx::TestDatabase` helps test schemas built with `DBIx::Class`.

- `Test::Class` offers an alternate mechanism for organizing test suites. It allows you to create classes in which specific methods group tests. You can inherit from test classes just as your code classes inherit from each other. This is an excellent way to reduce duplication in test suites. See Curtis Poe's excellent `Test::Class` series[2]. The newer `Test::Routine` distribution offers similar possibilities through the use of Moose (Moose, pp. 107).
- `Test::Differences` tests strings and data structures for equality and displays any differences in its diagnostics. `Test::LongStrin` adds similar assertions.
- `Test::Deep` tests the equivalence of nested data structures (Nested Data Structures, pp. 58).
- `Devel::Cover` analyzes the execution of your test suite to report on the amount of your code your tests actually exercises. In general, the more coverage the better–although 100% coverage is not always possible, 95% is far better than 80%.
- `Test::Most` gathers several useful test modules into one parent module. It saves time and effort.

See the Perl QA project[3] for more information about testing in Perl.

Handling Warnings

While there's more than one way to write a working Perl program, some of those ways can be confusing, unclear, and even incorrect. Perl's warnings system can help you avoid these situations.

Producing Warnings

Use the `warn` builtin to emit a warning:

```
warn 'Something went wrong!';
```

`warn` prints a list of values to the STDERR filehandle (Input and Output, pp. 138). Perl will append the filename and line number of the `warn` call unless the last element of the list ends in a newline.

The core `Carp` module extends Perl's warning mechanisms. Its `carp()` function reports a warning from the perspective of the calling code. Given a function like:

```
use Carp 'carp';

sub only_two_arguments {
    my ($lop, $rop) = @_;
    carp( 'Too many arguments provided' ) if @_ > 2;
    ...
}
```

. . . the arity (Arity, pp. 64) warning will include the filename and line number of the *calling* code, not `only_two_arguments()`. `Carp`'s `cluck()` is similar, but it produces a backtrace of *all* function calls which led to the current function.

`Carp`'s verbose mode adds backtraces to all warnings produced by `carp()` and `croak()` (Reporting Errors, pp. 73) throughout the entire program:

```
$ perl -MCarp=verbose my_prog.pl
```

Use `Carp` when writing modules (Modules, pp. 143) instead of `warn` or `die`.

[2] http://www.modernperlbooks.com/mt/2009/03/organizing-test-suites-with-testclass.html

[3] http://qa.perl.org/

Injecting Carp

Sometimes you'll have to debug code written without the use of `carp()` or `cluck()`. In that case, use the `Carp::Always` module to add backtraces to all `warn` or `die` calls: `perl -MCarp::Always some_program.pl`.

Enabling and Disabling Warnings

The venerable `-w` command-line flag enables warnings throughout the program, even in external modules written and maintained by other people. It's all or nothing–though it can help you if you have the time and energy to eliminate warnings and potential warnings throughout the entire codebase. This was the only way to enable warnings in Perl programs for many years.

The modern approach is to use the `warnings` pragma (or an equivalent such as `use Modern::Perl;`). This enables warnings in *lexical* scopes. If you've used `warnings` in a scope, you're indicating that the code should not normally produce warnings.

Global Warnings Flags

The `-W` flag enables warnings throughout the program unilaterally, regardless of any use of `warnings`. The `-X` flag *disables* warnings throughout the program unilaterally. Neither is common.

All of `-w`, `-W`, and `-X` affect the value of the global variable `$^W`. Code written before the `warnings` pragma came about in spring 2000 may `localize` `$^W` to suppress certain warnings within a given scope.

Disabling Warning Categories

Use `no warnings;` with an argument list to disable selective warnings within a scope. Omitting the argument list disables all warnings within that scope.

`perldoc perllexwarn` lists all of the warnings categories your version of Perl understands. Most of them represent truly interesting conditions, but some may be actively unhelpful in your specific circumstances. For example, the `recursion` warning will occur if Perl detects that a function has called itself more than a hundred times. If you are confident in your ability to write recursion-ending conditions, you may disable this warning within the scope of the recursion–though tail calls may be better (Tail Calls, pp. 76).

If you're generating code (Code Generation, pp. 150) or locally redefining symbols, you may wish to disable the `redefine` warnings.

Some experienced Perl hackers disable the `uninitialized` value warnings in string-processing code which concatenates values from many sources. If you're careful about initializing your variables, you may never need to disable this warning, but sometimes the warning gets in the way of writing concise code in your local style.

Making Warnings Fatal

If your project considers warnings as onerous as errors, you can make them fatal. To promote *all* warnings into exceptions within a lexical scope:

```
use warnings FATAL => 'all';
```

You may also make specific categories of warnings fatal, such as the use of deprecated constructs:

```
use warnings FATAL => 'deprecated';
```

With proper discipline, this can produce very robust code–but be cautious. Many warnings come from conditions that Perl can detect only when your code is *running*. If your test suite fails to identify all of the warnings you might encounter, fatalizing

these warnings may cause your program to crash. Newer versions of Perl often add new warnings. Upgrading to a new version without careful testing might cause new exceptional conditions. More than that, any custom warnings you or the libraries you use will also be fatal (Registering Your Own Warnings, pp. 137).

If you enable fatal warnings, do so only in code that you control and never in library code you expect other people to use.

Catching Warnings

If you're willing to work for it, you can catch warnings as you would exceptions. The `%SIG` variable (see `perldoc perlvar`) contains handlers for out-of-band signals raised by Perl or your operating system. Assign a function reference to `$SIG{__WARN__}` to catch a warning:

```
{
    my $warning;
    local $SIG{__WARN__} = sub { $warning .= shift };

    # do something risky
    ...

    say "Caught warning:\n$warning" if $warning;
}
```

Within the warning handler, the first argument is the warning's message. Admittedly, this technique is less useful than disabling warnings lexically–but it can come to good use in test modules such as `Test::Warnings` from the CPAN, where the actual text of the warning is important.

`%SIG` is a global variable, so `local`ize it in the smallest possible scope.

Registering Your Own Warnings

The `warnings::register` pragma allows you to create your own warnings which users can enable and disable lexically. From a module, `use` the pragma:

```
package Scary::Monkey;

use warnings::register;
```

This will create a new warnings category named after the package `Scary::Monkey`. Enable these warnings with `use warnings 'Scary::Monkey'` and disable them with `no warnings 'Scary::Monkey'`.

Use `warnings::enabled()` to test if the caller's lexical scope has enabled a warning category. Use `warnings::warnif()` to produce a warning only if warnings are in effect. For example, to produce a warning in the `deprecated` category:

```
package Scary::Monkey;

use warnings::register;

sub import {
    warnings::warnif( 'deprecated',
        'empty imports from ' . __PACKAGE__ . ' are now deprecated'
    ) unless @_;
}
```

See `perldoc perllexwarn` for more details.

Files

Most programs interact with the real world mostly by reading, writing, and otherwise manipulating files. Perl began as a tool for system administrators and is still a language well suited for text processing.

Input and Output

A *filehandle* represents the current state of one specific channel of input or output. Every Perl program starts with three standard filehandles, `STDIN` (the input to the program), `STDOUT` (the output from the program), and `STDERR` (the error output from the program). By default, everything you `print` or `say` goes to `STDOUT`, while errors and warnings go to `STDERR`. This separation of output allows you to redirect useful output and errors to two different places–an output file and error logs, for example.

Use the `open` builtin to initialize a filehandle. To open a file for reading:

```
open my $fh, '<', 'filename' or die "Cannot read '$filename': $!\n";
```

The first operand is a lexical which will contain the filehandle. The second operand is the *file mode*, which determines the type of file operation (reading, writing, appending, et cetera). The final operand is the name of the file on which to operate. If the `open` fails, the `die` clause will throw an exception, with the reason for failure in the `$!` magic variable.

You may open files for writing, appending, reading and writing, and more. Some of the most important file modes are:

- `<`, which opens a file for reading.
- `>`, which open for writing, clobbering existing contents if the file exists and creating a new file otherwise.
- `>>`, which opens a file for writing, appending to any existing contents and creating a new file otherwise.
- `+<`, which opens a file for both reading and writing.
- `-|`, which opens a pipe to an external process for reading.
- `|-`, which opens a pipe to an external process for writing.

You may also create filehandles which read from or write to plain Perl scalars, using any existing file mode:

```
open my $read_fh,  '<', \$fake_input;
open my $write_fh, '>', \$captured_output;

do_something_awesome( $read_fh, $write_fh );
```

`perldoc perlopentut` explains in detail more exotic uses of `open`, including its ability to launch and control other processes, as well as the use of `sysopen` for finer-grained control over input and output. `perldoc perlfaq5` includes working code for many common IO tasks.

> **Remember `autodie`?**
>
> Assume the examples in this section have `use autodie;` enabled so as to elide explicit error handling. If you choose not to use `autodie`, check the return values of *all* system calls to handle errors appropriately.

Unicode, IO Layers, and File Modes

In addition to the file mode, you may add an *IO encoding layer* which allows Perl to encode to or decode from a Unicode encoding. For example, to read a file written in the UTF-8 encoding:

```
open my $in_fh,  '<:encoding(UTF-8)', $infile;
```

...or to write to a file using the UTF-8 encoding:

```
open my $out_fh, '>:encoding(UTF-8)', $outfile;
```

Two-argument `open`

Older code often uses the two-argument form of `open()`, which jams the file mode with the name of the file to open:

```
open my $fh, "> $file" or die "Cannot write to '$file': $!\n";
```

Perl must extract the file mode from the filename. That's a risk; anytime Perl has to guess at what you mean, it may guess incorrectly. Worse, if `$file` came from untrusted user input, you have a potential security problem, as any unexpected characters could change how your program behaves.

The three-argument `open()` is a safer replacement for this code.

The Many Names of `DATA`

The special package global `DATA` filehandle represents the current file of source code. When Perl finishes compiling a file, it leaves `DATA` open and pointing to the end of the compilation unit *if* the file has a `__DATA__` or `__END__` section. Any text which occurs after that token is available for reading from `DATA`. The entire file is available if you use `seek` to rewind the filehandle. This is useful for short, self-contained programs. See `perldoc perldata` for more details.

Reading from Files

Given a filehandle opened for input, read from it with the `readline` builtin, also written as `<>`. A common idiom reads a line at a time in a `while()` loop:

```
open my $fh, '<', 'some_file';

while (<$fh>) {
    chomp;
    say "Read a line '$_'";
}
```

In scalar context, `readline` reads a single line of the file and returns it, or `undef` if it's reached the end of file (test that condition with the `eof` builtin). Each iteration in this example returns the next line or `undef`. This `while` idiom explicitly checks the *definedness* of the variable used for iteration, so only the end of file condition will end the loop. This idiom is equivalent to:

```
open my $fh, '<', 'some_file';

while (defined($_ = <$fh>)) {
    chomp;
    say "Read a line '$_'";
    last if eof $fh;
}
```

Why use `while` and not `for`?

`for` imposes list context on its operands. When in list context, `readline` will read the *entire* file before processing *any* of it. `while` performs iteration and reads a line at a time. When memory use is a concern, use `while`.

Every line read from `readline` includes the character or characters which mark the end of a line. In most cases, this is a platform-specific sequence consisting of a newline (`\n`), a carriage return (`\r`), or a combination of the two (`\r\n`). Use `chomp` to remove it.

The cleanest way to read a file line-by-line in Perl is:

```
open my $fh, '<', $filename;

while (my $line = <$fh>) {
    chomp $line;
    ...
}
```

Perl assumes that files contain text by default. If you're reading *binary* data–a media file or a compressed file, for example–use `binmode` before performing any IO. This will force Perl to treat the file data as pure data, without modifying it in any way, such as translating `\n` into the platform-specific newline sequence. While Unix-like platforms may not always *need* `binmode`, portable programs play it safe (Unicode and Strings, pp. 19).

Writing to Files

Given a filehandle open for output, `print` or `say` to write to the file:

```
open my $out_fh, '>', 'output_file.txt';

print $out_fh "Here's a line of text\n";
say   $out_fh "... and here's another";
```

Note the lack of comma between the filehandle and the next operand.

Filehandle Disambiguation

Damian Conway's *Perl Best Practices* recommends enclosing the filehandle in curly braces as a habit. This is necessary to disambiguate parsing of a filehandle contained in anything other than a plain scalar–a filehandle in an array or hash or returned from an object method–and it won't hurt anything in the simpler cases.

Both `print` and `say` take a list of operands. Perl uses the magic global `$,` as the separator between list values. Perl uses any value of `$\` as the final argument to `print` (but always uses `\n` as an implicit final argument to `say`). Remember that `$\` is `undef` by default. These two examples produce the same result:

```
my @princes = qw( Corwin Eric Random ... );
local $\    = "\n\n";

# prints a list of princes, followed by two newlines
print @princes;

local $\    = '';
print join( $,, @princes ) . "\n\n";
```

Closing Files

When you've finished working with a file, `close` its filehandle explicitly or allow it to go out of scope. Perl will close it for you. The benefit of calling `close` explicitly is that you can check for–and recover from–specific errors, such as running out of space on a storage device or a broken network connection.

As usual, `autodie` handles these checks for you:

```
use autodie qw( open close );

open my $fh, '>', $file;

...

close $fh;
```

Special File Handling Variables

For every line read, Perl increments the value of the variable `$.`, which serves as a line counter.

`readline` uses the current contents of `$/` as the line-ending sequence. The value of this variable defaults to the most appropriate line-ending character sequence for text files on your current platform. The word *line* is a misnomer, however. `$/` can contain any sequence of characters (but *not* a regular expression). This is useful for highly-structured data in which you want to read a *record* at a time.

Given a file with records separated by two blank lines, set `$/` to `\n\n` to read a record at a time. Use `chomp` on a record read from the file to remove the double-newline sequence.

Perl *buffers* its output by default, performing IO only when the amount of pending output exceeds a threshold. This allows Perl to batch up expensive IO operations instead of always writing very small amounts of data. Yet sometimes you want to send data as soon as you have it without waiting for that buffering–especially if you're writing a command-line filter connected to other programs or a line-oriented network service.

The `$|` variable controls buffering on the currently active output filehandle. When set to a non-zero value, Perl will flush the output after each write to the filehandle. When set to a zero value, Perl will use its default buffering strategy.

Automatic Flushing

Files default to a fully-buffered strategy. `STDOUT` when connected to an active terminal–but *not* another program–uses a line-buffered strategy, where Perl flushes `STDOUT` every time it encounters a newline in the output.

Instead of cluttering your code with a global variable, use the `autoflush()` method to change the buffering behavior of a lexical filehandle:

```
open my $fh, '>', 'pecan.log';
$fh->autoflush( 1 );
```

You can call any method provided by `IO::File` on a filehandle. For example, the `input_line_number()` and `input_record_separator()` methods do the job of `$.` and `$/` on individual filehandles. See the documentation for `IO::File`, `IO::Handle`, and `IO::Seekable`.

Directories and Paths

Working with directories is similar to working with files, except that you cannot *write* to directories. Open a directory handle with the `opendir` builtin:

```
opendir my $dirh, '/home/monkeytamer/tasks/';
```

The `readdir` builtin reads from a directory. As with `readline`, you may iterate over the contents of directories one entry at a time or you may assign everything to an array in one swoop:

```
# iteration
while (my $file = readdir $dirh) {
```

```
    ...
}

# flatten into a list, assign to array
my @files = readdir $otherdirh;
```

In a `while` loop, `readdir` sets `$_`:

```
opendir my $dirh, 'tasks/circus/';

while (readdir $dirh) {
    next if /^\./;
    say "Found a task $_!";
}
```

The curious regular expression in this example skips so-called *hidden files* on Unix and Unix-like systems, where a leading dot prevents them from appearing in directory listings by default. It also skips the two special files `.` and `..`, which represent the current directory and the parent directory respectively.

The names returned from `readdir` are *relative* to the directory itself. (Remember that an *absolute* path is a path fully qualified to its filesystem.) If the *tasks/* directory contains three files named *eat*, *drink*, and *be_monkey*, `readdir` will return `eat`, `drink`, and `be_monkey` instead of *tasks/eat*, *tasks/drink*, and *task/be_monkey*.

Close a directory handle with the `closedir` builtin or by letting it go out of scope.

Manipulating Paths

Perl offers a Unixy view of your filesystem and will interpret Unix-style paths appropriately for your operating system and filesystem. If you're using Microsoft Windows, you can use the path *C:/My Documents/Robots/Bender/* just as easily as you can use the path *C:\My Documents\Robots\Caprica Six*.

Even though Perl uses Unix file semantics consistently, cross-platform file manipulation is much easier with a module. The core `File::Spec` module family lets you manipulate file paths safely and portably. It's a little clunky, but it's well documented.

The `Path::Class` distribution on the CPAN has a nicer interface. Use the `dir()` function to create an object representing a directory and the `file()` function to create an object representing a file:

```
use Path::Class;

my $meals = dir( 'tasks', 'cooking' );
my $file  = file( 'tasks', 'health', 'robots.txt' );
```

You can get file objects from directories and vice versa:

```
my $lunch      = $meals->file( 'veggie_calzone' );
my $robots_dir = $robot_list->dir;
```

You can even open filehandles to directories and files:

```
my $dir_fh    = $dir->open;
my $robots_fh = $robot_list->open( 'r' ) or die "Open failed: $!";
```

Both `Path::Class::Dir` and `Path::Class::File` offer further useful behaviors–though beware that if you use a `Path::Class` object of some kind with an operator or function which expects a string containing a file path, you need to stringify the object yourself. This is a persistent but minor annoyance. (If you find it burdensome, try `Path::Tiny` as an alternative.)

```
my $contents = read_from_filename( "$lunch" );
```

File Manipulation

Besides reading and writing files, you can also manipulate them as you would directly from a command line or a file manager. The file test operators, collectively called the `-X` operators, examine file and directory attributes. To test that a file exists:

```
say 'Present!' if -e $filename;
```

The `-e` operator has a single operand, either the name of a file or handle to a file or directory. If the file or directory exists, the expression will evaluate to a true value. `perldoc -f -X` lists all other file tests.

`-f` returns a true value if its operand is a plain file. `-d` returns a true value if its operand is a directory. `-r` returns a true value if the file permissions of its operand permit reading by the current user. `-s` returns a true value if its operand is a non-empty file. Look up the documentation for any of these operators with `perldoc -f -r`, for example.

The `rename` builtin can rename a file or move it between directories. It takes two operands, the old path of the file and the new path:

```
rename 'death_star.txt', 'carbon_sink.txt';

# or if you're stylish:
rename 'death_star.txt' => 'carbon_sink.txt';
```

There's no core builtin to copy a file, but the core `File::Copy` module provides both `copy()` and `move()` functions. Use the `unlink` builtin to remove one or more files. (The `delete` builtin deletes an element from a hash, not a file from the filesystem.) These functions and builtins all return true values on success and set `$!` on error.

`Path::Class` also provides convenience methods to remove files completely and portably as well as to check certain file attributes.

Perl tracks its current working directory. By default, this is the active directory from where you launched the program. The core `Cwd` module's `cwd()` function returns the name of the current working directory. The builtin `chdir` attempts to change the current working directory. Working from the correct directory is essential to working with files with relative paths.

The CPAN module `File::chdir` makes manipulating the current working directory easier. If you're a fan of the command line and use `pushd` and `popd`, see also `File::pushd`.

Modules

You've seen functions, classes, and data structure used to organize code. Perl's next mechanism for organization and extension is the module. A *module* is a package contained in its own file and loadable with `use` or `require`. A module must be valid Perl code. It must end with an expression which evaluates to a true value so that the Perl parser knows it has loaded and compiled the module successfully.

There are no other requirements–only strong conventions.

When you load a module, Perl splits the package name on double-colons (`::`) and turns the components of the package name into a file path. This means that `use StrangeMonkey;` causes Perl to search for a file named *StrangeMonkey.pm* in every directory in `@INC` in order, until it finds one or exhausts the list.

Similarly, `use StrangeMonkey::Persistence;` causes Perl to search for a file named `Persistence.pm` in every directory named *StrangeMonkey/* present in every directory in `@INC`, and so on. `use StrangeMonkey::UI::Mobile;` causes Perl to search for a relative file path of *StrangeMonkey/UI/Mobile.pm* in every directory in `@INC`.

The resulting file may or may not contain a package declaration matching its filename–there is no such technical *requirement*–but you'll cause confusion without that match.

perldoc Tricks

`perldoc -l Module::Name` will print the full path to the relevant *.pm* file, if that file contains *documentation* in POD form. `perldoc -lm Module::Name` will print the full path to the *.pm* file. `perldoc -m Module::Name` will display the contents of the *.pm* file.

Organizing Code with Modules

Perl does not require you to use modules, packages, or namespaces. You may put all of your code in a single *.pl* file, or in multiple *.pl* files you `require` as necessary. You have the flexibility to manage your code in the most appropriate way, given your development style, the formality and risk and reward of the project, your experience, and your comfort with deploying code.

Even so, a project with more than a couple of hundred lines of code benefits from module organization:

- Modules help to enforce a logical separation between distinct entities in the system.
- Modules provide an API boundary, whether procedural or OO.
- Modules suggest a natural organization of source code.
- The Perl ecosystem has many tools devoted to creating, maintaining, organizing, and deploying modules and distributions.
- Modules provide a mechanism of code reuse.

Even if you do not use an object-oriented approach, modeling every distinct entity or responsibility in your system with its own module keeps related code together and separate code separate.

Using and Importing

When you load a module with `use`, Perl loads it from disk, then calls its `import()` method with any arguments you provided. That `import()` method takes a list of names and exports functions and other symbols into the calling namespace. This is merely convention; a module may decline to provide an `import()`, or its `import()` may perform other behaviors. Pragmas (Pragmas, pp. 129) such as `strict` use arguments to change the behavior of the calling lexical scope instead of exporting symbols:

```
use strict;
# ... calls strict->import()

use File::Spec::Functions 'tmpdir';
# ... calls File::Spec::Functions->import( 'tmpdir' )

use feature qw( say unicode_strings );
# ... calls feature->import( qw( say unicode_strings ) )
```

The `no` builtin calls a module's `unimport()` method, if it exists, passing any arguments. This is most common with pragmas which introduce or modify behavior through `import()`:

```
use strict;
# no symbolic references or barewords
# variable declaration required

{
    no strict 'refs';
    # symbolic references allowed
    # strict 'subs' and 'vars' still in effect
}
```

Both `use` and `no` take effect during compilation, such that:

```
use Module::Name qw( list of arguments );
```

...is the same as:

```
BEGIN {
    require 'Module/Name.pm';
    Module::Name->import( qw( list of arguments ) );
}
```

Similarly:

```
no Module::Name qw( list of arguments );
```

...is the same as:

```
BEGIN {
    require 'Module/Name.pm';
    Module::Name->unimport(qw( list of arguments ));
}
```

...including the `require` of the module.

Missing Methods Never Missed

If `import()` or `unimport()` does not exist in the module, Perl will produce no error. These methods are truly optional.

You *may* call `import()` and `unimport()` directly, though outside of a `BEGIN` block it makes little sense to do so; after compilation has completed, the effects of `import()` or `unimport()` may have little effect (given that these methods tend to modify the compilation process by importing symbols or toggling features).

Portable programs are careful about case even if they don't have to be.

Both `use` and `require` are case-sensitive. While Perl knows the difference between `strict` and `Strict`, your combination of operating system and file system may not. If you were to write `use Strict;`, Perl would not find *strict.pm* on a case-sensitive filesystem. With a case-insensitive filesystem, Perl would happily load *Strict.pm*, but nothing would happen when it tried to call `Strict->import()`. (*strict.pm* declares a package named `strict`. `Strict` does not exist and thus has no `import()` method, which is not an error.)

Exporting

A module can make package global symbols available to other packages through a process known as *exporting*–often by calling `import()` implicitly or directly.

The core module `Exporter` is the standard way to export symbols from a module. `Exporter` relies on the presence of package global variables such as `@EXPORT_OK` and `@EXPORT`, which list symbols to export when requested.

Consider a `StrangeMonkey::Utilities` module which provides several standalone functions:

```
package StrangeMonkey::Utilities;

use Exporter 'import';
```

```
our @EXPORT_OK = qw( round translate screech );

...
```

Any other code now can use this module and, optionally, import any or all of the three exported functions. You may also export variables:

```
push @EXPORT_OK, qw( $spider $saki $squirrel );
```

Export symbols by default by listing them in `@EXPORT` instead of `@EXPORT_OK`:

```
our @EXPORT = qw( monkey_dance monkey_sleep );
```

...so that any `use StrangeMonkey::Utilities;` will import both functions. Be aware that specifying symbols to import will *not* import default symbols; you only get what you request. To load a module without importing any symbols, use an explicit empty list:

```
# make the module available, but import() nothing
use StrangeMonkey::Utilities ();
```

Regardless of any import lists, you can always call functions in another package with their fully-qualified names:

```
StrangeMonkey::Utilities::screech();
```

Simplified Exporting

The CPAN module `Sub::Exporter` provides a nicer interface to export functions without using package globals. It also offers more powerful options. However, `Exporter` can export variables, while `Sub::Exporter` only exports functions. The CPAN module `Moose::Exporter` offers a powerful mechanism to work with Moose-based systems, but the learning curve is not shallow.

Distributions

A *distribution* is a collection of metadata and modules (Modules, pp. 143) into a single redistributable, testable, and installable unit. The easiest way to configure, build, package, test, and install Perl code is to follow the CPAN's conventions. These conventions govern how to package a distribution, how to resolve its dependencies, where to install the code and documentation, how to verify that it works, how to display documentation, and how to manage a repository. These guidelines have arisen from the rough consensus of thousands of contributors working on tens of thousands of projects.

A distribution built to CPAN standards can be tested on several versions of Perl on several different hardware platforms within a few hours of its uploading, with errors reported automatically to authors–all without human intervention. When people talk about CPAN being Perl's secret weapon, this is what they mean.

You may choose never to release any of your code as public CPAN distributions, but you *can* use CPAN tools and conventions to manage even private code. The Perl community has built amazing infrastructure. Take advantage of it.

Attributes of a Distribution

Besides modules, a distribution includes several files and directories:

- *Build.PL* or *Makefile.PL*, a driver program used to configure, build, test, bundle, and install the distribution.

- *MANIFEST*, a list of all files contained in the distribution. This helps tools verify that a bundle is complete.
- *META.yml* and/or *META.json*, a file containing metadata about the distribution and its dependencies.
- *README*, a description of the distribution, its intent, and its copyright and licensing information.
- *lib/*, the directory containing Perl modules.
- *t/*, a directory containing test files.
- *Changes*, a human-readable log of every significant change to the distribution.

A well-formed distribution must contain a unique name and single version number (often taken from its primary module). Any distribution you download from the public CPAN should conform to these standards. The public CPANTS service[4] evaluates each uploaded distribution against packaging guidelines and conventions and recommends improvements. Following the CPANTS guidelines doesn't mean the code works, but it does mean that CPAN packaging and installation tools should understand the distribution.

CPAN Tools for Managing Distributions

The Perl core includes several tools to manage distributions:

- `CPAN.pm` is the official CPAN client. While by default this client installs distributions from the public CPAN, you can also use your own repository instead of or in addition to the public repository.
- `ExtUtils::MakeMaker` is a complex but well-used system of modules used to package, build, test, and install Perl distributions. It works with *Makefile.PL* files.
- `Test::More` (Testing, pp. 131) is the basic and most widely used testing module used to write automated tests for Perl software.
- `TAP::Harness` and `prove` (Running Tests, pp. 132) run tests and interpret and report their results.

In addition, several non-core CPAN modules make your life easier as a developer:

- `App::cpanminus` is a configuration-free CPAN client. It handles the most common cases, uses little memory, and works quickly.
- `App::perlbrew` helps you to manage multiple installations of Perl. Install new versions of Perl for testing or production, or to isolate applications and their dependencies.
- `CPAN::Mini` and the `cpanmini` command allow you to create your own (private) mirror of the public CPAN. You can inject your own distributions into this repository and manage which versions of the public modules are available in your organization.
- `Dist::Zilla` automates away common distribution tasks. While it uses either `Module::Build` or `ExtUtils::MakeMaker`, it can replace *your* use of them directly. See http://dzil.org/ for an interactive tutorial.
- `Test::Reporter` allows you to report the results of running the automated test suites of distributions you install, giving their authors more data on any failures.
- Carton and Pinto are two newer projects which help manage and install code's dependencies. Neither is in widespread use yet, but they're both under active development.
- `Module::Build` is an alternative to `ExtUtils::MakeMaker`, written in pure Perl. While it has advantages, it's not as widely used or maintained.

[4] http://cpants.perl.org/

Designing Distributions

The process of designing a distribution could fill a book (such as Sam Tregar's *Writing Perl Modules for CPAN*), but a few design principles will help you. Start with a utility such as `Module::Starter` or `Dist::Zilla`. The initial cost of learning the configuration and rules may seem like a steep investment, but the benefit of having everything set up the right way (and in the case of `Dist::Zilla`, *never* going out of date) relieves you of tedious busywork.

A distribution should follow several non-code guidelines:

- *Each distribution performs a single, well-defined purpose.* That purpose may even include gathering several related distributions into a single installable bundle. Decompose your software into individual distributions to manage their dependencies appropriately and to respect their encapsulation.
- *Each distribution contains a single version number.* Version numbers must always increase. The semantic version policy[5] is sane and compatible with Perl's approach.
- *Each distribution provides a well-defined API.* A comprehensive automated test suite can verify that you maintain this API across versions. If you use a local CPAN mirror to install your own distributions, you can re-use the CPAN infrastructure for testing distributions and their dependencies. You get easy access to integration testing across reusable components.
- *Distribution tests are useful and repeatable.* The CPAN infrastructure supports automated test reporting. Use it!
- *Interfaces are simple and effective.* Avoid the use of global symbols and default exports; allow people to use only what they need. Do not pollute their namespaces.

The UNIVERSAL Package

Perl's builtin `UNIVERSAL` package is the ancestor of all other packages–it's the ultimate parent class in the object-oriented sense (Moose, pp. 107). `UNIVERSAL` provides a few methods for its children to use, inherit, or override.

The VERSION() Method

The `VERSION()` method returns the value of the `$VERSION` variable of the invoking package or class. If you provide a version number as an optional parameter, the method will throw an exception if the queried `$VERSION` is not equal to or greater than the parameter.

Given a `HowlerMonkey` module of version `1.23`, its `VERSION()` method behaves as:

```
my $hm = HowlerMonkey->new;

say HowlerMonkey->VERSION;     # prints 1.23
say $hm->VERSION;              # prints 1.23
say $hm->VERSION( 0.0  );      # prints 1.23
say $hm->VERSION( 1.23 );      # prints 1.23
say $hm->VERSION( 2.0  );      # exception!
```

There's little reason to override `VERSION()`.

The DOES() Method

The `DOES()` method supports the use of roles (Roles, pp. 112) in programs. Pass it an invocant and the name of a role, and the method will return true if the appropriate class somehow does that role through inheritance, delegation, composition, role application, or any other mechanism.

The default implementation of `DOES()` falls back to `isa()`, because inheritance is one mechanism by which a class may do a role. Given a `Cappuchin`, its `DOES()` method behaves as:

[5] http://semver.org/

```
say Cappuchin->DOES( 'Monkey'       );  # prints 1
say $cappy->DOES(    'Monkey'       );  # prints 1
say Cappuchin->DOES( 'Invertebrate' );  # prints 0
```

Override `DOES()` if you manually consume a role or otherwise somehow provide allomorphic equivalence.

The can() Method

The `can()` method takes a string containing the name of a method or function. It returns a function reference, if it exists. Otherwise, it returns a false value. You may call this on a class, an object, or the name of a package.

Given a class named `SpiderMonkey` with a method named `screech`, get a reference to the method with:

```
if (my $meth = SpiderMonkey->can( 'screech' )) {...}
```

This technique leads to the pattern of checking for a method's existence before dispatching to it:

```
if (my $meth = $sm->can( 'screech' ) {
    # method; not a function
    $sm->$meth();
}
```

Use `can()` to test if a package implements a specific function or method:

```
use Class::Load;

die "Couldn't load $module!" unless load_class( $module );

if (my $register = $module->can( 'register' )) {
    # function; not a method
    $register->();
}
```

Module::Pluggable

The CPAN module `Class::Load` simplifies the work of loading classes by name. `Module::Pluggable` makes it easier to build and manage plugin systems. Get to know both distributions.

The isa() Method

The `isa()` method takes a string containing the name of a class or the name of a core type (`SCALAR`, `ARRAY`, `HASH`, `Regexp`, `IO`, and `CODE`). Call it as a class method or an instance method on an object. `isa()` returns a true value if its invocant is or derives from the named class, or if the invocant is a blessed reference to the given type.

Given an object `$pepper` (a hash reference blessed into the `Monkey` class, which inherits from the `Mammal` class), its `isa()` method behaves as:

```
say $pepper->isa( 'Monkey'  );  # prints 1
say $pepper->isa( 'Mammal'  );  # prints 1
say $pepper->isa( 'HASH'    );  # prints 1
say Monkey->isa(  'Mammal'  );  # prints 1

say $pepper->isa( 'Dolphin' );  # prints 0
say $pepper->isa( 'ARRAY'   );  # prints 0
say Monkey->isa(  'HASH'    );  # prints 0
```

Any class may override `isa()`. This can be useful when working with mock objects (`Test::MockObject` and `Test::MockModule`, for example) or with code that does not use roles (Roles, pp. 112). Be aware that any class which *does* override `isa()` generally has a good reason for doing so.

Does a Class Exist?

While both `UNIVERSAL::isa()` and `UNIVERSAL::can()` are methods (Method-Function Equivalence, pp. 173), you may *safely* use the latter as a function solely to determine whether a class exists in Perl. If `UNIVERSAL::can( $classname, 'can' )` returns a true value, someone somewhere has defined a class of the name `$classname`. That class may not be usable, but it does exist.

Extending UNIVERSAL

It's tempting to store other methods in `UNIVERSAL` to make them available to all other classes and objects in Perl. Avoid this temptation; this global behavior can have subtle side effects, especially in code you didn't write and don't maintain.

With that said, occasional abuse of `UNIVERSAL` for *debugging* purposes and to fix improper default behavior may be excusable. For example, Joshua ben Jore's `UNIVERSAL::ref` distribution makes the nearly-useless `ref()` operator usable. The `UNIVERSAL::can` and `UNIVERSAL::isa` distributions can help you debug anti-polymorphism bugs (Method-Function Equivalence, pp. 173). `Perl::Critic` can detect those and other problems.

Outside of very carefully controlled code and very specific, very pragmatic situations, there's no reason to put code in `UNIVERSAL` directly, especially given the other design alternatives.

Code Generation

Novice programmers write more code than they need to write. They start with long lists of procedural code, then discover functions, then parameters, then objects, and–perhaps–higher-order functions and closures.

As you improve your skills, you'll write less code to solve the same problems. You'll use better abstractions. You'll write more general code. You can reuse code–and when you can add features by deleting code, you'll achieve something great.

Writing programs to write programs for you–*metaprogramming* or *code generation*–allows you to build reusable abstractions. While you can make a huge mess, you can also build amazing things. Metaprogramming techniques make Moose possible, for example (Moose, pp. 107).

The `AUTOLOAD` technique (AUTOLOAD, pp. 90) for missing functions and methods demonstrates this technique in a specific form: Perl's function and method dispatch system allows you to control what happens when normal lookup fails.

eval

The simplest code generation technique is to build a string containing a snippet of valid Perl and compile it with the string `eval` operator. Unlike the exception-catching block `eval` operator, string `eval` compiles the contents of the string within the current scope, including the current package and lexical bindings.

A common use for this technique is providing a fallback if you can't (or don't want to) load an optional dependency:

```
eval { require Monkey::Tracer } or eval 'sub Monkey::Tracer::log {}';
```

If `Monkey::Tracer` is not available, this code defines a `log()` function which will do nothing. This simple example is deceptive; getting `eval` right takes effort. You must handle quoting issues to include variables within your `eval`d code. Add more complexity to interpolate some variables but not others:

```
sub generate_accessors {
    my ($methname, $attrname) = @_;

    eval <<"END_ACCESSOR";
```

```
    sub get_$methname {
        my \$self = shift;
        return \$self->{$attrname};
    }

    sub set_$methname {
        my (\$self, \$value) = \@_;
        \$self->{$attrname}  = \$value;
    }
END_ACCESSOR
}
```

Woe to those who forget a backslash! Good luck convincing your syntax highlighter what's happening! Worse yet, each invocation of string `eval` builds a new data structure representing the entire code, and compiling code isn't free. Yet even with its limitations, this technique is simple and useful.

Parametric Closures

While building accessors and mutators with `eval` is straightforward, closures (Closures, pp. 84) allow you to add parameters to generated code at compilation time *without* requiring additional evaluation:

```
sub generate_accessors {
    my $attrname = shift;

    my $getter = sub {
        my $self = shift;
        return $self->{$attrname};
    };

    my $setter = sub {
        my ($self, $value) = @_;
        $self->{$attrname} = $value;
    };

    return $getter, $setter;
}
```

This code avoids unpleasant quoting issues and compiles each closure only once. It limits the memory used by sharing the compiled code between all closure instances. All that differs is the binding to the `$attrname` lexical. In a long-running process or a class with a lot of accessors, this technique can be very useful.

Installing into symbol tables is reasonably easy, if ugly:

```
my ($get, $set) = generate_accessors( 'pie' );

no strict 'refs';
*{ 'get_pie' } = $get;
*{ 'set_pie' } = $set;
```

Think of the asterisk as a *typeglob sigil*, where a *typeglob* is Perl jargon for “symbol table”. Dereferencing a string like this refers to a symbol in the current *symbol table*, which is the section of the current namespace which contains globally-accessible symbols such as package globals, functions, and methods. Assigning a reference to a symbol table entry installs or replaces that entry. To promote an anonymous function to a method, store that function's reference in the symbol table.

Assigning to a symbol table symbol with a string, not a literal variable name, is a symbolic reference. You must disable `strict` reference checking for the operation. Many programs have a subtle bug in similar code, as they assign and generate in a single line:

```
no strict 'refs';

*{ $methname } = sub {
    # subtle bug: strict refs disabled here too
};
```

This example disables strictures for the outer block *as well as the body of the function itself*. Only the assignment violates strict reference checking, so disable strictures for that operation alone:

```
{
    my $sub = sub { ... };

    no strict 'refs';
    *{ $methname } = $sub;
}
```

If the name of the method is a string literal in your source code, rather than the contents of a variable, you can assign to the relevant symbol directly:

```
{
    no warnings 'once';
    (*get_pie, *set_pie) = generate_accessors( 'pie' );
}
```

Assigning directly to the glob does not violate strictures, but mentioning each glob only once *does* produce a "used only once" warning you can disable with the `warnings` pragma.

Symbol Tables Simplified

Use the CPAN module `Package::Stash` to modify symbol tables for you.

Compile-time Manipulation

Unlike code written explicitly as code, code generated through string `eval` gets compiled while your program is running. Where you might expect a normal function to be available throughout the lifetime of your program, a generated function might not be available when you expect it.

Force Perl to run code–to generate other code–during compilation by wrapping it in a `BEGIN` block. When the Perl parser encounters a block labeled `BEGIN`, it parses and compiles the entire block, then runs it (unless it has syntax errors). When the block finishes running, parsing will continue as if there had been no interruption.

The difference between writing:

```
sub get_age    { ... }
sub set_age    { ... }

sub get_name   { ... }
sub set_name   { ... }

sub get_weight { ... }
sub set_weight { ... }
```

...and:

```
sub make_accessors { ... }

BEGIN {
    for my $accessor (qw( age name weight )) {
        my ($get, $set) = make_accessors( $accessor );

        no strict 'refs';
        *{ 'get_' . $accessor } = $get;
        *{ 'set_' . $accessor } = $set;
    }
}
```

...is primarily one of maintainability. You could argue for and against either form.

Within a module, any code outside of functions executes when you `use` the module, because of the implicit `BEGIN` Perl adds around the `require` and `import` (Importing, pp. 72). Any code outside of a function but inside the module will execute *before* the `import()` call occurs. If you `require` the module, there is no implicit `BEGIN` block. After parsing finishes, Perl will run code outside of the functions.

Beware of the interaction between lexical *declaration* (the association of a name with a scope) and lexical *assignment*. The former happens during compilation, while the latter occurs at the point of execution. This code has a subtle bug:

```
use UNIVERSAL::require;

# buggy; do not use
my $wanted_package = 'Monkey::Jetpack';

BEGIN {
    $wanted_package->require;
    $wanted_package->import;
}
```

...because the `BEGIN` block will execute *before* the assignment of the string value to `$wanted_package` occurs. The result will be an exception from attempting to invoke the `require()` method on an undefined value.

The `UNIVERSAL::require` CPAN distribution adds a `require()` method to `UNIVERSAL`.

Class::MOP

Unlike installing function references to populate namespaces and to create methods, there's no simple way to create classes dynamically in Perl. Moose comes to the rescue, with its bundled `Class::MOP` library. It provides a *meta object protocol*–a mechanism for creating and manipulating an object system by manipulating objects.

Rather than writing your own fragile string `eval` code or trying to poke into symbol tables manually, you can manipulate the entities and abstractions of your program with objects and methods.

To create a class:

```
use Class::MOP;

my $class = Class::MOP::Class->create( 'Monkey::Wrench' );
```

Add attributes and methods to this class when you create it:

```
my $class = Class::MOP::Class->create(
    'Monkey::Wrench' => (
        attributes => [
            Class::MOP::Attribute->new( '$material' ),
            Class::MOP::Attribute->new( '$color' ),
        ],
        methods => {
            tighten => sub { ... },
            loosen  => sub { ... },
        },
    ),
);
```

...or to the metaclass (the object which represents that class) once created:

```
$class->add_attribute(
    experience => Class::MOP::Attribute->new( '$xp' )
);

$class->add_method( bash_zombie => sub { ... } );
```

A MOP gives you more than the ability to create new entities as the program runs. You get to look inside existing (MOP-aware) code. For example, to determine the characteristics of the class, use the `Class::MOP::Class` methods:

```
my @attrs = $class->get_all_attributes;
my @meths = $class->get_all_methods;
```

Similarly `Class::MOP::Attribute` and `Class::MOP::Method` allow you to create and manipulate and introspect attributes and methods.

Overloading

Perl is not a pervasively object oriented language. Its core data types (scalars, arrays, and hashes) are not objects with methods, but you *can* control the behavior of your own classes and objects, especially when they undergo coercion or contextual evaluation. This is *overloading*.

Overloading is subtle but powerful. Consider how an object behaves in boolean context. In boolean context, an object will evaluate to a true value, unless you overload boolification. Why would you do this? Suppose you're using the Null Object pattern[6] to make your own objects appear false in boolean context.

You can overload an object's behavior for almost every operation or coercion: stringification, numification, boolification, iteration, invocation, array access, hash access, arithmetic operations, comparison operations, smart match, bitwise operations, and even assignment. Stringification, numification, and boolification are the most important and most common.

Overloading Common Operations

The `overload` pragma associates functions with overloadable operations. Pass the pragma argument pairs, where the key is the name of a type of overload and the value is a function reference. A `Null` class which overloads boolean evaluation so that it always evaluates to a false value might resemble:

```
package Null {
    use overload 'bool' => sub { 0 };
```

[6] http://www.c2.com/cgi/wiki?NullObject

```
    ...
}
```

It's easy to add a stringification:

```
package Null {
    use overload
        'bool' => sub { 0 },
        '""'   => sub { '(null)' };
}
```

Overriding numification is more complex, because arithmetic operators tend to be binary ops (Arity, pp. 64). Given two operands both with overloaded methods for addition, which overloading should take precedence? The answer needs to be consistent, easy to explain, and understandable by people who haven't read the source code of the implementation.

`perldoc overload` attempts to explain this in the sections labeled *Calling Conventions for Binary Operations* and *MAGIC AUTOGENERATION*, but the easiest solution is to overload numification (keyed by `'0+'`) and tell `overload` to use the provided overloads as fallbacks:

```
package Null {
    use overload
        'bool'   => sub { 0 },
        '""'     => sub { '(null)' },
        '0+'     => sub { 0 },
        fallback => 1;
}
```

Setting `fallback` to a true value gives Perl the option to use any other defined overloads to perform an operation. If that's not possible, Perl will act as if there were no overloads in effect. This is often what you want.

Without `fallback`, Perl will only use the specific overloadings you have provided. If someone tries to perform an operation you have not overloaded, Perl will throw an exception.

Overload and Inheritance

Subclasses inherit overloadings from their ancestors. They may override this behavior in one of two ways. If the parent class defines overloadings in terms of function references, a child class must do the same to override its parent's behavior.

The alternative is to define overloadings in terms of method *name*. This allows child classes to customize their behavior by overriding those methods:

```
package Null {
    use overload
        'bool'   => 'get_bool',
        '""'     => 'get_string',
        '0+'     => 'get_num',
        fallback => 1;

    sub get_bool { 0 }
}
```

Any child class can do something different for boolification by overriding `get_bool()`:

```
package Null::ButTrue {
    use parent 'Null';

    sub get_bool { 1 }
}
```

Uses of Overloading

Overloading may seem like a tempting tool to use to produce symbolic shortcuts for new operations. The `IO::All` CPAN distribution pushes this idea to its limit to produce a simple and elegant API. Yet for every brilliant API refined through the appropriate use of overloading, a dozen more messes congeal. Sometimes the best code eschews cleverness in favor of simplicity.

Overriding addition, multiplication, and even concatenation on a `Matrix` class makes sense because the existing notation for those operations is pervasive. A new problem domain without that established notation is a poor candidate for overloading, as is a problem domain where you have to squint to make Perl's existing operators match a different notation.

Damian Conway's *Perl Best Practices* suggests one other use for overloading: to prevent the accidental abuse of objects. For example, overloading numification to `croak()` for objects which have no reasonable single numeric representation can help you find and fix real bugs.

Taint

Some Perl features can help you write secure programs. These tools are no substitute for careful thought and planning, but they *reward* caution and understanding and can help you avoid subtle mistakes.

Taint mode (or *taint*) is a sticky piece of metadata attached to all data which comes from outside of your program. Any data derived from tainted data is also tainted. You may use tainted data within your program, but if you use it to affect the outside world–if you use it insecurely–Perl will throw a fatal exception.

Using Taint Mode

`perldoc perlsec` explains taint mode in copious detail.

Launch your program with the `-T` command-line argument to enable taint mode. If you use this argument on the `#!` line of a program, you must run the program directly. If you run it as `perl mytaintedappl.pl` and neglect the `-T` flag, Perl will exit with an exception–by the time Perl encounters the flag on the `#!` line, it's missed its opportunity to taint the environment data in `%ENV`, for example.

Sources of Taint

Taint can come from two places: file input and the program's operating environment. The former is anything you read from a file or collect from users in the case of web or network programming. The latter includes any command-line arguments, environment variables, and data from system calls. Even operations such as reading from a directory handle produce tainted data.

The `tainted()` function from the core module `Scalar::Util` returns true if its argument is tainted:

```
die 'Oh no! Tainted data!' if Scalar::Util::tainted( $sketchy_data );
```

Removing Taint from Data

To remove taint, you must extract known-good portions of the data with a regular expression capture. That captured data will be untainted. For example, if your user input consists of a US telephone number, you can untaint it with:

```
die 'Number still tainted!' unless $number =~ /(\(/d{3}\) \d{3}-\d{4})/;

my $safe_number = $1;
```

The more specific your pattern is about what you allow, the more secure your program can be. The opposite approach of *denying* specific items or forms runs the risk of overlooking something harmful. Far better to disallow something that's safe but unexpected than that to allow something harmful which appears safe. Even so, nothing prevents you from writing a capture for the entire contents of a variable–but in that case, why use taint?

Removing Taint from the Environment

The superglobal `%ENV` represents the environment variables of the system where you're running your program. This data is tainted because forces outside of the program's control can manipulate values there. Any environment variable which modifies how Perl or the shell finds files and directories is an attack vector. A taint-sensitive program should delete several keys from `%ENV` and set `$ENV{PATH}` to a specific and well-secured path:

```
delete @ENV{ qw( IFS CDPATH ENV BASH_ENV ) };
$ENV{PATH} = '/path/to/app/binaries/';
```

If you do not set `$ENV{PATH}` appropriately, you will receive messages about its insecurity. If this environment variable contained the current working directory, or if it contained relative directories, or if the directories could be modified by anyone else on the system, a clever attacker could perpetrate mischief.

For similar reasons, `@INC` does not contain the current working directory under taint mode. Perl will also ignore the `PERL5LIB` and `PERLLIB` environment variables. Use the `lib` pragma or the `-I` flag to `perl` to add library directories to the program.

Taint Gotchas

Taint mode is all or nothing. It's either on or off. This sometimes leads people to use permissive patterns to untaint data and, thus, gives the illusion of security. In that case, taint is busywork which provides no real security. Review your untainting rules carefully.

Unfortunately, not all modules handle tainted data appropriately. This is a bug which CPAN authors should take more seriously. If you have to make legacy code taint-safe, consider the use of the `-t` flag, which enables taint mode but reduces taint violations from exceptions to warnings. This is not a substitute for full taint mode, but it allows you to secure existing programs without the all or nothing approach of `-T`.

CHAPTER 10

Perl Beyond Syntax

"Simple" means different things to different programmers. Effective programmers understand how Perl's features interact and combine. Their fluent code takes advantage of language patterns and idioms. The result of this Perlish thinking is concise, powerful, and useful code–and it's simple when you understand it.

Idioms

Every language has common patterns of expression, or *idioms*. The earth revolves, but we speak of the sun rising or setting. We brag about clever hacks but cringe at nasty hacks and code smells.

Perl has idioms; they're both language features and design techniques. They're mannerisms and mechanisms that give your code a Perlish accent. You don't have to use them to get your job done, but they play to Perl's strengths.

The Object as `$self`

Perl's object system (Moose, pp. 107) treats the invocant of a method as a mundane parameter. Regardless of whether you invoke a class or an instance method, the first element of `@_` is always the invocant. By convention, most Perl code uses `$class` as the name of the class method invocant and `$self` for the name of the object invocant. This convention is strong enough that useful extensions such as `Moops` assume you will use `$self` as the name of object invocants.

Named Parameters

Perl loves lists. Lists are a fundamental element of Perl. List flattening and binding lets you chain together multiple expressions to manipulate data in every way possible.

While Perl's argument passing simplicity (everything flattens into `@_`) is occasionally too simple, assigning from `@_` in list context allows you to unpack named parameters as pairs. The fat comma (Declaring Hashes, pp. 43) operator turns an ordinary list into an obvious list of pairs of arguments:

```
make_ice_cream_sundae(
    whipped_cream => 1,
    sprinkles     => 1,
    banana        => 0,
    ice_cream     => 'mint chocolate chip',
);
```

You can unpack these parameters into a hash and treat that hash as if it were a single argument:

```
sub make_ice_cream_sundae {
    my %args    = @_;
    my $dessert = get_ice_cream( $args{ice_cream} );
    ...
}
```

Hash or Hash Ref?

Perl Best Practices suggests passing hash references instead. This allows Perl to perform caller-side validation of the hash reference. If you pass the wrong number of arguments, you'll get an error where you *call* the function.

This technique works well with `import()` (Importing, pp. 72) or other methods; process as many parameters as you like before slurping the remainder into a hash:

```
sub import {
    my ($class, %args)  = @_;
    my $calling_package = caller();
    ...
}
```

The Schwartzian Transform

The *Schwartzian transform* is an elegant demonstration of the pervasive list handling idiom borrowed from Lisp. Suppose you have a Perl hash which associates the names of your co-workers with their phone extensions:

```
my %extensions = (
    '000' => 'Damian',
    '002' => 'Wesley',
    '042' => 'Robin',
    '088' => 'Nic',
);
```

Hash Key Quoting Rules

Fat comma hash key quoting only works on things that look like barewords. With the leading zero, these keys look like octal numbers. Everyone makes this mistake at least once.

To sort this list by name alphabetically, you must sort the hash by its values, not its keys. Getting the values sorted correctly is easy:

```
my @sorted_names = sort values %extensions;
```

...but you need an extra step to preserve the association of names and extensions, hence the Schwartzian transform. First, convert the hash into a list of data structures which will be easier to sort–in this case, two-element anonymous arrays:

```
my @pairs = map  { [ $_, $extensions{$_} ] } keys %extensions;
```

`sort` takes this list of anonymous arrays and compares their second elements (the names) as strings:

```
my @sorted_pairs = sort { $a->[1] cmp $b->[1] } @pairs;
```

The block provided to `sort` receives arguments in two package-scoped (Scope, pp. 77) variables: `$a` and `$b`. (See `perldoc -f sort` for an extensive discussion of the implications of this scoping.) The `sort` block takes its arguments two at a time. The first becomes the contents of `$a` and the second the contents of `$b`. If `$a` should sort ahead of `$b` in the results, the block must return -1. If both values sort to the same position, the block must return 0. Finally, if `$a` should sort after `$b` in the results, the block should return 1. Any other return values are errors.

Know Your Data

Reversing the hash *in place* would work if no one had the same name. This particular data set presents no such problem, but code defensively.

The `cmp` operator performs string comparisons and the `<=>` performs numeric comparisons. Given `@sorted_pairs`, a second `map` operation converts the data structure to a more usable form:

```
my @formatted_exts = map { "$_->[1], ext. $_->[0]" } @sorted_pairs;
```

. . . and now you can print the whole thing:

```
say for @formatted_exts;
```

The Schwartzian transformation chains all of these expressions together to elide those temporary variables:

```
say for
    map  { " $_->[1], ext. $_->[0]"          }
    sort {   $a->[1] cmp   $b->[1]           }
    map  { [ $_      =>    $extensions{$_} ] }
    keys %extensions;
```

Read the expression from right to left, in evaluation order. For each key in the extensions hash, make a two-item anonymous array containing the key and the value. Sort that list of anonymous arrays by their second elements, the hash values. Format a string of output from those sorted arrays.

The Schwartzian transform pipeline of `map-sort-map` transforms a data structure into another form easier for sorting and then transforms it back into the first from–or another form.

While this sorting example is simple, consider the case of calculating a cryptographic hash for a large file. The Schwartzian transform is especially useful because it effectively caches any expensive calculations by performing them once in the first-executed `map`.

Easy File Slurping

`local` is essential to managing Perl's magic global variables. You must understand scope (Scope, pp. 77) to use `local` effectively–but if you do, you can use tight and lightweight scopes in interesting ways. For example, to slurp files into a scalar in a single expression:

```
my $file = do { local $/; <$fh> };

# or
my $file; { local $/; $file = <$fh> };
```

`$/` is the input record separator. `local`izing it sets its value to `undef`, pending assignment. As the value of the separator is undefined, Perl happily reads the entire contents of the filehandle in one swoop. Because a `do` block evaluates to the value of the last expression evaluated within the block, this evaluates to the data read from the filehandle: the contents of the file. At the end of the expression, `$/` has reverted to its previous state and `$file` contains the contents of the file.

The second example avoids a second copy of the string containing the file's contents; it's not as pretty, but it uses the least amount of memory.

File::Slurper

This useful example is admittedly maddening for people who don't understand both `local` and scoping. The `File::Slurper` module from the CPAN is a worthy (and often faster) alternative.

Handling Main

Many programs commonly set up several file-scoped lexical variables before handing off processing to other functions. It's tempting to use these variables directly, rather than passing values to and returning values from functions, especially as programs grow. Unfortunately, these programs may come to rely on subtleties of what happens when during Perl's compilation process–a variable you *thought* would be initialized to a specific value may not get initialized until much later. Remember that Perl requires no special syntax for creating closures (Closures, pp. 84)–you can close over a lexical variable inadvertently.

To avoid this, wrap the main code of your program in a single function, `main()`. Encapsulate your variables to their proper scopes. Then add a single line to the beginning of your program, after you've used all of the modules and pragmas you need:

```
#!/usr/bin/perl

use Modern::Perl;

exit main( @ARGV );

sub main {
    ...

    # successful exit
    return 0;
}

sub other_functions { ... }
```

Calling `main()` *before* anything else in the program forces you to be explicit about initialization and compilation order. Calling `exit` with `main()`'s return value prevents any other bare code from running.

Controlled Execution

The effective difference between a program and a module is in its intended use. Users invoke programs directly, while programs load modules after execution has already begun. Yet both modules and programs are merely Perl code. Making a module executable is easy. So is making a program behave as a module (useful for testing parts of an existing program without formally making a module). All you need to do is to discover *how* Perl began to execute a piece of code.

`caller`'s single optional argument governs the number of call frames (Recursion, pp. 74) to look back through. `caller(0)` reports information about the current call frame. To allow a module to run correctly as a program *or* a module, put all executable code in functions, add a `main()` function, and write a single line at the start of the module:

```
main() unless caller(0);
```

If there's *no* caller for the module, someone invoked it directly as a program (with `perl path/to/Module.pm` instead of `use Module;`).

Improved Caller Inspection

The eighth element of the list returned from `caller` in list context is a true value if the call frame represents `use` or `require` and `undef` otherwise. While that's more accurate, few people use it.

Postfix Parameter Validation

The CPAN has several modules which help verify the parameters of your functions; `Params::Validate` and `MooseX::Params::Validat` are two good options. Simple validation is easy even without those modules.

Suppose your function takes exactly two arguments. You *could* write:

```
use Carp 'croak';

sub groom_monkeys {
    if (@_ != 2) {
        croak 'Can only groom two monkeys!';
    }
    ...
}
```

...but from a linguistic perspective, the consequences are more important than the check and deserve to be at the *start* of the expression:

```
croak 'Can only groom two monkeys!' unless @_ == 2;
```

This early return technique–especially with postfix conditionals–can simplify the rest of the code. Each such assertion is effectively a single row in a truth table. Alternately, function signatures (Real Function Signatures, pp. 69) of some kind will handle this case for you.

Regex En Passant

Many Perl idioms rely on the fact that expressions evaluate to values:

```
say my $ext_num = my $extension = 42;
```

That clunky code demonstrates how to use the value of one expression in another expression. This isn't a new idea; you've likely used the return value of a function in a list or as an argument to another function before. You may not have realized its implications.

Suppose you want to extract a first name from a first name plus surname combination with a precompiled regular expression in `$first_name_rx`:

```
my ($first_name) = $name =~ /($first_name_rx)/;
```

In list context, a successful regex match returns a list of all captures (Capturing, pp. 99, and Perl assigns the first one to `$first_name`.

To remove all non-word characters to create a useful user name for a system account, you could write:

```
(my $normalized_name = $name) =~ tr/A-Za-z//dc;
```

Non-Destructive Substitutions

Newer code can use the non-destructive substitution modifier `/r`: `my $normalized_name = $name =~ tr/A-Za-z//dcr;`.

First, assign the value of `$name` to `$normalized_name`. The parentheses change precedence so that assignment happens first. The assignment expression evaluates to the *variable* `$normalized_name`. That variable becomes the first operand to the transliteration operator.

This technique works on other in-place modification operators:

```
my $age = 14;
(my $next_age = $age)++;

say "I am $age, but next year I will be $next_age";
```

Unary Coercions

Perl's type system almost always does the right thing when you choose the correct operators. Use the string concatenation operator and Perl will treat both operands as strings. Use the addition operator and Perl will treat both operands as numeric.

Occasionally you have to give Perl a hint about what you mean with a *unary coercion* to force a specific evaluation of a value.

Add zero to treat a value as numeric:

```
my $numeric_value = 0 + $value;
```

Double negate a value to treat it as a boolean:

```
my $boolean_value = !! $value;
```

Concatenate a value with the empty string to treat it as a string:

```
my $string_value = '' . $value;
```

The need for these coercions is vanishingly rare, but it happens. While it may look like it would be safe to remove a "useless" `+ 0` from an expression, doing so may well break the code.

Global Variables

Perl provides several *super global variables*. They're not scoped to a package or file. They're really, truly global. Unfortunately, any direct or indirect modifications of these variables may change the behavior of other parts of the program. Experienced Perl hackers have memorized some of them. Few people have memorized all of them–they're terse. Only a handful are regularly useful. `perldoc perlvar` contains the exhaustive list of these variables.

Managing Super Globals

As Perl evolves, it moves more global behavior into lexical behavior, so that you can avoid many of these globals. When you must use them, use `local` in the smallest possible scope to constrain any modifications. You are still susceptible to any changes made to these variables from code you *call*, but you reduce the likelihood of surprising code *outside* of your scope. As the easy file slurping idiom (Easy File Slurping, pp. 160) demonstrates, `local` is often the right approach:

```
my $file; { local $/; $file = <$fh> };
```

The effect of localizing $/ lasts only through the end of the block. There is a low chance that any Perl code will run as a result of reading lines from the filehandle and change the value of $/ within the do block.

Not all cases of using super globals are this easy to guard, but this often works.

Other times you need to *read* the value of a super global and hope that no other code has modified it. Catching exceptions with an eval block is susceptible to at least one race condition where DESTROY() methods invoked on lexicals that have gone out of scope may reset $@:

```
local $@;

eval { ... };

if (my $exception = $@) { ... }
```

Copy $@ *immediately* after catching an exception to preserve its contents. See also Try::Tiny instead (Exception Caveats, pp. 128).

English Names

The core English module provides verbose names for punctuation-heavy super globals. Import them into a namespace with:

```
use English '-no_match_vars'; # unnecessary in 5.20 and 5.22
```

This allows you to use the verbose names documented in perldoc perlvar within the scope of this pragma.

Three regex-related super globals ($&, $‘, and $') used to impose a global performance penalty for *all* regular expressions within a program. This has been fixed in Perl 5.20. If you forget the -no_match_vars import, your program will suffer the penalty even if you don't explicitly read from those variables. Modern Perl programs can use the @- variable instead of them.

Useful Super Globals

Most programs can get by with using only a couple of the super globals. You're most likely to encounter only a few of these variables.

$/ (or $INPUT_RECORD_SEPARATOR from the English pragma) is a string of zero or more characters which denotes the end of a record when reading input a record at a time. By default, this is your platform-specific newline character sequence. If you undefine this value, Perl will attempt to read the entire file into memory. If you set this value to a *reference* to an integer, Perl will try to read that many *bytes* per record (so beware of Unicode concerns). If you set this value to an empty string (''), Perl will read in a paragraph at a time, where a paragraph is a chunk of text followed by an arbitrary number of newlines.

$. ($INPUT_LINE_NUMBER) contains the number of records read from the most recently-accessed filehandle. You can read from this variable, but writing to it has no effect. Localizing this variable will localize the filehandle to which it refers. Yes, that's confusing.

$| ($OUTPUT_AUTOFLUSH) governs whether Perl will flush everything written to the currently selected filehandle immediately or only when Perl's buffer is full. Unbuffered output is useful when writing to a pipe or socket or terminal which should not block waiting for input. This variable will coerce any values assigned to it to boolean values.

@ARGV contains the command-line arguments passed to the program.

$! ($ERRNO) is a dualvar (Dualvars, pp. 51) which contains the result of the *most recent* system call. In numeric context, this corresponds to C's errno value, where anything other than zero indicates an error. In string context, this evaluates to the appropriate system error string. Localize this variable before making a system call (implicitly or explicitly) to avoid overwriting the errno value for other code elsewhere. Perl's internals make system calls sometimes, so the value of this variable can change out from under you. Copy it *immediately* after causing a system call for accurate results.

$" ($LIST_SEPARATOR) contains the string used to separate array and list elements interpolated into a string.

%+ contains named captures from successful regular expression matches (Named Captures, pp. 100).

$@ ($EVAL_ERROR) contains the value thrown from the most recent exception (Catching Exceptions, pp. 127).

`$0` (`$PROGRAM_NAME`) contains the name of the program currently executing. You may modify this value on some Unix-like platforms to change the name of the program as it appears to other programs on the system, such as `ps` or `top`.

`$$` (`$PID`) contains the process id of the currently running instance of the program as the operating system understands it. This will vary between `fork()`ed programs and *may* vary between threads in the same program.

`@INC` holds a list of filesystem paths in which Perl will look for files to load with `use` or `require`. See `perldoc -f require` for other items this array can contain.

`%SIG` maps OS and low-level Perl signals to function references used to handle those signals. Trap the standard Ctrl-C interrupt by catching the `INT` signal, for example. See `perldoc perlipc` for more information about signals and signal handling.

Alternatives to Super Globals

IO and exceptions are the worst perpetrators of action at a distance. Use `Try::Tiny` (Exception Caveats, pp. 128) to insulate you from the tricky semantics of proper exception handling. `local`ize and copy the value of `$!` to avoid strange behaviors when Perl makes implicit system calls. Use `IO::File` and its methods on lexical filehandles (Special File Handling Variables, pp. 141) to limit unwanted global changes to IO behavior.

CHAPTER 11

What to Avoid

Perl is a malleable language. You can write programs in whichever creative, maintainable, obfuscated, or bizarre fashion you prefer. Good programmers write code that they want to maintain, but Perl won't decide for you what *you* consider maintainable.

Perl isn't perfect. Some features are difficult to use correctly. Others seem great but don't work all that well. Some have strange edge cases. Knowing what to avoid in Perl–and when to avoid it–will help you write robust code that survives the twin tests of time and real users.

Barewords

Perl's parser understands Perl's builtins and operators. It uses sigils to identify variables and other punctuation to recognize function and method calls. Yet sometimes the parser has to guess what you mean, especially when you use a *bareword*–an identifier without a sigil or other syntactically significant punctuation.

Good Uses of Barewords

Though the `strict` pragma (Pragmas, pp. 129) rightly forbids ambiguous barewords, some barewords are acceptable.

Bareword hash keys

Hash keys in Perl are usually *not* ambiguous because the parser can identify them as string keys; `pinball` in `$games{pinball}` is obviously a string.

Occasionally this interpretation is not what you want, especially when you intend to *evaluate* a builtin or a function to produce the hash key. To make these cases clear, pass arguments to the function, use parentheses, or prepend a unary plus to force the evaluation of the builtin:

```
# the literal 'shift' is the key
my $value = $items{shift};

# the value produced by shift is the key
my $value = $items{shift @_}

# the function returns the key
my $value = $items{myshift( @_ )}

# unary plus indicates the builtin shift
my $value = $items{+shift};
```

Bareword package names

Package names are also barewords. If your naming conventions rule that package names have initial capitals and functions do not, you'll rarely encounter naming collisions. Even still, Perl must determine how to parse `Package->method`. Does it mean "call a function named `Package()` and call `method()` on its return value?" or "Call a method named `method()` in the `Package` namespace?" The answer depends on the code Perl has already compiled when it encounters that method call.

Force the parser to treat `Package` as a package name by appending the package separator (`::`) or make it a literal string:

```
# probably a class method
Package->method;

# definitely a class method
Package::->method;

# a slightly less ugly class method
'Package'->method;

# package separator
my $q = Plack::Request::->new;

# unambiguously a string literal
my $q = 'Plack::Request'->new;
```

Almost no real code does this, but it's unambiguous, so be able to read it.

Bareword named code blocks

The special named code blocks `AUTOLOAD`, `BEGIN`, `CHECK`, `DESTROY`, `END`, `INIT`, and `UNITCHECK` are barewords which *declare* functions without the `sub` builtin. You've seen this before (Code Generation, pp. 150):

```
package Monkey::Butler;

BEGIN { initialize_simians( __PACKAGE__ ) }

sub AUTOLOAD { ... }
```

While you *can* declare `AUTOLOAD()` without using `sub`, few people do.

Bareword constants

Constants declared with the `constant` pragma are usable as barewords:

```
# don't use this for real authentication
use constant NAME     => 'Bucky';
use constant PASSWORD => '|38fish!head74|';

return unless $name eq NAME && $pass eq PASSWORD;
```

These constants do *not* interpolate in double-quoted strings.

Constants are a special case of prototyped functions (Prototypes, pp. 170). When you predeclare a function with a prototype, the parser will treat all subsequent uses of that bareword specially–and will warn about ambiguous parsing errors. All other drawbacks of prototypes still apply.

Ill-Advised Uses of Barewords

No matter how cautiously you code, barewords still produce ambiguous code. You can avoid the worst abuses, but you will encounter several types of barewords in legacy code.

Bareword hash values

Some old code may not take pains to quote the *values* of hash pairs:

```
# poor style; do not use
my %parents = (
```

```
    mother => Annette,
    father => Floyd,
);
```

When neither the `Floyd()` nor `Annette()` functions exist, Perl will interpret these barewords as strings. `strict 'subs'` will produce an error in this situation.

Bareword function calls

Code written without `strict 'subs'` may use bareword function names. Adding parentheses will make the code pass strictures. Use `perl -MO=Deparse,-p` (see `perldoc B::Deparse`) to discover how Perl parses them, then parenthesize accordingly.

Bareword filehandles

Prior to lexical filehandles (Filehandle References, pp. 57), all file and directory handles used barewords. You can almost always safely rewrite this code to use lexical filehandles. Perl's parser recognizes the special exceptions of `STDIN`, `STDOUT`, and `STDERR`.

Bareword sort functions

The second operand of the `sort` builtin can be the *name* of a function to use for sorting. While this is rarely ambiguous to the parser, it can confuse *human* readers. The alternative of providing a function reference in a scalar is little better:

```
# bareword style
my @sorted = sort compare_lengths @unsorted;

# function reference in scalar
my $comparison = \&compare_lengths;
my @sorted     = sort $comparison @unsorted;
```

The second option avoids the use of a bareword, but the result is longer. Unfortunately, Perl's parser *does not* understand the single-line version due to the special parsing of `sort`; you cannot use an arbitrary expression (such as taking a reference to a named function) where a block or a scalar might otherwise go.

```
# does not work
my @sorted = sort \&compare_lengths @unsorted;
```

In both cases, the way `sort` invokes the function and provides arguments can be confusing (see `perldoc -f sort` for the details). Where possible, consider using the block form of `sort` instead. If you must use either function form, add a comment about what you're doing and why.

Indirect Objects

Perl is not a pure object-oriented language. It has no operator `new`; a constructor is anything which returns an object. By convention, constructors are class methods named `new()`, but you can name these methods anything you want, or even use *functions*. Several old Perl OO tutorials promote the use of C++ and Java-style constructor calls:

```
my $q = new Alces; # DO NOT USE
```

. . . instead of the obvious method call:

```
my $q = Alces->new;
```

These examples produce equivalent behavior, except when they don't.

Bareword Indirect Invocations

In the indirect object form (more precisely, the *dative* case) of the first example, the method precedes the invocant. This is fine in spoken languages where verbs and nouns are more obvious, but it introduces parsing ambiguities in Perl.

Because the method's name is a bareword (Barewords, pp. 166), the parser uses several heuristics to figure out the proper interpretation of this code. While these heuristics are well-tested and *almost* always correct, their failure modes are confusing. Things get worse when you pass arguments to a constructor:

```
my $obj = new Class( arg => $value ); # DO NOT USE
```

In this example, the *name* of the class looks like a function call. Perl can and does often get this right, but its heuristics depend on which package names the parser has seen, which barewords it has already resolved, how it resolved those barewords, and the *names* of functions already declared in the current package. For an exhaustive list of these conditions, you have to read the source code of Perl's parser–not something the average Perl programmer wants to do (see `intuit_method` in *toke.c*, if you're really curious–but feel free to forget this suggestion ever existed).

Imagine running afoul of a prototyped function (Prototypes, pp. 170) with a name which just happens to conflict somehow with the name of a class or a method called indirectly, such as a poorly-named `JSON()` method in the same file where the `JSON` module is used, to pick an example that actually happened. This is rare, but it's very unpleasant to debug. Avoid indirect invocations instead.

Indirect Notation Scalar Limitations

Another danger of the indirect syntax is that the parser expects a single scalar expression as the object. Printing to a filehandle stored in an aggregate variable *seems* obvious, but it is not:

```
# DOES NOT WORK
say $config->{output} 'Fun diagnostic message!';
```

Perl will attempt to call `say` on the `$config` object.

`print`, `close`, and `say`–all builtins which operate on filehandles–operate in an indirect fashion. This was fine when filehandles were package globals, but lexical filehandles (Filehandle References, pp. 57) make the indirect object syntax problems obvious. To solve this, disambiguate the subexpression which produces the intended invocant:

```
say {$config->{output}} 'Fun diagnostic message!';
```

Alternatives to Indirect Notation

Direct invocation notation does not suffer this ambiguity problem. To construct an object, call the constructor method on the class name directly:

```
my $q   = Plack::Request->new;
my $obj = Class->new( arg => $value );
```

This syntax *still* has a bareword problem in that if you have a function named `Request` in the `Plack` namespace, Perl will interpret the bareword class name as a call to the function, as:

```
sub Plack::Request;

# you wrote Plack::Request->new, but Perl saw
my $q = Plack::Request()->new;
```

Disambiguate this syntax as usual (Bareword package names, pp. 166).

For the limited case of filehandle operations, the dative use is so prevalent that you can use the indirect invocation approach if you surround your intended invocant with curly brackets. You *can* use methods on lexical filehandles, though almost no one ever does this for `print` and `say`.

The CPAN module `Perl::Critic::Policy::Dynamic::NoIndirect` (a plugin for `Perl::Critic`) can analyze your code to find indirect invocations. The CPAN module `indirect` can identify and prohibit their use in running programs:

```
# warn on indirect use
no indirect;

# throw exceptions on their use
no indirect ':fatal';
```

Prototypes

A *prototype* is a piece of metadata attached to a function or variable. A function prototype changes how Perl's parser understands it.

Prototypes allow you to define your own functions which behave like builtins. Consider the builtin `push`, which takes an array and a list. While Perl would normally flatten the array and list into a single list passed to `push`, Perl knows to treat the array as a *container* and does not flatten its values. In effect, this is like passing a reference to an array and a list of values to `push`–because Perl's parser understands this is what `push` needs to do.

Function prototypes attach to function declarations:

```
sub foo         (&@);
sub bar         ($$) { ... }
my  $baz = sub (&&) { ... };
```

Any prototype attached to a forward declaration must match the prototype attached to the function declaration. Perl will give a warning if this is not true. Strangely you may omit the prototype from a forward declaration and include it for the full declaration–but the only reason to do so is to win a trivia contest.

The builtin `prototype` takes the name of a function and returns a string representing its prototype.

To see the prototype of a builtin, prepend `CORE::` to its name for `prototype`'s operand:

```
$ perl -E "say prototype 'CORE::push';"
\@@
$ perl -E "say prototype 'CORE::keys';"
\%
$ perl -E "say prototype 'CORE::open';"
*;$@
```

`prototype` will return `undef` for those builtins whose functions you cannot emulate:

```
say prototype 'CORE::system' // 'undef'
# undef; cannot emulate builtin system

say prototype 'CORE::prototype' // 'undef'
# undef; builtin prototype has no prototype
```

Remember `push`?

```
$ perl -E "say prototype 'CORE::push';"
\@@
```

The `@` character represents a list. The backslash forces the use of a *reference* to the corresponding argument. This prototype means that `push` takes a reference to an array and a list of values. You might write `mypush` as:

```
sub mypush (\@@) {
    my ($array, @rest) = @_;
    push @$array, @rest;
}
```

Other prototype characters include `$` to force a scalar argument, `%` to mark a hash (most often used as a reference), and `&` to identify a code block. See `perldoc perlsub` for more information.

The Problem with Prototypes

Prototypes change how Perl parses your code and how Perl coerces arguments passed to your functions. While these prototypes may superficially resemble function signatures (Real Function Signatures, pp. 69) in other languages, they are very different. They do not document the number or types of arguments functions expect, nor do they map arguments to named parameters.

Prototype coercions work in subtle ways, such as enforcing scalar context on incoming arguments:

```
sub numeric_equality($$) {
    my ($left, $right) = @_;
    return $left == $right;
}

my @nums = 1 .. 10;

say 'They're equal, whatever that means!' if numeric_equality @nums, 10;
```

...but only work on simple expressions:

```
sub mypush(\@@);

# compilation error: prototype mismatch
# (expects array, gets scalar assignment)
mypush( my $elems = [], 1 .. 20 );
```

To debug this, users of `mypush` must know both that a prototype exists, and the limitations of the array prototype. That's a lot of cognitive burden to put on a user–and if you think this error message is inscrutable, wait until you see the *complicated* prototype errors.

Good Uses of Prototypes

Prototypes *do* have a few good uses that outweigh their problems. For example, you can use a prototyped function to override one of Perl's builtins. First check that you *can* override the builtin by examining its prototype in a small test program. Then use the `subs` pragma to tell Perl that you plan to override a builtin. Finally declare your override with the correct prototype:

```
use subs 'push';

sub push (\@@) { ... }
```

Beware that the `subs` pragma is in effect for the remainder of the *file*, regardless of any lexical scoping.

You may also usefully use prototypes to define compile-time constants. When Perl encounters a function declared with an empty prototype (as opposed to *no* prototype) *and* this function evaluates to a single constant expression, the optimizer will turn all calls to that function into constants instead of function calls:

```
sub PI () { 4 * atan2(1, 1) }
```

All subsequent code will use the calculated value of pi in place of the bareword `PI` or a call to `PI()`, with respect to scoping and visibility.

The core pragma `constant` handles these details for you. The `Const::Fast` module from the CPAN creates constant scalars which you can interpolate into strings.

A reasonable use of prototypes is to extend Perl's syntax to operate on anonymous functions as blocks. The CPAN module `Test::Exception` uses this to good effect to provide a nice API with delayed computation. Its `throws_ok()` function takes three arguments: a block of code to run, a regular expression to match against the string of the exception, and an optional description of the test:

```
use Test::More;
use Test::Exception;

throws_ok
    { my $unobject; $unobject->yoink }
    qr/Can't call method "yoink" on an undefined/,
    'Method on undefined invocant should fail';

done_testing();
```

The exported `throws_ok()` function has a prototype of `&$;$`. Its first argument is a block, which becomes an anonymous function. The second argument is a scalar. The third argument is optional.

Careful readers may have spotted the absence of a comma after the block. This is a quirk of Perl's parser, which expects whitespace after a prototyped block, not the comma operator. This is a drawback of the prototype syntax. If that bothers you, use `throws_ok()` without taking advantage of the prototype:

```
use Test::More;
use Test::Exception;

throws_ok(
    sub { my $unobject; $unobject->yoink() },
    qr/Can't call method "yoink" on an undefined/,
    'Method on undefined invocant should fail' );

done_testing();
```

`Test::Fatal` allows similar testing and uses a simpler approach to avoid this ambiguity.

Ben Tilly suggests a final good use of prototypes, to define a custom named function to use with `sort`:

```
sub length_sort ($$) {
    my ($left, $right) = @_;
    return length($left) <=> length($right);
}

my @sorted = sort length_sort @unsorted;
```

The prototype of `$$` forces Perl to pass the sort pairs in `@_`. `sort`'s documentation suggests that this is slightly slower than using the package globals `$a` and `$b`, but using lexical variables often makes up for any speed penalty.

Method-Function Equivalence

Perl's object system is deliberately minimal (Blessed References, pp. 118). A class is a package, and Perl does not distinguish between a function and a method stored in a package. The same builtin, `sub`, declares both. Perl will happily dispatch to a function called as a method. Likewise, you can invoke a method as if it were a function–fully-qualified, exported, or as a reference–if you pass in your own invocant manually.

Invoking the wrong thing in the wrong way causes problems.

Caller-side

Consider a class with several methods:

```
package Order {

    use List::Util 'sum';

    sub calculate_price {
        my $self = shift;
        return sum( 0, $self->get_items );
    }

    ...
}
```

Given an `Order` object `$o`, the following invocations of this method *may* seem equivalent:

```
my $price = $o->calculate_price;

# broken; do not use
my $price = Order::calculate_price( $o );
```

Though in this simple case, they produce the same output, the latter violates object encapsulation by bypassing method lookup.

If `$o` were instead a subclass or allomorph (Roles, pp. 112) of `Order` which overrode `calculate_price()`, that example has just called the wrong method. Any change to the implementation of `calculate_price()`, such as a modification of inheritance or delegation through `AUTOLOAD()`–might break calling code.

Perl has one circumstance where this behavior may seem necessary. If you force method resolution without dispatch, how do you invoke the resulting method reference?

```
my $meth_ref = $o->can( 'apply_discount' );
```

There are two possibilities. The first is to discard the return value of the `can()` method:

```
$o->apply_discount if $o->can( 'apply_discount' );
```

The second is to use the reference itself with method invocation syntax:

```
if (my $meth_ref = $o->can( 'apply_discount' )) {
    $o->$meth_ref();
}
```

When `$meth_ref` contains a function reference, Perl will invoke that reference with `$o` as the invocant. This works even under strictures, as it does when invoking a method with a scalar containing its name:

```
my $name = 'apply_discount';
$o->$name();
```

There is one small drawback in invoking a method by reference; if the structure of the program changes between storing the reference and invoking the reference, the reference may no longer refer to the most appropriate method. If the `Order` class has changed such that `Order::apply_discount` is no longer the right method to call, the reference in `$meth_ref` will not have updated.

That's an unlikely circumstance, but limit the scope of a method reference when you use this invocation form just in case.

Callee-side

Because it's *possible* (however inadvisable) to invoke a given function as a function or a method, it's possible to write a function callable as either.

The `CGI` module has these two-faced functions. Every one of them must apply several heuristics to determine whether the first argument is an invocant. This causes problems. It's difficult to predict exactly which invocants are potentially valid for a given method, especially when you may have to deal with subclasses. Creating an API that users cannot easily misuse is more difficult too, as is your documentation burden. What happens when one part of the project uses the procedural interface and another uses the object interface?

If you *must* provide a separate procedural and OO interface to a library, create two separate APIs.

Automatic Dereferencing

Perl can automatically dereference certain references on your behalf. Given an array reference in `$arrayref`, you can write:

```
push $arrayref, qw( list of values );
```

Given an expression which returns an array reference, you can do the same:

```
push $houses{$location}[$closets], \@new_shoes;
```

The Autoderef Experiment

After Perl 5.18, you must enable this feature with `use experimental 'autoderef';`. That should be a sign to tread carefully here.

The same goes for the array operators `pop`, `shift`, `unshift`, `splice`, `keys`, `values`, and `each` and the hash operators `keys`, `values`, and `each`. If the reference provided is not of the proper type–if it does not dereference properly–Perl will throw an exception. While this may seem more dangerous than explicitly dereferencing references directly, it is in fact the same behavior:

```
my $ref = sub { ... };

# will throw an exception
push  $ref, qw( list of values );

# will also throw an exception
push @$ref, qw( list of values );
```

Unfortunately, this automatic dereferencing has two problems. First, it only works on plain variables. If you have a `blessed` array or hash, a `tied` hash, or an object with array or hash overloading, Perl will throw a runtime exception instead of dereferencing the reference.

Second, remember that `each`, `keys`, and `values` can operate on both arrays and hashes. You can't look at:

```
my @items = each $ref;
```

. . . and tell whether `@items` contains a list of key/value pairs or index/value pairs, because you don't know whether you should expect `$ref` to refer to a hash or an array. Yes, choosing good variable names will help, but this code is intrinsically confusing.

Neither of these drawbacks make this syntax *unusable* in general, but its rough edges and potential for confusing readers make it less useful than it could be.

Tie

Where overloading (Overloading, pp. 154) allows you to customize the behavior of classes and objects for specific types of coercion, a mechanism called *tying* allows you to customize the behavior of primitive variables (scalars, arrays, hashes, and filehandles). Any operation you might perform on a tied variable translates to a specific method call on an object.

For example, the `tie` builtin allows core `Tie::File` module to treat files as if they were arrays of records, letting you `push` and `pop` and `shift` as you see fit. (`tie` was intended to use file-backed stores for hashes and arrays, so that Perl could use data too large to fit in available memory. RAM was more expensive 20 years ago.)

The class to which you `tie` a variable must conform to a defined interface for a specific data type. Read `perldoc perltie` for an overview, then see the core modules `Tie::StdScalar`, `Tie::StdArray`, and `Tie::StdHash` for specific details. Start by inheriting from one of those classes, then override any specific methods you need to modify.

When Class and Package Names Collide

If `tie` weren't confusing enough, `Tie::Scalar`, `Tie::Array`, and `Tie::Hash` define the necessary interfaces to tie scalars, arrays, and hashes, but `Tie::StdScalar`, `Tie::StdArray`, and `Tie::StdHash` provide the default implementations.

Tying Variables

To tie a variable:

```
use Tie::File;
tie my @file, 'Tie::File', @args;
```

The first operand is the variable to tie. The second is the name of the class into which to tie it. `@args` is an optional list of arguments required for the tying function. In the case of `Tie::File`, `@args` should contain a valid filename.

Tying functions resemble constructors: `TIESCALAR`, `TIEARRAY()`, `TIEHASH()`, or `TIEHANDLE()` for scalars, arrays, hashes, and filehandles respectively. Each function returns a new object which represents the tied variable. Both `tie` and `tied` return this object, though most people use `tied` in a boolean context.

Implementing Tied Variables

To implement the class of a tied variable, inherit from a core module such as `Tie::StdScalar`, Then override the specific methods for the operations you want to change. In the case of a tied scalar, these are likely `FETCH` and `STORE`, possibly `TIESCALAR()`, and probably not `DESTROY()`.

Here's a class which logs all reads from and writes to a scalar:

```
package Tie::Scalar::Logged {
    use Tie::Scalar;
    use parent -norequire => 'Tie::StdScalar';

    sub STORE {
        my ($self, $value) = @_;
```

```
        Logger->log("Storing <$value> (was [$$self])", 1);
        $$self = $value;
    }

    sub FETCH {
        my $self = shift;
        Logger->log("Retrieving <$$self>", 1);
        return $$self;
    }
}
```

Assume that the `Logger` class method `log()` takes a string and the number of frames up the call stack of which to report the location.

Within the `STORE()` and `FETCH()` methods, `$self` works as a blessed scalar. Assigning to that scalar reference changes the value of the scalar. Reading from it returns its value.

Similarly, the methods of `Tie::StdArray` and `Tie::StdHash` act on blessed array and hash references, respectively. Again, `perldoc perltie` explains the methods tied variables support, such as reading or writing multiple values at once.

Isn't `tie` Fun?

The `-norequire` option prevents the `parent` pragma from attempting to load a file for `Tie::StdScalar`, as that module is part of the file *Tie/Scalar.pm*. That's right, there's no *.pm* file for `Tie::StdScalar`. Isn't this fun?

When to use Tied Variables

Tied variables seem like fun opportunities for cleverness, but they can produce confusing interfaces. Unless you have a very good reason for making objects behave as if they were builtin data types, avoid creating your own ties without good reason. `tied` variables are also much slower than builtin data types.

With that said, tied variables can help you debug tricky code (use the logged scalar to help you understand *where* a value changes) or to make certain impossible things possible (access large files without running out of memory). Tied variables are less useful as the primary interfaces to objects; it's often too difficult and constraining to try to fit your whole interface to that supported by `tie()`.

A final word of warning is a sad indictment of lazy programming: a lot of code goes out of its way to *prevent* use of tied variables, often by accident. This is unfortunate, but library code is sometimes fast and lazy with what it expects, and you can't always fix it.

CHAPTER 12

Next Steps with Perl

Perl isn't perfect, but it is malleable–because no single configuration is ideal for every programmer and every purpose. Some useful behaviors are available as core libraries. More are available from the CPAN. Effective Perl programmers take full advantage of the options available to them.

Useful Core Modules

Perl's language design process has always tried to combine practicality with expandability, but it was as impossible to predict the future in 1994 as it is in 2015. Perl 5 expanded the language and made the CPAN possible, but it also retained backwards compatibility with most Perl 1 code written as far back as 1987.

The best Perl code of 2015 is very different from the best Perl code of 1994 or the best Perl code of 1987, and part of that is due to its core library.

The strict Pragma

The `strict` pragma (Pragmas, pp. 129) allows you to forbid (or re-enable) various powerful language constructs which offer potential for accidental abuse.

`strict` forbids symbolic references, requires variable declarations (Lexical Scope, pp. 77), and prohibits the use of undeclared barewords (Barewords, pp. 166). While symbolic references are occasionally necessary (Using and Importing, pp. 144), the use of a variable as a variable name offers the possibility of subtle errors of action at a distance–or, worse, the possibility of poorly-validated user input manipulating private data for malicious purposes.

Requiring variable declarations helps to detect typos in variable names and encourages proper scoping of lexical variables. It's easier to see the intended scope of a lexical variable if all variables have `my` or `our` declarations in the appropriate scope.

`strict` takes effect in lexical scopes. See `perldoc strict` for more details.

The warnings Pragma

The `warnings` pragma (Handling Warnings, pp. 135) controls the reporting of various warning classes, such as attempting to stringify the `undef` value or using the wrong type of operator on values. It also warns about the use of deprecated features.

The most useful warnings explain that Perl had trouble understanding what you meant and had to guess at the proper interpretation. Even though Perl often guesses correctly, disambiguation on your part will ensure that your programs run correctly.

The `warnings` pragma takes effect in lexical scopes. See `perldoc perllexwarn` and `perldoc warnings` for more details.

Asking for More Help

If you use both `warnings` with `diagnostics`, you'll get expanded diagnostic messages for each warning present in your programs, straight out of `perldoc perldiag`. It's a great help when learning Perl, but be sure to disable `diagnostics` before deploying your program, lest you fill up your logs or expose debugging information to users.

The autodie Pragma

Perl leaves error handling (or error ignoring) up to you. If you forget to check the return value of every `open()` call, for example, you could try to read from a closed filehandle–or worse, lose data as you try to write to one. The `autodie` pragma changes this for you. If you write:

```
use autodie;

open my $fh, '>', $file;
```

. . . an unsuccessful `open()` call will throw an exception. Given that the most appropriate approach to a failed system call is throwing an exception, this pragma can remove a lot of boilerplate code and allow you the peace of mind of knowing that you haven't forgotten to check a return value.

One caveat of `autodie` is that it can be a sledgehammer when you need a finishing hammer; if you only need a couple of system calls checked for you, you can limit its imports accordingly. See `perldoc autodie` for more information.

Perl Version Numbers

If you encounter a piece of Perl code without knowing when it was written or who wrote it, can you tell which version of Perl it requires? If you have a lot of experience with Perl both before and after the release of Perl 5.10, you might remember which version added `say` and when `autodie` entered the core. Otherwise, you might have to guess, trawl through `perldelta` files, or use `CPAN::MinimumVersion` from the CPAN.

There's no requirement for you to add the minimum required Perl version number to all new code you write, but it *can* clarify your intentions. For example, if you've tested your code with Perl 5.18 and use only features present in Perl 5.18, write:

```
use 5.018;
```

. . . and you'll document your intent. You'll also make it easier for tools to identify the particular features of Perl you may or may not use in this code. If someone comes along later and proves that the code works just fine on Perl 5.14, you can change the version number–and you'll do so based on practical evidence.

What's Next?

Although Perl includes an extensive core library, it's not comprehensive. Many of the best modules are available outside of the core, from the CPAN (The CPAN, pp. 9). The `Task::Kensho` meta-distribution includes several other distributions which represent the best the CPAN has to offer. When you need to solve a problem, look there first.

The CPAN has plenty of other gems, though. For example, if you want to:

- *Access a database via SQL*, use the `DBI` module
- *Embed a lightweight, single-file database*, use the `DBD::SQLite` module
- *Manage your database schemas*, use `Sqitch`
- *Represent database entities as objects*, use `DBIx::Class`
- *Perform basic web programming*, use `Plack`
- *Use a powerful web framework*, use `Mojolicious`, `Dancer`, or `Catalyst`
- *Process structured data files*, use `Text::CSV_XS` (or `Text::CSV`)
- *Manage module installations for applications*, use `Carton`
- *Manipulate numeric data*, use `PDL`
- *Manipulate images*, use `Imager`

- *Access shared libraries*, use `FFI::Platypus`
- *Extract data from XML files*, use `XML::Rabbit`
- *Keep your code tidy*, use `Perl::Tidy`
- *Watch for problems beyond strictures and warnings*, use `Perl::Critic`

. . . and the list goes on. Skim the CPAN recent uploads page[1] frequently to see what's new and what's updated.

Thinking in Perl

As is true of any creative endeavor, learning Perl never stops. While "Modern Perl" describes how the best Perl programmers approach their craft, their techniques and tools always evolve. What's great in 2015 and 2016 might not have been imagined even five years ago–and the greatness of 2020 and beyond might be mere inklings in the mind of an enterprising Perl hacker right now.

Now you have the chance to shape that future. It's up to you to continue discovering how to make Perl work for you and how to make Perl better, whether learning from the global Perl community, perusing the documentation of the core and CPAN modules, and by careful practice, discovering what works for you and what helps you write the right code.

Perl's not perfect (though it improves, year after year, release after release). It can be as clean or as messy as you need it to be, depending on the problems you have to solve. It's up to you to use it well.

As a wise person once said, "May you do good things with Perl."

[1] http://search.cpan.org/recent

Index