SECOND EDITION

Intermediate Perl

Randal L. Schwartz, brian d foy, and Tom Phoenix

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Intermediate Perl, Second Edition
by Randal L. Schwartz, brian d foy, and Tom Phoenix

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Revision History for the :
See *http://oreilly.com/catalog/errata.csp?isbn=* for release details.

ISBN: ----
1334217228

Table of Contents

CHAPTER 1
Foreword

Perl's object-oriented mechanism is classic prestidigitation. It takes a collection of Perl's existing non-OO features such as packages, references, hashes, arrays, subroutines, and modules, and then—with nothing up its sleeve—manages to conjure up fully functional objects, classes, and methods. Seemingly out of nowhere.

That's a great trick. It means you can build on your existing Perl knowledge and ease your way into OO Perl development, without first needing to conquer a mountain of new syntax or navigate an ocean of new techniques. It also means you can progressively fine-tune OO Perl to meet your own needs, by selecting from the existing constructs the one that best suits your task.

But there's a problem. Since Perl co-opts packages, references, hashes, arrays, subroutines, and modules as the basis of its OO mechanism, to use OO Perl you already need to understand packages, references, hashes, arrays, subroutines, and modules.

And there's the rub. The learning curve hasn't been eliminated; it's merely been pushed back half a dozen steps.

So then: how are you going to learn everything you need to know about non-OO Perl so you can start to learn everything you need to know about OO Perl?

This book is the answer. In the following pages, Randal draws on two decades of using Perl, and four decades of watching *Gilligan's Island* and *Mr. Ed*, to explain each of the components of Perl that collectively underpin its OO features. And, better still, he then goes on to show exactly how to combine those components to create useful classes and objects.

So if you still feel like Gilligan when it comes to Perl's objects, references, and modules, this book is just what the Professor ordered.

And that's straight from the horse's mouth.

—Damian Conway, May 2003

Preface

Almost 20 years ago (nearly eternity in Internet Time), Randal Schwartz wrote the first edition of *Learning Perl*. In the intervening years, Perl itself has grown substantially from a "cool" scripting language used primarily by Unix system administrators to a robust object-oriented programming language that runs on practically every computing platform known to mankind, and maybe some that aren't.

Throughout its six editions, *Learning Perl* has remained about the same size, around 300 pages, and continued to cover much of the same material to remain compact and accessible to the beginning programmer. But there is much more to learn about Perl though.

Randal called the first edition of this book *Learning Perl Objects, References, and Modules*, and we renamed its update *Intermediate Perl*, but we like to think of it as just *Learning More Perl*.[1] This is the book that picks up where *Learning Perl* leaves off. We show you how to use Perl to write larger programs.

As in *Learning Perl*, we designed each chapter to be small enough to read in just an hour or so. Each chapter ends with a series of exercises to help you practice what you've just learned, and the answers in the appendix for your reference. And like *Learning Perl*, we've developed the material in this book for a teaching environment and use in that setting.

Unless we note otherwise, everything in this book applies equally well to Perl on any platform, whether that is Unix, Linux, Windows ActivePerl from ActiveState, Strawberry Perl, or any other modern implementation of Perl. To use this book you just need to be familiar with the material in *Learning Perl* and have the ambition to go further.

After you finish this book, you've seen most of the core Perl language concepts that you'll need. The next book in the series is *Mastering Perl*, which focuses on applying what you already know to writing effective and robust Perl applications as well as managing the Perl software development lifecycle.

1. Don't ask why it isn't called that. We must have had hundreds of emails on the subject. Okay, ask, since we know you're going to anyway. You never really stop learning Perl, so *Learning More Perl* doesn't really tell you much about the book. Our editor choose the name which tells you what to expect.

Structure of This Book

There are three major sections of this book. The first section deals with references, which are the key to complex data structures as well as object-oriented programming. The second section introduces objects and how Perl implements object-oriented programming. The third and last section deals with Perl's module structure, testing, and the community infrastructure for distributing your work.

You should read this book from front to back, stopping to do the exercises. Each chapter builds on preceding chapters, and we'll assume that you know the material from those chapters as we discuss new topics.

Chapter 2 - Introduction
: An introduction to the material.

Chapter 3 - Using Modules
: Use Perl's core modules as well as modules from other people. We're going to show you how to create your own modules later in the book, but until we do you can still use modules you already have.

Chapter 4 - Intermediate Foundations
: Pick up some intermediate Perl skills you'll need for the rest of the book.

Chapter 5 - Introduction to References
: Introduce a level of redirection to allow the same code to operate on different sets of data.

Chapter 6 - References and Scoping
: Learn how Perl manages to keep track of pointers to data, and an introduction to anonymous data structures and autovivification.

Chapter 7 - Complex Data Structures
: Create, access, and print arbitrarily deep and nested data structures including arrays of arrays and hashes of hashes.

Chapter 8 - Subroutine References
: Capture behavior as an anonymous subroutine which you create dynamically and execute later.

Chapter 9 - Filehandle References
: Store filehandles in scalar variables that you can easily pass around your program or store in data structures.

Chapter 10 - Regular Expression References
: Compile regular expressions without immediately applying them, and use them as building blocks for larger patterns.

Chapter 11 - Reference Tricks
: Sorting complex operations, the *Schwartzian Transform*, and working with recursively defined data.

Chapter 12 - Packages
: Build larger programs by separating code into separate files and namespaces.

Chapter 13 - New Distributions
: Create a Perl distribution as your first step toward object-oriented programming.

Chapter 14 - Introduction to Objects
: Work with classes, method calls, inheritance, and overriding.

Chapter 15 - Introduction to Testing
: Start to test your modules so you find problems with the code as you create it.

Chapter 16 - Objects with Data
: Add per-instance data, including constructors, getters, and setters.

Chapter 17 - Advanced Objects
: Use multiple inheritance, automatic methods, and references to filehandles.

Chapter 18 - Exporter
: How `use` works, and how we can decide what to exports, and how we can create our own import routines.

Chapter 19 - Object Destruction
: Add behavior to an object that is going away, including object persistence.

Chapter 20 - Moose
: Moose is an object framework available on CPAN.

Chapter 21 - Advanced Testing
: Learn more sophisticated testing techniques, including some of the most popular testing modules, code coverage, and profiling.

Chapter 22 - Advanced Perl Testing
: Test complex aspects of code and meta-code things such as documentation and test coverage.

Chapter 23 - Contributing to CPAN
: Share your work with the world by uploading it to CPAN.

Appendix A
: Where to go to get answers.

Conventions Used in This Book

The following typographic conventions are used in this book:

Constant width
: Used for function names, module names, filenames, environment variables, code snippets, and other literal text

Italics
: Used for emphasis and for new terms where they are defined

Comments and Questions

Please address comments and questions concerning this book to the publisher:

- O'Reilly Media
- 1005 Gravenstein Highway North
- Sebastopol, CA 95472
- (800) 998-9938 (in the United States or Canada)
- (707) 829-0515 (international/local)
- (707) 829-0104 (fax)

The web page for this book, which lists errata, examples, or any additional information, can be found at:

- *http://www.oreilly.com/catalog/lrnperlorm*

To comment or ask technical questions about this book, send email to:

- bookquestions@oreilly.com (*mailto:bookquestions@oreilly.com*)

For more information about books, conferences, Resource Centers, and the O'Reilly Network, see the O'Reilly web site at:

- *http://www.oreilly.com*

Acknowledgments

From Randal. In the preface of the first edition of *Learning Perl*, I acknowledged the Beaverton McMenamin's Cedar Hills Pub[2] just down the street from my house for the "rent-free booth-office space" while I wrote most of the draft on my Powerbook 140. Well, like wearing your lucky socks every day when your favorite team is in the playoffs, I wrote nearly all of this book (including these words) at the same brewpub, in hopes that the light of success of the first book will shine on me twice. (As I update this preface for the second edition, I can see that my lucky socks do indeed work!)

This McM's has the same great local microbrew beer and greasy sandwiches, but they've gotten rid of my favorite pizza bread, replacing it with new items like marionberry cobbler (a local treat) and spicy jambalaya. (And they added two booths, and put in some pool tables.) Also, instead of the Powerbook 140, I'm using a Titanium Powerbook, with 1,000 times more disk, 500 times more memory, and a 200-times-faster CPU running a real Unix-based operating system (OS X) instead of the limited MacOS. I also uploaded all of the draft sections (including this one) over my 144K cell-phone modem and emailed them directly to the reviewers, instead of having to wait to rush home to my 9600-baud external modem and phone line. How times have changed!

2. http://www.mcmenamins.com/

So, thanks once again to the staff of the McMenamin's Cedar Hills Pub for the booth space and hospitality.

Like the previous editions of *Learning Perl*, I also owe much of what I'm saying here and how I'm saying it to the students of Stonehenge Consulting Services who have given me immediate precise feedback (by their glazed eyes and awkwardly constructed questions) when I was exceeding the "huh?" factor threshold. With that feedback over many dozens of presentations, I was able to keep refining and refactoring the materials that paved the way for this book.

Speaking of which, those materials started as a half-day "What's new in Perl 5?" summary commissioned by Margie Levine of Silicon Graphics, in addition to my frequently presented onsite four-day Llama course (targeted primarily for Perl Version 4 at the time). Eventually, I got the idea to beef up those notes into a full course and enlisted fellow Stonehenge presenter Joseph Hall for the task. (He's the one that selected the universe from which the examples are drawn.) Joseph developed a two-day course for Stonehenge in parallel with his excellent *Effective Perl Programming* book, which we then used as the course textbook (until now).

Other Stonehenge instructors have also dabbled a bit in the "Packages, References, Objects, and Modules" course over the years, including Chip Salzenberg and Tad McClellan. But the bulk of the recent changes have been the responsibility of my senior trainer Tom Phoenix, who has been "Stonehenge employee of the month" so often that I may have to finally give up my preferred parking space. Tom manages the materials (just as Tad manages operations) so I can focus on being the president and the janitor of Stonehenge.

Tom Phoenix contributed most exercises in this book and a timely set of review notes during my writing process, including entire paragraphs for me to just insert in place of the drivel I had written. We work well as a team, both in the classroom and in our joint writing efforts. It is for this effort that we've acknowledged Tom as a coauthor, but I'll take direct blame for any parts of the book you end up hating: none of that could have possibly been Tom's fault.

And last but not least, a special thanks to brian d foy, who sheparded this book into its second revision, and wrote most of the changes between the previous edition and this edition.

Of course, a book is nothing without a subject and a distribution channel, and for that I must acknowledge longtime associates Larry Wall and Tim O'Reilly. Thanks guys, for creating an industry that has paid for my essentials, discretionary purchases, and dreams, for nearly 20 years.

And, as always, a special thanks to Lyle and Jack for teaching me nearly everything I know about writing and convincing me that I was much more than a programmer who might learn to write: I was also a writer who happened to know how to program. Thank you.

And to you, the reader of this book, for whom I toiled away the countless hours while sipping a cold microbrew and scarfing down a piece of incredible cheesecake, trying to avoid spilling on my laptop keyboard: thank you for reading what I've written. I sincerely hope I've contributed (in at least a small way) to your Perl proficiency. If you ever meet me on the street, please say "Hi."[3] I'd like that. Thank you.

From brian. I have to thank Randal first, since I learned Perl from the first edition of *Learning Perl*, and learned the rest teaching the Llama and Alpaca courses for Stonehenge Consulting. Teaching is often the best way to learn.

The most thanks has to go to the Perl community, the wonderfully rich and diverse group of people that have made it a pleasure to work with the language and make the tools, web sites, and modules that make Perl so useful. Many people have contributed indirectly to this book through my other work and discussions with them. There are too many to list, but if you've ever done anything with Perl with me, there's probably a little of you in this book.

From Both of Us. Thanks to our reviewers, XXX for providing comments on the draft of this book.

Thanks also to our many students who have let us know what parts of the course material have needed improvement over the years. It's because of you that we're all so proud of it today.

Thanks to the many Perl Mongers who have made us feel at home as we've visited your cities. Let's do it again sometime.

And finally, our sincerest thanks to our friend Larry Wall, for having the wisdom to share his really cool and powerful toys with the rest of the world so that we can all get our work done just a little bit faster, easier, and with more fun.

3. And yes, you can ask a Perl question at the same time. I don't mind.

CHAPTER 2
Introduction

Welcome to the next step in your understanding of Perl. You're probably here either because you want to learn to write programs that are more than 100 lines long or because your boss has told you to do so.

See, our *Learning Perl* book was great because it introduced the use of Perl for short and medium programs (which is most of the programming done in Perl, we've observed). But, to avoid having "the Llama book" be big and intimidating, we deliberately and carefully left a lot of information out.

In the pages that follow, you can get "the rest of the story" in the same style as our friendly Llama book. It covers what you need to write programs that are 100 to 10,000 (or even longer) lines long.

For example, you'll learn how to work with multiple programmers on the same project by writing reuseable Perl modules that you can wrap in distributions useable by the common Perl tools. This is great, because unless you work 35 hours each day, you'll need some help with larger tasks. You'll also need to ensure that your code all fits with the other code as you develop it for the final application.

This book will also show you how to deal with larger and more complex data structures, such as what we might casually call a "hash of hashes" or an "array of arrays of hashes of arrays." Once you know a little about references, you're on your way to arbitrarily complex data structures that can make your life much easier.

And then there's the buzzworthy notion of object-oriented programming, which allows parts of your code (or hopefully code from others) to be reused with minor or major variations within the same program. The book will cover that as well, even if you've never seen objects before.

An important aspect of working in teams is having a release cycle and tests for unit and integration testing. You'll learn the basics of packaging your code as a distribution and providing unit tests for that distribution, both for development and for verifying that your code works in the ultimate end environment.

And, just as was promised and delivered in *Learning Perl*, we'll entertain you along the way by interesting examples and bad puns. We've sent Fred and Barney and Betty and Wilma home, though. A new cast of characters will take the starring roles.)

What Should You Know Already?

We'll presume that you've already read *Learning Perl*, using at least the fifth edition, or at least pretend you have, and that you've played enough with Perl to already have those basics down. For example, you won't see an explanation in this book that shows how to access the elements of an array or return a value from a subroutine.

Make sure you know the following things, all of which we covered in *Learning Perl*:

- How to run a Perl program on your system
- The three basic Perl variable types: scalars, arrays, and hashes
- Control structures such as `while`, `if`, `for`, and `foreach`
- Subroutines
- Basic regular expressions
- Perl operators such as `grep`, `map`, `sort`, and `print`
- File manipulation such as `open`, file reading, and `-X` (file tests)

You might pick up deeper insight into these topics in this book, but we're going to presume you know the basics.

strict and warnings

We introduced the `strict` and `warnings` pragmas in *Learning Perl*, and we expect that you'll use them for all of your code. However, for most of the code that you'll see in this book, we're going to assume that we've already turned on `strict` and `warnings` so we don't distract from the examples with repeated boilerplate code, just like we leave off the shebang line and the usual documentation bits. When we present full examples, we'll include these pragmas as well.

You might want to do what we do. Instead of starting a program from scratch, we open a template that has the usual bits of code in it. Until you develop your own template, complete with standard documentation and your favorite way of doing things, you can start with this simple one that you can assume is around all of our code examples:

```
#!/usr/local/bin/perl
use strict;
use warnings;



__END__
```

Perl 5.14

This book is current up to at least Perl 5.14, released in 2011, and some of the modules we discuss might have updates until the last minutes before the ink hits the paper. Since we generally present the basic ideas of Perl and usually only brief overviews of modules, you should always check the modules' documentation for any updates.

Some of the newer features require us to explicitly state that we want to use them so they don't disturb programs targetting earlier versions of Perl. The easiest way to enable these features is to tell Perl which version we require. The number `5.014` has to have three digits after the decimal point (in case there is ever a Perl 5.140):

```
use 5.014;

say "Hello World!";
```

You can also write this with the `v` notation and its multiple parts:

```
use v5.14.1;
```

Whenever we write some code that requires a feature from a specific version of `perl`, we'll insert that `use 5.016` line using the first version which made that feature available. If we can we'll also show a version of the code that can work with earlier versions of Perl if we can. We consider Perl 5.8, first released in 2002, to be the earliest version that anyone should use, so code samples that don't specify a version assume Perl 5.8. In general, we strive to write code that works for as many people and as many versions of Perl as possible, but we also want you to be as up-to-date as you wish to be.

To learn more about some of the basics of Perl 5.14, you might want to check out *Learning Perl, Sixth Edition*.

What About All Those Footnotes?

Like *Learning Perl*, this book relegates some of the more esoteric items out of the way for the first reading and places those items in footnotes.[1] You should skip those the first time through and pick them up on a rereading. You will not find anything in a footnote that you'll need to understand any of the material we present later.

What's with the Exercises?

It's very important that you do the exercises. Hands-on training gets the job done better. The best way to provide this training is with a series of exercises after every half-hour to hour of presentation. Of course, if you're a speed reader, the end of the chapter may come a bit sooner than a half hour. Slow down, take a breather, and do the exercises!

1. Like this.

Each exercise has a "minutes to complete" rating. We intend for this rating to hit the midpoint of the bell curve, but don't feel bad if you take significantly longer or shorter. Sometimes it's just a matter of how many times you've faced similar programming tasks in your studies or jobs. Use the numbers merely as a guideline.

Every exercise has its answer in Appendix A. Again, try not to peek; you'll ruin the value of the exercise.

How to get help

As the book authors, we're always glad to help when we can, but we're already inundated with more email than we can manage. There are several online resources where you can get help, either from us directly, or many of the other helpful people in the Perl community.

- Stackoverflow - *http://www.stackoverflow.com*

  Stackoverflow is a no-pay question-and-answer site for all sorts of programming questions, and there are many clueful Perlers who regularly answer your questions. You're likely to get very good answers within an hour, and for free. You might even get an answer from one of the authors.

- Perlmonks - *http://www.perlmonks.org*

  Perlmonks is an online Perl community where you can ask questions, post your thoughts on Perl, and interact with other Perlers. If there's something about Perl, people have probably already discussed it at Perlmonks. You can search the archives or start a new discussion.

- *learn@perl.org* and *http://learn.perl.org*

  The *learn@perl.org* mailing list is specifically designed as a safe place for Perl neophytes to ask questions without fear that you are bothering anyone. It's just waiting for your questions, no matter how basic you think they are.

- *module-authors@perl.org*

  If your question is specifically about writing and distributing modules, there's a special mailing list for that: *module-authors@perl.org*.

- *comp.lang.perl.misc*

  If usenet is more of your thing, you can ask questions on *comp.lang.perl.misc*. Several long-time Perl users monitor the group, and sometimes they are even helpful.

What if I'm a Perl Course Instructor?

If you're a Perl instructor who has decided to use this as your textbook, you should know that each set of exercises is short enough for most students to complete the whole set in 45 minutes to an hour, with a little time left over for a break. Some chapters'

exercises should be quicker, and some may take longer. That's because once all those little numbers in square brackets were written, we discovered that we don't know how to add.

So let's get started. Class begins after you turn the page...

CHAPTER 3
Using Modules

The killer feature of Perl is the Comprehensive Perl Archive Network, which we just call CPAN[1]. Perl already comes with many modules, but there are many more third-party modules available from CPAN. If we have a problem to solve or a task to complete with Perl, there's probably a module on CPAN that will help us. An effective Perl programmer is the one who uses CPAN wisely. We covered this briefly in *Learning Perl*, but we'll cover it again here. It's that important.

Module are the building blocks for our programs. They can provide reusable subroutines, variables, and even object-oriented classes. On our way to building our own modules, we'll show you some of those you might be interested in. We'll also look at the basics of using modules that others have already written.

As we noted in *Learning Perl*, we don't have to understand everything about modules and how they work on the inside to use them (although by the end of this book you should know much more about that). By following the examples in the module documentation, we can still get quite a bit done. To jump start our Perl, we'll start to use Perl modules right away even though we explain their mechanics and special syntax later.

The Standard Distribution

Perl comes with many of the popular modules already. Indeed, most of the over 66 MB of the Perl 5.14 distribution is from modules. In October 1996, Perl 5.003_07 had 98 modules. Today, in the middle of 2011, Perl 5.14.0 has 652[2]. Indeed, this is one of the advantages of Perl: it already comes with a lot of stuff that we need to make useful and complex programs without doing extra work ourselves.

1. See *http://www.cpan.org/* or one of its search interfaces, *http://search.cpan.org/* and *https://www.metacpan.org*
2. Use `Module::CoreList` to discover that count for yourself. That's what we did to get those numbers, after all.

Throughout this book, we'll try to identify which modules comes with `perl` (and in most cases, with which version they started coming with `perl`). We'll call these "core modules" or note that they're in "the standard distribution". If we have `perl`, we should have these modules. Since we're using Perl 5.14 as we write this, we'll assume that's the current version of Perl when we consider what's in core.

As we develop our code, we may want to consider if we want to use only core modules so that we can be sure that anyone with `perl` will have that module as long as they have at least the same version as us. We'll avoid that debate here, mostly because we love CPAN too much to do without it. We'll also show you have to figure out which modules come with which version of Perl in a moment.

Exploring CPAN

CPAN is certainly the most attractive feature of Perl, and it got that way by the hard work of some dedicated volunteers providing tools and services to make it easy for people to release quality software and easy for users to evaluate and install the modules. Although this isn't a comprehensive list of useful CPAN tools, it includes the services we most often use. Starting with this list, you'll quickly find the other useful services too.

CPAN Search - http://search.cpan.org
: The most popular and well-known CPAN search service is Graham Barr's CPAN Search. You can browse or search for modules, and each distribution page has links to the important facts and information about that distribution, including information from third-parties such as test results, bug reports, and so on.

MetaCPAN - https://www.metacpan.org
: MetaCPAN is the next generation discovery interface for CPAN. It does just about everything that CPAN Search does, but adds an API so you can write your own applications on top of their data.

CPAN Testers - http://testers.cpan.org
: Every module that an author uploads to CPAN is automatically tested. An army of testers downloads the current releases and tests them on their platforms. They send their results back to CPAN Testers, which collates all of the reports. As a module author, we have a free testing service. As a module user, we can check test reports to judge a distribution's quality or to see if it is likely to work on our setup.

CPANdeps - http://deps.cpantesters.org/
: David Cantrell went a bit further than CPAN Testers by combining information about module dependencies with test reports. Instead of relying solely on a distibutions own tests, you can see the liklihood of installation problems by noting the test results from the entire dependency chain. One of the frustrating tasks with any software installation is a failure in the middle of the process, and CPANdeps can help you head off those problems. As part of the service, David also maintains the

C5.6PAN and C5.8PAN, which are specialized versions of CPAN with only the latest version of each module that work on Perl 5.6 and Perl 5.8 respectively.

CPAN RT - http://rt.cpan.org
: RT is the issue tracker from Best Practical, and they've kindly set up a service for CPAN authors. Every module of CPAN automatically gets an issue queue in RT, and for many modules, RT is the main issue queue. Some authors may have other bug-tracking preferences, but RT is a good place to start.

Using Modules

Almost every Perl module comes with documentation, and even though we might not know how all of the magic behind-the-scenes works, we really don't have to worry about that stuff if we know how to use the interface. That's why the interface it's there, after all: to hide the details.

On our local machine, we can read the module documentation with the `perldoc` command[3]. We give it the module name we're interested in and it prints its documentation:

```
% perldoc File::Basename

NAME

   fileparse - split a pathname into pieces

   basename - extract just the filename from a path

   dirname - extract just the directory from a path

SYNOPSIS

   use File::Basename;

   ($name,$path,$suffix) = fileparse($fullname,@suffixlist)
   fileparse_set_fstype($os_string);
   $basename = basename($fullname,@suffixlist);
   $dirname = dirname($fullname);
```

We've included on the top portion of the documentation to show you the most important section (at least, the most important when you're starting). Module documentation typically follows the old Unix manpage format which starts with a NAME and SYNOPSIS section.

The synopsis gives us examples of the module's use, and if we can suspend understanding for a bit and follow the example, we can use the module. That is to say, it may be that you're not yet familiar with some of the Perl techniques and syntax in the synopsis, but you can generally just follow the example and make everything work.

3. You can also use the *http://perldoc.perl.org* website. It has the documentation for several versions of Perl, in both HTML and PDF formats.

Now, since Perl is a mix of procedural, functional, object-oriented, and other sorts of language types, Perl modules come in variety of different interfaces. We'll employ these modules in slightly different fashions, but as long as we can check the documentation, we shouldn't have a problem.

Functional Interfaces

To load a module, we use the Perl built-in `use`. We're not going to go into all of the details here, but we'll get to those in Chapter 12 and Chapter 18. At the moment we just want to use the module. Let's start with `File::Basename`, that same module from the core distribution. To load it into our script, we say:

```
use File::Basename;
```

When we do this, `File::Basename` introduces three subroutines, `fileparse`, `basename`, and `dirname`,[4] into our script[5]. From this point forward we can use the subroutines just as if we had defined them directly in the same file:

```
my $basename = basename( $some_full_path );
my $dirname  = dirname(  $some_full_path );
```

These routines pick out the filename and the directory parts of a pathname. For example, if `$some_full_path` were *D:\Projects\Island Rescue\plan7.rtf* (presumably, the program is running on a Windows machine), then `$basename` would be *plan7.rtf* and the `$dirname` would be *D:\Projects\Island Rescue*.

The `File::Basename` module knows what sort of system it's on, and thus its functions figure out how to correctly parse the strings for the different delimiters we might encounter.

However, suppose we already had a `dirname` subroutine? We've now overwritten it with the definition provided by `File::Basename`! If we had turned on warnings, we would have see a message stating that, but otherwise, Perl really doesn't care.

Selecting What to Import

Fortunately, we can tell the `use` operation to limit its actions by specifying a list of subroutine names following the module name, called the `import list`:

```
use File::Basename ('fileparse', 'basename');
```

Now the module only gives us those two subroutines and leaves our own `dirname` alone. Of course, this is awkward to type, so more often we'll see this written with the quotewords operator:

```
use File::Basename qw( fileparse basename );
```

4. As well as a utility routine, `fileparse_set_fstype`.

5. Actually, it imports them into the current package, but we haven't told you about those yet.

In fact, even if there's only one item, we tend to write it with a `qw( )` list for consistency and maintenance; often we'll go back to say "give me another one from here," and it's simpler if it's already a `qw( )` list.

We've protected the local `dirname` routine, but what if we still want the functionality provided by `File::Basename`'s `dirname`? No problem. We just spell it out with its full package specification:

```
my $dirname = File::Basename::dirname($some_path);
```

The list of names following `use` doesn't change which subroutines are defined in the module's package (in this case, `File::Basename`). We can always use the full name regardless of the import list, as in:[6]

```
my $basename = File::Basename::basename($some_path);
```

In an extreme (but extremely useful) case, we can specify an empty list for the import list, as in:

```
use File::Basename ();              # no import
my $base = File::Basename::basename($some_path);
```

An empty list is different from an absent list. An empty list says "don't give me anything," while an absent list says "give me the defaults." If the module's author has done her job well, the default will probably be exactly what we want.

Object-Oriented Interfaces

Contrast the subroutines imported by `File::Basename` with what another core module has by looking at `File::Spec`. The `File::Spec` module is designed to support operations commonly performed on file specifications. (A file specification is usually a file or directory name, but it may be a name of a file that doesn't exist—in which case, it's not really a filename, is it?)

Unlike the `File::Basename` module, the `File::Spec` module has a primarily object-oriented interface. We load the module with `use`, as we did before:

```
use File::Spec;
```

However, since this module has an object-oriented interface[7] it doesn't import any subroutines. Instead, the interface tells us to access the functionality of the module using its class methods. The `catfile` method joins a list of strings with the appropriate directory separator:

```
my $filespec = File::Spec->catfile( $homedir{gilligan},
  'web_docs', 'photos', 'USS_Minnow.gif' );
```

6. You don't need the ampersand in front of any of these subroutine invocations because the subroutine name is already known to the compiler following `use`.

7. We can use `File::Spec::Functions` if we want a functional interface.

This calls the class method `catfile` of the `File::Spec` class, which builds a path appropriate for the local operating system and returns a single string.[8] This is similar in syntax to the nearly two dozen other operations provided by `File::Spec`.

The `File::Spec` module provides several other methods for dealing with file paths in a portable manner. You can read more about portability issues in the `perlport` documentation.

A more typical object-oriented module: Math::BigInt

So as not to get dismayed about how "un-OO" the `File::Spec` module seems since it doesn't create objects, let's look at yet another core module, `Math::BigInt`, which can handle integers beyond Perl's native reach[9]:

```
use Math::BigInt;

my $value = Math::BigInt->new(2); # start with 2

$value->bpow(1000);               # take 2**1000

print $value->bstr, "\n";         # print it out
```

As before, this module imports nothing. Its entire interface uses class methods, such as `new`, against the class name to create instances, and then calls instance methods, such as `bpow` and `bstr`, against those instances.

Fancier output with modules

One of Perl's strengths is its reporting capabilities. You might think that is limited to just text, but with the right module, you can create just about any format. For instance, with `Spreadsheet::WriteExcel`, you can be the star of your office as you make not only useful, but nicely formatted Excel documents.

As you may already know from using the Excel application directly, you start with a workbook and put your stuff in worksheets. Starting with the code directly from the documentation, we easily create our first worksheet:

```
use Spreadsheet::WriteExcel;

# Create a new Excel workbook
my $workbook = Spreadsheet::WriteExcel->new('perl.xls');

# Add a worksheet
my $worksheet = $workbook->add_worksheet();
```

8. That string might be something like */home/gilligan/web_docs/photos/USS_Minnow.gif* on a Unix system. On a Windows system, it would typically use backslashes as directory separators. This module lets us write portable code easily, at least where file specs are concerned.

9. Behind the scenes, Perl is limited by the architecture its on. It's one of the few places where the hardware shows through.

From there, we can insert information. Like Excel, the module can track rows and columns as letters for the rows and a number for the column. To put something in the first cell, we use the `write` method by following the example in the documentation:

```
$worksheet->write( 'A1', 'Hello Excel!' );
```

Inside our program, however, it's probably easier to track both rows and columns as numbers, so `Spreadsheet::WriteExcel` does that too. The `write` method is smart enough to recognize which one we are using, although we have to remember that the module counts from zero, so the first row is 0 and the first column is 0:

```
$worksheet->write( 0, 0, 'Hello Excel' ); # in Excel's A1 cell
```

This allows us to do quite a bit already, but we can do even more by making our worksheet look a little prettier. First we have to create a format:

```
my $red_background = $workbook->add_format(
  color    => 'white',
  bg_color => 'red',
  bold     => 1,
  );

my $bold = $workbook->add_format(
  bold     => 1,
  );
```

Once we have a format, we can use it with our calls to `write` by supplying it as the last argument:

```
$worksheet->write( 0, 1, 'Colored cell', $red_background );
$worksheet->write( 0, 1, 'bold cell', $bold );
```

Besides `write`, there are several methods that handle specific types of data. If we wanted to insert the string `01234` exactly like that, we don't want Excel to ignore the leading `0`. Without giving Excel a hint, however, it does its best to guess what the data is. We tell Excel that it is a string by using `write_string`:

```
my $product_code = '01234';
$worksheet->write_string( 0, 2, $product_code );
```

There are several other specializations of the `write` method, so check out the module documentation to see what else you can put into a cell.

Besides data, we can also create formulas. We could use the `write_formula` method, but we our string starts with an `=`,

```
$worksheet->write( 'A2', 37 );
$worksheet->write( 'B2', 42 );
$worksheet->write( 'C2', '= A2 + B2');
```

There's a lot more to this module, and you should be able to quickly figure out its other feature by checking its documentation. We'll also show some more examples as we go through the chapters on references.

What's in Core?

Core, or *the Standard Library* or *Distribution* or *Version*, is the set of modules and add-ons that comes with the standard distribution (the one you'd download from CPAN). When people talk about "core", they are most often talking about the set of modules that you can count on any particular Perl having, usually so you can be sure that someone using your program doesn't need to install extra modules.

This situation has become a bit fuzzy, though. Some distributions, such as Strawberry Perl[10] or ActivePerl[11], add extra modules to its distribution. Some vendor versions, such as OS X, add modules to the perls they distribute with their operating system, or even change some of the the standard modules. Those situations aren't that annoying. Annoyance comes from the vendors that remove parts from the standard distribution or break up the standard distribution into multiple vendor packages so you have to work to get what you should already have[12].

The `Module::CoreList` module is really just a module that pulls together the historical information about the modules can came with the versions of Perl 5 and give us a programmatic way to access them. It's a mix of variables and object-like interfaces.

We can see which version of a module came with a particular version of Perl, which we specify with five digits after the decimal place (three for the minor version and two for the patch level):

```
use Module::CoreList;

print $Module::CoreList::version{5.01400}{CPAN}; # 1.9600
```

Sometimes we want to know it the other way around; which version of Perl first put a module into the standard library? `Module::Build` is the Perl build system, which we'll show in Chapter Chapter 13. `Module::CoreList` has been part of the Standard Library since Perl 5.8.9:

```
use Module::CoreList;

Module::CoreList->first_release('Module::Build'); # 5.009004
```

If we just want to check a module's first release, we don't need to write a program since one already comes with `Module::CoreList`. We run the `corelist` program:

```
$ corelist Module::Build

Module::Build was first released with perl 5.009004
```

10. *http://www.strawberryperl.com*
11. *http://www.activestate.com/activeperl*
12. According to perl's license, these vendor's aren't allowed to call these modified version "perl", but they do anyway.

If you have the a recent version of Perl, should should also already have `Module::CoreList`, which we know by using the module to find out about itself:

```
$ corelist

Module::CoreList was first released with perl 5.009002
```

The Comprehensive Perl Archive Network

CPAN is the result of many volunteers working together, many of whom were originally operating their own little (or big) Perl FTP sites back before that Web thing came along. They coordinated their efforts on the *perl-packrats* mailing list in late 1993 and decided that disk was getting cheap enough that the same information should be replicated on all sites rather than having specialization on each site. The idea took about a year to ferment, and Jarkko Hietaniemi established the Finnish FTP site as the CPAN mothership from which all other mirrors could draw their daily or hourly updates.

Part of the work involved rearranging and organizing the separate archives. Places were established for Perl binaries for non-Unix architectures, scripts, and Perl's source code itself. However, the modules portion has come to be the largest and most interesting part of the CPAN.

The modules in CPAN are organized as a symbolic-link tree in hierarchical functional categories, pointing to author directories where the actual files are located. The modules area also contains indicies that are generally in easy-to-parse-with-Perl formats, such as the `Data::Dumper` output for the detailed module index. Of course, these indicies are all derived automatically from databases at the master server using other Perl programs. Often, the mirroring of the CPAN from one server to another is done with a now-ancient Perl program called *mirror.pl*.

From its small start of a few mirror machines, CPAN has now grown to over 200 public archives in all corners of the Net, all churning away updating at least daily, sometimes as frequently as hourly. No matter where we are in the world, we can find a nearby CPAN mirror from which to pull the latest goodies.

The incredibly useful CPAN Search (*http://search.cpan.org*) will probably become your favorite interface. From that website, you can search for modules, look at their documentation, browse through their distributions, inspect their CPAN Testers reports, and many other things.

Installing Modules from CPAN

Installing a simple module from CPAN can be straightforward. We can use the `cpan` program that comes with Perl. We tell it which modules that we want to install. If we want to install the `Perl::Critic` module which can perform an automated code review, I just give `cpan` that module name:

```
$ cpan Perl::Critic
```

The first time we run this we might have to go through the configuration steps to initialize `CPAN.pm`, but after that it should get directly to work. The program downloads the module and starts to build it. If the module depends on other modules, `cpan` will automatically fetch and the build those as well.

If we start `cpan` with no arguments, we start the interactive shell from `CPAN.pm`. From the shell prompt, we can issue commands. We can install `Perl::Tidy`, the module which can clean up the formatting of Perl code:

```
$ cpan
cpan> install Perl::Tidy
```

To read about the other features of `cpan`, you can read its documentation with `perldoc`:

```
$ perldoc cpan
```

`CPANPLUS` became core with Perl 5.10, and it provides another programmatic interface to CPAN. It works much like `CPAN.pm`, but also has some extra features which we won't show here. `CPANPLUS` has the `cpanp` command, and we use the `-i` switch with it to install modules:

```
$ cpanp -i Perl::Tidy
```

Like `cpan`, we can start an interactive shell then install the module we need. In this case, we install the module that allows us to programmatically create Excel spreadsheets:

```
$ cpanp
CPAN Terminal> i Spreadsheet::WriteExcel
```

To read about the other features of `cpanp`, you can read its documentation with *perldoc*:

```
$ perldoc cpanp
```

CPANminus

There's another handy tool, *cpanm* (for *cpanminus*), although it doesn't come with Perl (yet). It's designed as a zero-conf, lightweight CPAN client that handles most of what most people want to do. You can simply download the single file from *http://xrl.us/cpanm* and follow its easy instructions to get started.

Once you have *cpanm*, you simply tell it which modules you want to install:

```
$ cpanm DBI WWW::Mechanize
```

Installing modules manually

We could also do the work ourseves that `cpan` does for us, which can at least be educational if you have never tried it before. If you understand what the tools are doing, you'll have an easier time tracking down problems as you run into them.

We download the module distribution archive, unpack it, and change into its directory. We use `wget` here, but it doesn't matter which tool you use. You have to find the exact URL to use, which you can get from one of the CPAN sites:

```
$ wget http://www.cpan.org/.../HTTP-Cookies-Safari-1.10.tar.gz
$ tar -xzf HTTP-Cookies-Safari-1.10.tar.gz
$ cd HTTP-Cookies-Safari-1.10
```

From there we go one of two ways (which we'll explain in detail in Chapter 13). If we find a file named *Makefile.PL*, we run this series of commands to build, test, and finally install the source:

```
$ perl Makefile.PL
$ make
$ make test
$ make install
```

If we don't have permission to install modules in the system-wide directories[13], we can tell perl to install them under another path by using the `INSTALL_BASE` argument:

```
$ perl Makefile.PL INSTALL_BASE=/Users/home/Ginger
```

To make perl look in that directory for modules, we can set the `PERL5LIB` environment variable. Perl adds those directories to its module directory search list. Here's how we'd do that for the Bourne shell:

```
$ export PERL5LIB=/Users/home/Ginger
```

We can also use the `lib` pragma to add to the module search path, although this is not as friendly since we have to change the code but also because it might not be the same directory on other machines where we want to run the code.

```
#!/usr/bin/perl
use lib qw(/Users/home/Ginger);
```

Backing up for a minute, if we found a *Build.PL* file instead of a *Makefile.PL*, the process is the same. These distributions use `Module::Build` to build and install code. Since `Module::Build` is not a core Perl module[14], we have to it before we could install the distribution that needs it.

```
$ perl Build.PL
$ perl Build
$ perl Build test
$ perl Build install
```

To install into our private directories using `Module::Build`, we add the `--install_base` parameter. We tell Perl how to find modules the same way we did before.

```
$ perl Build.PL --install_base /Users/home/Ginger
```

13. These directories were set when the administrator installed perl, and we can see them with *perl -V*.

14. At least not yet. It should be part of perl5.10 though

Sometimes we find both *Makefile.PL* and *Build.PL* in a distribution. What do we do then? We can use either one. Play favorites if you like.

Setting the Path at the Right Time

Perl finds modules by looking through the directories in the special Perl array, `@INC`. When your perl was compiled, a default list of directories was chosen for the module search path. You can see these in the output you get from running `perl` with the `-V` command-line switch:

```
$ perl -V
```

You can also write a Perl one-liner to print them:

```
$ perl -le "print for @INC"
```

The `use` statement executes at compile time, so it looks at the module search path, `@INC`, at compile time. That can break our program in hard-to-understand ways unless we take `@INC` into consideration. We need to make our `@INC` modifications before we try to load modules.

For example, suppose we have our own directory under */home/gilligan/lib*, and we installed our own `Navigation::SeatOfPants` module in */home/gilligan/lib/Navigation/SeatOfPants.pm*. When we load our module, Perl won't find it.

```
use Navigation::SeatOfPants;
```

Perl complains to us that it can't find the module in `@INC` and shows us all of the directories it has in that array:

```
Can't locate Navigation/SeatofPants.pm in @INC (@INC contains: ...)
```

You might think that we should just add our module directory to `@INC` before we call the `use`. However, even adding:

```
unshift @INC, '/Users/gilligan/lib';   # broken
use Navigation::SeatOfPants;
```

doesn't work. Why? Because the `unshift` happens at runtime, long after the `use` was attempted at compile time. The two statements are lexically adjacent but not temporally adjacent. Just because we wrote them next to each other doesn't mean they execute in that order. We want to change `@INC` before the `use` executes. One way to fix this is to add a `BEGIN` block around the `unshift`:

```
BEGIN { unshift @INC, '/Users/gilligan/lib'; }
use Navigation::SeatOfPants;
```

Now the `BEGIN` block compiles and executes at compile time, setting up the proper path for the following `use`.

However, this is noisy and prone to require far more explanation than you might be comfortable with, especially for the maintenance programmer who has to edit your code later. Let's replace all that clutter with that simple pragma we used before:

```
use lib '/Users/gilligan/lib';
use Navigation::SeatOfPants;
```

Here, the `lib` pragma takes one or more arguments and adds them at the beginning of the `@INC` array, just like `unshift` did before.[15] It works because it executes at compile time, not runtime. Hence, it's ready in time for the `use` immediately following.

Because a `use lib` pragma will pretty much always have a site-dependent pathname, it is traditional and we encouraged you to put it near the top of the file. This makes it easier to find and update when we need to move the file needs to a new system or when the `lib` directory's name changes. (Of course, we can eliminate `use lib` entirely if we can install our modules in a standard `@INC` locations, but that's not always practical.)

Think of `use lib` as not "use this library," but rather "use this path to find my libraries (and modules)." Too often, we see code written like:

```
use lib '/Users/gilligan/lib/Navigation/SeatOfPants.pm'; # WRONG
```

and then the programmer wonders why it didn't pull in the definitions. Be aware that `use lib` indeed runs at compile time, so this also doesn't work:

```
my $LIB_DIR = '/Users/gilligan/lib';
...
use lib $LIB_DIR;     # BROKEN
use Navigation::SeatOfPants;
```

Certainly Perl establishes the declaration of the `$LIB_DIR` variable at compile time (so we won't get an error with `use strict`, although the actual `use lib` should complain), but the actual assignment of the */home/gilligan/lib/* value doesn't happen until runtime. Oops, too late again!

At this point, we need to put something inside a `BEGIN` block or perhaps rely on yet another compile-time operation: setting a constant with `use constant`:

```
use constant LIB_DIR => '/Users/gilligan/lib';
...
use lib LIB_DIR;
use Navigation::SeatOfPants;
```

There. Fixed again. That is, until we need the library to depend on the result of a calculation. (Where will it all end? Somebody stop the madness!) This should handle about 99 percent of our needs.

We don't always have to know the path ahead of time either. In the previous examples we've hard-coded the paths. If we don't know what those will be because we're passing

15. `use lib` also unshifts an architecture-dependent library below the requested library, making it more valuable than the explicit counterpart presented earlier.

code around to several machines, the FindBin module, which comes with Perl, can help. It finds the full path to the script directory so we can use it to build paths.

```
use FindBin qw($Bin);
```

Now, in $Bin is the path to the directory that holds our script. If we have our libraries in the same directory, our next line can be:

```
use lib $Bin;
```

If we have the libraries in a directory close to the script directory, we put the right path components together to make it work:

```
use lib "$Bin/lib";    # in a subdirectory

use lib "$Bin/../lib"; # up one, then down into lib
```

So, if we know the relative path from the script directory, we don't have to hard-code the whole path. This makes the script more portable.

Some of these techniques will matter much more to you when you start writing your own modules in Chapter 13.

Setting the Path Outside the Program

The use lib has a big drawback. You have to put the library path in the source. You might have your local modules installed in one place, but your coworkers have them in another. You don't want to change the source every time you get it from a teammate, and you don't want to list everyone's locations in the source. Perl offers a couple of ways that you can extend the module search path without bothering the source.

Extending @INC with PERL5LIB

The Skipper must edit each program that uses his private libraries to include those lines from the previous section. If that seems like too much editing he can instead set the PERL5LIB environment variable to the directory name. For example, in the C shell, he'd use the line:

```
setenv PERL5LIB /home/skipper/perl-lib
```

In Bourne-style shells, he'd use something like:

```
export PERL5LIB=/home/skipper/perl-lib
```

The Skipper can set PERL5LIB once and forget about it. However, unless Gilligan has the same PERL5LIB environment variable his program will fail! While PERL5LIB is useful for personal use, we can't rely on it for programs we intend to share with others. (And we can't make our entire team of programmers add a common PERL5LIB variable. Believe us, we've tried.)

The PERL5LIB variable can include multiple directories separated by colons on Unix-like systems and semicolons on Windows-like systems (other than that you're on your own). Perl inserts all specified directories at the beginning of @INC:

```
$ export PERL5LIB=/home/skipper/perl-lib:/usr/local/lib/perl5 # Unix

C:> set PERL5LIB="C:/lib/skipper;C:/lib/perl5"   # Windows
```

While a system administrator might add a setting of PERL5LIB to a system-wide startup script, most people frown on that. The purpose of PERL5LIB is to enable non-administrators to extend Perl to recognize additional directories. If a system administrator wants additional directories he merely needs to recompile and reinstall Perl.

Extending @INC with -I

If Gilligan recognizes that one of the Skipper's programs is missing the proper directive, Gilligan can either add the proper PERL5LIB variable or invoke perl directly with one or more -I options. For example, to invoke the Skipper's get_us_home program, the command line might be something like:

```
perl -I/home/skipper/perl-lib /home/skipper/bin/get_us_home
```

Obviously, it's easier for Gilligan if the program itself defines the extra libraries. But sometimes just adding a -I fixes things right up.[16] This works even if Gilligan can't edit the Skipper's program. He still has to be able to read it, of course, but Gilligan can use this technique to try a new version of his library with the Skipper's program, for example.

local::lib

By default, the CPAN tools want to install new modules into the same directories where *perl* is, but you probably don't have permission to create files there, or might have to invoke some sort of administrator privilege to do so. This is a common problem with Perl neophytes because they don't realize how easy it is to install Perl modules anywhere they like. Once you know how to do that, you can install and use any module you like without bugging a sysadmin to do it for you.

The local::lib module, which you'll have to get from CPAN since it doesn't come with *perl* (yet), sets various environment variables that affect where CPAN clients install modules and where Perl programs will look for those modules. You can see what they set by loading the module on the command line using the -M switch, but without any-

16. Extending @INC with either PERL5LIB or -I also automatically adds the version- and architecture-specific subdirectories of the specified directories. Adding these directories automatically simplifies the task of installing Perl modules that include architecture- or version-sensitive components, such as compiled C code.

thing other arguments. In that case, `local::lib` prints out its settings using the Bourne shell commands you can stick right in one of your login files:

```
$ perl -Mlocal::lib
export PERL_LOCAL_LIB_ROOT="/Users/Ginger/perl5";
export PERL_MB_OPT="--install_base /Users/Ginger/perl5";
export PERL_MM_OPT="INSTALL_BASE=/Users/Ginger/perl5";
export PERL5LIB="...";
export PATH="/Users/Ginger/perl5/bin:$PATH";
```

The trick is installing `local::lib` so you can start using it. You can bootstrap `local::lib` by downloading and installing the module by hand:

```
$ perl Makefile.PL --bootstrap
$ make install
```

Once you have `local::lib`, you can use it with the CPAN tools. The *cpan* client supports `local::lib` if you use the `-I` switch to install modules[17]:

```
$ cpan -I Set::Crossproduct
```

The *cpanm* tool is a bit smarter. If you've already set the same environment variables `local::lib` would set for you, it uses them. If not, it checks the default module directories for write permissions. If you don't have write permissions, it automatically uses `local::lib` for you. If you want to be sure to use `local::lib` explicitly, you can do that:

```
$ cpanm --local-lib HTML::Parser
```

If you are using `local::lib`, you simply load that module in your program so your program knows where to find your installed modules.

```
# inside your Perl program
use local::lib;
```

We've shown you `local::lib` using the default settings, but it has a way to work with any path that you want to use. On the command line, you can give an import list to the module you load with `-M`:

```
$  perl5.14.2 -Mlocal::lib='~/perlstuff'
export PERL_LOCAL_LIB_ROOT="/Users/Ginger/perlstuff";
export PERL_MB_OPT="--install_base /Users/Ginger/perlstuff";
export PERL_MM_OPT="INSTALL_BASE=/Users/Ginger/perlstuff";
export PERL5LIB="/Users/Ginger/foo/lib/perl5/darwin-2level:/Users/Ginger/perlstuff/lib/perl5";
export PATH="/Users/Ginger/perlstuff/bin:$PATH";
```

Exercises

You can find the answers to these exercises in "This is section" on page 183.

17. You need a recent version of `CPAN.pm` or the `App::Cpan` module. The `local::lib` feature was added for the Perl v5.14

Exercise 1 [25 min]

Read the list of files in the current directory and convert the names to their full path specification. Don't use the shell or an external program to get the current directory. The `File::Spec` and `Cwd` modules, both of which come with Perl, should help. Print each path with four spaces before it and a newline after it.

Exercise 2 [35 min]

Parse the International Standard Book Number from the back of this book (we'd tell you what it is right here, but we don't know it yet. If you don't have the book, try 9780596102067, which is the ISBN from the previous edition of this book). Install the `Business::ISBN` module from CPAN and use it to extract the group code and the publisher code from the number.

Exercise 3 [20 min]

Install the `local::lib` module and use it when you install `Module::CoreList` (or another module if you like). Write a program that report the name and first release date for all the modules in Perl v5.14.2. Read the documentation for `local::lib` to see if it has special installation instructions.

CHAPTER 4
Intermediate Foundations

Before we get started on the meat of the book, we want to introduce some intermediate level Perl idioms that we use throughout the book. These are the things that typically set apart the beginning and intermediate Perl programmers. We'll also introduce you to the first cast of characters that we'll use in the examples throughout the book.

List operators

A list is just a collection of scalars, which, as you remember from *Learning Perl*, are just single values. Lists are the values themselves, and sometimes we store lists in arrays, the container that holds an ordered list. List operators do something with multiple elements, and most don't care if they use a literal list, the return values from a subroutine, or an array variable.

You already know about several list operators in Perl, but you may not have thought of them as working with lists. The most common list operator is probably `print`. We give it one or more arguments and it puts them together for us:

```
print 'Two castaways are ', 'Gilligan', ' and ', 'Skipper', "\n";
```

There are several other list operators that you already know about from *Learning Perl*. The `sort` operator puts its input list in order. In their theme song, the castaways don't come in alphabetical order, but sort can fix that for us:

```
my @castaways = sort qw(Gilligan Skipper Ginger Professor Mary-Ann);
```

The `reverse` operator returns a list in the opposite order.

```
my @castaways = reverse qw(Gilligan Skipper Ginger Professor Mary-Ann);
```

We can even use these operators "in-place" by having the same array on both the righthand and lefthand sides of the assignment. Perl figures out the righthand side first, knows the result, and then assigns that back to the original variable name:

```
my @castaways = qw(Gilligan Skipper Ginger Professor);
push @castaways, 'Mary-Ann';
```

```
@castaways = reverse @castaways;
```

Perl has many other operators that work with lists, and once you get used to them you'll find yourself typing less and expressing your intent more clearly.

List filtering with grep

The `grep` operator a "testing expression" and a list of values. It takes each item from the list in turn and places it into the `$_` variable. It then evaluates the testing expression in a scalar context. If the expression evaluates to a true value, `grep` passes `$_` on to the output list:

```
my @lunch_choices = grep is_edible($_), @gilligans_posessions.
```

In a list context, the `grep` operator returns a list of all such selected items. In a scalar context `grep` returns the number of selected items:

```
my @results = grep EXPR, @input_list;
my $count   = grep EXPR, @input_list;
```

Here, *`EXPR`* stands in for any scalar expression that should refer to `$_` (explicitly or implicitly). For example, to find all the numbers greater than 10, in our `grep` expression we check if `$_` is greater than `10`:

```
my @input_numbers = (1, 2, 4, 8, 16, 32, 64);
my @bigger_than_10 = grep $_ > 10, @input_numbers;
```

The result is just 16, 32, and 64. This uses an explicit reference to `$_`. Here's an example of an implicit reference to `$_` from the pattern match operator:

```
my @end_in_4 = grep /4$/, @input_numbers;
```

And now we get just 4 and 64.

While the `grep` is running, it shadows any existing value in `$_`; which is to say that `grep` borrows the use of this variable, but puts the original value back when it's done. The variable `$_` isn't a mere copy of the data item, though; it is an alias for the actual data element, similar to the control variable in a `foreach` loop.

If the testing expression is complex, we can hide it in a subroutine:

```
my @odd_digit_sum = grep digit_sum_is_odd($_), @input_numbers;

sub digit_sum_is_odd {
  my $input = shift;
  my @digits = split //, $input;  # Assume no nondigit characters
  my $sum;
  $sum += $_ for @digits;
  return $sum % 2;
}
```

Now we get back the list of 1, 16, and 32. These numbers have a digit sum with a remainder of "1" in the last line of the subroutine, which counts as true.

The syntax comes in two forms, though: we just showed you the expression form, and now here's the block form. Rather than define an explicit subroutine that we'd use for only a single test, we can put the body of a subroutine directly in line in the `grep` operator, using the block forms. In the block form of `grep`, there's no comma between the block and the input list[1]:

```
my @results = grep {
  block;
  of;
  code;
} @input_list;

my $count = grep {
  block;
  of;
  code;
} @input_list;
```

Just like the expression form, `grep` temporarily places each element of the input list into `$_`. Next, it evaluates the entire block of code. The last evaluated expression in the block is the testing expression, and like all testing expressions, it's evaluated in a scalar context. Because it's a full block, we can introduce variables that are scoped to the block. Let's rewrite that last example to use the block form:

```
my @odd_digit_sum = grep {
  my $input = $_;
  my @digits = split //, $input;   # Assume no nondigit characters
  my $sum;
  $sum += $_ for @digits;
  $sum % 2;
} @input_numbers;
```

Note the two changes: the input value comes in via `$_` rather than an argument list, and we removed the keyword `return`. In fact, we would have been wrong to keep the `return` because we're no longer in a separate subroutine: just a block of code.[2] Of course, we can optimize a few things out of that routine since we don't need the intermediate variables:

```
my @odd_digit_sum = grep {
  my $sum;
  $sum += $_ for split //;
  $sum % 2;
} @input_numbers;
```

We can do whatever we link in that `grep` block. Suppose we have a list of URLs in `@links` and we want to know which ones aren't good anymore. We can send that list

1. This is a little quirk of Perl syntax. That block of code is really an anonymous subroutine, just like the ones that you saw with `sort` in *Learning Perl* and that we'll talk about in Chapter 8.
2. The `return` would have exited the subroutine that contains this entire section of code. And yes, some of us have been bitten by that mistake in real, live coding on the first draft.

of links through the grep, check them with HTTP::SimpleLinkChecker (available on CPAN), and only pass through the links that don't have an error:

```
use HTTP::SimpleLinkChecker qw(check_link);

my @bad_links = grep {
  check_link( $_ );
  ! $HTTP::SimpleLinkChecker::ERROR;
} @links;
```

Feel free to crank up the explicitness if it helps you and your coworkers understand and maintain the code. That's the main thing that matters.

Transforming lists with map

The map operator has a very similar syntax to the grep operator and shares a lot of the same operational steps. For example, it temporarily places items from a list into $_ one at a time, and the syntax allows both the expression block forms.

However, the expression is for transformation instead of testing. The map operator evaluates the expression in a list context, not a scalar context like grep. Each evaluation of the expression gives a portion of the list that comes the final list. The overall result is the list concatenation of all individual results. In a scalar context, map returns the number of elements that are returned in a list context. But map should rarely, if ever, be used in anything but a list context.

Let's start with a simple example:

```
my @input_numbers = (1, 2, 4, 8, 16, 32, 64);
my @result = map $_ + 100, @input_numbers;
```

For each of the seven items map places into $_, we get a single output result: the number that is 100 greater than the input number, so the value of @result is 101, 102, 104, 108, 116, 132, and 164.

But we're not limited to having only one output for each input. Let's see what happens when each input produces two output items:

```
my @result = map { $_, 3 * $_ } @input_numbers;
```

Now there are two items for each input item: 1, 3, 2, 6, 4, 12, 8, 24, 16, 48, 32, 96, 64, and 192. We can store those pairs in a hash, if we need a hash showing what number is three times a small power of two:

```
my %hash = @result;
```

Or, without using the intermediate array from the map:

```
my %hash = map { $_, 3 * $_ } @input_numbers;
```

That was fine for making a meaningful value for each hash key, but sometimes we don't care about the value because we just want to use the hash as an easier way to check

that an element is in a list. In that case, we can give the keys any value just to have it in the hash. Using `1` is a good value to use:

```
my %hash = map { $_, 1 } @castaways;

my $person = 'Gilligan';

if( exists $hash{$person} ) {
  print "$person is a castaway.\n";
}
```

You can see that `map` is pretty versatile; we can produce any number of output items for each input item. And we don't always need to produce the same number of output items. Let's see what happens when we break apart the digits:

```
my @result = map { split //, $_ } @input_numbers;
```

The inline block of code splits each number into its individual digits. For 1, 2, 4, and 8, you get a single result. For 16, 32, and 64, we get two results per number. When `map` concatenates the results lists, we end up with 1, 2, 4, 8, 1, 6, 3, 2, 6, and 4.

If a particular invocation results in an empty list, `map` concatenates that empty result into the larger list, contributing nothing to the list. We can use this feature to select and reject items. For example, suppose we want only the split digits of numbers ending in 4:

```
my @result = map {
  my @digits = split //, $_;
  if ($digits[-1] == 4) {
    @digits;
  } else {
    (  );
  }
} @input_numbers;
```

If the last digit is 4, we return the digits themselves by evaluating `@digits`, which is in list context. If the last digit is not 4, we return an empty list, effectively removing results for that particular item. Thus, we can always use a `map` in place of a `grep`, but not vice versa.

Of course, everything we can do with `map` and `grep`, we can also do with explicit `foreach` loops. But then again, we can also code in assembler or by toggling bits into a front panel.[3] The point is that proper application of `grep` and `map` can help reduce the complexity of the program, allowing us to concentrate on high-level issues rather than details.

3. If you're old enough to remember those front panels.

Trapping errors with eval

Many lines of ordinary code have the potential to terminate a program prematurely if something goes wrong:

```
my $average = $total / $count;  # divide by zero?
print "okay\n" unless /$match/; # illegal pattern?

open MINNOW, '>ship.txt'
or die "Can't create 'ship.txt': $!";  # user-defined die?

implement($_) foreach @rescue_scheme;  # die inside sub?
```

Just because something has gone wrong with one part of our code, that doesn't mean that we want everything to crash. Perl uses the `eval` operator as its error trapping mechanism[4]:

```
eval { $average = $total / $count } ;
```

If an error happens while running code inside an `eval` block, the block stops executing. But even though the code inside the block is finished, perl continues running the code just after the `eval`. It's most common after an `eval` to immediately check `$@`, which will either be empty (meaning that there was no error) or the dying words Perl had from the code which failed, perhaps something like `"divide by zero"` or a longer error message:

```
eval { $average = $total / $count } ;
print "Continuing after error: $@" if $@;

eval { rescue_scheme_42() } ;
print "Continuing after error: $@" if $@;
```

The semicolon is needed after the `eval` block because `eval` is a function (not a control structure, such as `if` or `while`).

The block is a true block, and may include lexical variables ("`my`" variables) and any other arbitrary statements. As a function, `eval` has a return value much like a subroutine's (the last expression evaluated, or a value returned early by the `return` keyword). Of course, if the code in the block fails, it returns no value; this gives `undef` in a scalar context, or an empty list in a list context. Thus, another way to calculate an average safely looks like this:

```
my $average = eval { $total / $count };
```

Now `$average` is either the quotient or `undef`, depending upon whether the operation completed successfully or not.

Perl even supports nested `eval` blocks. The power of an `eval` block to trap errors extends for so long as it's executing, so it catches errors deep within nested subroutine calls.

4. The `eval` is Perl's primitive exception mechanism. See *Mastering Perl* for a deeper discussion of handling errors in Perl, including some modules that have more fancy exception frameworks

`eval` can't trap the most serious of errors, though: the ones in which perl itself stops running. These include things such as an uncaught signal, running out of memory, and other catastrophes. `eval` doesn't catch syntax errors, either; because perl compiles the `eval` block with the rest of the code, it catches syntax errors at compile time, not at run time. It doesn't catch warnings either (although Perl does provide a way to intercept warning messages; see `$SIG{__WARN__}`).

For very simple operations, a straight `eval` is fine. For reasons we won't bore you with here, correctly handling complex situations can be tricky. Fortunately for you, these problems are handled correctly by the `Try::Tiny` module (available on CPAN):

```
use Try::Tiny;
my $average = try { $total / $count } catch { "NaN" };
```

Dynamic code with `eval`

There's also a second form of `eval` whose parameter is a string expression instead of a block. It compiles and executes code from a string at runtime. While this is useful and supported, it is also dangerous if any untrustworthy data has gotten into the string. With a few notable exceptions, we recommend you avoid `eval` on a string. We'll use it a bit later, and you might see it in other people's code, so we'll show you how it works anyway:

```
eval '$sum = 2 + 2';
print "The sum is $sum\n";
```

Perl executes that code in the lexical context of the code around it, meaning that its as if we had typed that `eval`-ed code right there. The result of the `eval` is the last evaluated expression, so we really don't need the entire statement inside the `eval`:

```
#!/usr/bin/perl

foreach my $operator ( qw(+ - * /) ) {
  my $result = eval "2 $operator 2";
  print "2 $operator 2 is $result\n";
}
```

Here, we go through the operators `+ - * /` and use each of those inside our `eval` code. In the string we give to `eval`, we interpolate the value of `$operator` into the string. The `eval` executes the code that the string represents and returns the last evaluated expression, which we assign it to `$result`.

If `eval` can't properly compile and run the Perl code we hand it, it sets `$@` just like in its block form. In this example, we want to trap any divide-by-zero errors, but we don't divide by anything (another sort of error):

```
print 'The quotient is ', eval '5 /', "\n";
warn $@ if $@;
```

The `eval` catches the syntax error and puts the message in `$@`, which we check immediately after calling `eval`:

```
The quotient is
syntax error at (eval 1) line 2, at EOF
```

In case you didn't catch our warning before, we'll say it again: be very careful with this form of `eval`. If you can find another way to do what you need, try that first. We'll use it later, in Chapter 12 to load code from an external file, but then we'll also show you a much better way to do that too.

The do Block

The `do` block is one of powerful but overlooked features of Perl. It provides a way to group statements as a single expression that we can use in another expression. It's almost like an inline subroutine. As with subroutines, the result of `do` is the last evaluated expression.

First, consider a bit of code to assign one of three possible values to a variable. We declare `$bowler` as a lexical variable, and we use an `if-elsif-else` structure to choose which value to assign to it. We end up typing the variable name four times just to get a single assignment:

```
my $passenger;
if( ...some condition... ) {
  $bowler = 'Mary-Ann';
}
elsif( ... some condition ... ) {
  $bowler = 'Ginger';
}
else {
  $bowler = 'The Professor';
}
```

However, with `do`, we only have to use the variable name once. We can assign to it at the same time that we declare it because we can combine everything else in the `do` as if it were a single expression:

```
my $passenger = do {
  if( ... some condition ... )    { 'Mary-Ann' }
  elsif( ... some condition ... ) { 'Ginger' }
  else                            { 'The Professor' }
};
```

The `do` is also handy for creating a scope around an operation. We might want to slurp all of a file's contents into a variable. One Perl idiom for doing that uses `do` to provide a scope for `$/`, the input record separtor, and a localized version of `@ARGV` so we can use the `<>` to handle all of the filehandle details for us:

```
my $file_contents = do {
  local $/;
  local @ARGV = ( $filename );
  <>
};
```

Like `eval`, `do` has a string argument form. Given a string instead of a block of code, `do` attempts to load a file with that name and execute its code right there:

```
do $filename;
```

The `do` finds the file and reads it, then essentially hands off the contents to the string form of `eval` to execute it. In the previous chapter, you saw `use` as a way to load modules, and we told you that happened at compile time. There's another way to load modules. The `require` built-in also loads modules, but does it at run time:

```
require List::Util;
```

In fact, a `use` is really just a `require` in a `BEGIN` block and a call to the class's `import`:

```
BEGIN {   # what use is really doing
  require List::Util;
  List::Util->import(...);
}
```

You had to give `use` a module name, but you can give `require` a filename just like you could with `do`:

```
require $filename;
```

In either case, the c<require> remembers which files it has already loaded so it won't do the work to reload the same file.

We'll show more of this when you start writing your own modules.

Exercises

You can find the answers to these exercises in "This is section" on page 183.

Exercise 1 [15 min]

Write a program which takes a list of filenames on the command line and uses `grep` to select the ones whose size in bytes is less than 1,000 bytes. Use `map` to transform the strings in this list, putting four space characters in front of each and a newline character after. Print the resulting list.

Exercise 2 [25 min]

Write a program that asks the user to enter a pattern (regular expression). Read this as data from the keyboard; don't get it from the command line arguments. Report a list of files in some hard-coded directory (such as `"/etc"` or `'C:\\Windows'`) whose names match the pattern. Repeat this until the user enters an empty string instead of a pattern. The user should not type the forward slashes which are traditionally used to delimit pattern matches in Perl; the input pattern is delimited by the trailing newline. Ensure that a faulty pattern, such as one with unbalanced parentheses, doesn't crash the program.

CHAPTER 5

Introduction to References

References are the basis for complex data structures, object-oriented programming, and fancy subroutine handling. They're the magic that was added between Perl versions 4 and 5 to make it all possible.

A Perl scalar variable holds a single value. An array holds an ordered list of scalars. A hash holds a collection of scalars as values, keyed by strings. Although a scalar can be an arbitrary string, which allows us to encode complex data in an array or hash, none of the three data types are well-suited to complex data interrelationships. This is a job for the reference. Let's look at the importance of references by starting with an example.

Performing the Same Task on Many Arrays

Before the *Minnow* can leave on an excursion (for example, a three-hour tour), we should check every passenger and crew member to ensure they have all the required trip items in their possession. Let's say that for maritime safety every person on-board the *Minnow* needs to have a life preserver, some sunscreen, a water bottle, and a rain jacket. We can write a bit of code to check for the Skipper's supplies:

```
my @required = qw(preserver sunscreen water_bottle jacket);
my %skipper  = map { $_, 1 }
  qw(blue_shirt hat jacket preserver sunscreen);

foreach my $item (@required) {
  unless ( exists $skipper{$item} ) { # not found in list?
    print "Skipper is missing $item.\n";
  }
}
```

Notice that we created a hash from the list of Skipper's items. That's a very common and useful operation. Since we want to check it a particular item is in Skipper's list, the easiest way to make all the items keys of a hash then check the hash with `exists`. Instead of typing out the hash comptelely, we use the `map` to create it from the list of items.

Of course, if we want to check on Gilligan and the Professor, we might write the following code:

```
my %gilligan = map { $_, 1 } qw(red_shirt hat lucky_socks water_bottle);
foreach my $item (@required) {
  unless ( exists $gilligan{$item} ) { # not found in list?
    print "Gilligan is missing $item.\n";
  }
}

my @professor = map { $_, 1 }
  qw(sunscreen water_bottle slide_rule batteries radio);
for my $item (@required) {
  unless ( exists $professor{$item} ) { # not found in list?
    print "The Professor is missing $item.\n";
  }
}
```

You may start to notice a lot of repeated code here and think that we should refactor that into a common subroutine that we can reuse (and you'd be right):

```
sub check_required_items {
  my $who = shift;
  my %whos_items = map { $_, 1 } @_; # the rest are the person's items

  my @required = qw(preserver sunscreen water_bottle jacket);

  for my $item (@required) {
    unless ( exists $whos_items{$item} ) { # not found in list?
      print "$who is missing $item.\n";
    }
  }
}

my @gilligan = qw(red_shirt hat lucky_socks water_bottle);
check_required_items('gilligan', @gilligan);
```

Perl gives the subroutine five items in its argument list, the `@_` array[1], initially: the name `gilligan` and the four items belonging to Gilligan. After the `shift`, `@_` only has the items. Thus, the `grep` checks each required item against the list.

So far, so good. We can check the Skipper and the Professor with just a bit more code:

```
my @skipper   = qw(blue_shirt hat jacket preserver sunscreen);
my @professor = qw(sunscreen water_bottle slide_rule batteries radio);
check_required_items('skipper', @skipper);
check_required_items('professor', @professor);
```

And for the other passengers, we repeat as needed. Although this code meets the initial requirements, we've got two problems to deal with:

1. If you need to review the details of Perl subroutines, check out Chapter 4 in *Learning Perl* or the *perlsub* documentation

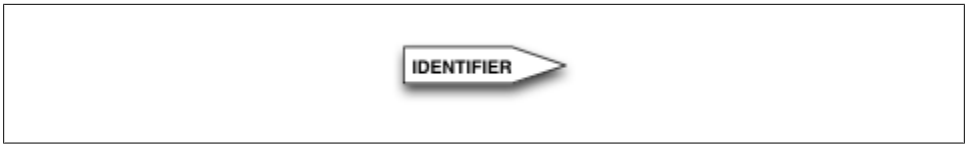

Figure 5-1. A partial PeGS diagram showing the identifier portion

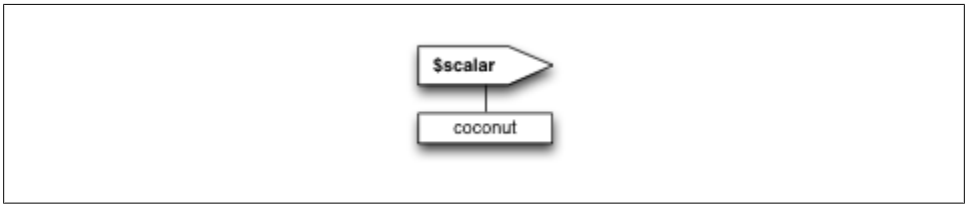

Figure 5-2. The PeGS diagram for a scalar

- To create `@_`, Perl copies the entire contents of the array we want to check. This is fine for a few items, but if the array is large, it seems a bit wasteful to copy the data just to pass it into a subroutine.
- Suppose we want to modify the original array to force the provisions list to include the mandatory items. Because we have a copy in the subroutine ("pass by value"), any changes we make to `@_` aren't reflected automatically in the corresponding provisions array.[2]

To solve either or both of these problems, we need pass by reference rather than pass by value. And that's just what the doctor (or Professor) ordered.

PeGS - Perl Graphical Structures

Before we get started with references, however, we want to introduce you to PeGS, a graphical representation of Perl data structures developed by Joseph Hall. With a pretty picture, some of these

Each PeGS diagram has two parts: the name of the variable and the data it contains. The name portion is at the top of the diagram as a box with a pointy right side. The variable name is inside the box:

For a scalar, we have a single box under the name to hold its single value:

For an array, which can have multiple value, we do a bit more. The data portion starts with a filled-in, solid bar at the top to denote that it is a collection (to distinguish the single element array from a scalar):

2. Actually, assigning new scalars to elements of `@_` after the `shift` modifies the corresponding variable being passed, but that still wouldn't let us extend the array with additional mandatory provisions.

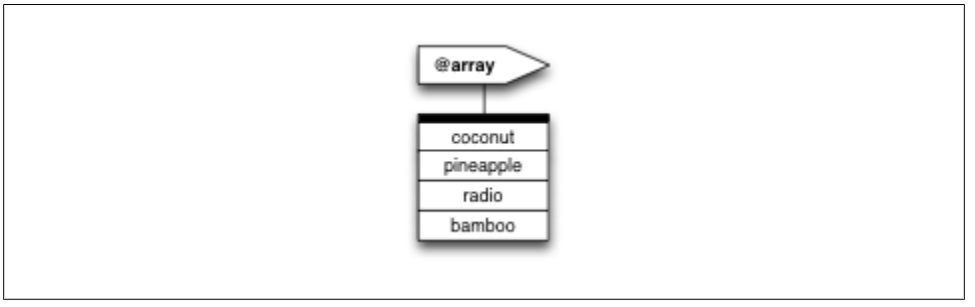

Figure 5-3. The PeGS diagram for an array

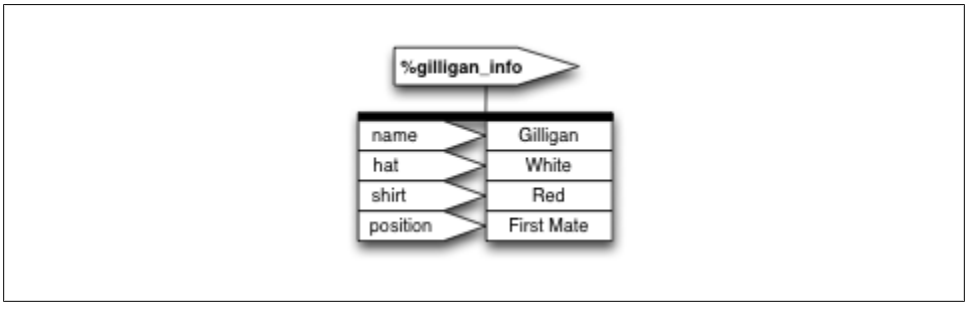

Figure 5-4. The PeGS diagram for a hash

The hash is even more fancy. Like the array, the data portion starts with a black bar, but under that it has two parts. On the left are the keys, in pointy-sided boxes pointing at the box on their right which shows the corresponding value.

We'll draw some of our complex data structures using these diagrams, and as we go further along we'll introduce some other features of PeGs.

Taking a Reference to an Array

Among its many other meanings, the backslash (\) character is also the "take a reference to" operator. When we use it in front of an array name, e.g., `\@skipper`, the result is a *reference* to that array. A reference to the array is like a pointer[3]: it points at the array, but is not the array itself.

A reference fits wherever a scalar fits. It can go into an element of an array or a hash, or into a plain scalar variable, like this:

```
my $ref_to_skipper = \@skipper;
```

3. We're not talking about pointers in the C sense, but in the dog sense. Think of an English pointer showing where the duck is, not a memory address

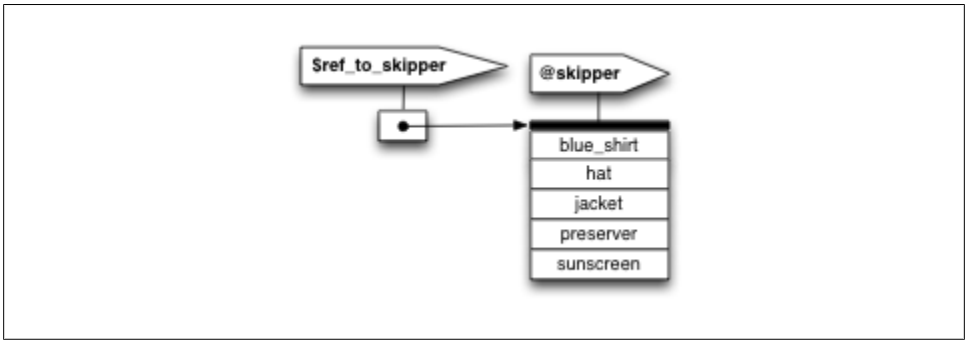

Figure 5-5. The PeGS diagram for a reference

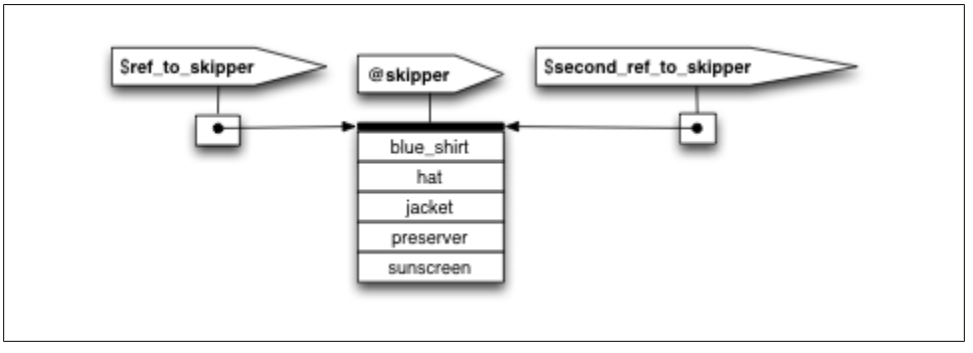

Figure 5-6. @skipper with one other reference pointing to the same data

In the PeGS notation, the reference is just a scalar so it's diagram looks like a scalar. However, in the data portion, it points to the data it references. Notice that the arrow from the reference to the data points specfically at the data, not the name of the variable that holds that data. That's going to be important very soon.

We can copy the reference to another reference, and both references point to the same data:

```
my $second_ref_to_skipper = $reference_to_skipper;
```

We can even do it again:

```
my $third_ref_to_skipper = \@skipper;
```

We can interchange all three references. We can even say they're identical, because, in fact, they are the same thing. When we compare reference with ==, we get back true if they point to the same data.

```
if ($reference_to_skipper == $second_reference_to_skipper) {
  print "They are identical references.\n";
}
```

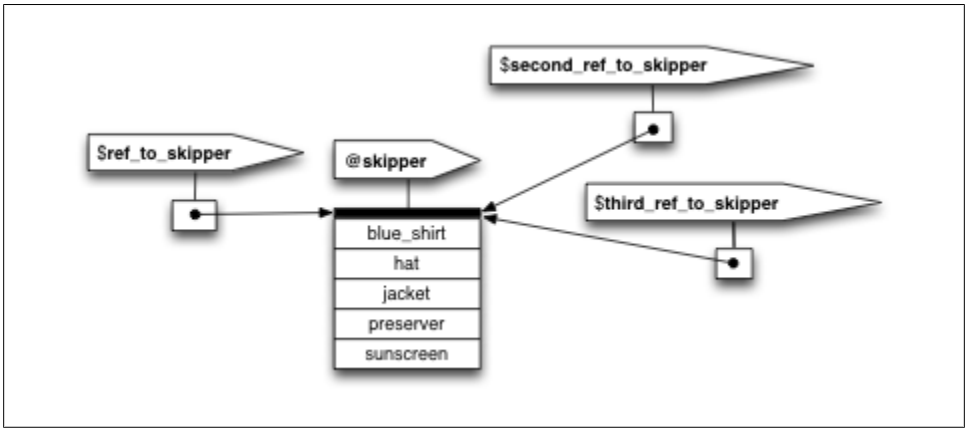

Figure 5-7. @skipper with two other references pointing to the same data

This equality compares the numeric forms of the two references. The numeric form of the reference is the unique memory address of the `@skipper` internal data structure, unchanging during the life of the data. If we look at the string form instead, with `eq` or `print`, we get a debugging string:

```
ARRAY(0x1a2b3c)
```

which again is unique for this array because it includes the hexadecimal (base 16) representation of the array's unique memory address. The debugging string also notes that this is an array reference. Of course, if we ever see something like this in our output, it almost certainly means we have a bug; users of our program have little interest in hex dumps of storage addresses!

Also note that the references need to point to the same data in Perl's memory, not two different data that just happen to have the same values.

Because we can copy a reference, and passing an argument to a subroutine is really just copying, we can use this code to pass a reference to the array into the subroutine:

```
my @skipper = qw(blue_shirt hat jacket preserver sunscreen);
check_required_items("The Skipper", \@skipper);

sub check_required_items {
  my $who = shift;
  my $items = shift;
  my @required = qw(preserver sunscreen water_bottle jacket);
  ...
}
```

Now `$items` in the subroutine is a reference to the array of `@skipper`. But how do we get from a reference back into the original array? We *dereference* the reference, of course.

Dereferencing the Array Reference

If we look at `@skipper`, we'll see that it consists of two parts: the `@` symbol and the name of the array. Similarly, the syntax `$skipper[1]` consists of the name of the array in the middle and some syntax around the outside to get at the second element of the array (index value 1 is the second element because index values start at 0).

Here's the trick: we can place any reference to an array in curly braces in place of the name of an array, ending up with a method to access the original array. That is, wherever we write `skipper` to name the array, we use the reference inside curly braces: `{ $items }`. For example, both of these lines refer to the entire array:

```
@  skipper
@{ $items }
```

whereas both of these refer to the second item of the array:[4]

```
$  skipper [1]
${ $items }[1]
```

By using the reference form, we've decoupled the code and the method of array access from the actual array. Let's see how that changes the rest of this subroutine:

```
sub check_required_items {
  my $who   = shift;
  my $items = shift;

  my %whos_items = map { $_, 1 } @{$items};

  my @required = qw(preserver sunscreen water_bottle jacket);
  for my $item (@required) {
    unless ( exists $whos_items{$item} ) { # not found in list?
      print "$who is missing $item.\n";
    }
  }
}
```

All we did was replace `@_` (the copy of the provisions list) with `@{$items}`, a dereferencing of the reference to the original provisions array. Now we can call the subroutine a few times as before:

```
my @skipper = qw(blue_shirt hat jacket preserver sunscreen);
check_required_items('The Skipper', \@skipper);

my @professor = qw(sunscreen water_bottle slide_rule batteries radio);
check_required_items('Professor', \@professor);

my @gilligan = qw(red_shirt hat lucky_socks water_bottle);
check_required_items('Gilligan', \@gilligan);
```

4. Note that we added whitespace in these two displays to make the similar parts line up. This whitespace is legal in a program, even though most programs won't use it.

In each case, `$items` points to a different array, so the same code applies to different arrays each time we invoke it. This is one of the most important uses of references: decoupling the code from the data structure on which it operates so we can reuse the code more readily.

Passing the array by reference fixes the first of the two problems we mentioned earlier. Now, instead of copying the entire provision list into the `@_` array, we get a single element which is a reference to that provisions array.

Could we have eliminated the two `shift`s at the beginning of the subroutine? Sure, but we sacrifice clarity:

```
sub check_required_items {
  my %whos_items = map {$_, 1} @{$_[1]};

  my @required = qw(preserver sunscreen water_bottle jacket);
  for my $item (@required) {
    unless (exists $whos_items{$item}) { # not found in list?
      print "$_[0] is missing $item.\n";
    }
  }
}
```

We still have two elements in `@_`. The first element is the passenger or crew member name, which we use in the error message. The second element is a reference to the correct provisions array, which we use in the `grep` expression.

Getting Our Braces Off

Most of the time, the array reference we want to dereference is a simple scalar variable, such as `@{$items}` or `${$items}[1]`. In those cases, we can drop the curly braces, unambiguously forming `@$items` or `$$items[1]`.

However, we cannot drop the braces if the value within the braces is not a simple scalar variable. For example, for `@{$_[1]}` from that last subroutine rewrite, we can't remove the braces. That's a single element access to an array, not a scalar variable.

This rule also means that it's easy to see where the "missing" braces need to go. When we see `$$items[1]`, a pretty noisy piece of syntax, we can tell that the curly braces must belong around the simple scalar variable, `$items`. Therefore, `$items` must be a reference to an array.

Thus, an easier-on-the-eyes version of that subroutine might be:

```
sub check_required_items {
  my $who   = shift;
  my $items = shift;

  my @required = qw(preserver sunscreen water_bottle jacket);
  for my $item (@required) {
    unless (grep $item eq $_, @$items) { # not found in list?
```

```
      print "$who is missing $item.\n";
    }
  }
}
```

The only difference here is that we removed the braces around `@$items`.

Modifying the Array

We solved the excessive copying problem with an array reference. Now let's modify the original array.

For every missing provision, we push that provision onto an array, forcing the passenger to consider the item:

```
sub check_required_items {
  my $who   = shift;
  my $items = shift;

my $whos_items = map { $_, 1 } @$items;

  my @required = qw(preserver sunscreen water_bottle jacket);
  my @missing = (  );

  for my $item (@required) {
    unless ( exists $whos_items{$item} ) { # not found in list?
      print "$who is missing $item.\n";
      push @missing, $item;
    }
  }

  if (@missing) {
    print "Adding @missing to @$items for $who.\n";
    push @$items, @missing;
  }
}
```

Note the addition of the `@missing` array. If we find any items missing during the scan, we push them into `@missing`. If there's anything there at the end of the scan, we add it to the original provision list.

The key is in the last line of that subroutine. We're dereferencing the `$items` array reference, accessing the original array, and adding the elements from `@missing`. Without passing by reference, we'd modify only a local copy of the data, which has no effect on the original array.

Also, `@$items` (and its more generic form `@{$items}`) works within a double-quoted string and interpolates like a normal, named array. We can't include any whitespace between the `@` and the immediately following character, although we can include nearly arbitrary whitespace within the curly braces as if it were normal Perl code.

Figure 5-8. The array @all_with_names holds a multilevel data structure containing strings and references to arrays

Nested Data Structures

In this next example, the array `@_` contains two elements, one of which is also an array. What if we take a reference to an array that also contains a reference to an array? We end up with a complex data structure, which can be quite useful.

For example, we can iterate over the data for the Skipper, Gilligan, and the Professor by first building a larger data structure holding the entire list of provision lists:

```
my @skipper = qw(blue_shirt hat jacket preserver sunscreen);
my @skipper_with_name = ('Skipper', \@skipper);
my @professor = qw(sunscreen water_bottle slide_rule batteries radio);
my @professor_with_name = ('Professor', \@professor);
my @gilligan = qw(red_shirt hat lucky_socks water_bottle);
my @gilligan_with_name = ('Gilligan', \@gilligan);
```

At this point, `@skipper_with_name` has two elements, the second of which is an array reference similar to what we passed to the subroutine. Now we group them all:

```
my @all_with_names = (
  \@skipper_with_name,
  \@professor_with_name,
  \@gilligan_with_name,
);
```

Note that we have just three elements in `@all_with_names`, each of which is a reference to an array which has two elements: the name and its corresponding initial provisions. A picture of that is in Figure 4-XXX.

Therefore, `$all_with_names[2]` is the array reference for the Gilligan's data. If we dereference it as `@{$all_with_names[2]}`, we get a two-element array, "`Gilligan`" and another array reference.

How do we access that array reference? Using our rules again, it's `${$all_with_names[2]}[1]`. In other words, taking `$all_with_names[2]`, we dereference it in an expression that would be something like `$DUMMY[1]` as an ordinary array, so we'll place `{$all_with_names[2]}` in place of `DUMMY`.

How do we call the existing `check_required_items()` with this data structure? The following code is easy enough:

```
for my $person (@all_with_names) {
  my $who = $$person[0];
  my $provisions_reference = $$person[1];
  check_required_items($who, $provisions_reference);
}
```

This requires no changes to our subroutine. The control variable `$person` goes through each of `$all_with_names[0]`, `$all_with_names[1]`, and `$all_with_names[2]`, as the loop progresses. When we dereference `$$person[0]`, we get "Skipper," "Professor," and "Gilligan," respectively. `$$person[1]` is the corresponding array reference of provisions for that person.

Of course, we can shorten this as well since the entire dereferenced array matches the argument list precisely:

```
for my $person (@all_with_names) {
  check_required_items(@$person);
}
```

or even:

```
check_required_items(@$_) for @all_with_names;
```

As you can see, various levels of optimization can lead to obfuscation. Be sure to consider where your head will be a month from now when you have to reread your own code. If that's not enough, consider the new person who takes over your job after you have left.[5]

Simplifying Nested Element References with Arrows

Look at the curly-brace dereferencing again. As in our earlier example, the array reference for Gilligan's provision list is `${$all_with_names[2]}[1]`. Now, what if we want to know Gilligan's first provision? We need to dereference *this* item one more level, so it's Yet Another Layer of Braces: `${${$all_with_names[2]}[1]}[0]`. That's a really noisy piece of syntax. Can we shorten that? Yes!

Everywhere we write `${DUMMY}[$y]`, we can write `DUMMY->[$y]` instead. In other words, we can dereference an array reference, picking out a particular element of that array by simply following the expression defining the array reference with an arrow and a square-bracketed subscript.

For this example, this means we can pick out the array reference for Gilligan with a simple `$all_with_names[2]->[1]`, and Gilligan's first provision with `$all_with_names[2]->[1]->[0]`. Wow, that's definitely easier on the eyes.

If *that* wasn't already simple enough, there's one more rule: if the arrow ends up between "subscripty kinds of things," such as square brackets, we can also drop the arrow because multiple subscripts imply a dereference already. `$all_with_names[2]->[1]->[0]` becomes `$all_with_names[2][1][0]`. Now it's looking even easier on the eyes.

The arrow has to be *between* non-subscripty things. Why wouldn't it be between subscripty things? Well, imagine a reference to the array `@all_with_names`:

5. O'Reilly Media has a great book to help you be nice to the next guy. *Perl Best Practices* by Damian Conway has 256 tips on writing more readable and maintainable Perl code.

```
my $root = \@all_with_names;
```

Now how do we get to Gilligan's first item?

```
$root -> [2] -> [1] -> [0]
```

More simply, using the "drop arrow" rule, we can use:

```
$root -> [2][1][0]
```

We cannot drop the first arrow, however, because that would mean an array `@root`'s third element, an entirely unrelated data structure. Let's compare this to the full curly-brace form again:

```
${${${$root}[2]}[1]}[0]
```

It looks much better with the arrow. Note, however, that no shortcut gets the entire array from an array reference. If we want all of Gilligan's provisions, we say:

```
@{$root->[2][1]}
```

Reading this from the inside out, we can think of it like this:

1. Take `$root`.
2. Dereference it as an array reference, taking the third element of that array (index number 2): `$root-`[2]>
3. Dereference that as an array reference, taking the second element of that array (index number 1): `$root-`[2][1]>
4. Dereference that as an array reference, taking the entire array. @{$root->[2][1]}

The last step doesn't have a shortcut arrow form. Oh well.[6]

References to Hashes

Just as we can take a reference to an array, we can also take a reference to a hash. Once again, we use the backslash as the "take a reference to" operator and store the result in a scalar:

```
my %gilligan_info = (
  name     => 'Gilligan',
  hat      => 'White',
  shirt    => 'Red',
  position => 'First Mate',
);
my $hash_ref = \%gilligan_info;
```

We can dereference a hash reference to get back to the original data. The strategy is the same as dereferencing an array reference. We write the hash syntax as we would have without references, and then replace the name of the hash with a pair of curly braces

6. It's not that it hasn't been discussed repeatedly by the Perl developers; it's just that nobody has come up with a nice backward-compatible syntax with universal appeal.

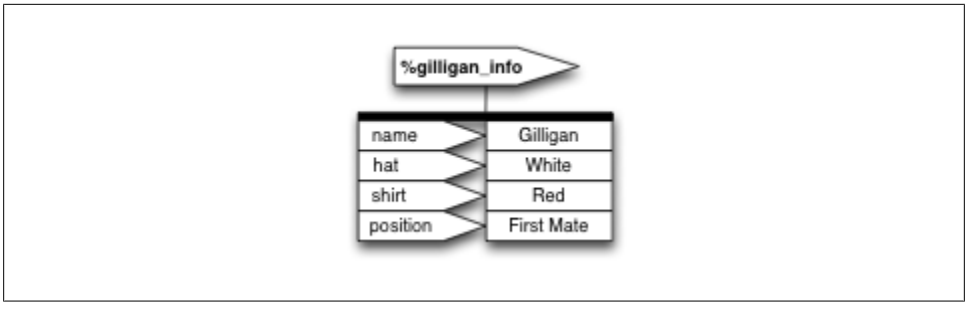

Figure 5-9. The PeGS structure for the %gilligan_info hash

surrounding the thing holding the reference. For example, to pick a particular value for a given key, we do this:

```
my $name = $ gilligan_info { 'name' };
my $name = $ { $hash_ref } { 'name' };
```

In this case, the curly braces have two different meanings. The first pair denotes the expression returning a reference while the second pair delimits the expression for the hash key.

To perform an operation on the entire hash, we proceed similarly:

```
my @keys = keys % gilligan_info;
my @keys = keys % { $hash_ref };
```

As with array references, we can use shortcuts to replace the complex curly-braced forms under some circumstances. For example, if the only thing inside the curly braces is a simple scalar variable (as shown in these examples so far), we can drop the curly braces:

```
my $name = $$hash_ref{'name'};
my @keys = keys %$hash_ref;
```

Like an array reference, when referring to a specific hash element, we can use an arrow form:

```
my $name = $hash_ref->{'name'};
```

Because a hash reference fits wherever a scalar fits, we can create an array of hash references:

```
my %gilligan_info = (
  name     => 'Gilligan',
  hat      => 'White',
  shirt    => 'Red',
  position => 'First Mate',
);
my %skipper_info = (
  name     => 'Skipper',
  hat      => 'Black',
  shirt    => 'Blue',
  position => 'Captain',
```

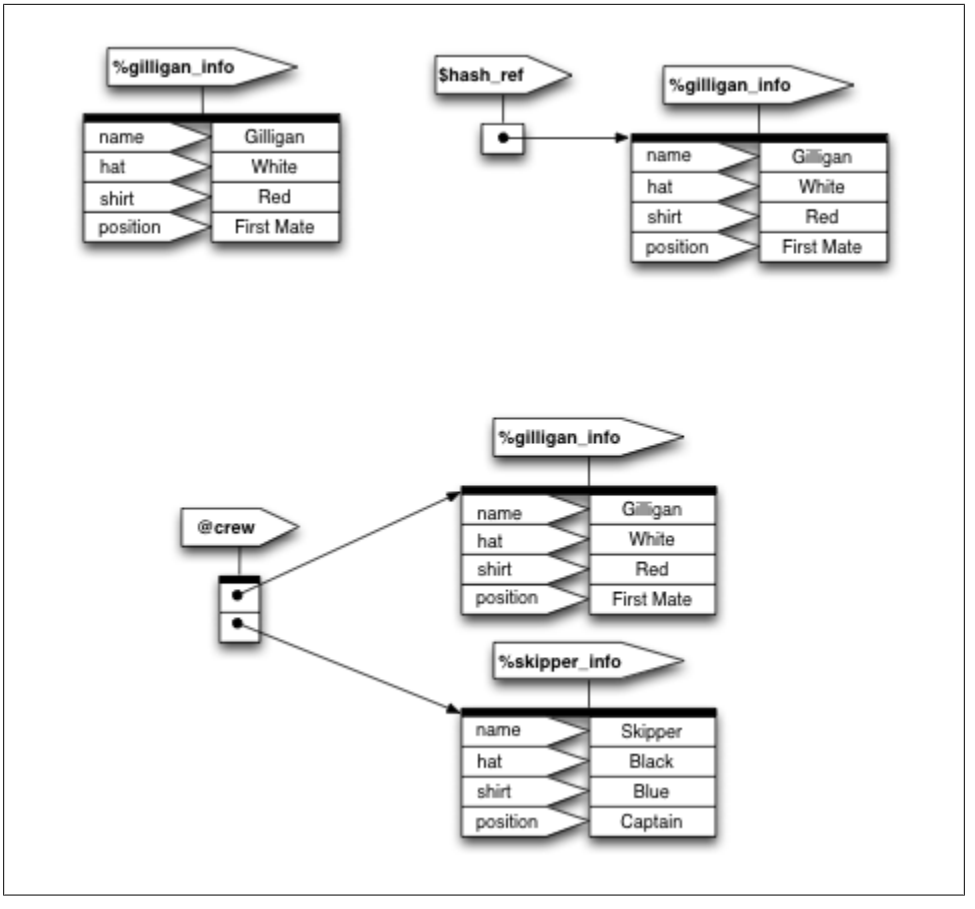

Figure 5-10. Crew Roster PeGs

```
);
my @crew = (\%gilligan_info, \%skipper_info);
```

Thus, `$crew[0]` is a hash reference to the information about Gilligan. We can get to Gilligan's name via any one of:

```
${ $crew[0] } { 'name' }
my $ref = $crew[0]; $$ref{'name'}
$crew[0]->{'name'}
$crew[0]{'name'}
```

On that last one, we can still drop the arrow between "subscripty kinds of things," even though one is an array bracket and one is a hash brace.

Let's print a crew roster:

```
my %gilligan_info = (
  name     => 'Gilligan',
  hat      => 'White',
```

```
  shirt    => 'Red',
  position => 'First Mate',
);
my %skipper_info = (
  name     => 'Skipper',
  hat      => 'Black',
  shirt    => 'Blue',
  position => 'Captain',
);
my @crew = (\%gilligan_info, \%skipper_info);

my $format = "%-15s %-7s %-7s %-15s\n";
printf $format, qw(Name Shirt Hat Position);
for my $crewmember (@crew) {
  printf $format,
    $crewmember->{'name'},
    $crewmember->{'shirt'},
    $crewmember->{'hat'},
    $crewmember->{'position'};
}
```

That last part looks very repetitive. We can shorten it with a hash slice[7]. Again, if the original syntax is:

```
@ gilligan_info { qw(name position) }
```

the hash slice notation from a reference looks like:

```
@ { $hash_ref } { qw(name position) }
```

We can drop the first brace pair because the only thing within is a simple scalar value, yielding:

```
@ $hash_ref { qw(name position) }
```

Thus, we can replace that final loop with:

```
for my $crewmember (@crew) {
  printf $format, @$crewmember{qw(name shirt hat position)};
}
```

There is no shortcut form with an arrow (`->`) for array slices or hash slices, just as there is no shortcut for entire arrays or hashes.

A hash reference prints as a string that looks like `HASH(0x1a2b3c)`, showing the hexadecimal memory address of the hash. That's not very useful to an end user and only barely more usable to the programmer, except as an indication of the lack of appropriate dereferencing.

7. For a review of hash slices, see *Learning Perl*, chapter 17. These are also documentated in *perldata*

Checking Reference Types

Once we start using references and passing them around, we have to ensure that we know which sort of reference we have. If we try to use the reference as a type that it is not, our program will blow up:

```
show_hash( \@array );

sub show_hash {
  my $hash_ref = shift;

  foreach my $key ( %$hash_ref ) {
    ...
  }
}
```

The `show_hash` subroutine expects a hash and trusts that we pass it one. However, since we passed it an array reference, our program blows up:

```
Not a HASH reference at line ...
```

If we went to be careful, we should check the argument to `show_hash` to ensure that its actually a hash reference. There are a couple ways that we could do this. The easy way uses `ref()`, which returns the reference type. We compare the return value from `ref()` to what we expected:

```
use Carp qw(croak);

sub show_hash {
  my $hash_ref = shift;
  my $ref_type = ref $hash_ref;
  croak "I expected a hash reference!"
    unless $ref_type eq 'HASH';

  foreach my $key ( %$hash_ref ) {
    ...
  }
}
```

That's looks odd to us, though, because we had to hardcode the literal string `HASH`. We never like to do that. We can, however, get rid of the literal string with a double use of `ref()`. We call `ref()` a second time with a prototypical version of the reference type that we expect:

```
croak "I expected a hash reference!"
  unless $ref_type eq ref {};
```

Alternatively, we can use the `constant` module to store the hash reference string:

```
use constant HASH => ref {}

croak "I expected a hash reference!"
  unless $ref_type eq HASH;
```

Using a constant looks a lot like a literal string, but it has an important difference: the constant will fail if we use the wrong name because it is not defined, but a wrong literal string will never fail because Perl has no way to know that we used the wrong string.

Our use of `ref()` has another problem, which we can't fully explain until we talk about objects. Since `ref()` returns a string that gives the reference type, if we have an object that can act like a hash reference[8]. In that case, `ref()` doesn't return the string `HASH` necessarily.

Instead of asking "What are you", it asks "What can you do?". We really only want to know if the argument to `show_hash` can act like a hash reference so it doesn't blow up. We don't specifically care that it is exactly a hash reference. For instance, the `Hash::AsObject` module lets us have a hash with either the traditional or an object-oriented interface. Although it can act like a hash reference, and its proper use expects us treat it like a hash reference. However, `ref` doesn't return `HASH` for those; it gives use the object type, `Hash::AsObject`. We'll see more of that when we talk about objects.

In that case, we might use an `eval` in which we try to do something hash-like. If the `eval` fails and returns false, we didn't have a hash:

```
croak "I expected a hash reference!"
  unless eval { keys %$ref_type; 1 }
```

If we expect to check this often, we should probably wrap the check in its own subroutine:

```
sub is_hash_ref {
  my $hash_ref = shift;

  return eval { keys %$ref_type; 1 };
}

croak "I expected a hash reference!"
  unless is_hash_ref( $ref_type );
```

Exercises

You can find the answers to these exercises in "This is section" on page 183.

Exercise 1 [5 min]

How many different things do these expressions refer to? Draw a PeGS structure for each.

```
$ginger->[2][1]
${$ginger[2]}[1]
$ginger->[2]->[1]
${$ginger->[2]}[1]
```

8. Objects that are not based on hash references can still *act* like hashes

Exercise 2 [30 min]

Using the final version of `check_required_items`, write a subroutine `check_items_for_all` that takes as its only parameter a hash reference that points to a hash whose keys are the people aboard the *Minnow*, and whose corresponding values are array references of the things they intend to bring on board.

For example, the hash reference might be constructed like so:

```
my @gilligan  = (... gilligan items  ...);
my @skipper   = (... skipper items   ...);
my @professor = (... professor items ...);

my %all = (
  Gilligan  => \@gilligan,
  Skipper   => \@skipper,
  Professor => \@professor,
);

check_items_for_all(\%all);
```

The newly constructed subroutine should call `check_required_items` for each person in the hash, updating their provisions list to include the required items.

Exercise 2 [20 minutes]

XXX: Add another exercise

CHAPTER 6
References and Scoping

We can copy and pass around references like any other scalar. At any given time, Perl knows the number of references to a particular data item. Perl can also create references to anonymous data structures that do not have explicit names and create references automatically as needed to fulfill certain kinds of operations. Let's look at copying references and how it affects scoping and memory usage.

More than One Reference to Data

Chapter 5 explored how to take a reference to an array `@skipper` and place it into a new scalar variable:

```
my @skipper = qw(blue_shirt hat jacket preserver sunscreen);
my $ref_to_skipper = \@skipper;
```

We can then copy the reference or take additional references, and they'll all refer to the same thing and are interchangeable:

```
my $second_ref_to_skipper = $reference_to_skipper;
my $third_ref_to_skipper  = \@skipper;
```

At this point, we have four different ways to access the data contained in `@skipper`:

```
@skipper
@$ref_to_skipper
@$second_ref_to_skipper
@$third_ref_to_skipper
```

Perl tracks how many ways it can access the data through a mechanism called *reference counting*. The original name counts as one, and each additional reference that we create (including copies of references) also counts as one. The total number of references to the array of provisions is now four.

We can add and remove references as we wish, and as long as the reference count doesn't hit zero, Perl maintains the array in memory and it is still accessible via any of the other access paths. For example, we might have a temporary reference:

```
check_provisions_list(\@skipper)
```

When this subroutine executes, Perl creates a fifth reference to the data and copies it into `@_` for the subroutine. The subroutine is free to create additional copies of that reference, which Perl notes as needed. Typically, when the subroutine returns, Perl discards all such references automatically, and we're back to four references again.

We can kill off each reference by using the variable for something other than a reference to the value of `@skipper`. For example, we can assign `undef` to the variable:

```
$ref_to_skipper = undef;
```

Or, maybe we just let the variable go out of scope:

```
my @skipper = ...;

{ # naked block
...
my $ref = \@skipper;
...
...
} # $ref goes out of scope at this point
```

In particular, a reference held in a subroutine's private (lexical) variable goes away at the end of the subroutine.

Whether we change the value or the variable itself goes away, Perl notes it as an appropriate reduction in the number of references to the data.

Perl recycles the memory for the array only when all references (including the name of the array) go away. In this case, Perl only reclaims memory when `@skipper` and all the references we created to it disappear.

Such memory is available to Perl for other data later in this program invocation, and generally Perl doesn't give it back to the operating system.

What if That Was the Name?

Typically, all references to a variable are gone before the variable itself. But what if one of the references outlives the variable name? For example, consider this code:

```
my $ref;

{
my @skipper = qw(blue_shirt hat jacket preserver sunscreen); # ref count is 1
$ref        = \@skipper;                                     # ref count is 2

print "$ref->[2]\n"; # prints jacket\n
}

print "$ref->[2]\n"; # still prints jacket\n                # ref count is 1
```

Immediately after we declare the `@skipper` array, we have one reference to the five-element list. After `$ref` is initialized, we'll have two, down to the end of the block. When the block ends, the `@skipper` name disappears. However, this was only one of the two ways to access the data! Thus, the five-element list is still in memory, and `$ref` still points to that data.

At this point, the five-element list is in an *anonymous array*, which is a fancy term for an array without a name.

Until the value of `$ref` changes, or `$ref` itself disappears, we can still use all the dereferencing strategies we used prior to when the name of the array disappeared. In fact, it's still a fully functional array that we can shrink or grow just as we do any other Perl array:

```
push @$ref, 'sextant'; # add a new provision
print "$ref->[-1]\n"; # prints sextant\n
```

We can even increase the reference count at this point:

```
my $copy_of_ref = $ref;
```

or equivalently:

```
my $copy_of_ref = \@$ref;
```

The data stays alive until we destroy the last reference:

```
$ref = undef; # not yet...
$copy_of_ref = undef; # poof!
```

Reference Counting and Nested Data Structures

The data remains alive until we destroy the last reference, even if that reference lives within a larger active data structure. Suppose an array element is itself a reference. Recall the example from Chapter 5:

```
my @skipper = qw(blue_shirt hat jacket preserver sunscreen);
my @skipper_with_name   = ('The Skipper', \@skipper);

my @professor = qw(sunscreen water_bottle slide_rule batteries radio);
my @professor_with_name = ('The Professor', \@professor);

my @gilligan = qw(red_shirt hat lucky_socks water_bottle);
my @gilligan_with_name  = ('Gilligan', \@gilligan);

my @all_with_names = (
  \@skipper_with_name,
  \@professor_with_name,
  \@gilligan_with_name,
  );
```

Imagine for a moment that the intermediate variables are all part of a subroutine:

```
my @all_with_names;

sub initialize_provisions_list {
  my @skipper = qw(blue_shirt hat jacket preserver sunscreen);
  my @skipper_with_name = ('The Skipper', \@skipper);

  my @professor = qw(sunscreen water_bottle slide_rule batteries radio);
  my @professor_with_name = ('The Professor', \@professor);

  my @gilligan = qw(red_shirt hat lucky_socks water_bottle);
  my @gilligan_with_name = ('Gilligan', \@gilligan);

  @all_with_names = ( # set global
    \@skipper_with_name,
    \@professor_with_name,
    \@gilligan_with_name,
  );
}

initialize_provisions_list(  );
```

We set the value of `@all_with_names` to contain three references. Inside the subroutine we have named arrays with references to arrays first placed into other named arrays. Eventually, the values end up in the global `@all_with_names`. However, as the subroutine returns, the names for the six arrays disappear. Each array has had one other reference taken to it, making the reference count temporarily two, and then back to one as the name disappears. Because the reference count is not yet zero the data continues to live on, although it is now referenced only by elements of `@all_with_names`.

Rather than assign the global variable, we can rewrite this without `@all_with_names` and return the list directly:

```
sub get_provisions_list {
  my @skipper = qw(blue_shirt hat jacket preserver sunscreen);
  my @skipper_with_name = ('The Skipper', \@skipper);

  my @professor = qw(sunscreen water_bottle slide_rule batteries radio);
  my @professor_with_name = ('The Professor', \@professor);

  my @gilligan = qw(red_shirt hat lucky_socks water_bottle);
  my @gilligan_with_name = ('Gilligan', \@gilligan);

  return (
    \@skipper_with_name,
    \@professor_with_name,
    \@gilligan_with_name,
  );
}

my @all_with_names = get_provisions_list(  );
```

Here, we create the value that we'll eventually store in `@all_with_names` as the last expression evaluated in the subroutine. The subroutine returns a three-element list. As long as the named arrays within the subroutine have had at least one reference taken

of them, and it is still part of the return value, the data remains alive. If we alter or discard the references in `@all_with_names`, Perl reduces the reference count for the corresponding arrays. If that means the reference count has become zero (as in this example), Perl also eliminates the arrays themselves. Because the arrays inside `@all_with_names` also contain a reference (such as the reference to `@skipper`), Perl reduces that reference count by one. Again, that reduces the reference count to zero, freeing that memory as well, in a cascading effect.

Removing the top of a tree of data generally removes all the data contained within. The exception is when we make additional copies of the references of the nested data. For example, if we copy Gilligan's provisions:

```
my $gilligan_stuff = $all_with_names[2][1];
```

then when we remove `@all_with_names`, we still have one live reference to what was formerly `@gilligan`, and the data from there downward remains alive.

The bottom line is simply: Perl does the right thing. If we still have a reference to data, we still have the data.

When Reference Counting Goes Bad

Reference-counting as a way to manage memory has been around for a long time. A really long time. The downside of reference counting is that it breaks when the data structure is not a `directed graph`—that is, when some parts of the structure point back in to other parts in a looping way. For example, suppose each of two data structures contains a reference to the other (see Figure 5-1):

```
my @data1 = qw(one won);
my @data2 = qw(two too to);

push @data2, \@data1;
push @data1, \@data2;
```

At this point, we have two names for the data in `@data1`: `@data1` itself and `@{$data2[3]}`, and two names for the data in `@data2`: `@data2` itself and `@{$data1[2]}`. We've created a loop. In fact, we can access `won` with an infinite number of names, such as `$data1[2][3][2][3][2][3][1]`.

What happens when these two array names go out of scope? Well, the reference count for the two arrays goes down from two to one, but not zero! And because it's not zero, Perl thinks there might still be a way to get to the data, even though there isn't! Thus, we've created a `memory leak`. A memory leak in a program causes the program to consume more and more memory over time. Ugh.

At this point, you're right to think that example is contrived. Of course we would never make a looped data structure in a real program! Actually, programmers often make these loops as part of doubly-linked lists, linked rings, or a number of other data structures, or even by accident. The key is that Perl programmers rarely do so because the

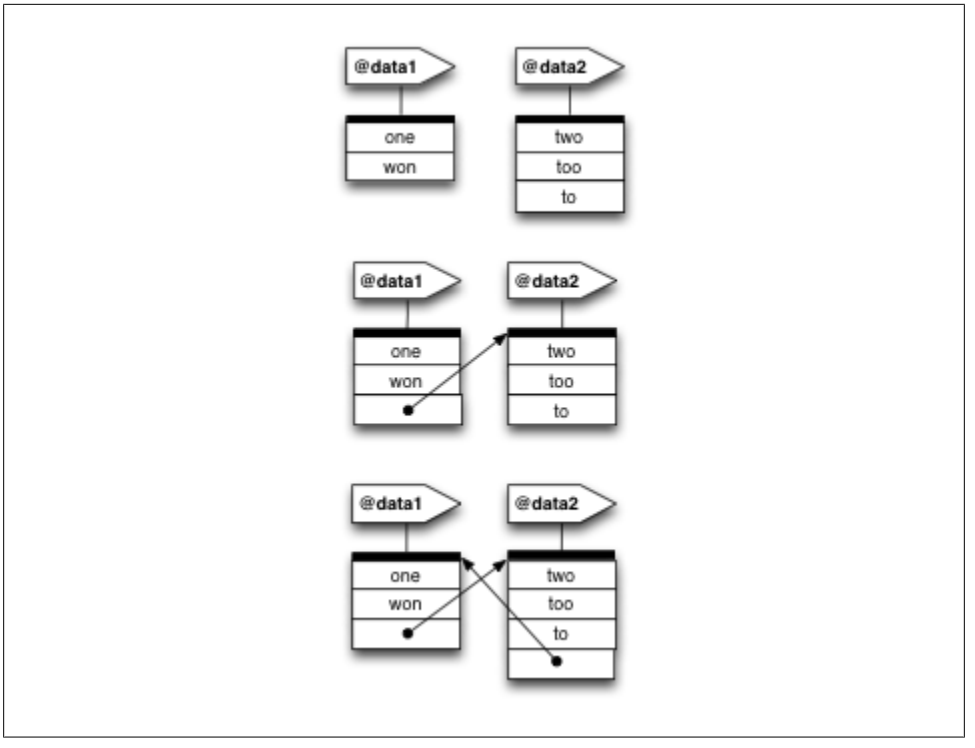

Figure 6-1. When the references in a data structure form a loop, Perl's reference-counting system may not be able to recognize and recycle the no-longer-needed memory space

most important reasons to use those data structures don't apply in Perl. Most of that deals with managing memory and connecting discontiguous memory blocks, which Perl does for us.

If you've used other languages, you may have noticed programming tasks that are comparatively easy in Perl. For example, it's easy to sort a list of items or to add or remove items, even in the middle of the list. Those tasks are difficult in some other languages, and using a looped data structure is a common way to get around the language's limitations. Why mention it here? Well, even Perl programmers sometimes copy an algorithm from another programming language. There's nothing inherently wrong with doing this, although it would be better to decide why the original author used a "loopy" data structure and recode the algorithm to use Perl's strengths. Perhaps you should use a hash instead, or perhaps the data should go into an array that will be sorted later.

Some upcoming version of Perl is likely to use *garbage collection* in addition to or instead of referencing counting.[1] Until then, we must be careful not to create circular references, or if we do, break the circle before the variables go out of scope. For example, the following code doesn't leak:

```
{
  my @data1 = qw(one won);
  my @data2 = qw(two too to);
  push @data2, \@data1;
  push @data1, \@data2;
  ... use @data1, @data2 ...
  # at the end:
  @data1 = (  );
  @data2 = (  );
}
```

We eliminated the reference to `@data2` from within `@data1`, and vice versa. Now the data have only one reference each, which all go to zero references at the end of the block. In fact, we can clear out either one and not the other and it still works nicely. Chapter 14 shows how to create weak references, which can help with many of these problems.

Creating an Anonymous Array Directly

In the `get_provisions_list` routine earlier, we created a half dozen array names that we used only so that we could take a reference to them immediately afterward. When the subroutine exited the array names all went away, but the references remained.

While creating temporarily named arrays would work in the simplest cases, creating such names becomes more complicated as the data structures become more detailed. We'd have to keep thinking of names of arrays just so we can forget them shortly thereafter.

We can reduce the namespace clutter by narrowing down the scope of the various array names. Rather than declaring them within the scope of the subroutine, we can create a temporary block:

```
my @skipper_with_name;
{
  my @skipper = qw(blue_shirt hat jacket preserver sunscreen);
  @skipper_with_name = ('The Skipper', \@skipper);
}
```

At this point, the second element of `@skipper_with_name` is a reference to the array formerly known as `@skipper`. However, that name is no longer relevant.

This is a lot of typing to simply say "the second element should be a reference to an array containing these elements." We can create such a value directly using the *anonymous array constructor*, which is Yet Another Use for square brackets:

```
my $ref_to_skipper_provisions =
  [ qw(blue_shirt hat jacket preserver sunscreen) ];
```

1. Just don't ask us about it. We wrote this book a long time before you got a chance to read it, so we didn't exactly know the details back then.

The square brackets take the value within (evaluated in a list context); establish a new, anonymous array initialized to those values; and (here's the important part) return a reference to that array. It's as if we had said:

```
my $ref_to_skipper_provisions;
{
  my @temporary_name =
  ( qw(blue_shirt hat jacket preserver sunscreen) );
  $ref_to_skipper_provisions = \@temporary_name;
}
```

We don't need to come up with a temporary name, and we don't need the extra noise of the temporary block. The result of a square-bracketed anonymous array constructor is an array reference, which fits wherever a scalar variable fits.

Now we can use it to construct the larger list:

```
my $ref_to_skipper_provisions =
  [ qw(blue_shirt hat jacket preserver sunscreen) ];
my @skipper_with_name = ('The Skipper', $ref_to_skipper_provisions);
```

Of course, we didn't actually need that scalar temporary, either. We can put a scalar reference to an array as part of a larger list:

```
my @skipper_with_name = (
  'The Skipper',
  [ qw(blue_shirt hat jacket preserver sunscreen) ]
);
```

Now let's walk through this. We've declared `@skipper_with_name`, the first element of which is the Skipper's name string, and the second element is an array reference, obtained by placing the five provisions into an array and taking a reference to it. So `@skipper_with_name` is only two elements long, just as before.

Don't confuse the square brackets with the parentheses here. They each have their distinct purpose. If we replace the square brackets with parentheses, we end up with a six-element list. If we replace the outer parentheses (on the first and last lines) with square brackets, we construct an anonymous array that's two elements long and then take the reference to that array as the only element of the ultimate `@skipper_with_name` array.[2] So, in summary, if we have this syntax:

```
my $fruits;
{
  my @secret_variable = ('pineapple', 'papaya', 'mango');
  $fruits = \@secret_variable;
}
```

we can replace it with:

```
my $fruits = ['pineapple', 'papaya', 'mango'];
```

2. In classrooms, we've seen that too much indirection (or not enough indirection) tends to contribute to the most common mistakes made when working with references.

Does this work for more complicated structures? Yes! Any time we need an element of a list to be a reference to an array, we can create that reference with an anonymous array constructor. In fact, we can also nest them in our provisions list:

```
sub get_provisions_list {
  return (
    ['The Skipper',   [qw(blue_shirt hat jacket preserver sunscreen)          ] ],
    ['The Professor', [qw(sunscreen water_bottle slide_rule batteries radio)  ] ],
    ['Gilligan',      [qw(red_shirt hat lucky_socks water_bottle)             ] ],
  );
}

my @all_with_names = get_provisions_list(  );
```

Walking through this from the outside in, we have a return value of three elements. Each element is an array reference, pointing to an anonymous two-element array. The first element of each array is a name string, while the second element is a reference to an anonymous array of varying lengths naming the provisions—all without having to come up with temporary names for any of the intermediate layers.

To the caller of this subroutine, the return value is identical to the previous version. However, from a maintenance point of view, the reduced clutter of not having all the intermediate names saves screen and brain space.

We can show a reference to an empty anonymous array using an empty anonymous array constructor. For example, if we add one "Mrs. Howell" to that travel list, as someone who has packed rather light, we'd simply insert:

```
['Mrs. Howell',
  [ ... ]
],
```

This is a single element of the larger list. This item is a reference to an array with two elements, the first of which is the name string, and the second of which is itself a reference to an empty anonymous array. The array is empty because Mrs. Howell hasn't packed anything for this trip.

Creating an Anonymous Hash

Similar to creating an anonymous array, we can also create an anonymous hash. Consider the crew roster from Chapter 5:

```
my %gilligan_info = (
  name     => 'Gilligan',
  hat      => 'White',
  shirt    => 'Red',
  position => 'First Mate',
);

my %skipper_info = (
  name     => 'Skipper',
  hat      => 'Black',
```

```
  shirt    => 'Blue',
  position => 'Captain',
);

my @crew = (\%gilligan_info, \%skipper_info);
```

The variables `%gilligan_info` and `%skipper_info` are just temporaries we needed to create the hashes for the final data structure. We can construct the reference directly with the *anonymous hash constructor*, which is Yet Another Meaning for curly braces, as we'll see. We can replace this:

```
my $ref_to_gilligan_info;

{
  my %gilligan_info = (
    name     => 'Gilligan',
    hat      => 'White',
    shirt    => 'Red',
    position => 'First Mate',
  );
  $ref_to_gilligan_info = \%gilligan_info;
}
```

with the anonymous hash constructor:

```
my $ref_to_gilligan_info = {
  name     => 'Gilligan',
  hat      => 'White',
  shirt    => 'Red',
  position => 'First Mate',
};
```

The value between the open and closing curly braces is an eight-element list. The eight-element list becomes a four-element anonymous hash (four key-value pairs). Perl takes a reference to this hash and returns as a single scalar value, which we assign to the scalar variable. Thus, we can rewrite the roster creation as:

```
my $ref_to_gilligan_info = {
  name     => 'Gilligan',
  hat      => 'White',
  shirt    => 'Red',
  position => 'First Mate',
};

my $ref_to_skipper_info = {
  name     => 'Skipper',
  hat      => 'Black',
  shirt    => 'Blue',
  position => 'Captain',
};

my @crew = ($ref_to_gilligan_info, $ref_to_skipper_info);
```

As before, we can now avoid the temporary variables and insert the values directly into the top-level list:

```
my @crew = (
  {
    name     => 'Gilligan',
    hat      => 'White',
    shirt    => 'Red',
    position => 'First Mate',
  },

  {
    name     => 'Skipper',
    hat      => 'Black',
    shirt    => 'Blue',
    position => 'Captain',
  },
);
```

Note we use trailing commas on the lists when the element is not immediately next to the closing brace, bracket, or parenthesis. This is a nice style element to adopt because it allows for easy maintenance. We can add or re-arrange lines quickly, or comment out lines without destroying the integrity of our list.

Now `@crew` is identical to the value it had before, but we no longer need to invent names for the intermediate data structures. As before, the `@crew` variable contains two elements, each of which is a reference to a hash containing keyword-based information about a particular crew member.

The anonymous hash constructor always evaluates its contents in a list context and then constructs a hash from key-value pairs, just as if we had assigned that list to a named hash. Perl returns a reference to that hash as a single value that fits wherever a scalar fits.

Now, a word from our parser: because blocks and anonymous hash constructors both use curly braces in roughly the same places in the syntax tree, the compiler has to make ad hoc determinations about which of the two we mean. If the compiler ever decides incorrectly, we might need to provide a hint to get what we want. To show the compiler that we want an anonymous hash constructor, put a plus sign before the opening curly brace: `+{ ... }`. To be sure to get a block of code, just put a semicolon (representing an empty statement) at the beginning of the block: `{; ... }`.

Autovivification

Let's look again at the provisions list. Suppose we were reading the data from a file, in this format:

```
The Skipper
  blue_shirt
  hat
  jacket
  preserver
  sunscreen
Professor
```

```
  sunscreen
  water_bottle
  slide_rule
Gilligan
  red_shirt
  hat
  lucky_socks
  water_bottle
```

We indent provisions with some whitespace, following a nonindented line with the person's name. Let's construct a hash of provisions. The keys of the hash will be the person's name, and the value will be an array reference to an array containing a list of provisions.

Initially, we might gather the data using a simple loop:

```
my %provisions;
my $person;

while (<>) {
  if (/^(\S.*)/) { # a person's name (no leading whitespace)
    $person = $1;
    $provisions{$person} = [  ] unless exists $provisions{$person};
  } elsif (/^\s+(\S.*)/) { # a provision
    die 'No person yet!' unless defined $person;
    push @{ $provisions{$person} }, $1;
  } else {
    die "I don't understand: $_";
  }
}
```

First, we declare the variables for the resulting hash of provisions and the current person. For each line that we read, we determine if it's a person or a provision. If it's a person, we remember the name and create the hash element for that person. The `unless exists` test ensures that we won't delete someone's provision list if his list is split in two places in the data file.

For example, suppose that "The Skipper" and " sextant" (note the leading whitespace) are at the end of the data file in order to list an additional data item.

The key is the person's name, and the value is initially a reference to an empty anonymous array. If the line is a provision, push it to the end of the correct array, using the array reference.

This code works fine, but it actually says more than it needs to. Why? Because we can leave out the line that initializes the hash element's value to a reference to an empty array:

```
my %provisions;
my $person;

while (<>) {
  if (/^(\S.*)/) { # a person's name (no leading whitespace)
    $person = $1;
```

```
    ## $provisions{$person} = [  ] unless exists $provisions{$person};
  } elsif (/^\s+(\S.*)/) { # a provision
    die 'No person yet!' unless defined $person;
    push @{ $provisions{$person} }, $1;
  } else {
    die "I don't understand: $_";
  }
}
```

What happens when we try to store that blue shirt for the Skipper? While looking at the second line of input, we'll end up with this effect:

```
push @{ $provisions{'The Skipper'} }, "blue_shirt";
```

At this point, `$provisions{"The Skipper"}` doesn't exist, but we're trying to use it as an array reference. To resolve the situation, Perl automatically inserts a reference to a new empty anonymous array into the variable and continues the operation. In this case, the reference to the newly created empty array is dereferenced, and we push the blue shirt to the provisions list.

This process is called *autovivification*. Any nonexisting variable, or a variable containing `undef`, which we dereference while looking for a variable location (technically called *an lvalue context*), is automatically stuffed with the appropriate reference to an empty item, and Perl allows the operation to proceed.

This is actually the same behavior we've probably been using in Perl all along. Perl creates new variables as needed. Before that statement, `$provisions{"The Skipper"}` didn't exist, so Perl created it. Then `@{ $provisions{"The Skipper"} }` didn't exist, so Perl created it as well.

For example, this works:

```
my $not_yet;                # new undefined variable
@$not_yet = (1, 2, 3);
```

We dereference the value `$not_yet` as if it were an array reference. But since it's initially `undef`, Perl acts as if we had explicitly initialized `$not_yet` as an empty array reference:

```
my $not_yet;
$not_yet = [  ]; # inserted through autovivification
@$not_yet = (1, 2, 3);
```

In other words, an initially empty array becomes an array of three elements.

This autovivification also works for multiple levels of assignment:

```
my $top;
$top->[2]->[4] = 'lee-lou';
```

Initially, `$top` contains `undef`, but because we dereference it as if it were an array reference, Perl inserts a reference to an empty anonymous array into `$top`. Perl then accesses the third element (index value 2), which causes Perl to grow the array to be three elements long. That element is also `undef`, so Perl stuffs it with a reference to another empty

anonymous array. We then spin out along that newly created array, setting the fifth element to `lee-lou`.

Autovivification and Hashes

Autovivification also works for hash references. If we dereference a variable containing `undef` as if it were a hash reference, a reference to an empty anonymous hash is inserted, and the operation continues.

One place this comes in very handy is in a typical data reduction task. For example let's say the Professor gets an island-area network up and running (perhaps using Coco-Net or maybe Vines), and now wants to track the traffic from host to host. He begins logging the number of bytes transferred to a log file, giving the source host, the destination host, and the number of transferred bytes:

```
professor.hut gilligan.crew.hut 1250
professor.hut lovey.howell.hut 910
thurston.howell.hut lovey.howell.hut 1250
professor.hut lovey.howell.hut 450
professor.hut laser3.copyroom.hut 2924
ginger.girl.hut professor.hut 1218
ginger.girl.hut maryann.girl.hut 199
...
```

Now the Professor wants to produce a summary of the source host, the destination host, and the total number of transferred bytes for the day. Tabulating the data is as simple reading the input line-by-line, breaking it up, and adding the latest value to what we had previously:

```
my %total_bytes;
while (<>) {
  my ($source, $destination, $bytes) = split;
  $total_bytes{$source}{$destination} += $bytes;
}
```

Let's see how this works on the first line of data. We'll execute:

```
$total_bytes{'professor.hut'}{'gilligan.crew.hut'} += 1250;
```

Because `%total_bytes` is initially empty, Perl doesn't find the first key of `professor.hut`, but it establishes an `undef` value for the dereferencing as a hash reference. (Keep in mind that an implicit arrow is between the two sets of curly braces here.) Perl sticks in a reference to an empty anonymous hash in that element, which it then immediately extends to include the element with a key of `gilligan.crew.hut`. Its initial value is `undef`, which acts like a zero when we add 1250 to it, and the result of 1250 is inserted back into the hash.

Any later data line that contains this same source host and destination host will re-use that same value, adding more bytes to the running total. But each new destination host extends a hash to include a new initially `undef` byte count, and each new source host

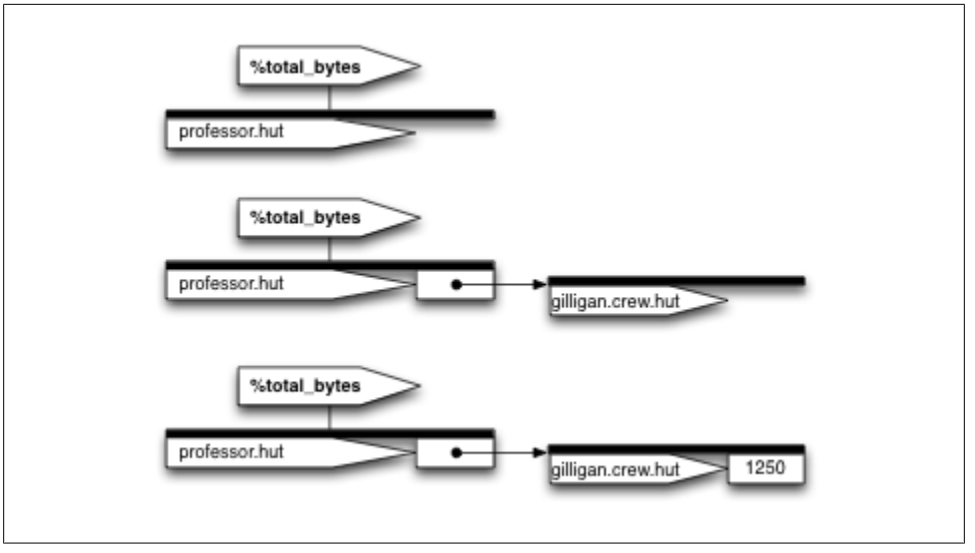

Figure 6-2. The autovivification of %total_bytes

uses autovivification to create a destination host hash. In other words, Perl does the right thing, as always.

Once we've processed the file, it's time to display the summary. First, we determine all the sources:

```
for my $source (keys %total_bytes) {
...
```

Now, we should get all destinations. The syntax for this is a bit tricky. We want all keys of the hash, resulting from dereferencing the value of the hash element, in the first structure:

```
for my $source (keys %total_bytes) {
  for my $destination (keys %{ $total_bytes{$source} }) {
....
```

For good measure, we should probably sort both lists to be consistent:

```
for my $source (sort keys %total_bytes) {
  for my $destination (sort keys %{ $total_bytes{$source} }) {
    print "$source => $destination:",
     " $total_bytes{$source}{$destination} bytes\n";
  }
  print "\n";
}
```

This is a typical data-reduction report generation strategy.[3] Simply create a hash-of-hashrefs (perhaps nested even deeper, as you'll see later), using autovivification to fill

3. You can read more examples in the Perl Data Structures Cookbook, the perldsc documentation page.

in the gaps in the upper data structures as needed, and then walk through the resulting data structure to display the results.

Exercises

You can find the answers to these exercises in "This is section" on page 183.

Exercise 1 [5 min]

Without running it, can you see what's wrong with this piece of a program? If you can't see the problem after a minute or two, see whether trying to run it will give you a hint of how to fix it (you might try turning on warnings).

```
my %passenger_1 = {
  name       => 'Ginger',
  age        => 22,
  occupation => 'Movie Star',
  real_age   => 35,
  hat        => undef,
};

my %passenger_2 = {
  name          => 'Mary Ann',
  age           => 19,
  hat           => 'bonnet',
  favorite_food => 'corn',
};

my @passengers = (\%passenger_1, \%passenger_2);
```

Exercise 2 [40 min]

The Professor's data file (mentioned earlier in this chapter) is available as `cocoa net.dat` in the files you can download from the O'Reilly web site. There may be comment lines (beginning with a pound sign); be sure to skip them. (That is, your program should skip them. You might find a helpful hint if *you* read them!)

Modify the code from this chapter so that each source machine's portion of the output shows the total number of bytes from that machine. List the source machines in order from most to least data transferred. Within each group, list the destination machines in order from most to least data transferred to that target from the source machine.

The result should be that the machine that sent the most data will be the first source machine in the list, and the first destination should be the machine to which it sent the most data. The Professor can use this printout to reconfigure the network for efficiency.

Exercise 3 [40 min]

Starting with your data structure from Exercise 2, rewrite the *coconet.dat* file so that it's in the same format, but sorted by source machine. You should report each destination machine once per source machine along with its total bytes transferred. XXX: show expected output

CHAPTER 7

Manipulating Complex Data Structures

Now that you've seen the basics of references, let's look at additional ways to manipulate complex data. We'll start by using the debugger to examine complex data structures and then use `Data::Dumper` to show the data under programmatic control. Next, you'll learn to store and retrieve complex data easily and quickly using `Storable`, and finally we'll wrap up with a review of `grep` and `map` and see how they apply to complex data.

Using the Debugger to View Complex Data

The Perl debugger can display complex data easily. For example, let's single-step through one version of the byte-counting program from Chapter 6:

```
my %total_bytes;
while (<>) {
  my ($source, $destination, $bytes) = split;
  $total_bytes{$source}{$destination} += $bytes;
}
for my $source (sort keys %total_bytes) {
  for my $destination (sort keys %{ $total_bytes{$source} }) {
    print "$source => $destination:",
     " $total_bytes{$source}{$destination} bytes\n";
  }
  print "\n";
}
```

Here's the data we'll use to test it:

```
professor.hut gilligan.crew.hut 1250
professor.hut lovey.howell.hut 910
thurston.howell.hut lovey.howell.hut 1250
professor.hut lovey.howell.hut 450
ginger.girl.hut professor.hut 1218
ginger.girl.hut maryann.girl.hut 199
```

We can do this a number of ways. One of the easiest is to invoke Perl with a `-d` switch on the command line:

```
myhost% perl -d bytecounts bytecounts-in

Loading DB routines from perl5db.pl version 1.19
Editor support available.

Enter h or `h h' for help, or `man perldebug' for more help.

main::(bytecounts:2):        my %total_bytes;
  D1 s
main::(bytecounts:3):        while (<>) {
  D1 s
main::(bytecounts:4):          my ($source, $destination, $bytes) = split;
  D1 s
main::(bytecounts:5):          $total_bytes{$source}{$destination} += $bytes;
  D1 x $source, $destination, $bytes
0  'professor.hut'
1  'gilligan.crew.hut'
2  1250
```

If you're playing along at home, be aware that each new release of the debugger works differently than any other, so your screen probably won't look exactly like ours. Also, if you get stuck at any time, type `h` for help, or look at `perldoc perldebug`.

The debugger shows each line of code before it executes it. That means that, at this point, we're about to invoke the autovivification, and we've got our keys established. The `s` command single-steps the program, while the `x` command dumps a list of values in a nice format. We can see that `$source`, `$destination`, and `$bytes` are correct, and now it's time to update the data:

```
D2 s
  main::(bytecounts:3):        while (<>) {
```

We've created the hash entries through autovivification. Let's see what we've got:

```
D2 x \%total_bytes
   0  HASH(0x132dc)
 'professor.hut' => HASH(0x37a34)
    'gilligan.crew.hut' => 1250
```

When we give `x` a hash reference, it dumps the entire contents of the hash, showing the key/value pairs. If any of the values are also hash references, it dumps those as well, recursively. What we'll see is that the `%total_bytes` hash has a single key of `professor.hut`, whose corresponding value is another hash reference. The referenced hash contains a single key of `gilligan.crew.hut`, with a value of `1250`, as expected.

Let's see what happens just after the next assignment:

```
D3 s
  main::(bytecounts:4):          my ($source, $destination, $bytes) = split;
D3 s
  main::(bytecounts:5):          $total_bytes{$source}{$destination} += $bytes;
D3 x $source, $destination, $bytes
  0  'professor.hut'
  1  'lovey.howell.hut'
  2  910
```

```
D4 s
  main::(bytecounts:3):        while (<>) {
D4 x \%total_bytes
  0  HASH(0x132dc)
 'professor.hut' => HASH(0x37a34)
    'gilligan.crew.hut' => 1250
    'lovey.howell.hut' => 910
```

Now we've added bytes flowing from `professor.hut` to `lovey.howell.hut`. The top-level hash hasn't changed, but the second-level hash has added a new entry. Let's continue:

```
D5 s
  main::(bytecounts:4):          my ($source, $destination, $bytes) = split;
D6 s
  main::(bytecounts:5):          $total_bytes{$source}{$destination} += $bytes;
D6 x $source, $destination, $bytes
  0  'thurston.howell.hut'
  1  'lovey.howell.hut'
  2  1250
D7 s
  main::(bytecounts:3):        while (<>) {
D7 x \%total_bytes
  0  HASH(0x132dc)
 'professor.hut' => HASH(0x37a34)
    'gilligan.crew.hut' => 1250
    'lovey.howell.hut' => 910
 'thurston.howell.hut' => HASH(0x2f9538)
    'lovey.howell.hut' => 1250
```

Ah, now it's getting interesting. A new entry in the top-level hash has a key of `thurston.howell.hut`, and a new hash reference, autovivified initially to an empty hash. Immediately after the new empty hash was put in place, a new key/value pair was added, indicating 1250 bytes transferred from `thurston.howell.hut` to `lovey.howell.hut`. Let's step some more:

```
D8 s
  main::(bytecounts:4):          my ($source, $destination, $bytes) = split;
D8 s
  main::(bytecounts:5):          $total_bytes{$source}{$destination} += $bytes;
D8 x $source, $destination, $bytes
  0  'professor.hut'
  1  'lovey.howell.hut'
  2  450
D9 s
  main::(bytecounts:3):        while (<>) {
D9 x \%total_bytes
  0  HASH(0x132dc)
 'professor.hut' => HASH(0x37a34)
    'gilligan.crew.hut' => 1250
    'lovey.howell.hut' => 1360
 'thurston.howell.hut' => HASH(0x2f9538)
    'lovey.howell.hut' => 1250
```

Now we're adding in some more bytes from `professor.hut` to `lovey.howell.hut`, reusing the existing value place. Nothing too exciting there. Let's keep stepping:

```
D10 s
  main::(bytecounts:4):          my ($source, $destination, $bytes) = split;
D10 s
  main::(bytecounts:5):          $total_bytes{$source}{$destination} += $bytes;
D10 x $source, $destination, $bytes
  0  'ginger.girl.hut'
  1  'professor.hut'
  2  1218
D11 s
  main::(bytecounts:3):        while (<>) {
D11 x \%total_bytes
  0  HASH(0x132dc)
 'ginger.girl.hut' => HASH(0x297474)
    'professor.hut' => 1218
 'professor.hut' => HASH(0x37a34)
    'gilligan.crew.hut' => 1250
    'lovey.howell.hut' => 1360
 'thurston.howell.hut' => HASH(0x2f9538)
    'lovey.howell.hut' => 1250
```

This time, we added a new source, `ginger.girl.hut`. Notice that the top level hash now has three elements, and each element has a different hash reference value. Let's step some more:

```
D12 s
  main::(bytecounts:4):          my ($source, $destination, $bytes) = split;
D12 s
  main::(bytecounts:5):          $total_bytes{$source}{$destination} += $bytes;
D12 x $source, $destination, $bytes
  0  'ginger.girl.hut'
  1  'maryann.girl.hut'
  2  199
D13 s
  main::(bytecounts:3):        while (<>) {
D13 x \%total_bytes
  0  HASH(0x132dc)
 'ginger.girl.hut' => HASH(0x297474)
    'maryann.girl.hut' => 199
    'professor.hut' => 1218
 'professor.hut' => HASH(0x37a34)
    'gilligan.crew.hut' => 1250
    'lovey.howell.hut' => 1360
 'thurston.howell.hut' => HASH(0x2f9538)
    'lovey.howell.hut' => 1250
```

Now we've added a second destination to the hash that records information for all bytes originating at `ginger.girl.hut`. Because that was the final line of data (in this run), a step brings us down to the lower `foreach` loop:

```
D14 s
  main::(bytecounts:8):        for my $source (sort keys %total_bytes) {
```

Even though we can't directly examine the list value from inside those parentheses, we can display it:

```
D14 x sort keys %total_bytes
    0  'ginger.girl.hut'
    1  'professor.hut'
    2  'thurston.howell.hut'
```

This is the list the `foreach` now scans. These are all the sources for transferred bytes seen in this particular logfile. Here's what happens when we step into the inner loop:

```
D15 s
    main::(bytecounts:9):          for my $destination (sort keys %{ $total_bytes{
  $source} }) {
```

At this point, we can determine from the inside out exactly what values will result from the list value from inside the parentheses. Let's look at them:

```
D15 x $source
  0  'ginger.girl.hut'
D16 x $total_bytes{$source}
  0  HASH(0x297474)
 'maryann.girl.hut' => 199
 'professor.hut' => 1218
D18 x keys %{ $total_bytes{$source } }
  0  'maryann.girl.hut'
  1  'professor.hut'
D19 x sort keys %{ $total_bytes{$source } }
  0  'maryann.girl.hut'
  1  'professor.hut'
```

Note that dumping `$total_bytes{$source}` shows that it was a hash reference. Also, the `sort` appears not to have done anything, but the output of `keys` is not necessarily in a sorted order. The next step finds the data:

```
D20 s
  main::(bytecounts:10):            print "$source => $destination:",
  main::(bytecounts:11):              " $total_bytes{$source}{$destination} bytes\n";
D20 x $source, $destination
  0  'ginger.girl.hut'
  1  'maryann.girl.hut'
D21 x $total_bytes{$source}{$destination}
  0  199
```

As we can see with the debugger, we can easily show the data, even structured data, to help us understand our program.

Viewing Complex Data with Data::Dumper

Another way to visualize a complex data structure rapidly is to *dump* it. A particularly nice dumping package is included in the Perl core distribution, called `Data::Dumper`. Let's replace the last half of the byte-counting program with a simple call to `Data::Dumper`:

```
use Data::Dumper;

my %total_bytes;
```

```
while (<>) {
  my ($source, $destination, $bytes) = split;
  $total_bytes{$source}{$destination} += $bytes;
}

print Dumper(\%total_bytes);
```

The `Data::Dumper` module defines the `Dumper` subroutine. This subroutine is similar to the `x` command in the debugger. We can give `Dumper` one or more values, and `Dumper` turns those values into a printable string. The difference between the debugger's `x` command and `Dumper`, however, is that the string generated by `Dumper` is Perl code:

```
$ perl bytecounts2 <bytecounts-in
$VAR1 = {
          'thurston.howell.hut' => {
                                     'lovey.howell.hut' => 1250
                                   },
          'ginger.girl.hut' => {
                                 'maryann.girl.hut' => 199,
                                 'professor.hut' => 1218
                               },
          'professor.hut' => {
                               'gilligan.crew.hut' => 1250,
                               'lovey.howell.hut' => 1360
                             }
        };
```

The Perl code is fairly understandable; it shows that we have a reference to a hash of three elements, with each value of the hash being a reference to a nested hash. We can evaluate this code and get a hash that's equivalent to the original hash. However, if we're thinking about doing this in order to have a complex data structure persist from one program invocation to the next, please keep reading.

`Data::Dumper`, like the debugger's `x` command, handles shared data properly. For example, go back to that "leaking" data from Chapter 6:

```
use Data::Dumper;
$Data::Dumper::Purity = 1; # declare possibly self-referencing structures
my @data1 = qw(one won);
my @data2 = qw(two too to);
push @data2, \@data1;
push @data1, \@data2;
print Dumper(\@data1, \@data2);
```

Here's the output from this program:

```
$VAR1 = [
          'one',
          'won',
          [
            'two',
            'too',
            'to',
            [  ]
          ]
```

```
        ];
$VAR1->[2][3] = $VAR1;
$VAR2 = $VAR1->[2];
```

Notice how we've created two different variables now, since there are two parameters to `Dumper`. The element `$VAR1` corresponds to a reference to `@data1`, while `$VAR2` corresponds to a reference to `@data2`. The debugger shows the values similarly:

```
D1 x \@data1, \@data2
    0  ARRAY(0xf914)
 0  'one'
 1  'won'
 2  ARRAY(0x3122a8)
    0  'two'
    1  'too'
    2  'to'
    3  ARRAY(0xf914)
       -> REUSED_ADDRESS
    1  ARRAY(0x3122a8)
 -> REUSED_ADDRESS
```

Note that the phrase `REUSED_ADDRESS` indicates that some parts of the data are actually references we've already seen.

Giving data structures names

XXX: Data::Dump Data::Printer

YAML

`Data::Dumper` is not the only game on the island though. Brian Ingerson came up with a Yet Another Markup Language (YAML) to provide a more readable (and more compact) dump. It works in the same way as `Data::Dumper`. We'll see more about YAML when we talk about modules later, so we won't say much about it here.

From the earlier example, we plug in YAML where we had `Data::Dumper`, and use `Dump()` where we had `Dumper()`.

```
use YAML;

my %total_bytes;

while (<>) {
  my ($source, $destination, $bytes) = split;
  $total_bytes{$source}{$destination} += $bytes;
  }

print Dump(\%total_bytes);
```

When you use the same data from the earlier example, you get this output:

```
--- #YAML:1.0
ginger.girl.hut:
```

```
    maryann.girl.hut: 199
    professor.hut: 1218
  professor.hut:
    gilligan.crew.hut: 1250
    lovey.howell.hut: 1360
  thurston.howell.hut:
    lovey.howell.hut: 1250
```

That's a lot easier to read because it takes up less space on the screen, which can be really handy when you have deeply nested data structures.

JSON

XXX:

Storing Complex Data with Storable

We can take the output of `Data::Dumper`'s `Dumper` routine, place it into a file, and then load the file to a different program. When we evaluate the code as Perl code we end up with two package variables, `$VAR1` and `$VAR2`, that are equivalent to the original data. This is called `marshaling` the data: converting complex data into a form that we can write to a file as a stream of bytes for later reconstruction.

However, another Perl core module is much better suited for marshaling: `Storable`. It's better suited because compared to `Data::Dumper`, `Storable` produces smaller and faster-to-process files. (The `Storable` module is standard in recent versions of Perl, but you can always install it from the CPAN if it's missing.)

The interface is similar to using `Data::Dumper`, except we must put everything into one reference. For example, let's store the mutually referencing data structures:

```
use Storable;
my @data1 = qw(one won);
my @data2 = qw(two too to);
push @data2, \@data1;
push @data1, \@data2;
store [\@data1, \@data2], 'some_file';
```

The file produced by this step is under 100 bytes, which is quite a bit shorter than the equivalent `Data::Dumper` output. It's also much less readable for humans. It's easy for `Storable` to read, as you'll soon see.[1] Next, fetch the data, again using the `Storable` module. The result will be a single array reference. We dump the result to see if it stored the right values:

```
use Storable;
my $result = retrieve 'some_file';
```

1. The format used by `Storable` is architecture byte-order dependent by default. Its documentation shows how to create byte-order-independent storage files.

```
use Data::Dumper;
$Data::Dumper::Purity = 1;
print Dumper($result);
```

Here's the result:

```
$VAR1 = [
          [
            'one',
            'won',
            [
              'two',
              'too',
              'to',
              [  ]
            ]
          ],
          [  ]
        ];
$VAR1->[0][2][3] = $VAR1->[0];
$VAR1->[1] = $VAR1->[0][2];
```

This is functionally the same as the original data structure. We're now looking at the two array references within one top-level array. To get something closer to what we saw before, we can be more explicit about the return value:

```
use Storable;
my ($arr1, $arr2) = @{ retrieve 'some_file' };
use Data::Dumper;
$Data::Dumper::Purity = 1;
print Dumper($arr1, $arr2);
```

or equivalently:

```
use Storable;
my $result = retrieve 'some_file';
use Data::Dumper;
$Data::Dumper::Purity = 1;
print Dumper(@$result);
```

and we'll get:

```
$VAR1 = [
          'one',
          'won',
          [
            'two',
            'too',
            'to',
            [  ]
          ]
        ];
$VAR1->[2][3] = $VAR1;
$VAR2 = $VAR1->[2];
```

just as we did in the original program. With `Storable`, we can store data and retrieve it later. More information on `Storable` can be found in `perldoc Storable`, as always.

Using the map and grep Operators

As the data structures become more complex, it helps to have higher-level constructs deal with common tasks such as selection and transformation. In this regard, Perl's `grep` and `map` operators are worth mastering.

Applying a Bit of Indirection

Some problems that may appear very complex are actually simple once we've seen a solution or two. For example, suppose we want to find the items in a list that have odd digit sums but don't want the items themselves. What we want to know is where they occurred in the original list.

All that's required is a bit of indirection.[2] First, we have a selection problem, so we use a `grep`. Let's not `grep` the values themselves but the index for each item:

```
my @input_numbers = (1, 2, 4, 8, 16, 32, 64);
my @indices_of_odd_digit_sums = grep {
  ...
} 0..$#input_numbers;
```

Here, the expression `0..$#input_numbers` will be a list of indices for the array. Inside the block, `$_` is a small integer, from 0 to 6 (seven items total). Now, we don't want to decide whether `$_` has an odd digit sum. We want to know whether the array element at that index has an odd digit sum. Instead of using `$_` to get the number of interest, use `$input_numbers[$_]`:

```
my @indices_of_odd_digit_sums = grep {
  my $number = $input_numbers[$_];
  my $sum;
  $sum += $_ for split //, $number;
  $sum % 2;
} 0..$#input_numbers;
```

The result will be the indices at which 1, 16, and 32 appear in the list: 0, 4, and 5. We could use these indices in an array slice to get the original values again:

```
my @odd_digit_sums = @input_numbers[ @indices_of_odd_digit_sums ];
```

The strategy here for an indirect `grep` or `map` is to think of the `$_` values as identifying a particular item of interest, such as the key in a hash or the index of an array, and then use that identification within the block or expression to access the actual values.

Here's another example: select the elements of `@x` that are larger than the corresponding value in `@y`. Again, we'll use the indices of `@x` as your `$_` items:

2. A famous computing maxim states that "there is no problem so complex that it cannot be solved with appropriate additional layers of indirection." Of course, with indirection comes obfuscation, so there's got to be a magic middle ground somewhere.

```
my @bigger_indices = grep {
  if ($_ > $#y or $x[$_] > $y[$_]) {
    1; # yes, select it
  } else {
    0; # no, don't select it
  }
} 0..$#x;
my @bigger = @x[@bigger_indices];
```

In the `grep`, `$_` varies from 0 to the highest index of `@x`. If that element is beyond the end of `@y`, we automatically select it. Otherwise, we look at the individual corresponding values of the two arrays, selecting only the ones that meet our match.

However, this is a bit more verbose than it needs to be. We could simply return the boolean expression rather than a explicit 1 or 0:

```
my @bigger_indices = grep {
  $_ > $#y or $x[$_] > $y[$_];
} 0..$#x;
my @bigger = @x[@bigger_indices];
```

More easily, we can skip the step of building the intermediate array by simply returning the items of interest with a `map`:

```
my @bigger = map {
  if ($_ > $#y or $x[$_] > $y[$_]) {
    $x[$_];
  } else {
    (  );
  }
} 0..$#x;
```

If the index is good, return the resulting array value. If the index is bad, return an empty list, making that item disappear.

Selecting and Altering Complex Data

We can use these operators on more complex data. Taking the provisions list from Chapter 6:

```
my %provisions = (
  'The Skipper'   => [qw(blue_shirt hat jacket preserver sunscreen)         ],
  'The Professor' => [qw(sunscreen water_bottle slide_rule batteries radio) ],
  'Gilligan'      => [qw(red_shirt hat lucky_socks water_bottle)            ],
);
```

In this case, `$provisions{"The Professor"}` gives an array reference of the provisions brought by the Professor, and `$provisions{"Gilligan"}[-1]` gives the last item Gilligan thought to bring.

We run a few queries against this data. Who brought fewer than five items?

```
my @packed_light = grep @{ $provisions{$_} } < 5, keys %provisions;
```

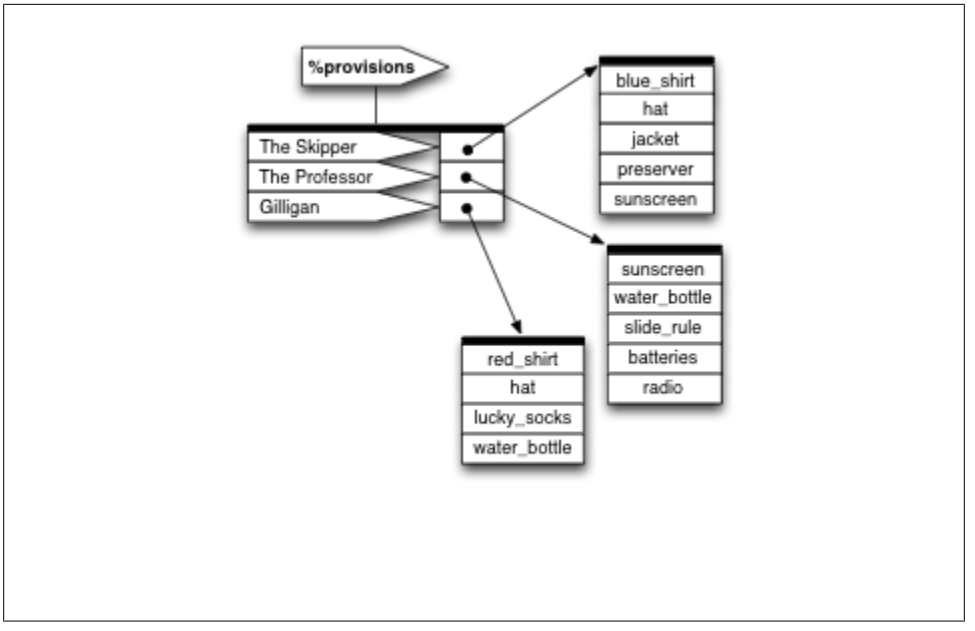

Figure 7-1. PeGS for provisions

In this case, `$_` is the name of a person. We take that name, look up the array reference of the provisions for that person, dereference that in a scalar context to get the count of provisions, and then compare it to 5. And wouldn't you know it; the only name is `Gilligan`.

Here's a trickier one. Who brought a water bottle?

```
my @all_wet = grep {
  my @items = @{ $provisions{$_} };
  grep $_ eq 'water_bottle', @items;
} keys %provisions;
```

Starting with the list of names again (`keys %provisions`), we pull up all the packed items first, and then use that list in an inner `grep` to count the number of those items that equal `water_bottle`. If the count is 0, there's no bottle, so the result is false for the outer `grep`. If the count is nonzero, we have a bottle, so the result is true for the outer `grep`. Now we see that the Skipper will be a bit thirsty later, without any relief.

We can also perform transformations. For example, turn this hash into a list of array references with each array containing two items. The first is the original person's name; the second is a reference to an array of the provisions for that person:

```
my @remapped_list = map {
  [ $_ => $provisions{$_} ];
} keys %provisions;
```

The keys of `%provisions` are names of the people. For each name, we construct a two-element list of the name and the corresponding provisions array reference. This list is

inside an anonymous array constructor, so we get back a reference to a newly created array for each person. Three names in; three references out.[3] Or, let's go a different way. Turn the input hash into a series of references to arrays. Each array will have a person's name and one of the items they brought:

```
my @person_item_pairs = map {
  my $person = $_;
  my @items = @{ $provisions{$person} };
  map [$person => $_], @items;
} keys %provisions;
```

Yes, a `map` within a `map`. The outer `map` selects one person at a time. We save this name in `$person`, and then we extract the item list from the hash. The inner `map` walks over this item list, executing the expression to construct an anonymous array reference for each item. The anonymous array contains the person's name and the provision item.

We had to use `$person` here to hold the outer `$_` temporarily. Otherwise, we can't refer to both temporary values for the outer `map` and the inner `map`.

Exercises

You can find the answers to these exercises in "This is section" on page 183.

Exercise 1 [20 min]

The program from Exercise 2 in Chapter 6 needs to read the entire data file each time it runs. However, the Professor has a new router log file each day and doesn't want to keep all that data in one giant file that takes longer and longer to process.

Fix up that program to keep the running totals in a data file so the Professor can simply run it on each day's logs to get the new totals.

Exercise 2 [5 min]

XXX:

3. If we had left the inner brackets off, we'd end up with six items out. That's not very useful, unless we're creating a different hash from them.

CHAPTER 8

Subroutine References

So far, we've shown references to three main Perl data types: scalars, arrays and hashes. We can also take a reference to a *subroutine* (sometimes called a *coderef*).

Why would we want to do that? Well, in the same way that taking a reference to an array lets us have the same code work on different arrays at different times, taking a reference to a subroutine allows the same code to call different subroutines at different times. Also, references permit complex data structures. A reference to a subroutine allows a subroutine to effectively become part of that complex data structure.

Put another way, a variable or a complex data structure is a repository of values throughout the program. A reference to a subroutine can be thought of as a repository of *behavior* in a program. The examples in this section show how this works.

Referencing a Named Subroutine

The Skipper and Gilligan are having a conversation:

```
sub skipper_greets {
  my $person = shift;
  print "Skipper: Hey there, $person!\n";
}

sub gilligan_greets {
  my $person = shift;
  if ($person eq "Skipper") {
    print "Gilligan: Sir, yes, sir, $person!\n";
  } else {
    print "Gilligan: Hi, $person!\n";
  }
}

skipper_greets("Gilligan");
gilligan_greets("Skipper");
```

This results in:

```
Skipper: Hey there, Gilligan!
Gilligan: Sir, yes, sir, Skipper!
```

So far, nothing unusual has happened. Note however that Gilligan has two different behaviors, depending on whether he's addressing the Skipper or someone else.

Now, have the Professor walk into the hut. Both of the Minnow crew greet the newest participant:

```
skipper_greets('Professor');
gilligan_greets('Professor');
```

which results in

```
Skipper: Hey there, Professor!
Gilligan: Hi, Professor!
```

Now the Professor feels obligated to respond:

```
sub professor_greets {
  my $person = shift;
  print "Professor: By my calculations, you must be $person!\n";
}

professor_greets('Gilligan');
professor_greets('Skipper');
```

resulting in:

```
Professor: By my calculations, you must be Gilligan!
Professor: By my calculations, you must be Skipper!
```

Whew. That's lot of typing and not very general. If each person's behavior is in a separate named subroutine and a new person walks in the door, we have to figure out what other subroutines to call. We could certainly do it with enough hard-to-maintain code, but we can simplify the process by adding a bit of indirection, just as we did with arrays and hashes.

First, let's use the "take a reference to" operator. It actually needs no introduction because it's that very same backslash as before:

```
my $ref_to_greeter = \&skipper_greets;
```

We're taking a reference to the subroutine `skipper_greets( )`. Note that the preceding ampersand is mandatory here, and the lack of trailing parentheses is also intentional. Perl stores the reference to the subroutine (a coderef) within `$ref_to_greeter`, and like all other references, it fits nearly anywhere a scalar fits.

There's only one reason to get back to the original subroutine by dereferencing the coderef: to invoke it. Dereferencing a code reference is similar to dereferencing other references. First start with the way we would have written it before we heard of references (including the optional ampersand prefix):

```
& skipper_greets ( 'Gilligan' )
```

Next, we replace the name of the subroutine with curly braces around the thing holding the reference:

```
& { $ref_to_greeter } ( 'Gilligan' )
```

There we have it. This construct invokes the subroutine currently referenced by `$ref_to_greeter`, passing it the single `Gilligan` parameter.

But boy-oh-boy, is that ugly or what? Luckily, the same reference simplification rules apply. If the value inside the curly braces is a simple scalar variable, we can drop the braces:

```
& $ref_to_greeter ( 'Gilligan' )
```

We can also flip it around a bit with the arrow notation:

```
$ref_to_greeter -> ( 'Gilligan' )
```

That last form is particularly handy when the coderef is in a larger data structure, as we'll show in a moment.

To have both Gilligan and the Skipper greet the Professor, we merely need to iterate over all the subroutines:

```
for my $greet (\&skipper_greets, \&gilligan_greets) {
  $greet->('Professor');
}
```

First, inside the parentheses, we create a list of two items, each of which is a coderef. The coderefs are then individually dereferenced, invoking the corresponding subroutine and passing it the `Professor` string.

We've seen the coderefs in a scalar variable and as an element of a list. Can we put these coderefs into a larger data structure? Certainly. Create a table that maps people to the behavior they exhibit to greet others, and then rewrite that previous example using the table:

```
sub skipper_greets {
  my $person = shift;
  print "Skipper: Hey there, $person!\n";
}

sub gilligan_greets {
  my $person = shift;
  if ($person eq 'Skipper') {
    print "Gilligan: Sir, yes, sir, $person!\n";
  } else {
    print "Gilligan: Hi, $person!\n";
  }
}

sub professor_greets {
  my $person = shift;
  print "Professor: By my calculations, you must be $person!\n";
}
```

```
my %greets = (
  Gilligan  => \&gilligan_greets,
  Skipper   => \&skipper_greets,
  Professor => \&professor_greets,
);

for my $person (qw(Skipper Gilligan)) {
  $greets{$person}->('Professor');
}
```

Note that `$person` is a name, which we look up in the hash to get to a coderef. Then we dereference that coderef, passing it the name of the person being greeted, and we get the correct behavior, resulting in:

```
Skipper: Hey there, Professor!
Gilligan: Hi, Professor!
```

Now have everyone greet everyone, in a very friendly room:

```
sub skipper_greets {
  my $person = shift;
  print "Skipper: Hey there, $person!\n";
}

sub gilligan_greets {
  my $person = shift;
  if ($person eq 'Skipper') {
    print "Gilligan: Sir, yes, sir, $person!\n";
  } else {
    print "Gilligan: Hi, $person!\n";
  }
}

sub professor_greets {
  my $person = shift;
  print "Professor: By my calculations, you must be $person!\n";
}

my %greets = (
  Gilligan  => \&gilligan_greets,
  Skipper   => \&skipper_greets,
  Professor => \&professor_greets,
);

my @everyone = sort keys %greets;
for my $greeter (@everyone) {
  for my $greeted (@everyone) {
    $greets{$greeter}->($greeted)
      unless $greeter eq $greeted; # no talking to yourself
  }
}
```

This results in:

```
Gilligan: Hi, Professor!
Gilligan: Sir, yes, sir, Skipper!
```

```
Professor: By my calculations, you must be Gilligan!
Professor: By my calculations, you must be Skipper!
Skipper: Hey there, Gilligan!
Skipper: Hey there, Professor!
```

Hmmm. That's a bit complex. Let's let them walk into the room one at a time:

```
sub skipper_greets {
  my $person = shift;
  print "Skipper: Hey there, $person!\n";
}

sub gilligan_greets {
  my $person = shift;
  if ($person eq 'Skipper') {
    print "Gilligan: Sir, yes, sir, $person!\n";
  } else {
    print "Gilligan: Hi, $person!\n";
  }
}

sub professor_greets {
  my $person = shift;
  print "Professor: By my calculations, you must be $person!\n";
}

my %greets = (
  Gilligan  => \&gilligan_greets,
  Skipper   => \&skipper_greets,
  Professor => \&professor_greets,
);

my @room; # initially empty
for my $person (qw(Gilligan Skipper Professor)) {
  print "\n";
  print "$person walks into the room.\n";
  for my $room_person (@room) {
    $greets{$person}->($room_person); # speaks
    $greets{$room_person}->($person); # gets reply
  }
  push @room, $person; # come in, get comfy
}
```

The result is a typical day on that tropical island:

```
Gilligan walks into the room.

Skipper walks into the room.
Skipper: Hey there, Gilligan!
Gilligan: Sir, yes, sir, Skipper!

Professor walks into the room.
Professor: By my calculations, you must be Gilligan!
Gilligan: Hi, Professor!
Professor: By my calculations, you must be Skipper!
Skipper: Hey there, Professor!
```

Anonymous Subroutines

In that last example, we never explicitly called subroutines such as `professor_greets()`; we only called them indirectly through the coderef. Thus, we wasted some brain cells to come up with a name for the subroutine used only in one other place, to initialize the data structure. But, just as we can create anonymous hashes and arrays, we can create anonymous subroutines!

Let's add another island inhabitant: Ginger. But rather than define her greeting behavior as a named subroutine, we create an anonymous subroutine:

```
my $ginger = sub {
  my $person = shift;
  print "Ginger: (in a sultry voice) Well hello, $person!\n";
};
$ginger->('Skipper');
```

An anonymous subroutine looks like an ordinary sub declaration, but there's no name (or prototype) between `sub` and the block which follows. It's also part of a statement, so we need a trailing semi-colon or other expression separator after it in most cases.

```
sub { ... body of subroutine ... };
```

The value in `$ginger` is a coderef, just as if we had defined the following block as a subroutine and then taken a reference to it. When we reach the last statement, we see:

```
Ginger: (in a sultry voice) Well hello, Skipper!
```

Although we kept the value in a scalar variable, we could have put that `sub { ... }` construct directly into the initialization of the greetings hash:

```
my %greets = (

  Skipper => sub {
    my $person = shift;
    print "Skipper: Hey there, $person!\n";
  },

  Gilligan => sub {
    my $person = shift;
    if ($person eq 'Skipper') {
      print "Gilligan: Sir, yes, sir, $person!\n";
    } else {
      print "Gilligan: Hi, $person!\n";
    }
  },

  Professor => sub {
    my $person = shift;
    print "Professor: By my calculations, you must be $person!\n";
  },

  Ginger => sub {
    my $person = shift;
```

```
    print "Ginger: (in a sultry voice) Well hello, $person!\n";
  },

);

my @room; # initially empty
for my $person (qw(Gilligan Skipper Professor Ginger)) {
  print "\n";
  print "$person walks into the room.\n";
  for my $room_person (@room) {
    $greets{$person}->($room_person); # speaks
    $greets{$room_person}->($person); # gets reply
  }
  push @room, $person; # come in, get comfy
}
```

Notice how much it simplifies the code. The subroutine definitions are right within the only data structure that references them directly. The result is straightforward:

```
Gilligan walks into the room.

Skipper walks into the room.
Skipper: Hey there, Gilligan!
Gilligan: Sir, yes, sir, Skipper!

Professor walks into the room.
Professor: By my calculations, you must be Gilligan!
Gilligan: Hi, Professor!
Professor: By my calculations, you must be Skipper!
Skipper: Hey there, Professor!

Ginger walks into the room.
Ginger: (in a sultry voice) Well hello, Gilligan!
Gilligan: Hi, Ginger!
Ginger: (in a sultry voice) Well hello, Skipper!
Skipper: Hey there, Ginger!
Ginger: (in a sultry voice) Well hello, Professor!
Professor: By my calculations, you must be Ginger!
```

Adding a few more castaways is as simple as putting the entry for the greeting behavior into the hash and adding them into the list of people entering the room. We get this scaling of effort because we've preserved the behavior as data over which we can iterate and look up, thanks to our friendly subroutine references.

Callbacks

A subroutine reference is often used for a *callback*. A callback defines what to do when a subroutine reaches a particular place in an algorithm. It gives us a chance to supply our own subroutine, the callback, for use at these points.

For example, the `File::Find` module exports a `find` subroutine that can efficiently walk through a given filesystem hierarchy in a fairly portable way. In its simplest form, we

give the `find` subroutine two parameters: a starting directory and "what to do" for each file or directory name found recursively below that starting directory. The "what to do" is specified as a subroutine reference:

```
use File::Find;
sub what_to_do {
  print "$File::Find::name found\n";
}
my @starting_directories = qw(.);

find(\&what_to_do, @starting_directories);
```

`find` starts at the current directory (.) and locates each file or directory. For each item, we call the subroutine `what_to_do( )`, passing it a few documented values through global variables. In particular, the value of `$File::Find::name` is the item's full pathname (beginning with the starting directory).

In this case, we're passing both data (the list of starting directories) and *behavior* as parameters to the `find` routine.

It's a bit silly to invent a subroutine name to use the name only once, so we can write the previous code using an anonymous subroutine, such as:

```
use File::Find;
my @starting_directories = qw(.);

find(
  sub {
    print "$File::Find::name found\n";
  },
  @starting_directories,
);
```

Closures

We could also use `File::Find` to find out some other things about files, such as their size. For the callback's convenience, the current working directory is the item's containing directory, and the item's name within that directory is found in `$_`.

In the previous code, we used `$File::Find::name` for the item's name. So which name is real, `$_` or `$File::Find::name`? `$File::Find::name` gives the name relative to the starting directory, but during the callback, the working directory is the one that holds the item just found. For example, suppose that we want `find` to look for files in the current working directory, so we give it `(".")` as the list of directories to search. If we call `find` when the current working directory is */usr*, `find` looks below that directory. When `find` locates */usr/bin/perl*, the current working directory (during the callback) is */usr/bin*. `$_` holds *perl* and `$File::Find::name` holds *./bin/perl*, which is the name relative to the directory in which we started the search.

All of this means that the file tests, such as `-s`, automatically report on the just-found item. Although this is convenient, the current directory inside the callback is different from the search's starting directory.

What if we want to use `File::Find` to accumulate the total size of all files seen? The callback subroutine cannot take arguments, and the caller discards its result. But that doesn't matter. When dereferenced, a subroutine reference can see all visible lexical variables when the reference to the subroutine is taken. For example:

```
use File::Find;

my $total_size = 0;
find(sub { $total_size += -s if -f }, '.');
print $total_size, "\n";
```

As before, we call the `find` routine with two parameters: a reference to an anonymous subroutine and the starting directory. When it finds names within that directory (and its subdirectories), it calls the anonymous subroutine.

Note that the subroutine accesses the `$total_size` variable. We declare this variable outside the scope of the subroutine but still visible to the subroutine. Thus, even though `find` invokes the callback subroutine (and would not have direct access to `$total_size`), the callback subroutine accesses and updates the variable.

The kind of subroutine that can access all lexical variables that existed at the time we declared it is called a *closure* (a term borrowed from the world of mathematics). In Perl terms, a closure is just a subroutine that references a lexical variable that has gone out of scope.

Furthermore, the access to the variable from within the closure ensures that the variable remains alive as long as the subroutine reference is alive. For example, let's number the output files:

```
use File::Find;

my $callback;
{
  my $count = 0;
  $callback = sub { print ++$count, ": $File::Find::name\n" };
}
find($callback, '.');
```

We declare a variable to hold the callback. We cannot declare this variable within the naked block (the block following that is not part of a larger Perl syntax construct), or perl will recycle it at the end of that block. Next, the lexical `$count` variable is initialized to 0. We then declare an anonymous subroutine and place its reference into `$callback`. This subroutine is a closure because it refers to the lexical `$count` variable that goes out of scope at the end of the block. Remember the semicolon after the anonymous subroutine; this is a statement, not a normal subroutine definition.

At the end of the naked block, the $count variable goes out of scope. However, because it is still referenced by subroutine in $callback, it stays alive as an anonymous scalar variable.[1] When the callback is invoked from find, the value of the variable formerly known as $count is incremented from 1 to 2 to 3, and so on.

Returning a Subroutine from a Subroutine

Although a naked block worked nicely to define the callback, having a subroutine return that subroutine reference instead might be more useful:

```
use File::Find;

sub create_find_callback_that_counts {
  my $count = 0;
  return sub { print ++$count, ": $File::Find::name\n" };
}

my $callback = create_find_callback_that_counts(  );
find($callback, '.');
```

It's the same process here, just written a bit differently. When we invoke create_find_callback_that_counts( ), we initialize the lexical variable $count to 0. The return value from that subroutine is a reference to an anonymous subroutine that is also a closure because it accesses the $count variable. Even though $count goes out of scope at the end of the create_find_callback_that_counts( ) subroutine, there's still a binding between it and the returned subroutine reference, so the variable stays alive until the subroutine reference is finally discarded.

If we reuse the callback, the same variable still has its most recently used value. The initialization occurred in the original subroutine (create_find_callback_that_counts), not the callback (unnamed) subroutine:

```
use File::Find;

sub create_find_callback_that_counts {
  my $count = 0;
  return sub { print ++$count, ": $File::Find::name\n" };
}

my $callback = create_find_callback_that_counts();
print "my bin:\n";
find($callback, 'bin');
print "my lib:\n";
find($callback, 'lib');
```

1. To be more accurate, the closure declaration increases the reference count of the referent, as if another reference had been taken explicitly. Just before the end of the naked block, the reference count of $count is two, but after the block has exited, the value still has a reference count of one. Although no other code may access $count, it will still be kept in memory as long as the reference to the sub is available in $callback or elsewhere.

This example prints consecutive numbers starting at 1 for the entries below `bin`, but then continues the numbering when we start entries in `lib`. The same `$count` variable is used in both cases. However, if we invoke the `create_find_callback_that_counts( )` twice, we get two different `$count` variables:

```
use File::Find;

sub create_find_callback_that_counts {
  my $count = 0;
  return sub { print ++$count, ": $File::Find::name\n" };
}

my $callback1 = create_find_callback_that_counts(  );
my $callback2 = create_find_callback_that_counts(  );
print "my bin:\n";
find($callback1, 'bin');
print "my lib:\n";
find($callback2, 'lib');
```

In this case, we have two separate `$count` variables, each accessed from within their own callback subroutine.

How would we get the total size of all found files from the callback? Earlier, we were able to do this by making `$total_size` visible. If we stick the definition of `$total_size` into the subroutine that returns the callback reference, we won't have access to the variable. But we can cheat a bit. For one thing, we can determine that we'll never call the callback subroutine with any parameters, so, if the subroutine receives a parameter, we make it return the total size:

```
use File::Find;

sub create_find_callback_that_sums_the_size {
  my $total_size = 0;
  return sub {
    if (@_) { # it's our dummy invocation
      return $total_size;
    } else { # it's a callback from File::Find:
      $total_size += -s if -f;
    }
  };
}

my $callback = create_find_callback_that_sums_the_size(  );
find($callback, 'bin');
my $total_size = $callback->('dummy'); # dummy parameter to get size
print "total size of bin is $total_size\n";
```

Distinguishing actions by the presence or absence of parameters is not a universal solution. Fortunately, we can create more than one subroutine reference in `create_find_callback_that_counts( )`:

```
use File::Find;

sub create_find_callbacks_that_sum_the_size {
```

```
  my $total_size = 0;
  return(sub { $total_size += -s if -f }, sub { return $total_size });
}

my ($count_em, $get_results) = create_find_callbacks_that_sum_the_size(  );
find($count_em, 'bin');
my $total_size = &$get_results(  );
print "total size of bin is $total_size\n";
```

Because we created both subroutine references from the same scope, they both have access to the same `$total_size` variable. Even though the variable has gone out of scope before we call either subroutine, they still share the same heritage and can use the variable to communicate the result of the calculation.

Returning the two subroutine references from the creating subroutine does not invoke them. The references are just data at that point. It's not until we invoke them as a callback or an explicit subroutine derefencing that they actually do their job.

What if we invoke this new subroutine more than once?

```
use File::Find;

sub create_find_callbacks_that_sum_the_size {
  my $total_size = 0;
  return(sub { $total_size += -s if -f }, sub { return $total_size });
}

## set up the subroutines
my %subs;
foreach my $dir (qw(bin lib man)) {
  my ($callback, $getter) = create_find_callbacks_that_sum_the_size(  );
  $subs{$dir}{CALLBACK}   = $callback;
  $subs{$dir}{GETTER}     = $getter;
}

## gather the data
for (keys %subs) {
  find($subs{$_}{CALLBACK}, $_);
}

## show the data
for (sort keys %subs) {
  my $sum = $subs{$_}{GETTER}->(  );
  print "$_ has $sum bytes\n";
}
```

In the section to set up the subroutines, we create three instances of callback-and-getter pairs. Each callback has a corresponding subroutine to get the results. Next, in the section to gather the data, we call `find` three times with each corresponding callback subroutine reference. This updates the individual `$total_size` variables associated with each callback. Finally, in the section to show the data, we call the getter routines to fetch the results.

The six subroutines (and the three `$total_size` variables they share) are reference-counted. When we modify `%subs` or it goes out of scope, the values have their reference counts reduced, recycling the contained data. (If that data also references further data, those reference counts are also reduced appropriately.)

Closure Variables as Inputs

While the previous examples showed closure variables being modified, closure variables are also useful to provide initial or lasting input to the subroutine. For example, let's write a subroutine to create a `File::Find` callback that prints files exceeding a certain size:

```
use File::Find;

sub print_bigger_than {
  my $minimum_size = shift;
  return sub { print "$File::Find::name\n" if -f and -s >= $minimum_size };
}

my $bigger_than_1024 = print_bigger_than(1024);
find($bigger_than_1024, 'bin');
```

We pass the `1024` parameter into the `print_bigger_than`, which then gets shifted into the `$minimum_size` lexical variable. Because we access this variable within the subroutine referenced by the return value of the `print_bigger_than` variable, it becomes a closure variable, with a value that persists for the duration of that subroutine reference. Again, invoking this subroutine multiple times creates distinct "locked-in" values for `$minimum_size`, each bound to its corresponding subroutine reference.

Closures are "closed" only on lexical variables, since lexical variables eventually go out of scope. Because a package variable (which is a global) never goes out of scope, a closure never closes on a package variable. All subroutines refer to the same single instance of the global variable.

To illustrate this in our live *Intermediate Perl* class, we created `File::Find::Closures`, a collection of generator subroutines that each return two closures. One closure we give to `find` and the other we use to get the list of matching files:

```
use File::Find;
use File::Find::Closures;

my( $wanted, $list_reporter ) = find_by_name( qw(README) );

find( $wanted, @directories );

my @readmes = $list_reporter->();
```

We don't intend anyone to really use this module so much as steal from it. Here's `find_by_min_size`, that creates closures to find the files with a size equal to or larger than the byte size you pass in:

```
use File::Spec::Functions qw(canonpath no_upwards);

sub find_by_min_size {
  my $min   = shift;

  my @files = ();

  sub { push @files, canonpath( $File::Find::name )
    if -s $_ >= $min },
  sub { @files = no_upwards( @files );
    wantarray ? @files : [ @files ] }
}
```

You can easily adapt that for your own needs and put it right into your program.

Closure Variables as Static Local Variables

A subroutine doesn't have to be an anonymous subroutine to be a closure. If a named subroutine accesses lexical variables and those variables go out of scope, the named subroutine retains a reference to the lexicals, just as we showed with anonymous subroutines. For example, consider two routines that count coconuts for Gilligan:

```
{
my $count;
sub count_one { ++$count }
sub count_so_far { return $count }
}
```

If we place this code at the beginning of the program, and we declare the variable `$count` inside the naked block scope. The two subroutines that reference the variable become closures. However, because they have a name, they will persist beyond the end of the scope just like any named subroutine. Since the subroutines persist beyond the scope and access variables declared within that scope, they become closures and thus can continue to access `$count` throughout the lifetime of the program.

So, with a few calls, we can see an incremented count:

```
count_one(  );
count_one(  );
count_one(  );
print 'we have seen ', count_so_far(  ), " coconuts!\n";
```

`$count` retains its value between calls to `count_one( )` or `count_so_far( )`, but no other section of code can access this `$count` at all.

In C, this is known as a *static local* variable: a variable that is visible to only a subset of the program's subroutines but persists throughout the life of the program, even between calls to those subroutines.

What if we wanted to count down? Something like this will do:

```
{
my $countdown = 10;
```

```
sub count_down { $countdown-- }
sub count_remaining { $countdown }
}

count_down(  );
count_down(  );
count_down(  );
print 'we're down to ', count_remaining(  ), " coconuts!\n";
```

That is, it'll do as long as we put it near the beginning of the program, before any invocations of `count_down( )` or `count_remaining( )`. Why?

This block doesn't work when we put it after those invocations because there are two functional parts to the first line:

```
my $countdown = 10;
```

One part is the declaration of `$countdown` as a lexical variable. That part is noticed and processed as the program is parsed during the *compile phase*. The second part is the assignment of 10 to the allocated storage. This is handled as perl executes the code during the *run phase*. Unless perl executes the run phase for this code, the variable has its initial `undef` value.

One practical solution to this problem is to change the block in which the static local appears into a `BEGIN` block:

```
BEGIN {
  my $countdown = 10;
  sub count_down { $countdown-- }
  sub count_remaining { $countdown }
}
```

The `BEGIN` keyword tells the Perl compiler that as soon as this block has been parsed successfully (during the compile phase), jump for a moment to run phase and run the block as well. Presuming the block doesn't cause a fatal error, compilation then continues with the text following the block. The block itself is also discarded, ensuring that the code within is executed precisely once in a program, even if it had appeared syntactically within a loop or subroutine.

state variables

Perl 5.10 introduced another way to make a private, persistent variable for a subroutine. We introduced these in *Learning Perl*, but we'll give a very brief review. Instead of creating a `BEGIN` block just to creating the scope for the lexical variable, we declare the variable inside the subroutine with `state`:

```
use 5.010;
sub countdown {
  state $countdown = 10;
  $countdown--;
}
```

We can use `state` in some places that some people don't normally consider a subroutine, like a `sort` block. If we wanted to watch the comparisons, we could also keep track of the comparison number without creating a variable outside of the `sort` block:

```
use 5.010;

my @array = qw( a b c d e f 1 2 3 );

print sort {
  state $n = 0;
  print $n++, ": a[$a] b[$b]\n";
  $a cmp $b;
  } @array;
```

We can also use `state` in a `map` block. If we want to sort lines but still remember their original position, we can use a `state` variable to keep track of the line number:

```
use 5.010;

my @sorted_lines_tuples =
  sort { $b->[1] <=> $a->[1] }
  map  { state $l = 0; [ $l++, $_ ] }
  <>;
```

The `state` variable has a limitation, though. So far, we can only initialize scalar variables with `state`. We can declare other types of variables, but we can't initialize them:

```
use 5.010;
sub add_to_tab {
  state @castaways = qw(Ginger Mary-Ann Gilligan); # compilation error
  state %tab = map { $_, 0 } @castaways;  # compilation error
  $countdown{'main'}--;
}
```

It gets messy to initialize them since we only want to do that once, not every time that we run the subroutine. Instead of that mess, we'll stick to scalars. But, wait! References are scalars, so we can initialize array or hash references:

```
use 5.010;
sub add_to_tab {
  my $castaway = shift;
  state $castaways = qw(Ginger Mary-Ann Gilligan); # works!
  state %tab = map { $_, 0 } @$castaways;  # works!
  $tab->{$castaway}++;
}
```

Finding Out Who We Are

Anonymous subroutines have a problem with identity; they don't know who they are! We don't really care if they don't have a name, but when it comes to them telling us who they are, a name would be quite handy. Suppose we want to write a recursive subroutine using anonymous subroutines. What name do we use to call the same subroutine again when we haven't even finished creating it?

```
my $countdown = sub {
  state $n = 5;
  return unless $n > -1;
  say $n--;
  WHAT_NAME???->();
};
```

We could do it in two steps so the variable holding the reference already exists:

```
my $countdown;
$countdown = sub {
  state $n = 5;
  return unless $n > -1;
  say $n--;
  $countdown->();
};
  $countdown->();
```

The output is our countdown:

```
5
4
3
2
1
0
```

That works because Perl doesn't care what is in `$sub` until it actually wants to use it. To get around this, Perl v5.16 introduces the `__SUB__` token to return a reference to the current subroutine:

```
my $sub = sub {
  state $n = 5;
  return unless $n > -1;
  say $n--;
  __SUB__->();
};
  $sub->();
```

This works with a named subroutine too:

```
sub countdown {
  state $n = 5;
  return unless $n > -1;
  say $n--;
  __SUB__->();
};
  countdown();
```

Enchanting Subroutines

How do we debug anonymous subroutines? When we start liberally passing them around, how do we know which one we have? For scalar, array, and hash values, we can dump their values[2]:

```
use v5.10;

my @array = ( \ 'xyz', [qw(a b c)], sub { say 'Buster' } );

foreach ( @array ) {
  when( ref eq ref \ ''   ) { say "Scalar $$_" }
  when( ref eq ref []     ) { say "Array @$_" }
  when( ref eq ref sub {} ) { say "Sub ???" }
}
```

The output lets us know what's on the inside, except for the subroutine, which we can't dereference without actually running it:

```
Scalar xyz
Array a b c
Sub ???
```

It's not that hard to make something useful here, but we have to pull out a lot of black magic we don't care to fully explain. This is some advanced kung fu, so don't worry if you don't get it all right now. This isn't something you'd want to do in everyday programming anyway.

First, we'll make the anonymous subroutine an object even though we haven't told you how to do that yet (after you read Chapter 16 you can come back to this). Inside this object, we overload stringification[3]. We get to tell Perl how to stringify this reference and we're going to add some useful information to it. That useful information is going to come from the `B` module, which can look into `perl`'s parse tree to do various things. We're going to use `B` to get the file and line number of the subroutine definition. This way we know where we have to go to fix things. Next, we use the `B::Deparse` module to turn the `perl` internal code back into human readable Perl with `coderef2text`. Finally, we put all of that together to form the string version of our subroutine reference:

```
use v5.14;

package MagicalCodeRef 1.00 {
  use overload '""' => sub {
    require B;

    my $ref = shift;
    my $gv = B::svref_2object($ref)->GV;

    require B::Deparse;
    my $deparse = B::Deparse->new;
    my $code = $deparse->coderef2text($ref);

    my $string = sprintf "---code ref---\n%s:%d\n%s\n---",
    $gv->FILE, $gv->LINE, $code;
    };
```

2. Some of this example first appeared in "Enchant closures for better debugging output", *http://www.effectiveperlprogramming.com/blog/1345*.

3. For details, see the `overload` module

```
  sub enchant { bless $_[1], $_[0] }
  }
```

When we create the anonymous subroutine, we pass it through our new mini-class and check for that when we want to stringify it:

```
my $sub = MagicalCodeRef->enchant( sub { say 'Gilligan!!!' } );

my @array = ( \ 'xyz', [qw(a b c)], $sub );

foreach ( @array ) {
  when( ref eq ref \ ''   )       { say "Scalar $$_" }
  when( ref eq ref []     )       { say "Array @$_" }
  when( ref eq 'MagicalCodeRef' ) { say "Sub $sub" }
}
```

Now, our output shows us where we defined the subroutine and what's in the subroutine (including an pragma settings in force at the time of the subroutine creation):

```
Scalar xyz
Array a b c
Sub ---code ref---
enchant.pl:26
{
  use warnings;
  use strict;
  no feature;
  use feature ':5.16';
  say 'Gilligan!!!';
}
---
```

Are you still with us? If so, let's throw in another wrinkle by making this a closure so the value for the name isn't in the subroutine:

```
my $sub = do {
  my $name = 'Gilligan';
  MagicalCodeRef->enchant( sub { say "$name!!!" } );
};
```

Now the output is less useful because we don't know what the value for `$name` is:

```
Scalar xyz
Array a b c
Sub ---code ref---
debug.pl:28
{
  use warnings;
  use strict;
  no feature;
  use feature ':5.16';
  say "$name!!!";
}
---
```

There's a module, `PadWalker`, that can look further back into `perl`'s parsing to find those closure variables. We used its `closed_over` function to get a hash of those variables then dump it with `Data::Dumper`:

```
use v5.14;

package MagicalCodeRef 1.01 {
  use overload '""' => sub {
    require B;

    my $ref = shift;
    my $gv = B::svref_2object($ref)->GV;

    require B::Deparse;
    my $deparse = B::Deparse->new;
    my $code = $deparse->coderef2text($ref);

    require PadWalker;
    my $hash = PadWalker::closed_over( $ref );

    require Data::Dumper;
      local $Data::Dumper::Terse = 1;
    my $string = sprintf "---code ref---\n%s:%d\n%s\n---\n%s---",
    $gv->FILE, $gv->LINE,
    $code,
    Data::Dumper::Dumper( $hash );
    };

  sub enchant { bless $_[1], $_[0] }
  }
```

Now we see the values for `$name`:

```
Scalar xyz
Array a b c
Sub ---code ref---
debug.pl:38
{
  use warnings;
  use strict 'refs';
  BEGIN {
    $^H{'feature_unicode'} = q(1);
    $^H{'feature_say'} = q(1);
    $^H{'feature_state'} = q(1);
    $^H{'feature_switch'} = q(1);
  }
  say "$name!!!";
}
---
{
  '$name' => \'Gilligan'
}
---
```

Still with us? Pat yourself on the back. That was tricky stuff. Now that you understand it, we hope you never have to use it. We wouldn't want your coworkers to strand you on a desert island.

Dumping Closures

Now that we've shown you the hard way to dump closures, we'll show you an easier way. You're probably used to that from us. We're like a good murder mystery; the first suspect is never the killer.

The `Data::Dump::Streamer` module is `Data::Dumper` on steroids, and it can handle code references and closures.

```
use Data::Dump::Streamer;

my @luxuries = qw(Diamonds Furs Caviar);

my $hash = {
  Gilligan     => sub { say 'Howdy Skipper!'     },
  Skipper      => sub { say 'Gilligan!!!!'       },
  'Mr. Howell' => sub { say 'Money money money!' },
  Ginger       => sub { say $luxuries[rand @luxuries] },
  };

Dump $hash;
```

We dump the value for hash, which gives us this output (minus some boring stuff we left out). Notice it also dumps `@luxuries` since it knows that the Ginger subroutine needs it:

```
my (@luxuries);
@luxuries = (
        'Diamonds',
        'Furs',
        'Caviar'
      );
$HASH1 = {
       Gilligan     => sub {...},
       Ginger       => sub {
               use warnings;
               use strict 'refs';
               BEGIN {
                 $^H{'feature_unicode'} = q(1);
                 $^H{'feature_say'} = q(1);
                 $^H{'feature_state'} = q(1);
                 $^H{'feature_switch'} = q(1);
               }
               say $luxuries[rand @luxuries];
               },
       "Mr. Howell" => sub {...},
       Skipper      => sub {...}
     };
```

Exercise

You can find the answer to this exercise in “This is section” on page 183.

Exercise [50 min]

The Professor modified some files on Monday afternoon, and now he's forgotten which ones they were. This happens all the time. He wants you to make a subroutine called `gather_mtime_between`, which, given a starting and ending timestamp, returns a pair of coderefs. The first one will be used with `File::Find` to gather the names of only the items that were modified between those two times; the second one should return the list of items found.

Here's some code to try; it should list only items that were last modified on the most recent Monday, although you could easily change it to work with a different day. (You don't have to type all of this code. This program should be available as the file named `ex6-1.plx` in the downloadable files, available on the O'Reilly web site.)

Hint: You can find a file's timestamp (`mtime`) with code such as:

```
my $timestamp = (stat $file_name)[9];
```

Because it's a slice, remember that those parentheses are mandatory. Don't forget that the working directory inside the callback isn't necessarily the starting directory in which `find` was called.

```
use File::Find;
use Time::Local;

my $target_dow = 1;        # Sunday is 0, Monday is 1, ...
my @starting_directories = (".");

my $seconds_per_day = 24 * 60 * 60;
my($sec, $min, $hour, $day, $mon, $yr, $dow) = localtime;
my $start = timelocal(0, 0, 0, $day, $mon, $yr);        # midnight today
while ($dow != $target_dow) {
  # Back up one day
  $start -= $seconds_per_day;        # hope no DST! :-)
  if (--$dow < 0) {
    $dow += 7;
  }
}
my $stop = $start + $seconds_per_day;

my($gather, $yield)  = gather_mtime_between($start, $stop);
find($gather, @starting_directories);
my @files = $yield->(  );

for my $file (@files) {
  my $mtime = (stat $file)[9];        # mtime via slice
  my $when = localtime $mtime;
```

```
    print "$when: $file\n";
}
```

Note the comment about DST. In many parts of the world, on the days when daylight savings time or summer time kicks in and out, the day is no longer 86,400 seconds long. The program glosses over this issue, but a more pedantic coder might take it into consideration appropriately.

CHAPTER 9
Filehandle References

We've seen arrays, hashes, and subroutines passed around in references, permitting a level of indirection to solve certain types of problems. We can also store `filehandles` in references, and we can open filehandles on more than just files. Let's look at the old problems then the new solutions.

The Old Way

In the olden days, Perl used barewords for filehandle names. The filehandle is another Perl data type, although people don't talk about it as a data type much since it doesn't get its own special sigil. You've probably already seen a lot of code that uses these bareword filehandles[1]:

```
open LOG_FH, '>>', 'castaways.log'
  or die "Could not open castaways.log: $!";
```

What happens if we want to pass around these filehandles so we could share them with other parts of our code, such as libraries? You've probably seen some tricky looking code that uses a typeglob or a reference to a typeglob[2]:

```
log_message( *LOG_FH, 'The Globetrotters are stranded with us!' );

log_message( \*LOG_FH, 'An astronaut passes overhead' );
```

In the `log_message()` routine, we take the first element off of the argument list and store it in another typeglob. Without going into too many details, a typeglob stores pointers to all the package variables of that name. When we assign one typeglob to another, we create aliases to the same data. We can now access the data, including the details of the filehandle, from another name. Then, when we use that name as a filehandle, Perl knows to look for the filehandle portion of the typeglob. We'd have a much easier time if filehandles had sigils!

1. We covered filehandle basics in *Learning Perl*
2. Learn more about typeglobs in *Mastering Perl*

```
sub log_message {
  local *FH = shift;

  print FH @_, "\n";
}
```

Notice our use of `local` there. A typeglob works with the symbol table, which means it's dealing with package variables. Package variables can't be lexical variables, so we can't use `my`. Since we don't want to stomp on anything else that might be named `FH` somewhere else in the script, we must use `local` to denote that the name `FH` has a temporary value for the duration of the `log_message` subroutine and that when the subroutine finishes, `perl` should restore any previous values to `FH` as if we were never there.

If all of that makes you nervous and wish that none of this stuff existed, that's good. Don't do this anymore! We put it in a section called "The Old Way" because there is a much better way to do it now. Pretend this section never existed and move onto the next one.

The Improved Way

Starting with Perl v5.6, `open` can create a filehandle reference in a normal scalar variable as long as the variable's value is undef. Instead of using a bareword for the filehandle name, we use a scalar variable whose value is undef:

```
my $log_fh;
open $log_fh, '>>', 'castaways.log'
  or die "Could not open castaways.log: $!";
```

If the scalar already has a value, this doesn't work because Perl won't stomp on our data. In the next example, Perl tries to use the value `5` as a symbolic reference, so it looks for a filehandle named `5`. That is, it tries to look for that filehandle, but `strict` will stop it:

```
my $log_fh = 5;
open $log_fh, '>>', 'castaways.log'
  or die "Could not open castaways.log: $!";
print $log_fh "We need more coconuts!\n";   # doesn't work
```

However, the Perl idiom is to do everything in one step. We can declare the variable right in the `open` statement. It looks funny at first, but after doing it a couple (okay, maybe several) times, you'll get used to it and like it better:

```
open my $log_fh, '>>', 'castaways.log'
  or die "Could not open castaways.log: $!";
```

When we want to print to the filehandle, we use the scalar variable instead of a bareword. Notice that there is still no comma after the filehandle:

```
print $log_fh "We have no bananas today!\n";
```

That syntax might look funny to you though, and even if it doesn't look funny to you, it might look odd to the person who has to read your code later. There's another problem, though. You may remember that `$_` is the default argument to `print`, even if you specify bareword filehandle:

```
$_ = 'Salt water batteries';
print;
print STDOUT;
print STDERR;
```

However, Perl doesn't know at compile-time what `$log_fh` is if that's your only argument to `print`. Should that be a filehandle or the value to send to standard output? Perl guesses that you want the value of `$log_fh` to go to standard output:

```
print $log_fh; # sends "GLOB(0x9bcd5c)" to standard output
```

We have the same problem if we accidently put a comma after our scalar filehandle. Perl thinks you want to send the value of `$log_fh` to standard output again:

```
# sends "GLOB(0x9bcd5c)Salt water batteries" to standard output
print $log_fh, 'Salt water batteries';
```

In *Perl Best Practices*, Damian Conway recommends putting braces around the filehandle portion to explicitly state what we intend. This syntax makes it look more like `grep` and `map` with inline blocks:

```
print {$log_fh} "We have no bananas today!\n";
```

Now we treat the filehandle reference just like any other scalar. We don't have to do any tricky magic to make it work:

```
log_message( $log_fh, 'My name is Mr. Ed' );

sub log_message {
  my $fh = shift;

  print $fh @_, "\n";
}
```

We can also create filehandle references from which we can read. We simply put the right thing in the second argument:

```
open my $fh, '<', 'castaways.log'
  or die "Could not open castaways.log: $!";
```

Now we use the scalar variable in place of the bareword in the line input operator. Before, we would have seen the bareword between the angle brackets:

```
while( <LOG_FH> ) { ... }
```

And now we see the scalar variable in its place:

```
while( <$log_fh> ) { ... }
```

In general, where we've seen the bareword filehandle we can substitute the scalar variable filehandle reference:

```
while( <$log_fh> ) { ... }
if( -t $log_fh ) { ... }
my $line = readline $log_fh;
close $log_fh;
```

In any of these forms, when the scalar variable goes out of scope (or we assign another value to it), Perl closes the file. We don't have to explicitly close the file ourselves.

Filehandles to strings

Since Perl 5.6, we can `open` a filehandle on a reference to a scalar instead of a file. That filehandle either reads from or writes to that string instead of a file (or pipe or socket).

Perhaps we need to capture the output that would normally go to a filehandle. If we `open` a write filehandle on a string and use that, the output never really has to leave our program and we don't have to pull any stunts to capture it:

```
open my $string_fh, '>', \ my $string;
```

For instance, we could save the state of a `CGI.pm` program, but we have to give `save` a filehandle. If we use our `$string_fh`, the data never have to leave our program:

```
use CGI;

open my $string_fh, '>', \ my $string;
CGI->save( $string_fh );
```

Similarly, `Storable` lets us pack data into a string, but `nstore` wants to save it to a named file, and `nstore_fd` will send it to a filehandle. If we want to capture it in a string, we use a string filehandle again:

```
use Storable;

open my $string_fh, '>', \ my $string;
nstore_fd \@data, $string_fh;
```

We could have merely used the `freeze` subroutine to save the result in a string:

```
my $string = freeze \@data;
```

However, using `freeze` means we are limited to only serializing to a string. If we decide to send the serialized data elsewhere, we're stuck. Using the string filehandle gives us flexibility. We can change our mind later.

We can also use filehandles to strings to capture output to `STDOUT` or `STDERR` for those pesky programs that want to clutter our screens. Sometimes we just want to keep our programs quiet, and somethings we want redirect the output. In this case, we have to close `STDOUT` first then reopen the filehandle. Since we usually don't want to lose the real `STDOUT`, we localize it within a scope so our replacement has a limited effect:

```
print "1. This goes to the real standard output\n";

my $string;
{
```

```
local *STDOUT;
close STDOUT;
open STDOUT, '>', \ $string;

print "2. This goes to the string\n";

$some_obj->noisy_method(); # this STDOUT goes to $string too
}

print "3. This goes to the real standard output\n";
```

We can design our own programs to be flexible so that others can decide where to send their data. If we let them specify the filehandle, they can decide if the output goes into a file, down a socket, or into a string. The best part of this flexibility is that the implementation is simple and the same for each of them:

```
sub output_to_fh {
  my( $fh, @data ) = @_;
  print $fh @data;
}
```

If we want to specialize `output_to_fh`, we just have to wrap it to provide the right sort of filehandle:

```
sub as_string {
  my( @data ) = @_;
  open my $fh, '>', \ my $string;
  output_to_fh( $string_fh, @data );
  $string;
}
```

Processing strings line-by-line

When we can treat a string as a file, many common tasks become much easier since we can use the filehandle interface. Consider, for instance, breaking a multiline string into lines. We could use `split` to break it up:

```
my @lines = split /$/, $multiline_string;
foreach my $line ( @lines ) {
  ... do something ...
}
```

Now, however, we have the data in two places, and the slightly annoying fragility that our pattern to `split` might not be the right one. Instead, we can open a filehandle for reading on a reference to that scalar then get its lines from the line input operator:

```
open my $string_fh, '<', \ $multiline_string;
while( <$string_fh> ) {
  ... do something ...
}
```

If our data come in from another source later, we just need a filehandle that we can read.

Since filehandle references are scalars, just like any other reference, we can treat them as scalars. Specifically, we can store them as array elements or hash values. We can have several filehandles open at the same time and decide later which one to use.

For instance, we want to go through our data from Chapter 7 and separate it into files based on the source machine (the first column):

```
professor.hut gilligan.crew.hut 1250
professor.hut lovey.howell.hut 910
thurston.howell.hut lovey.howell.hut 1250
professor.hut lovey.howell.hut 450
ginger.girl.hut professor.hut 1218
ginger.girl.hut maryann.girl.hut 199
```

We could read a line, open a filehandle based on the source machine name, store the filehandle in a hash, and print the rest of the line to that filehandle:

```
use v5.10;  # for state

while( <> ) {
  state $fhs;

  my( $source, $destination, $bytes ) = split;

  unless( exists $fhs->{$source} ) {
    open my $fh, '>>', $source or die '...';
    $fhs->{$source} = $fh;
  }

  say { $fhs->{$source} } "$destination $bytes";
}
```

We declare the `$fh` variable with `state` so it's private to the `while` loop but retains its value between iterations. Each time we need to create a new filehandle, we add it as a value to that hash reference. When the `while` is done, the `$fhs` goes out of scope, closing our filehandles for us.

If you are using an older Perl, you can get the same thing by moving `$fhs` out of the `while`, but scoping it with a naked block:

```
use v5.10;  # for state

{ # naked block to scope $fhs
  my $fhs;

  while( <> ) {
    ...
  }

}
```

IO::Handle and Friends

Behind the scenes, Perl is really using the `IO::Handle` module to work its filehandle magic, so our filehandle scalar is really an object[3] for an `IO::Handle` object. We could instead write our operations as methods:

```
use IO::Handle;

open my $fh, '>', $filename or die '...';
$fh->print( 'Coconut headphones' );
$fh->close;
```

As of Perl v5.14, we don't have to explicitly load `IO::Handle`, but with earlier versions we need to do that ourselves.

The `IO::Handle` package is a base class for input-output things, so it handles a lot more than just files. Unless we're creating new `IO` modules, we probably shouldn't use `IO::Handle` directly. Instead, we use some of the handy modules built on top of it. We haven't told you about object-oriented programming yet (it's in Chapter Chapter 14, so we almost have), but in this case you just have to follow the example in its documentation.

Some of these modules do some of the same things that we can already do with Perl's built-in `open` (depending on which version of Perl we have), but they can be handy when we want to decide as late as possible which module should handle input or output. Instead of using the built-in `open`, we use the module interface. To switch the behavior, we simply change the module name. Since we've set up our code to use a module interface, it's not that much work to switch modules.

IO::File

The `IO::File` module subclasses `IO::Handle` to work with files. It comes with the standard Perl distribution so you should already have it. There are a variety of ways to create an `IO::File` object.

We can create the filehandle reference with the one argument form of the constructor. We check the result of the operation by looking for a defined value in the filehandle reference variable:

```
use IO::File;

my $fh = IO::File->new( '> castaways.log' )
  or die "Could not create filehandle: $!";
```

3. Have you ever wondered why there is no comma after the filehandle portion of the `print`? It's really the indirect object notation (which we haven't mentioned yet unless you've read the whole book before you read the footnotes, like we told you to do in preface!

Since we don't like combining the `open` mode with the filename (for the same reasons as regular `open`), we'll use one of the other calling conventions. The optional second argument is the filehandle mode[4]:

```
my $read_fh  = IO::File->new( 'castaways.log', 'r' );

my $write_fh = IO::File->new( 'castaways.log', 'w' );
```

Using a bitmask as the mode allows for more granular control. The `IO::File` module supplies the constants:

```
my $append_fh = IO::File->new(
  'castaways.log', O_WRONLY|O_APPEND );
```

Besides opening named files, we might want to `open` an anonymous temporary file. On systems which support this sort of thing, we simply create the new object to get a read-write filehandle:

```
my $temp_fh = IO::File->new_tmpfile;
```

As before, Perl closes these files when the scalar variable goes out of scope, but if that's not enough, we can do it ourselves explicitly by either calling `close` or undefing the filehandle:

```
$temp_fh->close;

undef $append_fh;
```

Perl v5.6 and later can open an anonymous, temporary file if we give it `undef` as a filename. We probably want to both read and write from that file at the same time. Otherwise, it's a bit pointless to have a file we can't find later:

```
open my $fh, '+>', undef
  or die "Could not open temp file: $!";
```

IO::Scalar

If we're using an ancient version of Perl that can't create filehandles to a scalar reference, we can use `IO::Scalar` module. It uses the magic of `tie` behind-the-scenes to give us a filehandle reference that appends to a scalar. This module doesn't come with the standard Perl distribution so we'll have to install it ourself most likely:

```
use IO::Scalar;

my $string_log = '';
my $scalar_fh = IO::Scalar->new( \$string_log );

print $scalar_fh "The Howells' private beach club is closed\n";
```

4. These are the ANSI C `fopen` mode strings. We can also use these with the built-in `open`. Indeed, `IO::File` uses the built-in `open` behind-the-scenes.

Now our log message ends up in the scalar variable `$string_log` instead of a file. What if we want to read from our log file though? We do the same thing. In this example, we create `$scalar_fh` just as we did before, then read from it with the line input operator. In our `while` loop we'll extract the log messages that contain Gilligan (which is probably most of them since he's always part of the mess):

```
use IO::Scalar;

my $string_log = '';
my $scalar_fh = IO::Scalar->new( \$string_log );

while( <$scalar_fh> ) {
  next unless /Gilligan/;
  print;
}
```

IO::Tee

What if we want to send output to more than one place at a time? What if we want to send it to a file *and* save it in a string at the same time? Using what we know already, we'd have to do something like this:

```
open my $log_fh, '>>', 'castaways.log'
  or die "Could not open castaways.log";
open my $scalar_fh, '>>', \ my $string;

my $log_message = "The Minnow is taking on water!\n"
print $log_fh    $log_message;
print $scalar_fh $log_message;
```

Of course, we could shorten that a bit so we only have one `print` statement. We use `foreach` to iterate through the filehandle references, store each in `$fh` in turn, and print to each one:

```
foreach my $fh ( $log_fh, $scalar_fh ) {
  print $fh $log_message;
}
```

That's still a bit too much work. In the `foreach` we had to decide which filehandles to include. What if we could just define a group of filehandles that answered to the same name? Well, that's what `IO::Tee` does for us. Imagine it like a tee connector on a bilge output pipe; when the water gets to the tee, it can flow it two different directions at the same time. When our output gets to `IO::Tee`, it can go to two (or more) different channels at the same time. That is, `IO::Tee` *multiplexes* output. In this example, the castaways' log message goes to both the log file and the scalar variable:

```
use IO::Tee;

my $tee_fh = IO::Tee->new( $log_fh, $scalar_fh );

print $tee_fh "The radio works in the middle of the ocean!\n";
```

That's not all though. If the first argument to `IO::Tee` is an input filehandle (the succeeding arguments must be output filehandles), we can use the same tee-d filehandle to read from input and write to the output. The source and destination channels are different but we get to treat them as a single filehandle. `IO::Tee` does something special in this case: when it reads a line from the input filehandle, it immediately prints them to the output filehandles:

```
use IO::Tee;

my $tee_fh = IO::Tee->new( $read_fh, $log_fh, $scalar_fh );

# reads from $read_fh, prints to $log_gh and $scalar_fh
my $message = <$tee_fh>;
```

The `$read_fh` doesn't have to be connected to a file, either. It might also be connected to a socket, a scalar variable, or an external command's output. We can create readable filehandles to external commands with `IO::Pipe` or anything else we can dream up.

IO::Pipe

Sometimes our data don't come from a file or a socket, but from an external command. We can use a piped open to read from the output from a command. The | after `$command` notes that output is coming out of the command and flowing through the pipe into our program:

```
open my $pipe, '-|', $command
  or die "Could not open filehandle: $!";

while( <$pipe> ) {
  print "Read: $_";
}
```

This might be a bit easier with a module to handle the details for us. `IO::Pipe` is a frontend to `IO::Handle`. Given a command, it hands the `fork` and the `exec` for us and leaves us with a filehandle from which we can read the output of the command:

```
use IO::Pipe;

my $pipe = IO::Pipe->new;

$pipe->reader( "$^X -V" ); # $^X is the current perl executable

while( <$pipe> ) {
  print "Read: $_";
}
```

Similarly, we can write to a command too. We put the pipe in front of `$command` to show that data are coming into our program:

```
open my $pipe, "| $command"
  or die "Could not open filehandle: $!";

foreach ( 1 .. 10 ) {
```

```
  print $pipe "I can count to $_\n";
}
```

We can let `IO::Pipe` handle that for us too:

```
use IO::Pipe;

my $pipe = IO::Pipe->new;

$pipe->writer( $command );

foreach ( 1 .. 10 ) {
  print $pipe "I can count to $_\n";
}
```

IO::Null and IO::Interactive

Sometimes we don't want to send our output anywhere, but we are forced to send it somewhere. In that case, we can use `IO::Null` to create a filehandle that simply discards anything that we give it. It looks and acts just like a filehandle, but does nothing:

```
use IO::Null;

my $null_fh = IO::Null->new;

some_printing_thing( $null_fh, @args );
```

Other times, we want output in some cases but not in others. If we are logged in and running our program in our terminal, we probably want to see lots of output. However, if we schedule the job through *cron*, we probably don't care so much about the output as long as it does the job. The `IO::Interactive` module is smart enough to tell the difference:

```
use IO::Interactive;

print { is_interactive } 'Bamboo car frame';
```

The `is_interactive` subroutine returns a filehandle. Since the call to the subroutine is not a simple scalar variable, we surround it with braces to tell Perl that it's the filehandle.

Now that you know about "do nothing" filehandles, you can replace some ugly code that everyone tends to write. In some cases you want output and in some cases you don't, so many people use a post-expression conditional to turn off a statement in some cases:

```
print STDOUT "Hey, the radio's not working!" if $Debug;
```

Instead of that, you can assign different values to `$debug_fh` based on whatever condition you want, then leave off the ugly `if $Debug` at the end of every `print`:

```
use IO::Null;

my $debug_fh = $Debug ? *STDOUT : IO::Null->new;
```

```
$debug_fh->print( "Hey, the radio's not working!" );
```

The magic behind `IO::Null` might give a warning about "print() on unopened filehandle GLOB" with the indirect object notation (e.g. `print $debug_fh`) even though it works just fine. We don't get that warning with the direct form.

Directory Handle References

In the same way that we can create references to filehandles, we can create directory handle references:

```
opendir my $dh, '.' or die "Could not open directory: $!";

foreach my $file  ( readdir( $dh ) ) {
  print "Skipper, I found $file!\n";
}
```

The directory handle reference obeys the same rules we laid out before. This only works if the scalar variable does not already have a value and the handle automatically close when the variable goes out of scope or we assign it a new value.

IO::Dir

We can use object-oriented interfaces for directory handles too. The `IO::Dir` module has been part of the standard Perl distribution since Perl 5.6. It doesn't add interesting new features but wraps the perl built-in functions[5]:

```
use IO::Dir;

my $dir_fh = IO::Dir->new( '.' ) or
  die "Could not open dirhandle! $!\n";

while( defined( my $file = $dir_fh->read ) ) {
  print "Skipper, I found $file!\n";
}
```

We don't have to create a new directory handle if we decide we want to go through the list again (perhaps later in the program). We can rewind the directory handle to start over:

```
while( defined( my $file = $dir_fh->read ) ) {
  print "I found $file!\n";
  }

# time passes
$dir_fh->rewind;

while( defined( my $file = $dir_fh->read ) ) {
```

5. For each `IO::Dir` method name, append "dir", and look at the documentation in `perlfunc`.

```
    print "I can still find $file!\n";
}
```

Exercises

You can find the answers to these exercises in "This is section" on page 183.

Exercise 1 [20 min]

Write a program that prints the date and the day of the week, but allow the user to choose to send the output either to a file, a scalar, or both at the same time. No matter which output channels the user selects, send the output with a single `print` statement. If the user chose to send the output to a scalar, at the end of the program print the scalar's value to standard output.

Exercise 2 [30 min]

The Professor has to read a log file that looks like[6]:

```
Gilligan: 1 coconut
Skipper: 3 coconuts
Gilligan: 1 banana
Ginger: 2 papayas
Professor: 3 coconuts
MaryAnn: 2 papayas
...
```

He wants to write a series of files, called `gilligan.info`, `maryann.info`, and so on. Each file should contain all the lines that begin with that name. (Names are always delimited by the trailing colon.) At the end, `gilligan.info` should start with:

```
Gilligan: 1 coconut
Gilligan: 1 banana
```

Now the log file is large, and the coconut-powered computer is not very fast, so he wants to process the input file in one pass and write all output files in parallel. How does he do it?

Hint: Use a hash, keyed by the castaway name, holding `IO::File` objects for each output file. Create them as necessary.

Exercise 3 [15 min]

Write a program that takes in multiple directory names from the command line then prints out their contents. Use a function that takes a directory handle reference that you made with `opendir` or `IO::Dir`.

6. You can download sample data files from *http://www.intermediateperl.com*.

CHAPTER 10
Regular Expression References

Beginning with Perl 5.005, we can compile regular expressions and keep references to them without using a match or substitution operator. We can do all of our regular expression handling and preparation before we actually want to use them. Since a regular expression reference is just a scalar like any other reference, we can store it in an array or hash, pass it as an argument, interpolate it into a string, or use it in the many other ways we can use a scalar.

Before Regular Expression References

Most people usually see regular expressions as part of a match or substitution operator:

```
m/\bcoco.*/
s/\bcoconut\b/coconet/
split /coconut/, $string;
```

The pattern, however, is separate from the operator, and the operator merely applies the pattern to a string.

We might already have an inkling of this since we can interpolate a regular expression into one of the operators:

```
my $pattern = 'coco.*';

if( m/$pattern/ ) {
  ...
  }
```

In that case, `$pattern` is just a string like any other string and it has no idea about how we are going to use it. After the match operator interpolates the variable, it has to compile the resulting regular expression. Since the match has no idea ahead of time if the pattern is valid, like it would in a literal pattern, it might be a fatal runtime error. To handle that, we can catch that at match time with an `eval`:

```
print 'Enter a pattern: ';
chomp( my $pattern = <STDIN> );
```

```
print "Enter some lines:\n";
while( <STDIN> ) {
  if( eval { m/$pattern/ } ) {
    print "Match: $_";
  }

  if( $@ ) {
    die "There was a regex problem: $@\n";
  }
}
```

The error message shows where the pattern goes wrong when we use an invalid pattern that has an unmatched parentheses:

```
$ perl
Enter a pattern: Gilligan)
Enter some lines:
Gilligan & Skipper
There was a regex problem: Unmatched ) in regex;
marked by <-- HERE in m/Fred) <-- HERE /
at test line 8, <STDIN> line 2.
```

We have to do a similar thing if we want to use a regular expression as a subroutine argument. We pass a string as the argument then compile the pattern inside the subroutine:

```
find_match( 'Gilligan)' );

sub find_match {
  my( $pattern ) = @_;

  if( eval { m/$pattern/ } ) {
    ...
  }
}
```

As with the previous case, we get the error too late to do anything about it. We could write a subroutine to test a pattern before we try to use it:

```
sub test_regex {
  my( $pattern ) = @_;
  local( $_, $@ );

  eval { /$pattern/ };
  return defined $@ ? 0 : 1;
}
```

We can then use this subroutine to try a pattern before we actually want to. This isn't ideal for a many reasons, including the side effects of the match variables and the possibility that a regex can run arbirtary code. We don't want to trigger either of those just to validate a pattern. Besides, all of this is pretty messy and annoying. We'd have a much easier time if we could pre-compile the regular expression without applying it.

Precompiled Patterns

Perl 5.005 introduced a new quoting mechanism, the `qr//` operator. It's like other generalized quoting mechanisms[1], but it gives us a reference to a compiled regular expression:

```
my $regex = qr/Gilligan|Skipper/;
```

This doesn't apply the pattern to anything yet; we'll do that later. We can even check our pattern by printing it:

```
print $regex
```

Perl stringifies the pattern[2]:

```
(?^:Gilligan|Skipper)
```

We'll show you more about those extra parts in the stringified version in a moment.

The `qr//` doesn't solve our invalid pattern problem though, so we should still use the eval if we are going to interpolate a string:

```
my $regex = eval { qr/$pattern/ };
```

Like the other generalized quoters, we can choose different delimiters, which we typically do to avoid a problem with a literal character that we want to use in the pattern:

```
my $regex = qr(/usr/local/bin/\w+);
```

There's one delimiter that's special, just as with the match operator. If we use the single quotes as our delimiter, Perl treats our pattern as a single-quoted string. That is, Perl will not perform any double-quoted interpolation:

```
my $regex = qr'\n\t'; # four characters, not two
```

Even though the single quote delimiter disables any interpolation, any regular expression meta characters are still special. For instance, the `$` is still the end-of-string anchor. If you want a literal `$`, you need to escape it so Perl doesn't think it's the end of string anchor:

```
my $regex = qr'\$\d+\.\d+';
```

The rest of the regular expression is as normal; the `\d` is still the digit character class, the <+> is still a quantifer, and so on.

Regular Expression Options

When we use the match or substitution operators, we can put all of the flags that affect the operation at the end, after the final delimiter:

1. See "Quote and Quote-like Operators" in *perlop*
2. Different versions of Perl will stringify the reference differently, so you shouldn't rely on a particular string form. This form comes from Perl v5.14.

```
m/Gilligan$/migc
s/(\d+)/ $1 + 1 /eg
```

Since we put all of the options at the end of the operator, we didn't have to distinguish between flags that affect the pattern and flags that affected the operator[3].

When we use the `qr//` quoting, we can add only flags that affect the pattern (`/x /i /s /m /p /o /a /l /d /u`). We have two ways to do this. First, we can add the flags to the end of the `qr//` just like we did with the operators:

```
qr/Gilligan$/mi;
```

We can also add the flags directly in the pattern itself. This isn't a feature of the `qr//` âyou can use it in any pattern. The special sequence `(?flags:pattern)` allow us to specify the modifiers within the pattern itself:

```
qr/(?mi:Gilligan$)/;
```

The modifiers apply to only the part of the pattern enclosed in its parentheses. Suppose that we wanted only part of the pattern to be case-insensitive. I can

```
qr/abc(?i:Gilligan)def/;
```

We can remove modifiers from part of a pattern by prefixing the flags

```
qr/abc(?-i:Gilligan)def/i;
```

We can even add and remove modifiers at the same time. First we specify those that we want to add

```
qr/abc(?x-i:G i l l i g a n)def/i;
```

Applying Regex References

When we want to apply our regular expression reference, we have many options. We can interpolate it into a match or substitution operator:

```
my $regex = qr/Gilligan/;
$string =~ m/$regex/;
$string =~ s/$regex/Skipper/;
```

We can also just bind to it directly without the explicit match operator. The binding operator recognizes the regular expression reference and performs the match:

```
$string =~ $regex;
```

A smart match is the same thing:

```
$string ~~ $regex;
```

3. "Know the difference between regex and match operator flags", *http://www.effectiveperlprogramming.com/blog/174*

Regexes as Scalars

Now that we can store pre-compiled regular expressions in scalars, we can use them in the same ways that we use other scalars. We can stored them in arrays and hashes and we can pass them as arguments to subroutines.

Suppose you want to match multiple patterns at the same time. You could store each pattern in an array then go through each of them until you find a match:

```
use v5.10.1;

my @patterns = (
  qr/(?:Wiley )?Gilligan/,
  qr/Mary-Ann/,
  qr/Ginger/,
  qr/(?:The )?Professor/,
  qr/Skipper/,
  qr/Mrs?. Howell/,
);

my $name = 'Ginger';

foreach my $pattern ( @patterns ) {
  if( $name ~~ $pattern ) {
  say "Match!";
  last;
  }
}
```

That's not very nice. We have an `if` surrounded by a `foreach`. With a smart match, we can match against an array of patterns. The smart match will distribute over the elements

```
use v5.10.1;

my @patterns = (
  qr/(?:Wiley )?Gilligan/,
  qr/Mary-Ann/,
  qr/Ginger/,
  qr/(?:The )?Professor/,
  qr/Skipper/,
  qr/Mrs?. Howell/,
);

my $name = 'Ginger';

say "Match!" if $name ~~ @patterns;
```

That's still not very good. Which of those patterns matched? We can put them in a hash so we can give them labels:

```
use v5.10.1;

my %patterns = (
  Gilligan   => qr/(?:Wiley )?Gilligan/,
```

```
  'Mary-Ann' => qr/Mary-Ann/,
  Ginger     => qr/Ginger/,
  Professor  => qr/(?:The )?Professor/,
  Skipper    => qr/Skipper/,
  'A Howell' => qr/Mrs?. Howell/,
);

my $name = 'Ginger';

my( $match ) = grep { $name =~ $patterns{$_} } keys %patterns;

say "Matched $match" if $match;
```

We will make one more refinement. The `grep` will keep looking even after it finds a match, so we'll use `first` from `List::Util`:

```
use v5.10.1;
use List::Util qw(first);

my %patterns = (
  Gilligan   => qr/(?:Wiley )?Gilligan/,
  'Mary-Ann' => qr/Mary-Ann/,
  Ginger     => qr/Ginger/,
  Professor  => qr/(?:The )?Professor/,
  Skipper    => qr/Skipper/,
  'A Howell' => qr/Mrs?. Howell/,
);

my $name = 'Ginger';

my( $match ) = first { $name =~ $patterns{$_} } keys %patterns;

say "Matched $match" if $match;
```

Let's get a bit more fancy. What if there are multiple names that match, but we want the rightmost match? As you should already know from *Learning Perl*, Perl's regular expression engine finds the leftmost, longest match. We'll create a subroutine called `rightmost` which, given a string and a list of patterns, returns the starting position of the rightmost match:

```
my $position = rightmost(
  'Mary-Ann and Ginger',
  qr/Mary/, qr/Gin/,
);
```

Here's how we'll do it. We'll start with `$rightmost` at -1. If we get that position as the return value, we know nothing matched. Remember, the first position is 0, so if we get that, we know the pattern matched at the beginning of the string.

We can go through `@patterns` with `each`. In its Perl 5.12 form, it returns the index and the value for the next element[4]. Inside the `while`, we use the conditional operator to

4. It's only mildly more annoying to do this without `each`. You go through the indices and extract the value yourself. You're missing out on the fun of the new features though.

select a value based on the result of the match. If it's a successful match, we choose `$-[0]`, and `-1` otherwise. The `@-` special variable remembers the starting match positions of the entire pattern and the capture variables. The number in `$-[0]` is the starting position of the entire match[5]. If the match position is higher (so, more to the right) than a previously remembered position, we store that in `$rightmost`.

```
use v5.12;

sub rightmost {
  my( $string, @patterns ) = @_;

  my $rightmost = -1;
  while( my( $i, $pattern ) = each @patterns ) {
  $position = $string =~ m/$pattern/ ? $-[0] : -1;
  $rightmost = $position if $position > $rightmost;
  }

return $rightmost;
}
```

Let's put that all together now. We'll get the patterns as a hash slice with the sorted list of keys from our `%patterns` hash:

```
use v5.12;

my %patterns = (
  Gilligan   => qr/(?:Wiley )?Gilligan/,
  'Mary-Ann' => qr/Mary-Ann/,
  Ginger     => qr/Ginger/,
  Professor  => qr/(?:The )?Professor/,
  Skipper    => qr/Skipper/,
  'A Howell' => qr/Mrs?. Howell/,
);

my $position = rightmost(
  'There is Mrs. Howell, Ginger, and Gilligan',
  @patterns{ sort keys %patterns }
);

say "Rightmost match at position $position";
```

There are some other ways that we can improve on this, but we're going to leave some of that fun for you in the exercises. You don't have to remember everything going on here; just know that you can pass a regular expression reference as a subroutine argument. After all, it's just a scalar like every other reference.

5. There's also `@+`, which has the ending positions.

Build up Regular Expressions

We can precompile patterns and interpolate them into the match operators. We can also interpolate the patterns into other patterns. In this way, we can build up complicated patterns from smaller, more manageable pieces:

```
my $howells    = qr/(?:Thurston|Mrs)/;
my $tagalongs  = qr/(?:Ginger|Mary Ann)/;
my $passengers = qr/(?:$howells|$tagalong)/;
my $crew       = qr/(?:Gilligan|Skipper)/;

my $everyone   = qr/(?:$crew|$passengers)/;
```

Any regular expression metacharacters are still special when we use the regular expression references in bigger patterns. Those alternations in `$howells` and `$tagalongs` are still alternations in `$passengers`.

This allows us to decompose long, complicated patterns into smaller, easily-digestable chunks that we can put together any way that we like. Not only that, we can put them together in different ways. If we want to match a different group of people, we just put different parts together:

```
my $poor_people = qr/(?:$tagalongs|$passengers|$crew)/;
```

Here's a longer example, although you shouldn't ever half to write this regular expression on your own. RFC 1738 specifies that format for URLs, and we can turn that specification into regular expressions. This series of regular expression references is almost a direct translation of RFC 1738:

```
my $alpha       = qr/(?:[a-z])/;
my $digit       = qr/(?:\d)/;
my $alphadigit  = qr/(?i:$alpha|$digit)/;
my $safe        = qr/(?:[\$_.+-])/;
my $extra       = qr/(?:[!*'\(\),])/;
my $national    = qr/(?:[{}|\\^~\[\]`])/;
my $reserved    = qr|(?:[;/?:@&=])|;
my $hex         = qr/(?:(?i:$digit|[A-F]))/;
my $escape      = qr/(?:%$hex$hex)/;
my $unreserved  = qr/(?:$alpha|$digit|$safe|$extra)/;
my $uchar       = qr/(?:$unreserved|$escape)/;
my $xchar       = qr/(?:$unreserved|$reserved|$escape)/;
my $ucharplus   = qr/(?:(?:$uchar|[;?&=])*)/;
my $digits      = qr/(?:(?:$digit){1,})/;

my $hsegment    = $ucharplus;
my $hpath       = qr|(?:$hsegment(?:/$hsegment)*)|;
my $search      = $ucharplus;

my $scheme      = qr|(?i:https?://)|;
my $port        = qr/(?:$digits)/;
my $password    = $ucharplus;
my $user        = $ucharplus;

my $toplevel    = qr/(?:$alpha|$alpha(?:$alphadigit|-)*$alphadigit)/;
```

```
my $domainlabel = qr/(?:$alphadigit|$alphadigit
  (?:$alphadigit|-)*$alphadigit)/x;
my $hostname    = qr/(?:(?:$domainlabel\.)*$toplevel)/;
my $hostnumber  = qr/(?:$digits\.$digits\.$digits\.$digits)/;
my $host        = qr/(?:$hostname|$hostnumber)/;
my $hostport    = qr/(?:$host(?::$port)?)/;
my $login       = qr/(?:$user(?::$password)\@)?/;

my $urlpath     = qr/(?:(?:$xchar)*)/;
```

Notice that each pattern has its own non-capturing parentheses, `(?:...)`, so that when we put them together with other patterns they don't interfere with each other. Finally we can put it all together to represent the URL to make a program to look for URLs in text:

```
# ... all of those other lines
use v5.10.1;

my $httpurl = qr|(?:$scheme$hostport(?:/$hpath(?:\?$search)?)?)|;

while( <> ) {
  say if /$httpurl/;
}
```

That is, we could put it all together like that, but there's a better way to do it. Keep reading.

Regex-Creating Modules

Since we can create pre-compiled regular expressions, many people have created modules that create patterns for us. Instead of creating complicated patterns ourselves, and perhaps missing an edge case or specifying part of it incorrectly, we can rely on these well known modules to supply the patterns for us.

Using Common Patterns

Abigail, one of Perl's regex masters, put together a module to supply most of the complicated patterns that people try to make themselves (and usually mess up). Instead of creating our own pattern, we can use the one that `Regexp::Common` provides. It exports a hash reference named `$RE` that has as its values the regular expression references that we need. That long program from the previous section reduces to this much simpler one:

```
use v5.10.1;
use Regexp::Common qw(URI);

while( <> ) {
  print if /$RE{URL}{HTTP}/;
}
```

The hash gives us a regular expression reference, which we can stringify like any other reference:

```
use v5.10.1;
use Regexp::Common qw(zip);

say $RE{zip}{US};
```

The string we get is a bit complicated:

```
(?:(?:(?:USA?)-){0,1}(?:(?:(?:[0-9]{3})(?:[0-9]{2}))
(?:(?:-)(?:(?:[0-9]{2})(?:[0-9]{2}))){0,1}))
```

There are many other sorts of regular expressions that we can get from this module. If we wanted to find IPv4 addresses, such as *10.1.0.37*, we can use one of the patterns from the net portion of the module:

```
use v5.10.1;
use Regexp::Common qw(net);

while( <> ) {
  say if /$RE{net}{IPv4}/;
}
```

Behind the scenes, the `Regexp::Common` uses ties[6], so it can activate a lot of special magic from the keys that we decide to use. Suppose that we wanted to match numbers. We can start with one of the `Regexp::Common` patterns for numbers:

```
use v5.10.1;
use Regexp::Common qw(number);

while( <> ) {
  say if /$RE{num}{int}/;
}
```

That finds decimal integers just fine, but we can modify the pattern to find hexadecimal integers:

```
use v5.10.1;
use Regexp::Common qw(number);

while( <> ) {
  say if /$RE{num}{int}{ -base => 16 }/;
}
```

That program prints lines that contain numbers, but if we just want the number and not the entire line we can add a `{-keep}` key so the pattern captures the match:

```
use v5.10.1;
use Regexp::Common qw(number);

while( <> ) {
  say $1 if /$RE{num}{int}{ -base => 16 }{-keep}/;
}
```

6. See *Mastering Perl* or *perltie*.

Since `$RE` is a magic hash, so we don't even have to put the keys in any particular order. We can put the special keys starting with `-` anywhere that we like:

```
use v5.10.1;
use Regexp::Common qw(number);

while( <DATA> ) {
  say $1 if /$RE{ -base => 16 }{num}{ -keep }{int}/;
}
```

Assembling Regular Expressions

The `Regexp::Common` modules gives us pre-defined patterns, but there are also modules to help us build regular expressions. For instance, the `Regexp::Assemble` module helps us build efficient alternations. Consider the situation where we have an alternation in which most of the branches have a common prefix. Suppose we want to match either Mr. Howell, Mrs. Howell, or Mary-Ann. We could make a simple alternation:

```
my $passenger = qr/(?:Mr. Howell|Mrs. Howell|Mary-Ann)/;
```

All alternatives start with an *M*, but our simple approach checks for that *M* each time. Two or them have an *r* as the second letter.

That's not very efficient because the match might have to look at the same character several times to make sure it's the same thing it matched last time. Using the `Regexp::Assemble` module, we can put together the different parts:

```
use v5.10.1;
use Regexp::Assemble;

my $ra = Regexp::Assemble->new;
for ( 'Mr. Howell', 'Mrs. Howell', 'Mary-Ann' ) {
  $ra->add($_);
}

say $ra->re;
```

The module figures out a good way to put those together as an alternation so we don't check any character more than we should:

```
(?^:M(?:rs?. Howell|ary-Ann))
```

Exercise

You can find the answers to these exercises in "This is section" on page 184.

Exercise [30 min]

XXX:

Exercise [30 min]

Get the `rightmost` program running[7]. Once you have the example working, modify `rightmost` program take a hash reference of patterns and return the key of the rightmost match. Instead of calling it like:

```
my $position = rightmost(
  'There is Mrs. Howell, Ginger, and Gilligan',
  @patterns{ sort keys %patterns }
);
```

call it like:

```
my $key = rightmost(
  'There is Mrs. Howell, Ginger, and Gilligan',
  \%patterns
);
```

Exercise [30 min]

XXX:

7. You can get the program at *http://www.intermediateperl.com* if you don't want to type the whole thing yourself.

CHAPTER 11
Practical Reference Tricks

This chapter looks at optimizing sorting and dealing with recursively defined data.

Fancier Sorting

Perl's built-in `sort` operator sorts text strings in their natural text order, by default.[1] This is fine if we want to sort text strings:

```
my @sorted = sort qw(Gilligan Skipper Professor Ginger Mary-Ann);
```

but gets pretty messy when we want to sort numbers:

```
my @wrongly_sorted = sort 1, 2, 4, 8, 16, 32;
```

The resulting list is `1`, `16`, `2`, `32`, `4`, `8`. Why didn't `sort` order these properly? It treats each item as a string and sorts them in string order. Any string that begins with `3` sorts before any string that begins with `4`.

If we don't want the default sorting order, we don't need to write an entire sorting *algorithm*, which is good news since Perl already has a good one of those. But no matter what sorting algorithm we use, at some point we have to look at item A and item B and decide which one comes first. That's the part we'll write: Code to handle just two items. Perl will do the rest.

By default, as Perl orders the items, it uses a string comparison. We can specify a new comparison using a *sort block* that we place between the `sort` keyword and the list of things to sort.[2] Within the sort block, `$a` and `$b` stand in for two of the items `sort` will compare. If we're sorting numbers, then `$a` and `$b` will be two numbers from our list.

1. My friends call that the "ASCIIbetical" ordering. However, modern Perl doesn't use ASCII; instead, it uses a default sort order, depending on the current locale and character set; see the `perllocale` (not `perllocal`!) manpage.

2. We can also used a name subroutine that `sort` invokes for each comparison.

The sort block must return a coded value to indicate the sort order. If `$a` comes before `$b` in our desired sorting order, it should return `-1`; it should return `+1` if `$b` comes before `$a`; if the order doesn't matter, it should return `0`. The order might not matter, for example, if it's a case-insensitive sort comparing `"FRED"` to `"Fred"`, or if it's a numeric sort comparing `42` to `42`.[3]

For example, to sort those numbers in their proper order, we can use a sort block comparing `$a` and `$b`, like so:

```
my @numerically_sorted = sort {
  if ($a < $b)    { -1 }
  elsif ($a > $b) { +1 }
  else            {  0 }
} 1, 2, 4, 8, 16, 32;
```

Now we have a proper numeric comparison, so we have a proper numeric sort. Of course, this is far too much typing, so we can use the spaceship operator, `<=>`, instead:

```
my @numerically_sorted = sort { $a <=> $b } 1, 2, 4, 8, 16, 32;
```

The spaceship operator returns `-1`, `0`, and `+1`, according to the rules we laid out. A descending sort is simple in Perl[4]:

```
my @numerically_descending =
    reverse sort { $a <=> $b } 1, 2, 4, 8, 16, 32;
```

But there is more than one way to do it. The spaceship operator is nearsighted; it can't see which one of its parameters comes from `$a` and which from `$b`; it sees only which value is to its left and which is to its right. If we reverse the position of `$a` and `$b`, the spaceship will sort everything in the opposite order:

```
my @numerically_descending =
    sort { $b <=> $a } 1, 2, 4, 8, 16, 32;
```

In every place the previous sort expression returned `-1`, this expression returns `+1`, and vice versa. Thus, the sort is in the opposite order, and so it doesn't need a `reverse`. It's also easy to remember because if `$a` is to the left of `$b`, we get out the lower items first, just like `a` and `b` would be in the resulting list.

Which way is better? When should we use a `reverse sort`, and when should we switch `$a` and `$b`? Well, in most cases it shouldn't matter much for efficiency, so it's probably best to optimize for clarity and use `reverse`. For a more complex comparison, however, a single `reverse` may not be up to the task.

3. Actually, we can use any negative or positive number in place of `-1` and `+1`, respectively. Recent Perl versions include a default sorting engine that is *stable*, so zero returns from the `sort` block cause the relative ordering of `$a` and `$b` to reflect their order in the original list. Older versions of Perl didn't guarantee such stability, and a future version might not use a stable sort, so don't rely on it.

4. As of perl5.8.6, Perl recognizes the `reverse sort` and does it without generating the temporary, intermediate list.

Like the spaceship operator, we can indicate a string sort with `cmp`, although this is rarely used alone because it is the default comparison. The `cmp` operator is most often used in more complex comparisons, as we'll show shortly.

Sorting with Indices

In the same way we used indices to solve a few problems with `grep` and `map` back in Chapter 4, we can also use indices with `sort` to get some interesting results. For example, let's sort the list of names from earlier:

```
my @sorted = sort qw(Gilligan Skipper Professor Ginger Mary_Ann);
print "@sorted\n";
```

which necessarily results in:

```
Gilligan Ginger Mary_Ann Professor Skipper
```

But what if we wanted to look at the original list and determine which element of the original list now appears as the first, second, third, and so on, element of the sorted list? For example, Ginger is the second element of the sorted list and was the fourth element of the original list. How do we determine that the second element of the final list was the fourth element of the original list?

Well, we can apply a bit of indirection. Let's not sort the actual names but rather the indices of each name:

```
#   0        1       2         3      4
  my @input = qw(Gilligan Skipper Professor Ginger Mary_Ann);
  my @sorted_positions = sort { $input[$a] cmp $input[$b] } 0..$#input;
  print "@sorted_positions\n";
```

This time, `$a` and `$b` aren't the elements of the list, but the indices. So instead of comparing `$a` to `$b`, we use `cmp` to compare `$input[$a]` to `$input[$b]` as strings. The result of the `sort` are the indices, in an order defined by the corresponding elements of `@input`. This prints `0 3 4 2 1`, which means that the first element of the sorted list is element 0 of the original list, Gilligan. The second element of the sorted list is element 3 of the original list, which is Ginger, and so on. Now we can rank information rather than just move the names around.

Actually, we have the inverse of the rank. We still don't know for a given name in the original list about which position it occupies in the output list. But with a bit more magic, we can get there as well:

```
#   0        1       2         3      4
  my @input = qw(Gilligan Skipper Professor Ginger Mary_Ann);
  my @sorted_positions = sort { $input[$a] cmp $input[$b] } 0..$#input;
  my @ranks;
  @ranks[@sorted_positions] = (0..$#sorted_positions);
  print "@ranks\n";
```

The code prints `0 4 3 1 2`. This means that Gilligan is position 0 in the output list, Skipper is position 4, Professor is position 3, and so on. The positions here are 0-based, so add `1` to get "human" ordinal values. One way to cheat is to use `1..@sorted_positions` instead of `0..$#sorted_positions`, so a way to dump it all out looks like:

```
#   0       1       2        3      4
  my @input = qw(Gilligan Skipper Professor Ginger Mary_Ann);
  my @sorted_positions = sort { $input[$a] cmp $input[$b] } 0..$#input;
  my @ranks;
  @ranks[@sorted_positions] = (1..@sorted_positions);
  for (0..$#ranks) {
    print "$input[$_] sorts into position $ranks[$_]\n";
  }
```

This results in:

```
Gilligan sorts into position 1
Skipper sorts into position 5
Professor sorts into position 4
Ginger sorts into position 2
Mary_Ann sorts into position 3
```

This general technique can be convenient if we need to look at our data in more than one way. Perhaps we keep many records in order by a numeric code for efficiency reasons, but we occasionally want to view them in alphabetical order as well. Or maybe the data items themselves are impractical to sort, such as a month's worth of server logs.

Sorting Efficiently

As the Professor tries to maintain the community computing facility (built entirely out of bamboo, coconuts, and pineapples, and powered by a certified Perl-hacking monkey), he continues to discover that people are leaving entirely too much data on the single monkey-powered filesystem and decides to print a list of offenders.

The Professor has written a subroutine called `ask_monkey_about()` which, given a castaway's name, returns the number of pineapples of storage they use. We have to ask the monkey because he's in charge of the pineapples. An initial naive approach to find the offenders from greatest to least might be something like:

```
my @castaways =
  qw(Gilligan Skipper Professor Ginger Mary_Ann Thurston Lovey);
my @wasters = sort {
  ask_monkey_about($b) <=> ask_monkey_about($a)
} @castaways;
```

In theory, this would be fine. For the first pair of names (Gilligan and Skipper), we ask the monkey "How many pineapples does Gilligan have?" and "How many pineapples does Skipper have?" We get back two values from the monkey and use them to order Gilligan and Skipper in the final list.

However, at some point, we have to compare the number of pineapples that Gilligan has with another castaway as well. For example, suppose the pair is Ginger and Gilligan. We ask the monkey about Ginger, get a number back, and then ask the monkey about Gilligan...again. This will probably annoy the monkey a bit, since we already asked earlier. But we need to ask for each value two, three, or maybe even four times just to put the seven values into order.

This can be a problem because it irritates the monkey.

How do we keep the number of monkey requests to a minimum? Well, we can build a table first. We use a `map` with seven inputs and seven outputs, turning each castaway item into a separate array reference, with each referenced array consisting of the castaway name and the pineapple count reported by the monkey:

```
my @names_and_pineapples = map {
  [ $_, ask_monkey_about($_) ]
} @castaways;
```

At this point, we asked the monkey seven questions in a row, but that's the last time we have to talk to the monkey! We now have everything we need to finish the task.

For the next step, we sort the arrayrefs, ordering them by the monkey-returned value:

```
my @sorted_names_and_pineapples = sort {
  $b->[1] <=> $a->[1];
} @names_and_pineapples;
```

In this subroutine, `$a` and `$b` are still two elements from the list of things to be sorted. When we're sorting numbers, `$a` and `$b` are numbers; when we're sorting references, `$a` and `$b` are references. We dereference them to get to the corresponding array itself, and pick out item 1 from the array (the monkey's pineapple value). Because `$b` appears to the left of `$a`, it'll be a descending sort as well. We want a descending sort because the Professor wants the first name on the list to be the person who uses the most pineapples.

We're almost done, but what if we just wanted the top names, rather than the names and pineapple counts? We merely need to perform another `map` to transform the references back to the original data:

```
my @names = map $_->[0], @sorted_names_and_pineapples;
```

Each element of the list ends up in `$_`, so we'll dereference that to pick out the element 0 of that array, which is just the name.

Now we have a list of names, ordered by their pineapple counts, and the monkey's off our backs, all in three easy steps.

The Schwartzian Transform

The intermediate variables between each of these steps were not necessary, except as input to the next step. We can save ourselves some brainpower by just stacking all the steps together:

```
my @names =
  map $_->[0],
  sort { $b->[1] <=> $a->[1] }
  map [ $_, ask_monkey_about($_) ],
  @castaways;
```

Because the `map` and `sort` operators are right to left, we have to read this construct from the bottom up. Take a list of `@castaways`, create some arrayrefs by asking the monkey a simple question, sort the list of arrayrefs, and then extract the names from each arrayref. This gives us the list of names in the desired order.

This construct is commonly called the *Schwartzian Transform*, which was named after Randal (but not by Randal), thanks to a Usenet posting he made many years ago. The Schwartzian Transform has since proven to be a very nice thing to have in our bag of sorting tricks.

If this transform looks like it might be too complex to memorize or come up with from first principles, it might help to look at the flexible and constant parts:

```
my @output_data =
  map $_->[0],
  sort { SORT COMPARISON USING $a->[1] AND $b->[1] }
  map [ $_, EXPENSIVE FUNCTION OF $_ ],
  @input_data;
```

The basic structure maps the original list into a list of arrayrefs, computing the expensive function only once for each; sorts those array refs, looking at the cached value of each expensive function invocation;[5] and then extracts the original values back out in the new order. All we have to do is plug in the proper two operations, and we're done. For example, to use the Schwartzian Transform to implement a case-insensitive sort, we could use code like this:[6]

```
my @output_data =
  map $_->[0],
  sort { $a->[1] cmp $b->[1] }
  map [ $_, "\U$_" ],
  @input_data;
```

5. An *expensive* operation is one that takes a relatively long time or a relatively large amount of memory.

6. This is an efficient way to do this only if the uppercasing operation is sufficiently expensive, which it might be if our strings tend to be very long or if we have a large enough number of them. For a small number of not-long strings, a simple `my @output_data = sort { "\U$a" cmp "\U$b"} @input_data` is probably more efficient. If in doubt, benchmark.

Multi-level sort with the Schwartzian Transform

If we need to sort on more than one criterion, the Schwartzian Transform is still up to the task.

```
my @output_data =
  map $_->[0],
  sort { SORT COMPARISON USING $a->[1] AND $b->[1] or
    ANOTHER USING $a->[2] AND $b->[2] or
  YET ANOTHER USING $a->[3] AND $b->[3] }
  map [ $_, SOME FUNCTION OF $_, ANOTHER, YET ANOTHER ],
  @input_data;
```

This code skeleton has a three-level `sort` comparison, using three computed values saved in the anonymous array (alongside the original data item to be sorted, which always comes first).

Recursively Defined Data

While the data we've processed with references up to this point has been rather fixed-structure, sometimes we have to deal with hierarchical data, which is often defined recursively.

For Example One, consider an HTML table that has rows containing cells—and some of those cells may also contain entire tables. Example Two could be a visual representation of a filesystem consisting of directories containing files and other directories. Example Three is a company organization chart, which has managers with direct reports, some of which may be managers themselves. And Example Four is a more complex organization chart, which can contain instances of the HTML tables of Example One, the filesystem representations of Example Two, or even entire organization charts.

We can use references to acquire, store, and process such hierarchical information. Frequently, the routines to manage the data structures end up as recursive subroutines.

Recursive algorithms deal with the unlimited complexity of their data by beginning with a base case, and building upon that. Recursive functions should all have a base, or trivial case, where it doesn't need to recurse and that all other recursions can eventually reach. That is, unless we have a lot of time on our hands to let the function recurse forever. The base case considers what to do in the simplest case: when the leaf node has no branches, when the array is empty, when the counter is at zero. In fact, it's common to have more than one base case in various branches of a recursive algorithm. A recursive algorithm with no base case is an infinite loop.

A recursive subroutine has a branch from which it calls itself to handle a portion of the task, and a branch that doesn't call itself to handle the base cases. In Example One above, the base case could be a table cell which is empty. There could also be base cases for empty tables and table rows. In Example Two, base cases would be needed for files, and perhaps for empty directories.

For example, a recursive subroutine handling the factorial function, which is one of the simplest recursive functions, might look like[7]:

```
sub factorial {
  my $n = shift;
  if ($n <= 1) {
    return 1;
  } else {
    return $n * factorial($n - 1);
  }
}
```

Here we have a base case where `$n` is less than or equal to 1, which does not invoke the recursive instance, along with a recursive case for `$n` greater than 1, which calls the routine to handle a portion of the problem (i.e., compute the factorial of the next lower number).

This task would probably be solved better using iteration rather than recursion, even though the classic definition of factorial is often given as a recursive operation.

Building Recursively Defined Data

Suppose we wanted to capture information about a filesystem, including the filenames and directory names, and their included contents. We'll represent a directory as a hash, in which the keys are the names of the entries within the directory, and values are `undef` for plain files. A sample */bin* directory looks like:

```
my $bin_directory = {
  cat  => undef,
  cp   => undef,
  date => undef,
  ... and so on ...
};
```

Similarly, the Skipper's home directory might also contain a personal *bin* directory (at something like *~skipper/bin*) that contains personal tools:

```
my $skipper_bin = {
  navigate            => undef,
  discipline_gilligan => undef,
  eat                 => undef,
};
```

Nothing in either structure tells where the directory is located in the hierarchy. It just represents the contents of some directory.

Go up one level to the Skipper's home directory, which is likely to contain a few files along with the personal *bin* directory:

7. This is the example you find because other languages have something called *tail recurion* where the compiler can recogize this situation and convert it to an iterative solution so it doesn't actually recurse. This means you have to be careful with Perl's recursion.

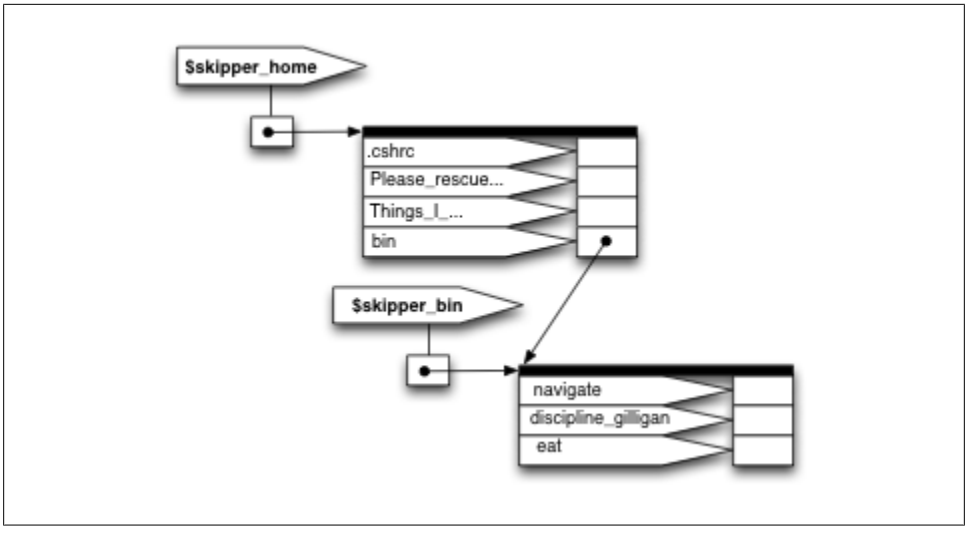

Figure 11-1. Skipper home PeGS

```
my $skipper_home = {
  '.cshrc'                      => undef,
  'Please_rescue_us.pdf'        => undef,
  'Things_I_should_have_packed' => undef,
  bin                           => $skipper_bin,
};
```

Ahh, notice that we have three files, but the fourth entry `bin` doesn't have `undef` for a value but rather the hash reference created earlier for the Skipper's personal *bin* directory. This is how we indicate subdirectories. If the value is `undef`, it's a plain file; if it's a hash reference, we have a subdirectory, with its own files and subdirectories. Of course, we can have combined these two initializations:

```
my $skipper_home = {
  '.cshrc'                      => undef,
  'Please_rescue_us.pdf'        => undef,
  'Things_I_should_have_packed' => undef,

  bin => {
    navigate            => undef,
    discipline_gilligan => undef,
    eat                 => undef,
  },
};
```

Now the hierarchical nature of the data starts to come into play.

Obviously, we don't want to create and maintain a data structure by changing literals in the program. We should fetch the data by using a subroutine. Write a subroutine that returns `undef` for a given pathname if the path is a file, or a hash reference of the

directory contents if the path is a directory. The base case of looking at a file is the easiest, so let's write that:

```
sub data_for_path {
  my $path = shift;
  if (-f $path) {
    return undef;
  }
  if (-d $path) {
    ...
  }
  warn "$path is neither a file nor a directory\n";
  return undef;
}
```

If the Skipper calls this on *.cshrc*, he'll get back an `undef` value, indicating that a file was seen.

Now for the directory part. We need a hash reference, which we declare as a named hash inside the subroutine. For each element of the hash, we call ourselves to populate the value of that hash element. It goes something like this:

```
sub data_for_path {
  my $path = shift;
  if (-f $path or -l $path) {        # files or symbolic links
    return undef;
  }
  if (-d $path) {
    my %directory;
    opendir PATH, $path or die "Cannot opendir $path: $!";
    my @names = readdir PATH;
    closedir PATH;
    for my $name (@names) {
    next if $name eq '.' or $name eq '..';
    $directory{$name} = data_for_path("$path/$name");
    }
    return \%directory;
  }
  warn "$path is neither a file nor a directory\n";
  return undef;
}
```

The base cases in this recursive algorithm are the files and symbolic links. This algorithm wouldn't correctly traverse the filesystem if it followed symbolic links to directories as if they were true (hard) links since it could end up in a circular loop if the symlink pointed to a directory that contained the symlink.[8] It would also fail to correctly traverse a malformed filesystem—that is, one in which the directories form a ring rather than a tree structure, say. Although malformed filesystems may not often be an issue,

8. Not that any of us have ever done that and wondered why the program took forever. The second time really wasn't our faults anyway, and the third time was just bad luck. That's our story and we're sticking to it.

recursive algorithms in general are vulnerable to errors in the structure of the recursive data.

For each file within the directory being examined, the response from the recursive call to `data_for_path` is `undef`. This populates most elements of the hash. When the reference to the named hash is returned, the reference becomes a reference to an anonymous hash because the name immediately goes out of scope. (The data itself doesn't change, but the number of ways in which we can access the data changes.)

If there is a subdirectory, the nested subroutine call uses `readdir` to extract the contents of that directory and returns a hash reference, which is inserted into the hash structure created by the caller.

At first, it may look a bit mystifying, but if we walk through the code slowly, we'll see it's always doing the right thing. Test the results of this subroutine by calling it on `.` (the current directory) and inspecting the result:

```
use Data::Dumper;
print Dumper(data_for_path('.'));
```

Obviously, this will be more interesting if our current directory contains subdirectories.

Displaying Recursively Defined Data

The `Dumper` routine of `Data::Dumper` displays the output nicely, but what if we don't like the format being used? We can write a routine to display the data. Again, for recursively defined data, a recursive subroutine is usually the key.

To dump the data, we need to know the name of the directory at the top of the tree because that's not stored within the structure:

```
sub dump_data_for_path {
my $path = shift;
my $data = shift;

if (not defined $data) { # plain file
  print "$path\n";
  return;
}
  ...
  }
```

For a plain file, dump the pathname; for a directory, `$data` is a hash reference. Let's walk through the keys and dump the values:

```
sub dump_data_for_path {
  my $path = shift;
  my $data = shift;

  if (not defined $data) { # plain file
    print "$path\n";
    return;
```

```
    }

  foreach (sort keys %$data) {
    dump_data_for_path("$path/$_", $directory{$_});
  }
}
```

For each element of the directory, we pass a path consisting of the incoming path followed by the current directory entry, and the data pointer is either `undef` for a file or a subdirectory hash reference for another directory. We can see the results by running:

```
dump_data_for_path('.', data_for_path('.'));
```

Again, this is more interesting in a directory that has subdirectories, but the output should be similar to calling `find` from the shell prompt:

```
find . -print
```

Avoiding Recursion

We used recursion in the previous examples so that we could show you how to do it, but it's not the only way to get the job done. Now we'll code our solutions using iterative solutions. Why? Most people learn recursion because some other language has a feature than can take what looks like recursive code and turn it into an iterative solution. It's easier to conceive as a recursive algorithm, but that doesn't mean that the program actually recurses. In Perl, however, we don't get that benefit of that behind-the-scenes rearrangement.

However, there are other benefits to iterative solutions. In the recursive version of `data_for_path`, we constructed the directory tree depth first, and we can only do it depth first. With a recursive solution we have to work all the way down to the bottom before we can move on to the next thing at the top level.

To make an iterative solution, we follow a basic template. We have to manage our own queue of things that we need to process. As long as we have things in the queue, we keep going. When we exhaust the queue, we return the data structure we created. The template looks like:

```
sub iterative_solution {
  my( $start ) = @_;

  my $data = {};
  my @queue = ( [ $start, $data ] );

  while( my $next = shift @queue ) {
    ... process current element ...
    ... add new things to @queue ...
    }

  return $data;
  }
```

In the template, each item in `@queue` carries along everything we need to process that element as an anonymous array. In this case, `$start` is the thing we need to process and `$data` is the reference where we need to store the result. Although we only have two things in that anonymous array, but we'll add some more later to add an additional, very attractive, feature.

First, let's merely translate the recursive solution into an iterative one, using the same depth first behavior:

```
use File::Basename;
use File::Spec::Functions;

my $data = data_for_path( '/Users/Gilligan/Desktop' );

sub data_for_path {
  my( $path ) = @_;

  my $data = {};

  my @queue = ( [ $path, $data ] );

  while( my $next = shift @queue ) {
    my( $path, $ref ) = @$next;

    my $basename = basename( $path );

    $ref->{$basename} = do {
      if( -f $path or -l $path ) { undef }
      else {
        my $hash = {};
        opendir my $dh, $path;
        my @new_paths = map {
          catfile( $path, $_ )
          } grep { ! /^\.\.?\z/ } readdir $dh;

        unshift @queue, map { [ $_, $hash ] } @new_paths;
        $hash;
      }
    };
  }

  $data;
}
```

Inside the `while` loop, we get the next element to process. It has the path it needs to process and the reference for its result. Since `$path` is the full path and we only want the filename, we get the basename to use as the key. Once we have the key to add to `$ref`, we have to decide what the value should be. Inside the `do` block, we have two branches: in the file or link case, the value is `undef`, and in the directory case, we have to create new items to process and add them to the queue.

To create the new element, we create a new reference in `$hash`. This is a reference we're going to store the result for the new item we need to process. This is the reference that

is going into the anonymous array for the item to process. This is the cool reference trick: even though we create the reference separately, when we assign it as the value to `$ref-{$basename}>`, even as the empty anonymous hash, it becomes part of the data structure. We don't know where it is in the data structure, but we don't have to know.

Since we want to make this a depth first algorithm, when we have a new element to process, we put it at the front of `@queue` by using `unshift`. There's nothing in this version that makes it better than the recursive solution. It's a bit longer, it's more complicated to understand the first time we see it, and it doesn't necessarily run any faster.

The breadth-first solution

Now that we've created the iterative solution, we can do quite a bit more than the recursive version ever could. We can easily change it to a breadth-first algorithm, and it's the matter of changing a single keyword. With `unshift` we put the newly discovered items at the front of the queue. If we use `push` instead, we put newly discovered items at the end of the queue:

```
# unshift @queue, map { [ $_, $hash ] } @new_paths;
push @queue, map { [ $_, $hash ] } @new_paths;
```

If it matters to you, the depth-first version is Last In First Out (LIFO) and the breadth-first version is First In First Out (FIFO). The code for either is almost identical. If you're a computer science purist, you may point out that the depth-first version is actually a stack instead of a queue, but let's just say it's a queue with a VIP section where more important VIPs keep cutting in line.

The breadth-first has an extremely attractive ability: we can easily stop at any level that we like. If we only want to go three levels deep, we just don't add any items to process that would be deeper than that. We could engineer the recursive solution to do this too, but it looks pretty ugly.

In our iterative solution, we keep track of our level in the anonymous arrays that we store in `@queue`. In the `else` branch in the `do` block, we only add elements to `@queue` if the current level is below the threshold since the elements we add are one level deeper:

```
use File::Basename;
use File::Spec::Functions;

my $data = data_for_path( "/Users/brian/Desktop", 2 );
print Dumper( $data ); use Data::Dumper;

sub data_for_path {
  my( $path, $threshold ) = @_;

  my $data = {};

  my @queue = ( [ $path, 0, $data ] );

  while( my $next = shift @queue ) {
```

```
        my( $path, $level, $ref ) = @$next;

        my $basename = basename( $path );

        $ref->{$basename} = do {
          if( -f $path or -l $path ) { undef }
          else {
            my $hash = {};
            if( $level < $threshold ) {
              opendir my $dh, $path;
              my @new_paths = map {
                catfile( $path, $_ )
                } grep { ! /^\.\.?\z/ } readdir $dh;

              push @queue, map { [ $_, $level + 1, $hash ] } @new_paths;
            }
            $hash;
          }
        };

      }

      $data;
    }
```

Nifty, eh? It gets even better, though. It's not too tough to let the user decide if they want depth-first or breadth first traversals. That might be interesting if they want to process a certain number of items at a time, leaving some of them unprocessed. We aren't going to spoil the fun you'll have as you work on that in Exercise 5 though.

Exercises

You can find the answers to these exercises in "This is section" on page 184.

Exercise 1 [15 min]

Using the `glob` operator, a naive sort of every name in the */bin* directory by their relative sizes might be written as:

```
my @sorted = sort { -s $a <=> -s $b } glob "/bin/*";
```

Rewrite this using the Schwartzian Transform technique.

If you don't have many files in the */bin* directory, perhaps because you don't have a Unix machine, change the argument to `glob` as needed.

Exercise 2 [15 min]

Read up on the `Benchmark` module, included with Perl. Write a program that will answer the question, "How much does using the Schwartzian Transform speed up the task of Exercise 1?"

Exercise 3 [10 min]

Using a Schwartzian Transform, read a list of words, and sort them in "dictionary order." Dictionary order ignores all capitalization and internal punctuation. Hint: The following transformation might be useful:

```
my $string = 'Mary-Ann';
$string =~ tr/A-Z/a-z/;       # force all lowercase
$string =~ tr/a-z//cd;        # strip all but a-z from the string
print $string;                # prints "maryann"
```

Be sure you don't mangle the data! If the input includes `the Professor`, and `The skipper`, the output should have them listed in that order, with that capitalization.

Exercise 4 [20 min]

Modify the recursive directory dumping routine so it shows the nested directories through indentation. An empty directory should show up as:

```
sandbar, an empty directory
```

while a nonempty directory should appear with nested contents, indented two spaces:

```
uss_minnow, with contents:
  anchor
  broken_radio
  galley, with contents:
    captain_crunch_cereal
    gallon_of_milk
    tuna_fish_sandwich
  life_preservers
```

Exercise 5 [20 minutes]

Modify the iterative version of `data_for_path` to handle both depth-first or breadth-first traversal. Use an optional third argument to allow the user to decide which one to use:

```
my $depth   =
  data_for_path( $start_dir, $threshold, 'depth-first' );

my $breadth =
  data_for_path( $start_dir, $threshold, 'breadth-first' );
```

CHAPTER 12

Building Larger Programs

This is section

This is a dummy para

CHAPTER 13

Creating Your Own Perl Distribution

This is section

This is a dummy para

CHAPTER 14

Introduction to Objects

This is section

This is a dummy para

CHAPTER 15

Introduction to Testing

This is section

This is a dummy para

CHAPTER 16

Objects with Data

This is section

This is a dummy para

CHAPTER 17

Some Advanced Object Topics

This is section

This is a dummy para

CHAPTER 18

Exporter

This is section

This is a dummy para

CHAPTER 19

Object Destruction

This is section

This is a dummy para

CHAPTER 20

This is a dummy title

This is section

This is a dummy para

CHAPTER 21

Essential Testing

This is section

This is a dummy para

CHAPTER 22

Advanced Testing

This is section

This is a dummy para

CHAPTER 23

Contributing to CPAN

This is section

This is a dummy para

APPENDIX

Appendix

This is section

This is a dummy para

This is section

This is a dummy para

This is section

This is a dummy para

This is section

This is a dummy para

This is section

This is a dummy para

This is section

This is a dummy para

This is section

This is a dummy para

This is section

This is a dummy para

This is section

This is a dummy para

This is section

This is a dummy para

This is section

This is a dummy para

This is section

This is a dummy para

This is section

This is a dummy para

This is section

This is a dummy para

This is section

This is a dummy para

This is section

This is a dummy para

This is section

This is a dummy para

This is section

This is a dummy para

This is section

This is a dummy para

This is section

This is a dummy para

This is section

This is a dummy para