Effective SOFTWARE DEVELOPMENT SERIES

Scott Meyers, Consulting Editor

Effective PERL PROGRAMMING

Ways to Write Better, More Idiomatic Perl

SECOND EDITION

Joseph N. Hall • Joshua A. McAdams • brian d foy

Effective Perl Programming

Second Edition

Effective Perl Programming

Ways to Write Better, More Idiomatic Perl

Second Edition

Joseph N. Hall
Joshua A. McAdams
brian d foy

Addison-Wesley
Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Hall, Joseph N., 1966–
Effective Perl programming : ways to write better, more idiomatic Perl / Joseph N. Hall, Joshua McAdams, Brian D. Foy. — 2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-321-49694-2 (pbk. : alk. paper)
1. Perl (Computer program language) I. McAdams, Joshua. II. Foy, Brian D III. Title.
QA76.73.P22H35 2010
005.13'3—dc22

2010001078

ISBN-13: 978-0-321-49694-2
ISBN-10: 0-321-49694-9
Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.
Second printing, June 2011

Contents at a Glance

This page intentionally left blank

Contents

Foreword

When I first learned Perl more than a decade ago, I thought I knew the language pretty well; and indeed, I knew the *language* well enough. What I didn't know were the idioms and constructs that really give Perl its power. While it's perfectly possible to program without these, they represent a wealth of knowledge and productivity that is easily missed.

Luckily for me, I had acquired the first edition of Joseph N. Hall's *Effective Perl Programming,* and it wasn't to be found in my bookshelf. Instead, it had an almost permanent place in my bag, where I could easily peruse it whenever I found a spare moment.

Joseph's format for *Effective Perl Programming* was delightfully simple: small snippets of wisdom; easily digested. Indeed, it formed the original inspiration for our free Perl Tips (http://perltraining.com.au/tips/) newsletter, which continues to explore both Perl and its community.

A lot can change in a language in ten years, but even more can change in the community's understanding of a language over that time. Consequentially, I was delighted to hear that not only was a second edition in the works, but that it was to be written by two of the most prominent members of the Perl community.

To say that brian is devoted to Perl is like saying that the sun's corona is rather warm. brian has not only literally written volumes on the language, but also publishes a magazine (*The Perl Review*), manages Perl's FAQ, and is a constant and welcome presence on community sites devoted to both Perl and programming.

Josh is best known for his efforts in running *Perlcast*, which has been providing Perl news in audio form since 2005. Josh's abilities to consistently interview the brightest and most interesting people in the world not only make him an ideal accumulator of knowledge, but also have me very jealous.

As such, it is with great pleasure that I have the opportunity to present to you the second edition of this book. May it help you on your way to Perl mastery the same way the first edition did for me.

—Paul Fenwick
Managing Director
Perl Training Australia

Preface

Many Perl programmers cut their teeth on the first edition of *Effective Perl Programming*. When Addison-Wesley first published it in 1998, the entire world seemed to be using Perl; the dot-com days were in full swing and anyone who knew a little HTML could get a job as a programmer. Once they had those jobs, programmers had to pick up some quick skills. *Effective Perl Programming* was likely to be one of the books those new Perl programmers had on their desks along with the bibles of Perl, *Programming Perl*[1] and *Learning Perl*[2].

There were many other Perl books on the shelves back then. Kids today probably won't believe that you could walk into a bookstore in the U.S. and see hundreds of feet of shelf space devoted to computer programming, and most of that seemed to be Java and Perl. Walk into a bookstore today and the computer section might have its own corner, and each language might have a couple of books. Most of those titles probably won't be there in six months.

Despite all that, *Effective Perl Programming* hung on for over a decade. Joseph Hall's insight and wisdom toward the philosophy of Perl programming is timeless. After all, his book was about thinking in Perl more than anything else. All of his advice is still good.

However, the world of Perl is a lot different than it was in 1998, and there's a lot more good advice out there. CPAN (the Comprehensive Perl Archive Network), which was only a few years old then, is now Perl's killer feature. People have discovered new and better ways to do things, and with more than a decade of additional Perl experience, the best practices and idioms have come a long way.

---

1. Larry Wall, Tom Christiansen, and Jon Orwant, *Programming Perl, Third Edition* (Sebastopol, CA: O'Reilly Media, 2000).

2. Randal L. Schwartz, Tom Phoenix, and brian d foy, *Learning Perl*, *Fifth Edition* (Sebastopol, CA: O'Reilly Media, 2008).

Since the first edition of *Effective Perl Programming*, Perl has changed, too. The first edition existed during the transition from Perl 4 to Perl 5, so people were still using their old Perl 4 habits. We've mostly done away with that distinction in this edition. There is only one Perl, and it is major version 5. (Don't ask us about Perl 6. That's a different book for a different time.)

Modern Perl now handles Unicode and recognizes that the world is more than just ASCII. You need to get over that hump, too, so we've added an entire chapter on it. Perl might be one of the most-tested code bases, a trend started by Michael Schwern several years ago and now part of almost every module distribution. Gone are the days when Perlers celebrated the Wild West days of code slinging. Now you can have rapid prototyping and good testing at the same time. If you're working in the enterprise arena, you'll want to read our advice on testing. If you're a regular-expression fiend, you'll want to use all of the new regex features that the latest versions of Perl provide. We'll introduce you to the popular ones.

Perl is still growing, and new topics are always emerging. Some topics, like Moose, the post-modern Perl object system, deserve their own books, so we haven't even tried to cover them here. Other topics, like POE (Perl Object Environment), object-relational mappers, and GUI toolkits are similarly worthy, and also absent from this book. We're already thinking about *More Effective Perl*, so that might change.

Finally, the library of Perl literature is much more mature. Although we have endeavored to cover most of the stuff we think you need to know, we've left out some areas that are much better covered in other books, which we list in Appendix B. That makes space for other topics.

—Joseph N. Hall, Joshua A. McAdams, and brian d foy

Preface from the first edition

I used to write a lot of C and C++. My last major project before stepping into the world of Perl full time was an interpreted language that, among other things, drew diagrams, computed probabilities, and generated entire FrameMaker books. It comprised over 50,000 lines of platform-independent C++ and it had all kinds of interesting internal features. It was a fun project.

It also took two years to write.

It seems to me that most interesting projects in C and/or C++ take months or years to complete. That's reasonable, given that part of what makes an undertaking interesting is that it is complex and time-consuming. But it also seems to me that a whole lot of ideas that start out being mundane and uninteresting become interesting three-month projects when they are expressed in an ordinary high-level language.

This is one of the reasons that I originally became interested in Perl. I had heard that Perl was an excellent scripting language with powerful string-handling, regular-expression, and process-control features. All of these are features that C and C++ programmers with tight schedules learn to dread. I learned Perl, and learned to like it, when I was thrown into a project where most of my work involved slinging text files around—taking output from one piece of software and reformatting it so that I could feed it to another. I quickly found myself spending less than a day writing Perl programs that would have taken me days or weeks to write in a different language.

How and why I wrote this book

I've always wanted to be a writer. In childhood I was obsessed with science fiction. I read constantly, sometimes three paperbacks a day, and every so often wrote some (bad) stories myself. Later on, in 1985, I attended the Clarion Science Fiction & Fantasy Writers' workshop in East Lansing, Michigan. I spent a year or so occasionally working on short-story manuscripts afterward, but was never published. School and work began to consume more and more of my time, and eventually I drifted away from fiction. I continued to write, though, cranking out a technical manual, course, proposal, or paper from time to time. Also, over the years I made contact with a number of technical authors.

One of them was Randal Schwartz. I hired him as a contractor on an engineering project, and managed him for over a year. (This was my first stint as a technical manager, and it was quite an introduction to the world of management in software development, as anyone who knows Randal might guess.) Eventually he left to pursue teaching Perl full time. And after a while, I did the same.

While all this was going on, I became more interested in writing a book. I had spent the past few years working in all the "hot" areas—C++, Perl, the Internet and World Wide Web—and felt that I ought to be able to find

something interesting in all that to put down on paper. Using and teaching Perl intensified this feeling. I wished I had a book that compiled the various Perl tricks and traps that I was experiencing over and over again.

Then, in May 1996, I had a conversation with Keith Wollman at a developers' conference in San Jose. I wasn't really trying to find a book to write, but we were discussing what sorts of things might be good books and what wouldn't. When we drifted onto the topic of Perl, he asked me, "What would you think of a book called *Effective Perl*?" I liked the idea. Scott Meyers's *Effective C++* was one of my favorite books on C++, and the extension of the series to cover Perl was obvious.

I couldn't get Keith's idea out of my head, and after a while, with some help from Randal, I worked out a proposal for the book, and Addison-Wesley accepted it.

The rest . . . Well, that was the fun part. I spent many 12-hour days and nights with FrameMaker in front of the computer screen, asked lots of annoying questions on the Perl 5 Porters list, looked through dozens of books and manuals, wrote many, many little snippets of Perl code, and drank many, many cans of Diet Coke and Pepsi. I even had an occasional epiphany as I discovered very basic things about Perl that I had never realized I was missing. After a while, a manuscript emerged.

This book is my attempt to share with the rest of you some of the fun and stimulation that I experienced while learning the power of Perl. I certainly appreciate you taking the time to read it, and I hope that you will find it useful and enjoyable.

—Joseph N. Hall
Chandler, Arizona
1998

Acknowledgments

For the second edition

Several people have helped us bring about the second edition by reading parts of the manuscript in progress and pointing out errors or adding things we hadn't considered. We'd like to thank Abigail, Patrick Abi Salloum, Sean Blanton, Kent Cowgill, Bruce Files, Mike Fraggasi, Jarkko Hietaniemi, Slaven Rezic, Andrew Rodland, Michael Stemle, and Sinan Ünür. In some places, we've acknowledged people directly next to their contribution.

Some people went much further than casual help and took us to task for almost every character. All the mistakes you don't see were caught by Elliot Shank, Paul Fenwick, and Jacinta Richardson. Anything left over is our fault: our cats must have walked on our keyboards when we weren't looking.

—Joseph N. Hall, Joshua A. McAdams, and brian d foy

From the first edition

This book was hard to write. I think mostly I made it hard on myself, but it would have been a lot harder had I not had help from a large cast of programmers, authors, editors, and other professionals, many of whom contributed their time for free or at grossly inadequate rates that might as well have been for free. Everyone who supported me in this effort has my appreciation and heartfelt thanks.

Chip Salzenberg and Andreas "MakeMaker" König provided a number of helpful and timely fixes to Perl bugs and misbehaviors that would have complicated the manuscript. It's hard to say enough about Chip. I've spent a little time mucking about in the Perl source code. I hold him in awe.

Many other members of the Perl 5 Porters list contributed in one way or another, either directly or indirectly. Among the most obviously helpful

and insightful were Jeffrey Friedl, Chaim Frenkel, Tom Phoenix, Jon Orwant (of *The Perl Journal*), and Charlie Stross.

Randal Schwartz, author, instructor, and "Just Another Perl Hacker," was my primary technical reviewer. If you find any mistakes, e-mail him. (Just kidding.) Many thanks to Randal for lending his time and thought to this book.

Thanks also to Larry Wall, the creator of Perl, who has answered questions and provided comments on a number of topics.

I've been very lucky to work with Addison-Wesley on this project. Everyone I've had contact with has been friendly and has contributed in some significant way to the forward progress of this project. I would like to extend particular thanks to Kim Fryer, Ben Ryan, Carol Nelson, and Keith Wollman.

A number of other people have contributed comments, inspiration, and/or moral support. My friends Nick Orlans, Chris Ice, and Alan Piszcz trudged through several revisions of the incomplete manuscript. My current and former employers Charlie Horton, Patrick Reilly, and Larry Zimmerman have been a constant source of stimulation and encouragement.

Although I wrote this book from scratch, some of it by necessity parallels the description of Perl in the Perl man pages as well as *Programming Perl*. There are only so many ways to skin a cat. I have tried to be original and creative, but in some cases it was hard to stray from the original description of the language.

Many thanks to Jeff Gong, for harassing The Phone Company and keeping the T-1 alive. Jeff really knows how to keep his customers happy.

Many thanks to the sport of golf for keeping me sane and providing an outlet for my frustrations. It's fun to make the little ball go. Thanks to *Master of Orion* and *Civilization II* for much the same reasons.

Most of all, though, I have to thank Donna, my soulmate and fiancée, and also one heck of a programmer. This book would not have come into being without her seemingly inexhaustible support, patience, and love.

—Joseph N. Hall
1998

About the Authors

Joseph N. Hall, a self-professed "computer whiz kid," grew up with a TI programmable calculator and a Radio Shack TRS-80 Model 1 with 4K RAM. He taught his first computer class at the age of 14. Joseph holds a B.S. in computer science from North Carolina State University and has programmed for a living since 1984. He has worked in UNIX and C since 1987 and has been working with Perl since 1993. His interests include software tools and programming languages, piano and electronic keyboards, and golf.

Joshua A. McAdams has been an active member of the Perl community for nearly five years. He is the voice of *Perlcast*, hosted two YAPC::NAs in Chicago, conducts meetings for Chicago.pm, has spoken about Perl at conferences around the world, and is a CPAN (Comprehensive Perl Archive Network) author. Though this is his first book, he has authored Perl articles for *The Perl Review* and the Perl Advent Calendar. For a day job, Josh has the privilege to work at Google, where his day-to-day development doesn't always involve Perl, but he sneaks it in when he can.

brian d foy is the coauthor of *Learning Perl, Fifth Edition* (O'Reilly Media, 2008), and *Intermediate Perl* (O'Reilly Media, 2006) and the author of *Mastering Perl* (O'Reilly Media, 2007). He established the first Perl user group, the New York Perl Mongers; publishes *The Perl Review;* maintains parts of the Perl core documentation; and is a Perl trainer and speaker.

This page intentionally left blank

Introduction

"Learning the fundamentals of a programming language is one thing; learning how to design and write effective programs in that language is something else entirely." What Scott Meyers wrote in the Introduction to Effective C++ is just as true for Perl.

Perl is a Very High Level Language—a VHLL for the acronym-aware. It incorporates high-level functionality like regular expressions, networking, and process management into a context-sensitive grammar that is more "human," in a way, than that of other programming languages. Perl is a better text-processing language than any other widely used computer language, or perhaps any other computer language, period. Perl is an incredibly effective scripting tool for UNIX administrators, and it is the first choice of most UNIX CGI scripters worldwide. Perl also supports object-oriented programming, modular software, cross-platform development, embedding, and extensibility.

Is this book for you?

We assume that you already have some experience with Perl. If you're looking to start learning Perl, you might want to wait a bit before tackling this book. Our goal is to make you a better Perl programmer, not necessarily a new Perl programmer.

This book isn't a definitive reference, although we like to think that you'd keep it on your desktop. Many of the topics we cover can be quite complicated and we don't go into every detail. We try to give you the basics of the concepts that should satisfy most situations, but also serve as a starting point for further research if you need more. You will still need to dive into the Perl documentation and read some of the books we list in Appendix A.

There is a lot to learn about Perl.

Once you have worked your way through an introductory book or class on Perl, you have learned to write what Larry Wall, Perl's creator, fondly refers to as "baby talk." Perl baby talk is plain, direct, and verbose. It's not bad—you are allowed and encouraged to write Perl in whatever style works for you.

You may reach a point where you want to move beyond plain, direct, and verbose Perl toward something more succinct and individualistic. This book is written for people who are setting off down that path. *Effective Perl Programming* endeavors to teach you what you need to know to become a fluent and expressive Perl programmer. This book provides several different kinds of advice to help you on your way.

- **Knowledge, or perhaps, "Perl trivia."** Many complex tasks in Perl have been or can be reduced to extremely simple statements. A lot of learning to program effectively in Perl entails acquiring an adequate reservoir of experience and knowledge about the "right" ways to do things. Once you know good solutions, you can apply them to your own problems. Furthermore, once you know what good solutions look like, you can invent your own and judge their "rightness" accurately.
- **How to use CPAN.** The Comprehensive Perl Archive Network is modern Perl's killer feature. With over 5 gigabytes of Perl source code, major frameworks, and interfaces to popular libraries, you can accomplish quite a bit with work that people have already done. CPAN makes common tasks even easier with Perl. As with any language, your true skill is your ability to leverage what has already been done.
- **How to solve problems.** You may already have good analytical or debugging skills from your work in another programming language. This book teaches you how to beat your problems using Perl by showing you a lot of problems and their Perl solutions. It also teaches you how to beat the problems that Perl gives you, by showing how to efficiently create and improve your programs.
- **Style.** This book shows you idiomatic Perl style, primarily by example. You learn to write more succinct and elegant Perl. If succinctness isn't your goal, you at least learn to avoid certain awkward constructs. You also learn to evaluate your efforts and those of others.
- **How to grow further.** This book doesn't cover everything you need to know. Although we do call it a book on advanced Perl, not a whole lot of advanced Perl can fit between its covers. A real compendium of

advanced Perl would require thousands of pages. What this book is really about is how you can make yourself an advanced Perl programmer—how you can find the resources you need to grow, how to structure your learning and experiments, and how to recognize that you have grown.

We intend this as a thought-provoking book. There are subtleties to many of the examples. Anything really tricky we explain, but some other points that are simple are not always obvious. We leave those to stand on their own for your further reflection. Sometimes we focus on one aspect of the example code and ignore the surrounding bits, but we try to make those as simple as possible. Don't be alarmed if you find yourself puzzling something out for a while. Perl is an idiosyncratic language, and in many ways is very different from other programming languages you may have used. Fluency and style come only through practice and reflection. While learning is hard work, it is also enjoyable and rewarding.

The world of Perl

Perl is a remarkable language. It is, in our opinion, the most successful modular programming environment.

In fact, Perl modules are the closest things to the fabled "software ICs" (that is, the software equivalent of integrated circuits, components that can be used in various applications without understanding all of their inner workings) that the software world has seen. There are many reasons for this, one of the most important being that there is a centralized, coordinated module repository, CPAN, which reduces the amount of energy wasted on competing, incompatible implementations of functionality. See Appendix A for more resources.

Perl has a minimal but sufficient modular and object-oriented programming framework. The lack of extensive access-control features in the language makes it possible to write code with unusual characteristics in a natural, succinct form. It seems to be a natural law of software that the most-useful features are also the ones that fit existing frameworks most poorly. Perl's skeletal approach to "rules and regulations" effectively subverts this law.

Perl provides excellent cross-platform compatibility. It excels as a systems administration tool on UNIX because it hides the differences between

different versions of UNIX to the greatest extent possible. Can you write cross-platform shell scripts? Yes, but with extreme difficulty. Most mere mortals should not attempt such things. Can you write cross-platform Perl scripts? Yes, easily. Perl also ports reasonably well between its UNIX birthplace and other platforms, such as Windows, VMS, and many others.

As a Perl programmer, you have some of the best support in the world. You have complete access to the source code for all the modules you use, as well as the complete source code to the language itself. If picking through the code for bugs isn't your speed, you have online support available via the Internet 24 hours a day, 7 days a week. If free support isn't your style, you can also buy commercial support.

Finally, you have a language that dares to be different. Perl is fluid. At its best, in the presence of several alternative interpretations, Perl does what you mean (sometimes seen as **DWIM**, "do what I mean"). A scary thought, perhaps, but it's an indication of true progress in computing, something that reaches beyond mere cycles, disk space, and RAM.

Terminology

In general, the terminology used with Perl isn't so different than that used to describe other programming languages. However, there are a few terms with slightly peculiar meanings. Also, as Perl has evolved, some terminology has faded from fashion and some new terminology has arisen.

In general, the name of the language is Perl, with a capital P, and `perl` is the name of the program that compiles and runs your source. Unless we are specifically referring to the interpreter, we default to using the capitalized version.

An **operator** in Perl is a nonparenthesized syntactical construct. (The arguments to an operator may, of course, be contained in parentheses.) A **list operator**, in particular, is an identifier followed by a list of elements separated by commas:

```
print "Hello", chr(44), " world!\n";
```

A **function** in Perl is an identifier followed by a pair of parentheses that completely encloses the arguments:

```
print("Hello", chr(44), " world!\n");
```

Now, you may have just noticed a certain similarity between list operators and functions. In fact, in Perl, there is no difference other than the syntax used. We will generally use the term "operator" when we refer to Perl built-ins like `print` and `open`, but may use "function" occasionally. There is no particular difference in meaning.

The proper way to refer to a subroutine written in Perl is just **subroutine**. Of course, "function," "operator," and even "procedure" will make acceptable literary stand-ins. Note that Perl's use of "function" isn't the same as the mathematical definition, and some computer scientists may shudder at Perl's abuse of the term.

All Perl **methods** are really subroutines that conform to certain conventions. These conventions are neither required nor recognized by Perl. However, it is appropriate to use phrases like "call a method," since Perl has a special method-call syntax that is used to support object-oriented programming. A good way of defining the (somewhat elusive) difference is that a method is a subroutine that the author intends you to call via method-call syntax.

A Perl **identifier** is a "C symbol"—a letter or underscore followed by one or more letters, digits, or underscores. Identifiers are used to name Perl variables. Perl variables are identifiers combined with the appropriate punctuation, as in `$a` or `&func`.

Although not strictly in keeping with the usage in the internals of Perl, we use the term **keyword** to refer to the small number of identifiers in Perl that have distinctive syntactical meanings—for example, `if` and `while`. Other identifiers that have ordinary function or operator syntax, such as `print` and `oct`, we call **built-ins**, if anything.

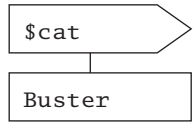
An **lvalue** (pronounced "ell value") is a value that can appear on the left-hand side of an assignment statement. This is the customary meaning of the term; however, there are some unusual constructs that act as lvalues in Perl, such as the `substr` operator.

Localizing a variable means creating a separate scope for it that applies through the end of the enclosing block or file. Special variables must be localized with the `local` operator. You can localize ordinary variables with either `my` or `local` (see Item 43 in Chapter 4). This is an unfortunate legacy of Perl, and Larry Wall wishes he had used another name for `local`, but life goes on. We say "localize with `my`" when it makes a difference.

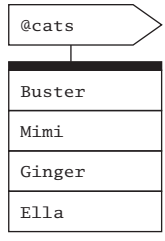
Notation

In this book we use Joseph's PEGS (PErl Graphical Structures) notation to illustrate data structures. It should be mostly self-explanatory, but here is a brief overview.

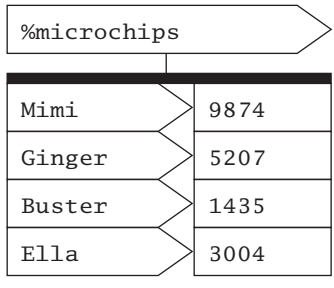
Variables are values with names. The name appears in a sideways "picket" above the value. A scalar value is represented with a single rectangular box:



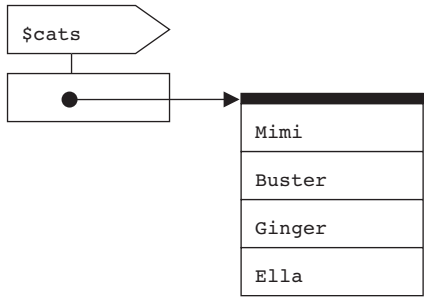
Arrays and lists have a similar graphical representation. Values are shown in a stack with a thick bar on top:



A hash is represented with a stack of names next to a stack of corresponding values:



References are drawn with dots and arrows as in those LISP diagrams from days of yore:



That's all there is to the basics.

Perl style

Part of what you should learn from this book is a sense of good Perl style.

Style is, of course, a matter of preference and debate. We won't pretend to know or demonstrate The One True Style, but we hope to show readers one example of contemporary, efficient, effective Perl style. Sometimes our style is inconsistent when that aids readability. Most of our preference comes from the *perlstyle* documentation.

The fact that the code appears in a book affects its style somewhat. We're limited in line lengths, and we don't want to write overly long programs that stretch across several pages. Our examples can't be too verbose or boring—each one has to make one or two specific points without unnecessary clutter. Therefore, you will find some deviations from good practice.

In some examples, we want to highlight certain points and de-emphasize others. In some code, we use `...` to stand in for code that we've left out. Assume that the `...` stands for real code that should be there. (Curiously, by the time this book hits the bookstores, that `...` should also be compilable Perl. Perl 5.12 introduces the "yadda yadda" operator, which compiles just fine, but produces a run time error when you try to execute it. It's a nice way to stub out code.)

Some examples need certain versions of Perl. Unless we specify otherwise, the code should run under Perl 5.8, which is an older but serviceable

version. If we use a Perl 5.10 feature, we start the example with a line that notes the version (see Item 2 in Chapter 1):

```
use 5.010;
```

We also ignore development versions of Perl, where the minor version is an odd number, such as 5.009 and 5.011. We note the earliest occurrence of features in the first stable version of Perl that introduces it.

Not everything runs cleanly under `warnings` or `strict` (Item 3). We advise all Perl programmers to make use of both of these regularly. However, starting all the examples with those declarations may distract from our main point, so we leave them off. Where appropriate, we try to be `strict` clean, but plucking code out of bigger examples doesn't always make that practical.

We generally minimize punctuation (Item 18). We're not keen on "Perl golf," where people reduce their programs to as few characters as they can. We just get rid of the unnecessary characters and use more whitespace so the important bits stand out and the scaffolding fades into the background.

Finally, we try to make the examples meaningful. Not every example can be a useful snippet, but we try to include as many pieces of real-world code as possible.

Organization

The first two chapters generally present material in order of increasing complexity. Otherwise, we tend to jump around quite a bit. Use the table of contents and the index, and keep the Perl documentation close at hand (perhaps by visiting http://perldoc.perl.org/).

We reorganized the book for the second edition. Appendix B shows a mapping from Items in the first edition to Items in this edition. We split some first-edition Items into many new ones and expanded them; some we combined; and some we left out, since their topics are well covered in other books. Appendix A contains a list of additional resources we think you should consider.

The book doesn't really stop when you get to the end. We're going to keep going at http://effectiveperlprogramming.com/. There you can find more news about the book, some material we left out, material we didn't have time to finish for the book, and other Perl goodies.

5 Files and Filehandles

It's easy to work with files in Perl. Its heritage includes some of the most powerful utilities for processing data, so it has the tools it needs to examine the files that contain those data and to easily read the data and write them again.

Perl's strength goes beyond mere files, though. You probably think of files as things on your disk with nice icons. However, Perl can apply its filehandle interface to almost anything. You can use the filehandle interface to do most of the heavy lifting for you. You can also store filehandles in scalar variables, and select which one you want to use later.

Item 51. Don't ignore the file test operators.

One of the more frequently heard questions from newly minted Perl programmers is, "How do I find the size of a file?" Invariably, another newly minted Perler will give a wordy answer that works, but requires quite a bit of typing:

```
my (
  $dev,   $ino,     $mode, $nlink, $uid,
  $gid,   $rdev,    $size, $atime, $mtime,
  $ctime, $blksize, $blocks
) = stat($filename);
```

Or, perhaps they know how to avoid the extra variables that they don't want, so they use a slice (Item 9):

```
my ($size) = ( stat $filename )[7];
```

When you are working this hard to get something that should be common, stop to think for a moment. Perl is specifically designed to make the common things easy, so this should be really easy. And indeed, it is if you use the `-s` file test operator, which tells you the file size in bytes:

```
my $size = -s $filename;
```

Many people overlook Perl's file test operators. Maybe they are old C programmers, maybe they've seen only the programs that other people write, or they just don't trust them. This is a shame; they are succinct and efficient, and tend to be more readable than equivalent constructs written using the `stat` operator. Curiously, the file test operators are the first functions listed in *perlfunc,* because they are under the literal `-X`. If you want to read about them, you tell `perldoc` to give you the function named `-X`:

```
% perldoc -f -X
```

File tests fit into loops and conditions very well. Here, for example, is a list of the text files in a directory. The `-T` file test decides if the contents are text by sampling part of the file and guessing.

Almost all file tests use `$_` by default:

```
my @textfiles = grep { -T } glob "$dir_name/*";
```

The `-M` and `-A` file tests return the modification and access times of the file, but in days relative to the start of the program. That is, Perl takes the time the program was started, subtracts the time the file was modified or accessed, and gives you back the result in days. Positive values are in the past, and negative values indicate times after the start of the program. That seems really odd, but it makes it easy to measure age in terms a human can understand. If you want to find the files that haven't been modified in the past seven days, you look for a `-M` value that is greater than 7:

```
my $old_files = grep { -M > 7 } glob '*';
```

If you want to find the files modified after your program started, you look for negative values. In this example, if `-M` returns something less than zero, `map` gives an anonymous array that has the name of the file and the modification age in days; otherwise, it gives the empty list:

```
my @new_files = map { -M < 0 ? [ $_, -M ] : () } glob '*';
```

Reusing work

If you want to find all of the files owned by the user running the program that are executable, you can combine the file tests in a `grep`:

```
my @my_executables = grep { -o and -x } glob '*';
```

The file test operators actually do the `stat` call for you, figure out the answer, and give it back to you. Each time you run a file test, Perl does another `stat`. In the last example, Perl did two `stats` on `$_`.

If you want to use another file test operator on the same file, you can use the virtual _ filehandle (the single underscore). It tells the file test operator to not call `stat` and instead reuse the information from the last file test or `stat`. Simply put the _ after the file test you want. Now you call only one `stat` for each item in the list:

```
my @my_executables = grep { -o and -x _ } glob '*';
```

Stacked file tests

Starting with Perl 5.10, you can stack file test operators. That is, you test the same file or filehandle for several properties at the same time. For instance, if you want to check that a file is both readable and writable by the current user, you list the `-r` and `-w` file tests before the file:

```
use 5.010;

if ( -r -w $file ) {
  print "File is readable and writable\n";
}
```

There's nothing especially magic about this, since it's a syntactic shortcut for doing each operation independently. Notice that the equivalent long form does the test closest to the file first:

```
if ( -w $file and -r $file ) {
  print "File is readable and writable\n";
}
```

Rewriting the example from the previous section, you'd have:

```
my @my_executables = grep { -o -x } glob '*';
```

Things to remember

- Don't call `stat` directly when a file test operator will do.
- Use the _ virtual filehandle to reuse data from the last `stat`.
- Stack file test operators in Perl 5.10 or later.

Item 52. Always use the three-argument `open`.

A long time ago, in a Perl far, far away, you had to specify the filehandle mode and the filename together:

```
open( FILE, '> output.txt' ) || die ...;  # OLD and WRONG
```

That code isn't so bad, but things can get weird if you use a variable for the filename:

```
open( FILE, $read_file ) || die ...;  # WRONG and OLD
```

Since the data in `$read_file` can do two jobs, specify the mode and the filename, someone might try to pull a fast one on you by making a weird filename. If they put a > at the beginning of the filename, all of a sudden you've lost your data:

```
$read_file = '> birdie.txt';  # bye bye birdie!
```

The two-argument form of `open` has a **magic** feature where it interprets these redirection symbols. Unfortunately, this feature can leave your code open to exploits and accidents.

Imagine that the person trying to wreak havoc on your files decides to get a little more tricky. They think that you won't notice when they open a file in read-write mode. This allows the input operator to work on an open file, but also overwrites your data:

```
$read_file = '+> important.txt';
```

They could even sneak in a pipe, which tells `open` to run a command:

```
$read_file = 'rm -rf / |';  # that's gonna hurt!
```

And now, just when you think you have everything working, the software trolls come out at three in the morning to ensure that your pager goes off just when you get to sleep.

Since Perl 5.6, you can use the three-argument `open` to get around this problem. By "can," we mean, "you always will from now on forever and ever."

When you want to read a file, you ensure that you *only* read from a file:

```
open my ($fh), '<', $read_file or die ...;
```

The filename isn't doing double duty anymore, so it has less of a chance of making a mess. None of the characters in `$read_file` will be special. Any redirection symbols, pipes, or other funny characters are literal characters.

Likewise, when you want to write to a file, you ensure that you get the right mode:

```
open my ($fh), '>',  $write_file  or die ...;
open my ($fh), '>>', $append_file or die ...;
```

The two-argument form of `open` protects you from extra whitespace. Part of the filename processing magic lets Perl trim leading and trailing whitespace from the filename. Why would you ever want whitespace at the beginning or end? We won't pretend to know what sorts of crazy things you want. With the three-argument `open`, you can keep that whitespace in your filename. Try it sometime: make a filename that starts with a newline. Did it work? Good. We'll let you figure out how to delete it.

Things to remember

- Use the three-argument form of `open` when you can.
- Use lexical scalars to store filehandle references.
- Avoid precedence problems by using `or` to check the success of `open`.

Item 53. Consider different ways of reading from a stream.

You can use the line input operator <> to read either a single line from a stream in a scalar context or the entire contents of a stream in a list context. Which method you should use depends on your need for efficiency, access to the lines read, and other factors, like syntactic convenience.

In general, the line-at-a-time method is the most efficient in terms of time and memory. The implicit `while (<>)` form is equivalent in speed to the corresponding explicit code:

```
open my ($fh), '<', $file or die;

while (<$fh>) {
  # do something with $_
}
```

```
while ( defined( my $line = <$fh> ) ) { # explicit version
    # do something with $line
}
```

Note the use of the `defined` operator in the second loop. This prevents the loop from missing a line if the very last line of a file is the single character 0 with no terminating newline—not a likely occurrence, but it doesn't hurt to be careful.

You can use a similar syntax with a `foreach` loop to read the entire file into memory in a single operation:

```
foreach (<$fh>) {
  # do something with $_
}
```

The all-at-once method is slower and uses more memory than the line-at-a-time method. If all you want to do is step through the lines in a file, you should use the line-at-a-time method, although the difference in performance will not be noticeable if you are reading a short file.

All-at-once has its advantages, though, when combined with operations like sorting:

```
print sort <$fh>;  # print lines sorted
```

If you need access to more than one line at a time, all-at-once may be appropriate. If you want to look at previous or succeeding lines based on the current line, you want to already have those lines. This example prints three adjacent lines when it finds a line with "Shazam":

```
my @f = <$fh>;
foreach ( 0 .. $#f ) {
  if ( $f[$_] =~ /\bShazam\b/ ) {
    my $lo = ( $_ > 0 )   ? $_ - 1 : $_;
    my $hi = ( $_ < $#f ) ? $_ + 1 : $_;
    print map { "$_: $f[$_]" } $lo .. $hi;
  }
}
```

You can still handle many of these situations with line-at-a-time input, although your code will definitely be more complex:

```
my @fh;
@f[ 0 .. 2 ] = ("\n") x 3;
```

```
for ( ; ; ) {
  # queue using a slice assignment
  @f[ 0 .. 2 ] = ( @f[ 1, 2 ], scalar(<$fh>) );
  last if not defined $f[1];
  if ( $f[1] =~ /\bShazam\b/ ) {  # ... looking for Shazam
    print map { ( $_ + $. - 1 ) . ": $f[$_]" } 0 .. 2;
  }
}
```

Maintaining a queue of lines of text with slice assignments makes this slower than the equivalent all-at-once code, but this technique works for arbitrarily large input. The queue could also be implemented with an index variable rather than a slice assignment, which would result in more complex but faster running code.

Slurp a file

If your goal is simply to read a file into memory as quickly as possible, you might consider clearing the line separator character and reading the entire file as a single string. This will read the contents of a file or stream much faster than either of the earlier alternatives:

```
my $contents = do {
  local $/;
  open my ($fh1), '<', $file1 or die;
  <$fh1>;
};
```

You can also just use the `File::Slurp` module to do it for you, which lets you read the entire file into a scalar to have it in one big chunk or read it into an array to have it line-by-line:

```
use File::Slurp;

my $text  = read_file('filename');
my @lines = read_file('filename');
```

Use `read` or `sysread` for maximum speed

Finally, the `read` and `sysread` operators are useful for quickly scanning a file if line boundaries are of no importance:

```
open my ($fh1), '<', $file1 or die;
open my ($fh2), '<', $file2 or die;

my $chunk = 4096;  # block size to read
my ( $bytes, $buf1, $buf2, $diff );

CHUNK: while ( $bytes = sysread $fh1, $buf1, $chunk ) {
  sysread $fh2, $buf2, $chunk;
  $diff++, last CHUNK if $buf1 ne $buf2;
}

print "$file1 and $file2 differ" if $diff;
```

Things to remember

- Avoid reading entire files into memory if you don't need to.
- Read entire files quickly with `File::Slurp`.
- Use `read` of `sysread` to quickly read through a file.

Item 54. Open filehandles to and from strings.

Since Perl 5.6, you can open filehandles on strings. You don't have to treat strings any differently from files, sockets, or pipes. Once you stop treating strings specially, you have a lot more flexibility about how you get and send data. Reduce the complexity of your application by reducing the number of cases it has to handle.

And this change is not just for you. Though you may not have thought that opening filehandles on strings was a feature, it is. People tend to want to interact with your code in ways that you don't expect.

Read from a string

If you have a multiline string to process, don't reach for a regex to break it into lines. You can open a filehandle on a reference to a scalar, and then read from it as you would any other filehandle:

```
my $string = <<'MULTILINE';
Buster
Mimi
```

```
Roscoe
MULTILINE

open my ($str_fh), '<', \$string;

my @end_in_vowels = grep /[aeiou]$/, <$str_fh>;
```

Later, suppose you decide that you don't want to get the data from a string that's in the source code, but you want to read from a file instead. That's not a problem, because you are already set up to deal with filehandles:

```
my @end_in_vowels = grep /[aeiou]$/, <$other_fh>;
```

It gets even easier when you wrap your output operations in a subroutine. That subroutine doesn't care where the data come from as long as it can read from the filehandle it gets:

```
my @matches = ends_in_vowel($str_fh);
push @matches, ends_in_vowel($file_fh);
push @matches, ends_in_vowel($socket);

sub ends_in_vowel {
  my ($fh) = @_;

  grep /[aeiou]$/, <$fh>;
}
```

Write to a string

You can build up a string with a filehandle, too. Instead of opening the string for reading, you open it for writing:

```
my $string = q{};

open my ($str_fh), '>', \$string;

print $str_fh "This goes into the string\n";
```

Likewise, you can append to a string that already exists:

```
my $string = q{};

open my ($str_fh), '>>', \$string;

print $str_fh "This goes at the end of the string\n";
```

You can shorten that a bit by declaring `$string` at the same time that you take a reference to it. It looks odd at first, but it works:

```
open my ($str_fh), '>>', \my $string;

print $str_fh "This goes at the end of the string\n";
```

This is especially handy when you have a subroutine or method that normally expects to print to a filehandle, although you want to capture that output in memory. Instead of creating a new file only to read it back into your program, you just capture it directly.

`seek` and `tell`

Once you have a filehandle to a string, you can do all the usual filehandle sorts of things, including moving around in this "virtual file." Open a string for reading, move to a location, and read a certain number of bytes. This can be really handy when you have an image file or other binary (non–line-oriented) format you want to work with:

```
use Fcntl qw(:seek);  # for the constants
my $string = 'abcdefghijklmnopqrstuvwxyz';

my $buffer;
open my ($str_fh), '<', \$string;

seek( $str_fh, 10, SEEK_SET ); # move ten bytes from start
my $read = read( $str_fh, $buffer, 4 );
print "I read [$buffer]\n";
print "Now I am at position ", tell($str_fh), "\n";

seek( $str_fh, -7, SEEK_CUR );  # move seven bytes back
my $read = read( $str_fh, $buffer, 4 );
print "I read [$buffer]\n";
print "Now I am at position ", tell($str_fh), "\n";
```

The output shows that you are able to move forward and backward in the string:

```
I read [klmn]
Now I am at position 14
I read [hijk]
Now I am at position 11
```

You can even replace parts of the string if you open the filehandle as read-write, using +< as the mode:

```
use Fcntl qw(:seek);  # for the constants
my $string = 'abcdefghijklmnopqrstuvwxyz';

my $buffer;
open my ($str_fh), '+<', \$string;

# move 10 bytes from the start
seek( $str_fh, 10, SEEK_CUR );
print $str_fh '***';
print "String is now:\n\t$string\n";

read( $str_fh, $buffer, 3 );
print "I read [$buffer], and am now at ",
  tell($str_fh), "\n";
```

The output shows that you've changed the string, but can also read from it:

```
String is now:
    abcdefghij***nopqrstuvwxyz
I read [nop], and am now at 16
```

You could do this with `substr`, but then you'd limit yourself to working with strings. When you do it with filehandles, you can handle quite a bit more.

Things to remember

- Treat strings as files to avoid special cases.
- Create readable filehandles to strings to break strings into lines.
- Create writeable filehandles to strings to capture output.

Item 55. Make flexible output.

When you use hard-coded (or assumed) filehandles in your code, you limit your program and frustrate your users. Some culprits look like these:

```
print "This goes to standard output\n";
print STDOUT "This goes to standard output too\n";
print STDERR "This goes to standard error\n";
```

When you put those sorts of statements in your program, you reduce the flexibility of the code, causing people to perform acrobatics and feats of magic to work around it. They shouldn't have to localize any filehandles or redefine standard filehandles to change where the output goes. Despite that, people still code like that because it's quick, it's easy, and mostly, they don't know how easy it is to do it better.

You don't need an object-oriented design to make this work, but it's a lot easier that way. When you need to output something in a method, get the output filehandle from the object. In this example, you call `get_output_fh` to fetch the destination for your data:

```
sub output_method {
  my ( $self, @args ) = @_;

  my $output_fh = $self->get_output_fh;

  print $output_fh @args;
}
```

To make that work, you need a way to set the output filehandle. That can be a set of regular accessor methods. `get_output_fh` returns `STDOUT` if you haven't set anything:

```
sub get_output_fh {
  my ($self) = @_;

  return $self->{output_fh} || *STDOUT{IO};
}

sub set_output_fh {
  my ( $self, $fh ) = @_   ;

  $self->{output_fh} = $fh;
}
```

With this as part of the published interface for your code, the other programmers have quite a bit of flexibility when they want to change how your program outputs data:

```
$obj->output_method("Hello stdout!\n");

# capture the output in a string
open my ($str_fh), '>', \$string;
$obj->set_output_fh($str_fh);
$obj->output_method("Hello string!\n");

# send the data over the network
socket( my ($socket), ... );
$obj->set_output_fh($socket);
$obj->output_method("Hello socket!\n");

# output to a string and STDOUT at the same time
use IO::Tee;
my $tee =
  IO::Tee->new( $str_fh, *STDOUT{IO} );
$obj->set_output_fh($tee);
$obj->output_method("Hello all of you!\n");

# send the data nowhere
use IO::Null;
my $null_fh = IO::Null->new;
$obj->set_output_fh($null_fh);
$obj->output_method("Hello? Anyone there?\n");

# decide at run time: interactive sessions use stdout,
# non-interactive session use a null filehandle
use IO::Interactive;
$obj->set_output_fh( interactive() );
$obj->output_method("Hello, maybe!\n");
```

It gets even better, though. You almost get some features for free. Do you want to have another method that returns the output as a string? You've already done most of the work! You just have to shuffle some filehandles around as you temporarily make a filehandle to a string (Item 54) as the output filehandle:

```
sub as_string {
  my ( $self, @args ) = @_;

  my $string = '';
  open my ($str_fh), '>', \$string;
```

```
    my $old_fh = $self->get_output_fh;
    $self->set_output_fh($str_fh);
    $self->output_method(@args);

    # restore the previous fh
    $self->set_output_fh($old_fh);

    $string;
}
```

If you want to have a feature to turn off all output, that's almost trivial now. You just use a null filehandle to suppress all output:

```
$obj->set_output_fh( IO::Null->new )
  if $config->{be_quiet};
```

Things to remember

- For flexibility, don't hard-code your filehandles.
- Give other programmers a way to change the output filehandle.
- Use `IO::Interactive` to check if someone will see your output.

Item 56. Use `File::Spec` or `Path::Class` to work with paths.

Perl runs on a couple hundred different platforms, and it's almost a law of software engineering that any useful program that you write will migrate from the system you most prefer to the system you least prefer. If you have to work with file paths, use one of the modules that handle all of the portability details for you. Not only is it safer, it's also easier.

Use `File::Spec` for portability

The `File::Spec` module comes with Perl, and the most convenient way to use it is through its function interface. It automatically imports several subroutines into the current namespace:

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

To construct a new path, you need the volume (maybe), the directory, and the filename. The volume and filename are easy:

```
my $volume = 'C:';
my $file   = 'perl.exe';
```

You have to do a bit of work to create the directory from its parts, but that's not so bad. The `rootdir` function gets you started, and the `catdir` puts everything together according to the local system:

```
my $directory =
  catdir( rootdir(), qw(strawberry perl bin) );
```

If you are used to Windows or UNIX, you may not appreciate that some systems, such as VMS, format the directory portion of the path the same as the filename portion. If you use `File::Spec`, however, you don't have to worry too much about that.

Now that you have all three parts, you can put them together with `catpath`:

```
my $full_path =
  catpath( $volume, $directory, $file );
```

On UNIX-like filesystems, `catpath` ignores the argument for the volume, so if you don't care about that portion, you can use `undef` as a placeholder:

```
my $full_path =
  catpath( undef, $directory, $file );
```

This might seem like a silly way to do that if you think that your program will ever run only on your local system. If you don't want to handle the portable paths, just don't tell anyone about your useful program, so you'll never have to migrate it.

`File::Spec` has many other functions that deal with putting together and taking apart paths, as well as getting the local representations to common paths such as the parent directory, the temporary directory, the devnull device, and so on.

Use `Path::Class` if you can

The `Path::Class` module is a wrapper around `File::Spec` and provides convenience methods for things that are terribly annoying to work out yourself. To start, you construct a file or a directory object. On Windows, you just give `file` your Windows path, and it figures it out. The `file` function assumes that the path is for the local filesystem:

```
use Path::Class qw(file dir);

my $file = file('C:/strawberry/perl/bin/perl.exe');
```

This path doesn't have to exist. The object in `$file` doesn't do anything to verify that the path is valid; it just deals with the rules for constructing paths on the local system.

If you aren't on Windows but still need to work with a Windows path, you use `foreign_file` instead:

```
my $file = foreign_file( 'Win32',
  'C:/strawberry/perl/bin/perl.exe' );
```

Now `$file` does everything correctly for a Windows path. If you need to go the other way and translate it into a path suitable for another system, you can use the `as_foreign` method:

```
# /strawberry/perl
my $unix_path = $file->as_foreign('Unix');
```

Once you have the object, you call methods to interact with the file.

To get a filehandle for reading, call `open` with no arguments. It's really just a wrapper around `IO::File`, so it's just like calling `IO::File->new`:

```
my $read_fh = $file->open
  or die "Could not open $file: $!";
```

If you want to create a new file, you start with a `file` object. That doesn't create the file, since the object simply deals with paths. When you call `open` and pass it the `>`, the file is created for you and you get back a write filehandle:

```
my $file = file('new_file');

my $fh = $file->open('>');

print $fh "Put this line in the file\n";
```

You can get the directory that contains the file, and then open a directory handle:

```
my $dir = $file->dir;
my $dh = $dir->open or die "Could not open $dir: $!";
```

If you already have a directory object, it's easy to get its parent directory:

```
my $parent = $dir->parent;
```

You read from the directory handle with `readdir`, as normal, and get the name of the file. As with any `readdir` operation, you get only the filename, so you have to add the directory portion yourself. That's not a problem when you use `file` to put it together for you:

```
while ( my $filename = readdir($dh) ) {
  next if $filename =~ /^\.\.?$/;
  my $file = file( $dir, $file );
  print "Found $file\n";
}
```

Things to remember

- Don't hard-code file paths with operating system specific details.
- Use `File::Spec` or `Path::Class` to construct portable paths.

Item 57. Leave most of the data on disk to save memory.

Datasets today can be huge. Whether you are sequencing DNA or parsing weblogs, the amount of data that is collected can easily surpass the amount of data that can be contained in the memory of your program. It is not uncommon for Perl programmers who work with large data sets to see the dreaded "Out of memory!" error.

When this happens, there are a few things you can do. One idea is to check how much memory your process can use. The fix might be as simple as having your operating system allocate more memory to the program.

Increasing memory limits is really only a bandage for larger algorithmic problems. If the data you are working with can grow, you're bound to hit memory limits again.

There are a few strategies that you can use to reduce the memory footprint of your program.

Read files line-by-line

The first and most obvious strategy is to read the data you are processing line-by-line instead of loading entire data sets into memory. You could read an entire file into an array:

```
open my ($fh), '<', $file or die;
my @lines = <$fh>;
```

However, if you don't need all of the data at once, read only as much as you need for the next operation:

```
open my ($fh), '<', $file or die;

while (<$fh>) {
  #... do something with the line
}
```

Store large hashes in DBM files

There is a common pattern of problem in which you have some huge data set that you have to cycle through while looking up values keyed in another potentially large data set. For instance, you might have a lookup file of names by ID and a log file of IDs and times when that ID logged in to your system. If the set of lookup data is sufficiently large, it might be wise to load it into a hash that is backed by a DBM file. This keeps the lookups on the filesystem, freeing up memory. In the `build_lookup` subroutine in the example below, it looks like you have all of the data in memory, but you've actually stored it in a file connected to a tied hash:

```
use Fcntl;  # For O_RDWR, O_CREAT, etc.

my ( $lookup_file, $data_file ) = @ARGV;

my $lookup = build_lookup($lookup_file);

open my ($data_fh), '<', $data_file or die;

while (<$data_fh>) {
  chomp;
  my @row = split;
```

```
  if ( exists $lookup->{ $row[0] } ) {
    print "@row\n";
  }
}

sub build_lookup {
  my ($file) = @_;
  open my ($lookup_fh), '<', $lookup or die;

  require SDBM_File;
  tie( my %lookup, 'SDBM_File', "lookup.$$",
    O_RDWR | O_CREAT, 0666 )
    or die
    "Couldn't tie SDBM file 'filename': $!; aborting";

  while ($lookup_fh) {
    chomp;
    my ( $key, $value ) = split;
    $lookup{$key} = $value;
  }

  return \%lookup;
}
```

Building the lookup can be costly, so you want to minimize the number of times that you have to do it. If possible, prebuild the lookup DBM file and just load it at run time. Once you have it, you shouldn't have to rebuild it. You can even share it between programs.

`SDBM_File` is a Perl implementation of DBM that doesn't scale very well. If you have `NDBM_File` or `GDBM_File` available on your system, opt for those instead.

Read files as if they were arrays

If key-based lookup by way of a hash isn't flexible enough, you can use `Tie::File` to treat a file's lines as an array, even though you don't have them in memory. You can navigate the file as if it were a normal array. You can access any line in the file at any time, like in this random fortune printing program:

```
use Tie::File;

tie my @fortunes, 'Tie::File', $fortune_file
  or die "Unable to tie $fortune_file";

foreach ( 1 .. 10 ) {
  print $fortunes[ rand @fortunes ];
}
```

Use temporary files and directories

If these prebuilt solutions don't work for you, you can always write temporary files yourself. The `File::Temp` module helps by automatically creating a unique temporary file name and by cleaning up the file after you are done with it. This can be especially handy if you need to completely create a new version of a file, but replace it only once you're done creating it:

```
use File::Temp qw(tempfile);

my ( $fh, $file_name ) = tempfile();

while (<>) {
  print {$fh} uc $_;
}

$fh->close;

rename $file_name => $final_name;
```

`File::Temp` can even create a temporary directory that you can use to store multiple files in. You can fetch several Web pages and store them for later processing:

```
use File::Temp qw(tempdir);
use File::Spec::Functions;
use LWP::Simple qw(getstore);

my ($temp_dir) = tempdir( CLEANUP => 1 );

my %searches = (
  google    => 'http://www.google.com/#hl=en&q=perl',
```

```
  yahoo     => 'http://search.yahoo.com/search?p=perl',
  microsoft => 'http://www.bing.com/search?q=perl',
);

foreach my $search ( keys %searches ) {
  getstore( $searches{$search},
    catfile( $temp_dir, $search ) ) );
}
```

There's one caution with `File::Temp`: it opens its files in binary mode. If you need line-ending translations or a different encoding (Item 73), you have the use `binmode` on the filehandle yourself.

Things to remember

- Store large hashes on disk in DBM files to save memory.
- Treat files as arrays with `Tie::File`.
- Use `File::Temp` to create temporary files and directories.

This page intentionally left blank

Index

Symbols

A

B

C

D

E

F

G

H

I

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z