

Effective SOFTWARE DEVELOPMENT SERIES

Scott Meyers, Consulting Editor



Effective PERL PROGRAMMING

Ways to Write Better, More Idiomatic Perl

SECOND EDITION



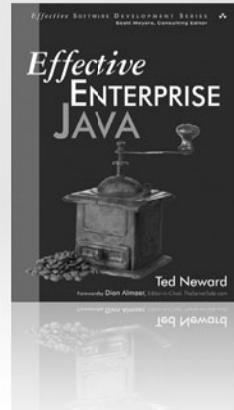
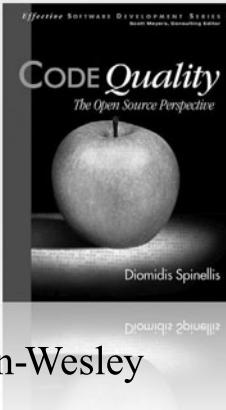
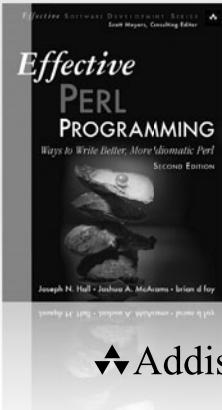
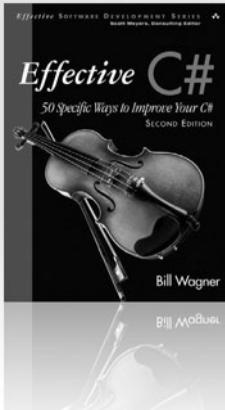
Joseph N. Hall • Joshua A. McAdams • brian d foy

Effective Perl Programming

Second Edition

The Effective Software Development Series

Scott Meyers, Consulting Editor



▼ Addison-Wesley

Visit informat.com/esds for a complete list of available publications.

The Effective Software Development Series provides expert advice on all aspects of modern software development. Books in the series are well written, technically sound, and of lasting value. Each describes the critical things experts always do—or always avoid—to produce outstanding software.

Scott Meyers, author of the best-selling books *Effective C++* (now in its third edition), *More Effective C++*, and *Effective STL* (all available in both print and electronic versions), conceived of the series and acts as its consulting editor. Authors in the series work with Meyers to create essential reading in a format that is familiar and accessible for software developers of every stripe.

PEARSON

▼ Addison-Wesley

Cisco Press

EXAM/CRAM

IBM
Press

QUE®

PRENTICE HALL

SAMS

Safari®
Books Online

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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informat.com/aw

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

Copyright © 2010 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

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

Foreword	xi
Preface	xiii
Acknowledgments	xvii
About the Authors	xix
Introduction	1
Chapter 1 The Basics of Perl	9
Chapter 2 Idiomatic Perl	51
Chapter 3 Regular Expressions	99
Chapter 4 Subroutines	145
Chapter 5 Files and Filehandles	179
Chapter 6 References	201
Chapter 7 CPAN	227
Chapter 8 Unicode	253
Chapter 9 Distributions	275
Chapter 10 Testing	307
Chapter 11 Warnings	357
Chapter 12 Databases	377
Chapter 13 Miscellany	391
Appendix A Perl Resources	435
Appendix B Map from First to Second Edition	439
Index	445

This page intentionally left blank

Contents

Foreword	xi	
Preface	xiii	
Acknowledgments	xvii	
About the Authors	xix	
Introduction	1	
Chapter 1	The Basics of Perl	9
Item 1. Find the documentation for Perl and its modules.	9	
Item 2. Enable new Perl features when you need them.	12	
Item 3. Enable strictures to promote better coding.	14	
Item 4. Understand what sigils are telling you.	17	
Item 5. Know your variable namespaces.	19	
Item 6. Know the difference between string and numeric comparisons.	21	
Item 7. Know which values are false and test them accordingly.	23	
Item 8. Understand conversions between strings and numbers.	27	
Item 9. Know the difference between lists and arrays.	31	
Item 10. Don't assign <code>undef</code> when you want an empty array.	34	
Item 11. Avoid a slice when you want an element.	37	
Item 12. Understand context and how it affects operations.	41	
Item 13. Use arrays or hashes to group data.	45	
Item 14. Handle big numbers with <code>bignum</code> .	47	
Chapter 2	Idiomatic Perl	51
Item 15. Use <code>\$_</code> for elegance and brevity.	53	
Item 16. Know Perl's other default arguments.	56	
Item 17. Know common shorthand and syntax quirks.	60	
Item 18. Avoid excessive punctuation.	66	
Item 19. Format lists for easy maintenance.	68	
Item 20. Use <code>foreach</code> , <code>map</code> , and <code>grep</code> as appropriate.	70	
Item 21. Know the different ways to quote strings.	73	
Item 22. Learn the myriad ways of sorting.	77	
Item 23. Make work easier with smart matching.	84	
Item 24. Use <code>given-when</code> to make a switch statement.	86	
Item 25. Use <code>do {}</code> to create inline subroutines.	90	

Item 26. Use <code>List::Util</code> and <code>List::MoreUtils</code> for easy list manipulation.	92
Item 27. Use <code>autodie</code> to simplify error handling.	96
Chapter 3 Regular Expressions	99
Item 28. Know the precedence of regular expression operators.	99
Item 29. Use regular expression captures.	103
Item 30. Use more precise whitespace character classes.	110
Item 31. Use named captures to label matches.	114
Item 32. Use noncapturing parentheses when you need only grouping.	116
Item 33. Watch out for the match variables.	117
Item 34. Avoid greed when parsimony is best.	119
Item 35. Use zero-width assertions to match positions in a string.	121
Item 36. Avoid using regular expressions for simple string operations.	125
Item 37. Make regular expressions readable.	129
Item 38. Avoid unnecessary backtracking.	132
Item 39. Compile regexes only once.	137
Item 40. Pre-compile regular expressions.	138
Item 41. Benchmark your regular expressions.	139
Item 42. Don't reinvent the regex.	142
Chapter 4 Subroutines	145
Item 43. Understand the difference between <code>my</code> and <code>local</code> .	145
Item 44. Avoid using <code>@_</code> directly unless you have to.	154
Item 45. Use <code>wantarray</code> to write subroutines returning lists.	157
Item 46. Pass references instead of copies.	160
Item 47. Use hashes to pass named parameters.	164
Item 48. Use prototypes to get special argument parsing.	168
Item 49. Create closures to lock in data.	171
Item 50. Create new subroutines with subroutines.	176
Chapter 5 Files and Filehandles	179
Item 51. Don't ignore the file test operators.	179
Item 52. Always use the three-argument <code>open</code> .	182
Item 53. Consider different ways of reading from a stream.	183
Item 54. Open filehandles to and from strings.	186
Item 55. Make flexible output.	189
Item 56. Use <code>File::Spec</code> or <code>Path::Class</code> to work with paths.	192
Item 57. Leave most of the data on disk to save memory.	195
Chapter 6 References	201
Item 58. Understand references and reference syntax.	201
Item 59. Compare reference types to prototypes.	209

Item 60. Create arrays of arrays with references.	211
Item 61. Don't confuse anonymous arrays with list literals.	214
Item 62. Build C-style <code>structs</code> with anonymous hashes.	216
Item 63. Be careful with circular data structures.	218
Item 64. Use <code>map</code> and <code>grep</code> to manipulate complex data structures.	221
Chapter 7 CPAN	227
Item 65. Install CPAN modules without admin privileges.	228
Item 66. Carry a CPAN with you.	231
Item 67. Mitigate the risk of public code.	235
Item 68. Research modules before you install them.	239
Item 69. Ensure that Perl can find your modules.	242
Item 70. Contribute to CPAN.	246
Item 71. Know the commonly used modules.	250
Chapter 8 Unicode	253
Item 72. Use Unicode in your source code.	254
Item 73. Tell Perl which encoding to use.	257
Item 74. Specify Unicode characters by code point or name.	258
Item 75. Convert octet strings to character strings.	261
Item 76. Match Unicode characters and properties.	265
Item 77. Work with graphemes instead of characters.	269
Item 78. Be careful with Unicode in your databases.	272
Chapter 9 Distributions	275
Item 79. Use <code>Module::Build</code> as your distribution builder.	275
Item 80. Don't start distributions by hand.	278
Item 81. Choose a good module name.	283
Item 82. Embed your documentation with Pod.	287
Item 83. Limit your distributions to the right platforms.	292
Item 84. Check your Pod.	295
Item 85. Inline code for other languages.	298
Item 86. Use XS for low-level interfaces and speed.	301
Chapter 10 Testing	307
Item 87. Use <code>prove</code> for flexible test runs.	308
Item 88. Run tests only when they make sense.	311
Item 89. Use dependency injection to avoid special test logic.	314
Item 90. Don't require more than you need to use in your methods.	317
Item 91. Write programs as modulinos for easy testing.	320
Item 92. Mock objects and interfaces to focus tests.	324
Item 93. Use SQLite to create test databases.	330
Item 94. Use <code>Test::Class</code> for more structured testing.	332

Item 95. Start testing at the beginning of your project.	335
Item 96. Measure your test coverage.	342
Item 97. Use CPAN Testers as your QA team.	346
Item 98. Set up a continuous build system.	348
Chapter 11 Warnings	357
Item 99. Enable warnings to let Perl spot suspicious code.	358
Item 100. Use lexical warnings to selectively turn on or off complaints.	361
Item 101. Use <code>die</code> to generate exceptions.	364
Item 102. Use Carp to get stack traces.	366
Item 103. Handle exceptions properly.	370
Item 104. Track dangerous data with taint checking.	372
Item 105. Start with taint warnings for legacy code.	375
Chapter 12 Databases	377
Item 106. Prepare your SQL statements to reuse work and save time.	377
Item 107. Use SQL placeholders for automatic value quoting.	382
Item 108. Bind return columns for faster access to data.	384
Item 109. Reuse database connections.	386
Chapter 13 Miscellany	391
Item 110. Compile and install your own perl.	391
Item 111. Use <code>Perl::Tidy</code> to beautify code.	394
Item 112. Use Perl Critic.	398
Item 113. Use <code>Log::Log4perl</code> to record your program's state.	403
Item 114. Know when arrays are modified in a loop.	410
Item 115. Don't use regular expressions for comma-separated values.	412
Item 116. Use unpack to process columnar data.	414
Item 117. Use pack and unpack for data munging.	416
Item 118. Access the symbol table with typeglobs.	423
Item 119. Initialize with BEGIN; finish with END.	425
Item 120. Use Perl one-liners to create mini programs.	428
Appendix A Perl Resources	435
Appendix B Map from First to Second Edition	439
Books	435
Websites	436
Blogs and Podcasts	437
Getting Help	437
Index	445

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 Sal-loum, 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 Piszcza 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, co-ordinated 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.

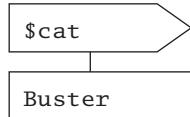
An **lvalue** (pronounced “ell value”) is a value that can appear on the left-hand side of an assignment statement. This is the customary meaning of the term; however, there are some unusual constructs that act as lvalues in Perl, such as the `substr` operator.

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.

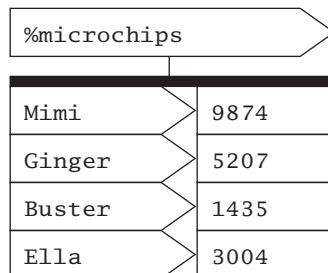
Variables are values with names. The name appears in a sideways “picket” above the value. A scalar value is represented with a single rectangular box:



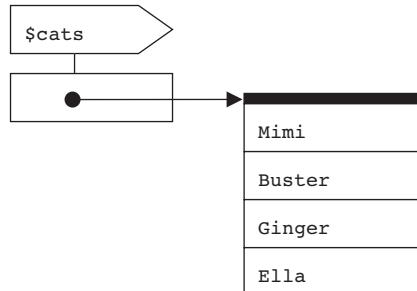
Arrays and lists have a similar graphical representation. Values are shown in a stack with a thick bar on top:



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 *perfunc*, 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` or `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:
abcdefghijklmnopqrstuvwxyz
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, $filename );
    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

& (ampersand), sigil for subroutines, 17
&&, and operator used in place of, 67–68
\$ anchor, 124
\$ (dollar sign), sigil for scalars, 17
\$_ (dollar underscore)
 built-ins that use, 58
 as default for many operations, 53–54
 localizing, 55
 main package and, 54
 -p switch and, 430
 programming style and, 55–56
\$' match variable, 117
\$` match variable, 117
\$. special variable, 429–430
\$& match variable, 117
\$1, \$2, \$3 capture variables, 103–105
\$@, for handling exceptions, 365, 370–371
\$dbh, per-class database connections, 389
\$^C variable, 426
" " (double quotes), options for quote strings, 73–74
%+ hash, labeled capture results in, 114–115

% (percent symbol), sigil for hashes, 17
() [], forcing list context, 60
() (parentheses). *See* Parentheses ()
* (asterisk), sigil for typeglobs, 17
' ' (single quotes), options for quote strings, 73
; (semicolon), for handling exceptions, 371
?
 for nongreedy quantifiers, 120
 using SQL placeholders, 383
?: (capture-free parenthesis), 116–117
@ (at), sigil for arrays
 for list of elements, 17
 overview of, 9
 @_
 as default argument, 56–57
 passing arguments, 154–157
@[{}], for making copies of lists, 64
@ specifier, parsing column data, 414–416
@ARGV
 as default argument, 57–58
 as default argument outside subroutines, 429
@INC, module search path, 428
[] (square brackets)
 anonymous arrays constructor, 60
 careful use of, 63

\ (reference operator), creating list of references, 61

^ anchor, matching beginning of a string with, 123–124

{ } (curly braces), careful use of, 63–64

||= operator, Orcish Maneuver using, 82

||, or operator used in place of, 67–68

~~ (smart match) operation. *See* smart match operation (~~)

<> (diamond) operator
careful use of, 63
for line-input operator, 24, 183

<=> (spaceship) operator, 83

== (equality) operator, 22

=> (fat arrow) operator
making key-value pairs, 61–62
for simulating named parameters, 62

_ (underscore), virtual file handler, 181

A

\A anchor, matching beginning of a string with, 123–124

Actions, `Module::Build`, 277

Additive operators, 42

Admin privileges, for installing CPAN modules, 228

Agile methodologies, 335

Aliases
for characters, 260
for typeglobs, 424

All-at-Once method, for reading from streams, 184

Alternation operators

avoiding backtracking in regular expressions, 132–133
character class used in place of, 134
precedence of, 100–101

Anchors

matching beginning with ^ or \A, 123–124
regular expressions and, 121
setting word boundaries with \b, 121–123

and operator, 67–68

Angle brackets, diamond operator (<>)
careful use of, 63
for line-input operator, 24

AnnoCPANn, 12

Anonymous arrays constructors, 214–215

Anonymous closures, 173

Apache Ant, 351–352

`Apache::DBI` module, 389

API, names to avoid, 287

`App` namespace, 285

`App::Ack` module, 250

Appenders, in `Log::Log4perl` configuration, 406–409

Arguments

default. *See* Default arguments
passing to subroutines, 160–162
passing with @_ , 56–57, 154–157
passwith with @ARGV. *See* @ARGV
returning from subroutines, 162

Arithmetic expressions, 99

Arithmetic operators, 42

Arrays

- \$, for retrieving scalar values, 9
- @, for retrieving lists of values, 17
- anonymous array constructors, 214–215
- avoiding slices when you want an element, 37–38
- creating arrays of arrays, 211–213
- creating prototypes, 168
- end of, 25
- for grouping data, 45–47
- knowing when loops are modifying, 410–412
- vs. lists, 31–32
- merging, 96
- namespace for, 19
- not assigning `undef` for empty arrays, 34–37
- not confusing slices with elements, 39–40
- reading files as if they were arrays, 197–198
- removing and returning elements from end of, 169–170
- slices for sorting, 40
- swapping values in, 60

ASCII

- telling Perl which encoding system to use, 257
- using non-ASCII characters for identifiers, 255

Assignment operators

- in list and scalar context, 43–44
- not assigning `undef` for empty arrays, 327
- redefining subroutines by assigning to their typeglobs, 34–37
- swapping values using list assignments, 60

`atof()`, for converting strings to numbers, 27

Atoms, 99–100

Author tests, 311–312

`autodie`, exception handling with, 96–98, 366

Automated environment, skipping tests in, 313–314

Automatic value quoting, SQL placeholders for, 382–384

Autovivification, of references, 207–208

B

`\b` and `\B`, matching boundaries between word and nonword characters, 121–123

Backreferences, capture buffers with, 105–106

Backtracking, in regular expressions

- avoiding, 132–133
- character class used in place of alternation, 134
- quantifiers and, 134–136

Barewords, caution in use of, 15–16

Barr, Graham, 228

`BEGIN` blocks, for initialization, 425–426

Benchmarking, in regular expressions, 139–141

Big-endian representation, 419–421

`bignum`, 47–49

Binary strings, 253

`bind_col` method, performance enhanced by, 385–386

binmode, for encoding system on selected filehandles, 258

Birmingham Perl Mongers, 307

blib module, 308

Blogs, Perl, 437

Books, on Perl, 435–436

Boolean context, converting numbers to strings before testing, 23

Branching, 88

Bugs, viewing or reporting, 239–240

Built-ins

- overriding, 328–329
- in Perl, 5
- that do not use `$_`, 59
- that use `$_` (dollar underscore), 58

Bunce, Tim, 227

Bytes. *See* Octets

C

C compiler, compiling `perl`, 392

C language

- Perl compared with, 51
- XS connecting to Perl, 301–302

Caching statement handles, 380–382

Call stacks, tracing using **Carp**, 366–369

Cantrell, David, 237

Capture

- labeling matches, 114–115
- noncapturing parentheses for grouping, 116–117
- parenthesis for, 116

Capture buffers. *See* Capture variables

Capture-free parenthesis `(?:)`, 116–117

Capture variables

capture buffers with backreferences, 105–106

overview of, 103–105

used in substitutions, 107

Carp module

checking enabled warning before triggering, 363

stack traces using, 366–369

Carriage return, in regular expression, 112–113

Catalyst module, 250

CGI::Simple module, 250

Character class, used in place of alternation, 134

Character sets, 254

Character strings, 253, 261–265

Characters

aliases used for, 260

converting octet strings to character strings, 261–265

getting code points from names, 260

getting names from code points, 259–260

matching, 266

metacharacters used as literal characters, 102–103

specifying by code point or name, 258–259

transliterating single, 128

zero-width assertions for matching, 121

charnames, 259

Circular data structures, 218–221

Classes

character class used in place of, 134

faking with `Test::MockObject`, 325–327

- sharing database connections across, 388–389
- subclass** argument of `Module::Build`, 277–278
- `Test::Class` module, 332–335
- Closures**
 - anonymous closures compared with state variable, 173
 - for locking in data, 171
 - private data for subroutines, 172–174
 - for sharing data, 175
- Code**
 - beautifying with `Perl::Tidy`, 394–396
 - reading code underlying CPAN modules, 241
 - spotting suspicious, 358–361
 - taint warnings for legacy, 375–376
- Code points**
 - defining custom properties, 268
 - getting from names, 260
 - getting names from, 259–260
 - specifying characters by, 258–259
 - Unicode mapping characters to, 253
- Collection types, arrays and hashes as, 45**
- Columns**
 - binding for performance, 384–386
 - using `unpack` to parse fixed-width, 414–416
- Combining characters, in Unicode, 264**
- Comma operator**
 - `=>` (fat arrow) operator compared with, 61
 - creating series not lists, 32–33
 - formatting lists for easy maintenance, 68–70
- Comma-separated values (CSVs), 412–414**
- Command-line arguments, decoding before using, 263–264**
- Command line, writing short programs on, 428–434**
 - `-a` and `-F` switches, 431
 - `-e` or `-E` switch, 428–429
 - `-i` switch, 431–432
 - `-M` switch, 432
 - `-n` switch, 429–430
 - overview of, 428
 - `-p` switch, 430–431
- Commands, Pod, 288–289**
- Comments, adding to regular expressions, 129–130**
- Comparison operators, string and numeric comparison, 21–23**
- Comparison (`sort`) subroutines, 8–80**
- Compatibility, cross-platform, 3**
- Compilation**
 - compile time warnings, 358–359
 - compiling regular expression only once, 137–138
 - precompiling regular expressions, 138–139
 - running code with `BEGIN` blocks at, 426–427
 - of your own `perl`, 391–393
- Complaints, selectively toggling using lexical warnings, 361–364**
- Complex behavior, encapsulating using smart match, 84–85**
- Complex data structures, manipulating, 221**

- Complex regular expressions, breaking down into pieces, 130–132
- Composite characters, graphemes and, 269
- Comprehensive Perl Archive Network.
 - See* CPAN (Comprehensive Perl Archive Network)
- confess**, full stack backtrace with, 369
- Configuration, checking Perl’s, 294–295
- Configure script, for compiling **perl**, 392
- Configuring CPAN clients
 - CPANPLUS, 230
 - CPAN.pm, 229
- Connections, database
 - reusing, 386
 - sharing, 387–390
 - too many, 386–387
- Context
 - affect on operations, 41
 - by assignment, 43–44
 - context-sensitive code, 41
 - forcing list context, 60–61
 - matching in list context, 108
 - names providing, 284
 - of numbers and strings, 27, 42
 - of scalars and lists, 42–43
 - of subroutines, 157–159
 - void context, 44
- Contextual::Return** module, 159–160
- Continuous integration and testing, 348–349
- Control structures, 53
- Conway, Damian, 159
- CORE::GLOBAL** namespace, 329
- Coverage testing. *See* Test coverage
- CP-1251 character set (Windows), 254
- CPAN (Comprehensive Perl Archive Network)
 - commonly used modules, 250–252
 - compiling **perl**, 392–393
 - configuring CPANPLUS, 230
 - configuring CPAN.pm, 229
 - considering need for upgrades, 236
 - considering which modules are needed, 235–237
 - contributing to, 246–249
 - CPAN Search, 239–241
 - CPAN Testers, 237
 - CPANTS Kwalitee, 241–242
 - ensuring that Perl can find your modules, 242–243
 - how to use, 2
 - injecting own modules with **CPAN::Mini::Inject**, 233–235
 - installing, 228–229
 - local::lib** module, 230–231
 - mitigating the risk of public code, 235
 - overview of, 227
 - researching modules before using, 237–238
 - setting up MiniCPAN, 232–233
 - updating configurations to use **perl**, 393
 - using MiniCPAN, 233
- CPAN Ratings
 - for module distributions, 240
 - web site interface to CPAN, 228
- CPAN RT issue tracker, 228
- CPAN Search
 - module documentation files, 11

- researching modules before installing, 239–241
 web site interface to CPAN, 228
- CPAN Testers**
 creating custom smoke testers, 348
 developer versions of, 347–348
 as a QA team, 346–347
 setting preferences, 347
 skipping tests in automated environment, 313–314
 testing modules before purchasing, 237
 viewing test results for newly uploaded distribution, 248
 viewing tests run on modules, 240
 web site interface to CPAN, 228
- cpan-upload tool**, 247–248
- CPAN::Mini module**, 232. *See also* MiniCPAN
- CPAN::Mini::Inject module**, 234
- CPANPLUS**
 configuring, 230
 reading PAUSE index files with, 227
- CPAN.pm**
 configuring, 229
 reading PAUSE index files with, 227
- CPANTS Kwalitee**, 241–242
- croak**, using in place of `warn` and `die`, 367–368
- Cross-platform compatibility**, in Perl, 3
- Cruise Control**
 building project in Apache Ant and, 351–352
 as continuous build system, 350
 formatting output for, 353–354
 for integration testing, 351–355
 overview of, 351
 setting up, 354–355
- CSVs (comma-separated values)**, 412–414
- Curly braces ({}), careful use of**, 63–64
- Currying subroutines**, 174
- D**
- Data structures**
 cautions with use of circular, 218–221
 manipulating complex, 221
- Data types, specifying column**, 386
- Database Interface**, 250. *See also* DBI module
- Databases**, 377–390
 automatic value quoting with SQL placeholders, 382–384
 binding return columns for performance, 384–386
 care in using Unicode with, 272–273
 creating test databases, 330–331
 mocking database layer, 325
 preparing multiple statements, 379–382
 reusing connections, 386–390
 reusing work with SQL statements, 377–379
- Date-Time module**
 overview of, 235–236
 mocking, 327
- DBD::Gofers module**, 389–390
- DBI module**
 care in use of Unicode with databases, 272–273
 in list of commonly used modules, 250
- DBI object, mocking**, 325–326

- DBIx::Class** module, 250
- DBM files, storing large hashes in, 196–197
- dclone**, for deep copies of arrays, 65
- DEBUG level, **Log::Log4perl**, 405–407
- Decoding, bytes into character strings, 254
- Default arguments**
 - @ARGV as, 57–58
 - @_ as, 56–57
 - \$_ (dollar underscore) for, 53–54
 - STDIN as, 58–59
 - table of, 59
- defined** operator, 35
- Dependencies**
 - keeping to a minimum, 317
 - managing, 238
- Dependency injection, for ease of testing, 314–317
- Devel::CheckOS** command, 293–294
- Devel::Cover** module, 343–345
- Development, separating from production, 238
- die**
 - generating exceptions with, 364–366
 - handling exceptions with, 371
 - using croak and Carp as alternative to, 366–368
- Directories**
 - Path::Class** module and, 193–195
 - setting up relative directory, 245–246
 - t*/directory for test files, 311
 - temporary, 198–199
 - xt*/ (extra tests) directory for author tests, 312
- Directory handles namespace, 19–20
- Distributed|Decentralized|Dark Perl Archive Network (DPAN), 238
- Distribution::Cooker** module, 282–283
- Distributions**
 - building with **Module::Build**, 275–278
 - checking for Pod coverage, 296–297
 - checking Pod formatting, 295–296
 - CPAN Ratings, 240
 - CPAN Testers, 248
 - creating from custom templates, 282–283
 - embedding documentation using Pod, 287–291
 - extending **Module::Starter**, 280–282
 - inline code for other languages, 298–301
 - limiting to right platforms, 292–295
 - naming modules, 283–287
 - overview of, 275
 - plug-ins for **Module::Starter**, 280
 - spell checking code, 297–298
 - starting with **Module::Starter**, 278–279
 - uploading to PAUSE (Perl Authors Upload Server), 247–248
- do {} syntax and usage**, 90–92
- Documentation**
 - embedding using Pod, 287–291
 - finding, 9–10
 - local, 12
 - Log::Log4perl**, 409
 - online, 11–12
 - Perl::Critic** custom policies, 403
 - perldiag**, 360
 - perldoc** reader, 10–11
 - perlstyle** documentation, 7

Dominus, Mark Jason, 177, 373–374
Double-quote interpolation
 making regular expressions readable, 130–132
 of regular expressions, 101–103
Double quotes (“ ”), options for quote strings, 73–74
DPAN
 (Distributed|Decentralized|Dark Perl Archive Network), 238
Dualvars, creating, 30–31

E

/e . See **Regular Expressions**
easy_init, initializing
 Log::Log4perl, 404–406
Elements
 avoiding slices when you want an element, 37–38
 checking for presence inside hashes, 26
 creating dynamic SQL elements, 384
 finding matches, 94
 iterating read-only over list elements, 70–71
 modifying in lists, 72
 modifying underlying array elements, 410–412
 not confusing slices with elements, 39–40
 selecting in lists, 73
Email module, 250
Encapsulation, of database handle, 388
Encode, for encoding and decoding strings, 262

Encoding
 code points into octets, 253
 setting default, 257–258
 setting on selected filehandles, 258
 telling Perl which encoding system to use, 256
Encoding::FixLatin module, 263
END blocks, running code before program termination with, 427
endianness, dealing with, 419–421
English module, 119
\$ENV{PATH}
 enabling taint checking, 374
 taint warnings for legacy code, 376
\$ENV{AUTOMATED_TESTING}, 314
Environment variables
 \$ENV{PATH}, 374, 376
 \$ENV{AUTOMATED_TESTING}, 314
 \$ENV{TEST_AUTHOR}, 311–312
 PATH environment variable, 393
 PERLTIDY environment variable, 396
 PERL5LIB, 231
 PERL_MM_OPT, 231
 MODULEBUILDRC, 231
 PERL5OPT, 345
 PERLTIDY, 30
 HARNESS_PERL_SWITCHES, 345
 sourcing from shell script, 250
Env::Sourced module, 250
eq operator, 22
.bashrc file, checking perltidy syntax, 396–397
Error handling
 with autodie, 96–98
 fatal errors, 362–363
 generating exceptions using die, 364–366
 overview of, 370–372

- ERROR level, `Log::Log4perl`, 404–405**
- eval**
 catching `autodie` exceptions with, 97–98
 catching exceptions with, 365
 checking matches that might not interpolate into valid regular expressions, 139
 substitutions using, 107
- eval {}**, for making copies of lists, 64
- eval-if**, 365, 370
- Excel**, Microsoft, 251–252
- Exception handling.** *See Error handling*
- exec argument**, of `prove` command, 310–311
- execute method**
 calling `bind_col` after, 385
 reusing work with SQL statements, 379
- exists operator**, checking to see if key is in a hash, 36
- ExtUtils::Makemaker module**
 coverage testing, 343–344
 forcing Git to run full build and test suite for every commit, 349–350
 modifying include paths, 243
 using Makefiles to build Perl distribution, 275
- F**
- Factories (generators)**
 for constructing dependencies objects, 316–317
- subroutines that create other subroutines, 174
- Fake module**, in testing, 325–326
- Fatal errors**, promoting some warnings to, 362–363
- FATAL level, `Log::Log4perl`**, 404–405
- feature pragma**, 13
- File tests**
 file test operators compared with `stat` operator, 180–181
 finding/reusing, 180–181
 not ignoring file test operators, 179–180
 stacking file test operators, 181
- File::Find::Closures module**, 175
- Filehandles.** *See also Files*
 hard-coded (or assumed), 189
 making output flexible, 189–192
 namespace for, 19–20
 options for working with paths, 192
 reading files from a string, 186–187
`seek` and `tell`, 188–189
 setting encoding system on selected, 258
`STDIN` as default argument, 58
 three-argument `open`, 182–183
 virtual filehandle, `_`, for reusing data from last `stat`, 181
 writing files to a string, 187–188
- Files**
`File::Slurp module`, 185
`File::Spec module`, 192–193
 finding/reusing file tests, 180–181
 making output flexible, 189–192
 maximizing speed in reading, 185–186
 memory saving options, 195

not ignoring file test operators, 179–180
 overview of, 179
`Path::Class` module and, 193–195
 reading file data line-by-line to save memory, 196
 reading files as if they were arrays to save memory, 197–198
 reading files from a string, 186–187
 reading from streams, 183–185
`seek` and `tell`, 188–189
 stacking file test operators, 181
 storing large hashes in DBM files, 196–197
 temporary files and directories for saving memory, 198–199
 three-argument `open`, 182–183
 working with paths, 192
 writing files to a string, 187–188

File::Slurp module, 185

File::Spec module, 192–193

File::Temp module, 198–199

Filled text, Pod, 289–291

Flexibility, of file output, 189–192

foreach

- iterating read-only over each element of a list, 70–71
- modifying elements of a list, 72
- modifying underlying array elements, 410–412

Formats

- consistent, 394–396
- namespace for, 19–20

Formatting outputs, for testing, 353–354

Function calls, simulating named parameters for, 62

Functions

list of built-in functions that use `$_`, 58
 in Perl, 4–5

G

Generators. *See Factories (generators)*

getc function, 59

get_logger, in `Log::Log4perl`, 405

Getopt::Long module, 251

git, pre-commit hooks in, 349

given-when

- as alternative to `if-elsif-else`, 86–87
- handling exceptions with `Try::Tiny`, 371
- intermingling code using, 89
- multiple branching with, 88
- smart matching and, 84, 87–88
- for switch statements, 86

glob, 24

Global variables, 146

Goatse, `=()=`, operator, 33

Graphemes

- overview of, 269–271
- in regular expressions, 271–272

Greed, of regular expressions, 119–121

grep

- finding/reusing file tests, 180–181
- modifying underlying array elements, 410–412
- selecting elements in a list, 73
- selecting references, 224–225

Grouping

- arrays or hashes for grouping data, 45–47
- noncapturing parentheses for, 116–117

H

\h and \H, in matching horizontal whitespace, 111–112

Hard-to-cover code, testing, 345–346

Hash operators, creating prototypes with, 168

Hashes

- checking to see if element is present inside, 26
- creating, 41
- for grouping data, 45–47
- namespace for, 19
- for passing named parameters, 164–168
- simulating C-style structs, 216–218
- storing in DBM files, 196–197
- symbol for, 17

Help, getting Perl, 437

Here doc strings, 76–77

Hex escapes, manipulating, 418–419

Hexadecimals

- converting numbers to/from strings, 27

Perl syntax for, 258–259

Hietaniemi, Jarkko, 227

Higher-order functions, 177

Higher Order Perl (Dominus), 177

Hook::LexWrap module, 328

Horizontal whitespace, 111–112

HTML::Parser module, 251

HTML::TreeBuilder module, 251

Huffman coding, in Perl, 51

I

Identifiers

- inappropriate use of, 16
- in Perl, 5
- using non-ASCII characters for, 255

Idiomatic Perl, 51–98

- avoiding excessive punctuation, 66–68
- careful use of braces ({}), 63–64
- default arguments (\$_), 53–56
- default arguments (@_), 56–57
- default arguments (@ARGV), 57–58
- default arguments (STDIN), 58–59
- error handling with autodie, 96–98
- forcing list context, 60–61
- foreach, map, and grep, 70–73
- format lists for easy maintenance, 68–70
- given-when for switch statements, 86–90
- inline subroutines, 90–92
- list assignments for swapping values, 60
- list manipulation options, 92–96
- making copies of lists, 64–66
- overview of, 51–53
- quote string options, 73–77
- simulating named parameters (=>), 61–63
- smart matching, 84–85
- sorting options, 77–84
- table of default arguments, 59

if-elsif-else statements
 given-when as alternative to, 86
 measuring test coverage, 342–343

Image::Magick module, 251

Include path
 configuring at compile time, 246
 modifying, 243
 searching for module locations,
 242–243

index operator, finding substrings
 with, 126–127

init, in Log::Log4perl
 configuration, 406

Initialization
 with BEGIN blocks, 425–426
`Log::Log4perl`, 404
 using => with initializers, 61

Inline code, for other languages in
 distributions, 298–301

Inline subroutines, 90–92

Inline::C module, 299

Inline::Java module, 299

Installation
 multiple perls, 392–393
 own perl, 391–392

Installing CPAN modules
`CPANPLUS` for, 230
`CPAN.pm` for, 229
 without admin privileges, 228–229

Integration testing, 351–355

interfaces, for focused testing, 324–325

Interpolation
 double-quote, 101–103
 precompiled regular expressions into
 match operator, 139

Inversion of control, 315

ISO-8859 character set (Latin), 254

J

Java, 299

JSON::Any module, 251

K

Key-value pairs, 61–62

Keywords, in Perl, 5

Kobe's Search

module documentation files, 11
 web site interface to CPAN, 228

König, Andreas, 227

L

Legacy code, 375–376

Lexical warnings, 361–364

Lexing strings, 108–110

Libraries

setting up module directories,
 230–231
 using private, 245

Line-by-line method

reading files, 196
 for reading from streams, 184

Line endings

newline and carriage return for,
 112–113
 as whitespace, 111

Line input operator (<>), 24, 183

List operator

`local` and `my` as, 153–154
in Perl, 4

`List::MoreUtil`

iterating over more than one list at a time, 95
for list manipulation, 92
merging arrays, 96

Lists

vs. arrays, 31–32
assignment in list context, 33–34
comma operator creating series not lists, 32
context of, 42–43
creating based on contents of another list, 71–72
forcing list context, 60–61
formatting for easy maintenance, 68–70
iterating over more than one list at a time, 95
iterating read-only over each element of, 70–71
making copies of, 64–66
manipulation options, 92–96
match operator used in list context, 108
modifying elements of, 72
selecting elements in, 73
swapping values using list assignments, 60
`wantarray` for writing subroutines that return, 157–159

`List::Util`

`first` function for finding matches, 94–95
for list manipulation, 92
`reduce` function for summing numbers, 93–94

Little-endian representation, dealing with, 419–421**`local`**

as list operator, 153–154
run time scoping with, 149–151
understanding the difference between `my` and `local`, 145
using on reference arguments, 163–164
when to use, 152–153

Local documentation, 12**Local namespace, 286****Local test databases, 330–331****Localizing**

`$_` (dollar underscore) default argument, 55
filehandles and directory handles with typeglobs, 425
typeglobs, 423–424

Localizing variables, in Perl, 5**`local::lib module`**

setting up module directories, 230–231
using private libraries, 245

Locking in data, closures for, 171

Logging. *See also Log::Log4perl*
defining logging levels, 404–405
detecting logging levels, 409
with print statements, 403–404

`Log::Log4perl`, 404–410

better configuration, 406–409
detecting logging level, 409
getting more information, 409–410
log levels defined by, 404–405
logging levels, 404–405
object-oriented interface, 405
overview of, 404

looping

- foreach loop, 89
- knowing when loops are modifying arrays, 410–412
- from the command line, 429–430
- while loop and, 429, 431

Low-level interfaces, XS language for, 301–306

Lvalues

- in Perl, 5
- slices, 38–39

LWP (libwww-perl), 251

LWP::Simple module, 251

M

/m flag, for turning on multiline mode, 123

main package, \$_ (dollar underscore) and, 54

make, compiling perl, 392

Makefiles, 275

Man pages, in Pod markup language, 291

map

- creating lists based on contents of another list, 71–72
- matching inside, 108
- modifying underlying array elements, 410–412
- nesting references with, 223–224
- slicing references with, 221–223

Mason module, 252

Mastering Perl (foy), 375

Match operator, used in list context, 108

Match variables

- /p flag used with, 119
- helping with text manipulation when computing replacement strings, 118
- performance, 118–119
- speed penalty of, 117–118
- used in substitutions, 107

Matching

- beginning of a string, 123–124
- characters, 121, 266
- end of a string, 124–125
- greedy behavior of regular expressions and, 119–121
- named capture for labeling matches, 114–115
- properties, 267–268

Maximum values, finding, 92–93

Memory saving options

- overview of, 195
- reading file data line-by-line, 196
- reading files as if they were arrays, 197–198
- storing large hashes in DBM files, 196–197
- temporary files and directories, 198–199

Methods

- faking with Test::MockObject, 325–327
- in Perl, 5
- redefining test methods with symbol table, 327–328

Microsoft Excel, 251–252

MinCPAN

- hosting private modules in, 233–235
- setting up, 232–233
- using, 233

Mock objects

- for focused testing, 324–325
- `Test::MockObject`, 325–327

Modular programming environments, 3

`Module::Build`

- building project in Apache Ant, 351–352
- custom actions, 277–278
- end user’s perspective, 276
- module author’s perspective, 276–277
- overview of, 275

Modules

- commonly used, 250–252
- considerations before creating new, 248–249
- considering which modules are needed, 235–237
- distributing. *See Distributions*
- ensuring that Perl can find, 242–243
- fake module in testing, 325–326
- installing CPAN modules, 228–229
- loading from command line, 432
- locating, 9–10
- managing dependencies and versions with DPAN, 238
- naming, 283–287
- quality checklist for, 241–242
- reading code underlying, 241
- researching before using, 237–238
- testing and ratings, 240–241

`Module::Starter`

- extending, 280–282
- plug-ins for, 280
- starting distributions with, 278–279

Modulinos, 320–324

Moose object system, for Perl, 251

Multiline mode, turning on with `/m` flag, 123

my

- copying and naming arguments, 154–155
- lexical (compile-time) scoping with, 146–149
- as list operator, 153–154
- understanding the difference between `my` and `local`, 145
- when to use, 151–152

N

`\N`, for non-newlines, 113

Named parameters

- `=>` operator simulating, 62
- hashes for passing, 164–168

Named subroutines, 172–173

Names

- aliasing with typeglobs, 423
- variable namespaces, 19–20

Naming modules, 283–287

- goals of, 284–285
- names to avoid, 286–287
- naming conventions, 285–286
- overview of, 283–284

Nesting references, with `map`, 223–224

Net namespace, 286–287

Newline

- for line endings, 112–113
- non-newlines, 113–114

`no warnings`, disabling warnings within a scope, 362

Non-newlines, 113–114

Nongreedy repetition operators, 120–121

`--noprofile` switch, 396

Normalizing Unicode strings, 271

Notion, Perl, 6–7

Numbers

- context of, 42
- converting to strings, 27–28
- handling big, 47–49
- knowing and testing false values, 23–25
- knowing difference between string and numeric comparisons, 21–23
- summing, 93–94
- using strings and numbers simultaneously, 28–30

Numeric contexts, 27

Numish strings, 22

O

/o flag, for compiling regular expressions, 138

Object-oriented interface,

`Log::Log4perl`, 405

Objects, sharing database connections across, 388

oct operator, for converting octal and hexadecimal values, 27

Octals

- converting numbers to/from strings, 27
- Perl syntax, 258–259

Octets

- converting octet strings to character strings, 261–265
- defined, 253

Online

- documentation, 11–12
- file checking with Perl Critic, 398

open

- `or` for checking success of, 183
- three-argument, 182–183

open pragma, setting default encoding with, 257–258

Operating systems. *See OSs (operating systems)*

Operators

- in Perl, 4
- precedence of, 100–101
- regular expression, 99–100

or operator

- checking success of `open`, 183
- using instead of `||`, 67–68

Orcish Maneuver (`||` cache), 81–82

OSs (operating systems)

- checking, 293–294
- skipping OS-dependent tests, 313

Overriding Perl built-ins, 328–329

P

/p flag, used with match variables, 119

pack

- computing with `unpack`, 417
- dealing with endianness, 419–421
- how it works, 416–417
- manipulating hex escapes, 418–419
- sorting with, 418
- `uuencoding` with, 421–422

Parentheses ()

- calling subroutines without using, 66–67
- careful use of, 63
- noncapturing parentheses for grouping, 116–117
- regular expression precedence and, 100–101

- Parsing comma-separated values,** 412–414
- PATH environment variable**
 - adding preferred Perl's location to, 393
 - enabling taint checking, 374
- Path::Class module,** 193–195
- Paths**
 - `File::Spec` module, 192–193
 - options for working with, 192
 - `Path::Class` module, 193–195
- PAUSE (Perl Authors Upload Server)**
 - developers versions and, 347–348
 - overview of, 227
 - registering with and uploading distributions to, 247–248
- PDL (Perl Data Language),** 251
- PEGS (PPerl Graphical Structures),** 6–7
- Per-class database connections,** 388–389
- Per-object database connections,** 388
- Performance**
 - binding return columns for, 384–386
 - match variables and, 118–119
- perl**
 - compiling, 391–393
 - compiling your own, 391–392
 - executing tests with, 308
 - installing multiple, 392–393
 - using, 393
- Perl Authors Upload Server.** *See* PAUSE (Perl Authors Upload Server)
- Perl, basics**
 - array slices, 37–38
 - arrays, 25
 - assignment in list context, 33–34
 - avoiding soft references, 16
 - bignum, 47–49
 - caution in using barewords, 15–16
 - comma operator, 32–33
 - context by assignment, 43–44
 - context of numbers and strings, 42
 - context of scalars and lists, 42–43
 - converting between strings and numbers, 27–28
 - creating dualvars, 30–31
 - creating hashes, 41
 - declaring variables, 15
 - enabling features as needed, 12–14
 - enabling strictures for better coding, 14
 - finding documentation and modules for, 9–10
 - grouping data in, 45–47
 - hash values, 26
 - knowing and testing false values, 23–25
 - lists vs. arrays, 31–32
 - local documentation, 12
 - lvalue slices, 38–39
 - online documentation, 11–12
 - overview of, 9
 - `perldoc` reader, 10–11
 - proper use of `undef` values, 34–37
 - sigils, 17–18
 - slices for sorting arrays, 40
 - slices vs. elements, 39–40
 - string vs. numeric comparisons, 21–23
 - understanding context and its affect on operations, 41
 - using strings and numbers simultaneously, 28–30
 - variable namespaces, 19–20
 - void context, 44
- Perl Best Practices (Conway),** 399–400

- Perl Critic**
- on command line, 398–399
 - custom policies, 403
 - overview of, 398
 - in test suite, 401–402
 - on web, 398
- Perl-Critic-Bangs** policies, 403
- Perl-Critic-Moose** policies, 403
- Perl-Critic-More** policies, 403
- Perl-Critic-StricterSubs** policies, 403
- Perl-Critic-Swift** policies, 403
- Perl Data Language (PDL)**, 251
- Perl Graphical Structures (PEGS)**, 6–7
- perl-packrats** mailing list, 227
- perlcritic** program, 398–400
- perldiag** documentation, 360
- perldoc** reader, 10–11
- perlfunc** documentation, 10, 180
- perlrun** documentation, 428
- perlsec** documentation, 375
- Perlstyle** documentation, 7
- Perlsyn**, 84
- Perl::Tidy**
- beautifying code with, 394–395
 - checking syntax, 396–397
 - configuring, 395–396
 - testing for, 397
- PERLTIDY** environment variable, 396
- .perltidyrc** file, 395–396
- perltoctoc**, 10
- perlunicode** documentation, 267
- perluniprops**, 267
- ping** method, sharing database connections, 388
- Placeholders, SQL**
- automatic value quoting, 382–384
 - constructing queries with, 378–379
 - preparing multiple statements, 379–382
- Platforms, limiting distributions to right platform**, 292–295
- Plug-ins, for Module::Starter**, 280
- Pod markup language**
- checking coverage, 296–297
 - checking formatting, 295–296
 - commands, 288–289
 - embedding documentation using Pod, 287–288
 - filled text, 289–291
 - man pages in, 291
 - spell checking code, 297–298
 - verbatim text in, 288
- Podcasts, Perl**, 437
- podchecker** tool, 295
- Pod::Spell**, 297
- POE multitasking and networking framework**, 251
- Poetry mode**, 15
- Policies**
- adding Perl::Critic custom, 403
 - using Perl Critic, 399–400
- Possessive, nonbacktracking quantifiers**, 135–136
- Pre-commit hooks**, 349–350
- Precedence**
- of operators, 100–101
 - using low precedence operators to avoid problems, 67–68
- Precompiling regular expressions**, 138–139

Precomposed version, of Unicode, 269

prefix directory
compiling perl, 392–393
using perl, 393

prepare method, reusing work with SQL statements, 378–379

prepare_cached method, caching statement handles with, 380–381

print statements, logging and, 403–404

Private data
for named subroutines, 172–173
for subroutine references, 173–174

Problem solving, with Perl, 2

Processes, sharing database connections across, 389–390

Production
enabling warnings in, 360–361
separating from development, 238

Profile, creating for single project, 396

--profile switch, 396

Programming style, 55–56

Programs, writing short, 428–434

Projects, testing from beginning, 335–342

Properties
defining, 268
matching, 267–268

Prototypes
characters and meanings, 170
comparing reference types to, 209
for creating array or hash operators, 168

prove command
for flexible test runs, 308–309
randomizing test order, 310
running periodically, 349

used with other languages than Perl, 310–311

Prussian Approach, for untainting data, 373–374

Public code, mitigating the risk of, 235

Punctuation, avoiding excessive, 66–68

Q

\Q, for using metacharacters as literal characters, 102–103

q option, for quote strings, 74–75

QA (quality assurance) team, CPAN Testers on, 346–347

qq options, for quote strings, 74–75

qr//, precompiling regular expressions with, 138–139

Quantifiers
avoiding backtracking from, 134–135
possessive, nonbacktracking form of, 135–136

quote_identifier, creating dynamic SQL elements, 384

quotemeta operator, 102–103

qw(), for quoteless lists, 75

qw option, for quote strings, 74–75

R

\R, for line endings, 113

Randomizing test order, 310

Reading files
as if they were arrays, 197–198
maximizing speed of, 185–186
to strings, 186–187

- README files, installing `perl` and,** 391
- reduce function, summing numbers,** 93–94
- Refactoring code, to increase testability,** 339
- Reference arguments**
- passing to subroutines, 160–162
 - passing with `@_`, 154–157
 - returning from subroutines, 162
 - using `local * on`, 163–164
- Reference operator (`\`)**
- comparing reference types, 210
 - creating list of references, 61
- References**
- anonymous arrays constructors for creating, 214–215
 - autovivification of, 207–208
 - basic types, 210
 - cautions with use of circular data structures, 218–221
 - comparing reference types, 210–211
 - comparing with prototypes, 209
 - creating, 202–204
 - creating arrays of arrays using, 211–213
 - creating list of, 61
 - hashes used to simulate C-style `structs`, 216–218
 - `Log::Log4perl`, 409
 - manipulating complex data structures, 221
 - nesting with `map`, 223–224
 - overview of, 201
 - parts of strictures, 14
 - Perl Critic and, 398–399
 - `ref` operator, 210
 - selecting with `grep`, 224–225
 - slicing with `map`, 221–223
 - soft, 16, 208–209
 - using, 205–207
 - using typeglobs instead of, 425
- Regexp::Common module,** 142–143
- Regexp::Trie module,** 133
- Regular expressions**
- adding whitespace and comments to, 129–130
 - atoms, 99–100
 - avoiding backtracking due to alternation, 132–133
 - avoiding backtracking from quantifiers, 134–135
 - avoiding for CSVs, 412–414
 - avoiding for simple string operations, 125–126
 - avoiding parsing fixed columns with, 414–416
 - benchmarking, 139–141
 - breaking complex regular expressions into pieces, 130–132
 - capture and match variables used in substitutions, 107
 - capture buffers with backreferences, 105–106
 - capture variables, 103–105
 - character class used in place of alternation, 134
 - comparing strings, 126
 - compiling only once, 137–138
 - double-quote interpolation, 101–103
 - extracting and modifying substrings, 127–128
 - finding substrings, 126–127
 - graphemes in, 271–272
 - greed and, 119–121
 - horizontal whitespace, 111–112
 - line endings, 112–113
 - making readable, 129

Regular expressions (*continued*)

- match operator used in list context, 108
- match variables, 117–119
- matching beginning of a string with ^ or \A, 123–124
- matching boundaries between word and nonword characters, 121–123
- matching the end of a string with \$ or \Z, 124–125
- named capture for labeling matches, 114–115
- non-newlines, 113–114
- noncapturing parentheses for grouping, 116–117
- operator precedence, 100–101
- overview of, 99
- /p flag used with match variables, 119
- possessive, nonbacktracking form of quantifiers, 135–136
- precompiling, 138–139
- solutions in `Regexp::Common` module, 142–143
- tokenizing with, 108–110
- transliterating single characters, 128
- vertical whitespace, 112
- zero-width assertions for matching characters, 121

Repetition operators

- greedy and nongreedy, 120–121
- precedence of, 100

Resources, Perl, 435–437

rindex operator, finding substrings, 126–127

Rolsky, Dave, 235

rootLogger, Log::Log4perl

- configuration, 406

Run time warnings

- overview of, 359
- speed penalty of enabling, 360

S

\s and \S, matching whitespace and, 111

Scalar context, 44

Scalar variables, namespace for, 19

Scalars

- \$ for, 37
- Boolean operations applied to, 23
- containing either string or numeric data, 27
- context of, 42–43
- lists as ordered collection of, 31

Schwartz, Randal, 82

Schwartzian Transform, 82–83

Schwerin, Michael, 236, 307

Scope

- global variables and, 146
- lexical (compile-time) scoping with my, 146–149
- run time scoping with local, 149–151

seek, files/filehandles and, 188–189

selectall_arrayref, automatic value quoting using SQL placeholders, 382–383

Separator-retention mode, disabling in split, 117

Sequence operators

- overview of, 101
- precedence of, 100–101

Setup methods, Test::Class module, 334

- severity switch, using Perl Critic, 399–400
- Sharing data, closures for, 175
- Sharing database connections, 387–390
- shift**
 - @ARGV as default argument outside subroutines, 57–58, 429
 - @_ as default argument inside subroutines, 56–57
 - reading arguments passed to subroutines, 155
- shuffle** option, `prove` command, 310
- Shutdown methods, `Test::Class` module, 334
- Sigils, 17–18
- Single quotes (` `), options for quote strings, 73
- SKIP blocks, in testing, 313–314
- Slices**
 - avoiding slices when you want an element, 37–38
 - defined, 37
 - knowing difference between slices and arrays, 9
 - of lists and arrays, 32
 - lvalue slices, 38–39
 - not confusing slices with elements, 39–40
 - for sorting arrays, 40
 - syntax for permuting contents of arrays, 60
 - using hash slices to creates hashes, 41
- Slicing references, with `map`, 221–223
- Smart match operation (`~~`)**
 - given-when and, 87–88
 - knowing difference between string and numeric comparisons, 22
 - making work easier with, 84–85
- Smoke testers, creating custom, 348
- SmokeRunner::Multi**, 350
- Smolder, for aggregating test results, 350–351
- Soft references, 16, 208–209
- sort** operator, 77–78
- Sorting**
 - advanced sorting, 80–83
 - comparison (`sort`) subroutines, 78–80
 - with `pack`, 418
 - slices for sorting arrays, 40
 - `sort` operator, 77–78
- Source code**
 - quoting, 75
 - Unicode in, 254–256
- Source control, pre-commit hooks, 349
- Special characters, 102–103
- Spell checking code, in distributions, 297–298
- split**
 - avoiding regular expressions for CSVs, 412–413
 - disabling separator-retention mode, 117
- Spreadsheet::ParseExcel** module, 252
- Spreadsheet::WriteExcel** module, 252
- sprintf()**, for converting numbers to strings, 27
- SQL-injection attacks, 382–383
- SQL statements**
 - automatic value quoting using placeholders, 382–384
 - preparing multiple statements, 379–382

- SQL statements (*continued*)**
- reusing work and saving time using, 377–379
- SQLite, creating test databases,** 330–331
- Square brackets ([])**
- anonymous arrays constructor, 60
 - careful use of, 63
- Startup methods, `Test::Class` module,** 334
- `stat` operator**
- file test operators compared with, 180–181
 - `virtual_filehandle` for reusing data from last `stat`, 181
- `state` flag, `prove` command,** 310
- `state` variable, anonymous closures**
- compared with, 173
- STDIN, as default argument,** 58–59
- Stein, Lincoln,** 236
- Streams, options for reading from,** 183–185
- `strict` pragma,** 14
- Strictures**
- enabling for better coding, 14
 - parts of, 14
- String contexts,** 27
- String operators**
- comparison operators, 126
 - number and string context and, 42
- Strings**
- avoiding regular expressions for simple string operations, 125–126
 - comparing, 126
 - context of, 42
 - converting octet strings to character strings, 261–265
 - converting to numbers, 27–28
 - extracting and modifying substrings, 127–128
 - finding substrings, 126–127
 - knowing and testing false values, 23–25
 - knowing difference between string and numeric comparisons, 21–23
 - matching beginning of, 123–124
 - matching end of, 124–125
 - normalizing Unicode strings, 271
 - opening filehandles to/from, 186
 - options for quote strings, 73–77
 - reading files from a, 186–187
 - tokenizing, 108–110
 - using strings and numbers simultaneously, 28–30
 - writing files to a, 187–188
- Structs (C-style), simulating with hashes,** 216–218
- Structured testing, `Test::Class` for,** 332–335
- Style**
- idiomatic Perl style, 2
 - preferences, 7–8
- `subclass` argument,** `Module::Build`, 277–278
- Subclasses, creating to override features,** 322–324
- Subroutines**
- calling without using parentheses, 66–67
 - closures for locking in data, 171
 - closures for sharing data, 175
 - comparison (`sort`), 78–80
 - of context, 157–159
 - `Contextual::Return` for fine control, 159–160

creating inline subroutines using do {} syntax, 90–92

creating new subroutines (currying), 176–178

global variables, 146

hashes for passing named parameters, 164–168

lexical (compile-time) scoping with my, 146–149

local * on reference arguments, 163–164

multiple array arguments, 170–171

my and local as list operators, 153–154

namespace for, 19–20

overview of, 145

passing arguments, 154–157, 160–162

passing typeglobs, 163, 424

in Perl, 5

private data for named subroutines, 172–173

private data for subroutine references, 173–174

prototypes for creating array or hash operators, 168

redefining by assigning to their typeglobs, 424

returning arguments, 162

run time scoping with local, 149–151

separating functionality into subroutines for ease of testing, 321

void context, 159

wantarray for context of, 157–159

when to use local, 152–153

when to use my, 151–152

subs

- being careful with barewords, 15–16
- parts of strictures, 14

Substitutions, capture variables in, 107

substr operator, extracting and modifying substrings, 127–128

Substrings

- extracting and modifying, 127–128
- finding, 126–127

Subversion, source control and, 350

Sum, of numbers, 93–94

Swapping values, using list assignments, 60

Switch statements, given-when for, 86–90

Symbol table

- accessing with typeglobs, 423–425
- redefining test methods with, 327–328

sysread, maximizing speed of reading files, 185–186

T

t/ directory, test files stored in, 311

-t operator, STDIN as default argument, 58

t/perlcriticrc file, 401

Taint checking

- defined, 357
- for legacy code, 375–376
- tracking dangerous data with, 372–375
- untainting data, 373–374

TAP (Test Anywhere Protocol), 307, 310

TDD (test-driven development), 335, 338

Teardown methods, Test::Class module, 334

tell, files/filehandles and, 188–189
Template Toolkit templates, 282
Templates, creating custom, 282–283
Template::Toolkit module, 252
Temporary directories, 198–199
Temporary files, 198–199
Test Anywhere Protocol (TAP), 307, 310
Test coverage
Devel::Cover used to watch test results, 343–345
 hard-to-cover code and, 345–346
 measuring, 342–343
Test-driven development (TDD), 335, 338
Test [n] methods, Test::Class module, 335
TEST_AUTHOR environment variable, 311–312
Test::Builder module, 332
Test::Class, for structured testing, 332–335
Test::Harness module, 307
Testing
 adapting test requirements to application needs, 317–319
 author tests, 311–312
 continuous builds and, 348–349
 CPAN Testers on QA team, 346–347
 creating custom smoke testers, 348
 Cruise Control for integration testing, 351–355
 dependency injection for avoiding special test logic, 314–317
Devel::Cover used to watch test results, 343–345
 hard-to-cover code and, 345–346

limiting tests to right situations, 313
 local test databases, 330–331
 measuring test coverage, 342–343
 mock objects and interfaces for focused testing, 324–325
 modulinos for easy testing, 320–324
 overriding Perl built-ins, 328–329
 overview of, 307–308
 pre-commit hooks, 349–350
 prove for flexible test runs, 308–309
 prove, running periodically, 349
 prove used with other languages than Perl, 310–311
 randomizing test order, 310
 redefining test methods with symbol table, 327–328
 setting CPAN Tester preferences, 347
 skipping tests in automated environment, 313–314
Smolder for aggregating test results, 350–351
SQLite used to create test databases, 330
 starting at project beginning, 335–342
Test::Class for structured testing, 332–335
Test::MockObject, 325–327
 using developer versions of CPAN Tester, 347–348
Test::MockObject, 325–327
Test::More module
 overview of, 236–237
 Schwern’s promotion of, 307
Test::Class module compared with, 332
Test::Perl::Critic, 401
Test::Perl::Critic::Progressive, 401–402

T

Test : : **PerlTidy**, 397
Text : : **CSV_xs** module, 252
Text : : **Template** module, 252
 Themes, using Perl Critic, 400
 There's More Than One Way To Do It (TMTOWTDI) acronym, 398
 Three-argument **open**, 182–183
Tie : : **File**, 197
 TMTOWTDI (There's More Than One Way To Do It) acronym, 398
 Tokenizing strings, 108–110
 Top-level namespaces, 286
 Transliteration operator (**tr///**), 128
 Tregar, Sam, 246
Regexp : : **Trie**, minimizing backtracking with, 133
try - catch exception handling, 371–372
Try : : **Tiny**
 catching exceptions with, 366
 handling exceptions properly, 370–372
 Tuples, storing point data as, 46
 Two-argument form, of **open**, 182–183
 Typeglobs
 accessing symbol table with, 423–425
 passing to subroutines, 163
 redefining subroutines by assigning to its typeglob, 327

U

UCS character set, 254
undef values, not assigning for empty arrays, 34–37

Unicode
 aliases used for characters, 260
 care in use with databases, 272–273
 converting octet strings to character strings, 261–265
 defining own properties, 268
 getting code points from names, 260
 getting names from code points, 259–260
 graphemes instead of characters, 269–272
 matching characters, 266
 matching properties, 267–268
 overview of, 253–254
 Perl handling, 99
 setting default encoding, 257–258
 setting encoding on selected filehandles, 258
 specifying characters by code point or name, 258–259
 telling Perl which encoding system to use, 256
 using in source code, 254–256
 Unicode Consortium Web site, 259
Unicode : : **CharName**, 261
uninitialized warning, turning off, 362
 UNIX, Perl as scripting tool for, 1
unpack
 dealing with endianness, 419–421
 how it works, 416–417
 manipulating hex escapes, 418–419
 parsing fixed-width columns, 414–416
 uuencoding with, 421–422
Upgrades
 considering need for, 236
 restricting frequency of, 238

use diagnostics, for warning information, 360

use directive, for enabling new features, 13

use keyword, for specifying right Perl version, 292–293

use warnings, enabling per-file warnings with, 358–361

UTF-8

- sort order based on, 78
- UCS encoding, 254
- utf8 pragma**, 255

utf8 pragma, 255

uuencoding, 421–422

V

\v, for matching vertical whitespace, 112

Values

- avoiding regular expressions for comma-separated values, 412–414
- finding maximum value, 92–93
- knowing and testing false, 23–25
- undef** values, 34–37

Variables

- alias variables, 424
- avoiding groups of, 46–47
- creating dualvars, 30–31
- declaring, 15
- environment variables. *See Environment variables*
- global variables, 146
- lexical (compile-time) scoping with **my**, 146–149
- localizing, 5
- log levels defined by **Log::Log4perl**, 404–405

namespaces for, 19–20

not interpolating into SQL statements, 382–383

run time scoping with **local**, 149–151

vars

- declaring variables, 15
- parts of strictures, 14

Verbatim text, in Pod, 288

Versions, specifying correct, 292–293

Vertical whitespace, 112

Very High Level Language (VHLL), 1

VHLL (Very High Level Language), 1

viacode, getting names from code points, 259–260

Virtual_filehandle, for reusing data from last **stat**, 181

Void context

- overview of, 44
- subroutines and, 159

W

Wall, Larry, 9, 41

wantarray operator, for context of subroutines, 157–159

warn, using **croak** in place of, 367–368

warnif, using predefined warning categories in modules, 363

Warnings

- enabling, 357–358
- generating exceptions using **die**, 364–366
- getting stack traces using **carp**, 366–369
- handling exceptions, 370–372

for legacy code, 375–376
 overview of, 357
 Perl Critic, 398–400
 promoting to fatal errors, 362–363
 for suspicious code, 358–361
 taint checking, 372–375
 toggling complaints using lexical,
 361–364
 understanding using diagnostics,
 360

Web. *See Online*

Websites, Perl, 436–437

when, in `foreach` loop, 89

while loop
`perl -n` switch and, 429
`perl -p` switch and, 431

Whitespace
 adding to regular expressions,
 129–130
 avoiding regular expressions for
 CSVs, 413
 horizontal, 111–112
 overview of, 110–111
 parsing column data with `unpack`
 and, 414–416
 Unicode and, 268
 vertical, 112

Word boundaries, matching, 121–123

Writing files, from strings, 187–188

**Writing Perl Modules for CPAN
 (Tregar)**, 246

WWW::Mechanize module, 251

X

/x flag, adding insignificant whitespace and comments to regular expressions, 130

XML::Compile module, 252

XML::Twig module, 252

XS language, 301–306
 generating boilerplate for C implementation, 302–303
 overview of, 301–302
 writing and testing XSUB,
 303–306

xsubpp compiler, 303

xt/ (extra tests) directory, author tests stored in, 312

xUnit functionality, Test::Class module for, 332

Y

YAML, 252

Z

\Z anchor, matching end of a string, 124

Zero-width assertions
 for matching characters, 121
 for word boundaries, 121–123

Zero-width atoms, 100–101