Mastering Perl

brian d foy
foreword by Randal L. Schwartz

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

Mastering Perl
by brian d foy

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

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Andy Oram
Production Editor: Adam Witwer
Proofreader: Sohaila Abdulali

Indexer: Joe Wizda
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrators: Robert Romano and Jessamyn Read

Printing History:
July 2007: First Edition.

This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN-10: 0-596-52724-1
ISBN-13: 978-0-596-52724-2

[M]

Table of Contents

Foreword

One of the problems we face at Stonehenge as professional trainers is to make sure that we write materials that are reusable in more than one presentation. The development expense of a given set of lecture notes requires us to consider that we'll need roughly two to four hundred people who are all starting in roughly the same place, and who want to end up in the same place, and who we can find in a billable situation.

With our flagship product, the *Learning Perl* course, the selection of topics was easy: pick all the things that nearly everyone will need to know to write single-file scripts across the broad range of applications suited for Perl, and that we can teach in the first week of classroom exposure.

When choosing the topics for *Intermediate Perl*, we faced a slightly more difficult challenge, because the "obvious" path is far less obvious. We concluded that in the second classroom week of exposure to Perl, people will want to know what it takes to write complex data structures and objects, and work in groups (modules, testing, and distributions). Again, we seemed to have hit the nail on the head, as the course and book are very popular as well.

Fresh after having updated our *Learning Perl* and *Intermediate Perl* books, brian d foy realized that there was still more to say about Perl just beyond the reach of these two tutorials, although not necessarily an "all things for all people" approach.

In *Mastering Perl*, brian has captured a number of interesting topics and written them down with lots of examples, all in fairly independently organized chapters. You may not find everything relevant to your particular coding, but this book can be picked up and set back down again as you find time and motivation—a luxury that we can't afford in a classroom. While you won't have the benefit of our careful in-person elaborations and interactions, brian does a great job of making the topics approachable and complete.

And oddly enough, even though I've been programming Perl for almost two decades, I learned a thing or two going through this book, so brian has really done his homework. I hope you find the book as enjoyable to read as I have.

—Randal L. Schwartz

Preface

Mastering Perl is the third book in the series starting with *Learning Perl*, which taught you the basics of Perl syntax, progressing to *Intermediate Perl*, which taught you how to create reusable Perl software, and finally this book, which pulls everything together to show you how to bend Perl to your will. This isn't a collection of clever tricks, but a way of thinking about Perl programming so you integrate the real-life problems of debugging, maintenance, configuration, and other tasks you'll encounter as a working programmer. This book starts you on your path to becoming the person with the answers, and, failing that, the person who knows how to find the answers or discover the problem.

Structure of This Book

Chapter 1, *Introduction: Becoming a Master*
: An introduction to the scope and intent of this book.

Chapter 2, *Advanced Regular Expressions*
: More regular expression features, including global matches, lookarounds, readable regexes, and regex debugging.

Chapter 3, *Secure Programming Techniques*
: Avoid some common programing problems with the techniques in this chapter, which covers taint checking and gotchas.

Chapter 4, *Debugging Perl*
: A little bit about the Perl debugger, writing your own debugger, and using the debuggers others wrote.

Chapter 5, *Profiling Perl*
: Before you set out to improve your Perl program, find out where you should concentrate your efforts.

Chapter 6, *Benchmarking Perl*
: Figure out which implementations do better on time, memory, and other metrics, along with cautions about what your numbers actually mean.

Chapter 7, *Cleaning Up Perl*
: Wrangle Perl code you didn't write (or even code you did write) to make it more presentable and readable by using `Perl::Tidy` or `Perl::Critic`.

Chapter 8, *Symbol Tables and Typeglobs*
: Learn how Perl keeps track of package variables and how you can use that mechanism for some powerful Perl tricks.

Chapter 9, *Dynamic Subroutines*
: Define subroutines on the fly and turn the tables on normal procedural programming. Iterate through subroutine lists rather than data to make your code more effective and easy to maintain.

Chapter 10, *Modifying and Jury-Rigging Modules*
: Fix code without editing the original source so you can always get back to where you started.

Chapter 11, *Configuring Perl Programs*
: Let your users configure your programs without touching the code.

Chapter 12, *Detecting and Reporting Errors*
: Learn how Perl reports errors, how you can detect errors Perl doesn't report, and how to tell your users about them.

Chapter 13, *Logging*
: Let your Perl program talk back to you by using `Log4perl`, an extremely flexible and powerful logging package.

Chapter 14, *Data Persistence*
: Store data for later use in other programs, a later run of the same program, or to send as text over a network.

Chapter 15, *Working with Pod*
: Translate plain ol' documentation into any format that you like, and test it, too.

Chapter 16, *Working with Bits*
: Use bit operations and bit vectors to efficiently store large data.

Chapter 17, *The Magic of Tied Variables*
: Implement your own versions of Perl's basic data types to perform fancy operations without getting in the user's way.

Chapter 18, *Modules As Programs*
: Write programs as modules to get all of the benefits of Perl's module distribution, installation, and testing tools.

Appendix A
: Explore these resources to continue your Perl education.

Appendix B
: My popular step-by-step guide to solving any Perl problem. Follow these steps to improve your troubleshooting skills.

Conventions Used in This Book

The following typographic conventions are used in this book:

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

Italics
: Used for emphasis, Perl documentation, filenames, and for new terms where they are defined

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact O'Reilly for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books *does* require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation *does* require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Mastering Perl* by brian d foy. Copyright 2007 O'Reilly Media, Inc., 978-0-596-52724-2."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Enabled

When you see a Safari® Enabled icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://safari.oreilly.com*.

Comments and Questions

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

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

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

http://www.oreilly.com/catalog/9780596527242

Additionally, while the book was being written, the author maintained a web site for this book where you can find additional information, including links to related resources and possible updates to the book:

http://www.pair.com/comdog/mastering_perl

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

bookquestions@oreilly.com

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

http://www.oreilly.com

Acknowledgments

Many people helped me during the year I took to write this book. The readers of the *Mastering Perl* mailing list gave constant feedback on the manuscript and sent patches, which I mostly applied as is, including those from Andy Armstrong, David H. Adler, Renée Bäcker, Anthony R. J. Ball, Daniel Bosold, Alessio Bragadini, Philippe Bruhat, Katharine Farah, Shlomi Fish, David Golden, Bob Goolsby, Ask Bjørn Hansen, Jarkko Hietaniemi, Joseph Hourcle, Adrian Howard, Offer Kaye, Stefan Lidman, Eric Maki, Josh McAdams, Florian Merges, Jason Messmer, Thomas Nagel, Xavier Noria, Les Peters, Bill Riker, Yitzchak Scott-Thoennes, Ian Sealy, Sagar R. Shah, Alberto Simões, Derek B. Smith, Kurt Starsinic, Adam Turoff, David Westbrook, and Evan Zacks. I'm quite reassured that their constant scrutiny kept me on the right path.

Tim Bunce provided gracious advice about the profiling chapter, which includes `DBI::Profile`, and Jeffrey Thalhammer updated me on the current developments with his `Perl::Critic` module.

Perrin Harkins, Rob Kinyon, and Randal Schwartz gave the manuscript a thorough beating at the end, and I'm glad I chose them as technical reviewers because their advice is always spot on.

Allison Randal provided valuable Perl advice and editorial guidance on the project, even though she probably dreaded my constant queries. Near the end of the year, Andy Oram took over as editor and helped me get the manuscript into shape so we could turn it into a book. The entire O'Reilly Media staff, from editorial, production, marketing, sales, and everyone else, was friendly and helpful, and it's always a pleasure to work with them. It takes much more than an author to create a book, so thank a random O'Reilly employee next time you see one.

Randal Schwartz, my partner at Stonehenge Consulting, warned me that writing a book was a lot of work and still let me mostly take the year off to do it. I started in Perl by reading his *Learning Perl* and am now quite pleased to be adding another book to the series. As Randal has told me many times "You'll get paid more at Starbucks and get health insurance, too." Authors write to share their thoughts with the world, and we write to make other people better programmers.

Finally, I have to thank the Perl community, which has been incredibly kind and supportive over the 10 years that I've been part of it. So many great programmers and managers helped me become a better programmer, and I hope this book does the same for people just joining the crowd.

CHAPTER 1
Introduction: Becoming a Master

This book isn't going to make you a Perl master; you have to do that for yourself by programming a lot of Perl, trying a lot of new things, and making a lot of mistakes. I'm going to help you get on the right path. The road to mastery is one of self-reliance and independence. As a Perl master, you'll be able to answer your own questions as well as those of others.

In the golden age of guilds, craftsmen followed a certain path, both literally and figuratively, as they mastered their craft. They started as apprentices and would do the boring bits of work until they had enough skill to become the more trusted journeymen. The journeyman had greater responsibility but still worked under a recognized master. When he had learned enough of the craft, the journeyman would produce a "master work" to prove his skill. If other masters deemed it adequately masterful, the journeyman became a recognized master himself.

The journeymen and masters also traveled (although people disagree on whether that's where the "journey" part of the name came from) to other masters, where they would learn new techniques and skills. Each master knew things the others didn't, perhaps deliberately guarding secret methods, or knew it in a different way. Part of a journeyman's education was learning from more than one master.

Interactions with other masters and journeymen continued the master's education. He learned from those masters with more experience and learned from himself as he taught journeymen, who also taught him because they brought skills they learned from other masters.

The path an apprentice followed affected what he learned. An apprentice who studied with more masters was exposed to many more perspectives and ways of teaching, all of which he could roll into his own way of doing things. Odd teachings from one master could be exposed by another, giving the apprentice a balanced view on things. Additionally, although the apprentice might be studying to be a carpenter or a mason, different masters applied those skills to different goals, giving the apprentice a chance to learn different applications and ways of doing things.

Unfortunately, we don't operate under the guild system. Most Perl programmers learn Perl on their own (I'm sad to say, as a Perl instructor), program on their own, and never get the advantage of a mentor. That's how I started. I bought the first edition of *Learning Perl* and worked through it on my own. I was the only person I knew who knew what Perl was, although I'd seen it around a couple of times. Most people used what others had left behind. Soon after that, I discovered comp.lang.perl.misc and started answering any question that I could. It was like self-assigned homework. My skills improved and I got almost instantaneous feedback, good and bad, and I learned even more Perl. I ended up with a job that allowed me to program Perl all day, but I was the only person in the company doing that. I kept up my homework on comp.lang.perl.misc.

I eventually caught the eye of Randal Schwartz, who took me under his wing and started my Perl apprenticeship. He invited me to become a Perl instructor with Stonehenge Consulting Services, and then my real Perl education began. Teaching, meaning figuring out what you know and how to explain it to others, is the best way to learn a subject. After a while of doing that, I started writing about Perl, which is close to teaching, although with correct grammar (mostly) and an editor to correct mistakes.

That presents a problem for *Mastering Perl*, which I designed to be the third book of a trilogy starting with *Learning Perl* and *Intermediate Perl*, both of which I've had a hand in. Each of those are about 300 pages, and that's what I'm limited to here. How do I encapsulate the years of my experience in such a slim book?

In short, I can't. I'll teach you what I think you should know, but you'll also have to learn from other sources. As with the old masters, you can't just listen to one person. You need to find other masters, too, and that's also the great thing about Perl: you can do things in so many different ways. Some of these masters have written very good books, from this publisher and others, so I'm not going to duplicate those topics here, as I discuss in a moment.

What It Means to Be a Master

This book takes a different tone from *Learning Perl* and *Intermediate Perl*, which we designed as tutorial books. Those mostly cover the details of the Perl language and only a little on the practice of programming. *Mastering Perl*, however, puts more responsibility on you, the reader.

Now that you've made it this far in Perl, you're working on your ability to answer your own questions and figure out things on your own, even if that's a bit more work than simply asking someone. The very act of doing it yourself builds your experience as well as not annoying your coworkers.

Although I don't cover other languages in this book, like *Advanced Perl Programming*, First Edition, by Sriram Srinivasan (O'Reilly) and *Mastering Regular Expressions* by Jeffrey Friedl (O'Reilly) do, you should learn some other languages. This

informs your Perl knowledge and gives you new perspectives, some that make you appreciate Perl more and others that help you understand its limitations.

And, as a master, you will run into Perl's limitations. I like to say that if you don't have a list of five things you hate about Perl and the facts to back them up, you probably haven't done enough Perl. It's not really Perl's fault. You'll get that with any language. The mastery comes in by knowing these things and still choosing Perl because its strengths outweigh the weakness for your application. You're a master because you know both sides of the problem and can make an informed choice that you can explain to others.

All of that means that becoming a master involves work, reading, and talking to other people. The more you do, the more you learn. There's no shortcut to mastery. You may be able to learn the syntax quickly, as in any other language, but that will be the tiniest portion of your experience. Now that you know most of Perl, you'll probably spend your time reading some of the "meta"-programming books that discuss the practice of programming rather than just slinging syntax. Those books will probably use a language that's not Perl, but I've already said you need to learn some other languages, if only to be able to read these books. As a master, you're always learning.

Becoming a master involves understanding more than you need to, doing quite a bit of work on your own, and learning as much as you can from the experience of others. It's not just about the code you write, because you have to deal with the code from many other authors too.

It may sound difficult, but that's how you become a master. It's worth it, so don't give up. Good luck!

Who Should Read This Book

I wrote this book as a successor to *Intermediate Perl*, which covered the basics of references, objects, and modules. I'll assume that you already know and feel comfortable with those features. Where possible, I make references to *Intermediate Perl* in case you need to refresh your skills on a topic.

If you're coming directly from another language and haven't used Perl yet, or have only used it lightly, you might want to skim *Learning Perl* and *Intermediate Perl* to get the basics of the language. Still, you might not recognize some of the idioms that come with experience and practice. I don't want to tell you not to buy this book (hey, I need to pay my mortgage!), but you might not get the full value I intend, at least not right away.

How to Read This Book

I'm not writing a third volume of "Yet More Perl Features." I want to teach you how to learn Perl on your own. I'm setting you on your own path to mastery, and as an apprentice, you'll need to do some work on your own. Sometimes this means I'll show

you where in the Perl documentation to get the answers (meaning I can use the saved space to talk about other topics).

What Should You Know Already?

I'll presume that you already know everything that we covered in *Learning Perl* and *Intermediate Perl*. By we, I mean the Stonehenge Consulting Services crew and best-selling Perl coauthors Randal Schwartz, Tom Phoenix, and me.

Most importantly, you should know these subjects, each of which implies knowledge of other subjects:

- Using Perl modules
- Writing Perl modules
- References to variables, subroutines, and filehandles
- Basic regular expression syntax and workings
- Object-oriented Perl

If I want to discuss something not in either of those books, I'll explain it in a bit more depth. Even if we did cover it in the previous books, I might cover it again just because it's that important.

What I Cover

After learning the basic syntax of Perl in *Learning Perl* and the basics of modules and team programming in *Intermediate Perl*, the next thing you need to learn are the idioms of Perl and the integration of the skills that you already have to create robust and scalable applications that other people can use without your help.

I'll cover some subjects you've seen in those two books, but in more depth. As we said in *Learning Perl*, we sometimes told white lies to simplify the details and to get you going as soon as possible without getting bogged down. Now it's time to get a bit dirty in the bogs.

Don't mistake my coverage of a subject for an endorsement, though. There are millions of Perl programmers in the world, and each has her own way of doing things. Part of becoming a Perl master involves reading quite a bit of Perl even if you wouldn't write that Perl yourself. I'll endeavor to tell you when I think you shouldn't do something, but that's really just my opinion. As you strive to be a good programmer, you'll need to know more than you'll use. Sometimes I'll show things I don't want you to use, but I know you'll see in code from other people. Oh well, it's not a perfect world.

Not all programming is about adding or adjusting features in code. Sometimes it's pulling code apart to inspect it and watch it do its magic. Other times it's about getting rid of code that you don't need. The practice of programming is more than creating

applications. It's also about managing and wrangling code. Some of the techniques I'll show are for analysis, not your own development.

What I Don't Cover

As I talked over the idea of this book with the editors, we decided not to duplicate the subjects more than adequately covered by other books. You need to learn from other masters, too, and I don't really want to take up more space on your shelf than I really need. Ignoring those subjects gives me the double bonus of not writing those chapters and using that space for other things. You should already have read those other books anyway.

That doesn't mean that you get to ignore those subjects, though, and where appropriate I'll point you to the right book. In Appendix A, I list some books I think you should add to your library as you move towards Perl mastery. Those books are by other Perl masters, each of whom has something to teach you. At the end of most chapters I point you toward other resources as well. A master never stops learning.

Since you're already here, though, I'll just give you the list of topics I'm explicitly avoiding, for whatever reason: Perl internals, embedding Perl, threads, best practices, object-oriented programming, source filters, and dolphins. This is a dolphin-safe book.

CHAPTER 2

Advanced Regular Expressions

Regular expressions, or just regexes, are at the core of Perl's text processing, and certainly are one of the features that made Perl so popular. All Perl programmers pass through a stage where they try to program everything as regexes and, when that's not challenging enough, everything as a single regex. Perl's regexes have many more features than I can, or want, to present here, so I include those advanced features I find most useful and expect other Perl programmers to know about without referring to *perlre*, the documentation page for regexes.

References to Regular Expressions

I don't have to know every pattern at the time that I code something. Perl allows me to interpolate variables into regexes. I might hard code those values, take them from user input, or get them in any other way I can get or create data. Here's a tiny Perl program to do `grep`'s job. It takes the firstF argument from the command line and uses it as the regex in the `while` statement. That's nothing special (yet); we showed you how to do this in *Learning Perl*. I can use the string in `$regex` as my pattern, and Perl compiles it when it interpolates the string in the match operator:*

```
#!/usr/bin/perl
# perl-grep.pl

my $regex = shift @ARGV;

print "Regex is [$regex]\n";

while( <> )
        {
        print if m/$regex/;
        }
```

* As of Perl 5.6, if the string does not change, Perl will not recompile that regex. Before Perl 5.6, I had to use the `/o` flag to get that behavior. I can still use `/o` if I don't want to recompile the pattern even if the variable changes.

I can use this program from the command line to search for patterns in files. Here I search for the pattern `new` in all of the Perl programs in the current directory:

```
% perl-grep.pl new *.pl
Regex is [new]
my $regexp = Regexp::English->new
my $graph = GraphViz::Regex->new($regex);
                [ qr/\G(\n)/,                "newline"     ],
                                               { ( $1, "newline char"     ) }
print YAPE::Regex::Explain->new( $ARGV[0] )->explain;
```

What happens if I give it an invalid regex? I try it with a pattern that has an opening parenthesis without its closing mate:

```
$ ./perl-grep.pl "(perl" *.pl
Regex is [(perl]
Unmatched ( in regex; marked by <-- HERE in m/( <-- HERE perl/
        at ./perl-grep.pl line 10, <> line 1.
```

When I interpolate the regex in the match operator, Perl compiles the regex and immediately complains, stopping my program. To catch that, I want to compile the regex before I try to use it.

The `qr//` is a regex quoting operator that stores my regex in a scalar (and as a quoting operator, its documentation shows up in *perlop*). The `qr//` compiles the pattern so it's ready to use when I interpolate `$regex` in the match operator. I wrap the `eval` operator around the `qr//` to catch the error, even though I end up `die`-ing anyway:

```
#!/usr/bin/perl
# perl-grep2.pl

my $pattern = shift @ARGV;

my $regex = eval { qr/$pattern/ };
die "Check your pattern! $@" if $@;

while( <> )
        {
        print if m/$regex/;
        }
```

The regex in `$regex` has all of the features of the match operator, including back references and memory variables. This pattern searches for a three-character sequence where the first and third characters are the same, and none of them are whitespace. The input is the plain text version of the *perl* documentation page, which I get with `perldoc -t`:

```
% perldoc -t  perl | perl-grep2.pl "\b(\S)\S\1\b"
    perl583delta        Perl changes in version 5.8.3
    perl582delta        Perl changes in version 5.8.2
    perl581delta        Perl changes in version 5.8.1
    perl58delta         Perl changes in version 5.8.0
    perl573delta        Perl changes in version 5.7.3
    perl572delta        Perl changes in version 5.7.2
    perl571delta        Perl changes in version 5.7.1
    perl570delta        Perl changes in version 5.7.0
```

```
    perl561delta        Perl changes in version 5.6.1
http://www.perl.com/       the Perl Home Page
http://www.cpan.org/       the Comprehensive Perl Archive
http://www.perl.org/       Perl Mongers (Perl user groups)
```

It's a bit hard, at least for me, to see what Perl matched, so I can make another change to my `grep` program to see what matched. The `$&` variable holds the portion of the string that matched:

```
#!/usr/bin/perl
# perl-grep3.pl

my $pattern = shift @ARGV;

my $regex = eval { qr/$pattern/ };
die "Check your pattern! $@" if $@;

while( <> )
        {
        print "$_\t\tmatched >>>$&<<<\n" if m/$regex/;
        }
```

Now I see that my regex is matching a literal dot, character, literal dot, as in `.8.`:

```
% perldoc -t perl | perl-grep3.pl  "\b(\S)\S\1\b"
                perl587delta        Perl changes in version 5.8.7
                                 matched >>>.8.<<<
                perl586delta        Perl changes in version 5.8.6
                                 matched >>>.8.<<<
                perl585delta        Perl changes in version 5.8.5
                                 matched >>>.8.<<<
```

Just for fun, how about seeing what matched in each memory group, the variables `$1`, `$2`, and so on? I could try printing their contents, whether or not I had capturing groups for them, but how many do I print? Perl already knows because it keeps track of all of that in the special arrays `@-` and `@+`, which hold the string offsets for the beginning and end, respectively, for each match. That is, for the match string in `$_`, the number of memory groups is the last index in `@-` or `@+` (they'll be the same length). The first element in each is for the part of the string matched (so, `$&`), and the next element, with index `1`, is for `$1`, and so on for the rest of the array. The value in `$1` is the same as this call to `substr`:

```
my $one = substr(
        $_,             # string
        $-[1],          # start position for $1
        $+[1] - $-[1]   # length of $1 (not end position!)
        );
```

To print the memory variables, I just have to go through the indices in the array `@-`:

```
#!/usr/bin/perl
# perl-grep4.pl

my $pattern = shift @ARGV;
```

```
my $regex = eval { qr/$pattern/ };
die "Check your pattern! $@" if $@;

while( <> )
        {
        if( m/$regex/ )
                {

                print "$_";

                print "\t\t\$&: ",
                        substr( $_, $-[$i], $+[$i] - $-[$i] ),
                        "\n";

                foreach my $i ( 1 .. $#- )
                        {
                        print "\t\t\$$i: ",
                                substr( $_, $-[$i], $+[$i] - $-[$i] ),
                                "\n";
                        }
                }
        }
```

Now I can see the part of the string that matched as well as the submatches:

```
% perldoc -t perl | perl-grep4.pl  "\b(\S)\S\1\b"
                perl587delta        Perl changes in version 5.8.7
                                $&: .8.
                                $1: .
```

If I change my pattern to have more submatches, I don't have to change anything to see the additional matches:

```
% perldoc -t perl | perl-grep4.pl  "\b(\S)(\S)\1\b"
                perl587delta        Perl changes in version 5.8.7
                                $&: .8.
                                $1: .
                                $2: 8
```

(?imsx-imsx:PATTERN)

What if I want to do something a bit more complex for my `grep` program, such as a case-insensitive search? Using my program to search for either "Perl" or "perl" I have a couple of options, neither of which are too much work:

```
% perl-grep.pl "[pP]erl"
% perl-grep.pl "(p|P)erl"
```

If I want to make the entire pattern case-insensitive, I have to do much more work, and I don't like that. With the match operator, I could just add the `/i` flag on the end:

```
print if m/$regex/i;
```

I could do that with the qr// operator, too, although this makes all patterns case-insensitive now:

```
my $regex = qr/$pattern/i;
```

To get around this, I can specify the match options inside my pattern. The special sequence (?imsx) allows me to turn on the features for the options I specify. If I want case-insensitivity, I can use (?i) inside the pattern. Case-insensitivity applies for the rest of the pattern after the (?i) (or for the rest of the enclosing parentheses):

```
% perl-grep.pl "(?i)perl"
```

In general, I can enable flags for part of a pattern by specifying which ones I want in the parentheses, possibly with the portion of the pattern they apply to, as shown in Table 2-1.

Table 2-1. Options available in the (?options:PATTERN)

Inline option	Description
(?i:PATTERN)	Make case-insensitive
(?m:PATTERN)	Use multiline matching mode
(?s:PATTERN)	Let . match a newline
(?x:PATTERN)	Turn on eXplain mode

I can even group them:

```
(?si:PATTERN)    Let . match a newline and make case-insensitive
```

If I preface the options with a minus sign, I turn off those features for that group:

```
(?-s:PATTERN)    Don't let . match a newline
```

This is especially useful since I'm getting my pattern from the command line. In fact, when I use the qr// operator to create my regex, I'm already using these. I'll change my program to print the regex after I create it with qr// but before I use it:

```
#!/usr/bin/perl
# perl-grep3.pl

my $pattern = shift @ARGV;

my $regex = eval { qr/$pattern/ };
die "Check your pattern! $@" if $@;

print "Regex ---> $regex\n";

while( <> )
        {
        print if m/$regex/;
        }
```

When I print the regex, I see it starts with all of the options turned off. The string version of regex uses (?-OPTIONS:PATTERN) to turn off all of the options:

```
% perl-grep3.pl "perl"
Regex ---> (?-xism:perl)
```

I can turn on case-insensitivity, although the string form looks a bit odd, turning off `i` just to turn it back on:

```
% perl-grep3.pl "(?i)perl"
Regex ---> (?-xism:(?i)perl)
```

Perl's regexes have many similar sequences that start with a parenthesis, and I'll show a few of them as I go through this chapter. Each starts with an opening parenthesis followed by some characters to denote what's going on. The full list is in *perlre*.

References As Arguments

Since references are scalars, I can use my compiled regex just like any other scalar, including storing it in an array or a hash, or passing it as the argument to a subroutine. The `Test::More` module, for instance, has a `like` function that takes a regex as its second argument. I can test a string against a regex and get richer output when it fails to match:

```
use Test::More 'no_plan';

my $string = "Just another Perl programmer,";
like( $string, qr/(\S+) hacker/, "Some sort of hacker!" );
```

Since `$string` uses `programmer` instead of `hacker`, the test fails. The output shows me the string, what I expected, and the regex it tried to use:

```
not ok 1 - Some sort of hacker!
1..1
#   Failed test 'Some sort of hacker!'
#                   'Just another Perl programmer,'
#     doesn't match '(?-xism:(\S+) hacker)'
# Looks like you failed 1 test of 1.
```

The `like` function doesn't have to do anything special to accept a regex as an argument, although it does check its reference type† before it tries to do its magic:

```
if( ref $regex eq 'Regexp' ) { ... }
```

Since `$regex` is just a reference (of type `Rexexp`), I can do reference sorts of things with it. I use `isa` to check the type, or get the type with `ref`:

```
print "I have a regex!\n" if $regex->isa( 'Regexp' );
print "Reference type is ", ref( $regex ), "\n";
```

† That actually happens in the `maybe_regex` method in `Test::Builder`.