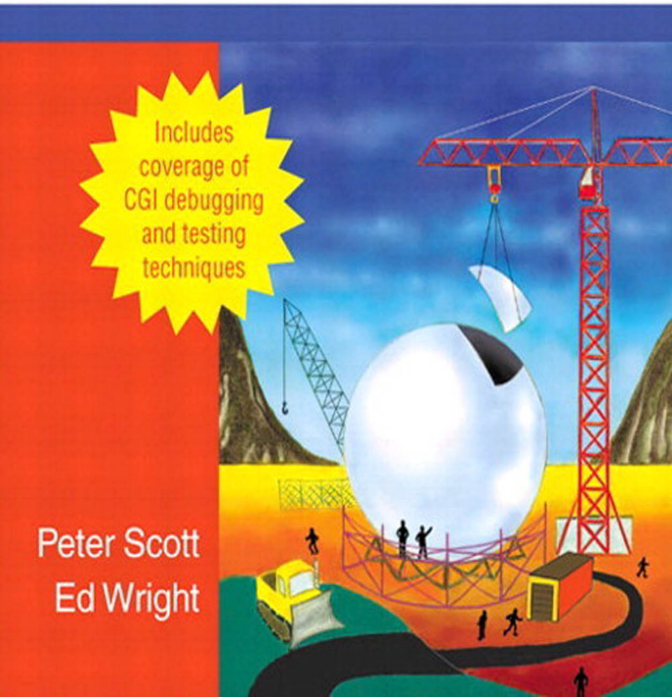
Perl Debugged



FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Perl Debugged

This page intentionally left blank

Perl Debugged

Peter Scott and Ed Wright

ADDISON-WESLEY

Boston ♦ San Francisco ♦ New York ♦ Toronto ♦ Montreal
London ♦ Munich ♦ Paris ♦ Madrid
Capetown ♦ Sydney ♦ Tokyo ♦ Singapore ♦ Mexico City

Pearson Education Corporate Sales Division
One Lake Street
Upper Saddle River, NJ 07458
(800) 382-3419
corpsales@pearsontechgroup.com

Visit AW on the Web: www.awl.com/cseng/

Library of Congress Cataloging-in-Publication Data

Scott, Peter.
Perl Debugged / Peter Scott and Ed Wright.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-70054-9
1. Perl (Computer program language) 2. Debugging in computer science.
I. Wright, Ed. II. Title.

QA76.73.P22 S39 2001 00-067550
005.13'3--dc21 CIP

2 3 4 5 6 7 8 9 10—CRS—0504030201
Second printing, May 2001

Peter:

To my father, for teaching me the joy of continual learning.

Ed:

To Grandma. Hey, Grandma, look!

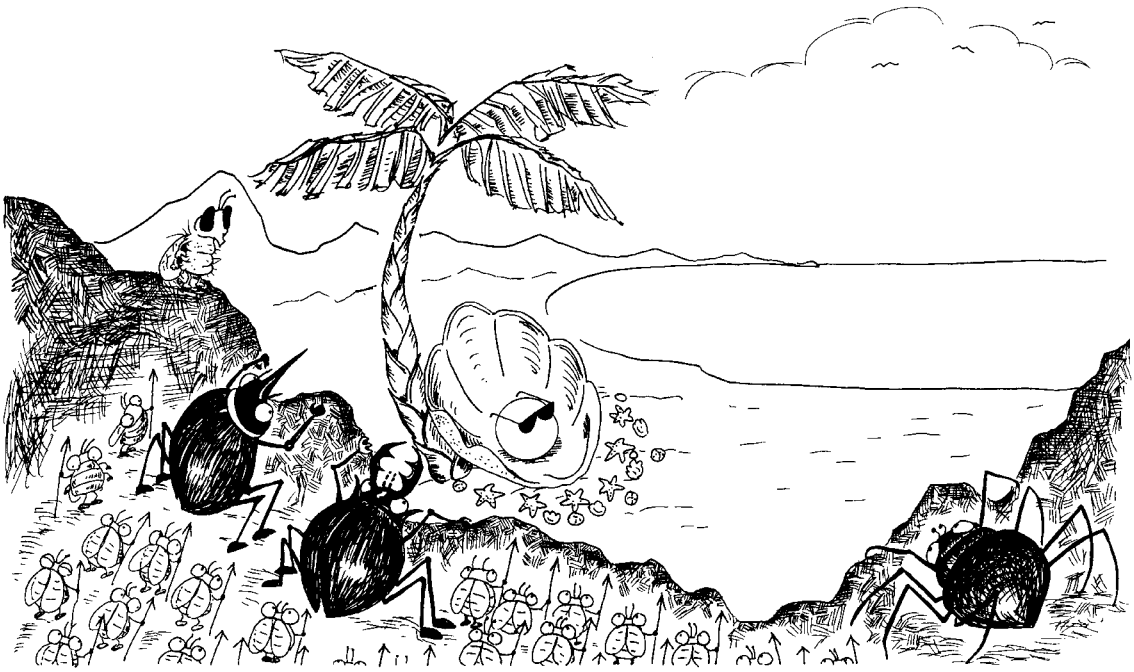
This page intentionally left blank

Contents

Preface

"It fills my head with ideas, only I don't really know what they are."

Alice in Through the Looking-Glass and What Alice Found There



Perlness

In the world of languages, the country of Perl is the great melting pot which welcomes all cultures, religions, and beliefs. "Give me your tired, your poorly-supported programmers, your huddled masses yearning to be free of artificial limitations," says Perl, and those who land on its shores find an environment where they are no longer hampered by a language designer's whimsical notions of elegant semantics and stifling syntactical purity.

Perl's universal availability and ease-of-use make it the most democratic programming language. Unlike many other languages, a relative beginner can write useful programs, whereas effective programmers in other languages normally need to spend a lot longer to learn syntax, operators, and functions. A Perl programmer may possess such expertise, or may be a newcomer who modified some example script to perform a new function.

But the newcomer has another problem: lack of debugging skills. Experience forces the canny to develop an innate knack for debugging due to years of accumulated pain. We want to minimize that pain, because we have suffered it. Perl's ease of use allows programmers with little knowledge to create usable, if fragile, code. The amount of time it takes to debug a Perl program can vary dramatically from person to person. Our goal is to help you minimize the development, debugging, and maintenance time you need for your own Perl programs.

Do not take the title of this book to imply we are debugging Perl itself in these pages. What few bugs exist in the Perl interpreter are a matter of minute exotica (or exotic minutiae), rapidly squashed by the fine volunteer crew supporting Perl. A more accurate title would have been *Debugging Your Perl Programs*, but that felt too pedestrian and loses the "unplugged" pun.

We wrote this book because we wanted you to see the development process at work. Most books on programming contain carefully crafted examples honed through sweaty practice to work perfectly and stand as mute testimonial

to the elegant style of the author. They don't show you the ugly, irritating process it took to get the examples into shape; yet those examples did not in fact spring into existence fully formed from the forehead of their creator. Because you will experience this same process when developing your programs, we want to guide you through it and describe various ways around the embarrassment, humiliation, and surprising pitfalls that stand between you and Great Programming.

Within this book, we describe the most common and annoying mistakes a new Perl programmer might make, and then detail the procedures to identify and correct those bugs and any others. You should have *some* knowledge of Perl; several fine tutorials exist to free us from the onerous responsibility of explaining scalars and arrays and hashes and the like. This preface includes a few references to some of the most useful of these tutorials.

We will not attempt to define or describe a proper programming "style." Style is as unique as an individual—but a few general rules create a common reference so that we can easily read each other's programs.

Neither is this a "how to program" book. Although we will probe into the mechanics and underpinnings of the general principle of programming at times, it is not our intention to inculcate a complete newcomer with the mindset of the programmer's discipline.

Who Are You?

If you've been programming in Perl anywhere from a week to a year and want to speed up your development cycle, this book is for you. We'll also address some issues related to developing in a team. This book is intended to assist those who have started learning Perl by providing practical advice on development practices.

What This Book Covers

Here's what you'll find in the rest of this book:

- *Chapter 1:* Introduction and a guided tour of the Perl documentation
- *Chapter 2:* Developing the right mindset for programming and developing effectively
- *Chapter 3:* "Gotchas" in Perl: Working your way around some of the tricky things to understand or get right in Perl programming
- *Chapter 4:* Antibugging: How to code defensively
- *Chapter 5:* How to instrument your code
- *Chapter 6:* How to test your Perl programs
- *Chapter 7:* A tour of the perl debugger: our guide to using this built-in tool
- *Chapter 8:* Types of syntax error and how to track down their causes
- *Chapter 9:* Run-time errors
- *Chapter 10:* Semantical errors: When your program appears to work but doesn't do the right thing
- *Chapter 11:* How to improve the performance of a resource-hungry (memory, CPU cycles, and so on) program
- *Chapter 12:* Tips and pitfalls for people coming to Perl from other languages
- *Chapter 13:* Common Gateway Interface (CGI) programming: special tips for debugging this type of Perl program
- *Chapter 14:* Conclusion
- *Appendix A:* Reference for the Perl debugger commands
- *Appendix B:* List of our "Perls of Wisdom"

We will spend a lot of time going through examples of problems and how you might debug them.

Getting Perl

While this isn't a book about how to install or build perl,[1] we owe you at least rudimentary instructions on how to get a perl of your own.

- For Windows machines, get the free ActivePerl distribution:
  `http://www.activeState.com/ActivePerl/download.htm`
- For Macintoshes:
  `http://www.cpan.org/ports/index.html#mac`
- For binary distributions for all other machines:
  `http://www.cpan.org/ports/`
- For the source of perl itself:
  `http://www.cpan.org/src/`

Building perl from source on a supported Unix architecture requires just these commands after you download and unpack the right file:

```
./Configure
make
make test
make install  # if the make test succeeds
```

The `Configure` step asks you zillions of questions, and most people won't have a clue what many of those questions are talking about; but the default answers `Configure` recommends are usually correct.

1. That's not a typo. By convention, big-P Perl refers to the language in the abstract, whereas little-p perl refers to the program that runs Perl programs.

For educational purposes, you may want to build a perl that has debugging enabled. (Here we refer to a perl that lets you use the special `-D` flag to enable the output of information that tells you what perl is doing with your program. This has nothing to do with Perl's built-in interactive debugger—which we discuss in Chapter 7—all perls have that.) If you want to do that, build perl from the source, and when `Configure` asks, "`Any additional cc flags?`" paste in whatever it already shows between brackets as a default and add "`-DDEBUGGING`". See the `perlrun` POD page (explained later) for more information.

We occasionally refer to modules that are not part of the core Perl distribution but that can be found on the Comprehensive Perl Archive Network (CPAN). For instructions on how to find, download, and install a module from CPAN, see `http://www.cpan.org/misc/cpan-faq.html`.

Typographical Conventions

We use the following conventions in this book:

- Standard text: Times Roman
- Author's comments: Arial
- Code examples and URLs: `Courier`
- User input: **`bold Courier`**

Sometimes our code examples have line breaks where none existed in the original. In places where these line breaks would cause problems or aren't obvious, we've put a backslash (\) at the end of the line to indicate that the line should be joined with the next one.

Occasionally, we want you to know which of us is talking to you, so we have this style of comment to let you know. (This is Ed, by the way.)

Those marginal icons are part of the terrific artwork created by my wife's sister, Ann Palmer, for this book. (This is Peter, by the way.)

For Further Reference

Visit this book's Web site at `http://www.perldebugged.com`.

Get introductions to Perl programming from the following (in rough order of usefulness):

- *Learning Perl,* 2nd ed., by Randal Schwartz and Tom Christiansen (O'Reilly & Associates, 1997)
- *Programming Perl,* 3rd ed., by Larry Wall, Tom Christiansen, and Jon Orwant (O'Reilly & Associates, 2000)
- *Perl, the Programmer's Companion,* by Nigel Chapman (John Wiley & Sons, 1998)
- *Elements of Programming with Perl,* by Andrew Johnson (Manning Publications, 1999)
- *Effective Perl Programming,* by Joseph Hall with Randal Schwartz (Addison-Wesley, 1998)

Perl Versions

In this book, we refer to the latest "stable" version of Perl, which is 5.6.0 as of this writing. The vast majority of what we say works unaltered on older versions of Perl 5, but not Perl 4. If you use any version of Perl older than 5.004_04, you should upgrade; 5.003 had issues such as security problems and memory leaks. You can find out the version number of your perl by passing it the `-v` flag:

```
% perl -v
This is perl, v5.6.0 built for i586-linux
Copyright 1987-2000, Larry Wall
[...]
```

Perl won't execute a script named on the command line if the `-v` flag is present. A more detailed description of your perl's configuration can be obtained with the `-V` flag; if you issue a bug report, the facility for doing that automatically includes this information with your report.[2]

A separate development track exists for Perl; you will know if you have one of those versions because the release number either contains an underscore followed by a number of 50 or larger or contains an odd number between two dots. Nothing is guaranteed to work in such a distribution; it's intended for testing. If you find you have one and you didn't want it, the person who downloaded your perl probably visited the wrong FTP link.

It was announced at the fourth annual Perl Conference (Monterey, California, July 2000) that Perl *6* development was beginning in earnest, and backward compatibility need not stand in the way of doing good things. As of press time, discussion continues on new language features.

2. Using the `perlbug` program included with your perl distribution.

Acknowledgments

I would like to thank David Noble for the original version of the `proxylog` tool and Ilya Zakharevich for educating me on the intricacies of the Perl debugger. My greatest thanks go to my wife Grace for her love and support through this intensely busy time.

As always, thanks to my grandmother, Mrs. Charles D. Wright Sr., who let me play with computers and read Doc Savage books, and to Mary O'Brien, who originally approached me about this project.

In addition, we both thank our editor, Mike Hendrickson, and his colleagues, Julie Debaggis and Marcy Barnes, for their tireless support, patience, and understanding while they took a risk on two newcomers to the field and gently ushered us into the fold of authordom. Among the rest of the tireless and often anonymous cast at Addison-Wesley we thank Cathy Comer, Mary Cotillo, Jacquelyn Doucette, John Fuller, Karin Hansen, Katie Noyes, Heather Olszyk, and Heather Peterson.

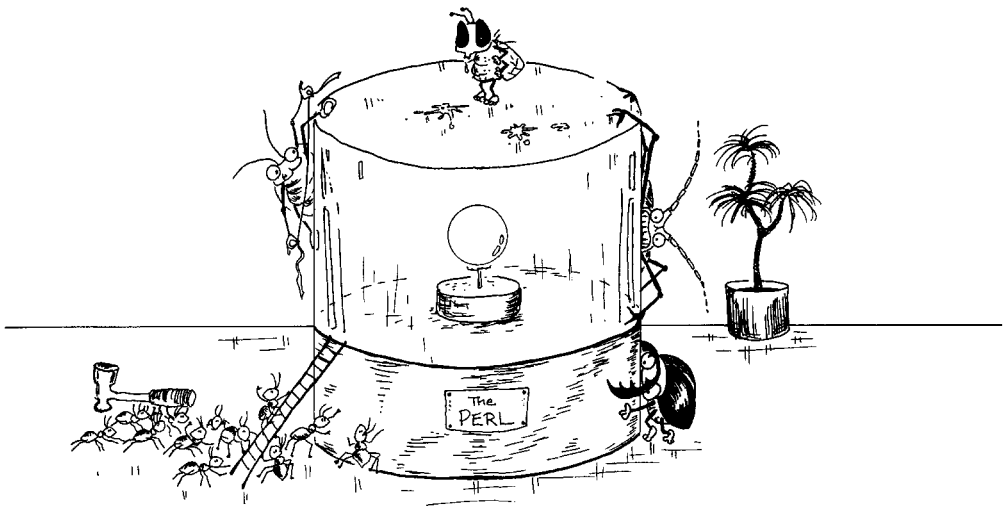
Our thanks also go to Elaine Ashton for reviews and screenshots, and to Brad Appleton, Sean M. Burke, Joseph Hall, Jarkko Hietaniemi, Michael G. Schwern, and other reviewers for their insightful and helpful comments, which have improved this book immeasurably. Any remaining errors are in no way attributable to them.

This page intentionally left blank

Chapter 3

Antibugging

"It is a capital mistake to theorize before one has data.
Insensibly one begins to twist facts to suit theories
instead of theories to suit facts."
Sherlock Holmes in A Scandal in Bohemia, *by Sir Arthur Conan Doyle*



As you can tell from the chapter title, we're not above coining horrible neologisms to save space. (When our publisher pays us by the word, we'll be happy to be more verbose, but paying us by the word would be as sensible as measuring programmers' performance by how many lines of code they write.)

Antibugging describes a set of programming techniques that minimize the probability of introducing a bug to a program at design time or input. You could also call it *defensive programming*. We're applying the principles we developed in the previous chapter to come up with some good Perl development behaviors.

3.1 Beginning

Before you turn your ideas into code, describe them in natural language.

5 **If you can't say it in natural language, you won't be able to say it in Perl.**

The Extreme Programming methodology (mentioned in more detail in Chapter 6) is helpful in this respect in that by insisting that tests be written before the code they test, and requiring programming in pairs, it encourages this kind of conversation.

I've seen this too many times: someone is having horrible difficulties doing an exercise and the code is bleeding all over the screen. Yet when I ask them to describe how it's supposed to work, they can't articulate it. This is a large clue that the approach wasn't completely thought out to begin with.

3.2 Writing Code

Your code should be as pleasing to the eye as possible, if for no other reason than you should like looking at it and not recoil at the prospect of again diving into an eructation of characters on the screen. And let's face it, programs need all the aesthetic help they can get because their primary purpose is utilitarian. Which of the following subroutines do you prefer?

```
sub iterate (&$$){
for ($_[1] = 0,my $i = 0;$i <= $_[2];
$_[1] = ++$i/$_[2]){&{$_[0] };
}
}
```

or

```
sub iterate (&$$)
   {
   for ($_[1] = 0, my $i = 0;
        $i <= $_[2];
        $_[1] = ++$i / $_[2])
      {
      &{$_[0]};
      }
   }
```

3.2.1 Style

The only believable comment to make about style is that you should have one. If you aren't already rabidly attached to a particular style, read the `perlstyle` manual page and adopt the one shown.

There's no excuse for not indenting properly. I have seen many people in my classes struggling with an exercise, needlessly handicapped because their code was all hugging the left margin for grim death. Regardless of how quick, short, or temporary

your code is, indenting it will save more time than it costs. (The Emacs `cperl` mode will autoindent simply by hitting the `TAB` key.)

6 Use a consistent style for each project.

A clean style is a good style, whatever style you choose. A suggestion: write your code as though you expect it to be published (and win the Turing Award)—using white space galore and ample comments.

Pick clear variable names. In general, the less often a variable appears in your code (or the smaller its scope), the longer its name should be since the reader will need more reminding of its purpose. There's nothing wrong though, with single-letter variable names for loop variables (for which they have a long and distinguished history) or mathematical or scientific quantities. (`$e=$m*$c**2` makes just as much sense as `$energy=$mass* $speed_of_light**2`; at least, it does to anyone who should be using code that computes it.)

3.2.2 Help from Your Editor

For those of you concerned about going blind counting and matching `()`, `{}`, and `[]` symbols in your programs, there are tools that can save you from the white cane. A smart editor that can help with some of the routine chores of beautifying a Perl program works wonders. If you have to reindent your code manually every time you remove an outer block, chances are you won't do it. Perl gurus are fond of stating, "Only perl can parse Perl," and as a corollary, no editor can be smart enough to lay out all possible Perl programs correctly. Fair enough; but some of them come close enough to be useful.

One of Peter's favorites is `cperl`, a Perl mode for Emacs maintained by Ilya Zakharevich. The standard Perl distribution includes an `emacs` directory that contains the file `cperl-mode.el`. If your Emacs doesn't already come

with `cperl`, copy this file to wherever you store local Emacs mode files (you can set up a location just for yourself in the `.emacs` file in your home directory); if you find another version there, determine which `cperl-mode.el` is more recent. Insert the line

```
(autoload 'perl-mode "cperl-mode" "alternate mode for editing \
Perl programs" t)
```

into your `.emacs` file to enable `cperl-mode` instead of the regular `perl-mode` that comes with Emacs.

`cperl` matches up the various `()`, `[]`, and `{}` for you.[1] It also performs syntax coloring, so if you configure Emacs to tint all strings in puce and your entire file suddenly blushes, you know you've left a quote mark out somewhere. Most usefully, it tracks nested levels of braces, and a tap of the `TAB` key takes you to the correct indentation point for the line. This catches all manner of typos, such as the dropped semicolon on line 1:

```
1  my $marsupial = 'wombat'
2      my $crustacean = 'king crab';
```

which is exposed when you hit `TAB` on line 2 and instead of lining up under line 1, the statement indents an extra level because the current statement is incomplete.

`cperl` does have a few problems, though: if you use the Perl 4 syntax for separating path components in a package qualified identifier with a single quote, it may get confused. But just change the apostrophe (`'`) to colons (`::`), and you're fine. Using certain characters (like `#` or `'`) in character classes in regular expressions confuses it too, but putting backslashes (`\`) in front of them (*backwhacking*) resolves the problem.

1. ... among other things; it also allows Emacs to function as a front end to the perl debugger. Debugging with Emacs is discussed in Chapter 7.

Other editors we like are: vim/gvim, which does an outstanding job of syntax coloring and has an excellent Windows port; and BBEdit on the Macintosh.

The subject of favorite editors is an intensely religious issue in software development, and you should expect others to disagree with your preference; as long as there are some equally heavyweight developers on your side, it doesn't matter what you pick or what anyone else says.

What should you expect from a good Perl editor? It should warn you when you have mismatched delimiters, it should automatically indent according to the style you want, and it should be as flexible and adaptable to your personal text-editing needs as possible. It should neither insert characters (e.g., invisible binary formatting hints) nor remove any (e.g., trailing spaces) on its own without warning you and giving you the option to disable this functionality selectively or globally.

3.2.3 Think Double

Whether your editor warns you of mismatched delimiters or not, here's a handy tip: enter the closing delimiter at the same time you type the opening one. Then insert what you need between them. Work from the outside in, and your chances of having a grouping typo are greatly reduced. For instance, if you plan to write something that'll end up as:

```
print table(map { Tr(map td($_), split /[:\s]+/) } <IN>);
```

(which, using `CGI.pm`, would print an HTML table from the remaining text coming from the filehandle `IN`, one row per line, forming table elements from elements separated by white space and/or colons) then your strategy for typing it would follow these steps:

```
1.  print table();
2.  print table(map {} <>);
3.  print table(map { Tr() } <IN>);
4.  print table(map { Tr(map td(), split //) } <IN>);
5.  print table(map { Tr(map td($_), split /[:\s]+/) } <IN>);
```

3.2.4 Clarity

Examine this very simple one-line program:

```
$x=1; $y=2; $z=$x+$y; print"$x+$y=$z\n";
```

This isn't difficult to understand: initialize `$x` and `$y`, set `$z` to their sum, then print all three. Suppose we trace its execution through the debugger (see Chapter 7 for a detailed discussion of the debugger).

```
% perl -de '$x=1; $y=2; $z=$x+$y; print"$x+$y=$z\n"'
main::(-e:1):$x=1; $y=2; $z=$x+$y; print"$x+$y=$z\n";
  DB<1> n
main::(-e:1):$x=1; $y=2; $z=$x+$y; print"$x+$y=$z\n"
  DB<1> n
main::(-e:1):$x=1; $y=2; $z=$x+$y; print"$x+$y=$z\n"
  DB<1> n
main::(-e:1):$x=1; $y=2; $z=$x+$y; print"$x+$y=$z\n"
  DB<1> n
1+2=3
Debugged program terminated...
  DB<1> q
```

What did we observe? The debugger prints the one-line program:

```
main::(-e:1):$x=1; $y=2; $z=$x+$y; print"$x+$y=$z\n"
```

4 times until the expected program output `1+2=3`. Why? The debugger displays the next line of code perl executes, not the next command. The program consists of four executable commands, but they're all on the same line; therefore we see the line each time the debugger executes a commend.

Recast the program to one command per line:

```
$x = 1;
$y = 2;
$z = $x+$y;
print"$x+$y=$z\n";
```

Then use the debugger. You can either put the four lines in a file or use a shell like the Bourne shell that allows you to type new lines between single quotes.

```
% perl -d dbformat.pl
main::(dbformat.pl:1):$x=1;
  DB<1> n
main::(dbformat.pl:2):$y=2;
  DB<1> n
main::(dbformat.pl:3):$z=$x+$y;
  DB<1> n
main::(dbformat.pl:4):print"$x+$y=$z\n";
  DB<1> n
1+2=3
Debugged program terminated...
  DB<1> q
```

This is easier to follow. The debugger functioned as it was designed to in both cases, but a style difference made it harder to interpret the debugger's output for the one-liner.

3.3 Observation

One of the most important skills you can hone as a developer is the power of observation. The biggest lesson to learn is how *not* to make things up. One way to learn this is to spend a stint answering calls on a help desk. Because so many of the people who call give such blatantly incomplete, inaccurate, or imaginary information, you will quickly realize the proper ways to describe a problem.

Why is this important to you as a developer? It's about better communication, and better understanding. *Better communication* because when you're working with someone else, you have to convey your ideas and experiences to them in the slow, limited medium of speech rather than in the lightning fast way in which richly expressive thoughts zip around in your head. Until development teams can use mind melds to communicate, you have to make the best of the bandwidth bottleneck called your tongue. *Better understanding* because whenever you articulate something inaccurately, even if you're the only one listening, part of you will take it at face value and then get confused.

How does this "making things up" show up? Let's say someone is running a program containing the following code for sending a report back to them:

```
open MAIL, "|$SENDMAIL $address"
  and print MAIL "Subject: Report for ". localtime, "\n\n"
  and print MAIL `/usr/local/bin/dbquery $month $day`
  and close MAIL
  or die "Unable to send to $address: $!\n";
```

But on execution they receive an empty e-mail message. On further inspection, they see the program (let's call it `frozzle`) had printed:

```
Can't exec "/usr/local/bin/dbquery": Permission denied at ...
```

At this point, they contact their system administrator and say, "Hey, the `frozzle` program doesn't have permission to deliver mail to me."

What happens? The sysadmin goes off on a wild goose chase for misset mailbox permissions, wrong firewall settings, or broken `sendmail` rules.

Well, usually not. Any sysadmin worth his or her salt will not take this problem report at face value but instead will ask these three questions:

- What did you do?
- How did the computer react?
- What were you expecting it to do instead?

And the answers to these questions require *absolutely no computer knowledge, intelligence, or guesswork whatsoever,* just observation:

- I ran the `frozzle` program.
- It printed `Can't exec "/usr/local/bin/dbquery"...` and I received a blank e-mail message.
- Usually it prints nothing, and I get a report in my e-mail.

However, today it is common for administrative layers to interpose themselves between the user and their system administrator. If the user has to call a help desk that transcribes their problem report into a trouble ticket database, then assigns a sysadmin from a pool to work on the problem, the chances are much higher the sysadmin will end up on a wild goose chase (particularly since the help desk probably assigned the ticket to the mail server group).

When you're dealing with customers, you'll train them to describe their problems in terms of answering the three questions. But even as a developer, you need to be adept at doing so yourself. When something goes wrong, know the difference between what happened and your guesses at what it means. As developers, we've often gone on goose chases much wilder than the one described earlier, all because we got some bee in our bonnet about what we thought was really happening.

3.4 Documentation

We know, real programmers don't document. "It was hard to write; it should be hard to understand." We feel your pain. However, the sanity you save may be your own. You could hop between jobs sufficiently rapidly that you never have to revisit a program you wrote more than six months ago.[2] Or you could document it so you'll understand what you did.

Reams of valuable material have been written on the subject of program documentation, most of it by people who aren't programmers or who would be considered fringe programmers at best by people who think of themselves as "real" programmers. If documentation is your nemesis, we don't intend to try and emulate the worthy people who can tell you exactly how and why you should write scads of hierarchical documents.

Because we hate it too.

---

2. But if you never write programs for your personal use, you can't consider yourself a *true* programmer.

So here, instead, is the lazy person's guide to documentation, based purely upon selfish motives. (Of course, in some programming environments, you have no choice but to write copious documentation, because management requires it. These environments are rare, in our experience.)

1. Imagine this is a program written by someone else, and you've been asked or ordered to take it over. What help do you need to understand it? As crazy as this scenario sounds, it's as close to the truth as anything else. Many times, you *will* have to do something with a program you wrote but haven't seen for more than six months, and many times, you won't remember nearly enough about how you wrote it to get back up to speed.
2. If it was quick to write and easy to understand, don't waste time documenting it. There *is* such a thing as self-documenting code, and the better you are at writing it, the less time you'll have to waste explaining yourself. Not all code is self-documenting, of course. Nevertheless, writing the following calls for some comment and scrutiny:

```
for (my $i = 0; $i <= $#a; $i++)
  {
  $b[$i] = $a[$i] * 86400;
  }
```

   Whereas in writing

```
@seconds = map $_ * $SECS_PER_DAY, @days
```

   it probably wouldn't.

3. The parts you *do* need to document are those that took you a long time to figure out. The amount of commenting accompanying any given piece of code should be proportional to the time it took to create it.

I once spent about a month getting *one number* right in a program. It was a driver for a communications board, and the number represented the combined flag settings for a rather important register. The vendor documentation was inadequate for divining the correct settings, and each attempt at ringing the combinations required many hours of exhaustive regression testing. That one number received a *lot* of commenting when we got it right.

4. It's not enough to say *how* things work, you also have to say *why*. Any time you make a choice that isn't obvious, explain it. Otherwise when you come back to it in six months, you'll be staring at that spot wondering, "Why did I do it that way? There's a much more obvious way..."

5. Comments are not enough. They may explain the how and the why of the code, but they don't say why you were doing it in the first place. Whether or not you put it in the same file as the code, somewhere you need a user guide for the person using your code, and you also need to capture the thoughts that went into the code's design. Why is there a module at this point in your class hierarchy? If you read your data from a flat file, instead of an Oracle database that everyone else uses, because your data file contains one more attribute than the database captures, say so. If it took thought to come up with, the thought needs to be captured somewhere.

Perl provides a couple of ways of documenting as you code, and in the ideal place for documentation—right next to the code itself.[3] The first way is the old dependable comment-to-end-of-line, used for informing a code maintainer of the operation of tricky code or why certain nonobvious choices were made. The second way is `POD`—plain old documentation—a rudimentary system of

3. Brad Appleton terms the reason that this is advantageous the *locality principle of reference documentation* and has an extensive discussion on it at `http://c2.com/cgi/wiki?LocalityOfReferenceDocumentation`.

embedding interface documentation in the program itself, which can later be extracted into Unix man pages, HTML, or other formats by programs included with perl.[4]

Comment the hard parts. Use POD.

This documentation is for the user of the program or module, someone who might be lucky enough never to need to look at your actual code. The advantage of putting this documentation in the same file as the code is twofold: it reduces the number of files you need to ship, and it enables you to put external documentation as close as you like to the actual code it refers to. (Although not out of sequence—`POD` is like a gap-toothed backwoods cousin of `WEB`, Donald Knuth's "literate programming" tool which was flexible enough to do that reordering and capable of producing wonderfully typeset annotated versions of your program. Unfortunately, the `WEB` user also had to learn TEX and learn how to embed TEX and program code in a hieroglyphic soup that made TECO macros read like Dr. Seuss. But enough nostalgia.)

`POD` is almost certainly the best choice for your user documentation, because people expect to find it in Perl files.

To be blunt, a proper program is defined by the set consisting of the executable code, comments that describe the how and why of the code, and documentation describing the function of the program (with examples). Real men and women comment their code!

4. Look for the programs `pod2man`, `pod2html`, or `pod2`anything-else-you-might-want-to-use.

3.5 Developing

There's no substitute for knowledge. Coding something you don't understand should set off warning bells in your brain: "What does the code I just wrote really do?" That's the time to experiment.

I have seen too many people take a shotgun approach to coding: they just fire a blast of characters onto the screen and rearrange them until something works. Too often, instructors foster the mindset that the only thing that counts is getting the right answer: I keep finding people who put something in a program "because it works"—in *that* particular situation—but don't understand what they are doing and so can't extend it to other situations or aren't aware of how it might fail.

Nevertheless, there are times when you have to do something quite beyond your current ability or the rate at which you can learn something new. Here are some useful ways to cheat:

1. **Steal.** Tom Christiansen and Nathan Torkington have gone and left a pile of code in a fat book[5] at your local bookstore which you can use for anything you want and they'll never complain. If you find the task you're trying to solve in there, have at it. You can always break your overall task into smaller logical units that are more generic and therefore more likely to be in the book.
2. **Mutate.** Start by using good code that does something similar to what you want, and gradually change it to fit your needs. Even though there may be code you don't understand in there, it may stay isolated from your changes, so you don't have to worry about it. (Anyone who has subclassed a module is familiar with

5. *The Perl Cookbook* (O'Reilly & Associates, 1998).

this way of thinking.) How do you recognize good code? Emulate the best.

Now, whenever we've asked who to emulate, people have been reluctant to name anyone in particular. We applaud their sense of fairness but the result is frustrating. Therefore we present our totally biased, incomplete, but still useful list of authors whose code you can find on the net, the copying of which will likely lead to good results.[6] We've put the names in alphabetical order to avoid any more outrage than necessary.

Gisle Aas
Greg Bacon
Graham Barr
Tim Bunce
Alan Burlison
Nigel Chapman
Tom Christiansen
Damian Conway
Simon Cozens
Mark-Jason Dominus
Jan Dubois
`brian d foy`
Uri Guttman
Joseph N. Hall
Jarkko Hietaniemi
Nick Ing-Simmons
Ronald Kimball
John Macdonald
Tad McClellan
Chris Nandor
Jon Orwant
Tom Phoenix
John Porter
Larry Rosler
Chip Salzenberg
Gurusamy Sarathy
Randal Schwartz
Michael G. Schwern
Lincoln Stein
Mike Stok
Nathan Torkington
Hugo van der Sanden
Johan Vromans
Larry Wall
Ilya Zakharevich

Note to the obviousness-challenged: this wasn't a popularity contest and we're not handing out awards. Of course there are dozens of other people we could have added, and hundreds more are worthy of inclusion in such a list if only we were aware of them all. (And if you're in that category, we apologize

6. However, if you see any code posted by these or other people with references to either golf or bowling, it means they're competing to see who can solve the problem in either the smallest or largest number of characters. The winner in either case is unlikely to be highly comprehensible. See `http://www.perlfaq.com/faqs/id/154`.

to you.) We print it here because it's the kind of list we wanted, and couldn't find, when we were beginning Perl. Arbitrary and incomplete as it is, we believe it to be superior to no list at all.

Now, when you're mutating code, be sure and test that the original works before you change it![7] This admonition has nothing to do with the reliability of its author, rather the fact that everyone's environment is different, and maybe the difference between yours and the author's is significant. (Sometimes we have spent ages trying to find bugs in changes we made while mutating code only to find that the problem was in the original.)

Then modify as little at a time as possible to get something you can test. Change the parts you understand the best first—like the names of input/output files and e-mail logging addresses—test, and keep going with the harder parts. With luck, you'll eventually end up with what you need without having to touch the parts you don't understand. (Anyone who's subclassed a module and had to override the constructor and other methods is familiar with this approach.)

3.6 Accident Prevention

These tips fall into the category of installing sprinklers instead of polishing your fire hose.

3.6.1 Be Strict

Put the directive `use strict` as the first line (after the `#!` line if there is one) of every file (including modules you write; strictness is lexically scoped, not program scoped). Especially if you don't understand why. This makes Perl, well, stricter about things that you could get away with without it that are nevertheless usually mistakes. It is telling that the one group of people whose expertise would most allow them to get away without using `strict` (the

7. This and some of our other tips fall under the category Joseph N. Hall termed "Don't overlook the obvious" in *Effective Perl Programming* (Addison-Wesley, 1998).

uppermost echelon of Perl experts) is the most adamant about its indispensability.[8]

Don't think it's faster to get your program working, then go back and insert the `use strict`, and clean up the resulting errors. It's much slower, for two reasons: first, developing without `strict`ness results in errors being harder to find and taking longer to fix. Second, when you add `strict`ness to a lengthy program that has never been `strict` before, the number of errors printed is likely to be enormous, and fixing them will probably not be trivial.

This is not an exaggeration, particularly if your non`strict` code has graduated to an operational environment. You may not have the option to shut things down while you recode and retest.

What makes `use strict` our friend is that most of the objections it raises prevent a program that is not following good practices from ever running. (This is called *raising a compile-time exception*. Although there is one class of errors it catches only at run-time.)

The biggest favor `use strict` does for you is to force you to declare all your variables with a `my` statement, or it will complain that you've referenced a variable without an explicit package name. In other words, variables not declared with `my` belong to a family of variables you seldom need to use; so seldom, in fact, that `use strict` requires you refer to such variables by their full names, which include the packages they live in.

Why is this such a huge favor? Because it guards against the ravages of the common typo. Consider this code:

```
my $enrolled_so_far = 0;
foreach my $student (@students)
   {
```

8. Except when they're entering the Obfuscated Perl Contest (`www.tpj.com/contest`), which relies heavily upon not using `strict`. That ought to tell you something.

```
    foreach my $class (keys %registry)
       {
       $enroled_so_far++, last
          if exists $registry{$class}{$student};
       }
    }
print "Number of enrollees so far = $enrolled_so_far\n";
```

Without `use strict`, this happily runs, producing the answer "`0`" no matter what the data. Because of the tiny typo in the innermost loop, perl increments a *different* variable called `$enroled_so_far`, which it automatically creates in this different family of variables we've been talking about. But under `use strict`, perl will refuse to run this code since `$enroled_so_far` has neither been introduced with a `my` statement nor is it qualified with a package name.[9]

Using `use strict` insulates you against a host of errors and parts of Perl that you may never need to learn.

Declare as many of your variables as possible to be lexical.

Many snippets of code in documentation or articles don't declare their variables with `my`; the authors consider it too encumbering for their examples. The code may work fine without `use strict`, but each variable is created automatically as a *dynamic* or *package* variable, which has slightly different behavior from a lexical variable and for which temporary versions are created using the old `local` keyword.

`use strict` forces you to distinguish between lexical variables and dynamic variables. There is very little reason to use the latter, or `local`, any more in Perl programs, with the exception of

- Special per-package variables like `@ISA`

9. Perl being Perl, there are always exceptions (except when there aren't), and you can prevent perl from complaining about unqualified package variables with the `our` or `use vars` pragmas.

- Localizing special variables like `$_` that can't be made lexical
- Package variables that you create and want to access from other classes for inheritance purposes

You will usually know what to do when you encounter one of these situations.

3.6.2 Let Yourself Off with a Warning

Always run your programs with the `-w` option to enable warnings. Some of these warnings come at compile time (for instance, declaring a `my` variable already declared in the same scope—you might think this would be reason enough to fail `use strict`, but Perl is curiously benevolent in this respect), and some of them come at run time because they can't be detected any earlier. (For instance, a warning that you used the value of a variable that was uninitialized or in some other way acquired the sentinel value `undef`.[10] Perl happily assigns a default value of `0` in numeric context and `""` in string context to this variable, but chances are, it's a mistake for you to be reading an undefined variable at that point.)[11]

Now, opinion is somewhat divided on the matter of leaving the `-w` flag on in operational, or delivered, code. The ayes contend that since you designed your program not to generate any warnings under `-w` (referred to as "`-w` clean"), if it ever does so, that is an error, and it ought to be visible as such. The nays retort that even if you've eliminated these possibilities, one day new warnings that your program triggers may get added to Perl, and you wouldn't want your customers' delicate sensibilities offended by Perl's sometimes off-the-wall remarks. Besides, sometimes the warnings can go to an output filter of some kind (like a Web server—we'll show how to work with errors in CGI programs in Chapter 13) and confuse it.

---

10. Another exception: the warning doesn't occur if the operation you're performing on the undefined variable is ++. If you look at some code that uses ++ like that, you'll see why this is useful.

11. If you have a variable that could legitimately acquire a value of `undef` by that point in your program, then test for it using the `defined` operator.

We squirm on the fence and eventually fall down on the side of the ayes. The best-written code can go wrong in ways we never anticipated,[12] and the best tools are those that trap all kinds of errors and package them up for the maintainers like a drawstring cat litter box liner. Find ways of redirecting warnings in production code so that the information is saved but not necessarily exposed to the user, because if something that you're doing today generates a warning in a later version of Perl, it may cause a failure in the version of Perl that follows.

Perl 5.6.0 added a feature that greatly extended the `-w` functionality: lexical warnings. `perldoc perllexwarn` tells you how you can insert variants of the `use warnings` pragma to selectively enable and disable classes of warning in lexical scopes. Because the capability does not work with versions older than 5.6.0, and this book is not concerned with fine control over warnings, we stick to the `-w` syntax.

Use `-w` and `use strict` in all your programs.

Sometimes the one-liners perl spits out courtesy of `-w` are too succinct to figure out at first glance. You can find a longer explanation of the message by looking it up in the `perldiag` manual page, or you can insert the line `use diagnostics;` in your program, which causes the longer explanation to print along with the one-liner.

Use `use diagnostics` to explain error messages.

An alternative to changing your program or browsing through `perldiag` is to use the `splain` program distributed with perl. Just filter your program's output through `splain`, and it appends the longer explanation to each error message as though you'd used `diagnostics` in the first place.

---

12. *Peter*: I once wrote a program that ran in a 24/7 real-time environment successfully for nine *years* before a bug was encountered.

3.7 Tips for Reducing Complexity

"Complexity is the enemy, and our aim is to kill it."

Jan Baan

One of Perl's greatest strengths is its expressiveness and extreme conciseness. Complexity is the bane of software development: when a program grows beyond a certain size, it becomes much harder to test, maintain, read, or extend. Unfortunately, today's problems mean this is true for every program we need. Anything you can do to minimize the complexity of your program will pay handsome dividends.

The complexity of a program is a function of several factors:

- The number of distinct lexical tokens
- The number of characters
- The number of branches in which control can pass to a different point
- The number of distinct program objects in scope at any time

Whenever a language allows you to change some code to reduce any of these factors, you reduce complexity.

3.7.1 Lose the Temporary Variables

The poster child for complexity is the temporary variable. Any time a language intrudes between you and the solution you visualize, it diminishes your ability to implement the solution. All languages do this to some degree; Perl less than most.[13] In most languages, you swap two variables `a` and `b` with the following algorithm:

13. We want to say "less than the rest," but we just *know* what sort of letters we'd get . . .

```
Declare temp to be of the same type as a and b
temp = a;
a    = b;
b    = temp;
```

But most languages are not Perl:

```
($b, $a) = ($a, $b);
```

Iterating over an array usually requires an index variable and a count of how many things are currently stored in the array:

```
int i;
for (i = 0; i < count_lines; i++)
   {
   strcat (line[i], suffix);
   }
```

Whereas in Perl, you have the `foreach` construct borrowed from the shell:

```
foreach my $line (@lines) { $line .= $suffix }
```

And if you feel put out by having to type `foreach` instead of just `for`, you're in luck, because they're synonyms for each other; so just type `for` if you want (Perl can tell which one you mean).

Because functions can return lists, you no longer need to build special structures just to return multivalued data. Because Perl does reference-counting garbage collection, you can return variables from the subroutine in which they are created and know that they won't be trampled on, yet their storage will be released later when they're no longer in use. And because Perl doesn't have strong typing of scalars, you can fill a hierarchical data structure with heterogeneous values without having to construct a union datatype and some kind of type descriptor.

Because built-in functions take lists of arguments where it makes sense to do that, you can pass them the results of other functions without having to construct an iterative loop:

```
unlink grep /~$/, readdir DIR;
```

And the `map` function lets you form a new list from an old one with no unnecessary temporary variables:

```
open PASSWD, '/etc/passwd' or die "passwd: $!\n";
my @usernames = map /^([^:]+)/, <PASSWD>;
close PASSWD;
```

Because Perl's arrays grow and shrink automatically and there are simple operators for inserting, modifying, or deleting array elements, you don't need to build linked lists and worry if you've got the traversal termination conditions right. And because Perl has the hash data type, you can quickly locate a particular chunk of information by key or find out whether a member of a set exists.

3.7.2 Scope Out the Problem

Of course, sometimes temporary variables are unavoidable. Whenever you create one though, be sure and do it in the innermost scope possible (in other words, within the most deeply nested set of braces containing all references to the variable).

Create variables in the innermost scope possible.

For example, let's say somewhere in my program I am traversing my Netscape history file and want to save the URLs visited in the last 10 days in `@URLs`:

```
use Netscape::History;
my $history = new Netscape::History;
my (@URLs, $url);
```

```
while (defined($url = $history->next_url() ))
   {
   push @URLs, $url if
             time - $url->last_visit_time < 10 * 24 * 3600;
   }
```

This looks quite reasonable on the face of it, but what if later on in our program we create a variable called `$history` or `$url`? We'd get the message

```
"my" variable $url masks earlier declaration in same scope
```

which would cause us to search backward in the code to find exactly which one it's referring to. Note the clause "`in same scope`"—if in the meantime you created a variable `$url` at a different scope, well, that may be the one you find when searching backward with a text editor, but it won't be the right one. You may have to check your indentation level to see the scope level.

This process could be time-consuming. And really, the problem is in the earlier code, which created the variables `$history` or `$url` with far too wide a scope to begin with. We can (as of perl 5.004) put the `my` declaration of `$url` right where it is first used in the `while` statement and thereby limit its scope to the `while` block. As for `$history`, we can wrap a bare block around all the code to limit the scope of those variables:

```
use Netscape::History;
my @URLs;
   {
   my $history = new Netscape::History;
   while (defined(my $url = $history->next_url() ))
      {
      push @URLs, $url
         if time - $url->last_visit_time < 10 * 24 * 3600;
      }
   }
```

If you want to create a constant value to use in several places, use `constant.pm` to make sure it can't be overwritten:

```
$PI = 3.14159265358979323384;

use constant PI => 3.14159265358979323384;

my $volume = 4/3 * PI * $radius ** 3;

$PI = 3.0;  # The 'Indiana maneuver' works!
PI  = 3.0;  # But this does not
```

In response to the last statement, Perl returns the error message, "`Can't modify constant item in scalar assignment`."

`constant.pm` creates a subroutine of that name which returns the value you've assigned to it, so trying to overwrite it is like trying to assign a value to a subroutine call. Although the absurdity of that may sound like sufficient explanation for how `use constant` works, in fact, the latest version of perl allows you to assign a value to a subroutine call, provided the result of the subroutine is a place where you could store the value. For example, the subroutine could return a scalar variable. The term for this feature is *lvaluable subroutine*. But since the results of the subroutines created by `use constant` aren't lvalues, lvaluable subroutines won't cause problems for them.

This page intentionally left blank

Index