

Chapter 1, Perl 5 Interactive Course



MEET PERL: THE BASICS

This book is designed for everyone--novices and gurus alike. We'll begin gently, with an introduction to some general programming constructs that you'll find in nearly every computer language: *if* statements, *while* loops, and so forth. But by chapter's end, you'll be able to write useful Perl programs. Admittedly, they'll be pretty simple programs, but as you'll see throughout the book, simple is usually all you need.

Some of you advanced readers are no doubt thinking to yourselves, "I've been programming C for eight years, and assembly language before that. I don't need a book to tell me how to write a *while* loop or how to think about subroutines." True. This book dispenses with the basics quickly, with thorough coverage of advanced topics such as networking, process management, and creating graphical applications from within Perl. No matter what other languages

you've programmed in, you'll find something new in each chapter, even this one: The last two lessons discuss *default variables* and *contexts*, two features of Perl that you won't find in other programming languages.

This chapter, like all 14 in this book, is divided into eight lessons, each of which has a quiz and a programming exercise. After running a simple program in Lesson 1, simple variables--*scalars*--are introduced in Lesson 2, followed by the *if* statements and *while* loops in Lessons 3 and 4. Lesson 5 introduces *arrays*, which are collections of scalars. The *for* statement is introduced in Lesson 6, and then default variables and contexts are discussed in Lessons 7 and 8. Finally, there are four additional programming exercises at the end of this (and every) chapter.

Perl is fun, powerful, useful, and (at times) quirky. Have fun, and proceed at whatever pace suits you!

THE LOOK AND FEEL OF PERL

Perl was born in 1987, when Larry Wall fused a system administration tool he had written with a UNIX utility called *awk*. Over time, new features were added: pattern matching, associative arrays, formats, a symbolic debugger, and oodles and oodles of functions. Now Perl is in its fifth incarnation (that's "Perl 5" to you), supports object-oriented programming, and is best known for systems administration, rapid prototyping, text processing, and the World Wide Web. Perl is one of the fastest-growing languages around. And getting started is easy.

Perl borrows features from a number of languages. C programmers will recognize many of Perl's functions and operators, as well as its interface to the UNIX operating system. But unlike C, Perl doesn't force you to compile binary executables that work on only one platform. In that respect, Perl resembles the UNIX shells, from which it also borrows facilities for process management and file manipulation. Perl also supports sophisticated pattern matching, which makes tasks that would take hundreds of lines in C possible in one or two lines. These components make Perl useful--and eclectic. Larry Wall concedes that PERL might well stand for Pathologically Eclectic Rubbish Lister, although his original choice was Practical Extraction and Report Language.

This book illustrates Perl concepts with working programs--lots of them. You'll learn a lot by typing them all in, but out of compassion, scripts for all the programs are available on the accompanying CD-ROM and at the Waite Group Press's FTP site ([ftp.waite.com](ftp://ftp.waite.com)).

Running a Perl Program

Let's run our first Perl program. Following that great tradition of computer language books, we'll begin with a program that prints *Hello world!*. Create a file (call it *hello*) containing the line shown in Listing 1-1.

Listing 1-1 A simple Perl program

```
print "Hello world! \n"; # prints Hello world! followed by a
newline (\n)
```

Don't forget the semicolon at the end! Every (well, nearly every) Perl statement ends in a semicolon. That's how Perl can tell where one statement ends and the next begins, so if you leave out a semicolon, Perl will complain.

Now let's run the program. Type *perl hello* at your command-line prompt if you have one, represented here as %.

```
% perl hello
```

```
RESULT Hello world!
```

If that didn't work, then Perl isn't in your path, probably because it hasn't been installed. In that case,

you'll want to retrieve it from the enclosed CD-ROM or from one of the sources listed in Appendix B.

If you want *hello* to be executable directly from the UNIX command line, so that you can just say *hello* instead of *perl hello*, you need to add one line to the beginning of your program. That line is *#!/location_of_your_Perl_binary*.

```
#!/usr/bin/perl
print "Hello world! \n"; # prints Hello world! followed by a
newline (\n)
```

(On all lines but the first, # is used to begin a comment. Use comments--liberally--to document your code so you, and others, can tell how your program works.)

On UNIX systems, you'll need to make the program executable with the *chmod* command.

```
% chmod +x hello
```

Then you can run the program just by typing its name.

```
% hello
```

RESULT

```
Hello world!
```

On a UNIX system, you may need to type *./hello* if *.* isn't in your path. (If you don't understand that, don't worry about it.)

If *hello* didn't run, that's probably because your Perl is somewhere other than */usr/bin*, where *hello* tried to find it. Perhaps your Perl is in */usr/local/bin*, or perhaps you're not on a UNIX system. (MS-DOS users: You won't be able to use the *#!* notation unless you install a utility such as 4DOS or convert your Perl program into a batch file with *pl2bat*.)

```
#!/usr/bin/perl
```

This book will use *#!/usr/bin/perl* as the first line of all its programs because Perl installs itself as */usr/bin/perl* on most UNIX systems. If Perl is somewhere else on your computer, you'll have to replace */usr/bin/perl* with the appropriate pathname.

The *instinfo* program (available from ftp.waite.com) contains a trick for invoking Perl on computers that have UNIX shells but don't recognize the *#!* notation. It'll also print out where Perl was installed. For more information, consult the *perlrun* documentation bundled with the Perl distribution.

If you wish to change the initial line of the programs from *#!/usr/bin/perl* to, say, *#!/usr/local/bin/perl*, use the *header* program, also available from ftp.waite.com.

Command-Line Flags

Command-line flags (also called switches) affect how Perl runs your program. If you read the introduction, you've seen one already: *-v*, which displays Perl's version number.

```
% perl -v
```

RESULT This is perl, version 5.004

Copyright 1987-1997, Larry Wall
Perl may be copied only under the terms of either the
Artistic License or the GNU General Public License, which may
be found in the Perl 5.0 source kit.

You'll be seeing a number of command-line flags throughout the book, and there's a full list in Appendix G. But there's one flag that you'll find especially useful: `-e`, which lets you execute Perl statements from your command line.

```
% perl -e 'print 4;'
```

RESULT 4

```
% perl -e 'print "Hello world! \n";'
```

RESULT Hello world!

MS-DOS computers might not allow single quotes after the `-e`, in which case you'll have to swap the single and double quotes:

```
% perl -e " print 'Hello world!'"
```

RESULT Hello world!

Just about anything you can do in a Perl program, you can do from the command line as well.

Command-line flags can also be placed inside programs, directly after `#!/usr/bin/perl`. Listing 1-2 shows *hello* with the `-w` flag, which warns the user about potential errors.

Listing 1-2 The *hello* program

```
#!/usr/bin/perl -w
print "Hello world! \n";
```

Because there aren't any errors in *hello*, the behavior won't change.

```
% hello
```

RESULT Hello world!

You can combine flags:

```
% perl -we 'print "Two \n lines \n "';'
```

RESULT Two

lines

Is Perl Compiled or Interpreted?

Good question. The short answer is *interpreted*, which means that your code can be run as is, without a compilation stage that creates a nonportable executable program. But this book occasionally mentions the Perl compiler. So what's going on?

Traditional compilers convert programs into machine language. When you run a Perl program, it's first compiled into bytecode, which is then converted (as the program runs) into machine instructions. So it's not quite the same as shells, or Tcl, which are "strictly" interpreted without an intermediate representation. Nor is it like most versions of C or C++, which are compiled directly into an architecture-dependent format. It's somewhere in between, along with Python and *awk* and Emacs' *.elc* files. Furthermore, there is a separate utility called the Perl Compiler, which you can use to make your own architecture-dependent binary, should you choose.

Chapter 1, Perl 5 Interactive Course

LESSON 2

SCALARS

Now let's make the example a bit more complex by introducing our first variable:

\$greeting. The *\$* tells Perl (and you) that *greeting* is a scalar variable, such as the number 5.4 or the string Hello. Other data types use different symbols; for example, arrays (which you'll learn about later in this chapter) always begin with a *@* and hashes (the subject of Chapter 5) always begin with a *%*.

The *hello2* program, shown in Listing 1-3, stores the string Hello in the scalar variable *\$greeting*. When *print()* displays its string, it replaces *\$greeting* with *Hello*, yielding the same output as the *hello* program.

Listing 1-3 *hello2*: Printing the value of *\$greeting*, a scalar

```
#!/usr/bin/perl -w
$greeting = 'Hello'; # Set the value of $greeting to 'Hello'
print "$greeting world!\n"; # and print it, followed by "
world!\n"
```

Let's run *hello2*, just to be sure.

```
% hello2
```

```
RESULT Hello world!
```

Double Quotes and Single Quotes

Eagle-eyed readers might have noticed a small difference between the strings on the second and third lines of the program. The second line uses single quotes (**'Hello'**), whereas the third line uses double quotes (**"\$greeting world!\n"**).

Perl performs *variable interpolation* within double quotes, which means that it replaces a variable's name with its value. Single quotes don't perform variable interpolation, which is why

```
$greeting = 'Hello';
print '$greeting';
```

prints *\$greeting*, whereas

```
$greeting = 'Hello';
print "$greeting";
```

prints *Hello*.

Double quotes do something else that single quotes don't do: They interpret backslashed characters. Inside double quotes, *\n* means newline, but inside single quotes, it remains a slash-in-front-of-an-*n*.

```
print "Hello, world!\n\n\n";
```

prints *Hello, world* followed by two blank lines.

```
print `Hello, world!\n\n\n`;
```

prints *Hello, world!\n\nn*. (Oops!)

"Plain" strings can be either double-quoted or single-quoted.

```
$string = "Hello";
```

```
$greeting = `Bonjour`;
```

But back to scalars. A scalar can be a string:

```
$greeting = `Bonjour`;
```

```
$salutation = "Hello, world!\n";
```

or a number:

```
$number = -5;
```

```
$e = 2.71828;
```

You can name a scalar anything (well, nearly anything) you want, but it should begin with \$, followed by a letter or an underscore, followed by more letters, digits, or underscores.

Like C and UNIX, Perl is case-sensitive, so capitalization matters. The following are all different (but legal) scalars:

```
$Greeting
```

```
$GREETING
```

```
$gReEtING
```

```
$greeting
```

You can make variable names as long as you wish.

Functions

print() is a Perl *function*. Most functions accept one or more arguments (or parameters) and produce a single return value. Some functions cause something else to happen--a *side effect*. In the case of print(), it's the side effect that's important: writing text to your screen.

In C, functions require parentheses around their arguments, as in **printf("Hello!\n")**. In Perl, parentheses are needed only to disambiguate between alternate interpretations. (That has to do with the relative precedence of Perl's operators. There's more on operators later in this chapter and a table of their precedences in Appendix F.)

Parentheses are seldom used with print(). Sometimes you can even omit the quotes, as in *hello3*, featured in Listing 1-4.

Listing 1-4 *hello3*: Quotes aren't always necessary

```
#!/usr/bin/perl -w
```

```
$greeting = `Hello`;
```

```
print $greeting;
```


print() sees just one argument: the scalar variable \$greeting, which contains the string *Hello*.

```
% hello3
```

```
RESULT Hello
```

Of course, *hello* and *hello2* print *Hello, world*, not just *Hello*. Luckily, print() can accept multiple arguments, so you can print both *Hello* and *world* with one statement.

Perl separates arguments with commas. In *hello4* (Listing 1-5), three arguments are supplied to print(): \$greeting, ' world!', and "\n".

Listing 1-5 *hello4*: Using commas to separate function arguments

```
#!/usr/bin/perl -w
$greeting = 'Hello';
print $greeting, ' world!', "\n";
% hello4
```

```
RESULT Hello world!
```

Now let's make an interactive program, one that asks the user to enter some text. Listing 1-6 shows a short example.

Listing 1-6 *spitback*: Grabbing user input with <>

```
#!/usr/bin/perl -w
print 'Enter a greeting: ';
$greeting = <>; # (Now the user enters a line of text)
print $greeting;
```

The angle brackets (<>, often called the "diamond operator") access what the user types. Don't worry about how it works--you'll learn more about that in Chapter 3. But it sure is handy, isn't it? (You can use it to access command-line arguments, too.)

```
% spitback
Enter a greeting: Howdy!
Howdy!
% spitback
```

```
RESULT Enter a greeting: I can type anything I want...
```

```
I can type anything I want...
```

You can use <> to query the user more than once. The *madlibs* program, shown in Listing 1-7, asks the user for a number, a noun, and a verb, and then combines all three into a single phrase. It also uses another Perl function, chomp(), and an operator, ., to join strings together. Note the similarity between the three chunks of code that grab and manipulate user entries.

Listing 1-7 *madlibs*: Retrieving and manipulating user-typed text

```
#!/usr/bin/perl -w

# Ask the user to type a number. Place the result in $num,
# chopping off the newline (created when the user pressed
# [ENTER]).

print 'Pick a number between 1 and 10: ';
$num = <>;
chomp $num;

# Now do the same thing for $noun.

print 'Gimme a noun: ';
$noun = <>;
chomp $noun;
$noun = $noun . 's'; # Pluralize noun by appending an 's'

# And do the same thing for $verb.

print 'Now type a verb: ';
$verb = <>;
chomp $verb;
$verb = 'a-' . $verb . 'ing'; # Add a prefix and a suffix to
$verb

print "\n$num $noun $verb\n";
```

When *madlibs* is run, the user is asked to provide a number, then a noun, and finally a verb. *madlibs* waits until the user presses [ENTER] before proceeding to the next statement.

% **madlibs**

RESULT Pick a number between 1 and 10: 10

Gimme a noun: lord
Now type a verb: leap
10 lords a-leaping

Notice how the three variables \$num, \$noun, and \$verb are all processed the same way, even though \$num is a number and \$noun and \$verb are strings. To Perl, they're all just scalars.

The **chomp()** and **chop()** Functions

In *madlibs*, every <>; was followed by a **chomp()**. That's because every time you press [ENTER], you append a newline to \$num or \$noun or \$verb. The **chomp()** function removes the newline from the end of its argument.

If you hadn't chomped your variables, the output would have looked like this:

10
lord

```
s a-leap  
ing
```

You could also have used `chop()` instead of `chomp()`. The `chop()` function is simple--it indiscriminately lops off the last character of a string. `chomp()` is a bit more selective--it removes the last character only if it's a line separator. (That's usually a newline, but you can define it to be anything you want with the `$/` variable, described in Chapter 6, Lesson 4.)

```
$string = "This has a newline:\n";  
chomp $string;
```

(Now `$string` is **This has a newline:.**)

```
$string = "This has a newline:\n";  
chop $string;
```

(`$string` is also **This has a newline:.**)

```
$string = "This does not";  
chomp $string;
```

(`$string` remains intact: **This does not.**)

```
$string = "This does not";  
chop $string;
```

(`$string` is now **This does no.**)

`chop()` returns the character lopped off (if any). That's why it's critical NOT to do this:



```
string = "This has a newline:\n";  
$string = chop $string;
```

because that sets `$string` to a solitary `\n`, annihilating everything else.

`chomp()` returns the number of characters that were lopped off. That might be 0, 1, or even more (but only if you've changed `$/`).

Because `chomp()` and `chop()` are functions, you can surround their arguments with parentheses if you want: `chomp($num)` is the same as `chomp $num`.

Operators

You can think of *operators* as functions that expect arguments in strange places, instead of in the comma-separated list required by normal functions such as `print()` or `chomp()` (Figure 1-1). You've seen a couple of operators already: the `=` operator, which expects arguments on its left and right:

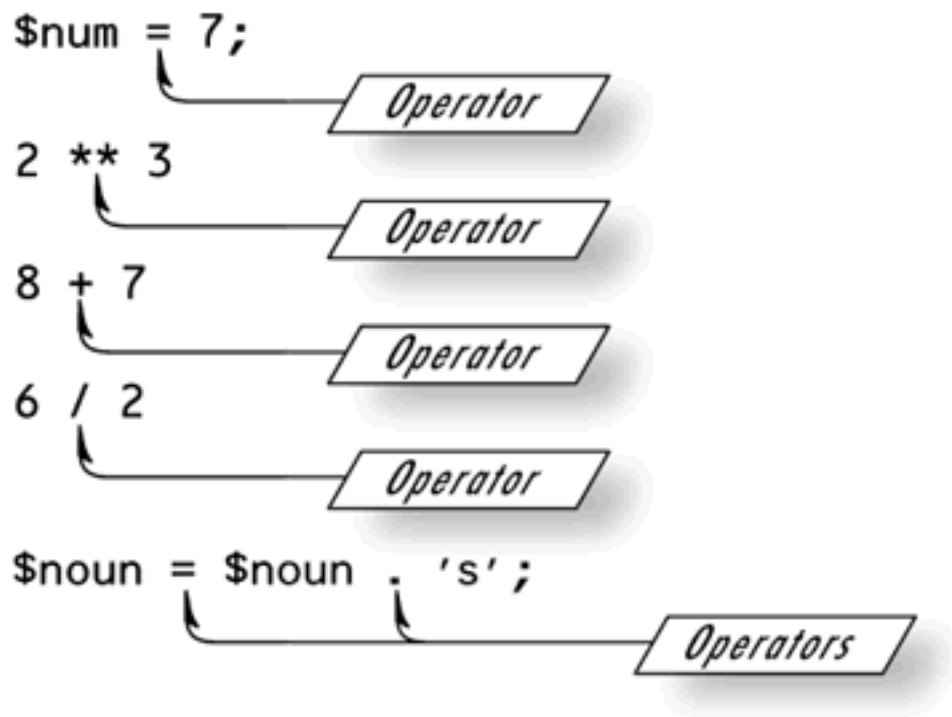


Figure 1-1
Operators

```
$x = 7
```

just like the arithmetic operators:

```
+ - * /
```

which behave as you might expect.

```
8 * 2
```

is 16, and

```
5 / 3
```

is 1.6666666666666667.

This statement employs two operators, = and +:

```
$x = 3 + 4
```

setting \$x to 7.

You've also seen the string concatenation operator (.), which glues two strings together. The *madlibs* program uses the . operator to turn *lord* into *lords*.

```
$noun = $noun . \'s\';
```

To make our leap into a-leaping, madlibs uses . twice.

```
$verb = \'a-\' . $verb . \'ing\';
```

Another operator is *x*, which expects a string (or an array--more about those in Lesson 5) on its left and a number on its right and duplicates the string (or the array) that many times.

```
$word = "ba" x 4;
```

sets \$word to *babababa*.

```
8x8
```

yields 88888888.

```
$a = ("rs" x 2 . "t") x 3;  
print $a;
```

prints *rsrstrsrstrsrst*.

Of course, not all operators take two arguments:

```
-7
```

might look like a simple number. Well, it is. But it's also the unary operator (-) applied to the argument 7.

Chapter 1, Perl 5 Interactive Course

LESSON 3

THE *IF* STATEMENT

Every programming language has some form of the *if* statement, which allows you to create a conditional statement that might or might not be executed. Until now, every program you've seen has executed all its statements in sequence. That's okay for simple tasks, but more complicated problems might require the program, say, to print something only if some condition is true and to do nothing otherwise.

Suppose you want a program to ask the user whether he or she wants to see a movie. If the user says yes, your program should ask the user for money. In other words:

```
if (the user types 'yes') { ask for money }.
```

Listing 1-8 features *tickets*, which shows how you use an *if* statement.

Listing 1-8 *tickets*: Using an *if* statement

```
#!/usr/bin/perl -w
print 'Would you like to see the 11:00am showing of
Casablanca? ';
$answer = <>;
chomp $answer;
if ($answer eq 'yes') { # If the user typed yes,
print "That'll be \$8.50, please.\n"; # ask him for money.
}
```

There are three new concepts in *tickets*: the *if* statement, the *eq* operator, and the backslashed character *\\$*. We'll tackle them in that order.

The *if* Statement

if (*CONDITION*) { *BODY* } executes *BODY* if the *CONDITION* is true.

Some examples of *if*:

```
if ($answer eq 'yes') { print 20; }
```

If *\$answer* is equal to *yes*, then *20* is printed.

```
if ($answer ne 'yes') {
$answer = 'maybe';
}
```

ne is the opposite of *eq*: If *\$answer* is not equal to *yes*, then *\$answer* is set to *maybe*.

In *tickets*, the user is asked for money only if he or she types *yes*. The condition is (**\$answer eq 'yes'**) and the *BODY* consists of a solitary `print()` statement. When an `if` statement is used in this way, it requires parentheses around the condition and curly braces around the body.

Here's a different way to use *if*:

```
#!/usr/bin/perl
print 'Would you like to see the 11:00am showing of Cape
Fear? ';
$answer = <>;
chomp $answer;
print "That'll be \$8.50, please.\n" if $answer eq 'yes';
```

Both programs behave exactly the same.

Alternative Syntaxes for `if`

STATEMENT *if* ***CONDITION*** is the same as `if (CONDITION) { STATEMENT }`.

STATEMENT *unless* ***CONDITION*** executes ***STATEMENT*** if ***CONDITION*** is FALSE.

As you'd expect,

```
print 'Hello' if $answer eq 'yes';
```

prints *Hello* if `$answer` is equal to *yes*.

```
$answer = 'maybe' if $answer ne 'yes';
```

sets `$answer` to *maybe* if `$answer` is not equal to *yes*, which is the same as saying

```
$answer = 'maybe' unless $answer eq 'yes';
```

Use whichever syntax you wish. That said, it's often a Good Thing to present readers with the important clause first.

```
print "Phase 1---Checking Carbon Monoxide levels" if $debug;
if ($sea_levels eq "rising") { $anxiety += 17; }
```

are easier to understand than

```
if ($debug) { print "Phase 1---Checking Carbon Monoxide
levels"; }
$anxiety += 17 if $sea_levels eq "rising";
```

One of the tenets of Perl is "There's more than one way to do it." You'll see this again and again throughout the book.

Expressions

An *expression* is anything that, when evaluated, yields some value (Figure 1-2). The condition of the *if* statement (**\$answer eq 'yes'**) is an expression, as are these:

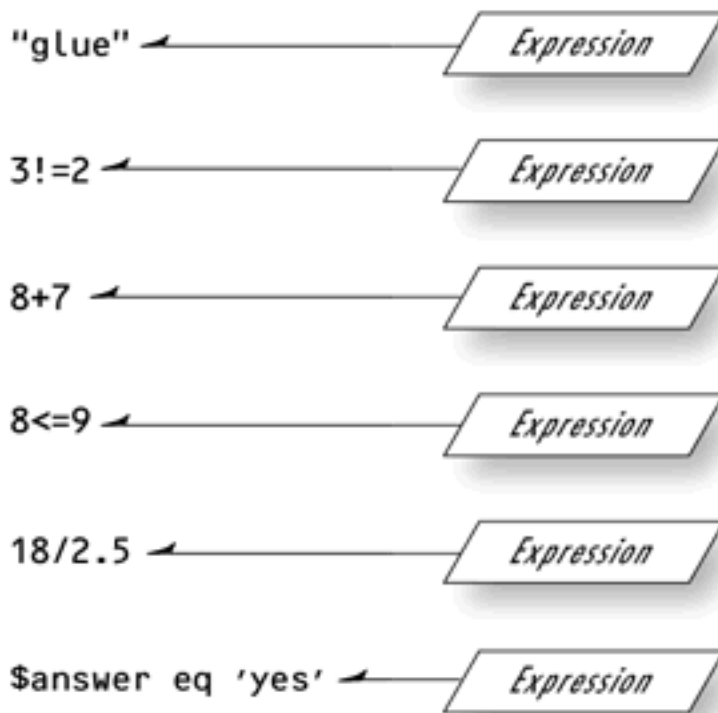


Figure 1-2

Expressions

4 * 5

evaluates to 20,

\$answer

evaluates to the value of \$answer, and

chop \$answer

evaluates to whatever character was lopped off from \$answer.

Every expression yields a particular value--usually a number or a string. The expression `4 * 5` yields 20, as you'd expect. But assignments are expressions, too: `$answer = <>` yields \$answer, once `<>` has been assigned to it. That's why you'll often see

```
$answer = <>;
```

```
chomp $answer;
```

written as

```
chomp($answer = <>);
```


TRUE and FALSE Values

All values are either TRUE or FALSE. The values shown in Table 1-1 (and no others) are FALSE.

Table 1-1 FALSE values

Example Value	Description
0	The number
""	An empty string
"0"	A string containing just the 0 character
()	An empty array--introduced in Lesson 5
Undefined values	All variables that haven't been assigned values

All other values are TRUE. Uninitialized variables are set to a FALSE value, so

```
print "false!" unless $uninitialized_variable;
```

prints *false!*, as do

```
print "false!" unless 0;
```

and

```
print "false!" unless "";
```

So (`$answer eq 'yes'`) will either be TRUE or FALSE, depending on the value returned by the *eq* operator. *eq* operates on strings, in this case `$answer` and `'yes'`, returning TRUE if the strings are equal and FALSE if they're not. So if `$answer` is *yes*, the condition evaluates to TRUE and the `print()` statement inside the curly braces is executed.

You'll often see *eq* used in just this way, as the condition of an *if* statement.

```
% perl -e ' if ("yes" eq "yes") { print "true\n"; } '
```

RESULT true

```
% perl -e ' if ("yes" eq "no") { print "true\n"; } '
```

prints nothing.

If you want to test the equality of numbers rather than strings, use the `==` operator, instead of *eq*.

```
% perl -e ' if (6 == 6) { print "true\n"; } '
```

RESULT true

```
% perl -e ' if (6 == 5.2) { print "true\n"; } '
```

As you've seen, *ne* is the opposite of *eq*. `!=` is the opposite of `==`.

```
% perl -e ' if ("yes" ne "no") { print "true\n"; } '
```

RESULT true

```
% perl -e ' if (6 != 5.2) { print "true\n"; } '
```

RESULT true

```
% perl -e ' $x = 8; if ($x == 8) { print "true\n"; } '
```

RESULT true

One common error (possibly the most common) among novice Perl programmers is using `==` when `=` is called for, and vice versa. Don't do it! Remember:

`==` Versus `=`

`==` *tests* two numeric values; `=` *assigns* a value to a variable.

`==` Versus `eq`

`==` tests *numbers*; `eq` tests *strings*.

You compare quantities in various ways. The arithmetic comparators, shown in Table 1-2, should all be familiar.

Table 1-2 Numeric comparators

< tr>

The Expression...	...Evaluates to TRUE If
<code>\$x > \$y</code>	The number <code>\$x</code> is greater than the number <code>\$y</code> .
<code>\$x < \$y</code>	The number <code>\$x</code> is less than the number <code>\$y</code> .
<code>\$x >= \$y</code>	The number <code>\$x</code> is greater than or equal to the number <code>\$y</code> .
<code>\$x <= \$y</code>	The number <code>\$x</code> is less than or equal to the number <code>\$y</code> .

There are analogous string comparators, as shown in Table 1-3.

Table 1-3 String comparators

The Expression...	...Evaluates to TRUE If
<code>\$word1 gt \$word2</code>	The string <code>\$word1</code> comes after the string <code>\$word2</code> .
<code>\$word1 lt \$word2</code>	The string <code>\$word1</code> comes before the string <code>\$word2</code> .
<code>\$word1 ge \$word2</code>	The string <code>\$word1</code> comes after, or is the same as, <code>\$word2</code> .
<code>\$word1 le \$word2</code>	The string <code>\$word1</code> comes before, or is the same as, <code>\$word2</code> .

Question: Is *Hello* less than or greater than *hello*? Or are they equal? The answer might surprise you: *Hello* comes before *hello* because string comparators don't use dictionary order. They use ASCII order, in which uppercase letters precede lowercase letters. (There's a table of ASCII characters in Appendix C.)

Backslashed Characters

You've learned that double quotes interpolate scalars and that scalars can be identified by their dollar sign. Sometimes you'll just want to print a plain dollar sign, using a backslash to prevent Perl from trying to interpolate variables.

```
print "That'll be \$8.50, please.\n";
```

See that backslash in front of the dollar sign? It's there to keep Perl from thinking that \$8 is a variable. If Perl sees \$8.50, it interpolates \$8, which is undefined, yielding

```
That'll be .50, please.
```

Because you want to print the dollar sign, you need to let Perl know by backslashing it. There's a simple rule about when to backslash characters inside double-quoted strings: Backslash weird characters (@, \$, and so on) to get their literal meanings, and backslash normal characters (*n*, *r*, *t*, and so on) to get their weird meanings (linefeed, carriage return, and tab, respectively).

if...else Statements

Now let's make the *if* statement a bit more complex by adding an *else* clause, as shown in Listing 1-9.

Listing 1-9 *tickets2*: An *if...else* statement

```
#!/usr/bin/perl -w
print "Would you like to see the 11:00am showing of Cape
Fear? ";
$answer = <>;
chomp $answer;
if ($answer eq 'yes') {
    print "That'll be \$8.50, please.\n";
} else {
    print "Okay--but you're missing a great movie. Have a nice
day!\n";
}
```

The *if* statement in *tickets2* contains an *else* clause. As before, if (`$answer eq 'yes'`), the program asks for money. But if that condition isn't satisfied, the other `print()` statement executes. The statements following an *else* clause must be enclosed within curly braces, just like the statements following the condition.

```
% tickets2
Would you like to see the 11:00am showing of Cape Fear? no
Okay--but you're missing a great movie. Have a nice day!
```

tickets2 assumes that the user typed *no* if he or she didn't type *yes*. What if you want your program to complain if the user said neither yes nor no? In other words:

```
if (the user types yes) { ask for money }
```

otherwise if (the user types no) { say okay }

otherwise { say huh? }

Then you need an *if...elsif...else* statement, shown in Listing 1-10.

Listing 1-10 *tickets3*: An *if...elsif...else* statement

```
#!/usr/bin/perl
print 'Would you like to see the 11:00am showing of Cape
Fear? ';
$answer = <>;
chomp $answer;
if ($answer eq 'yes') { # If 'yes' was typed...
print "That'll be \$8.50, please.\n";
} elsif ($answer eq 'no') { # Otherwise, if 'no' was typed...
print "Okay--but you're missing a great movie. Have a nice
day!\n";
} else { # Otherwise...
print "I'm sorry, I didn't understand that.\n";
}
```

You can have as many *elsif* clauses as you want. *tickets3* just needs one because there are only three possible outcomes (*yes*, *no*, and everything else), and two of those are caught by the *if* clause and the *else* clause.

```
% tickets3
```

```
RESULT Would you like to see the 11:00am showing of Cape Fear? maybe
I'm sorry, I didn't understand that.
```

Assignment Operators

Assigning values to variables is usually done with the = sign, as above with **\$answer** = <>. However, programmers frequently perform some operation just prior to the assignment, such as this string concatenation:

```
$noun = $noun . 's'
```

or this addition:

```
$num = $num + 2
```

In these cases, you can often combine the operators by saying

```
$noun .= 's'
```

or

```
$num += 2
```

Many operator-assignment combinations can be abbreviated, as shown in Table 1-4.

Table 1-4 Some assignment operators

Statement	Equivalent to	Meaning
<code>\$noun .= 's'</code>	<code>\$noun = \$noun . 's'</code>	Appends an 's' to \$noun
<code>\$string x= 3\$</code>	<code>string = \$string x 3</code>	Replaces \$string with three copies of itself
<code>\$num += 7</code>	<code>\$num = \$num + 7</code>	Increments \$num by 7
<code>\$actors -= 2</code>	<code>\$actors = \$actors - 2</code>	Decrements \$actors by 2
<code>\$bam *= 3.14</code>	<code>\$bam = \$bam * 3.14</code>	Multiplies \$bam by 3.14
<code>\$score /= 0</code>	<code>\$score = \$score / 0</code>	Divides \$score by 0 (don't!)

Exponentiation

Perl, unlike C, has an exponentiation operator: `**`. The following one-line program prints *512* because 8^3 is 512.

```
% perl -e 'print 8 ** 3;'
```

RESULT 512

Sure enough, `**=` is a perfectly valid Perl operator. This one-liner prints *16* because that's what 2^4 equals.

```
% perl -e '$num = 2; $num **= 4; print $num;'
```

RESULT 16

Lesson 2 remarked that `-7` isn't really a simple number, but an operation (unary minus) applied to the number 7. Here's proof:

```
% perl -e 'print -7 ** 2'
```

RESULT -49

Huh? Squares should always be positive. What's going on?

Precedence

The `-` operator has lower precedence than `**`, so `-7 ** 2` is interpreted as `-(7 ** 2)`, not as `(-7) ** 2`. To square `-7`, you need parentheses.

That's why `2 + 3 * 5` yields 17 and not 25. `*` has a higher precedence than `+`, so that's `2 + (3 * 5)`, not `(2 + 3) * 5`.

Appendix F contains a list of operator precedences.

Autoincrement and Autodecrement

There's an even shorter shorthand for the common tasks of adding or subtracting 1 from a number. Instead of `$var += 1`, you can use `$var++`, which increments `$var` by 1, returning the value before the increment. `++$var` also increments `$var`, but returns the value after the increment. The `--` operator works in a similar fashion, decrementing its operand.

```
% perl -e ' $a = 7; $b = $a++; print "$a, $b"; '
```

RESULT 8, 7

```
% perl -e ' $a = 7; $b = ++$a; print "$a, $b"; '
```

RESULT 8, 8

Curiously, the ++ operator can also be applied to strings.

```
$grade = 'A';
```

```
$grade++;
```

(\$grade is now *B*.)

```
$initials = 'jaa';
```

```
$initials++;
```

(\$initials is now *jab*.)

What happens when you try to increment past z? If there's a letter preceding the z, the increment "carries over."

```
$monogram = 'ez';
```

```
$monogram++;
```

(\$monogram is now *fa*.)

And if there's no preceding letter, the increment "wraps around" to make one.

```
$name = 'z';
```

```
$name++;
```

(\$name is now *aa*!)

```
$acronym = 'NBZZ';
```

```
$acronym++;
```

(\$acronym is now *NCAA*.)

When Is a String Not a String?

The behavior of ++ might suggest using other arithmetic operators with strings. DON'T.

Arithmetic operators can't really perceive strings. They see 0 instead. So,

```
"hambone" == "tbone"
```

is TRUE because 0 is equal to 0. (Thus, every string is >= every other string, and no string is < any other string.) Always use the string comparators *eq*, *ne*, *lt*, *le*, *gt*, *ge*, and *cmp* (which are introduced in Chapter 4, Lesson 6) when appropriate.

The ?: Operator

All the operators you've seen use either one or two arguments. **?:** takes three. It's similar in spirit to an *if* statement.

```
expression1 ? expression2 : expression3
```

evaluates to expression2 if expression1 is TRUE, and expression3 otherwise. So,

```
($num == 5) ? 6 : 7
```

evaluates to 6 if \$num is 5, and 7 if \$num is anything else.

```
$a = ($string ne `dodo`) ? $string : $string . `s`;
```

sets \$a to \$string if \$string isn't equal to *dodo*, and *dodos* otherwise.

```
1 ? 2 ? 3 : 4 : 5
```

evaluates to 3. (Because it's really the same as `1 ? (2 ? 3 : 4) : 5`.)

The ! Operator

Here's an operator that takes only one argument: `!`, which means *not*. It can be applied to either numbers or strings.

```
! ($answer eq "hello")
```

is the same as `$answer ne "hello"`,

```
!1
```

is 0, and

```
!0
```

is 1.

You could test whether \$num is FALSE with

```
if (!$num) {  
  print "num is zero (or an empty string, or undefined). \n";  
}
```

Chapter 1, Perl 5 Interactive Course

LESSON 4

INTRODUCTION TO LOOPS

Loops execute the same statements over and over until some stop condition is satisfied. There are three commonly used looping constructs in Perl: *while*, *foreach*, and *for*. We'll explore them (in that order) in Lessons 4, 5, and 6.

The while Loop

while (*CONDITION*) { *BODY* } repeatedly executes the statements in *BODY* as long as *CONDITION* is TRUE.

To understand loops better, let's play with a little algorithm that demonstrates loopish behavior:

1. Start with some integer *n*.
2. If *n* is even, divide it by 2. Otherwise, multiply it by 3 and add 1.
3. If *n* is 1, stop; otherwise, go to Step 2.

Eventually, *n* will become 1 (we think--no one really knows for sure), but the number of steps will vary depending on the choice of *n*. 16 plummets to 1 quickly. 27 takes much longer. Experiment!

Let's rewrite these steps in a slightly different way:

1. While (*n* isn't 1), do the following:
2. If (*n* is even) {divide it by 2}, else {multiply it by 3 and add 1}.

The *meander* program (Listing 1-11) implements this algorithm using a *while* loop and an *if...else* statement.

Listing 1-11 *meander*: Using a *while* loop

```
#!/usr/bin/perl -w
print "Please type an integer: ";
$num = <>;
chomp $num;
while ($num != 1) { # As long as $num isn't 1, do:
    print "$num ";
    # If $num is even, halve it.
    # Otherwise, multiply by 3 and add 1.
    if (($num % 2) == 0) { $num /= 2; }
```

```

else { $num *= 3; $num++; }
}
print $num, "\n"; # Print the last number (which must
# be 1, because the loop finished).

```

The *while* loop first tests to see whether \$num is equal to 1. If it isn't, the statements inside the outer curly braces are executed. Then Perl returns to the top of the loop and checks \$num again. This continues until \$num finally does become 1, at which time the loop exits and Perl moves to the next statement in the program.

Here's *meander* in action:

```
% meander
```

```
RESULT Please type an integer: 15
```

```
15 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1
```

```
% meander
```

```
RESULT Please type an integer: 20
```

```
20 10 5 16 8 4 2 1
```

```
% meander
```

```
RESULT Please type an integer: 256
```

```
256 128 64 32 16 8 4 2 1
```

Beware Infinite Loops!

Suppose a loop has the condition ($\$num \neq 1$). Then there should be some statement inside the loop that modifies \$num. In fact, it's downright critical! Otherwise, \$num stays at whatever it was before entering the loop, the condition never becomes FALSE, and the loop never exits. See whether you can determine why these loops never end:



```
#!/usr/bin/perl
$num = 0;
while ($num == 0) {
print "num is $num.\n";
}

```

Bug! The loop variable \$num never changes.



```
#!/usr/bin/perl
$num = 1;
while ($num != 10) {
print "$num is an odd number.\n";
$num += 2;
}

```

Bug! The stop condition is stepped over.

The first loop never modifies \$num at all. The second loop modifies \$num, but not in a way that ever satisfies the stop condition because \$num skips from 9 to 11.

The % Operator

The % operator (pronounced "mod," short for modulus) yields the remainder when the left side is divided by the right. Some examples:

```
% perl -e 'print 17 % 5'
RESULT 2
% perl -e 'print 6 % 3'
RESULT 0
% perl -e 'print 25 % 20'
RESULT 5
```

Because all even numbers are evenly divisible by 2, the remainder will be 0, so \$num % 2 is 0 only if \$num is even. (And should you ever find yourself needing a statement like \$big = \$big % 2, you can abbreviate it just like the assignment operators you saw earlier: \$big %= 2.)

The int() Function

You could also have tested for evenness with the int() function. int() lops off everything after the decimal point and returns the whole (integral) part.

```
int(3.14159)
is 3.
int(5.00000)
is 5.
int(-2.71828)
is -2.
```

Because odd numbers yield a fractional part when divided by 2 (e.g., 7 / 2 is 3.5) and even numbers don't, the int() function gives us another way to determine evenness. If int(\$num / 2) is equal to \$num / 2, you know that \$num must be even. Listing 1-12 demonstrates this property.

Listing 1-12 *oddeven*: Using the int() function

```
#!/usr/bin/perl -w
print "Enter a number: ";
chomp($num = <>); # equivalent to $num = <>; chop $num;
if (int($num/2) != ($num/2)) { # If $num is odd,
print "$num is odd.\n"; # say so!
}else { print "$num is even.\n"; }
```

Let's try *oddeven* with a few different inputs.

```
% oddeven
```

RESULT Enter a number: 5

5 is odd.

% oddeven

RESULT Enter a number: 248

248 is even.

% oddeven

RESULT Enter a number: 4.8

4.8 is odd.

because $2 \neq 2.4$. (*oddeven* classifies all fractional numbers as odd.)

Rounding Numbers

You can round a number to the nearest integer using `int()`.

```
$nearest_integer = int($number + 0.5)
```

works for positive numbers, and

```
$nearest_integer = ($number > 0) ? int($number + 0.5) :  
int($number - 0.5)
```

works for both positive and negative numbers.

```
$money = int(100 * $money) / 100
```

truncates `$money` after the hundredths place, so 16.6666 becomes 16.66. The index entry here should be `money`, not `$money`.

Check the `printf()` and `sprintf()` functions (Chapter 6, Lesson 4) for another way to round numbers to a particular number of digits. The POSIX module (Chapter 8, Lesson 8) includes `floor()` and `ceil()` functions, different ways to round numbers to the nearest integer.

Integer Arithmetic

Perl stores all numbers, integers and real numbers alike, in a floating point representation. If you're looking to speed up numeric processing, read about the *integer* pragma in Chapter 8, Lesson 1.

Chapter 1, Perl 5 Interactive Course

LESSON 5

ARRAYS

Scalars contain just one value. *Arrays* (sometimes called *lists*) contain multiple scalars glued together--they can hold many values or none at all (Figure 1-3). Array variables begin with an @ sign (e.g., @movies or @numbers), and their values can be represented as a comma-separated list of scalars enclosed in parentheses.

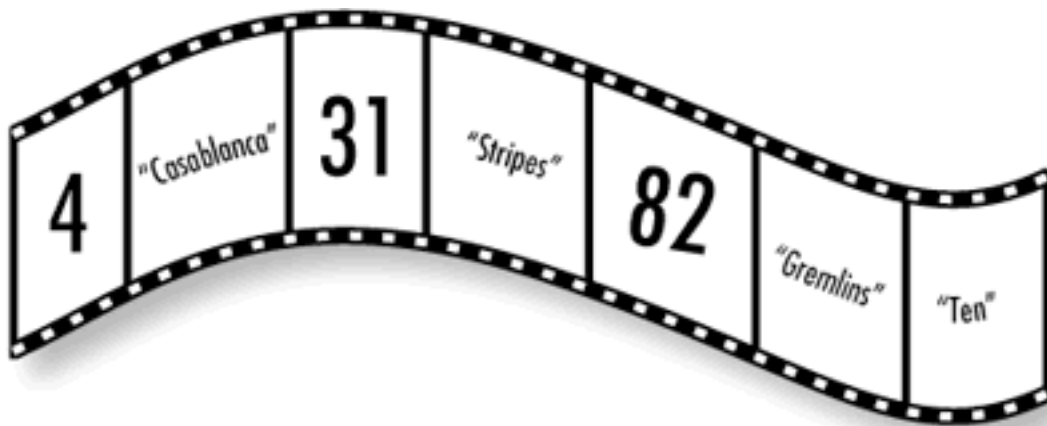


Figure 1-3

An Array

```
@movies = ('Casablanca', 'Star Wars', 'E.T.');
```

is an array of three strings.

```
@numbers = (4, 6, 6, 9);
```

is an array of four numbers.

```
$var = 7;  
@values = ($var, $var, 2, $var + 1)
```

sets @values to (7, 7, 2, 8).

Functions such as print() expect an array of arguments. That's why you separate them with commas, as in

```
print $greeting, ' world!', "\n";
```

or

```
print(5, 6, 7);
```

or even

```
@array = (5, 6, 7);
```

```
print @array, "\n";
```

You can mix data types in an array:

```
@numbers = (6, 7, "nine");  
@years = (1996.25, `two thousand`, 1066);
```

and combine multiple arrays into one:

```
@things = (@numbers, "hello", @years); # @things now has  
seven elements.
```

Here's an array with no elements at all:

```
@empty_array = ();
```

The *foreach* Loop

Let's switch gears a little bit and explore our second loop construct: *foreach*, which iterates through the elements of an array. The *marquee* program, shown in Listing 1-13, uses an array to store a list of movies and a *foreach* loop to print all of them.

Listing 1-13 *marquee*: Initializing and looping through an array

```
#!/usr/bin/perl -w  
  
@movies = ('Casablanca', 'Stripes', 'Gremlins', 'Sleeper');  
  
print "Now showing at the Perlplex:\n";  
foreach $movie (@movies) {  
    print $movie, "\n";  
}
```

The first line of *marquee* initializes the array @movies so that it contains four strings. Then the *foreach* statement steps through @movies, changing \$movie to the "current" array element and printing it. The result:

```
% marquee
```

```
RESULT Now showing at the Perlplex:
```

```
Casablanca  
Stripes  
Gremlins  
Sleeper
```

The *foreach* Loop

foreach *SCALAR* (*ARRAY*) { *BODY* } executes the statements in *BODY* once for every *ARRAY* element. The "current" array element is stored in *SCALAR*.

When the *foreach* loop begins, \$movie becomes the first array element: *Casablanca*. The *BODY* prints it, and when the loop begins again, \$movie is set to the next array element: *Stripes*. This continues until @movies runs out of elements.

If `@movies` had been undefined or had been set to an empty array `()`, the *BODY* would have been skipped over entirely.

Array Indexing

All arrays are ordered: You can refer to the first element, or the third, or the last. Like C, Perl uses *zero-based indexing*, which means that the elements are numbered starting at zero. So if the array `@movies` has four elements, the first element is number 0, and the last element is number 3. It's important to remember this when accessing an array's elements.

There are a few different ways to access the elements of an array. This can get a little confusing; the important thing to remember is that you can always tell what something is by its identifier. Anything beginning with a `$` is a scalar and anything beginning with an `@` is an array.

- You can access entire arrays with `@`.

```
@movies = ( 'Casablanca', 'Star Wars', 'E.T.', 'Home Alone' );
```

sets `@movies` to an array of four strings.

```
@movies2 = @movies;
```

copies the contents of `@movies` to `@movies2`.

- You can access individual scalar elements with `$` and `[]`.

```
$movies[1]
```

is the second element of `@movies`. (Remember, `$movies[0]` is the first element.)

```
$movies[3] = 'Home Alone 2';
```

sets the fourth element of `@movies` to `'Home Alone 2'`.

- You can access a subarray (called a slice) with `@` and `[]`.

```
@movies[1..3]
```

is `('Star Wars', 'E.T.', 'Home Alone')`.

```
@movies[0..2] = ( 'Big', 'Jaws', 'Batman' );
```

sets the first three elements of `@movies`, yielding

```
( 'Big', 'Jaws', 'Batman', 'Home Alone 2' )
```

The `..` operator creates an implicit array: `1..4` is (in most ways) equivalent to `(1, 2, 3, 4)`. You can use this operator to loop through a range of numbers.

```
% perl -e 'foreach $num (2..9) { print $num; }'
```


However, you shouldn't do this for very large ranges. Use a *for* loop instead. See the next lesson.

A Few Words About Initial Characters

The syntax for accessing scalars and arrays isn't arbitrary. Remember the cardinal rule: You can identify what's being accessed by the initial character. Single pieces of data (scalars) always begin with a \$ (such as \$var, \$array[2]), and multiple values (arrays) always begin with an @, such as @array, or @array[0..2], or even @array[7], which is an array that happens to contain just one element. Don't be misled by the single element inside the brackets: **@array[7] is NOT a scalar!**

Other data types, introduced in the next four chapters, use different symbols: & (subroutines), % (hashes), and * (symbol table entries).

Out of Bounds!

What happens if you take the four-element @movies and assign elements beyond the end of the array? You know that the last element of @movies is \$movies[3]. If you say

```
@movies[3..5] = ( `Doctor Detroit`, `Grease`, `Saturday Night  
Fever` );
```

then Perl automatically extends @movies for you.

The Last Index of an Array

The scalar \$#array contains the last index of @array. If @array has four elements, then \$#array is 3. You can always access (or modify) the last element of an array with this notation:

```
$movies[$#movies] = `The Last Seduction`
```

sets the last element of @movies to *The Last Seduction*, regardless of how many elements @movies contains.

Furthermore, you can extend an array by setting this value. After saying

```
$#movies = 10
```

@movies will have 10 elements, no matter how many it had before. Any new elements created in this way will be undefined but will exist anyway. (There's a subtle difference between existing and being defined, explained further in Chapter 5, Lesson 5.)

The \$[Special Variable

\$[is a special variable. You'll see a slew of special variables throughout the book. This particular variable is a scalar (but of course, you knew that from the \$) containing the first index of all arrays. Because Perl arrays have zero-based indexing, \$[will--almost always--be 0. But if you set \$[to 1, all your arrays will use one-based indexing. Don't do it--\$[may disappear in future versions of Perl. It won't be used in any of the programs in this book.

Negative Array Indexes

You can access the n th-to-last element of an array with negative array indexes.

```
$movies[-1]
```

is the last element of @movies. And

```
$movies[-3]
```

is the third-to-last element of @movies.

```
$movies[-10] = 'The Adventures of Buckaroo Banzai';
```

sets the tenth-to-last element of @movies to *The Adventures of Buckaroo Banzai*.

Adding and Removing Array Elements

Perl supplies several functions for moving array elements around. We'll discuss five now: `push()`, `pop()`, `shift()`, `unshift()`, and `splice()`.

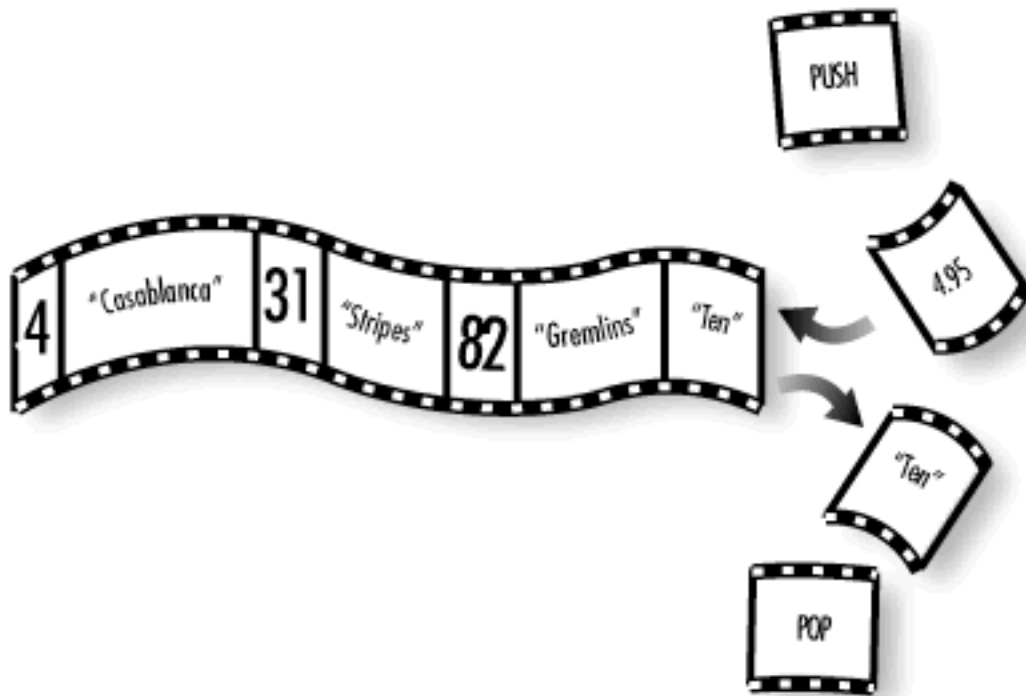


Figure 1-4
push() in action

push() and pop()

`push()` and `pop()` manipulate elements on the end (the right side) of an array.

The push() and pop() Functions

push (*ARRAY*, *LIST*) appends the *LIST* of elements to the end of *ARRAY* (Figure 1-4).

pop (*ARRAY*) removes and returns the last *ELEMENT* of *ARRAY*.

pop() is to arrays as chop() is to strings--they both lop off and return the last item.

unshift()* and *shift()

unshift() and shift() are similar to push() and pop() but operate on the beginning of the array instead of the end.

The unshift() and shift() Functions

unshift (*ARRAY*, *LIST*) prepends the *LIST* of elements to the beginning of *ARRAY*.

shift (*ARRAY*) removes and returns the first element of *ARRAY*.

Let's say you want to remove a movie from @movies. Here are a few ways to do it:

```
pop@movies; # Removes the last element of @movies
shift@movies; # Removes the first element of @movies
```

pop() and shift() return the element that was removed from the array.

```
$last = pop@movies; # Removes the last element of @movies,
# placing it in $last
```

```
$first = shift@movies; # Removes the first element of
@movies,
# placing it in $first
```

To add an element to @movies, you can use either push() or unshift().

```
push @movies, 'Ben Hur'; # Adds 'Ben Hur' to the end of
@movies
```

```
unshift@movies, 'Ben Hur'; # Adds 'Ben Hur' to the beginning
of @movies
```

splice()

To access (or modify) a chunk of an array, you can use splice().

The splice() Function

splice(*ARRAY*, *OFFSET*, *LENGTH*, *LIST*) removes and returns *LENGTH* elements from *ARRAY* (starting at index *OFFSET*), replacing them with *LIST*.

You can omit the *LENGTH* or *LIST* arguments, as shown.

```
splice @movies, 4
```

removes and returns all elements after (and including) \$movies[4].

```
splice @movies, 4, 2
```

removes and returns \$movies[4] and \$movies[5].

```
splice @movies, 4, 2, `UHF`, `Under Siege`
```

removes and returns \$movies[4] and \$movies[5], replacing them with *UHF* and *Under Siege*.

But you don't need splice() all that often because you can access array values without any functions at all.

```
@array[3, 5, 7] = ("hi", 7.5, "there")
```

sets \$array[3] to hi, \$array[5] to 7.5, and \$array[7] to *there*, as does

```
($array[3], $array[5], $array[7]) = ("hi", 7.5, "there")
```

whereas

```
@array[3..5] = ("hi", 7.5, "there")
```

sets \$array[3] to hi, \$array[4] to 7.5, and \$array[5] to *there*, which is equivalent to

```
splice(@array, 3, 3, "hi", 7.5, "there")
```

Chapter 1, Perl 5 Interactive Course

LESSON 6

ADVANCED LOOPS

We've covered two types of loops so far: *foreach* and *while*. In this lesson, you'll learn about the other commonly used loop statement in Perl--*for*--as well as three loop operators (*next*, *last*, and *redo*) that allow you to exert finer control over the sequence of execution.

The *for* Loop

Perl's *for* loop is just like C's *for* loop--it lets you specify an event that occurs after every loop iteration and what happens before your loop begins. *for* doesn't do anything that a *while* loop can't do--it just does it more concisely.

The *for* Loop

for(*START*; *STOP*; *ACTION*) { *BODY* } initially executes the *START* statement and then repeatedly executes *BODY*, as long as the *STOP* condition remains true. The *ACTION* statement is executed after each iteration.

Unlike in C, the curly braces are mandatory.

Listing 1-14 shows an example that uses a *for* loop to print the first 10 letters of the alphabet.

Listing 1-14 *alphabet*: Using a *for* loop

```
#!/usr/bin/perl -w
for ($letter = 'a'; $letter lt 'k'; $letter++) { # Start $letter at 'a',
print $letter; # and increment until 'k' is reached.
}
```

The *for* loop first sets *\$letter* to *a* and then checks to make sure that the stop condition is satisfied. It is (*a* is less than *k*), so it prints *\$letter*. Then it performs the increment statement: *\$letter++*. Now *\$letter* is *b* and the loop begins again. This continues until *\$letter* becomes *k*, at which time the loop exits.

```
% alphabet
```

```
RESULT abcdefghij
```

Listing 1-15 shows yet another *tickets* program. *tickets4* prints a list of movies, asks users to choose movies from the list, and then repeats those choices. It uses one *for* loop for each of the three tasks.

Listing 1-15 *tickets4*: Some *for* loops

```
#!/usr/bin/perl
@movies = ('Casablanca', 'E.T.', 'Star Wars', 'Home Alone');
@choices = (); # An empty array for storing user choices.

# Start $i at 0, and loop through 3, incrementing by 1 each time,
# and printing the $ith element of @movies.
for ($i = 0; $i <= 3; $i++) {
print "Movie number $i is $movies[$i] \n";
}
```

```

print "\nHow many people are in your party? ";
chomp($num = <>);

# Start $i at 1, and loop through $num, incrementing by 1 each time,
# and pushing the user's choice onto the end of @movies.
for ($i = 1; $i <= $num; $i++) {
    print "What movie would you like to see, person #$i? ";
    chomp($choice = <>);
    push(@choices, $choice);
}

print "\nFor your party of $num:\n";

# Start $i at 0, and loop through $num - 1, incrementing by 1 each time,
# and printing which movies people want to see.
for($i = 0; $i < $num; $i++) { # Count from 0 to $num-1
    print "Person ", $i+1, " would like to see $movies[$choices[$i]]. \n";
}

```

A sample session:

```
% tickets4
```

```
RESULT Movie number 0 is Casablanca
```

```

Movie number 1 is E.T.
Movie number 2 is Star Wars
Movie number 3 is Home Alone

```

```

How many people are in your party? 3
What movie would you like to see, person #1? 2
What movie would you like to see, person #2? 0
What movie would you like to see, person #3? 3

```

```

For your party of 3:
Person 1 would like to see Star Wars.
Person 2 would like to see Casablanca.
Person 3 would like to see Home Alone.

```

Let's examine each *for* loop in turn. The first loop is the simplest.

```
for ($i = 0; $i <= 3; $i++) { print "Movie number $i is $movies[$i] \n"; }
```

First, the start statement is executed: $i = 0$.

Then i is compared to 3. It's less than or equal to 3, so the block is executed.

Then i is incremented and the loop executes again.

The second loop is similar to the first, except that i starts at 1 instead of 0 and the stop condition depends on the value of num instead of being fixed at 3. If there are four people in the party, the loop is executed four times.

The third *for* loop starts i at 0 and continues until i is no longer less than num . Both the second and the third loops execute num times--but there's a difference:

The second loop counts from 1 to num , whereas the third loop counts from 0 to $num-1$.

Loop Control: *next*, *last*, and *redo*

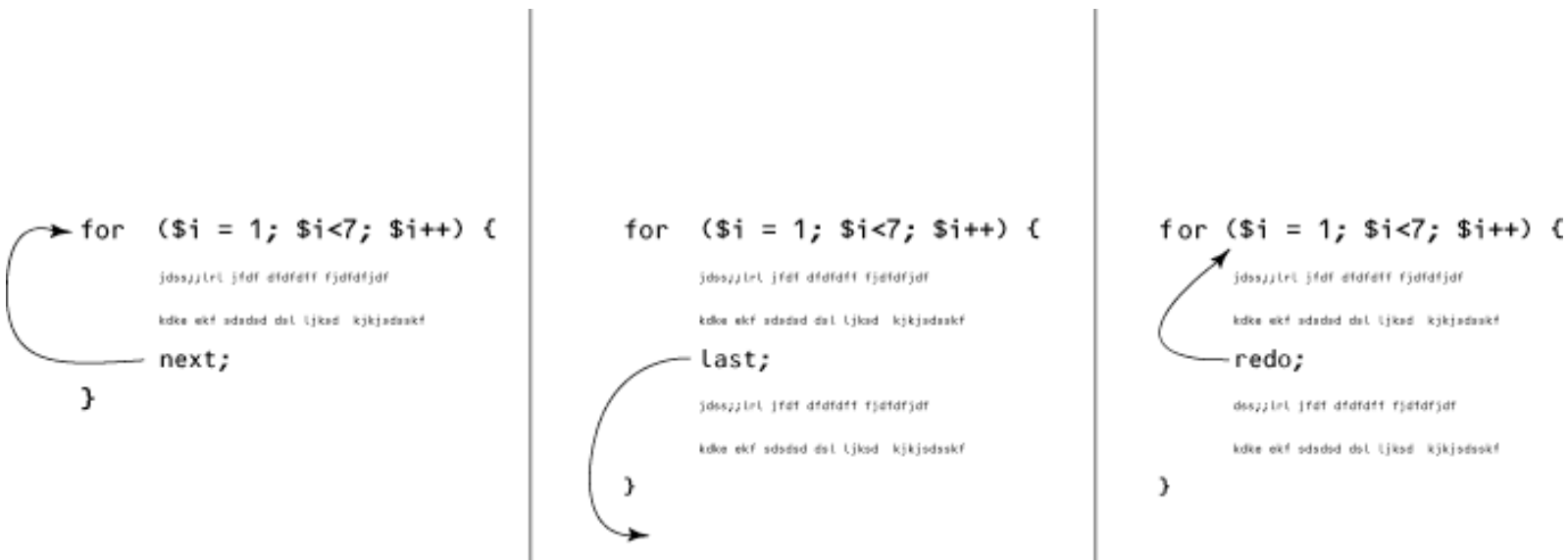


Figure 1-5
next, last, and redo

Instead of always beginning each loop at the top and mindlessly proceeding to the bottom, you can jump around a bit with *next*, *last*, and *redo* (Figure 1-5).

- *next* begins another loop iteration, executing the *ACTION* beforehand.
- *last* exits the loop.
- *redo* jumps to the top of the loop without executing the *ACTION*.

We'll demonstrate *next*, *last*, and *redo* with two programs. *tickets5* uses *next* and *last*, and *lotto* uses *redo*.

Suppose a 12-year-old runs one of the *tickets* programs. Children under 17 aren't allowed to see R-rated movies without parental permission, and this little tyke doesn't have a parent handy. Of course, you want your computer program to enforce the law dutifully, so it should skip over the R-rated movies.

tickets5, shown in Listing 1-16, immediately moves to the next movie if the current movie is R-rated, and declares the current iteration to be the last if the user says *yes* to a movie.

Listing 1-16 *tickets5*: Using *next* and *last* to control loop flow

```
#!/usr/bin/perl -w
@movies = ('Star Wars', 'Porky's', 'Rocky 5', 'Terminator 2',
           'The Muppet Movie');

@ratings = ('PG', 'R', 'PG-13', 'R', 'G'); # The ratings for @movies.
print "Welcome to Perlplex Cinema, little tyke! \n";
for ($i = 0; $i <= 4; $i++) { # Count from 0 to 4
    next if $ratings[$i] eq 'R'; # Skip over R-rated movies
    print "Would you like to see $movies[$i]? ";
    chomp($answer = <>);
```



```

last if $answer eq `yes`; # Exit the loop, if the
# child agrees to a movie
}
if ($answer eq `yes`) { print "That'll be \$4.25\n" }
else { print "I'm sorry you don't see anything you like.\n" }

```

The *next* statement checks the rating of the *\$movie* title in *@movies*. The condition (*\$ratings[\$i] eq 'R'*) triggers on *Porky's* and *Terminator 2*, so those movies won't be mentioned to the child. If the child says *yes* to any of the movies, the *last* statement triggers and the loop immediately exits, at which time the user is asked for money.

```
% tickets5
```

```
RESULT Welcome to Perlplex Cinema, little tyke!
```

```
Would you like to see Star Wars? no
```

(The *next* statement then skips over *Porky's*.)

```
Would you like to see Rocky 5? yes
```

(The *last* statement then exits the loop.)

```
That'll be $4.25.
```

(Youth discount.)

Suppose you want to begin another loop iteration, but without evaluating the stop condition or the increment step. The *redo* statement does just that, demonstrated in *lotto* (Listing 1-17).

Listing 1-17 *lotto*: Using *redo* to begin another iteration

```

#!/usr/bin/perl -w
print "** Pick 4 LottoMatic ** \n";
print "** Type four digits (0 through 9) ** \n\n";

for ($i = 1; $i <= 4; $i++) {
print "Choice $i: ";
chomp ($digit[$i] = <>);
redo if $digit[$i] > 9;
redo if $digit[$i] < 0;
}

print "Your choices: @digit \n";

```

lotto wants users to enter four digits, one at a time. If the user enters a number that isn't between 0 and 9, one of the *redo* statements triggers, sending control back to just past the top of the loop so that *\$i* is neither incremented nor compared to 4. It's a way to say, "Let's try that again."

```
% lotto
```

```
RESULT ** Pick 4 LottoMatic **
```

```
** Type four digits (0 through 9) **
```

```
Choice 1: 7
```

```
Choice 2: 12
```

(The first *redo* triggers.)

```
Choice 2: 1
```

```
Choice 3: 8
```

```
Choice 4: -3
```

(The second *redo* triggers.)

```
Choice 4: 6
```

```
Your choices: 7 1 8 6
```

Ssssshhhh!

A confession: *for* and *foreach* are actually just different names for the same thing. You can loop through arrays with *for*, and you can specify a start, stop, and increment with *foreach*. So why are there two names for the same statement? Because C programmers like their *for* loops and shell programmers like their *foreach* loops. Even though you've been let in on this little secret (few Perl programmers know this!), you shouldn't take advantage of it because other programmers will think of C when they see *for* and think of shells when they see *foreach*. This book pretends they're distinct, but now you know better.

Chapter 1, Perl 5 Interactive Course

LESSON 7

THE DEFAULT VARIABLE \$_

The President said that the President will sign any bill that arrives on the President's desk.

The President said that he'll sign any bill that arrives on his desk.

Pronouns such as "his" shorten and simplify our sentences, relying upon listeners to interpret the right value in its place. Wouldn't it be nice if programming languages could do that? Unfortunately, most programming languages can't: If a function expects two arguments, you can't just say, "Here's the first argument; use the obvious choice for the second."

Perl has something not found in many other programming languages: the default variable `$_`.

Some functions and operators can use `$_` in place of an argument. If so, you can omit that argument and Perl will use the contents of `$_` instead. That might sound horrible; after all, won't that sometimes lead to surprising behavior? Not often, and statements with fewer arguments are easier to read. You'll find `$_` quite handy.

Anyway, let's see what `$_` can do. Consider the following *foreach* loop:

```
foreach $movie (@movies) { print $movie }
```

Thanks to `$_`, you can eliminate `$movie`.

```
foreach (@movies) { print }
```

"Huh?" you ask? "Where'd `$movie` go? And how does `print()` know what to print?"

Here's how it works. When a *foreach* loop has no iteration variable (the `$movie` in **foreach** `$movie` (`@movies`)), it uses `$_` instead. And if `print()` is given nothing to print, it uses the contents of `$_`. So you don't need `$movie` at all.

Several of the functions you've seen use `$_`:

```
chomp;
```

is the same as

```
chomp$_;
```

and

```
print;
```

is the same as

```
print$_;
```

and

```
int;
```

is the same as

```
int$_;
```

Whenever a *while* loop is given `<>` as its condition, it stores each line of user input in `$_`. That's demonstrated in *parrot* (Listing 1-18), which echoes everything the user types.

Listing 1-18 *parrot*: Evaluating `<>` inside a *while* loop

```
#!/usr/bin/perl -w
while (<>) { # Stuff whatever the user types into $_
    print; # ...and print it!
}
```

Look ma, no variables!

```
% parrot
```

```
RESULT This is line one.
```

```
This is line one.
```

```
Hello?
```

```
Hello?
```

```
Polly wanna cracker?
```

```
Polly wanna cracker?
```

```
[CTRL]-[D]
```

[CTRL]-[D] terminates standard input, so the *while* loop exits. (Use [CTRL]-[Z] followed by a [RETURN] on MS-DOS computers.)

You can also assign values to `$_`, just like any other variable.

```
$_ = "Poughkeepsie";
print;
```

prints *Poughkeepsie*.

@ARGV

`$_` isn't the only default variable. (`@_` is another one--an array used to pass arguments to subroutines--but that won't be discussed until Chapter 4.) Another useful variable, `@ARGV`, contains all the command-line arguments provided to your program.

The first command-line argument can be found in `$ARGV[0]`, the second command-line argument can be found in `$ARGV[1]`, and so on. (That's not quite the same as C's `argv`, which stores the program name in `argv[0]`. Perl stores the filename in `__FILE__`, the process name in `$0`, and the name of the Perl binary in `$^X`. More about those in later chapters.)

Suppose *program* is a Perl program called with three command-line arguments:

```
% program 5.6 index 0
```

Then `@ARGV` will be equal to `(5.6, 'index', 0)`.

None of the programs you've seen so far has used any arguments, so here's *cube* (Listing 1-19), which raises its command-line argument to the third power.

Listing 1-19 *cube*: Using @ARGV

```
#!/usr/bin/perl -w
$num = $ARGV[0]; # Set $num equal to the first command-line
argument
print $num ** 3; # Print its cube.
% cube 5
```

(\$ARGV[0], first element of @ARGV, is 5.)

```
125
```

```
% cube -7
```

```
RESULT -343
```

```
% cube 28.98
```

```
RESULT 24338.574792
```

```
% cube
```

(Whoops! Forgot the command-line argument!)

```
0
```

(That's okay--Perl assumes a 0.)

shift() uses @ARGV as a default variable:

```
shift;
```

removes and returns the leftmost argument of @ARGV. pop() returns the rightmost argument of @ARGV.

Listing 1-20 shows another way to write *cube*.

Listing 1-20 *cube2*: Using shift() to extract program arguments

```
#!/usr/bin/perl -w
$num = shift;
print $num ** 3;
```

An even shorter way is demonstrated in *cube3*, shown in Listing 1-21.

Listing 1-21 *cube3*: Using shift() to extract program arguments

```
#!/usr/bin/perl -w
print shift() ** 3;
```

Of course, @ARGV can hold more than one command-line argument. Listing 1-22 shows a program called *expo* that exponentiates two command-line arguments. *expo* expects the base and the exponent on the command line, in that order.

Listing 1-22 expo: Accessing the first two command-line arguments

```
#!/usr/bin/perl -w
# These next two lines are equivalent to ($base, $exponent) =
@ARGV
$base = shift; # Remove the leftmost argument, shortening
@ARGV.
$exponent = shift; # Remove the (now-)leftmost argument
# Now @ARGV should be empty.

print $base ** $exponent;
```

```
% expo 5 3
```

(Raises 5 to the 3rd power)

```
RESULT 125
```

```
% expo 2 0.5
```

(Computes the square root of 2)

```
RESULT 1.414213562373095
```

```
% expo 1.00001 100000
```

(Computes the 100000th power of 1.00001)

```
RESULT 2.718268237192297
```

```
% expo -1 .5
```

(Attempts to compute the square root of -1)

```
RESULT NaN
```

(Meaning "not a number." Your computer may print something slightly different.)

Chapter 1, Perl 5 Interactive Course

LESSON 8

CONTEXTS

"Which **movie** do you want to see?"

"Top Gun"

"Which **movies** do you want to see?"

"Top Gun, Cocktail, Risky Business"

Each of these questions expects a different answer. The first expects a single movie--a scalar. The second question expects a list of movies--an array. The questions are the same; the *contexts* are different.

Every Perl operation is evaluated in a particular context. There are two major contexts: scalar and list. Some Perl operations act differently depending on the context, yielding an array in a list context and a scalar in a scalar context. This sounds awfully abstract, so let's look at some examples.

```
@array = (1, 3, 6, 10);  
@new_array = @array;
```

That's a list context: The = operator sees the array on its left and so copies all of @array.

```
$length_of_array = @array;
```

That's a scalar context: The = operator sees the scalar on its left and so copies *the number of elements* into \$length_of_array.

(That last statement should set off a warning bell: The left side and right side are different types. That's not necessarily an error, but it does mean that something strange is going to happen.)

When an array is evaluated in a scalar context by certain operations (e.g., =, or <, or ==), it "becomes" the number of elements in the array. So this iterates through @array:

```
for ($i=0; $i < @array; $i++)
```

When you see an expression like `$i < @array`, you should ask yourself, "Is that comparison occurring in a scalar context or a list context?" Because `<` wants to compare two numbers, @array is evaluated in a scalar context, returning the length of the array. So if there are four elements in @array, \$i starts at 0 and stops before 4. (That's what you want because `$array[3]` is the last element of a four-element array. Remember: Perl arrays have zero-based indexing.)

Some operations don't merely act differently depending on the context; they *create* the context for you. That's why you often have to look at both sides of the operator as well as the operator itself to determine what happens.

```
$b = (4, 5, 6)
```

performs the assignment in a scalar context, setting \$b to the last element: 6.

```
@array = (4, 5, 6);  
$b = @array;
```

also performs the assignment in a scalar context, but sets \$b to 3, the number of elements.

```
@b = (4, 5, 6)
```

performs the assignment in a list context, setting @b to (4, 5, 6).

```
($a, $b, $c) = (4, 5, 6)
```

also uses a list context, setting \$a to 4, \$b to 5 and \$c to 6.

The + operator creates a scalar context, but that doesn't tell you everything you need to know: Does + convert arrays to scalars by using the length? Or does it use the last element? See for yourself.

```
4 + (100)
```

is 104.

```
4 + (25, 200);
```

is 204.

print() creates a list context. That's why

```
print "abc", "def", ("ghi", "jkl")
```

prints *abcdefghijkl*.

Consider

```
print 4 + (25, 200);
```

which prints *204*, because the + sees only the last element of (25, 200). Compare that to these statements, which might look like they should behave similarly:

```
@x = (25, 200);  
print 4 + @x;
```

This prints 6, because the + operator uses a scalar context in which @x evaluates to the number of elements. The difference is very subtle, but worth remembering: Some operations distinguish between an array (such as @x) and a mere list of elements (such as (25,200)). And all of this happens before print() gets a chance to interpret @x with a list context.

The scalar() Function

Sometimes you want an expression to be evaluated as a scalar, even though Perl might want to evaluate it as a list. You can force a scalar context with the scalar() function.

```
@list = (1, 4, 6, 10, 15);  
print "@list";
```

prints *1 4 6 10 15*, as you'd expect.

But look:

```
@list = (1, 4, 6, 10, 15);
```



```
print scalar(@list);
```

prints 5.

Congratulations! Now that you've covered the basics of Perl syntax, you know enough to write useful programs in Perl. Don't worry if everything isn't crystal clear--it doesn't need to be. As long as you understand how conditional statements and loops work, and understand the differences between scalars and arrays, you're well prepared for Chapter 2.

Review

Humphrey: I understand loops a lot better now. But I'm not sure what to make of these contexts and default variables.

Ingrid: Me neither. I've never seen anything like them in any programming language.

Humphrey: They seem dangerous. Programming languages shouldn't be like real languages. They should enforce a uniform style and syntax so there's only one way to solve a problem. That way, it's easy to understand programs other people write.

Ingrid: You won't like Perl, then, because it gives you several ways to do just about everything.

Chapter 2, Perl 5 Interactive Course



PATTERN MATCHING: REGULAR EXPRESSIONS

Many tasks require recognizing patterns in text: extracting a column of text from the output of a command, replacing certain strings of text with others, selecting elements from an array that fulfill some condition. All use patterns that match some pieces of data but not others. In Perl, patterns can be represented compactly and efficiently as regular expressions.

Perl's regular expressions make it possible to write powerful but concise programs--often as short as one or two lines--that require dozens or even hundreds of lines in other programming languages. Regular expressions are the heart and soul of Perl, and the source of Perl's reputation for being useful but cryptic.

In a sense, regular expressions constitute a little programming language of their own. When you construct a regular expression, you combine symbols (each of which is a simple pattern) into a more complicated structure that defines a more sophisticated pattern. Think of these symbols (and there are a lot of them in this chapter) as elemental building blocks; you can sculpt them into architectural marvels, as long as you arrange them very precisely.

Here's a teaser, to show you the power of regular expressions. The command:

```
perl -p -e 's/\b([aeiouy]*)(\S+)\s?/$2$1ay /gi'
```

converts text into pig Latin.

```
echo "pattern match" | perl -p -e  
's/\b([aeiouy]*)(\S+)\s?/$2$1ay /gi'
```

prints

What more could you ask for?

LESSON 1

REGULAR EXPRESSIONS

Regular expressions (often called regexes) are patterns of characters. When you construct a regular expression, you're defining a pattern (or "filter") that accepts certain strings and rejects all others (see Figure 2-1).

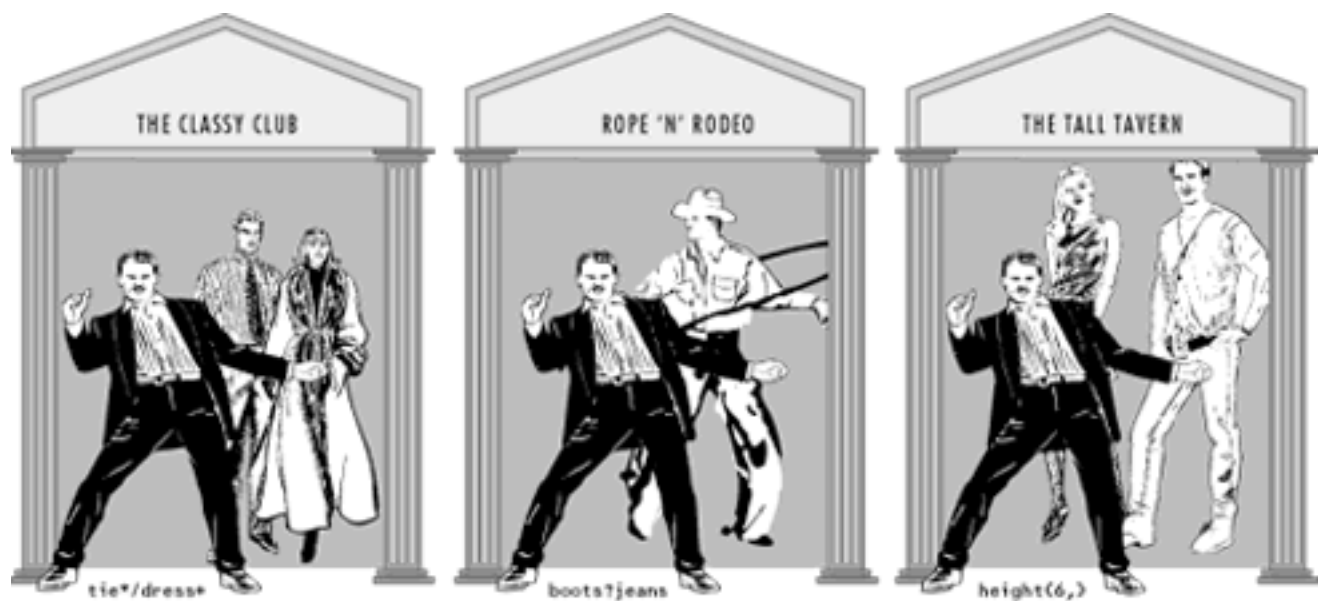


Figure 2-1

Patterns are selective, like bouncers in a nightclub: They accept some strings and reject others

Table 2-1 shows a few regular expressions. Eyeball each regex on the left and see whether you can infer the rule: Why does the pattern match the strings in the middle column, but not the strings in the right column? All the funny characters that define the pattern (*, +, \s, |, and so on) will be explained in greater detail throughout this chapter (plus there's a summary in Appendix I), so think of this as a puzzle, nothing more.

Table 2-1 Some regular expressions

Pattern	Strings That Match The Pattern	Strings That Don't

/fossil/	<i>fossil and fossiliferous</i>	<i>ossify and fossssil</i>
/bo*/	<i>b and bo and booo</i>	<i>ooooo and splat</i>
/a(ha)*/	<i>a and aha and ahaa and ahahaha...</i>	<i>ho and hhhh</i>
/hm+/	<i>hm and hmm and hmmm...</i>	<i>h and m</i>
/yooho{2,4}/	<i>yoohoo and yoohooo and yoohoooo</i>	<i>yooho</i>
/Joh?n/	<i>Jon and John</i>	<i>Jan and Johhn</i>
/black white/	<i>black and white</i>	<i>green</i>
/high\s+low/	<i>high low (with some whitespace in between)</i>	<i>highlow and highXXXlow</i>
/high\S+low/	<i>highXXXlow (with some non-whitespace in between)</i>	<i>highlow and high low</i>
^\\$\\d\\.\\d\\d/	<i>\$1.01 and \$8.95 and \$6.70...</i>	<i>23 and \$19.95</i>
/.../	<i>Any three characters</i>	<i>no and hi and we and us</i>

As you can see, regular expressions are usually enclosed in slashes. The various metacharacters (e.g. *, +, ., ?, \d, \s, \S) in Table 2-1 each change the regex behavior in their own way, affecting which strings match and regex and which strings don't.

Making Substitutions

So what can you do with regexes? Well, one common task is to make substitutions in a chunk of text using the `s///` operator.

The `s///` Operator

`s/REGEX/STRING/MODIFIERS` replaces strings matching **REGEX** with **STRING**.

`s///` returns the number of substitutions made and operates on the default variable `$_`, unless you supply another scalar with `=~` or `!~`, introduced in Lesson 3 of this chapter.

The *modifiers* `/e`, `/g`, `/i`, `/m`, `/o`, `/s`, and `/x` are characters that you can add to the end of the `s///` expression to affect its behavior. They'll be discussed throughout this chapter.

Listing 2-1 shows a simple program that uses `s///` to replace occurrences of *which* with *that*.

Listing 2-1 strunk: Using `s///` to replace one string with another

```
#!/usr/bin/perl
while (<>) { # assign each input line to $_
    s/which/that/; # in $_, replace "which" with "that"
    print; # print $_
}
```

```
}
```

strunk's pattern consists of five literal characters: **which**. That's about as simple as patterns get: Only strings containing those five characters in that order will match.

```
% strunk
```

```
RESULT which
```

```
that
```

```
the one which has the broomstick?
```

```
the one that has the broomstick?
```

```
[CTRL]-[D]
```

Wouldn't it be nice if *strunk* could read files, too? Surprise--it can! For a detailed explanation about the subtleties of `<>` that make this possible, see the beginning of Chapter 3. For now, however, we'll demonstrate this ability using the text file *grammar*, which appears in Listing 2-2.

Listing 2-2 *grammar*

```
The videotape which malfunctioned is in the corner.
```

```
That which is, is not; that which is not, is.
```

On a UNIX system, you can pipe this file to the standard input of *strunk* by typing `cat grammar | strunk`. Or, you can just supply *grammar* as a command-line argument.

```
% strunk grammar
```

```
RESULT The videotape that malfunctioned is in the corner.
```

```
That that is, is not; that which is not, is.
```

Hold on! If you look closely at *strunk*'s output, you'll notice that the substitution worked only for the first *which* in each line. The second *which* in line two remained unchanged. That's because `s///`, by default, replaces only the first matching string. To change all strings, use the `/g` modifier, which makes global replacements.

The `/g` Modifier for `s///`

The `/g` modifier performs global search and replace.

Modifiers are tacked on to the end of `s///`, so

```
s/which/that/g
```

replaces all occurrences of *which* with *that*, and

```
s/2/20/g
```

replaces all 2s with 20s.

Alternatives

The vertical bar (`|`) is used to separate alternatives in regexes. (In this context, the `|` is pronounced "or.") Listing 2-3, *abbrevi8*, replaces several similar-sounding words with the same abbreviation.

Listing 2-3 *abbrevi8*: Using `s///g` to make global substitutions

```
#!/usr/bin/perl -w
while (<>) {
    s/too|to|two/2/g; # Change all too's and to's and two's to 2s
    s/four|fore|for/4/g; # Change all four's and fore's and for's
    to 4s
    s/ought|oh|owe|nothing/0/g; # etc.
    s/eight|ate/8/g;
    print;
}
```

Each of the four lines above is evaluated once for each line of text the user types.

```
% abbrevi8
```

RESULT I ought to owe nothing, for I ate nothing.

```
I 0 2 0 0, 4 I 8 0.
```

Now isn't that much clearer?

There's nothing stopping you from using variables as replacement strings.

```
$replacement = `that`;
s/which/$replacement/;
```

replaces *which* with *that*. You can even use variables as patterns.

```
$pattern = `eight|ate`;
s/$pattern/8/;
```

replaces *eight* or *ate* with 8.

Pause for a minute to think about how you might use the `s///` operator to swap two text strings, so that *lion* becomes *tiger* and *tiger* becomes *lion*. Why wouldn't the following lines work?



```
#!/usr/bin/perl
while (<>) {
    s/lion/tiger/g; # Replace all lions with tigers
    s/tiger/lion/g; # Replace all tigers with lions
    print;
}
```

The problem is that the `s///` operations are executed in sequence. First, all *lions* become *tigers*, and then all *tigers* (including the former *lions*) become *lions*: There won't be any *tigers* left at all. Oops.

You'll fare better if you transform *lions* into a string that isn't in the text first. Let's say the word *tigon* doesn't appear anywhere in the text. Then you can swap *lions* and *tigers* by transforming *lion* into *tigon*, then *tiger* into *lion*, and finally *tigon* into *tiger*, as shown in Listing 2-4.

Listing 2-4 *tigons*: Using `s///` to swap two words

```
#!/usr/bin/perl -w
while (<>) {
    s/lion/tigon/g; # Replace all lions with tigons
    s/tiger/lion/g; # Replace all tigers with lions
    s/tigon/tiger/g; # Replace all tigons with tigers
    print;
}
```

The `-n` and `-p` Command-Line Flags

You can wrap an implicit *while* loop around your script by providing the `-n` flag on either the first line of your program or on the command line when you call perl.

```
#!/usr/bin/perl -n
s/lion/tiger/g;
print;
```

is exactly the same as

```
#!/usr/bin/perl
while (<>) {
    s/lion/tiger/g;
    print;
}
```

Or, if you'd prefer a one-liner, you can use the `-e` flag as well. Just say

```
% perl -n -e 's/which/that/g; print;' your_filename
```

or combine the flags:

```
% perl -ne 's/which/that/g; print;' your_filename
```

If you use the `-p` flag instead of `-n`, the loop supplies the `print()` for you.

```
% perl -pe 's/which/that/g' your_filename
```

How's that for brevity?

Chapter 2, Perl 5 Interactive Course

LESSON 2

METACHARACTERS AND SPECIAL CHARACTERS

The last lesson introduced regular expressions by showing you how to substitute one string for another. But regular expressions don't have to be literal strings. Listing 2-5 shows a program called *cleanse* that removes extraneous whitespace: All consecutive spaces, tabs, or newlines are replaced by a single space.

Listing 2-5 *cleanse*: Using `\s` and `+`

```
#!/usr/bin/perl -w
while (<>) {
    s/\s+/ /g; # Replace one or more spaces with a single space
    print $_, "\n";
}
```

There are two new building blocks here: the `\s` metacharacter and the `+` special character. *cleanse* combines them, resulting in the following behavior:

```
% cleanse
```

```
RESULT Here issometext withextrawhitespace
```

```
Here is some text with extra whitespace
```

```
And here's some more.
```

```
And here's some more.
```

The `\s` and `\S` Metacharacters

`\s` matches any whitespace, including spaces, tabs, and newlines.

`\S` matches anything that doesn't match `\s`.

`\s` is the first metacharacter you've encountered. A *metacharacter* is a symbol with a special meaning inside a regular expression: In the case of `\s`, that meaning is "any whitespace." You'll see more metacharacters later in the chapter.

So if `\s` stands for any whitespace character, what's `\S`? There's a simple rule: Capitalized metacharacters are the opposite of their lowercase counterparts. So `\S` represents anything that isn't whitespace, such as a letter, a digit, or a punctuation mark.

```
/\s\s/
```

matches any two whitespace characters: two spaces, or a space and a tab, or a newline and a space, and so on.

```
/hello\sthere/
```


matches *hello*, followed by a space or a tab or a newline, followed by *there*.

`/\S\S\s/`

matches any two non-whitespace characters followed by one character of whitespace: "we" or "us\t" or "64\n".

`\s+` has two metacharacters: `\s` and `+`.

Metacharacters `+`, `*`, `?`, and `{,}`

`+` means one or more.

`*` means zero or more.

`?` means zero or one.

`{a,b}` means at least *a* but not more than *b*.

"One or more what?" you ask. One or more of whatever precedes the symbol. In *cleanse*, that was an `\s`, so `\s+` means one or more whitespace characters.

Kleene Plus: `+`

The `+` special character (formally called "Kleene plus"--*Kleene* is pronounced "kleeneh") means "one or more." As you saw in *cleanse*, `s\s+/g` replaces any consecutive spaces with a single space.

`/a/`

means one *a*, but

`/a+/`

means one or more *a*'s, and

`/\s/`

means one whitespace character, but

`/\s+/`

means one or more whitespace characters.

Kleene Star: `*`

`*` is like `+` but means "zero or more" instead of "one or more."

`/\s*/`

means zero or more whitespace characters. So

`/hmm+/`

matches *hm* and *hmm* and *hmmm* and *hmmmm*, whereas

`/hmm*/`

matches all those as well, but also matches a single *h*, and

`/b*/`

matches every string because all strings contain zero or more *b*'s. Do not move your eyes away from this page until that makes sense. Most faulty regular expressions can be traced to misuse of `*`.

Curly Braces

What if you want no less than two and no more than four *m*'s? Then you want curly braces, which match a limited number of occurrences, in contrast to + and *, which are unbounded.

`/hm{2,4}/`

matches *hmm* and *hmmm* and *hmmmm*. It's equivalent to `/hmm|hmmm|hmmmm/`.

You can set a lower bound without an upper bound:

`/hm{3,}/`

matches *hmmm* and *hmmmm* and *hmmmmm*... (It's equivalent to `/hmmmm*/` and `/hmmm+/. Do you see why?)`

You CAN'T set an upper bound without a lower bound: `/hm{,3}/` won't do what you want!

Question Marks

? means *zero or one*. You can use the ? special character as a shorthand for {0,1}.

`/Joh?n/`

matches *Jon* and *John*: 'Jo' followed by 0 or 1 *h*'s followed by an *n*. (Question marks that follow Kleene stars, Kleene pluses, or other question marks have a special meaning: Check out the section titled "Greediness" in Lesson 7.)

The . Metacharacter

Here's another special character: the dot. Often, you'll want to express "any character" in your regexes. You can do that with the . metacharacter, which matches anything except a newline.

`/./` matches any character (except newline).

`/.../` matches any three characters (except newlines).

`/.*/` matches any number (including zero) of characters (except newlines).

. always matches exactly one character, so `/the./` matches *there* and *their* and *theta* and *theme*, but not *the* or *then*. If you wanted to match those as well, you could do it with the special characters you just learned, such as

`/the.?.?/`

or

`/the.{0,2}/`

Experiment!

Let's say a contract has been e-mailed to you and you want to make four changes.

1. Replace the word *whereas* with *since*.

2. Replace signature lines with your name (which, coincidentally, happens to be Your Name).

3. Replace all dates with sometime.

4. Replace the phrase one-half with 1/2.

Listing 2-6 shows a program that uses *, +, ?, and {} to get the job done.

Listing 2-6 *destuff*: Using *, ?, and {} in regular expressions

```
#!/usr/bin/perl -wn
s/_{3,}/Your Name/g; # Replaces any series of >= 3
underscores
s/Whereas/Since/g; # Replaces Whereas
s!one-half!1/2!g; # Replaces one-half
# Replaces dates in May
s/May\s\S\S?,\s*\S+/sometime/g;
print;
```

Let's run *destuff* on the text file *contract*, the contents of which are shown here:

```
Whereas the party of the first part, _____, and the
party of the second part, known as That Guy, have entered
into a contract in good faith as of May 4,1996 and: Whereas
_____ was on May 5, 1996 rendered one-half of the
payment and will be rendered the remaining one-half as of May
30, 1997 upon request by That Guy. Whereas and hereunto this
day of May 16, 1996 forthwith undersigned: Signature:
_____
```

Here's the result:

```
% destuff contract
```

RESULT Since the party of the first part, Your Name, and the party of the

```
of sometime and: Since Your Name was on sometime rendered 1/2
of the payment
and will be rendered the remaining 1/2 as of sometime upon
request by
That Guy.
Since and hereunto this day of sometime forthwith
undersigned:
Signature: Your Name
```

This uses all the metacharacters that you've learned so far, as well as one new feature: the ability of s/// to use a delimiter other than a slash. Because *destuff* replaces *one-half* with 1/2, you might be tempted to write *s/one-half/1/2/g*. But because of the extra slash in 1/2, Perl would think "replace one-half with 1, using the modifier 2, and, hey, what's that extra /g doing there?" Luckily, you can use different delimiters to separate the chunks of s///.

```
s!one-half!1/2!g
s#one-half#1/2#g
```

s@one-half@1/2@g

s&one-half&1/2&g

Most nonalphanumeric characters are valid delimiters.

Chapter 2, Perl 5 Interactive Course

LESSON 3

MATCHING

Regular expressions (Figure 2-2) can be used for more than just substitutions. And they don't have to operate on \$_, like every example you've seen so far. Regular expressions can be used simply to match strings, without replacing anything.

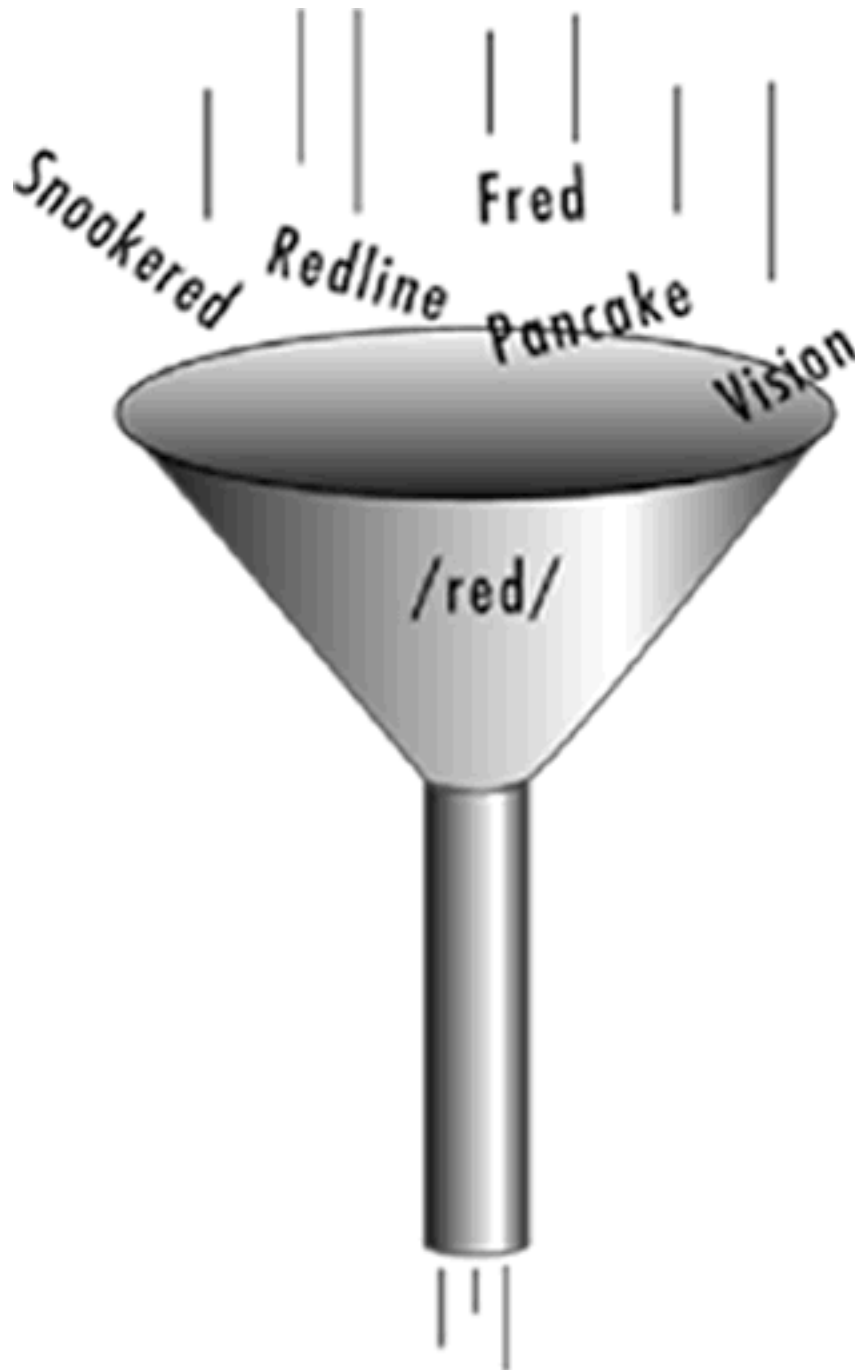


Figure 2-2

Regular expressions can simply *identify* matching substrings without replacing them

The `m//` operator matches a regular expression.

The `m//` Operator

`m/REGEX/MODIFIERS` is true if `$_` matches ***REGEX***, and false otherwise.

In a scalar context, every match returns a value: true if the string (`$_` by default) matched the regular expression, and false otherwise. In a list context, `m//` returns an array of all matching strings. `/g`, `/i`, `/m`, `/o`, `/s`, and `/x` are all optional modifiers discussed later in this chapter.

Like `s///`, `m//` lets you use different delimiters.

`m!flurgh!`

matches *flurgh*.

`m@.\s.@"`

matches any character (except newline), followed by whitespace, followed by any character (except newline).

`m#90?#`

matches both 9 and 90.

You can use any nonalphanumeric, non-whitespace character as a delimiter except `?`, which has a special meaning (described in Chapter 4, Lesson 5). But you should use `/` whenever you can, just because that's what everyone else does. `m//` encourages this practice by letting you omit the *m* if you use `/` as the delimiter.

`/.\./`

matches any character (except newline), followed by a period.

`if (/flag/) { $sentiment++; }`

increments `$sentiment` if `$_` contains *flag*.

`print "Spacey" if /\s/;`

prints *Spacey* if `$_` contains whitespace.

All of the examples you've seen so far for `s///` and `m//` apply regexes to `$_`. You can specify other scalars with the `=~` and `!~` operators.

The =~ and !~ Operators

STRING =~ **m/REGEX/** is true if **STRING** matches **REGEX**.

STRING !~ **m/REGEX/** is true if **STRING** does not match **REGEX**.

STRING =~ **s/REGEX/STRING/** replaces occurrences of **REGEX** with **STRING**.

When used with **m//**, **=~** resembles the equality operators **==** and **eq** because it returns true if the string on the left matches its regex and false otherwise. Used with **s//**, it behaves more like **=** and actually performs the *substitution* on the string.

Some examples:

```
if ($answer =~ /yes/) { print "Good"; }
```

prints *Good* if \$answer contains *yes*.

```
$line =~ s/ //g if $line =~ /.{80,}/;
```

removes consecutive whitespace from \$line if it has more than 80 characters.

```
$string =~ s/coffee/tea/;
```

replaces the first occurrence of *coffee* with *tea* in \$string.

```
print "That has no digits!" if $string !~ /\d/;
```

prints *That has no digits!* if \$string contains no digits.

The program shown in Listing 2-7, *agree*, uses **=~** to identify whether the user typed *yes* or *no*.

Listing 2-7 *agree*: Using **=~** to determine whether a scalar variable matches a pattern

```
#!/usr/bin/perl -w
print `Yes or no? `;
chomp($answer = <>);
if ($answer =~ /y/) { # true if $answer is y or yes or yeah
or no way.
print "I'm glad you agree!\n";
}
```

Here's *agree* in action:

```
% agree
```

```
RESULT Yes or no? yup
```

```
I'm glad you agree!
```

```
% agree
```

```
RESULT Yes or no? nope
```

```
% agree
```

```
RESULT Yes or no? oh yeah
```

```
I'm glad you agree!
```

Remember all those *tickets* programs from Chapter 1? Let's make a new version, *tickets6*, that's a bit more useful; this time, it will allow the user to specify a substring instead of having to supply the full movie title (see Listing 2-8).

Listing 2-8 *tickets6*: Specifying a substring

```
#!/usr/bin/perl
@movies = ( `Flashdance`, `Invasion of the Body Snatchers`,
`King Kong`,
`Raiders of the Lost Ark`, `Flash Gordon` );
print `What movie would you like to see? `;
chomp($_ = <>);
foreach $movie (@movies) {
    $found = ($movie =~ /$_/); # $found is true if $movie
    contains $_
    if ($found) {
        print "Oh! You mean $movie!\n";
        last; # exit the foreach loop
    }
}
if (!$found) { print "HMMMMM. Never heard of $_.\n"; }
```

Pay close attention to the line in bold. First, *\$movie* is matched to the regex */\$_/*. If *\$movie* contains *\$_*, the match returns true and *\$found* is set to 1. If that's the case, the subsequent if statement fires and the *foreach* loop exits.

```
% tickets6
```

```
RESULT What movie would you like to see? Raiders
```

```
Oh! You mean Raiders of the Lost Ark!
```

```
% tickets6
```

```
RESULT What movie would you like to see? Kong
```

```
Oh! You mean King Kong!
```

```
% tickets6
```

```
RESULT What movie would you like to see? body snatchers
```

```
HMMMMM. Never heard of body snatchers.
```

Whoops! *tickets6* didn't recognize *body snatchers* because the capitalization wasn't correct. Wouldn't it be nice if you didn't have to worry about capitalization? Luckily, *s///* and *m//* have the */i* modifier, which tells Perl to perform matches that are not case-sensitive.

The */i* Modifier for *s///* and *m//*

The */i* modifier enables matching that is not case-sensitive.

Had *tickets6* used */\$_/i* in place of */\$_/*, it would have accepted both *body snatchers* and *Body Snatchers* (and *boDy SnAtchERS*...).

If you want to compare two strings ignoring capitalization (as in C's `strcasecmp()`), you could use the `/i` modifier and regex anchors (covered in Lesson 5). Or you could use these functions instead:

The `lc()`, `uc()`, `lcfirst()`, and `ucfirst()` Functions

`lc(STRING)` returns *STRING* with all letters lowercase.

`uc(STRING)` returns *STRING* with all letters uppercase.

`lcfirst(STRING)` returns *STRING* with the first letter lowercase.

`ucfirst(STRING)` returns *STRING* with the first letter uppercase.

Both

```
lc($string1) eq lc($string2)
```

and

```
uc($string1) eq uc($string2)
```

will be true if `$string1` and `$string2` are equivalent strings (regardless of case).

But back to *tickets6*.

```
% tickets6
```

```
RESULT What movie would you like to see? Flash
```

```
Oh! You mean Flashdance!
```

Hmmm...*Flash Gordon* would have been a better choice. Unfortunately, *tickets6* immediately matched *Flash* to *Flashdance*, leaving the *foreach* loop before *Flash Gordon* was tested. The `\b` metacharacter can help.

The `\b` and `\B` Metacharacters

`\b` is a word boundary; `\B` is anything that's not.

So

```
/or\b/
```

matches any word that ends in *or*, and

```
/\bmulti/
```

matches any word that begins with *multi*, and

```
/\b$_\b/
```

matches any string that contains `$_` as an unbroken word. That's what *tickets6* needs.

The *tickets7* program (see Listing 2-9) uses two new regex features, `\d` and parentheses, to ask users to pay for their movie.

Listing 2-9 *tickets7*: Using parentheses and `\d`

```
#!/usr/bin/perl -w
print `That will be $7.50, please: `;
chomp($money = <>);
```

```
# If the user types 6.75, this places 6 into $dollars and 75
into $cents.
($dollars, $cents) = ($money =~ /(\d+)\.(\d\d)/);
$cents += ($dollars * 100);
if ($cents < 750) { print "That's not enough!\n" }
```

The line in bold first matches \$money against the regex `/(\d+)\.(\d\d)/`, which separates the digits before and after the decimal point, sticking the results in \$dollars and \$cents.

```
% tickets7
```

```
RESULT That will be $7.50, please: $6.75
```

```
That's not enough!
```

As you might have guessed, `\d` matches digits.

The `\d` and `\D` Metacharacters

`\d` is any digit (0 through 9); `\D` is any character that isn't.

These statements all do the same thing:

```
print 'That's not a number from 0 to 9!' if ! /\d/;
print 'That's not a number from 0 to 9!' unless m^\d^;
if ($_ !~ /\d/) { print 'That's not a number from 0 to 9!' }
print 'That's not a number from 0 to 9!' unless
/0|1|2|3|4|5|6|7|8|9/;
```

Remember the Perl motto: There's more than one way to do it.

Parentheses

You've probably noticed that parentheses are used to isolate expressions in Perl, just as in mathematics. They do the same thing, and a little bit more, inside regular expressions. When a regex contains parentheses, Perl remembers which substrings matched the parts inside and returns a list of them when evaluated in a list context (e.g., `@array = /(\w+) (\w+) (\w+)/`).

Parentheses

There are two reasons to enclose parts of a regular expression in parentheses. First, parentheses group regex characters together (so that metacharacters such as `+` and `*` can apply to the entire group). Second, they create *backreferences* so you can refer to the matching substrings later.

Grouping

Let's say you want a program to recognize "funny words." A funny word is a word starting with an *m*, *b*, or *bl*, followed by an *a*, *e*, or *o* repeated at least three times, and ending in an *m*, *p*, or *b*. Here are some funny words:

```
meeeeeeep
boooooooooom
```

blaaap

A pattern that matches funny words will have three parts.

1.m|b|bl--The prefix: *m* or *b* or *bl*

2.a{3,}|e{3,}|o{3,}--The middle: 3 or more of *a*'s, *e*'s, or *o*'s

3.m|p|b--The suffix: *m* or *p* or *b*

You can't just concatenate these segments into

```
/m|b|bla{3,}|e{3,}|o{3,}m|p|b/
```

because the wrong chunks get ORed together. Instead, you should group the segments properly with parentheses.

```
/(m|b|bl)(a{3,}|e{3,}|o{3,})(m|p|b)/
```

You can use parentheses yet again to apply the {3,} to each of *a*, *e*, and *o*.

```
/(m|b|bl)(a|e|o){3,}(m|p|b)/
```

Backreferences

Each pair of parentheses creates a temporary variable called a *backreference* that contains whatever matched inside (Figure 2-3). In a list context, each match returns an array of all the backreferences.

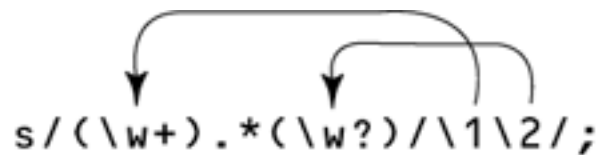


Figure 2-3

Backreferences "refer back" to parenthetical matches

```
($word) = ($line =~ /(\S+)/);
```

places the first word of `$line` into `$word` if there is one. If not, `$word` is set to false.

```
($word1, $word2) = ($line =~ /(\S+)\s+(\S+)/);
```

places the first two words of `$line` into `$word1` and `$word2`. If there aren't two words, neither `$word1` nor `$word2` is set.

```
$word = ($line =~ /(\S+)/);
```

is a scalar context, so the parentheses don't matter. \$word is set to 1 if \$line contains a word and false otherwise.

Backreferences are also stored in the temporary variables \$1, \$2, \$3, and so forth. \$1 contains whatever substring matched the first pair of parentheses, \$2 contains whatever substring matched the second pair of parentheses, and so on, as high as you need.

The regular expression `/(\S+)\s+(\S+)/` contains two pairs of parentheses, so two backreferences are created for any matching string. `twoword`, shown in Listing 2-10, demonstrates how this can be used.

Listing 2-10 *twoword*: Using backreferences

```
#!/usr/bin/perl -w
$_ = shift;
/(\S+)\s+(\S+)/; # two chunks of non-space
print "The first word is $1 \n"; # The substring matching the
first (\S+).
print "The second word is $2 \n"; # The substring matching
the second (\S+).
% twoword "hello there"
```

RESULT The first word is hello

The second word is there

`nameswap`, shown in Listing 2-11, swaps the user's first and last names.

Listing 2-11 *nameswap*: Using backreferences

```
#!/usr/bin/perl -w
print `Type your name: `;
chomp($_ = <>);
s/(\S+) (\S+)/$2 $1/; # Swap the first and second words
print;
```

The two pairs of parentheses create two backreferences: \$1 refers to the first (\S+), and \$2 refers to the second.

```
% nameswap
```

RESULT Type your name: Abraham Lincoln

Lincoln Abraham

\1, \2, \3...

The backreferences \1 and \2 and \3 (and so on) have the same meaning as \$1 and \$2 and \$3, but are valid only inside `s///` or `m///` expressions. In particular, they're valid inside the pattern itself. That lets you represent repeated substrings inside the pattern.

Suppose you want to match any phrase that contains a double word, like *Pago Pago* or *chocolate*

chocolate chip. You could use `(\S+)` to match the first word and `\1` to match the second occurrence of the same word, as shown in Listing 2-12.

Listing 2-12 *couplet*: Using `\1`

```
#!/usr/bin/perl -w
while (<>) {
    print "Echo! \n" if /(\S+)\s\1/;
}
```

couplet matches *double double* but not *Hello world!* or *soy product*.

```
% couplet
```

```
RESULT double double toil and trouble
```

```
Echo!
Hello world!
mime pantomime
A thousand thousand is a million.
Echo!
```

The programs you've seen so far in this chapter use `\S+` to define words. That classifies strings such as "3.14159" and ":-)" and "world!" as words unto themselves, which might upset strict grammarians. They should use `\w`.

The Metacharacters `\w` and `\W`

`\w` matches any letter, number, or underscore.

`\W` matches any character not matched by `\w`.

Listing 2-13 shows *advent*, the world's shortest and hardest adventure game. It lets you type an adventure-style command and then rejects whatever you typed, using `\w` to identify characters belonging to words and three backreferences.

Listing 2-13 *advent*: A simple yet difficult adventure game

```
#!/usr/bin/perl
while (<>) {
    chomp;
    # Match a first word (the verb), any extra words,
    # and an optional last word (the noun).
    ($verb, $extra_words, $noun) = /(\w+)\s?(\w+\s)*(\w+)?/;
    if ($noun) { print "Where is $noun?\n"; }
    else { print "How do I $verb?\n"; }
}
```

advent's regex matches a word, followed by an optional space, followed by zero or more words followed by a space, followed by a final optional word. That's a fairly contorted pattern; what matters is that it plops the first word into `$verb`, the last word into `$noun`, and any other words in the middle into `$other_words`. (For an uncluttered version of this regex, see the end of this chapter's Lesson 7.)

Then, if a \$noun exists, *advent* prints *Where is \$noun?*. Otherwise, it prints *How do I \$verb?*.

```
% advent
```

```
RESULT Look up
```

```
Where is up?
```

```
run
```

```
How do I run?
```

```
read the perl book
```

```
Where is book?
```

Chapter 2, Perl 5 Interactive Course

LESSON 4

TRANSLATIONS

This lesson introduces the `tr///` operator, which is often perceived as related to `s///` and `m//`. However, the similarity is cosmetic only--it has nothing to do with regular expressions. `tr///` translates one set of characters into another set.

The `tr///` Operator

`tr/FROMCHARS/TOCHARS/` replaces *FROMCHARS* with *TOCHARS*, returning the number of characters translated.

A synonym for `tr///` is `y///`, in deference to the UNIX `sed` utility.

FROMCHARS and *TOCHARS* can be lists of individual characters or *ranges*: *a-z* is the range of 26 letters from *a* to *z*.

Some examples:

```
tr/a-z/A-Z/;
```

uppercases every letter of `$_`.

```
tr/pet/sod/;
```

in `$_`, replaces *p*'s with *s*'s, *e*'s with *o*'s, and *t*'s with *d*'s.

```
$string =~ tr/0-9/a-j/;
```

replaces (in `$string`) all 0s with *a*'s, 1s with *b*'s, and so on.

Some people write as if their Caps Lock key were permanently on. YOU KNOW THE TYPE, DON'T YOU?????? Let's mute their enthusiasm a little bit with `tr///` (see Listing 2-14).

Listing 2-14 *mute*: Using `tr///` to translate uppercase into lowercase

```
#!/usr/bin/perl -wn
tr/A-Z/a-z/;
print;
% mute
```

RESULT HEY IZ ANYBODY HERE?

```
hey iz anybody here?
I AM THE GREATEST
i am the greatest
```

`tr/A-Z/a-z/` translates the range of characters from *A* to *Z*, inclusive, to the range of characters from *a* to *z*.

So *A* becomes *a*, *B* becomes *b*, and so forth. Ranges use ASCII order, so `tr/Z-A/z-a/` and `tr/A-5/a-5/` won't do what you want. (If you're not familiar with ASCII, see Appendix C.)

You can mix individual characters with ranges.

```
tr/A-CF/W-Z/
```

translates *A* to *W*, *B* to *X*, *C* to *Y*, and *F* to *Z*.

```
tr/A-C;!X-Z/123##123/
```

translates *A* and *X* to 1, *B* and *Y* to 2, *C* and *Z* to 3, and `;` and `!` to `#`.

If parentheses/brackets/curly braces are used to delimit *FROMCHARS*, then you can use another set of parentheses/brackets/curly braces to delimit *TOCHARS*.

```
tr[0-9][#####]
```

translates all digits to pound signs, and

```
tr{.!}(!.)
```

swaps periods and exclamation points.

If *TOCHARS* is shorter than *FROMCHARS*, the last character of *TOCHARS* is used repeatedly.

```
tr/A-Z/a-k/
```

translates *A* to *a*, *B* to *b*, *C* to *c*,...,*J* to *j*, and *K* through *Z* to *k*.

Because `tr///` returns the number of replaced characters, you can use it to count the occurrences of certain characters. (If you want to count the occurrences of substrings within a string, use the `/g` modifier for `m//`, described in Lesson 8 of this chapter.)

```
$number_of_letters = ($string =~ tr/a-zA-Z/a-zA-Z/);
```

```
$number_of_spaces = ($string =~ tr/ / /);
```

If *FROMCHARS* is the same as *TOCHARS*, `tr///` lets you omit *TOCHARS*.

```
$number_of_digits = ($string =~ tr/0-9//);
```

```
$number_of_capitals = ($string =~ tr/A-Z//);
```

`tr///` has three optional modifiers: `/c`, `/d`, and `/s`.

The `/c`, `/d`, and `/s` Modifiers for `tr///`

`/c` complements *FROMCHARS*.

`/d` deletes any matched but unreplaced characters.

`/s` squashes duplicate characters into just one.

The `/c` modifier complements *FROMCHARS*: It translates any characters that aren't in the *FROMCHARS* set. To change any nonalphabetic characters in `$string` to spaces, you could say

```
tr/A-Za-z/ /c
```

You could remove all capital letters with the `/d` modifier, which *deletes* any characters in *FROMCHARS* lacking a counterpart in *TOCHARS*.

```
tr/A-Z//d;
```

deletes all capital letters.

Finally, you can *squash* duplicate replaced characters with the /s modifier:

```
tr/!//s;
```

replaces multiple occurrences of ! with just one. It's equivalent to s!/+/!/g.

As with s/// and m//, you can combine tr/// modifiers.

```
tr/A-Z//dc
```

deletes any characters complementing the range A-Z.

Chapter 2, Perl 5 Interactive Course

LESSON 5

PATTERN ANCHORS AND SOME MORE SPECIAL VARIABLES

The patterns you've used so far match substrings regardless of their position in the string. In Lesson 3, we used the *agree* (see Listing 2-7) to illustrate simple matching. Listing 2-15 features *agree* again (this time beefed up with *m//* and */i* modifiers).

Listing 2-15 *agree*: An example of matching with ~

```
#!/usr/bin/perl -w
print `Yes or no? `;
chomp($answer = <>);
if ($answer =~ m/y/i) { # true if $answer is y or yes or okay
    or NO WAY!
    print "I'm glad you agree!\n";
}
```

But even with its improvements, we may still be able to fool *agree*.

```
% agree
```

```
RESULT Yes or no? yes
```

```
I'm glad you agree!
```

```
% agree
```

```
Yes or no? not really
```

```
I'm glad you agree!
```

Whoops! Because *not really* contains a *y*, *agree* agreed. We need a way to *anchor* the *y* to the beginning of the string so that only strings starting with *y* match.



Anchor: The ^ and \$ Special Characters

^ matches the beginning of the string.

\$ matches the end.

Use ^ and \$ whenever you can; they increase the speed of regex matches.

`^` and `$` don't match particular characters, just positions. A few examples:

```
/^Whereas/
```

matches any string beginning with *Whereas*.

```
/\s$/
```

matches any string ending with whitespace.

```
$greeting =~ s/Goodbye$/Adieu/;
```

replaces the *Goodbye* at the end of `$greeting`, if there is one, with *Adieu*.

```
/^Turkey stuffing$/
```

matches only one string: *Turkey stuffing*.

Because it's anchored at both beginning and end, `/^Turkey stuffing$/` is the first regular expression in this chapter that matches exactly one string. (But it's useless: If there's only one possible match, you should be using *eq* instead.) *agree2* (see Listing 2-16) matches any `$answer` that begins with a *y* or a *Y*.

Listing 2-16 *agree2*: Using `^` to anchor a pattern

```
#!/usr/bin/perl
print `Yes or no? `;
chomp($answer = <>);
if ($answer =~ /^y/i) { # True if $answer is y or yes or Yo
or yeah or yahoo!
print "I'm glad you agree!\n";
}
% agree2
```

```
RESULT Yes or no? Yes
```

```
I'm glad you agree!
% agree2
```

```
RESULT Yes or no? no way!
```

```
% agree2
```

```
RESULT Yes or no? yup
```

```
I'm glad you agree!
```

The `/m` and `/s` Modifiers for `s///` and `m///`

`/m` lets `^` and `$` match more than once inside a string.

`/s` lets `.` match newlines.

`m//` and `s///` each have two related modifiers: `/m` and `/s`.

Got that? By default, `m//` and `s///` assume that each pattern is a single line of text. But strings can have embedded newlines, e.g., "Line ONE \n Line TWO. \n". The `/m` modifier tells Perl to treat that as two separate strings. *onceupon* (see Listing 2-17) shows the `/m` modifier in action.

Listing 2-17 onceupon: Using the /m modifier

```
#!/usr/bin/perl -w
$story = "Once upon a time\nthere was a bad man, who
died.\nThe end.\n";
$story =~ s/^\w+\b/WORD/mg; # Turn the first word of every
line into WORD
print $story;
```

The `^` in `s/^\w+\b/WORD/mg` matches not just once, but four times: at the beginning of the string, as usual, and after each of the three newlines.

`% onceupon`

RESULT WORD upon a time

WORD was a bad man, who died.
WORD end.

The Metacharacters `\A` and `\Z`

`\A` matches the beginning of a string, like `^`.

`\Z` matches the end of a string, like `$`.

But unlike `^` and `$`, neither `\A` nor `\Z` matches multiple times when `/m` is used.

When using `/m`, `^` matches multiple times: at the beginning of the string and immediately after every newline. Likewise, `$` matches immediately before every newline and at the very end of the string. `\A` and `\Z` are stricter; they match only at the actual beginning and end of the entire string, ignoring the embedded newlines. If we had substituted `\A` for `^` in *onceupon* as follows:

```
$story =~ s/\A\w+\b/WORD/mg;
```

then only the first word would have been uppercase:

```
WORD upon a time
there was a bad man, who died.
The end.
```

Because the `/s` modifier lets `.` match newlines, you can now match phrases split across lines.

```
$lines = "Perl 5 is object\noriented";
print "Hit! \n" if $lines =~ /object.oriented/;
print "Hit with /m! \n" if $lines =~ /object.oriented/m;
print "Hit with /s! \n" if $lines =~ /object.oriented/s;
```

will print only *Hit with /s!*.

`/m` and `/s` aren't mutually exclusive. Suppose you want to remove everything between the outer `BEGIN` and `END` below.

```
We'd like to keep this line
BEGIN
but not this line
or this one,
```

```
or the BEGIN or END.
```

```
END
```

```
This line should stay, too.
```

You can do it using `/m` to match the `BEGIN` and `END` and `/s` to treat the newlines as plain characters.

```
s/^BEGIN.*^END//sm
```

Perl also provides a few special variables for accessing the results of a match.

The Special Variables `$``, `$'`, `$&`, and `$+`

`$`` (or `$PREMATCH`) returns everything before the matched string.

`$'` (or `$POSTMATCH`) returns everything after the matched string.

`$&` (or `$MATCH`) returns the entire matched string.

`$+` (or `$LAST_PAREN_MATCH`) returns the contents of the last parenthetical match.

Each of these variables has two names: a short name (`$``, `$'`, `$&`, `$+`) and a synonymous English name (`$PREMATCH`, `$POSTMATCH`, `$MATCH`, `$LAST_PAREN_MATCH`). You can use the English names only if you place the statement **use English** at the top of your program.

```
$_ = "Macron Pilcrow";  
/cr../;  
print $&;
```

prints *cron*, as does

```
use English;  
$_ = "Macron Pilcrow";  
/cr../;  
print $MATCH;
```

The **use English** statement loads the English module, which makes the longer names available. Modules are the subject of Chapter 8; the English module is discussed in Chapter 8, Lesson 2. Every "punctuation" special variable has an English name (e.g., `$_` can be called `$ARG`); there's a complete list in Appendix H.

`quadrang` (see Listing 2-18) matches the string *quadrangle* to a very simple regex: `/d(.)`.

Listing 2-18 *quadrang*: Using `$``, `$'`, `$&`, and `$+`

```
#!/usr/bin/perl -wl  
$_ = `quadrangle`;  
/d(.)//; # match a d followed by any character  
print $i; # What's before dr  
print $'; # What's after dr  
print $&; # What matched /d(.)/  
print $+; # What matched (.)
```

If it didn't match, all four variables would be empty strings. But *quadrangle* does match, so here's what you see:

```
% quadrang
```

```
RESULT qua  
      angle  
      dr  
      r
```

Why was each variable printed on a separate line? Because *quadrang* uses the **-l** command-line flag.

The -l Command-Line Flag

When the **-l** command-line flag is supplied, Perl adds a newline after every `print()`.

The **-l** flag lets you skip the newlines at the end of every `print()`. It does a little bit more, too: It automatically `chomp()`s input when used with **-n** or **-p** and, if you supply digits afterward, it assigns that value (interpreted as an octal number) to the line terminator `$\` (see Chapter 6, Lesson 4).

Chapter 2, Perl 5 Interactive Course

LESSON 6

ADVANCED REGEXES AND grep()

The last three lessons of this chapter cover a wide assortment of regex features. The most frequently used features are covered first.

Control Characters

You've seen that you can include `\n` in both `print()` statements and regular expressions. That's also true of `\r` (carriage return, which means "move the cursor to the left edge of the page/screen", in contrast to `\n`, which either means "move down one line" or "move to the left edge and down one line," depending on what sort of computer you're using). In fact, most any ASCII control character can be used in both double-quoted strings and regexes. Table 2-2 shows some of the commonly used control characters. A notable exception from this list is `\b`, which is a backspace when printed, but a word boundary inside a regex.

Table 2-2 Some control characters

Control Character	Meaning
<code>\n</code>	Newline
<code>\t</code>	Tab
<code>\a</code>	Alarm (the system beep)
<code>\r</code>	Return
<code>\f</code>	Formfeed
<code>\cD</code>	[CTRL]-[D]
<code>\cC</code>	[CTRL]-[C]

So if you just want to match tabs and not spaces, you can say

```
print "Tabby" if /\t/;
```

Want to annoy everyone within earshot? Put on some earplugs, and type

```
perl -e 'print "\a" while 1'
```

Hexadecimal and Octal Characters Inside Regexes

`\xNUM` is a hexadecimal (base 16) number.

`\oNUM` is an octal (base 8) number. *NUM* must be two or three digits.

If you don't know what hexadecimal or octal numbers are, there's a short description in Chapter 4, Lesson 8. Hexadecimal numbers (that is, numbers in base 16 instead of our more familiar base 10) are represented with a `\x` prefix.

```
\x1a
```

is 26,

```
\x100
```

is 256, and

```
\xfffff
```

is 1048575.

A backslash followed by two or three digits is interpreted as an octal number (that is, a number in base 8): `\00` is ASCII 0 (the null character), and `\107` is ASCII 71: an uppercase G.

You can remove any non-7-bit-ASCII characters from a string with

```
$string =~ tr/\000-\177//cd;
```

because decimal 127 is octal 177.

Because a newline is 10 in ASCII and 10 is a in hexadecimal, you can replace periods with newlines in three ways:

```
s/\./\n/g
```

or

```
s/\./\xa/g
```

or even

```
s/\./\12/g
```

Perl defines five characters to let you tailor the behavior of regexes with respect to capitalization.

`\l`, `\u`, `\L`, `\U`, `\Q`, and `\E`

`\l` lowercases the next character.

`\u` uppercases the next character.

`\L` lowercases everything until the following `\E`.

`\U` uppercases everything until the following `\E`.

`\Q` quotes metacharacters until the following `\E`.

Suppose you want to match *The Perl Journal* without sensitivity to case, except that you want to ensure that the *T* and *P* and *J* are capitalized. Here's how you could do it:

```
/\uThe \uPerl \uJournal/
```

which is the same as

```
/\UT\Ehe \UP\Eerl \UJ\Eournal/
```

Both match *The Perl Journal* and *THE PERL JOURNAL* but not *the perl journal* or *The perl journal*.

`\l` and `\u` and `\L` and `\U` can be used to match particular capitalizations, as *capital* (see Listing 2-19)

demonstrates.

Listing 2-19 *capital*: Using \l, \u, \L, and \U

```
#!/usr/bin/perl -w
$name = "franklin";
while (<>) {
    print "matched \\l \n" if /\l$name/;
    print "matched \\u \n" if /\u$name/;
    print "matched \\L...\E \n" if /\L$name\E/;
    print "matched \\U...\E \n" if /\U$name\E/;
}
```

Lines typed by the user are matched against four variations of the pattern `/franklin/`.

```
% capital
```

```
RESULT franklin
```

```
matched \l
matched \L...\E
Franklin
matched \u
FRANKLIN
matched \U...\E
```

When might you want to use `\Q`? As you've seen, your programs can construct their own patterns:

```
$movie = `w*`
```

or even construct them from user input, as *tickets6* did:

```
chomp($_ = <>);
if ($movie =~ /$_/);
```

Patterns constructed from variables in this way might inadvertently contain metacharacters. Suppose the user types

```
The Color of $
```

or

```
The * Chamber
```

The `$pattern` in *woolf* (see Listing 2-20) contains a question mark, which is interpreted as a special character.

Listing 2-20 *woolf*: A question mark interpreted as a special character

```
#!/usr/bin/perl
$pattern = "Who's Afraid of Virginia Woolf?"; # ? will mean 0
or 1 f's
$movie = "Who's Afraid of Virginia Wool";
if ($movie =~ /$pattern/) { print "Whoops!"; }
```

```
else { print "Okay."; }  
% woOLF
```

RESULT Whoops!

You can implicitly backslash metacharacters by using `\Q` and `\E`, as shown in Listing 2-21.

Listing 2-21 *woOLF2*: Using `\Q` to avoid interpreting regex characters

```
#!/usr/bin/perl -l  
$pattern = "Who's Afraid of Virginia Woolf?";  
$movie = "Who's Afraid of Virginia Wool";  
if ($movie =~ /\Q$pattern\E/) { print "Whoops!"; }  
else { print "Okay."; }
```

Because `$pattern` is between `\Q` and `\E`, the `?` takes its literal meaning instead of its regex meaning.

```
% woOLF2
```

RESULT Okay.

Ranges

You've seen that `tr///` lets you specify a range of characters. You can use ranges with `s///` and `m//` as well, using square brackets (`[]`).

Character Sets

Inside regular expressions, character sets can be enclosed in square brackets.

```
s/[A-Z]\w*//g;
```

removes all words beginning with a capital letter.

Inside brackets, characters are implicitly ORed:

```
s/[aeiou]//g;
```

removes all vowels.

All single-character metacharacters lose their special meanings inside brackets. The hyphen adopts a new meaning, as you've seen, and so does `^`, which when placed at the beginning of the set, complements the rest of the set, so that

```
s/[ ^b]//g;
```

removes anything that's not a b.

```
/[A-Z][ ^A-Z]/;
```

matches any string that contains a capital letter followed by a noncapital.

```
s/[ ^aeiou\s. ]//g;
```

removes everything but vowels, whitespace, and periods.

grep()

It's been a while since we've seen any new functions. In fact, there haven't been any in this chapter until now, because `m//`, `s///`, and `tr///` are actually operators, not functions. But here's one: `grep()`.

The `grep()` Function

`grep(EXPRESSION, ARRAY)` selects any elements from *ARRAY* for which *EXPRESSION* is TRUE.

`grep()` returns a subarray of *ARRAY*'s elements. Often, but not always, *EXPRESSION* is a regular expression.

```
@has_digits = grep(/\d/, @array)
```

`@has_digits` now contains any elements of `@array` with digits.

Within a `grep()`, `$_` is temporarily set to each element of the array, which you can use in the *EXPRESSION*:

```
@odds = grep($_ % 2, @numbers)
```

`@odds` now contains all the odd numbers in `@numbers`.

You can even use `grep()` to emulate a *foreach* loop that modifies array elements.

```
grep(s/a/b/g, @strings)
```

replaces all *a*'s with *b*'s in every element of `@strings`. It also returns a fresh array, which the above statement ignores. Try to avoid this; people who hate side effects consider this poor form because a *foreach* loop is a "cleaner" alternative (not to mention faster).

Movie sequels are never as good as the originals. Luckily, they're usually easy to identify by the Roman numerals in their titles. We'd like to weed them out from an array of movies. We'll do this using `grep()` (the origin of this word is a mystery, but it probably stood for "Generate Regular Expression and Print." That's not what the Perl `grep` does, however). *nosequel* (see Listing 2-22) demonstrates `grep()`.

Listing 2-22 *nosequel*: Using `grep()` to extract array elements

```
#!/usr/bin/perl -w
@movies = ('Taxi Driver', 'Rocky IV', 'Casablanca',
'Godfather II',
'Friday the 13th Part VI', 'I, Claudius', 'Pulp Fiction',
'Police Academy III');
# extract elements that don't (!) contain a word boundary
# (\b) followed by
# one or more I's or V's ((I|V)+), at the end of the string
# ($).
@good_movies = grep( ! /\b(I|V)+$/, @movies);
print "@good_movies";
```

nosequel excludes any movie with *I*'s and *V*'s at the end of its title. (Hopefully, we'll never need to

exclude *X*'s or *L*'s.)

% nosequel

RESULT Taxi Driver Casablanca I, Claudius Pulp Fiction

Chapter 2, Perl 5 Interactive Course

LESSON 7

map() AND ADVANCED REGEXES II

Pat yourself on the back; you've now covered most regular expressions. The remaining two lessons in this chapter cover some regex arcana: features that you should know about, but might well never need.

map()

But first, here's `map()`, a function similar to `grep()`.

The `map()` Function

`map(EXPRESSION, ARRAY)` returns a new array constructed by applying *EXPRESSION* to each element.

A full-fledged block of statements can be used instead of an *EXPRESSION*.

`map()` is kind of like a proactive `grep()`: It applies *EXPRESSION* to each element of *ARRAY*, returning the result.

You could use `int()` as the *EXPRESSION*:

```
@ints = map(int, @numbers);
```

`@ints` now contains the results of applying `int()` to each element of `@numbers`. `@numbers` remains unchanged.

As with `grep()`, you can change array elements by modifying `$_`.

```
@ints = map($_ = int($_), @numbers);
```

replaces each element of `@numbers` with its integer part.

```
@sequels = map($_ .= 'I', @movies)
```

adds an `I` to the end of each element of `@movies`, both modifying `@movies` itself and returning the result. As with `grep()`, some people frown upon this usage because a *for-each* loop is faster and more intuitive to the novice Perl programmer.

Now back to regexes proper. There are still a few modifiers you haven't seen: `/o`, `/e`, and `/x`.

The `/o` Modifier

`/o` tells Perl to interpret the pattern only once.

The `/o` modifier is useful when your pattern contains a variable, but you don't want the pattern to change

whenever the variable does. Normally, if a variable changes, the pattern containing the variable will change as well. The /o modifier instructs Perl to stick with the first value so that even if the variable varies, the pattern stays the same.

The /o modifier is usually used inside a loop. The program *vowel*, in Listing 2-23, prints *pro has a vowel* on the first iteration but not the second because the pattern /\$vowel/o is frozen.

Listing 2-23 *vowel*: Using the /o modifier to "lock" a pattern

```
#!/usr/bin/perl -wl
$vowel = 'a|e|i|o|u'; # vowels for our pattern
foreach ('pro', 'sly') {
    print "$_ has a vowel" if $_ =~ /$vowel/o; # lock our
    pattern,
    $vowel = 'a|e|i|o|u|y'; # so this won't change it.
}
```

On the first iteration, *pro* matches /\$vowel/ because it contains an *o*, as you'd expect. Then \$vowel is changed--a *y* is added to the pattern. But because the /o modifier was used, the regular expression won't notice the change and *sly* won't match.

% vowel

RESULT pro has a vowel

Even if the variable doesn't change, you should still use /o when you can--it'll save time.

Greediness

You've probably noticed that +, *, ?, and {} all match as many characters as they can. Sometimes, that's what you want to happen--but not always. Often, you want the exact opposite behavior: You want those operators to match as *few* characters as possible.

Greediness

+, *, ?, and {} are all greedy: They match as much text as they can. An extra ? curbs their greed.

Now we have four new regex operators, as shown in Table 2-3.

Table 2-3 Minimal matching regex operators

Operator	Meaning
+?	One or more times; as few as possible
*?	Zero or more times; as few as possible
??	Zero or once; zero, if possible
{x, y}?	Between <i>x</i> and <i>y</i> ; as few as possible

When two (or more) greedy operators both want to match the same characters, the leftmost one wins. So

```
`bbbb` =~ /^(.+)b(.+)$/;
```

places *bbb* in \$1 and *b* in \$2 because the left parenthetical match takes precedence. You can make an operator antigreedy (selfless? altruistic? philanthropic?) by appending a question mark.

```
`bbbb` =~ /^(.+?)b(.+)$/;
```

puts *b* in \$1 and *bbb* in \$2.

The /e Modifier

The */e* modifier to the *s///* function evaluates the replacement as a Perl expression.

Suppose you want to replace a pattern with the result of some expression, rather than a simple string. The */e* modifier causes the replacement portion to be interpreted as a full-fledged Perl expression.

```
s/(\d)/$1+1/g
```

adds "+1" to all digits. No big deal. But

```
s/(\d)/$1+1/ge
```

replaces every digit with the expression $\$1 + 1$. So 4 becomes 5 and 618 becomes 729. To have 618 become 619, you'd need *\d+* instead of *\d*.

fahr2cel (see Listing 2-24) demonstrates using */e* to replace a pattern with an evaluated expression that converts Fahrenheit temperatures to Celsius.

Listing 2-24 *fahr2cel*: Using */e* to replace a pattern with an evaluated expression

```
#!/usr/bin/perl -wl
$_ = shift;
s/Fahrenheit (\d+)/`Celsius ` . int(($1-32)*.55555)/e;
print;
```

Note that *Celsius* is concatenated to *int((\$1-32)*.55555)* with the *.* operator.

```
% fahr2cel "Fahrenheit 451"
```

```
RESULT Celsius 232
```

The concatenation is necessary because */e* forces the entire replacement string to be interpreted as an expression. So this wouldn't have worked:



```
s/Fahrenheit (\d+)/Celsius int(($1-32)*.55555)/e;
```

because *Celsius int((\$1-32)*.55555)* isn't a valid Perl expression. (To Perl, it looks more like a function call.)

Unlike other modifiers, it sometimes makes sense to duplicate */e*. Suppose you want to access the value of a variable given its name. You can do that with two instances of */e*, as *colors* (see Listing 2-25) demonstrates.

Listing 2-25 *colors*: Using /e twice

```
#!/usr/bin/perl -l
$red = 141;
$blue = 255;
$green = 92;
@colors = ( `red`, `blue`, `green` );
foreach (@colors) {
    s/(\w+)/`$` . $1/ee; # replaces variable names with values
}
print "@colors"; # prints 141 255 92
```

@colors is first set to (`red`, `blue`, `green`). Then `s/(\w+)/`$` . $1/ee` is applied to each element. The first /e replaces red with the string \$red, which the second /e then replaces with its value: 141.

```
% colors
```

```
RESULT 141 255 92
```

Uncluttering Regexes

Now that you've learned how to construct elaborate regular expressions, it's time to make them a bit more readable using the /x modifier.

The /x Modifier

The /x modifier tells Perl to ignore whitespace inside a pattern.

As you've seen, regular expressions can get pretty messy. Perl 5 lets you clean them up with the /x modifier. With /x, you can use whitespace to separate parts of your pattern for legibility.

```
/ \d+ (\w*) . {4,8} \S \S+ /x
```

is equivalent to

```
/\d+(\w*).{4,8}\S\S+/
```

but much easier to understand.

Remember the "funny words" pattern from Lesson 3? /x can be used to distinguish its three parts without affecting the meaning.

```
/(m|b|bl)(a|e|o){3,}(m|p|b)/
```

is equivalent to

```
/ (m|b|bl) (a|e|o){3,} (m|p|b) /x
```

The /x option also lets you embed comments in your regexes.

```
print "yes" if /\d# this matches if $_ contains a digit/x
```

/x also lets you split regexes across lines:

```
print "Match!" if / \d {2,4} | # two, three, or four digits,
or
[A-Z] {2,4} | # two, three, or four capital
```



```
# letters, or
\S\s {2,4}? | # a non-whitespace character followed
# by two, three, or four whitespace
# characters.
/x;
```

So you can transform the guts of *advent* from the information-dense

```
($verb, $other_words, $noun) =
(/\b+(\w+)\b\s+\b(.*)\b\s?\b(\w+)\b/);
```

to the longer but more understandable

```
($verb, $other words, $noun) = (/ \b (\w+) \b # first word
\s+ \b (.*) \b # second words
\s? \b (\w+) \b # third word
/x
```

Chapter 2, Perl 5 Interactive Course

LESSON 8

ADVANCED REGEXES III

This last lesson covers a few final goodies, including regex extensions, which are features that have been (and will continue to be) added to Perl's regular expressions by the Perl development team.

But first, another modifier: `/g`. You've already seen how `/g` performs global substitutions. Its behavior with `m//` is similar--it matches globally.

The `/g` Modifier for `m//`

In a list context, **`m//g`** returns a list of all parenthetical matches.

In a scalar context, **`m//g`** iterates through the string, returning the "next" matching substring.

The `/g` modifier is pretty intuitive for the `s///` operator: It's just "global search and replace." But its behavior for `m//` is a bit more subtle. In a list context, it returns an array of all the substrings matched by parentheses. If there are no parentheses, it returns a list of all the matched strings, as if there were parentheses around the whole pattern.

In a scalar context, `m//g` iterates through the string, returning TRUE each time it matches and false when it eventually runs out of matches. It remembers where it left off last time and restarts the search at that point.

Listing 2-26, *aristot*, demonstrates `m//g`'s usage in a list context.

Listing 2-26 *aristot*: Using `m//g` in a list context

```
#!/usr/bin/perl -wl
$_ = 'Well begun is half done.';
@words = m/(\w+)/g;
print "@words";
```

The `m//g` expression breaks up `$_` into an array of non-overlapping words, which is then assigned to `@words`.

```
% aristot
```

```
RESULT Well
```

```
begun
is
half
done
```

Listing 2-27 provides an example of `m//g` in a scalar context.

Listing 2-27 *fibonacc*: Using `m//g` in a scalar context

```
#!/usr/bin/perl
$_ = `The first five are 1 1 2 3 5, and the second five are 8
13 21 34 55.`;
while (m/(\d+)/g) { # As long as there's still a number,
print "$1 "; # print the number matched by (\d+)
}
```

Every iteration through the *while* loop grabs the next number from `$_`.

```
% fibonacc
```

```
RESULT 1 1 2 3 5 8 13 21 34 55
```

Note that you *can't* use `m//g` in a scalar context to determine the number of matches:

```
$number = m/(\d+)/g;
print "There were $number matches.;
```

because `m/(\d+)/g` returns a TRUE value, not the contents of the parenthetical match, as you might expect. Instead, you have to use `m//g` in a list context and then count the number of the elements in the array.

```
@numbers = m/(\d+)/g;
print "There were ", scalar(@numbers), " matches.;"
```

Sometimes you'll want to know *where* in the string that `m//g` found its match. That's what `pos()` tells you.

The `pos()` Function

`pos (STRING)` returns the offset of the last **`m//g`** search on *STRING*.

If *STRING* is omitted, `$_` is used. You can assign values to `pos()`, too.

```
pos($string) = 15;
```

makes the next `m//g` search begin at position 15, which is the 16th character.

`pos()` can be used to find the current match position:

```
$_ = `Four score and seven years ago our fathers brought
forth`;
/(\b\w*o\w*\b)/g; # Match words with an 'o' in them.
print pos; # prints 4, the position of the space after
`Four`.
```

The `\G` Metacharacter

The `\G` metacharacter matches only where the previous **`m//g`** left off.

The first `/(\b\w*o\w*\b)/g` in Listing 2-28 matches the first word containing an `o`: *Four*, after which `m//g` is positioned at the end of the word: position 4. The second word containing an `o` is *score*, so after the second `/(\b\w*o\w*\b)/g`, the position becomes 10. Then **`pos = 30`** skips ahead to position 30. At that

point, the next o is in our, so the third match advances just past *our* to 34.

Listing 2-28 *4father*: Using pos() and \G to maneuver m//g

```
#!/usr/bin/perl -w
$_ = `Four score and seven years ago our fathers brought
forth`;
/(\b\w*o\w*\b)/g; # matches words containing an o.
print `Position 1 is `, pos, "\n"; # Prints 4, the position
after 'Four'
/(\b\w*o\w*\b)/g; # match again
print `Position 2 is `, pos, "\n"; # Prints 10, the position
after 'score'
pos = 30; # Skip ahead to character 31
/(\b\w*o\w*\b)/g; # match again
print `Position 3 is `, pos, "\n"; # Prints 34, the position
after 'our'
($rest) = /(\G.+)/g; # Stores everything after pos in $rest.
print "Rest of the string: $rest"; # Prints 'fathers brought
forth'
```

The final m//g match uses \G to place all the characters after the current position into \$rest.

% *4father*

RESULT Position 1 is 4

Position 2 is 10

Position 3 is 34

Rest of the string: fathers brought forth

Regex Extensions

Perl isn't set in stone. It's continually evolving, getting cleaner and faster with each new release. The difference between releases is getting smaller and smaller; there are only a few syntax differences between Perl 4 and Perl 5. Perl 5 has some new functions and operators, and it's object-oriented, but for the most part, Perl 5 is backward-compatible: Most Perl 4 scripts don't have to be changed to run under Perl 5.

But what happens if some new regex metacharacters pop up a year from now? Will your Perl scripts suddenly stop working? No, because Perl supports an *extension syntax* for regular expressions. Any new regex features will (hopefully) use it rather than change how today's regexes are evaluated.

Regex Extension Syntax

Future regular expression features will use (**?char pattern**).

All extensions are enclosed in parentheses and start with a question mark, followed by some character that identifies the particular extension.

There are currently five extensions:

1. **(*?#text*)**--A comment. The text is ignored.
2. **(*?:regex*)**--Allows you to group regex characters without creating a backreference.
3. **(*?=regex*)**--A zero-width positive lookahead assertion: For example, `/\w+(?=\t)/` matches a word followed by a tab, but without including the tab in `$&`.
4. **(*?!regex*)**--A zero-width negative lookahead assertion: `/hand(?!\drawn)/` matches any occurrence of hand that isn't followed by drawn.
5. **(*?imsx*)**--One or more embedded regex modifiers. This is useful if you want one of the four modifiers `/i`, `/m`, `/s`, or `/x` to affect only part of your pattern.

study()

Perhaps the best-named function in Perl (or any other language) is `study()`. Suppose you have a long string and you're going to be performing a lot of pattern matches on it. Perl can `study()` the string, optimizing subsequent matches.

The `study()` Function

`study(STRING)` optimizes Perl's representation of *STRING* for subsequent pattern matches.

Perl has a limited attention span: It can `study()` only one string at a time. And `study()` doesn't actually change the string; it just modifies Perl's internal representation.

```
$dictionary = `a aardvark aardwolf Alcatraz Algernon ammonia
Antares ...
study $dictionary;
```

optimizes `$dictionary` so that matches might take less time. There's no guarantee of a speedup, and conducting the `study()` takes time, so it might not be worth your while. Try it and see--the `Benchmark` module in Chapter 8, Lesson 5, will let you determine whether `study()` helps.

What does `study()` actually do? It builds a data structure based on the frequency distribution of the characters in *STRING*. So the less random your string, the more `study()` will help. English prose will benefit much more than raw database information, which in turn will benefit much more than a string containing digits of $\frac{1}{4}$.

Review

Cybill: I think I can bluff my way through regular expression, if I have an Appendix 1 handy. But the code does look like gibberish sometimes.

Robert: Mine certainly does. And I'm not going to use the `/x` modifier to clean it up, either. I agree with that old saw, "If it was hard to write, it should be hard to read."

Cybill: Concise doesn't always mean faster, though. I was playing around with the `Benchmark` module from Chapter 8, Lesson 5, and found that `/up|down` is actually slower than `/up|/down/`. I would have guessed the other way around.

Chapter 3, Perl 5 Interactive Course



THE OUTSIDE WORLD: FILES AND FILEHANDLES

This chapter is about interactions between your program and the outside world. Mostly, that means files: how to read them; how to write them; how to create, delete, edit, inspect, and lock them.

Filehandles are data structures through which your program can manipulate files. You can think of filehandles as gates: Your program first opens a gate, then sends some data through, and then closes the gate. There are different types of gates, too: one-way and two-way, slow and fast, wide and narrow.

Filehandles are useful for more than just accessing files. Filehandles opened as pipes can send and receive data from programs as well as files. In Lesson 5, you'll see how Perl programs can use pipes to generate e-mail (helpful for mailing lists, bug reports, or vacation programs) and filter incoming e-mail automatically. Filehandles can be used to read from directories, too.

Some of the features (such as symbolic links and file permissions) described in this chapter are specific to UNIX. You'll know which these are by the UNIX symbol in the margin.

LESSON 1

INTRODUCTION TO I/O

So far, you've seen two data types: scalars and arrays. A third data type, called a *filehandle*, acts as a gate between your program and the outside world: files, directories, or other programs. Filehandles can be created in a variety of ways, and the manner of creation affects how data passes between "outside" and "inside." This lesson covers some of the most common tasks associated with files; the details of filehandle creation are deferred until Lesson 2.

Filehandles

In Chapter 1 and Chapter 2, you used empty angle brackets (<>) for two purposes: to read files supplied on the command line and to grab what the user types. In both cases, the angle brackets implicitly open a filehandle. When your program grabs user input, it's using the STDIN filehandle. When your program reads command-line files, it's using the ARGV filehandle. And whenever it print()s something, it's using the STDOUT filehandle. Filehandles are typically all uppercase.

Perl "fills in" empty angle brackets with either STDIN or ARGV for you. But to disambiguate between the two, you can supply the filehandle yourself:

```
$answer = <STDIN>
```

sets \$answer to the next line of standard input.

```
$line = <ARGV>
```

sets \$line to the next line of the command-line files. When your program reads command-line files in this manner, the special variable \$ARGV is set to the current filename.

Empty angle brackets behave like <ARGV> when there's still data to be read from command-line files, and behave like <STDIN> otherwise. That's why

```
while (<>)
```

grabbed user input in *advent*:

```
% advent
```

but read the contract file in *destuff*:

```
% destuff contract
```

STDIN and ARGV are special filehandles that Perl defines for you. But when you make your own files or interact with external programs from your Perl script, you'll create your own.

Angle Brackets

<*FILE*> retrieves data from the *FILE* filehandle.

The context of the angle brackets determines how much data is read:

```
$line = <ARGV>
```

evaluates <ARGV> in a scalar context, retrieving the next line from the ARGV filehandle.

```
@lines = <ARGV>
```

evaluates <ARGV> in a list context, retrieving all the lines from ARGV.

Standard Filehandles

Perl uses the three standard UNIX filehandles: STDIN, STDOUT, and STDERR.

STDIN--Standard input: what the user types--program input

STDOUT--Standard output: what's printed to the user's screen--program output

STDERR--Standard error: error messages printed to the user's screen

STDIN is used for input; STDOUT and STDERR are used for output. In general, your programs will read from STDIN with `<>` and write to STDOUT (and occasionally STDERR) with `print()`.

Reading from a File

Listing 3-1 shows a simple program called *perlcat* (similar to the UNIX *cat* or MS-DOS *type* programs) that demonstrates opening a file for input and reading from that file using angle brackets. (The `open()` function is discussed at length in Lesson 2.)

Listing 3-1 *perlcat*: Using angle brackets to read from a filehandle (in a scalar context)

```
#!/usr/bin/perl -w
# Create a new filehandle called MOVIES connected to the file
movies.lst
open(MOVIES, "movies.lst");
while (<MOVIES>) { # set $_ to each line of <MOVIES>
    print; # ...and print it
}
```

`while(<MOVIES>)` is similar to `while($_ = <MOVIES>)`; each iteration through the loop reads the next line of *movies.lst* (available from ftp.waite.com) and assigns it to `$_`. The loop exits when there are no more lines to be read.

Listing 3-2 shows a utility that prints all lines from a file containing a particular word.

Listing 3-2 *search*: A search program

```
#!/usr/bin/perl -w
($word, $file) = @ARGV;
open(HANDLE, $file);
while (<HANDLE>) {
    print if /\b$word\b/i;
}
```

The *search* program creates a filehandle (imaginatively called `HANDLE`) associated with `$file`. The *while* loop iterates through the lines of `$file`, assigning each to `$_`, which is then printed if it contains `$word`.

So if `$word` is *rocky* and `$file` is *movies.lst*, you might see the following output:

```
% search rocky movies.lst
```

RESULT Rocky

Rocky II

Rocky III

Rocky IV

Rocky Mountain
Rocky Mountain Mystery
Rocky V
The Rocky Horror Picture Show

Exiting a Program

Whenever a program ends, it exits with a particular *exit status*. Usually, if the program is successful, that value is 0; if the program fails in some way, that value is some other integer that helps identify why the program failed.

By default, Perl programs exit with a value of 0 when the end is reached. If you want to exit with a different value or you want to end the program prematurely, use `exit()`.

The `exit()` Function

`exit(NUMBER)` exits the program with the status *NUMBER*.

If *NUMBER* is omitted, 0 is used.

```
exit(5);
```

exits the program with status 5.

```
exit if $value > 99;
```

exits (with status 0) if `$value` is greater than 99.

What happens if the *search* program tries to `open()` a file that doesn't exist? Then the `open()` will fail, returning a FALSE value. Search could use that value to `exit()` so that it doesn't try to read from an invalid filehandle:

```
exit unless open(HANDLE, $file);
```

Even better would be to print an informative error message as the program exits, with the `die()` function.

The `die()` Function

`die(STRING)` exits the program with the current value of the special variable `$!` and prints *STRING*.

If a *STRING* (or an array) is provided, it is printed to `STDERR`; otherwise, *Died* is printed by default.

`die()` uses the system error (contained in the special variable `$!`) as the exit status, so that users (or other programs) will know that your program didn't die of natural causes. `die()`'s message is printed to `STDERR`, which in all likelihood will show up on your screen.

It's always a good idea to use `die()` whenever there's a chance that your `open()` will fail. Listing 3-3 adds it to the *search* program.

Listing 3-3 *search2*: Using `die()`

```
#!/usr/bin/perl -w
($word, $file) = @ARGV;
open(FILE, $file) || die "I can't open $file!\n";
while (<FILE>) {
    print if /$word/i;
}
```

If the `open()` fails, the `die()` statement is executed. For instance, suppose the user mistakenly types *movies.lsr*:

```
% search2 rocky movies.lsr
```

RESULT I can't open movies.lsr! What's that `||` mean? It's OR, a logical operator; the statement on its right is executed only if the statement on its left is false. We'll take a brief interlude from filehandles to discuss `||` and friends.

Logical Operators

Logical operators are like conjunctions in English--they let you change the meaning of an expression (such as the not in "*not* that one!") or specify a relation between two expressions (such as the or in "Are you hungry or thirsty?").

You've probably seen variants in other programming languages, and you've seen one already in Perl:

```
! ($answer == 9)
```

(TRUE if \$answer is not equal to 9.)

Here are some more:

```
(/b/ || ($string eq "quit"))
```

(TRUE if \$_ contains *b* OR if \$string is *quit*.)

```
(( $num >= 0 ) && ( $num <= 10 ))
```

(TRUE if \$num is between 0 AND 10.)

The fourth line of *search2*:

```
open(FILE, $file) || die "I can't open $file!\n";
```

contains two expressions ORed together: the *open* command and the *die* command. When you OR two expressions with `||`, the second expression is evaluated only if the first expression is FALSE. So if the `open()` is successful, it evaluates to 1 and the `die()` is never reached.

The `&&` operator ANDs two expressions: The second expression is evaluated only if the first expression is TRUE.

So neither

```
5 || print "hi"
```

nor

```
0 && print "hi"
```

prints anything, but

```
0 || print "hi"
```

and

```
1 && print "hi"
```

both print *hi*.

The AND, OR, and NOT operations all have spelled-out counterparts:

```
not $answer == 0
$num >= 0 and $num <= 10
/b/ or ($string eq "quit")
```

each of which exhibits the same behavior, but at a lower precedence--they don't stick very tightly to their surrounding arguments. Another way to think of that is that parentheses turn their backs on them.

Here's an example of where precedence makes a difference:

```
print 0 || die "Ack!"
```

is equivalent to `print (0 || die "Ack!")`, so the program immediately dies without printing anything. But

```
print 0 or die "Ack!"
```

is equivalent to `(print 0) || (die "Ack!")`, so it prints 0 instead of dying.

You'll often see this construct at the beginning of programs:

```
$name = shift || "perluser";
```

which sets `$name` to the first command-line argument, if there is one, and `perluser` otherwise.

However, this won't work:



```
$name = shift or "perluser";
```

Because `=` has a stronger precedence than OR, this is the same as `($name = shift) or "perluser"`.

OR's lower precedence is often desirable when using `open()`:

```
open FILE, "vidi.dat" || die;
```

is equivalent to `open(FILE), ("vidi.dat" || die)`, which is legal but doesn't make much sense. The OR operator has lower precedence than the comma, so



```
open FILE, "vidi.dat" or die;
```

is equivalent to `open(FILE, "vidi.dat") or die`, which is much more useful.

Appendix F has a list of all Perl operator precedences. When in doubt, sprinkle your code liberally with parentheses.

The \$! Special Variable

Back to the program: It helps to know why a file couldn't be opened.

The \$! Special Variable

\$! (or **\$ERRNO**, or **\$OS_ERROR**) contains the system error. In a numeric context, it contains the error number. In a string context, it contains the error string.

Until now, you've had to know only about list contexts and scalar contexts. Scalar contexts can be further subdivided into *numeric* context and *string* context.

Why is this helpful? Most UNIX systems associate the error string *No such file or directory* with the code 2. If an `open()` fails because it couldn't find a file, it'll place that error value into **\$!**, which you can access as a number in a numeric context:

```
if ($! == 2) { print "Using default file instead." }
```

or as a string in a string context:

```
if ($! eq "No such file or directory") { print "Using default file instead." }
```

or even

```
if ($! =~ /\bfile\b/) { print "There's a problem with your file." }
```

If you try to open a file that doesn't exist, **\$!** is set to both 2 and *No such file or directory*. If you try to open a file that you're not authorized to read, **\$!** is set to both 13 and *Permission denied*. In general, your operating system will set **\$!**, but you can set it yourself if you want to.

You'll often see `die()` clauses that use **\$!** to convey a little more information to the user. Here's the most common invocation:

```
open(FILE, $file) or die "I can't open $file: $! \n";
```

prints strings of the form:

```
I can't open /etc/passwd: Permission denied.
```

Table 3-1 shows some typical error codes you'll encounter in **\$!**. The full list of name-to-number mappings varies from computer to computer. On UNIX systems, you might find a copy in `/usr/include/sys/errno.h`. Or you could check for yourself:
`foreach (0..31) { $! = $_; print "$!\n" }.`

Table 3-1 Sample values of **\$!**

Numeric Context	String Context
1	Not owner
2	No such file or directory
5	I/O error
13	Permission denied

Sometimes your program might detect an error that is serious enough to warrant an error message, but not so critical that the program should die(). In that case, you'll want your program to warn() the user.

The warn() Function

warn(*STRING*) prints *STRING* like die() but doesn't terminate the program.

Like die(), warn()'s argument doesn't actually have to be a string; it can be an array, in which case the contents of the array are printed. So warn() actually isn't all that different from print(); it just prints to STDERR, the system error stream, which in all likelihood will appear on your screen.

```
warn "Caution! This probably won't work";
```

prints *Caution! This probably won't work* to STDERR (as well as the program name and line number), and

```
open(FILE, "/tmp/index") or warn "Couldn't open index: $!
\n";
```

warns users (with the system error string from \$!) if */tmp/index* couldn't be opened.

warn() doesn't need to have any arguments at all. If your program should be called with command-line arguments, you could warn users who omit them:

```
warn unless @ARGV;
```

Perl then exhibits a little malaise:

```
Warning: something's wrong at ./my_program line 1.
```

Use warn() and die() often. They'll help you debug your programs, and they'll give users a better notion of what might have gone wrong. Some more sophisticated systems for catching and explaining errors can be found in Chapter 10.

Writing to a File

When you print() something, you're using another filehandle: STDOUT (standard output). STDOUT is the default filehandle for output, which is why you don't need to specify it in your print statements. But you could, if you wanted to:

```
print STDOUT "Hello there!";
```

prints *Hello there!*.

Note that there is no comma between the filehandle and the first argument.

To write to a file, open() it as before, inserting a > character to tell Perl that you'll be dumping information into that file. (The > has the same meaning in Perl as it does in UNIX shells and MS-DOS.)

openfile, shown in Listing 3-4, opens a file for writing and then uses that filehandle with print().

Listing 3-4 *openfile*: Opening and writing to a file

```
#!/usr/bin/perl -w
$file = "open.out";
open (FILE, (integral)>$file(integral)) || die "Can't write
to $file: error $!\n";
print FILE "This is the first line of $file.\n"; # note: no
comma!
```

If successful, *openfile* creates a new file named *open.out*, which you can view with UNIX *cat* or MS-DOS *type*, or even Perl:

```
perl -e `open(FILE, "open.out"); while (<FILE>) { print; }`
```

In-Place Editing

Often, instead of reading in one file and writing out the results to a different file, you'll want to modify a file "in place" so that the same file is used for both reading and writing. Specifying the **-i** flag on the command line places files into in-place editing mode.

The -i Flag (and the \$^I Special Variable)

The **-iEXTENSION** flag tells Perl to edit files (listed on the command line) in place. If an *EXTENSION* is provided, a backup file with that *EXTENSION* is made.

\$^I (or **\$INPLACE_EDIT**) is a special variable containing the *EXTENSION*; setting **\$^I** places the program into in-place editing mode.

(**\$^I** is a scalar whose name begins with a caret. It has nothing to do with [CTRL]-[I]. **\$INPLACE_EDIT** is a synonym for **\$^I**, available if you **use English**.)

```
perl -pi -e `s/bark/woof/g` kennel
```

replaces *bark* with *woof* in the file named *kennel*. No backup file is created.

```
perl -pi.bak -e `s/cool/kool/g` slang
```

replaces *cool* with *kool* in the file *slang*, leaving a copy of the original file in *slang.bak*.

Let's use in-place editing to remove initial whitespace from every line of this file:

```
spacey
A foolish consistency is the hobgoblin of little minds.
Ralph Waldo Emerson
Essays, 1841
```

Use *spacey* as input (and output) for the *despace* program, which enters in-place editing mode by setting **\$^I**, as shown in Listing 3-5.

Listing 3-5 *despace*: Using `$^I` to enter in-place editing mode

```
#!/usr/bin/perl -w
# Enter in-place editing mode, with an extension of `.old`
$^I = `.old`;
while (<>) {
    s/^\s+//; # remove leading whitespace
    print; # prints to the input file, not STDOUT!
}
% despace spacey
```

Normally, `print()` prints to standard output. But under in-place editing mode, `print()` prints to the input file instead:

```
% cat spacey
```

RESULT A foolish consistency is the hobgoblin of little minds.

Ralph Waldo Emerson

Essays, 1841

The original file is saved in *spacey.old*.

You could have used the `-i` flag on the command line instead:

```
perl -i.old -e 'while (<>) { s/^\s+//; print }' spacey
```

or even combined it with the `-p` flag:

```
perl -p -i.old -e 's/^\s+//' spacey
```

Chapter 3, Perl 5 Interactive Course

LESSON 2

READING AND WRITING

You've learned how to read from one file and how to write to another. But what if you want to read and write to the same file? Then you need to `open()` a gate that swings both ways. Without further ado, here's the complete usage for `open()`.

The `open()` Function

`open(FILEHANDLE, STRING)` opens the filename given by *STRING* and associates it with *FILEHANDLE*.

`open()` returns a TRUE value upon success and an UNDEF value otherwise.

Certain characters at the beginning of *STRING* affect how the filehandle is created:

- If *STRING* begins with `<` (or nothing at all), it is opened for reading.
- If *STRING* begins with `>`, it is opened for writing.
- If *STRING* begins with `>>`, it is opened for appending.
- If *STRING* begins with `+` (i.e., `+`, `<+`, or `>+`), it is opened for both reading and writing (or appending).
- If *STRING* is `-`, STDIN is opened.
- If *STRING* is `>-`, STDOUT is opened.
- If *STRING* begins with `-|` or `|-`, your process will `fork()` to execute the piped command. (Piped commands are covered in Lesson 3 of this chapter; `fork()` is discussed in Chapter 9, Lesson 3.)

Here are some typical lines you might use in your programs:

```
open(FILE, ">>$file") or die "Can't append to $file.\n";
open(FILE, "+<$file") or die "Can't open $file for reading
and writing.\n";
open(FILE, "+>$file") or die "Can't open $file for writing
and reading.\n";
```

There's an important difference between `+<` and `>+`. Both forms give you read and write access, but `+<` preserves the data that was in the file, whereas `>+` immediately overwrites the data, just like `>`. That's why one common procedure for modifying a file is to

1. Read in the entire file with `open(FILE, $file)` and `@lines = <FILE>`.
2. Close the filehandle.
3. Operate upon `@lines` (which is in speedy memory) rather than `FILE` (which is probably on a slow disk).
4. Write the new file contents with `open(FILE, ">$file")`.
5. Close that filehandle.

But how do you close files?

The `close()` Function

`close(FILEHANDLE)` closes the file associated with *FILEHANDLE*.

When your program exits, Perl closes all open filehandles. If you're going to `open()` the same *FILEHANDLE* twice inside your program, you don't need to close it in between; Perl will do that for you. So when do you need to `close()` a file? Well, one common misconception is that data is written to a file as soon as you `print()` it. It's not. If you look at a file before it's been closed, you're probably not seeing all the data; you've got to `close()` it to be sure (unless you use the special variable `$|`, described in Lesson 7).

Another reason to use `close()` is that most operating systems (including UNIX and MS-DOS) get a bit clogged: They can have only a certain number of files open simultaneously. A system error (accessible through `$!`) will arise if you try to open too many files without closing some first (unless you use the `FileHandle` module, described in Chapter 8, Lesson 6).

Listing 3-6 shows a program called *dialog* that records an actor's lines from a script using `open()` and `close()`.

Listing 3-6 *dialog*: Recording lines

```
#!/usr/bin/perl -w
print `Who are you? `;
chomp($actor = <>);
print "What's the name of your movie? ";
chomp($movie = <>);
open(MOVIE, ">$movie") or die "Can't write to $movie: $! \n";
print "\n Okay, $actor, let's hear your lines! \n";
print "Type `done` when you're finished. \n";
while (1) { # Loop forever
    $line = <>; # Grab the next line of input
    last if $line eq "done\n"; # Leave the loop when `done`
    print MOVIE $line; # Otherwise, print $line to MOVIE;
}
close MOVIE; # Close MOVIE
```

Say the user types in lines but then forgets to type *done*. Instead, the user presses [CTRL]-[C], aborting the program:

```
% dialog
```

RESULT Who are you? Arnold

```
What's the name of your movie? Quadriceps
Okay, Arnold, let's hear your lines!
Type 'done' when you're finished.
I'll be back.
[CTRL]-[C]
```

Will *Quadriceps* have any data in it? Probably not. Most computers are block-buffered by default. Block-buffered filehandles are flushed (i.e., written to disk) only when a certain amount of data accumulates (a block's worth, perhaps 512 or 8192 bytes) or when the file is closed. Because Arnold aborted *dialog*, the file wasn't closed and his precious lines were probably lost. Three conditions trigger a disk write:

1. Enough data accumulates in the I/O buffer.
2. A `close()` (or other function) flushes the output buffer.
3. The program exits.

What happens if you run *dialog* a second time, using the same movie name? The lines from the first dialog would be lost, because the file is opened write-only, obliterating the previous data.

You can ensure that no data will be lost by opening the file for appending, as shown in Listing 3-7.

Listing 3-7 *dialog2*: Appending data to a file

```
#!/usr/bin/perl -w
print 'Who are you? ';
chomp($actor = <>);
print "What's the name of your movie? ";
chomp($movie = <>);
print "\nOkay, $actor, let's hear your lines! \n";
print "Type 'done' when you're finished. \n";
open(MOVIE, ">>$movie") or die "Can't append to $movie: $!
\n";
while (1) {
    $line = <>;
    last if $line eq "done\n";
    print MOVIE $line;
}
close MOVIE;
```

Moving Around in a File

Now that you know how to open a file for reading, writing, and appending, how can you move around in the file? When you do, how can you tell where you are?

For every filehandle associated with a file (as opposed to a filehandle associated with a pipe or

socket--you'll learn about those in later chapters) a *file pointer* marks a particular position in the file. When you first open a file, the file pointer is set to 0. Subsequent reads or writes advance the file pointer automatically. But sometimes you'll want to move that file pointer around yourself, with `seek()`.

The `seek()` and `tell()` Functions

`seek(FILEHANDLE, POSITION, WHENCE)` moves *FILEHANDLE*'s file pointer to *POSITION* (as measured from *WHENCE*).

`tell(FILEHANDLE)` returns the current file position of *FILEHANDLE*.

`seek()` returns 1 upon success and 0 otherwise. File positions are measured in bytes. *WHENCE* can be 0 for the beginning of the file, 1 for the current position of the filehandle, or 2 for the end of the file. Some examples:

```
seek(FILE, 0, 0);
```

moves to the beginning of the file.

```
seek(FILE, 100, 0);
```

moves to 100 bytes from the beginning of the file.

```
seek(FILE, 80, 1);
```

moves forward 80 bytes.

```
seek(FILE, -10, 1);
```

moves back 10 bytes.

```
seek(FILE, 256, 2);
```

moves to 256 bytes before the end of the file.

```
seek(FILE, 0, 2);
```

```
print tell(FILE);
```

moves to the end of the file and prints the position, yielding the number of bytes in the file.

The *nextline* program (see Listing 3-8) uses `seek()` and `tell()` in combination to return the next full line after a specified place in a file.

Listing 3-8 *nextline*: Using `seek()` and `tell()` to move around a file

```
#!/usr/bin/perl  
($position, $filename) = @ARGV;  
open(FILE, $filename) or die "Can't open $filename: $! \n";  
# move to $position bytes from the beginning of $filename.  
seek(FILE, $position, 0) or die "Can't seek: $! \n";  
$trash = <FILE>; # Get the rest of the line  
$realpos = tell(FILE); # find out the current file position  
$nextline = <FILE>; # grab the next line  
print "The next line of $filename is $nextline.\n";  
print "It actually starts at $realpos.\n";
```

The `seek()` command in *nextline* moves the file pointer to `$position` bytes from the beginning (*WHENCE*

is 0; that's the beginning) of the file. Unless you get lucky, `$position` is now somewhere in the middle of a line. You want to advance to the next line, so *nextline* disposes of the remainder of the line with `$trash = <FILE>`. Then *nextline* uses `tell(FILE)` to find out where the next line really starts, before actually reading it with `$nextline = <FILE>`.

nextline is guaranteed to grab a full line from the file, provided the `$position` argument isn't beyond the end of the file:

```
% nextline 10890 movies.lst
```

RESULT The next line of movies.lst is Shining, The
It actually starts at 10899.

`tell()` tells you the position of the file in bytes. But you can also measure a file by lines as well. It's not quite as easy to move around a file by lines, because to a filesystem, newlines don't have any special meaning--they're just bytes. (In particular, they're bytes containing the value 10; see Appendix C.)

However, you can find out how many lines you've read from a filehandle with the `$.` special variable.

The `$.` Special Variable

`$.` (or `$NR`, or `$INPUT_LINE_NUMBER`) is the current input line number of the last filehandle read.

Until your program reads some file, the value of `$.` is undefined. After the first line of some file has been read, `$.` becomes 1, after the second line, 2, and so on. If you have multiple files on the command line, `$.` won't be reset for every file. Only an explicit `close()` resets `$.` to 0.

`$.` has two English names: `$NR` and `$INPUT_LINE_NUMBER`, both available when you use the English module.

Listing 3-9 shows a program that uses `$.` to tally the number of lines in all command-line files.

Listing 3-9 *numlines*: Using `$.`

```
#!/usr/bin/perl -w
while (<>) {} # Read each line, which increments $.
print "Total number of lines so far: $.\n";
```

The *while* loop in *numlines* doesn't do anything, but it has the side effect of incrementing `$.`:

```
% numlines /etc/passwd /etc/passwd
```

RESULT Total number of lines so far: 101

```
Total number of lines so far: 202
```

```
% numlines /etc/passwd /etc/passwd numlines
```

RESULT Total number of lines so far: 101

```
Total number of lines so far: 202
```

```
Total number of lines so far: 205
```

Shortening Files

You've seen how to extend a file by opening it for appending. If you want to shorten a file, you can open the file for reading, read the data, close the file, open it again for writing, and then write some subset of the data out to the same file.

Or you can simply use the `truncate()` function.

The `truncate()` Function

`truncate(FILE, LENGTH)` truncates *FILE* to *LENGTH* bytes.

FILE can be either a filehandle or a filename, so both

```
truncate("/tmp/myfile", 42);
```

and

```
open(FILE, "+</tmp/myfile");
truncate(FILE, 42);
```

will shrink */tmp/myfile* to 42 bytes.

Suppose you have a file with a bunch of garbage binary data at the end. You want to save only the wholesome 7-bit ASCII text that appears before the first 8-bit character. Listing 3-10 shows *shave*, which will truncate() the file at that point.

Listing 3-10 *shave*: Using `truncate()` to lop off the end of a file

```
#!/usr/bin/perl -w
$filename = shift;
open (FILE, $filename) or die "Can't read $filename: $! \n";
while (<FILE>) {
    last if /[^\000-\177]/;
}
$trunk = tell(FILE);
truncate($filename, $trunk);
close FILE;
```

shave reads lines from *\$filename* until it finds a line that contains a fancy character, truncating the file at that point.

Chapter 3, Perl 5 Interactive Course

LESSON 3

EXTERNAL PROGRAMS AND PIPES

You'll often want your program to make use of the output of external programs, such as UNIX *finger*, MS-DOS *dir*, or another Perl program. In this lesson, you'll learn about methods for accessing the input and output of such programs.

Backquotes

You can execute a command in the operating system and stash the output in a variable by enclosing the command in backquotes. Backquotes perform variable interpolation, just like double quotes.

Backquotes

``COMMAND`` executes *COMMAND* in a subshell (usually */bin/sh*), returning *COMMAND*'s output.

Here's an example of using backquotes to store the date (assuming you have the *Date* program on your computer) in the scalar `$date`:

```
$date = `date`;
```

If you want to parse the output of an `ls -l` command (which yields multiple lines), you could assign the lines to an array, one line per array element:

```
@lslines = `ls -l`;
```

or, on MS-DOS computers,

```
@dirlines = `dir`;
```

system()

The `system()` function is similar to backquotes. To understand the difference between the two, you need to distinguish between the output of a command (what it prints) and the return value of a command (whether it succeeded or failed).

Backquotes return the output of a command; `system()` returns the return value of a command. If an external command prints a value, backquotes will intercept that output, but `system()` will let the command go ahead and print to your program's `STDOUT`.

The system() Function

`system(PROGRAM, ARGS)` executes the *PROGRAM* and waits for it to return.

`system()` returns the exit status of *PROGRAM* multiplied by 256.

You can think of `system()` as a window into the command line. You can execute using `system()`--arguments and all--any program that you might execute from the command line, as shown in Listing 3-11.

Listing 3-11 *systest*: Using `system()` to execute a program

```
#!/usr/bin/perl -w
system `ls`, `-l`; # execute the ls program with argument -l
# MS-DOS users should use system `dir` instead.
```

Compare the following (on an MS-DOS machine; UNIX users should use `ls`):

```
`dir`
```

prints nothing at all.

```
print `dir`
```

prints the directory contents.

```
system "dir"
```

prints the directory contents.

```
print system "dir"
```

prints the directory contents, followed by 0.

``dir`` doesn't print anything; it returns its output. But `system `dir`` lets *dir* do its own printing; it returns *dir*'s return value, which is 0. (Like many commands, *dir* returns 0 on success.)

On UNIX computers, you can use `system()` to call the *stty* program and turn off echoing, which you might want to do when the user enters a password. Listing 3-12 shows a program that uses `system()` this way.

Listing 3-12 *password*: Using `system()` to execute the UNIX *stty* command

```
#!/usr/bin/perl
print "Please enter your password: ";
system `stty`, `-echo`; # Turn echoing off
chomp($password = <>);
system `stty`, `echo`; # Turn echoing back on
print "\nThanks.\n";
```

Here's what happens when *password* is run:

```
% password
```

RESULT Please enter your password: <user enters password>

```
Thanks.
```


Executing Another Perl Program

Suppose you want your Perl program to execute a chunk of Perl code located in another file. You can evaluate the contents of another Perl script with the `do()` operator.

The `do()` Operator

`do(FILENAME)` evaluates the Perl code in *FILENAME*.

Using `do()` is just like copying the external file into your program. So if *doable.pl* contains the following lines:

```
$path = `/tmp`;
```

then the program shown in Listing 3-13 sets `$path` to `/tmp`.

Listing 3-13 *include*

```
#!/usr/bin/perl
do 'doable.pl';
print $path;
```

include executes the contents of *doable.pl*:

```
% include
/tmp
```

Piping Data to and from a Program

What if you want your program to receive data from a process that doesn't provide its output all at once? Or what if you want your program to send data to a process, cogitate a bit, and then send more data? For tasks such as these, you'll want to open a *pipe* to an external program.

Opening a Pipe to a Program

`open(HANDLE, "COMMAND|")` lets you read from the output of *COMMAND*.

`open(HANDLE, "|COMMAND")` lets you write to the input of *COMMAND*.

Both these forms return the process ID (see Chapter 9, Lesson 2) of *COMMAND*.

On some computers, the *who* command prints who's logged on. By opening up a pipe from *who*, you can access its output, as shown in Listing 3-14.

Listing 3-14 *wholist*: Accessing the output of an external program

```
#!/usr/bin/perl -w
open (PIPEFROM, "who|");
while (<PIPEFROM>) {
    print;
```



```
}  
close PIPEFROM;
```

If the `who` command isn't found, `$?` will be set to 256 after the `close()`.

Listing 3-15 demonstrates piping data to an external program, in this case `write`.

Listing 3-15 *autorite*: Sending input to an external program

```
#!/usr/bin/perl -w  
open (PIPETO, "|write the boss\@biz.corp.com");  
print PIPETO "Hey, what's up?\n";  
print PIPETO "You don't say?\n";  
print PIPETO "Huh! Go figure.\n";  
print PIPETO "Listen, I gotta go.\n";  
close PIPETO;
```

You can't pipe data both to and from a command. If you want to read the output of a command that you've opened with the `|COMMAND` form, send the output to a file:

```
open(FILE, "|command > /tmp/output");
```

and then open that file (`/tmp/output`) for reading.

But if you really need to simulate a bidirectional pipe (as you might if your program were interacting with Telnet or FTP), use `Open2`, `Open3`, or `Comm.pl`, which are library files described in Chapter 9.

Pipe Errors

Suppose `COMMAND` doesn't exist. You might think that the `open()` will fail, so that the error could be caught with a `die()` clause:

```
open(PIPETO, "| COMMAND") or die "Pipe failed: $! "
```

But the pipe can still succeed even if the `COMMAND` fails. So even if `COMMAND` doesn't exist, your program will continue merrily on its way, unable to write data to `PIPETO`. (If a program tries to write data to a closed pipe, a `PIPE` signal will be generated; signals are covered in Chapter 9, Lesson 2.)

```
open(PIPEFROM, "COMMAND | ") or die "Pipe failed: $! ";
```

won't do what you want, either. If `COMMAND` doesn't exist, the pipe will still succeed; your program will simply never see any data over `PIPEFROM`.

Only when you `close()` the pipe will you be able to see the exit status of `COMMAND`. Perl places the exit status of the pipe into the special variable `$?`.

The `$?` Special Variable

`$?` (or `$CHILD_ERROR`) contains the status of the last external program.

`$?` is actually the return value of the `wait()` system call, which is equal to $(256 * \text{the exit value}) + \text{whatever signal the process died from, if any}$. (If that doesn't make sense to you, don't worry about it.)

On some UNIX systems, the `ls` command returns a 0 upon success and a 2 if the file couldn't be found.

So

```
@ls_lines = `ls erewhon`;  
print $?;
```

prints 512 if *erewhon* doesn't exist, because 2 is the error code for *No such file or directory* and $2 * 256$ is 512.

Deleting Files

You've seen how to create new files with `open()`. How can you destroy old ones? You can perform that dastardly deed with `unlink()`.

The `unlink()` Function

`unlink(FILES)` deletes the *FILES*, returning the number of files successfully deleted.

```
unlink $filename;
```

deletes `$filename`.

Don't forget to provide the full path:

```
unlink "/var/adm/messages";
```

deletes the file `/var/adm/messages`, but

```
unlink "messsages";
```

deletes the file *messages* from the current directory.

Listing 3-16 shows a program that saves some disk space by `unlink()`ing any files ending in *bak*, *~*, or *old*.

Listing 3-16 *cleanbak*: Using `unlink()`

```
#!/usr/bin/perl -wl  
$file = shift;  
@killfiles = ($file . `~.bak`, $file . `~`, $file . `~.old`);  
foreach (@killfiles) { print "Attempting to delete $_"; }  
$num = unlink @killfiles;  
print "$num file(s) actually deleted.";
```

Given a filename on the command line, *cleanbak* creates an array, `@killfiles`, containing the names of three files to be `unlink()`ed:

```
% cleanbak particle
```

RESULT Attempting to delete particle.bak

```
Attempting to delete particle~  
Attempting to delete particle.old  
2 files actually deleted.
```

Don't try to `unlink()` a directory. It probably won't work and if it does, you'll wish it hadn't. Use `rmdir()`, described in Lesson 8, instead.

Chapter 3, Perl 5 Interactive Course

LESSON 4

UNIT

INTERACTING WITH UNIX

Different operating systems have different ways of manipulating files and filesystems. In this lesson, you'll learn about methods for manipulating symbolic links, how to lock files, and how to change file permissions. These concepts are (mostly) particular to UNIX, so it's a good bet that if you're running Perl on a non-UNIX system, most of these functions won't do much. In that case, you can skip ahead to Lesson 5.

Links

A *symbolic link* (or *symlink*) is a special kind of file that refers to another file. The symlink itself doesn't contain data--it's just a pointer to the location of the real file elsewhere in the filesystem. If you read from or write to the symlink, it behaves like the original file. But if you delete the symlink, the original file won't notice. (Most flavors of UNIX, but not all, support symlinks.)

You can create a symlink, but not a hard link, to a directory.

A *hard link* is like a symlink, except it shares data with the original file. The original filename and the name of the hard link are "separate but equal"--they both point to the exact same spot on your disk. But if you `unlink()` one, the data will remain accessible by the other.

A file vanishes from the filesystem only when (a) it's not currently open and (b) all the hard links have been removed. `unlink()`ing a file just removes one link; if there's a hard link somewhere else, the file won't disappear. (That's why it's called `unlink()` and not `delete()`.)

In UNIX, you can see where a symlink points with the `ls -l` command:

```
% ls -l
-rwsrxxx  4 system root      40960 bigprog
-rwxr-xr-- 2 system perluser 109   original
lrwxrwxrwx 1 system perluser 12    symlink -> original
-rwxrrr-- 2 system perluser 109   hardlink
```

Here you see that *symlink* is a symbolic link to *original*. And it's a pretty good guess that *hardlink* is a hard link to *original*--it has the same size as the original and, hey, it's named *hardlink*.

Perl provides the following operations for creating symlinks and hard links.

The `symlink()` and `link()` Functions

symlink(OLDFILE, NEWFILE) creates a *NEWFILE* symbolically linked to *OLDFILE*.

link(OLDFILE, NEWFILE) creates a *NEWFILE* hard linked to *OLDFILE*.

Both `symlink()` and `link()` return 1 if successful and 0 otherwise, and both produce a fatal error if they're not implemented on your computer. But as long as you're running UNIX, you're probably safe. Listing 3-17 is a demonstration.

Listing 3-17 *linktest*: Creating symbolic links and hard links

```
#!/usr/bin/perl -w
$file = `original`;
symlink($file, (integral)symlink(integral)) or warn "Couldn't make
symlink. \n";
link ($file, (integral)hardlink(integral)) or warn "Couldn't make
hard link. \n";
```

If you now `unlink($file)`, both *symlink* and *hardlink* will still exist, but *symlink* will be useless (because it refers to a nonexistent file), whereas *hardlink* will contain the same data as before.

You can find out where a *symlink* points with `readlink()`. (See Listing 3-18.)

The `readlink()` Function

readlink(SYMLINK) returns the name of the file pointed to by *SYMLINK*.

If *SYMLINK* is omitted, `readlink` uses `$_`.

Listing 3-18 *wheresym*: Using `readlink()` to find out where a symlink points

```
#!/usr/bin/perl -w
symlink("Abbott", "Costello"); # Now Costello is a symlink to
Abbott
print readlink((integral)Costello(integral)), "\n"; # Prints
Abbott
```

Like `symlink()`, `readlink()` generates a fatal error on computers that don't support symbolic links. (You can recover from that error by enclosing the `symlink()` in an `eval()` block, e.g., `eval { symlink('Abbott', 'Costello') }`. `eval()` blocks are described in Chapter 9, Lesson 1.)

Assuming your computer supports symlinks, running *wheresym* yields the file pointed to by *Costello*:

```
% wheresym
```

```
RESULT Abbott
```

because *Costello* now "points" to *Abbott*:

```
% ls -l Costello
```

```
RESULT lrwxr-xr-x 1 perluser system 1 Sep 20 13:18 Costello -> Abbott
```

File Locking

Sometimes two people (or processes) will try to write to the same file at the same time. In UNIX, that's a big problem; often the file loses all its data. If your program writes to a file and there's a chance that two copies of the program might run simultaneously or that someone else might edit the file while you're writing it, you'll want to lock the file to ensure that your program has exclusive access.

There are two ways to lock files. The method of last resort is `fcntl()`, described in Chapter 9. The easy way is with `flock()`.

The flock() Function

flock(FILEHANDLE,OPERATION) performs the locking *OPERATION* on *FILEHANDLE*.

The *OPERATION* is a number. On most systems, the locking operations have the values shown in Table 3-2. Hopefully, your Perl will come with the operation name predefined. Otherwise, you can try substituting the number (e.g., 2) for the name (e.g., LOCK_EX).

Table 3-2 flock() operations

Lock Name	Value	Action
LOCK_SH	1	Creates a shared lock on filehandle
LOCK_EX	2	Waits until the filehandle can be locked exclusively
LOCK_NB	4	Locks immediately or fails (nonblocking)
LOCK_UN	8	Unlocks filehandle

Listing 3-19 shows an ultrasafe way to append data to a file using `flock()`.

Listing 3-19 *addsafe*: Using flock() to lock a file

```
#!/usr/bin/perl -w
use Fcntl 'LOCK_EX', 'LOCK_UN'; # to define LOCK_EX and LOCK_UN
$file = shift;
open (FILE, ">>$file") or die "Can't append to $file: $! \n";
flock(FILE, LOCK_EX); # wait until the file can be locked
seek (FILE, 0, 2); # move to the end of the file
print FILE "Some new data for the end of $file.\n";
flock(FILE, LOCK_UN); # unlock the file
close FILE;
```

After opening the file, *addsafe* attempts to place an exclusive lock on the filehandle so that no other processes will write to it. If another program is already locking the file, *addsafe* waits until the file is unlocked.

Once *addsafe* gets its lock, it moves to the end of the file, adds a line of text, and then removes its lock. So

```
% addsafe movies.lst
```

might return immediately, having written its data. But if another process locks the file, *addsafe* will wait until it can acquire the lock before proceeding to the next statement in the program.

A nonblocking lock is demonstrated in Chapter 13, Lesson 6.

If your computer doesn't support flock(), Perl will emulate it for you using fcntl(), which is described in Chapter 9, Lesson 6.

UNIX Permissions

It's very nice to use flock() to ensure that no one will overwrite your file. But what if you want to keep a file away from prying eyes? What if you don't want other people even reading your file? Then on most UNIX filesystems you need to set the *permission bits*.

UNIX associates a permission with each file. The permission contains several values ORed together, as shown in Table 3-3.

Table 3-3 UNIX permissions

If the File Is	By	Then Permission Will Include the Octal Numbers
Readable	Anyone	00004
Writable		00002
Executable		00001
Readable	Members of the file's group	00040
Writable		00020
Executable		00010
Readable	The file's owner	00400
Writable		00200
Executable		00100
Setuid	(See Chapter 11)	04000
Setgid		02000
Sticky		01000

The permission is represented as a single octal number consisting of the numbers in the right column ORed together, which happens to be the same as adding in this case. For instance, if a file is readable and writable and executable by its group, the permission will be 040 | 020 | 010 = 070. If a file is readable by the owner, the group, and anyone, the permission will be 0400 | 0040 | 0004 = 0444.

The permission can also be represented symbolically, with letters instead of numbers, as demonstrated by the `ls -l` output from earlier in this lesson:

% ls -l					
-rwsrxxx	4	system	root	40960	bigprog
-rwxr-xr--	2	system	perluser	109	original
lrwxrwxrwx	1	system	perluser	12	symlink -> original

-rwxr-- **2** **system** **perluser** **109** **hardlink**

The first column shows the permission for each file. Consider the file *original*, which has permissions **-rwxr-xr--** (the first character isn't really part of the permission; it just indicates whether the entry is a directory (**d**) or a symbolic link (**l**) or a plain file (**-**).)

1. The first three permission characters are the owner's permissions. *original*'s **rw**x means that it's readable, writable, and executable by its owner, *perluser*.
2. The second group of three are the group's permissions. The **r-x** indicates that *original* is readable and executable by the system group.
3. The final three characters are the permissions for everyone else. *original*'s **rr** means that it's readable by everyone.

bigprog's permission bits, **-rws--x--x**, tell you that it's readable and writable by its owner (root), and that it's executable by everyone else. The **s** tells you that it's a setuid program, which means that anyone running the program will temporarily assume its owner's identity. More on that in Chapter 11.

Both UNIX and Perl let you change the permissions of a file with `chmod()` and its ownership with `chown()`.

The `chmod()` and `chown()` Functions

`chmod(MODE, FILES)` changes the permissions of *FILES* to *MODE*.

`chown(UID, GID, FILES)` changes the owner and group of *FILES* to *UID* and *GID*.

Both `chmod()` and `chown()` return the number of files successfully changed. `chown()` is restricted on some of the more security-conscious operating systems. UID and GID are user IDs and group IDs (see Chapter 11, Lesson 1).

You'll usually want to specify the *MODE* as an octal number, for example:

```
chmod (0100, $file)
```

to change *\$file*'s permissions to **--x-----**, and

```
chmod (0777, $file)
```

to change *\$file*'s permissions to **rwxrwxrwx**.

Listing 3-20 shows a simple program called *paranoid* that, when run, will immediately change its own permissions to keep anyone but you from reading, writing, or executing it.

Listing 3-20 *paranoid*: Using `chmod()` to change file permissions

```
#!/usr/bin/perl -w  
print "They're out to get me!\n";  
chmod(0700, (integral)paranoid(integral)); # changes paranoid to  
-rwx-----  
print "I feel much better now.\n";
```

If you want all files you create to restrict certain permissions, you can specify that (in either UNIX or Perl) with `umask()`. The `umask()` contains the default permissions that you want to withhold from your files: A `umask()` of 0 means that any file you create will be readable and writable by everyone. (You need to `chmod()` the file

explicitly to make it executable.)

The umask() Function

umask(*NUMBER*) instructs Perl to "mask out" the file permissions represented by *NUMBER*.

umask() returns the old umask. You can find out your current umask by typing

```
% perl -le 'print umask'
18
```

which prints the umask as a decimal number. It's more helpful to print the umask as an octal number, using printf(), which is discussed at length in Chapter 6, Lesson 4.

```
% perl -le 'printf "%o", umask'
22
```

The 22 permission (which is really 022) indicates that files created by your programs will be readable and writable by you and merely readable by your group (that's the first 2) and everyone else (that's the second 2).

Listing 3-21 shows a program that sets the value of umask() to 0, which makes newly created files readable and writable by everyone.

Listing 3-21 *mask*: Using umask() to change default permissions

```
#!/usr/bin/perl
printf "The current umask is %o", umask, "\n";
open (FILE1, ">The_Mask");
umask 0;
open (FILE2, ">Masks");
```

When you run this program, it creates a file called *The_Mask* with whatever permissions your original umask() specified. Then the umask() is set to 0, which is maximally permissive, making *mask* readable and writable by everyone.

If *perluser*'s umask() is 022, here's what the directory might look like after running *mask*:

```
% ls -l
```

```
-rw-rrr--      1  perluser  system    0   Apr  2 15:10   The_Mask
-rw-rw-rw-      1  perluser  system    0   Apr  2 15:10   Masks
```

utime()

In addition to changing the permissions and ownership of files, you can change their access and modification times with utime().

The utime() Function

utime(*ACCESS*, *MODIFICATION*, *FILES*) sets the access and modification times of *FILES* to *ACCESS* and *MODIFICATION*.

ACCESS and *MODIFICATION* must be times in UNIX format (the number of seconds since January 1, 1970; see `time()` in Chapter 5, Lesson 3). But here's how to set the access time of the file *chronon* one minute into the past and the modification time one minute into the future:

```
utime(time - 60, time + 60, `chronon`)
```

Chapter 3, Perl 5 Interactive Course

LESSON 5

UNIT 3

USING INTERNET APPLICATIONS

One of the great things about Perl is the ease with which you can use external programs. You saw a glimpse of that in Lesson 2; now you'll get a chance to put some of that knowledge to good use. First, you'll learn how to send and receive e-mail from within a Perl program; then you'll see a demonstration of how your Perl program can simulate a human connecting to an Internet service.

Chances are, you spend a big part of your day composing and reading electronic mail. In this lesson, you'll learn how to make your programs send e-mail and how to write programs that can filter or reply to your e-mail as it arrives.

Generating E-Mail from Your Perl Programs

Every Internet e-mail message consists of two parts: a header and a body. The header contains a series of fields and values separated by colons. These identify who the e-mail is from, where it's going, and what the subject of the message is. The body contains the message itself. Here's a simple mail message:

```
From: perluser@waite.com
To: an30879@anon.penet.fi
Subject: This is the last line of the header.
This is the first line of the body.
This is the last line of the body.
```

Say you have a program that's going to run for the next 12 hours. You don't want to stay at work all night, but you're willing to read e-mail from home to check on its progress. So you'd like your program to send you e-mail when something goes wrong. Because we are trying to be optimistic, Listing 3-22 presents a program that e-mails you when something goes right, namely, when Fermat's Last Theorem (a conjecture that there are no positive integers a , b , c , and n that satisfy $a^n + b^n = c^n$ for $n > 2$) is proven FALSE.

Listing 3-22 *fermat*: Sending e-mail from a Perl program

```
#!/usr/bin/perl -w
$your_address = (integral)user@machine.domain.com(integral); #
single quotes are important here!
$my_login = (integral)fermat(integral);
$my_address = (integral)fermat@waite.com(integral); # and here,
because of the @ signs.
$n = 3;
for ($a = 1; $a <= 1000; $a++) {
for ($b = 1; $b <= 1000; $b++) {
$c = (($a ** $n) + ($b ** $n)) ** (1/$n);
```

```

if ($c == int($c)) {
# open a pipe to the mail program /usr/lib/sendmail.
# You may need to change the -t -f$my_login to $my_address.
# You may also need to use an entirely different mail program,
# such as /usr/ucb/mail, in which case the arguments
# will definitely be different.
open (MAIL, (integral)|/usr/lib/sendmail -t
-f$my_login(integral));
print MAIL (integral)From: $my_address \n(integral);
print MAIL (integral)To: $your_address \n(integral);
print MAIL (integral)Subject: Margin scribblings. \n(integral)
print MAIL (integral)\n(integral); # separates the header from
the body
print MAIL "$a to the ${n}th power plus \n";
print MAIL "$b to the ${n}th power equals \n";
print MAIL "$c to the ${n}th power. \n";
close MAIL;
}
}
}

```

Test the lines in Listing 3-22 in bold by themselves, first changing `$my_address` to your e-mail address and then supplying any body you wish between the **print MAIL "\n"** and the **close MAIL**.

This program assumes that your computer can send SMTP (a.k.a., Internet) e-mail, and that `/usr/lib/sendmail` is a mail program on your system. If your computer doesn't have `/usr/lib/sendmail`, substitute the name of some other mail program, such as `/usr/ucb/mail` for Berkeley UNIX systems, or `/bin/mail`, or maybe even just plain *mail*, or *blat* for Windows NT (available from <ftp://ftp.csusm.edu/pub/winworld/nt/>). When in doubt, test the mail program from your command line. If it doesn't work for you, it won't work for Perl.

A `Net::SMTP` module (modules are the subject of Chapter 8) is available on the CPAN (see Appendix B). `Net::SMTP` provides a more general interface to mail facilities so that scripts that generate e-mail can be portable across platforms (unlike *fermat*). `Net::SMTP` is described in issue 1 of *The Perl Journal*.

(In case you're wondering what `${n}th` means: That's just a way of delimiting `$n` so that Perl doesn't interpolate `$nth`.)

When *fermat* finds a solution to $a^n + b^n = c^n$, it opens a pipe to `/usr/lib/sendmail`. The **-t** flag tells *sendmail* that you're composing mail, and the **-f** flag sets the name of the From field. The first three `print()` statements create the header of the mail message. The fourth `print()` adds a blank line, which is necessary to separate the header from the body. The final three `print()` statements constitute the body of the message. The `close()` statement closes the pipe, after which *sendmail* sends the message on its way.

You have new mail.

Filtering Incoming E-Mail

Now that you know how to generate e-mail from a Perl program, you can use Perl to read e-mail and take some action based upon its contents, such as automatically replying to the message or appending it to a mail folder or running a program. This is demonstrated with a program called *mailfilt* that you can modify for your

own purposes.

To use *mailfilt*, you need some way of ensuring that every piece of e-mail you get is sent to the STDIN of this program. The procedure for setting this up will vary from system to system; you should check with someone at your site who knows what to do.

The most common way to do this on a UNIX system with local mail delivery is with a *.forward* file. If you don't have a *.forward* file, create one in your home directory containing the following line:

```
\your_username_goes_here, |~/mailfilt"
```

If you place that file in your home directory and your computer has local mail delivery and your system heeds *.forward* files, then incoming mail will be kept in your normal mail spool as usual (that's what the *\your_username_goes_here* does) and will also be piped through the *mailfilt* program in your home directory.

(E-mail is important and you don't want to lose any. You also don't want to set up a mail loop in which you end up with a program sending mail to itself that sends mail to itself that sends mail to itself. So be careful and test your *.forward* file and your programs as soon as you create them. What works for me might not work for you.)

mailfilt, shown in Listing 3-23, separates every incoming message into header and body. It's looking for two types of messages in particular: e-mail with *perl* in the subject line and messages from your boss (conveniently named *The Boss*). If someone sends you mail about Perl, it's automatically appended to a file called *perlmail*. Whenever your boss sends you mail, *mailfilt* immediately replies, making it seem like you work 24 hours a day.

mailfilt is the longest program you've seen so far in this book, but the length is more than made up for by its usefulness.

Listing 3-23 *mailfilt*: Separating incoming messages

```
#!/usr/bin/perl -w
$header = ``; # $header will contain the header
$body = ``; # $message will contain the body
# Parse each line in the header of the message
while (<>) {
    last if /^$/;
    $header .= $_;
    ($field, $value) = /^(\S*)\s*(.*)/;
    if (length($field) > 1) { $fields{$field} = $value; }
    else { $fields{$field} .= " $value"; }
}
while (<>) { $body .= $_; # Store the rest of the message in
$body
# Append any mail with Perl in the subject line to the file
perlmail
if ($fields{'Subject:'} =~ /\bperl\b/i) {
    open (PERL, ">>perlmail");
    print PERL "Message from $fields{'From:'}\n";
    print PERL $body;
    close PERL;
```

```

}
# If the message is from The Boss, reply to it
if ($fields{'From:'} =~ /\bThe Boss\b/i) {
$myname = "Employee";
open (MAIL, "|/usr/lib/sendmail -t -f$myname");
print MAIL "From: $myname \n";
print MAIL "To: $fields{'From:'} \n";
print MAIL "Subject: Re: $fields{'Subject:'} \n";
print MAIL "\n";
print MAIL "Sir, \n\n";
print MAIL "The depth of your insights \n";
print MAIL "never fails to amaze me. \n\n";
print MAIL "Rest assured that I will devote my full \n";
print MAIL "attention to this matter. \n";
close MAIL;
}

```

Whenever anyone sends you e-mail, your system will launch *mailfilt* and pipe the message to *mailfilt*'s STDIN (assuming the right thing happens with your *forward* file). The first while loop iterates through the header, identifying the fields and values and placing them in the %fields hash (don't worry about how hashes work; you'll learn about them in Chapter 5). The loop stops when a blank line (**last if** /^\$/) is encountered, at which point the program has read all the header and none of the body. The second *while* loop tacks all the following lines onto \$body.

The next *if* statement checks to see whether the subject contains the word perl. If it does, the message is appended to the *perlmail* file. The last if statement also checks to see whether the message is from *The Boss*. If so, *mailfilt* generates an e-mail message in response.

Using an Internet Service

The great thing about pipes is that they don't care what's on the other side. They're just conduits for your data. If the program on the other end of your pipe happens to be *telnet*, or FTP, or some other program that sends bits across the Net, you can read from or write to that program as if it were a file.

The Weather Underground is an Internet service provided by the University of Michigan. It collects data from the U.S. National Weather Service, making the information available to Internet users, who can connect to the service via telnet. The Weather Underground is frequently used in the following way: You telnet to the service (located at um-weather.sprl.umich.edu, port 3000) and type in the city or state abbreviation. It then displays the weather forecast for that area.

The service has a menu-based interface; it expects a person on the other end of the connection. However, you can have a Perl program do the typing for you, through a pipe, as shown in Listing 3-24.

Listing 3-24 *weather*: Using an Internet service

```

#!/usr/bin/perl -w $place = shift || `BOS`; # set $place to
$ARGV[0] or BOS. $host = "um-weather.sprl.umich.edu"; $port =
3000; print "Connecting to $host on port $port\n"; open(WEATHER,
"(sleep 2; echo $place; sleep 2) | telnet $host $port |"); while

```

```
(<WEATHER>) { print; } close WEATHER;
```

The city abbreviation for Boston is BOS--let's see what the Weather Underground predicts:

```
% weather BOS
```

```
Connecting to um-weather.sprl.umich.edu on port 3000
```

```
...intro deleted...
```

```
Temp(F)      Humidity(%)      Wind(mph)      Pressure(in)      Weather
```

```
=====
```

```
33           35%              NORTH at 14    30.12           Clear
```

```
...location info deleted...
```

```
TODAY...SUNSHINE THEN INCREASING CLOUDS LATE IN THE DAY. HIGH  
NEAR 50. WIND BECOMING SOUTH 10 TO 20 MPH.
```

```
TONIGHT...CLOUDY AND BREEZY. SCATTERED SHOWERS AFTER MIDNIGHT.
```

```
LOW 40 TO 45. SOUTH WIND 10 TO 20 MPH. CHANCE OF RAIN 50 PERCENT.
```

```
TUESDAY...WINDY WITH SHOWERS AND POSSIBLY A THUNDERSTORM. HIGH 55  
TO 60. CHANCE OF RAIN 80 PERCENT.
```

When the WEATHER filehandle is opened, it immediately telnetts to the Weather Underground and pipes (**sleep 2; echo \$place; sleep 2**) into it. The UNIX commands *sleep* and *echo* are used by *weather* to imitate someone at the keyboard: After the telnet connection is opened, two seconds elapse and then BOS is sent across the connection. Two more seconds elapse and then the input side of the connection is closed.

Because the open() statement has a vertical bar at its end, you know WEATHER is piped open for reading, so you can access the forecast with <WEATHER>.

If you're familiar with the UNIX programs *at* or *cron*, you can merge this program with the mail generation capabilities of *fermat* to create a program that mails you the forecast every morning before you leave for work.

Chapter 3, Perl 5 Interactive Course

LESSON 6

INSPECTING FILES

Files come in all shapes and sizes: They can be large or small, text or binary, executable or read-protected. Perl provides a whole slew of file tests that let you inspect these and other characteristics.

File tests are funny-looking functions: Each is a hyphen followed by a single letter.

File Tests

-TEST FILE is true only if **FILE** satisfies **TEST**.

FILE can be either a filename or a filehandle.

For example, one file test is **-d**, which tests whether **FILE** is a directory:

```
if (-d "/tmp") { print "/tmp is a directory."; }
```

A few of the file tests return values. These two lines will print the size of the file associated with **FILE**:

```
open (FILE, $file);
print "$file has ", -s FILE, " bytes.";
```

Table 3-4 shows the most commonly used file tests. A full list is in Appendix E.

Table 3-4 The most commonly used file tests

File Test	Meaning
-e	File exists.
-f	File is a plain file.
-d	File is a directory.
-T	File seems to be a text file.
-B	File seems to be a binary file.
-r	File is readable.
-w	File is writable.
-x	File is executable.
-s	Size of the file (returns the number of bytes).
-z	File is empty (has zero bytes).

Listing 3-25 features *inspect*, which performs several tests on a file.

Listing 3-25 *inspect*: Using various file tests

```
#!/usr/bin/perl -w
$file = shift;
if (-e $file) { print "$file exists. \n" }
(-z $file) && print "But it's empty. \n";
print "It's executable. \n" if -x $file;
if (-r $file && -w $file) { print "It's readable and
writable. \n"; }
```

Every file is either -T or -B. Perl looks at the beginning of the file: If there are too many strange characters, it's -B. Otherwise, it's -T.

Empty files satisfy both -T and -B, which is why you'll often want to use -s first:

```
if (-s $file && -T $file) {
print "$file is a text file.";
}
```

rather than

```
if (-T $file) {
print "$file is a text file. But it might be empty.";
}
```

File tests check one characteristic of a file. The `stat()` function, on the other hand, returns an array of 13 file statistics.

The `stat()` and `lstat()` Functions

`stat(FILE)` returns a 13-element array giving the "vital statistics" of *FILE*.

`lstat(SYMLINK)` returns the same array for the symbolic link *SYMLINK*.

FILE can be either a filename or a filehandle. `stat()`'s 13 elements are shown in Table 3-5.

Table 3-5 Fields returned by `stat()`

Index	Value
0	The device
1	The file's inode
2	The file's mode
3	The number of hard links to the file
4	The user ID of the file's owner
5	The group ID of the file
6	The raw device
7	The size of the file

8	The last time the file was accessed
9	The last time the file was modified
10	The last time the file's status changed
11	The block size of the system
12	The number of blocks used by the file

The time values returned by `stat()` (indices 8, 9, and 10) are expressed as the number of seconds since January 1, 1970.

Suppose you want to find the size of a file. Now you know four ways to do it: with `seek()` and `tell()`, with backquotes and `ls -l`, with the `-s` file test, and with `stat()` (see Listing 3-26).

Listing 3-26 *foursize*: Four ways to compute the size of a file

```
#!/usr/bin/perl -w
$filename = shift;
# Method 1: using seek() and tell()
open(FILE, $filename);
seek(FILE, 0, 2); # seek() to the end of the file;
$size = tell(FILE); # then find out where you are

print "Size: $size \n";
# Method 2: using the output of a program, in this case wc.
# A different regex would be used for UNIX ls or MS-DOS dir.
# Quiz: why are the parentheses needed around $size?
$wc = `wc $filename`;
($size) = ($wc =~ /\d+\s+\d+\s+(\d+)/);

print "Size: $size \n";
# Method 3: using the -s file test

$size = (-s $filename); # -s returns the size of the file
print "Size: $size \n";
# Method 4: using stat()
size = (stat($filename))[7]; # the eighth element returned by
stat()

print "Size: $size \n";
```

_, `stat()`, and File Tests

`stat()`, `lstat()`, and all file tests allow you to substitute a lone underscore for the last file accessed with any of those operations. So you can rewrite **inspect** as shown in Listing 3-27.

Listing 3-27 *inspect2*: Using the special filehandle

```
#!/usr/bin/perl -w
$file = shift;
if (-e $file) { print "$file exists. \n"; }
(-z _) && print "But it's empty. \n";
print "It's executable. \n" if -x _;
if (-r _ && -w _) { print "It's readable and writable. \n"; }
```

Chapter 3, Perl 5 Interactive Course

LESSON 7

FILE BITS AND BYTES

Many of the I/O functions you've learned about so far assume that a file is a collection of lines. But your operating system doesn't know about lines; it only knows about bytes. In this lesson, you'll learn about functions tailored for manipulating streams of bytes, as well as some other miscellaneous functions, such as `eof()` and `rename()`.

Testing for "End of File"

The `eof()` function tests whether the file pointer is at the end of the file.

The `eof()` Function

`eof(FILEHANDLE)` returns **1** if *FILEHANDLE*'s file pointer is at the end of the file or if *FILEHANDLE* isn't open.

`eof` performs the same function for the last file read.

`eof()` performs the same function for all the files in `@ARGV` as a whole, returning **1** at the end of the last file.

Be sure to use the right form of `eof`. Parentheses matter! Listing 3-28 will print the last line of each file listed on the command line.

Listing 3-28 *eachlast*: Printing the last line of each file listed on the command line

```
#!/usr/bin/perl
while (<>) {
    print if eof; # true at the end of each file
}
```

Listing 3-29 will print the last line of the last file on the command line.

Listing 3-29 *lastlast*: Printing the last line of the command line

```
#!/usr/bin/perl -w
while (<>) {
    print if eof(); # true at the end of the last file only
}
```

You can also use `eof()` in a range. Listing 3-30 prints every line of a file after (and including) the first line containing *begin*.

Listing 3-30 *cuthere*: Using `eof()` in a range

```
#!/usr/bin/perl -w
while(<>) {
    print if /begin/..eof
}
```

Reading Bytes Instead of Lines

Sometimes you don't want to read a file line by line but byte by byte. You can do this with two quite similar functions: `read()` and `sysread()`.

The `read()` Function

`read(FILEHANDLE, VAR, LENGTH, OFFSET)` reads *LENGTH* bytes from *FILEHANDLE*, placing the result in the scalar variable *VAR* (starting at *OFFSET*).

If *OFFSET* is omitted, `read()` uses 0 by default. *VAR* will grow or shrink to fit the number of bytes actually read. `read()` is essentially the opposite of `print()`.

Listing 3-31 shows a program that will print the first \$numbytes bytes from a file, ignoring newlines.

Listing 3-31 *readbyte*: Using `read()` to read a specified number of bytes from a file

```
#!/usr/bin/perl
($numbytes, $file) = @ARGV;
open(FILE, $file) or die "Can't read $file: $! \n";
read(FILE, $string, $numbytes);
print $string;
```

Note that `read()` will always start from the beginning of the file if you don't specify an *OFFSET*, and that \$. isn't affected by `read()`s. `read()` often actually reads more than *LENGTH* bytes, depending on the buffering provided by your system.

If that's not acceptable and you're willing to forego some of the standard I/O precautions, you can use the `sysread()` and `syswrite()` functions.

The `sysread()` and `syswrite()` Functions

`sysread(FILEHANDLE, VAR, LENGTH, OFFSET)` reads *LENGTH* bytes of data from *FILEHANDLE*, placing the result in the scalar variable *VAR* (starting at *OFFSET*).

`syswrite (FILEHANDLE, VAR, LENGTH, OFFSET)` writes *LENGTH* bytes of data from *VAR* (starting at *OFFSET*) to *FILEHANDLE*.

`sysread()` and `syswrite()` return the number of bytes actually read, or undef if an error occurs. `sysread()` returns 0 when the end of the file is reached.

`sysread()` will grow or shrink `VAR` to fit the number of bytes actually read.

`sysread()` and `syswrite()` are lower-level functions than `read()` and `print()`--don't mix them on the same filehandle.

Listing 3-32 is a short example of `sysread()` and `syswrite()`.

Listing 3-32 *sysio*: Using `sysread()` and `syswrite()`

```
#!/usr/bin/perl
$dots = '.' x 100; # create 100 dots
open(FILE, "+>canvas") or die $!; # open for reading and
writing
syswrite(FILE, $dots, 50); # write 50 bytes (dots)
seek(FILE, 0, 0); # move back to the beginning of the file
sysread (FILE, $text, 50); # read 50 bytes into $text
print $text;
```

`syswrite()` writes the first 50 characters of `$dots` into *FILE*, advancing the file pointer as it does so. When the file pointer is reset with `seek()`, those 50 characters can be read with `sysread()`:

```
% sysio
.....
```

Buffering

One common problem with file I/O arises when programmers don't realize that filehandles are (by default) block-buffered, so that data is written only when a certain amount accumulates. `STDOUT` is partially to blame for this misconception because of its schizophrenic behavior: It's line-buffered (and occasionally unbuffered) when its output is a terminal (e.g., your screen) and block-buffered otherwise. You can change the buffering style with the `$|` special variable.

The `$|` Special Variable

`$|` (or `$OUTPUT_AUTOFLUSH`) sets the type of buffering on the currently selected output filehandle. If `$|` is set to zero, Perl relies on the buffering provided by your operating system. If `$|` is set to anything but zero, Perl flushes the filehandle after every **`print()`**.

How can you change the currently selected filehandle? With the `select()` function.

The `select()` Function

`select(FILEHANDLE)` sets the default output to *FILEHANDLE*, returning the previously selected filehandle.

If no *FILEHANDLE* is supplied, `select()` simply returns the currently selected filehandle.

Whenever any filehandle besides STDOUT is select()ed, the user won't see anything printed to the screen. That's why you'll usually want to select() some filehandle, set \$| to 1 to unbuffer it, and then immediately select(STDOUT) to return to normal afterward:

```
select(FILE); $| = 1; select(STDOUT);
```

Because the \$| = 1 unbuffers *FILE*, writing will be slower but more satisfying, as you'll be able to see the effect of print()s immediately. Consider this sequence of statements:

```
print FILE "This line probably won't be written to disk now.
\n";
select(FILE); $| = 1; select(STDOUT);
print FILE "This line will be written immediately. \n";
select(FILE); $| = 0; select(STDOUT);
print FILE "This line probably won't.";
```

Unbuffering a filehandle slows down I/O. getc(), which reads a single character from a filehandle, slows it down even more.

The getc() Function

getc(*FILEHANDLE*) returns the next character from *FILEHANDLE*.

getc() returns an empty string when the end of the file is reached. If *FILEHANDLE* is omitted, getc() uses STDIN as a default.

Listing 3-33 is a simple little program that prints the first two characters from a file.

Listing 3-33 *twochar*: Using getc()

```
#!/usr/bin/perl -w
open(FILE, shift);
print getc(FILE), getc(FILE);
```

If the */etc/passwd* file begins with *root*, *twochar* will print *ro*:

```
% twochar /etc/passwd
ro
% twochar twochar
#!
```

Unfortunately, you can't use getc() to grab individual keystrokes as they occur. If that's your goal, use the Term::ReadKey module, described in Chapter 9, Lesson 6.

Non-UNIX users get pretty short shrift in this book. Many of the Perl features discussed in this chapter (and in earlier ones) are applicable only under UNIX. But here's a function just for non-UNIX users.

The binmode() Function

binmode(*FILEHANDLE*) places *FILEHANDLE* in binary mode.

binmode() should be called after open() but before any reading or writing occurs. It has no effect under UNIX because UNIX doesn't distinguish between binary and text files.

So

```
open(FILE, 'SETUP.EXE');  
binmode(FILE);
```

prepares *SETUP.EXE* for binary reading.

You've covered a lot of functions in this chapter. Perhaps you're sick of files by now. Or perhaps you're only sick of their names. If that's the case, you can use `rename()` to change them.

The `rename()` Function

`rename(BEFORE, AFTER)` changes the name of *BEFORE* to *AFTER*.

```
rename('Big', 'Batman');
```

changes the name *Big* to *Batman*, and

```
rename 'La_Femme_Nikita', 'Point_of_No_Return';
```

renames *La Femme Nikita* to *Point of No Return*.

Chapter 3, Perl 5 Interactive Course

LESSON 8

ACCESSING DIRECTORIES

You've seen that a file consists of a sequence of bytes. Well, a directory is just a sequence of files. So you can open a directory (and associate it with a directory handle), read from that directory (iterating by files instead of lines), and move from file to file in that directory. (You can't write to a directory, however...what would that mean, anyway?)

If your program needs to search through a directory tree, use the `File::Find` module in Chapter 8, Lesson 6.

Accessing Directories

Just as you `open()` and `close()` files, you `opendir()` and `closedir()` directories.

The `opendir()` and `closedir()` Functions

`opendir(DIRHANDLE, DIRNAME)` opens the directory *DIRNAME*.

`closedir(DIRHANDLE)` closes *DIRHANDLE*.

`opendir()` returns `TRUE` if successful and `FALSE` otherwise.

```
opendir(DIR, "/tmp")
```

opens the */tmp* directory and

```
closedir(DIR)
```

closes it.

Merely opening and closing directories isn't much fun. `readdir()` tells you what's actually in the directory.

The `readdir()` Function

`readdir(DIRHANDLE)` returns the next file from *DIRHANDLE* (in a scalar context) or the rest of the files (in a list context).

If there are no more files in *DIRHANDLE*, `readdir()` returns `UNDEF` in a scalar context and an empty array in a list context.

Listing 3-34 is a simple version of *ls*.

Listing 3-34 *perldir*: Using opendir(), readdir(), and closedir()

```
#!/usr/bin/perl
$dirname = shift;
opendir(DIR, $dirname) or die "Can't open $dirname: $! \n";
@files = readdir(DIR);
closedir(DIR);
foreach $file (@files) {print "$file\n";}
```

Running *perldir* on the fictitious directory from Lesson 4 yields

```
% perldir /book/lesson4dir
bigprog
original
symlink
hardlink
```

Listing 3-35 is a program that removes empty files from a directory. Note when `$dirname/$file` is used instead of `$file`.

Listing 3-35 *killzero*: Using directory functions to remove empty files

```
#!/usr/bin/perl -w
$dirname = shift;
opendir(DIR, $dirname) or die "Can't open $dirname: $! \n";
@files = readdir(DIR);
foreach $file (@files) {
    if (-z "$dirname/$file") {
        print "Deleting $dirname/$file...\n";
        unlink "$dirname/$file";
    }
}
closedir(DIR);
```

The `-z` file test returns TRUE if the file is empty (contains 0 bytes), in which case the file is `unlink()`ed:

```
% killzero /tmp
Deleting /tmp/status...
Deleting /tmp/score...
Deleting /tmp/this_file_has_no_bytes...
```

You can move around directories with `telldir()`, `seekdir()`, and `rewinddir()`.

The `telldir()`, `seekdir()`, and `rewinddir()` Functions

telldir(DIRHANDLE) returns the current position of **DIRHANDLE**.

seekdir(DIRHANDLE, POSITION) sets **DIRHANDLE** to read from **POSITION**, which should be a value returned by `telldir()`.

rewinddir(DIRHANDLE) sets **DIRHANDLE** to the top of the directory.

Making (and Unmaking) Directories

You can create and destroy directories with `mkdir()` and `rmdir()`.

The `mkdir()` and `rmdir()` Functions

`mkdir(DIRNAME, MODE)` creates the directory *DIRNAME* with the UNIX protection specified by *MODE* (modified by the current `umask`).

`rmdir(DIRNAME)` deletes the directory *DIRNAME*, but only if it's empty.

Both `mkdir()` and `rmdir()` return 1 upon success and 0 upon failure (and set `$!`).

(Why isn't there a `renamedir()`? Because `rename()` works just fine on directories.)

Changing Directories

Frequently, you'll write programs that manipulate several files or subdirectories within a parent directory. When you do, you can save a lot of time by specifying pathnames relative to that directory rather than always using absolute pathnames. You can manage this with the `chdir()` function (and, under certain circumstances, the `chroot()` function).

The `chdir()` and `chroot()` Functions

`chdir(DIR)` changes the working directory to *DIR*.

`chroot(DIR)` makes *DIR* the root directory for the current process so that pathnames beginning with `"/` will actually refer to paths from *DIR*. Only the superuser can **`chroot()`**.

`chdir()` and `chroot()` return 1 upon success and 0 otherwise (setting `$!`). `chdir()` changes to the user's home directory if *DIR* is omitted.

There's no way to undo a `chroot()`.

```
chdir `~/tmp`
```

will change the current directory of your Perl program to `/tmp`, so that a subsequent **`open(FILE, ">logfile")`** will create `/tmp/logfile`.

Listing 3-36 is an example that creates files in the user's home directory, regardless of where the program is run.

Listing 3-36 *homedir*

```
#!/usr/bin/perl -w  
chdir(); # cd to the user's home directory  
mkdir(`.secret_dir`, 0777);
```

Globbering

To *glob* a string means to replace certain "wildcard" characters with filenames. You do it every time you type

```
ls *
```

or

```
dir *.*
```

or

```
ls ???
```

on your command line. You can do the same thing from Perl with the `glob()` function.

The `glob()` Function

`glob(STRING)` returns an array of the filenames matching the wildcards in *STRING*.

Before there was a `glob()` function, angle brackets were needed to glob filenames:

```
@files = <*>;
```

places all the filenames in the current directory into `@files`. And

```
while (<*.c>) { print }
```

prints all the filenames ending in `.c`.

You can still use angle brackets if you want, but `glob()` is faster and safer, as shown in Listing 3-37.

Listing 3-37 *perldir2*: Using `glob()`

```
#!/usr/bin/perl -wl
@files = glob('*');
foreach (@files) { print }
```

perldir2 globs `*`, which returns an array of all the files in the current directory:

```
% perldir2
```

```
RESULT bigprog
original
symlink
hardlink
```

Review

Lana: You're a sysadmin, right? Why don't you write a script that sends e-mail to users when they clutter up filesystems with old core files?

John: That's a good idea. Using `readdir()`, I could search through every home directory. But core files won't necessarily be in the top level—they might be buried somewhere inside the the home directory. How can my script search every subdirectory, every subsubdirectory, and so on?

Lana: Well, you could use the *find2perl* program discussed in Chapter 14, or the *File::Find* module in Chapter 8, Lesson 6. But if you wanted to do everything in one program, you'd want a recursive subroutine. We'll learn all about subroutines in the next chapter.

Chapter 1, Perl 4 Interactive Course



DIVIDE AND CONQUER: SUBROUTINES

Some programs are throwaways, written in a few minutes for a very specific task, to be used only once. Others are large, sprawling organisms, interacting with other programs and intended for years of use. The more complicated the task, the larger the program. As programs get larger, they become harder and harder to read. Subroutines make your code easier to understand by allowing you to refer to many lines of code with one: a subroutine call.

Nearly all the programs in this book are tiny because that's the best way to illustrate how a function is used: Minimize the surrounding clutter while maximizing utility. But when you write hundred- and thousand-line Perl programs, you need to organize your program into chunks of statements that correspond to how you think about the problem to be solved. By grouping related statements together and letting you execute those statements with a single expression,

subroutines allow you to divide your programs into chunks of code that correspond to concepts and conquer program intractability. Used as directed, subroutines:

- **Make your code more readable because they group related statements together**
- **Result in better program organization**
- **Make your code shorter**

This chapter starts off with the basics of subroutines: how to declare them and how to get data in and out of a subroutine. It'll also cover *blocks*, which are another way of grouping Perl statements together, and *scoping*, which involves localizing variables to a particular block. Along the way, you'll learn about Perl's mathematical functions.

INTRODUCTION TO SUBROUTINES

Until this chapter, every program you've encountered in this book has been a simple sequence of statements. With the exception of loops and conditionals, every statement has been executed in order. A *subroutine* is a chunk of code that can be placed anywhere in your program and executed by calling that subroutine.

How can you tell when to use a subroutine? If a chunk of code makes sense by itself--in other words, if it performs a self-contained task--it probably merits a subroutine. You should be able to give each of your subroutines a concise and complete name; if you can't, that probably means that you're doing too much or too little in that subroutine. And you should be able to understand what the subroutine does without referring to how the subroutine was called.

Subroutines are defined with the **sub** declaration.

The sub Declaration

sub *NAME* { *STATEMENTS* } defines a subroutine named *NAME*.

A defined subroutine isn't of much use unless you invoke it. Here's a simple subroutine declaration:

```
sub my_subroutine { print `Yo.`; }
```

(Subroutine declarations aren't statements, so you don't need a semicolon at the end.) `my_subroutine()` doesn't do anything until it's invoked:

```
my_subroutine();
```

prints *Yo*.

The very first program in Chapter 1 was *hello*. Listing 4-1 presents a similar program that uses a subroutine.

Listing 4-1 *hellosub*: Calling the hello subroutine

```
#!/usr/bin/perl -w

sub hello {
    print "Hello, world!\n";
}

hello(); # Execute the hello() subroutine
```

There are two parts to *hellosub*: a subroutine declaration that defines what happens when someone calls the subroutine `hello()` and a subroutine call that invokes the subroutine.

```
% hellosub
```

```
RESULT Hello, world!
```

Invoking Subroutines

Sometimes it won't be clear to Perl whether you're calling a subroutine or using a string:

```
$a = some_sub;
```

sets \$a to the value returned by the some_sub() subroutine, but only if the subroutine has been declared prior to that statement. Otherwise, \$a is set to the eight-character string "some_sub". This ambiguity gets you into trouble sometimes. (Strings without quotes are called barewords; to outlaw them, add the statement *use strict "subs"* to your program. See Chapter 8, Lesson 1.)

You can identify subroutines with an &. Scalars always begin with \$ and arrays always begin with @; subroutines sometimes begin with &.

The & Symbol

You can put an & before a name to identify it as a subroutine call.

In Perl 5, the & is required only when the name might refer to something else (such as a variable) or when the subroutine definition appears after the subroutine call and you omit the parentheses. So you might see subroutines invoked as

```
&some_sub
```

or

```
some_sub( )
```

or

```
&some_sub( )
```

or just plain

```
some_sub
```

&s are also handy when you want to give a subroutine the same name as a built-in Perl function. Suppose you wrote a pair of subroutines named open() and close(). Because open() and close() already exist in Perl, you need the &s, as shown in Listing 4-2.

Listing 4-2 *door*: Using & to invoke subroutines

```
#!/usr/bin/perl -wl  
sub open {  
print "The door is now open.";  
}  
sub close {  
print "The door is now shut.";  
}  
&open( );  
&close( );
```

open() and close() are called with & to avoid confusion with Perl's open() and close():

```
% door
```

RESULT The door is now open.

The door is now shut.

Nothing prevents you from declaring subroutines on one line:

```
sub open { print "The door is now open." }
```

Subroutine Output

Many people confuse subroutines and functions, often using *function* to refer to any built-in command provided by Perl and *subroutine* to refer to anything that they write themselves. That's the common usage and that's what we'll use in this book. But the actual distinction is a bit more subtle: A function is a subroutine that returns a value. (So all functions are subroutines, but not all subroutines are functions.)

When you want to specify a return value for a subroutine (and by doing so, make it a function), you can use *return*.

The return Statement

return *EXPRESSION* causes an immediate exit from a subroutine, with the return value *EXPRESSION*.

Listing 4-3 shows a subroutine that returns one scalar, a string.

Listing 4-3 *Jubjub*: Returning a value

```
#!/usr/bin/perl
print `The secret password is `, password(), ".\n";
sub password {
    return "Jubjub";
}
```

The print() statement evaluates password(), yielding a string:

```
% password
```

RESULT The secret password is Jubjub.

Remember the *meander* program from Chapter 1? That's the one that multiplies odd numbers by 3 and adds 1, and divides even numbers by 2, over and over. Here it is again (Listing 4-4), with a subroutine to do the dirty work.

Listing 4-4 *meander2*: Returning either of two values

```
#!/usr/bin/perl -w
$num = shift;
while (1) {
    print "$num ";
    last if $num == 1;
    $num = next_number();
}
```



```

sub next_number {
return (3 * $num + 1) if $num % 2; # return the value (3 *
$num) + 1
return $num / 2; # otherwise, return $num / 2
}

```

Inside the *while* loop, \$num is set to the return value of next_number(), which is 3*\$num + 1 if \$num is odd, and \$num/2 otherwise.

```

% meander2 6
6 3 10 5 16 8 4 2 1

```

You don't always need *return*. Subroutines automatically return the result of the last expression. So the next_number() subroutine in *meander2* could have been written as

```

sub next_number {
return (3 * $num + 1) if $num % 2; # return the value (3 *
$num) + 1
$num / 2; # return the value $num / 2
}

```

Subroutines can return arrays:

```

return @result;
return ("sneaker", "pump", "clog");

```

In *chord* (Listing 4-5), the guess_chord() subroutine returns a three-element array if the user asks for any of the three chords it knows about.

Listing 4-5 *chord*: Returning an array

```

#!/usr/bin/perl -wl
$chord = shift;
@notes = guess_chord();
print "@notes";
sub guess_chord {
return (´C´, ´E´, ´G´) if $chord eq ´C-major´;
return (´D´, ´F#´, ´A´) if $chord eq ´D-major´;
return (´E´, ´G#´, ´B´) if $chord eq ´E-major´;
}

```

The array returned by guess_chord() is copied into @notes and printed:

```

% chord D-major
D F# A
% chord C-major
C E G

```

Chapter 4, Perl 5 Interactive Course

LESSON 2

ARGUMENT PASSING AND SOME MATH

The last two subroutines in Lesson 1 used global variables to determine their return values: The value returned by *meander2*'s `next_number()` subroutine depended on `$num`, and the value returned by *chord*'s `guess_chord()` depended on `$chord`.

Ideally, subroutines shouldn't need to rely on global variables. And because Perl is ideal, it allows you to provide an array of arguments to a subroutine.

You can pass arguments to a subroutine with the `@_` array.

The `@_` Array

Perl places the arguments to a subroutine in the array `@_`.

`@_` is similar to a normal array (`@movies` or `@prices`); it just has a strange name and a few strange behaviors. You can access the first element with `$_[0]`, the second with `$_[1]`, and so forth. Whenever any subroutine is called, its arguments (if any) are supplied in `@_`. So when you provide scalar arguments to a subroutine, you can access the first scalar with `$_[0]`, the second with `$_[1]`, and so on.

Listing 4-6 is a simple subroutine that takes one scalar as input.

Listing 4-6 *square*: Subroutine arguments are stored in `@_`

```
#!/usr/bin/perl -w1
foreach $i (1..10) {
    print "$i squared is ", square($i); # call square() with
    input $i
}
sub square {
    return $_[0] ** 2; # ...which becomes $_[0]
}
```

`square($i)` calls the `square()` subroutine with one argument: `$i`.

```
% square
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
```

```
6 squared is 36
7 squared is 49
8 squared is 64
9 squared is 81
10 squared is 100
```

Of course, because `@_` is an array, you can pass in more than one scalar, as shown in Listing 4-7.

Listing 4-7 *threesum*: Passing multiple values into a subroutine

```
#!/usr/bin/perl -wl
print addthree(7, 11, 21); # call addthree with @_ = (7, 11,
21)
sub addthree {
return $_[0] + $_[1] + $_[2]; # return the sum of the first
three args
}
```

The `addthree()` subroutine returns the sum of the first three arguments in the input array:

```
% threesum
```

```
RESULT 39
```

You can pass any array you like as `@_`. `threeadd` uses `@ARGV` (Listing 4-8).

Listing 4-8 *threeadd*: Passing an array into a subroutine

```
#!/usr/bin/perl -wl
print addthree(@ARGV); # pass in all the command-line args
sub addthree {
return $_[0] + $_[1] + $_[2]; # return the sum of the first
three
}
% 3sum 19.95 5 2
```

```
RESULT 26.95
```

`@_` need not have a fixed length. *average*, shown in Listing 4-9, computes the mean of its arguments, no matter how many there are.

Listing 4-9 *average*: Computing the mean

```
#!/usr/bin/perl -l
print mean(@ARGV) if @ARGV;
sub mean {
foreach $i (@_) { # Loop through the input array
$total += $i;
```

```

}
return $total / ($#_ + 1); # return $total divided by the
number of args
}

```

Wondering what `$#_+1` means? Remember that for any array, `$#array` is the last index of `@array`. So `$#array+1` is the number of elements in `@array` (assuming your program hasn't messed with `$[,` the array base). And `@_` is just another array, so `$#_ +1` is the number of arguments.

(It would be more correct to say `$#_ + $[+ 1` instead of just `$#_ + 1`, in case your program messes with `$[,` but your program probably shouldn't be doing that anyway.)

```
% average 6 7
```

```
RESULT 6.5
```

```
% average 6 7 8 8 10
```

```
RESULT 7.8
```

None of the subroutines you've seen so far accepts more than one type. But you can pass in both a scalar and an array, as long as you pass in the scalar first (Listing 4-10).

Listing 4-10 *prefix*: Passing a scalar and an array to a subroutine

```

#!/usr/bin/perl -wl
# add_prefix takes a scalar ($prefix) and an array (@words),
in that order.
sub add_prefix {
    ($prefix, @words) = @_;
    foreach (@words) { $_ .= $prefix };
    return @words;
}
@colors = ('red', 'blue', 'green', 'white', 'black');
@colors = add_prefix('s', @colors);
print "@colors";

```

The `add_prefix()` subroutine is called with two arguments: the scalar value `'s'` and the array value `@colors`. Inside the subroutine, the first element of `@_` (that's the `s`) is assigned to `$prefix`, and the rest of `@_` (that's `@colors`) is assigned to `@words`. `add_prefix()` then modifies `@words`, adding the prefix to each element, and then returns the new array: `('reds', 'blues', 'greens', 'whites', 'blacks')`. That return value is then assigned to `@colors`, overwriting the old elements.

```
% prefix
```

```
RESULT reds blues greens whites blacks
```

You've seen how to pass multiple scalars and a scalar and an array, but you haven't seen how to pass multiple arrays. That's a bit trickier. Because all arguments are merged into the single array `@_`, you lose

the boundaries between multiple arrays. In Lesson 7 of this chapter, you'll learn two techniques for passing as many arrays as you wish.

A Bit of Math: `sqrt()` and `abs()`

In this lesson and the next, you'll learn about Perl's mathematical operations. They'll be demonstrated using subroutines, of course.

Perl 4, like C, has no built-in function to calculate the absolute value of a number. However, Perl 5, eager to please, provides `abs()`. And though you can find the square root of a number by saying (*NUMBER* ** 0.5), you can also use the `sqrt()` function, which is a tad faster.

`sqrt(NUMBER)` returns the square root of *NUMBER*.

The `sqrt()` and `abs()` Functions

`abs(NUMBER)` returns the absolute value of *NUMBER*.

Both `sqrt()` and `abs()` use `$_` if *NUMBER* is omitted. Taking the square root of a negative number results in a fatal error.

In Listing 4-11, the `absroot()` subroutine ensures that you don't attempt to compute the square root of a negative number: It returns the square root of the absolute value of the input.

Listing 4-11 *absroot*: Using `sqrt()` and `abs()`

```
#!/usr/bin/perl -w
print "Absolute square root: ", absroot(shift), "\n";
# absroot() returns the square root of the absolute value of
its input.
sub absroot {
    return sqrt(abs($_[0]));
}
```

`absroot(shift)` calls `absroot()` with one argument: the result of `shift()`, which is the first command-line argument. The `absroot()` subroutine then returns `sqrt(abs($_[0]))`, the square root of the absolute value of `$_[0]`.

```
% absroot
```

```
RESULT Gimme a number: 4
```

```
Absolute square root: 2
```

```
Gimme a number: -4
```

```
Absolute square root: 2
```

```
Gimme a number: -1000
```

```
Absolute square root: 31.6227766016838
```

Exponential Notation

Perl uses *exponential notation* for very large and very small numbers: 1 million (10 to the sixth power)

can be represented as either 1000000 or 1e+6, 3.5 trillion (3.5 to the twelfth power) can be represented as 3.5e+12, and .000000000000000098 can be represented as 9.8e-16.

Einstein's theory of special relativity tells us that when an object moves, it shrinks relative to someone at rest. The faster it moves, the smaller it gets. You can figure out how much smaller (and make good use of the `sqrt()` function) with the *einstein* program (Listing 4-12).

Listing 4-12 *einstein*: Using the `sqrt()` function

```
#!/usr/bin/perl -w
$light = 3.535488e+12; # speed of light, in feet per hour
print `What is the length of your car, in feet? `;
chomp($length = <>);
print `How fast is it traveling, in feet per hour? `;
chomp($velocity = <>);
print "\nTo an observer at rest, your car is actually ",
    lorentz($length, $velocity), " feet long. \n";
# Computes the Lorentz transform, yielding the actual length
of the car.
sub lorentz {
    $length *= sqrt(1 - ($velocity ** 2)/($light ** 2))
}
```

Let's run *einstein*:

```
% einstein
```

```
RESULT What is the length of your car, in feet? 10
```

```
How fast is it traveling, in feet per hour? 3e+12
To an observer at rest, your car is actually
8.245625087316931 feet long.
```

Now that you've covered special relativity, you shouldn't mind a quiz.

Chapter 4, Perl 5 Interactive Course

LESSON 3

RECURSION AND MORE MATH

The *absroot* program contains the following subroutine:

```
sub absroot { sqrt(abs($_[0])); }
```

That's a subroutine (*absroot*) that calls a function (*sqrt*) that calls another function (*abs*). As you might expect, your subroutines can call one another. Listing 4-13 is a simple movie-theater simulation that defines three subroutines and a single statement at the end that calls all three.

Listing 4-13 *theater*: A subroutine calling a subroutine calling a subroutine

```
#!/usr/bin/perl -w
$budget = 2000;
@movies = ('Vertigo', 'Rope', 'The Nutty Professor',
           'Heathers', 'Brazil');
@prices = (200, 150, 350, 1000, 800);
@attendances = (300, 200, 300, 800, 500);
debit_budget(get_price(choose_movie));
sub debit_budget {
    $budget -= $_[0];
    print "Now you've got \$$budget. \n";
}
sub get_price { $prices[$_]; }
sub choose_movie {
    $highest_attendance = 0;
    foreach (0..$#movies) {# for each movie
        if ($attendances[$_] > $highest_attendance) {# if it's more
            popular
            $best = $_;# remember the movie
            $highest_attendance = $attendances[$_];# and its attendance
        }
    }
    print "Movie choice: $movies[$best] \n";
    return $best;# return the best
}
```

`debit_budget(get_price(choose_movie))` looks confusing. The evaluation happens from the inside out:

First, `choose_movie()` is called; the returned value is used as input to `get_price()`. Finally, the return value of `get_price` is fed to `debit_budget()`.

The result is a program that

1. Uses `choose_movie()` to choose the movie with the highest attendance
2. Uses `get_price()` to determine the price of that movie (returning the appropriate value from `@prices`)
3. Uses `debit_budget()` to subtract that price from `$budget`

```
% theater
```

```
RESULT Movie choice: Heathers
```

```
Now you've got $1000.
```

Recursion

A subroutine can even call itself. That's called *recursion*. Any quantity defined in terms of itself is recursive; hence the following "joke" definition of recursion:

```
re.cur.sion \ri-`ker-zhen \n See RECURSION
```

Certain problems lend themselves to recursive solutions, such as computing the factorial of a number. The factorial of n , denoted $n!$, is $n * (n - 1) * (n - 2) * \dots * 1$. So $4!$ is $4 * 3 * 2 * 1 = 24$, and $5!$ is $5 * 4 * 3 * 2 * 1$, which is the same as $5 * 4!$. That suggests a recursive implementation:

$$\text{factorial}(n) = n * \text{factorial}(n - 1)$$

Listing 4-14 shows *factor*, which contains the recursive subroutine `factorial`.

Listing 4-14 *factor*: A demonstration of a recursive function

```
#!/usr/bin/perl -w
sub factorial {
    if ($_[0] < 2) { return 1 }
    return $_[0] * factorial($_[0] - 1); # factorial calls itself
}
print factorial($ARGV[0]), "\n";
```

`factorial($n)` returns $n * \text{factorial}(n - 1)$ unless n is less than 2, in which case it returns 1.

When `factorial(6)` is called in line 5, it returns the value $6 * \text{factorial}(5)$; `factorial(5)` returns the value $5 * \text{factorial}(4)$:

$6 * \text{factorial}(5) =$

$6 * 5 * \text{factorial}(4) =$

$6 * 5 * 4 * \text{factorial}(3) =$

$6 * 5 * 4 * 3 * \text{factorial}(2) =$

$6 * 5 * 4 * 3 * 2 * \text{factorial}(1) =$

$6 * 5 * 4 * 3 * 2 * 1 =$

720

```
% factorial 6
```

```
RESULT 720
```

Listing 4-15 presents a program that uses `factorial()` to approximate the sine of a number, using the Maclaurin series expansion:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \dots$$

$3! \ 5! \ 7! \ 9! \ 11!$

Listing 4-15 *calcsine*: Using `factorial()` to compute the sine of a number

```
#!/usr/bin/perl
$x= shift;
$num_terms= shift || 20;# $ARGV[1] if present, 20 otherwise.
print "The sine of $x is approximately $sine.\n";
sub factorial {# calculate the factorial of $_[0]
if ($_[0] < 2) { return 1 }
return $_[0] * factorial($_[0] - 1);
}
sub coeff {# calculate one term
$_[2] * ($_[0] ** $_[1]) / factorial($_[1]);
}
for ($i = 0; $i < $num_terms; $i++) {
$sine += coeff($x, $i*2+1, (-1) ** $i);
}
```

The first argument to *calcsine* is x ; the second number is the number of Maclaurin terms to be computed. The more terms, the better the approximation.

```
% calcsine 0 4
```

```
RESULT The sine of 0 is approximately 0.
```

```
% calcsine 3.14159 4
```

```
RESULT The sine of 3.14159 is approximately -0.0752174014719282.
```

```
% calcsine 3.14159
```

```
RESULT The sine of 3.14159 is approximately 2.6535897932977e-06.
```

```
% calcsine 0.785398163397448 9
```

RESULT The sine of 0.785398163397448 is approximately 0.707106781186547.

```
% calcsine 1.570796326794897 10
```

RESULT The sine of 1.570796326794897 is approximately 1.

sin(), cos(), and atan2()

Of course, you don't really need to do this to compute the sine of a number because Perl provides `sin()`, as well as `cos()` and `atan2()`.

The `sin()`, `cos()`, and `atan2()` Functions

`sin(NUMBER)` returns the sine of *NUMBER*, expressed in radians.

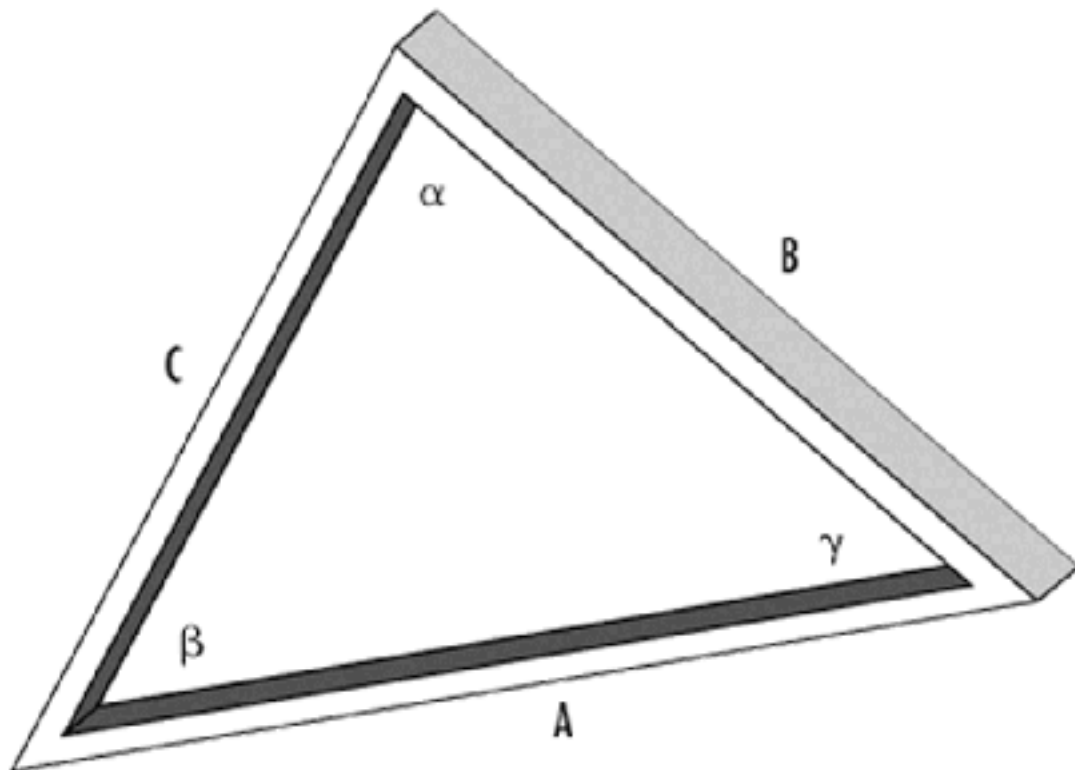
`cos(NUMBER)` returns the cosine of *NUMBER*, expressed in radians.

`atan2(X, Y)` returns the arctangent of *X/Y* in the range $-\frac{1}{4}$ to $\frac{1}{4}$.

You might find these helpful:

- `sub tan { sin($_[0]) / cos($_[0]) }`
- `sub sec { 1/cos($_[0]) }`
- `sub csc { 1/sin($_[0]) }`
- `$pi = 4 * atan2(1,1);`

Listing 4-16 is a short program that uses `cos()` and `$pi` to calculate the length of the third side of a triangle, given the other two sides and the angle between them (expressed in degrees). Check out Figure 4-1.



$$C = \sqrt{A^2 + B^2 - 2AB \cos(\gamma)}$$

Figure 4-1

The law of cosines

Listing 4-16 *lawcos*: Finding the third side of a triangle

```
#!/usr/bin/perl -w
($side1, $side2, $deg) = @ARGV;
$pi = 4 * atan2(1,1);
sub deg_to_rad { $_[0] * ($pi / 180) }
sub law_of_cosines {
    sqrt(($_[0] ** 2) + ($_[1] ** 2) - (2 * $_[0] * $_[1] *
    cos($_[2])));
}
print "The third side is ", law_of_cosines($side1, $side2,
deg_to_rad($deg)), "\n";
```

The `deg_to_rad()` subroutine converts degrees to radians. Most people think about angles in

degrees, but most computers think about them in radians--Perl is no exception.

Suppose you want to know the length of a side opposite a 60-degree angle. In a 30-60-90 triangle, the adjacent sides might be length 1 and 2, so you could use lawcos as follows:

```
% lawcos 1 2 60
```

RESULT The third side is 1.732050808

exp() and log()

If you wanted to, you could approximate factorial(*n*) with (get ready) Stirling's formula:

^— ^—

$$2^{1/4} n^n + 1/2 e^{-n} < n! < 2^{1/4} n^n + 1/2 e^{-n} (1 + 1/4 n)$$

but that doesn't help you much unless you can calculate *e*

n

. Luckily, Perl supplies the exp() function.

The exp() Function

exp(*NUMBER*) returns *e* to the power of *NUMBER*.

stirling, shown in Listing 4-17, uses Stirling's formula to compute factorials.

Listing 4-17 *stirling*: Using exp()

```
#!/usr/bin/perl -w
$pi = 4 * atan2(1,1);
$n = shift;
sub stirling_low {
    sqrt(2 * $pi) * ($_[0] ** ($_[0] + .5)) * exp(-$_[0]);
}
sub stirling_high {
    sqrt(2 * $pi) * ($_[0] ** ($_[0] + .5)) * exp(-$_[0]) * (1 +
    $_[0]/4);
}
print "$n! is between ", stirling_low($n), " and ",
    stirling_high($n), ".\n";
```

In both *stirling_low()* and *stirling_high()*, exp() computes *e* to the power of *-n*.

```
% stirling 4
```

RESULT 4! is between 23.5061751328933 and 47.0123502657866.

```
% stirling 6
```

RESULT 6! is between 710.078184642184 and 1775.19546160546.

% stirling 10

RESULT 10! is between 3598695.61874103 and 12595434.6655936.

% stirling 20

RESULT 20! is between 2.42278684676112E+018 and 1.45367210805667E+019.

The log() Function

log(*NUMBER*) returns the natural logarithm (base *e*) of *NUMBER*.

If you try to take the log() of a negative number, Perl will complain, and rightfully so.

Listing 4-18 is a program that uses log() to calculate logarithms in any base.

Listing 4-18 *logbase*: Calculating logarithms using the natural logarithm log()

```
#!/usr/bin/perl -w
($base, $number) = @ARGV;
($base > 0) && ($number > 0) or die "Both $base and $number
must be > 0.";
sub logbase { log($_[1]) / log($_[0]) }
print "The log (base $base) of $number is ", logbase($base,
$number), ".\n";
% logbase 10 2
```

RESULT The log (base 10) of 2 is .301029995663981

% logbase 2 10

RESULT The log (base 2) of 10 is 3.32192809488736

That's enough math for a while.

Chapter 4, Perl 5 Interactive Course

LESSON 4

SCOPING

Every variable you've seen so far has been global, which means that it's visible throughout the entire program (a white lie; see Chapter 7 for the truth). Any part of your program, including subroutines, can access and modify that variable. So if one subroutine modifies a global variable, all the others will notice.

You'll often want your subroutines to have their own copy of variables. Your subroutines can then modify those variables as they wish without the rest of your program noticing. Such variables are said to be *scoped*, or *localized*, because only part of your program can see that variable.

Scoped variables:

- Lead to more readable programs, because variables stay close to where they're used
- Save memory because variables are deallocated upon exit from the subroutine

There are two ways to scope a variable: with `my()`, which performs lexical scoping, and with `local()`, which performs dynamic scoping.

The `my()` and `local()` Declarations

`my(ARRAY)` performs lexical scoping: The variables named in *ARRAY* are visible only to the current block.

`local(ARRAY)` performs dynamic scoping: The variables named in *ARRAY* are visible to the current block and any subroutines called from within that block.

Use `my()` unless you really need `local()`. `my()` is speedier than `local()`, and safer in the sense that variables don't leak through to inner subroutines.

```
my($x) = 4
```

declares a lexically scoped scalar `$x` and sets it to 4.

```
local($y) = 6
```

declares a dynamically scoped scalar `$y` and sets it to 6.

```
my(@array)
```

declares a lexically scoped array `@array`, which is set to an empty array.

```
my($z, @array) = (3, 4, 5);
```

declares a lexically scoped `$z` set to 3 and a lexically scoped `@array` set to (4, 5).

Scoping is complex, so you'll need lots of examples. Pay careful attention to Listing 4-19.

Listing 4-19 *scoping*: Lexically scoped my() versus dynamically scoped local()

```
#!/usr/bin/perl -w
$x = 1;
$y = 2;
# Uses the global variables; prints 1 and 2
print "Before: x is $x and y is $y.\n";
outer;
# Same here: $x and $y are still 1 and 2.
print "Afterwards: x is $x and y is $y.\n";
sub outer {
my($x); # declare a new $x, lexically scoped
local($y); # declare a new $y, dynamically scoped
$x = 3;
$y = 4;
# uses the local copies: 3 and 4
print "Outer subroutine: x is $x and y is $y.\n";
inner($x, $y);
}
# The $x from outer() isn't visible here, so the global $x is
used.
# This prints 1 and 4
sub inner {
print "Inner subroutine: x is $x and y is $y.\n";
}
```

Before `outer()` is called, the global variables `$x` and `$y` are set to 1 and 2, respectively. Then, in `outer()`, two new variables (also named `$x` and `$y`) are declared: `$x` is declared with `my()`, creating a lexical scope, and `$y` is declared with `local()`, creating a dynamic scope. Because `$y` is dynamically scoped, the `inner()` subroutine sees the new value of `$y` but not `$x`. When `outer()` exits, `$x` and `$y` then revert to their original, global values.

```
% scoping
Before: x is 1 and y is 2.
Outer subroutine: x is 3 and y is 4.
Inner subroutine: x is 1 and y is 4.
Afterwards: x is 1 and y is 2.
```

You'll often assign subroutine arguments to scoped variables:

```
sub marine {
my($first) = @_;
}
```

(Assign the first element of `@_` to `$first`.)

```
sub commander {
my($first, $second) = @_;
```

```
}
```

(Assign the first element to \$first and the second to \$second.)

```
sub scribe {  
my(@all) = @_;  
}
```

(Assign all elements to @all.)

```
sub lime {  
my($first, @rest) = @_;  
}
```

(Assign the first element to \$first and the rest to @rest.)

The *polynomy* program, shown in Listing 4-20, calculates the value of a polynomial, demonstrating my().

Listing 4-20 *polynomy*: Passing a scalar and an array into a subroutine

```
#!/usr/bin/perl -w  
sub polycalc {  
my($x, @coeffs) = @_; # Just like ($first, @rest) = @_  
my($i, $total) = (0, 0); # Set both $i and $total to 0  
foreach $coeff (@coeffs) {  
$total += $coeff * ($x ** $i);  
$i++;  
}  
return $total;  
}  
print "When x is -5, 8 + 14x is ", polycalc(-5, 8, 14),  
".\n";  
print "When x is 6, 2 + 9x + 3x^2 is ", polycalc(6, 2, 9, 3),  
".\n";
```

The line my(\$x, @coeffs) = @_ assigns the first value of @_ to \$x, and the rest of the values of @_. And my(\$i, \$total) initializes both \$i and \$total to zero.

```
% polynomy  
When x is -5, 8 + 14x is -62.  
When x is 6, 2 + 9x + 3x^2 is 164.
```

Both my() and local() provide an array context, which is why



```
my ($line) = <FILE>
```

and



```
local ($line) = <FILE>
```


probably won't do quite what you want. They'll read all the lines in FILE, assigning the first to \$line and throwing away the rest. To read only the first line, force a scalar context with scalar():

```
my($line) = scalar(<FILE>)
```

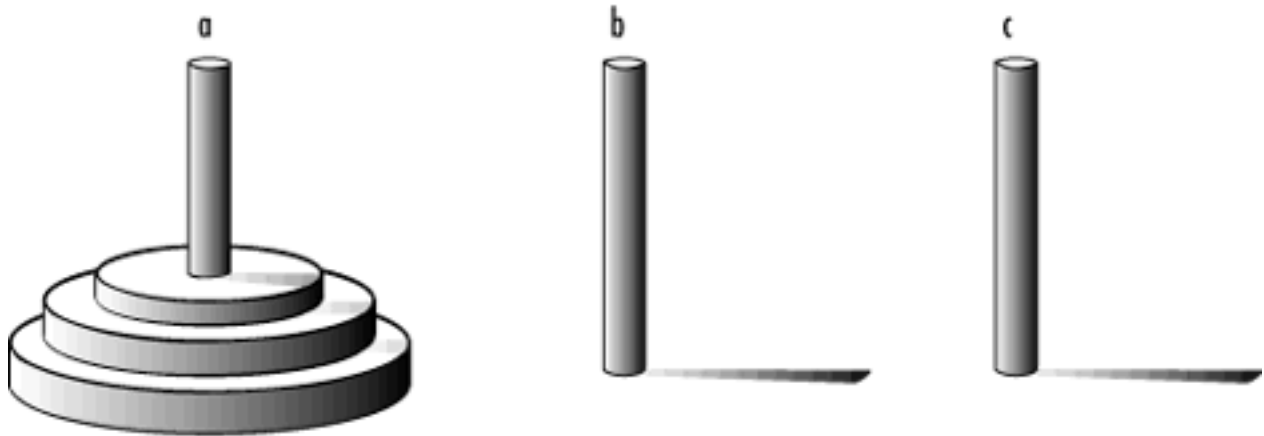


Figure 4-2

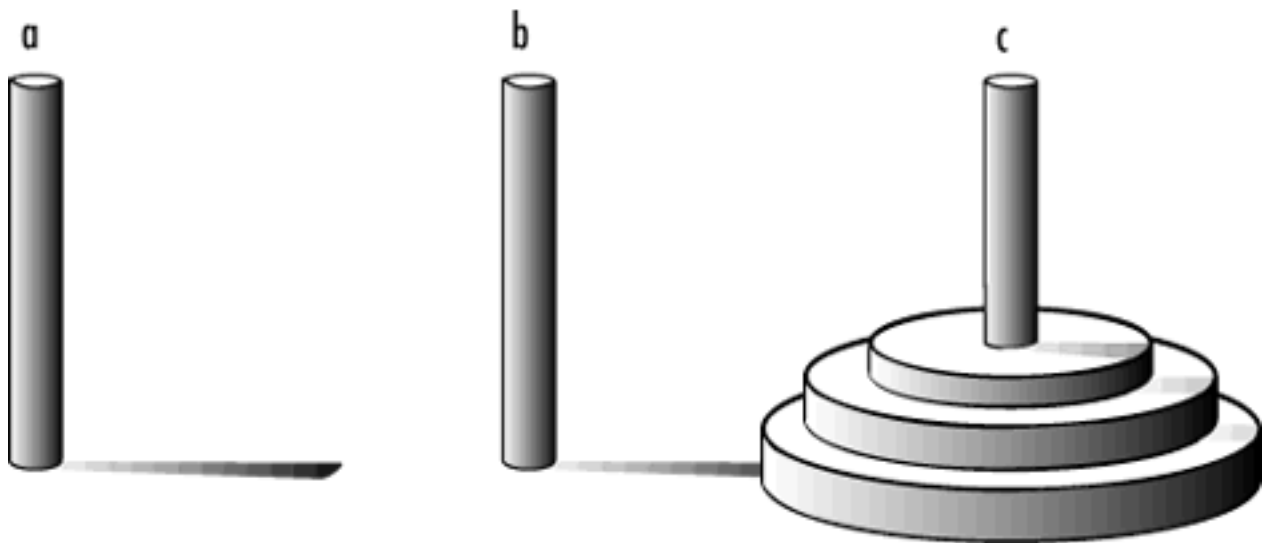


Figure 4-3

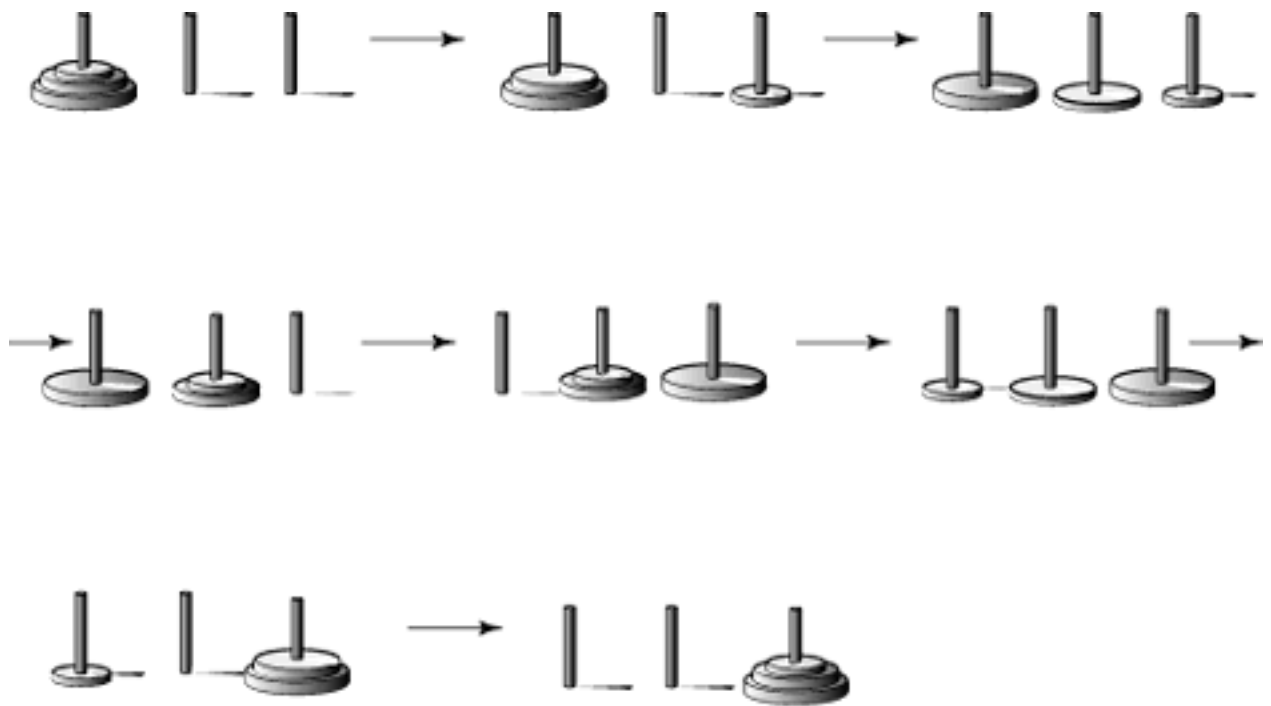


Figure 4-4

Chapter 4, Perl 5 Interactive Course

LESSON 5

BLOCKS

This chapter is mostly about subroutines. But it's also about organizing programs and, in particular, how to group statements that belong together. A *block* is a collection of statements enclosed in curly braces. You've seen a few examples already:

```
if (@ARGV) { shift; print "here's a block"; }
while ($i < 5) { print "here's another"; $i++ }
sub routine { print "and here's a third." }
```

You've also seen statements for moving around within blocks: *last*, *next*, and *redo*, which were used to control loops. You can think of a block as being a loop that's executed just once. In this lesson, we'll learn a few additional methods for navigating blocks.

Blocks don't need to be part of an *if*, *while*, or *sub* statement. They can exist all by themselves, as shown in Listing 4-21.

Listing 4-21 *blokhed1*: A simple block

```
#!/usr/bin/perl -w
{
print "This is a block, all by itself. \n";
print "Looks weird, doesn't it? \n";
}
```

Blocks are, in a sense, miniprograms. They can have their own variables, as shown in Listing 4-22.

Listing 4-22 *blokhed2*: A variable declared inside a block

```
#!/usr/bin/perl -w
{
my ($var) = 7;
print "My lucky number is $var. \n";
}
print "Now I have no lucky number. \n";
```

Labels

What if you have two nested *for* loops and you want to exit both loops if some condition occurs?

```
for (...) {  
  for (...) {  
    exit_both_loops if condition;  
  }  
}
```

How can you exit both loops? A simple `last` won't work because that just exits the innermost loop. You need a label.

Labels

A label names a specific place in a program so that you can refer to that point.

By convention, labels are capitalized, but they don't have to be. Here's a label named TAG:

```
TAG: print "Any statement can follow a label.";
```

Labels are handy when you want to navigate relative to a loop that isn't the innermost. Listing 4-23 uses a label to jump from the inner loop to the outer.

Listing 4-23 *primes*: Using labels to control program flow

```
#!/usr/bin/perl -w  
@primes = ();  
NUMBER: foreach $number (2..100) {  
  foreach $prime (grep($_ <= sqrt $number, @primes)) {  
    next NUMBER unless $number % $prime; # jumps to NUMBER  
  }  
  push(@primes, $number);  
}  
print "@primes";
```

The *primes* program uses the Sieve of Eratosthenes to search (efficiently!) for prime numbers. The outer loop iterates through the numbers, from 2 to 100, to be tested for primeness. The inner loop tests all possible divisors of that number. If it finds a divisor (`$number % $prime`), it discards the number by jumping back to the outer loop: `next NUMBER`

```
% primes
```

```
RESULT 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83  
89 97
```

goto

Labels don't have to name loops. You can place a label just about anywhere and then have your program jump to that point, with the vaunted *goto* statement.

The goto Statement

goto *label* jumps to the statement labeled with *label* .

You can also *goto* a label chosen at runtime or to a subroutine (which invokes it). See the *perlfunc* documentation for a more detailed explanation.

Here's a simple example:

```
print "hello";
goto BYE;
print "smalltalk";
BYE: print "goodbye";
```

This snippet prints *hello* and then skips directly to the BYE label, avoiding small talk and printing *goodbye*.

It's very chic to hate the *goto* statement. Critics claim (and rightfully so) that it hampers program readability. Listing 4-24 provides proof.

Listing 4-24 *scare*: The *goto* statement jumps to a label

```
#!/usr/bin/perl -w
PHANTOM: print "Booo\n";
WRAITH: print "Hisss\n";
GHOUL: print "Shriek\n";
goto WRAITH;
goto PHANTOM;
```

Try to determine what *scare* does by eyeballing it.

```
% scare
```

```
RESULT Booo
```

```
Hisss
Shriek
Hisss
Shriek
Hisss
Shriek
Hisss
Shriek
Hisss
Shriek
[CTRL]-[C]
```

There's an infinite loop: *goto* PHANTOM is never reached, because *goto* WRAITH bumps Perl up to the WRAITH label over and over again.

That's the problem with *goto*. *Scare* doesn't look like it contains an infinite loop. In fact, it doesn't seem to have any loops at all.

do()...until()

In Chapter 3, you learned that *do* can execute external Perl files. You can also use *do* to mimic a *while* loop, with a twist: The condition is evaluated at the end of the loop instead of the beginning.

The do and until Statements

do *BLOCK* executes *BLOCK* once. It's often used with *until* or *while* to execute a loop before testing the condition.

until *CONDITION BLOCK* behaves like *while*, except that the loop executes until the *CONDITION* is TRUE, that is, while *CONDITION* is FALSE. You can use *until* wherever you can use *while*.

Caveat: You can't use "loop control" (*last*, *next*, or *redo*) inside *do...until* blocks. Listing 4-25 is an example of a *do...until* statement.

Listing 4-25 *do_until*: Using do and until to execute a block before testing the condition

```
#!/usr/bin/perl -w
do {
  chomp($input = <>);
  print "Pondering $input...\n";
} until $input =~ /^the end$/i;
```

do_until immediately enters the *do* loop. It seems to behave just like a *while* loop:

```
% do_until
```

```
RESULT why
```

```
Pondering why...
the end
Pondering the end...
```

but the *CONDITION* isn't checked until the end of the first iteration:

```
% do_until
```

```
RESULT the end
```

```
Pondering the end...
```

continue Blocks

Sometimes you'll want to perform some task after every loop iteration, but before the condition is evaluated again. *for* loops let you do that with their third clause: the *\$i++* in *for (\$i=0; \$i<10; \$i++)*. For other loops, you can use a *continue* block.

continue Blocks

while *CONDITION BLOCK1* *continue BLOCK2* executes *BLOCK2* after every *BLOCK1* iteration (and before *CONDITION* is evaluated).

BLOCK2 is evaluated regardless of how the *BLOCK1* iteration terminated: either with next or "naturally," by reaching the last statement in the block.

loopback, in Listing 4-26, uses a *continue* block to simulate

```
for ($j = 0; $j < 4; $j++) { print "$j "}
```

with a *while* loop.

Listing 4-26 *loopback*: Using *continue*

```
#!/usr/bin/perl -w
$j = 0;
while ($j < 4) {
    print "$j\n";
    } continue { $j++; }
```

After every iteration, `$j++` is executed:

```
% loopback
```

```
RESULT 0
```

```
1
2
3
```

reset()

One use for *continue* blocks is to "back up," obliterating some variable created during a loop, with the `reset()` function.

The `reset()` Function

reset *CHARACTERS* resets all variables whose names begin with a character in the range of *CHARACTERS*. Their values are wiped out, making it seem as though those variables never held a value.

`reset()` is dangerous. Be careful not to wipe out variables inadvertently--`reset 'A'` (and `reset 'A-Z'`) obliterates `@ARGV`. Use with caution!

You can specify a range of characters using a hyphen. *train*, in Listing 4-27, uses `reset()` to give `@derailed` amnesia.

Listing 4-27 *train*: `reset()` wipes out the value of `@derailed`

```
#!/usr/bin/perl -w
@train = ('engine', 'dining', 'lounge', 'sleeper',
    'caboose');
while (@train) {
    my $car = shift @train;
    push (@derailed, $car);
```

```
print "Derailed: @derailed\n";
} continue {
reset `d` if $car eq `sleeper`; # reset @derailed
}
```

After every iteration of the *while* loop, the *continue* block is executed, wiping out the value of @derailed (because it begins with a *d*) and printing the current value of \$car:

```
% train
```

```
RESULT Derailed: engine
```

```
Derailed: engine dining
```

```
Derailed: engine dining lounge
```

```
Derailed: engine dining lounge sleeper
```

```
Derailed: caboose
```

??

You learned about `m//` in Chapter 2; another match operation is identical to `m//` in all respects but matches only once between calls to `reset()` when no argument is provided, that is,

```
reset;
```

There's no name for this operation other than `??`, because those are the characters used to delimit the match:

```
?perl?
```

matches the regex *perl* only once.

findall, shown in Listing 4-28, prints all lines that contain the word *perl*.

Listing 4-28 *findall*

```
#!/usr/bin/perl -w
while (<>) {
print if /perl/;
}
```

In particular, when run on itself, *findall* finds both occurrences:

```
% findall findall
```

```
RESULT #!/usr/bin/perl
```

```
print if /perl/;
```

But if `??` is used, only the first match counts, as shown in Listing 4-29.

Listing 4-29 *findonce*

```
#!/usr/bin/perl -w
while (<>) {
print if ?perl?;
}
```



```
% findonce findonce
```

```
RESULT #!/usr/bin/perl
```

But if you add a *continue* block with a `reset()`, you can get it to find both occurrences, as shown in Listing 4-30.

Listing 4-30 *findall2*

```
#!/usr/bin/perl -w
while (<>) {
    print if ?perl?;
} continue {
    reset;
}
% findall2 findall2
```

```
RESULT #!/usr/bin/perl
```

```
    print if ?perl?;
```

?? is a valid Perl 5 operator, but might not live to see Perl 6.

Chapter 4, Perl 5 Interactive Course

LESSON 6

ORGANIZING ARRAYS AND AUTOLOADING FUNCTIONS

This lesson describes some methods for manipulating arrays: how to sort them, how to reverse them, how to convert them to strings, and how to construct them out of strings. There's an unrelated but interesting tidbit at the end: a catchall subroutine that your programs can call when an undefined subroutine is called.

Sorting

You'll often want to rearrange the elements of an array. Perhaps you want your array to be ordered from smallest to largest, or by the length of strings, or by some other criterion. Perl provides a `sort()` function for this purpose.

You've seen scalars and arrays used as arguments to subroutines. `sort()` expects something new as an argument--a subroutine.

The `sort()` Function

`sort SUBROUTINE ARRAY` sorts *ARRAY* using the comparison function *SUBROUTINE* and returns the sorted array. Inside the *SUBROUTINE*, *\$a* and *\$b* are automatically set to the two elements to be compared.

If you use `sort()` without a subroutine, it sorts in string order, as shown in Listing 4-31.

Listing 4-31 *easysort*: Without a subroutine, `sort()` will sort in string order

```
#!/usr/bin/perl -w
@colors = ("red", "blue", "white", "black", "green");
@sorted = sort @colors;
foreach (@sorted) { print "$_ "; }
```

`sort()` does the right thing with `@colors`:

```
% easysort
```

```
RESULT black blue green red white
```

String order isn't the same as numeric order.

Because 1 comes before 2 in ASCII, *badsort* places 10000 before 209. Listing 4-32 won't sort properly.

Listing 4-32 *badsort*: String



```
#!/usr/bin/perl
@prices = (99, 10000, 209, 21, 901);
@sorted = sort @prices;
foreach (@sorted) { print "$_ "; }
% badsort
```

RESULT 10000 209 21 901 99

Whoops!

If a *SUBROUTINE* is provided to `sort()`, it's called in a peculiar way: Instead of using `@_` to pass arguments, the elements to be compared magically appear in `$a` and `$b`. The subroutine returns a positive integer if `$a` should appear before `$b` in the sorted array, a negative integer if `$b` should appear before `$a`, and 0 if the order doesn't matter.

There are four common pitfalls when using `sort()`.

1. Using `$_[0]` and `$_[1]` in the subroutine instead of `$a` and `$b`
2. Forgetting that string order is not the same as numeric order
3. Forgetting that string order is not the same as dictionary order (because capitals precede lowercase)
4. Thinking that `sort()` modifies the input array

The `<=>` and `cmp` Operators

Both of these operators return 1 if the scalar on the left is greater than the scalar on the right, -1 if the reverse is true, and 0 if the two quantities are equal. `<=>` operates on numbers; `cmp` operates on strings. So

```
5 <=> 2
```

is 1,

```
2 <=> 5
```

is -1, and

```
2.3 <=> 2.3
```

is 0.

```
"zither" cmp "aardvark"
```

is 1,

```
"aardwolf" cmp "zygote"
```

is -1, but

```
"aardwolf" cmp "Zygote"
```

is 1 because lowercase letters follow uppercase letters in ASCII.

The numerically subroutine uses `<=>` to turn *badsort* into *goodsort* (Listing 4-33).

Listing 4-33 *goodsort*: Using a subroutine as an argument to `sort()`

```
#!/usr/bin/perl -w
@prices = (99, 10000, 209, 21, 901);
sub numerically { $a <=> $b }
@sorted = sort numerically @prices;
foreach (@sorted) { print $_, " "; }
% goodsort
```

RESULT 21 99 209 90 10000

That's better. Listing 4-34 sorts strings in dictionary (rather than string) order.

Listing 4-34 *abcsort*: The `alphabetically()` subroutine is used as an argument to `sort()`

```
#!/usr/bin/perl
@movies = ('The Addams Family', 'clash of the titans',
           'Kramer vs. Kramer');
sub alphabetically { lc($a) cmp lc($b); }
@sorted = sort alphabetically @movies;
foreach (@sorted) { print $_, "\n"; }
```

Without the `alphabetically()` subroutine, **clash of the titans** would become the last element of `@sorted`, because ASCII `c` comes after ASCII `T` and ASCII `K`.

```
% abcsort
```

RESULT clash of the titans

```
Kramer vs. Kramer
The Addams Family
```

`sort()` lets you supply an "inline" subroutine:

```
@numeric_sort = sort { $a <=> $b } @array;
@alphabetic_sort = sort { lc($a) cmp lc($b) } @array;
```

and you can sort an array into itself:

```
@array = sort { $a cmp $b } @array;
```

and you can use any sorting rule you like. This places elements divisible by 7 before elements that aren't:

```
sub by_seven {
return 1 if ($a % 7);
return -1 if ($b % 7);
return 0;
}
```

```
@sevens = sort by_seven (5, 18, 14, 10, 98, 21, 3, 1);
```

Three caveats about `sort()`:

- `sort()` doesn't modify its input. It returns a fresh array instead.
- Don't modify `$a` or `$b` inside the subroutine. You'll be sorry.
- Make sure that `sort()` is evaluated in a list context. In a scalar context, you'll end up with the number of elements in the sorted array.

Joining Arrays and Splitting Strings

Converting between data types is easy in Perl. You can glue array elements together into a string with `join()` and break up a string into an array with `split()`.

The `join()` and `split()` Functions

`join(SEPARATOR, ARRAY)` returns a string constructed by joining the elements of *ARRAY* with the *SEPARATOR* between each element.

`split(PATTERN, STRING, LIMIT)` returns an array (with no more than *LIMIT* elements) constructed by using the *PATTERN* as a delimiter to distinguish elements in *STRING*.

PATTERN is usually a regular expression enclosed in slashes.

`split()` has a number of defaults:

- If *STRING* is omitted, `split()` splits `$_`, as you'd expect.
- If *PATTERN* is also omitted, `split()` splits on whitespace.
- If *LIMIT* is omitted, `split()` creates as many elements as necessary.

joiner (Listing 4-35) combines array elements, using the `join()` function.

Listing 4-35 *joiner*: The `join()` function glues array elements together, returning a string

```
#!/usr/bin/perl -w
@movies = ('Amadeus', 'E.T.', 'Frankenstein', 'Eraserhead');
print join('+', @movies);
% joiner
```

RESULT Amadeus+E.T.+Frankenstein+Eraserhead

You can use `split()` to extract the words from a string. Listing 4-36 shows the advent program from Chapter 2, modified to use `split()`.

Listing 4-36 *advent2*: By default, `split()` splits on whitespace

```
#!/usr/bin/perl -w
while (<>) {
    chomp;
    @words = split;
    if (@words > 2) { print "Where is $words[$#words]?\n"; }
    else { print "How do I $words[0]?\n"; }
}
% advent2
```

RESULT go to school

```
Where is school?
open door
How do I open?
pet unicorn with sword
Where is sword?
```

Suppose you have a file of sentences, formatted to a particular column width, called *opinion*, that looks like this:

```
Of that there can
be no doubt. However, the situation
demands careful handling. In the
opinion of the Court, the defendant
should be spanked right good.
```

Listing 4-37 shows a program that splits *opinion* into sentences, assuming every sentence ends in a period and every period ends a sentence.

Listing 4-37 *sentence*: Using `split()` and `join()`

```
#!/usr/bin/perl -w
open (FILE, $ARGV[0]) or die "Can't read $ARGV[0]: $! ";
@lines = <FILE>; # read all the lines from FILE
# Glue all lines together into $big_string.
$big_string = join('', @lines);
$big_string =~ s/\n//g; # Remove all newlines
$big_string =~ s/\.$//; # Remove the final period (to be
added later)
# Create @sentences by dividing $big_string on
period-followed-by-whitespace.
@sentences = split(/\.\s/, $big_string);
foreach (@sentences) { print $_, ".\n"; }
```

The `join()` gloms all the *opinion* lines together, so `$big_string` looks like this:

```
Of that there can\n be no doubt. However, the situation\n
demands...
```

After removing the new lines with `s///`, the string is split into elements ending in a period (`\.`) followed by one or more spaces (`\s+`). Each element of `@sentences` now contains the text of a sentence, without the final period or new line, so *sentence* adds those in when it prints:

```
% sentence opinion
```

```
RESULT Of that there can be no doubt.
```

```
However, the situation demands careful handling.
```

```
In the opinion of the Court, the defendant should be spanked  
right good.
```

Array Interpolation

One simple way to join without `join()`ing is to let double quotes interpolate the array for you:

```
@array = (1, 2, 3, "four");  
$a = "@a";
```

(*\$a* is now *1 2 3 four*).

The elements of `@array` were interpolated into the string with spaces separating the elements. Why a space? Because that's the default value of the special variable `$"`.

The `$"` Special Variable

`$"` (or `$LIST_SEPARATOR`) is inserted between each pair of array elements when the array is inside a double-quoted string.

You can set your own value of `$"`:

```
$" = "-"  
@array = (1, 2, 3, "four");  
$a = "@array";
```

(now *\$a* is *1-2-3-four*).

There's nothing stopping you from using more than one character:

```
$" = ", "  
@array = (1, 2, 3, "four");  
$a = "@array";
```

(now *\$a* is *1, 2, 3, four*)

or an entire word:

```
$" = " and-a "  
@array = (1, 2, 3, "four");  
$a = "@array";
```

(now *\$a* is *1 and-a 2 and-a 3 and-a four*).

Reversing Variables

The `reverse()` function reverses an array or a scalar, depending on context. (`reverse()` flips hashes, too. Hashes are covered in Chapter 5.)

The `reverse()` Function

`reverse(ARRAY)` returns the elements of *ARRAY* with the order of the elements flipped. In a scalar context, **`reverse()`** character-reverses *ARRAY*.

`reverse()`'s behavior can be surprising unless you pay close attention to the context in which it's called. Some examples:

```
print reverse(1,2,3,4)
prints 4321,

@array = ( 'Tilden', 'Beat', 'Hayes' );
print reverse @array;
prints HayesBeatTilden, and

print reverse 1089
prints 1089 because print() creates a list context. But

$number = reverse 1089
print $number;
prints 9801 because "$number =" creates a scalar context.

$words = reverse("live", "dog");
print $words;
prints godevil, and

$string = scalar(reverse "stalag 17");
print $string;
prints 71 galats.
```

Listing 4-38 is an example of `reverse()`.

Listing 4-38 *reverse1*: Using `reverse()`

```
#!/usr/bin/perl -wl
@movies = ( 'Broadcast News', 'The Paper', 'Citizen Kane',
            'Newsfront' );
print join(' ', reverse @movies);
print join(' ', reverse sort @movies);
```

The first `print()` reverses `@movies`, which is then `join()`d. The second `print()` sorts `@movies` first:

```
% reverse1
```

```
RESULT Newsfront, Citizen Kane, The Paper, Broadcast News
         The Paper, Newsfront, Citizen Kane, Broadcast News
```


Because 0..9 acts like an array, you can reverse that as well. Instead of a *for* loop with a decrement, you can `reverse()` a range, as shown in Listing 4-39.

Listing 4-39 *reverse2*: Reversing a range of numbers

```
#!/usr/bin/perl -wl
foreach (reverse 3..9) { print "Iteration $_" }
% reverse2
```

RESULT Iteration 9

```
Iteration 8
Iteration 7
Iteration 6
Iteration 5
Iteration 4
Iteration 3
```

Listing 4-40 reverses all the lines of a file (or STDIN).

Listing 4-40 *reverse3*: Reversing all lines of a file

```
#!/usr/bin/perl -w
print reverse <>;
```

Here's what *reverse3* would do to the *opinion* file above (assuming there's no whitespace after the last line!):

```
% reverse3 opinion
```

RESULT should be spanked right good.opinion of the Court, the defendant
demands careful handling. In the
be no doubt. However, the situation
Of that there can

If that first line seems a bit funny to you, consider where the new lines are in *opinion*.

Autoloading

You've seen default variables and default filehandles. Perl offers a feature you won't find in many other programming languages: a default subroutine.

If you define a function called `AUTOLOAD()` (yes, capitalization matters), then any calls to undefined subroutines will call `AUTOLOAD()` instead. The name of the missing subroutine is accessible within this subroutine as `$AUTOLOAD`. The arguments are passed via `@_`, as usual.

`AUTOLOAD()` is actually meant to be used with the object-oriented features discussed in Chapter 7. But it can be helpful anyway, as shown in Listing 4-41.

Listing 4-41 *autoload*: Using AUTOLOAD() to catch undefined subroutine calls

```
#!/usr/bin/perl -w
sub AUTOLOAD {
my(@args) = @_;
print "I've never heard of a subroutine named $AUTOLOAD.\n";
print "I'm ignoring your arguments: ", join(' ', @args),
".\n";
}
tetrade(3, 2);
% autoload
```

RESULT I've never heard of a subroutine named main::tetrade.
I'm ignoring your arguments: 3 2.

In later chapters, you'll learn how autoloading can be used to load certain subroutines on demand, that is, only when they're actually needed by the program.

Chapter 4, Perl 5 Interactive Course

LESSON 7

ARRAYS AND SUBROUTINES

In this lesson, we'll cover two tricky concepts:

- How to pass multiple arrays (and filehandles) into a subroutine
- How a subroutine can determine its context

Passing Multiple Arrays into a Subroutine

When subroutine arguments are merged into the array `@_`, the boundaries between arrays are lost: `((1, 2, 3), (4, 5, 6))` becomes `(1, 2, 3, 4, 5, 6)`. There are two ways to keep multiple array arguments intact: passing references and passing globs.

Passing References

References are discussed at length in Chapter 7, Lesson 2. But Listing 4-42 shows what references as arguments look like.

Listing 4-42 *argref*: Passing references

```
#!/usr/bin/perl -wl
@array1 = (1, 2, 3);
@array2 = (4, 5, 6);
multarray(\@array1, \@array2); # note the backslashes!
sub multarray {
    my($array1ref, $array2ref) = @_;
    my(@array1) = @$array1ref; # the arrays themselves...
    my(@array2) = @$array2ref; # ...note the @$ notation!
    print "Array 1 is @array1";
    print "Array 2 is @array2";
}
```

`routine(\@array1, \@array2)` doesn't pass two arrays--it passes two references to arrays. The routine() subroutine then converts them back into regular arrays with the strange-looking `@$` notation:

```
% argref
```

```
RESULT Array 1 is 1 2 3
```

```
Array 2 is 4 5 6
```

Passing Globs

Globs are tricky. Before they're explained, look at Listing 4-43.

Listing 4-43 *argglob*: Passing globs

```
#!/usr/bin/perl -wl
@array1 = (1, 2, 3);
@array2 = (4, 5, 6);
multarray(*array1, *array2); # note the stars!
sub multarray {
    local(*array1, *array2) = @_; # stars are here too
    print "Array 1 is @array1"; # ...but not here!
    print "Array 2 is @array2";
}
```

**array1* and **array2* are *globs*. Globs are a strange new data type: **name* stands for any or all of *\$name*, *@name*, *&name*, *name*, and *%name* (that's a hash; more on those in Chapter 5). When you pass a glob into a subroutine, you're saying, in effect, "I'm just passing in the name of the variable; I won't identify the type (scalar, filehandle, array, or something else) until later in the subroutine." This is often called *type globbing* and it has nothing to do with the *glob()* function from Chapter 3, Lesson 8.

In *argglob*, the *multarray()* subroutine accepts the two globs (with *local()*; you can't use *my()* with globs) and then simply pretends that they're arrays.

```
% argglob
```

```
RESULT Array 1 is 1 2 3
```

```
Array 2 is 4 5 6
```

Modifying Subroutine Arguments

There are two styles of argument passing in programming languages: *pass by value* and *pass by reference*. The difference is subtle, but worth taking the time to understand.

- **Pass by value:** The arguments to a subroutine are local copies, not the variables themselves. Changing a variable inside the subroutine doesn't change it outside.
- **Pass by reference:** The subroutine has access to the actual variables, so changing a variable inside the subroutine also changes it outside.

Normally, when you pass a variable into a Perl subroutine, you're using pass by reference. That's not the same as passing an actual reference (with a backslash); it just means that you're passing a container holding a value instead of the value itself.

But because array assignments copy individual elements (when you say *@a = @b*, you're creating a new array *@a* that happens to have the same elements as *@b*), argument passing can often seem like pass by value, as demonstrated by the following ineffectual subroutine:



```
@a = (1,2,3);
subappend {
  my (@array) = @_;
  push (@array, 4);
}
append(@a);
```

Even after calling `append()`, `@a` remains (1, 2, 3), because `@array` is freed as soon as the subroutine ends.

If you pass references or globs, you can modify `@a` from inside the subroutine.

This passes `@a` as a reference:

```
@a = (1, 2, 3);
append(\@a);
subappend {
  my($arrayref) = @_;
  push (@$arrayref, 4);
}
```

Now `@a` is (1, 2, 3, 4).

This passes `@a` as a glob:

```
@a = (1,2,3);
append(*a);
subappend {
  local (*array) = @_;
  push (@array, 4);
}
```

Now `@a` is (1, 2, 3, 4).

Passing a Filehandle into a Subroutine

There are three ways to pass a filehandle into a subroutine. If you can store the name of the filehandle in a string, you can pass that string:

```
$my_handle = "STDOUT";
my_print($my_handle);
# Note NO comma between $filehandle and $string---
# just like print STDOUT $string.
sub my_print {
  my ($filehandle) = @_;
  print $filehandle "Hello! \n";
}
```

That's a little slow. You can use pass by name instead, and pass a glob:

```
my_print(*STDOUT);
my_print(*LOG);
```

The safest method of all is to pass a reference to the glob:

```
my_print(\*STDOUT);
my_print(\*LOG);
```

wantarray()

Many of Perl's functions display strikingly different behaviors, depending on the context in which they are called. If you want your subroutine to do the same, it'll have to determine its context with `wantarray()`.

The wantarray() Function

wantarray() returns true if the enclosing subroutine was called in a list context.

mirror, shown in Listing 4-44, contains the `flippoint()` subroutine, which accepts an n-dimensional point as input.

- If `flippoint()` is evaluated in a list context, it returns another point--the original point reflected across the origin.
- If `flippoint()` is evaluated in a scalar context, it returns the distance of that point from the origin.

Listing 4-44 *mirror*: Using `wantarray()`

```
#!/usr/bin/perl -w
@point1 = (1, 1); # A two-dimensional point
@flipped = flippoint(@point1); # List context
print "Array context: ", join(" ", @flipped), "\n";
$distance = flippoint(@point1); # Scalar context
print "Scalar context: ", $distance, "\n\n";
@point2 = (3, -4, 5); # A three-dimensional point
print "List context: ", join(" ", flippoint(@point2)), "\n";
print "Scalar context: ", scalar(flippoint(@point2)), "\n";
# Accept a point ($x, $y). (Actually, an n-dimensional point
# ($x, $y, $z, ...))
# Return (-$x, -$y) if called in an array context,
# and sqrt($x*$x + $y*$y) if called in a scalar context.
sub flippoint {
    my (@points) = @_;
    my $distance = 0;
    return map($_ * -1, @points) if wantarray; # Called in list
    context
    foreach (@points) { # Called in scalar context
        $distance += $_ ** 2;
    }
    sqrt($distance);
}
```

```
}
```

mirror feeds two points, @point1 and @point2, to flippoint(). Each point is placed in a list context first and then in a scalar context.

```
% mirror
```

```
RESULT Array context: -1 -1
```

```
Scalar context: 1.4142135623731
```

```
Array context: -3 4 -5
```

```
Scalar context: 7.07106781186548
```

Chapter 4, Perl 5 Interactive Course

LESSON 8

BITS AND BASES

Any number can be represented in any base (Table 4-1). We're used to thinking about numbers in base 10 (because we have ten fingers) so that the individual digits range from 0 to 9, but you can represent any number equally well in other bases. Computers prefer base 2--that is, binary.

Table 4-1 Common bases for programming

Base Name		Digit Values	How 15 Is Represented in That Base
2	Binary	0,1	1111, because 15 is $(1*2^3) + (1*2^2) + (1*2^1) + (1*2^0)$
8	Octal	0,1,2,3,4,5,6,7	17, because 15 is $(1*8^1) + (7*8^0)$
10	Decimal	0,1,2,3,4,5,6,7,8,9	15, because 15 is $(1*10^1) + (5*10^0)$
16	Hexadecimal	0,1,2,3,4,5,6,7,8,9, A,B,C,D,E,F	F, because 15 is $(15*16^0)$

chr(), ord(), oct(), and hex()

You can convert numbers from bases 8 and 16 to base 10 with oct() and hex(), and to or from ASCII values (which have nothing to do with bases) with chr() and ord(). (The *base2base* program in Chapter 10 lets you convert between arbitrary bases.)

The chr(), ord(), oct(), and hex() Functions

ord(*CHARACTER*) returns the ASCII value of *CHARACTER*.

chr(*NUMBER*) returns a character given its ASCII *NUMBER*.

oct(*NUMBER*) returns the decimal value of the octal *NUMBER*.

hex(*NUMBER*) returns the decimal value of the hexadecimal *NUMBER*.

If ord() is given a string instead of a character, it operates upon the first character.

```
ord('A')
```


is 65 because A is 65 in ASCII.

```
chr(65)
```

is A.

```
oct(17)
```

is 15.

```
hex('f')
```

is also 15.

If only Perl had a dec() function for decimal numbers, you could write

```
oct 31 == dec 25
```

justifying the old joke about programmers thinking Halloween is the same as Christmas. Yuk yuk.

Listing 4-45 is a demonstration of all four functions.

Listing 4-45 *charplay*: Using oct(), hex(), chr(), and ord()

```
#!/usr/bin/perl -w
$_ = 17;
print "Octal $_ is ", oct, " in base ten.\n";
print "Hexadecimal $_ is ", hex, " in base ten.\n";
print "Gimme an A! ", chr(65), "!\n"; # ASCII 65 is A
print "Route ", ord('B'), "!\n"; # ASCII 66 is B
```

Here's what *charplay* prints:

```
% charplay
Octal 17 is 15 in base ten.
Hexadecimal 17 is 23 in base ten.
Gimme an A! A!
Route 66.
```

Bitwise Operators

Sometimes you'll want to think about a scalar as a sequence of bits instead of a sequence of characters (like a string) or a sequence of decimal digits (like conventional base 10 numbers). Perl provides several *bitwise operators* that you can use to manipulate numbers bit by bit.

The Bitwise Operators

BITS1* | *BITS2 yields the bitwise OR of ***BITS1*** and ***BITS2***.

BITS1* & *BITS2 yields the bitwise AND of ***BITS1*** and ***BITS2***.

BITS1* ^ *BITS2 yields the bitwise XOR of ***BITS1*** and ***BITS2***.

~ *BITS* yields the bitwise negation of ***BITS***.

BITS* << *NUMBITS yields ***BITS*** shifted to the left by ***NUMBITS*** bits.

BITS* >> *NUMBITS yields ***BITS*** shifted to the right by ***NUMBITS*** bits.

These probably won't make too much sense unless you've played with these operations before. Here's an example of each:

```
10 | 3
```

is 11 because 10 is 1010 in binary, 3 is 0011 in binary, and 1010 ORed with 0011 is 1011, which is 11 in decimal. To see why, look at the binary numbers depicted so that the bits to be ORed are aligned vertically:

```
1 0 1 0
or 0 0 1 1
-----
1 0 1 1
10 & 3
```

is 2 because 1010 ANDed with 0011 is 0010.

```
10 ^ 3
```

is 9 because 1010 XORed with 0011 is 1001.

```
~10
```

is 4294967285 (on a 32-bit machine) because binary 10 is actually 00000000000000000000000000001010, so its negation is

```
111111111111111111111111111110101.
```

```
10 << 2
```

is 40 because 1010 shifted 2 bits to the left is 101000.

```
10 >> 1
```

is 5 because 1010 shifted 1 bit to the right is 101.

Every bitwise operator has a corresponding assignment operator:

```
|= &= ^= ~= <<= >>=
```

In Chapter 3, you learned how `chmod()` can be used to set UNIX permissions for a file (using an octal number as the first argument). If you forget how to construct an octal number (by prefacing the number with 0), you can assemble proper arguments to `chmod()` using decimal numbers, `|`, and `<<`.

```
chmod(07, $file);# -----rwx
chmod(7, $file);# -----rwx, because decimal 7 equals octal 7
chmod(017, $file);# -----xrw
chmod(15, $file);# -----xrw, because decimal 15 is octal 17
chmod(0xf, $file);# -----xrw, because hexadecimal 15 is
octal 17
chmod((1 << 3) | 7), $file);# -----xrw, because 1 << 3 is
octal 10,
# and 010 | 7 is 017.
chmod(070, $file);# ---rwx---
chmod(7 << 3, $file);# ---rwx---, because 7 << 3 is 070
chmod(0700, $file);# rwx-----
chmod(7 << 6, $file);# rwx-----, because 7 << 6 is 0700
```

```
chmod(0701, $file);# rwx-----x
chmod((7 << 6) | 1, $file);# rwx---rwx
chmod(455, $file);# rwx---rwx, because (7 << 6) | 1 is 449
```

vec()

You'll find in later chapters that it's often helpful (and efficient) to use a string to represent a *vector* of integers. It's just a regular string, but you can access and modify the individual integers (each of which must have the same number of bits) with `vec()`.

The `vec()` Function

`vec(STRING, INDEX, BITLENGTH)` treats *STRING* as a sequence of *BITLENGTH*-long numbers, returning the *INDEX*th number.

BITLENGTH must be 1, 2, 4, 8, 16, or 32. *INDEX* starts at zero, so

```
vec($vector, 0, 1)
```

retrieves the first bit of `$vector` and

```
vec($chars, 4, 8)
```

retrieves the fifth 8-bit number in `$chars`.

You can also assign to `vec()`:

```
vec($chars, 4, 8) = 65;
```

sets the fifth 8-bit number in `$chars` to 65.

Because regular ASCII letters are eight bits in Perl, you can manipulate strings by setting *BITLENGTH* to 8:

```
vec($blood, 0, 8) = 65;
vec($blood, 1, 8) = 66;
vec($blood, 2, 8) = 79;
print $blood;
```

prints ABO.

Review

Mark: Every computer language I've ever seen lets programmers define functions-except BASIC. Perl is the only one I've seen tha lets the programmer define a "default" function. I bet you could even use it to correct typos, so if a programmer mistypes a function name, the correct function is called instead.

Carrie: That's a good idea, but I don't think we've learned quite enough Perl for that. I'm still struggling with scoping...I had to stare at that example in Lesson 4 for about 10 minutes.

Mark: It is kind of confusing. I think of lexical scoping as more selfish than dynamic scoping: "MY variable! All mine! You can't have it!"

Carrie: Cute. So are Perl's global variables lexical or dynamic?

Mark: I suppose you could say that they're dynamic, because subroutines can see the variables. But Perl doesn't really have global variables... it just has variables with a packagewide scope. We'll learn about packages in the next chapter and in Chapter 7.

Chapter 5, Perl 5 Interactive Course



MINCING WORDS: HASHES

Regular arrays are useful for enumerating scalars, so that you can access element number 12 or loop through a series of values from beginning to end. Like regular arrays, hashes (also called associative arrays) are collections of scalars, except that there's no order to the elements. Instead, they associate a string with each element. So you don't refer to "element number 12" in a hash; you refer to "the element associated with the string Harry." That element might be Harry's age, his last name, or any other scalar.

Arrays and hashes are each appropriate for different tasks. When the order of your data matters, you want a conventional array: for an alphabetical listing of words, for recording user input, or for storing the lines in a file. You want a hash when it's more important to name values than to order them.

Hashes strike a balance between time efficiency (how long it takes to retrieve your data) and space efficiency (how much memory the data occupies). They're optimized for speedy retrieval, but take a little more space to store than regular arrays because of their internal organization. That organization is chosen by Perl--not the programmer--but that's a small price to pay for the ability to store data as key-value pairs.

Hashes are the most complex data structure we've covered so far, which is why there's an entire chapter devoted to them (with some fun stuff thrown in: randomness, sleeping, and telling time).

LESSON 1

WHAT'S A HASH?

Regular arrays are indexed by numbers; hashes are indexed by strings. Hashes let you associate one scalar with another, so you can think of a hash as being like a Rolodex that lets you make up your own labels. As you'll see in Lesson 5, you can use hashes to construct nearly any data structure.

Suppose you want to store a baseball roster. The roster should map last names to positions: right field, center field, and so on. How would you do it? You could do it with two regular arrays, in which one array contains the ordered positions and the other contains the names of the players in those positions:

```
@positions = ( 'RF', 'CF', 'LF', 'SS', '1B', '2B', '3B', 'P',  
              'C' );  
@names = ( 'Ruth', 'Mays', 'Cobb', 'Ripken Jr.', 'Gehrig',  
           'Hornsby', 'Schmidt', 'Young', 'Berra' );
```

But there are a number of problems here: Searching for a position, or a player name, means you have to iterate through the entire array. If you rearrange one array, you have to rearrange the other as well. If you delete an element of one array, you have to delete the corresponding element from the other. That's quite a bit of maintenance for such a simple association. Wouldn't you much rather have one structure that associates names with positions?

```
%players = ( RF => 'Ruth', CF => 'Mays', LF => 'Cobb',  
            SS => 'Ripken Jr.', 1B => 'Gehrig', 2B => 'Hornsby',  
            3B => 'Schmidt', P => 'Young', C => 'Berra' );
```

%players is a hash containing nine key-value pairs. All hashes begin with %, just as scalars start with \$, regular arrays start with @, subroutines start with &, and type globs start with *. (There aren't many symbols left on your keyboard.)

Hashes (Associative Arrays)

%HASH = (KEY1 => VALUE1, KEY2 => VALUE2) creates the hash **%HASH**, associating the string **KEY1** with **VALUE1**, and the string **KEY2** with **VALUE2**.
\$HASH{KEY} = VALUE associates **KEY** with **VALUE**.

There's no limit to the number of values you can store in a hash.
Note the similarity between hashes and arrays, as shown in Table 5-1.

Table 5-1 Arrays and hashes

Arrays	Hashes	
The entire structure	@array	%hash
Initialization	@array = (1, 2, 3)	%hash = (name1 => "value1")
A single element	\$array[number]	\$hash{scalar}

There are two ways to store data in hashes:

```
%humble = ( name => 'Abraham', age => 45, occupation =>
```

```
`President`);
```

creates the %humble hash, initialized with three key-value pairs, and

```
$foods{lemon} = `fruit`;
```

adds a single key-value pair that associates the key *lemon* with the value *fruit*. The %foods hash is created, if it didn't already exist.

Keys can be any string:

```
$firstnames{Grant} = `Cary`;
$firstnames{Streep} = `Meryl`;
$age{earth} = 5e+9;
$rank{1} = "One Flew Over The Cuckoo's Nest";
$rank{2} = `Jaws`;
```

You need quotes around the key only if it might be mistaken for something else, like a function call. But it never hurts to use them:

```
$director{`Pulp Fiction`} = `Tarantino`;
$firstnames{"O'Clock"} = "Ben";
$players{`1B`} = "Gehrig";
```

and of course, you can supply keys and values as variables:

```
$hash{$key} = $value;
```

Just be sure that you DON'T do this:

```
%hash = {key1 => "value1", key2 => "value2"}
```

because hashes, like arrays, are declared with parentheses:

```
%hash = (key1 => "value1", key2 => "value2")
%empty_hash = ();
```

Now that you know what hashes look like, you might recall seeing them already, in the *mailfilt* program from Chapter 3. *mailfilt* uses the %fields hash to store the header of the mail message. Listing 5-1 shows the relevant loop.

Listing 5-1 Using a hash to store mail fields and values

```
while (<>) {
  last if /^$/;
  $header .= $_;
  ($field, $value) = /^(\\S*)\\s*(.*)/;
  if (length($field) > 1) {
    $fields{$field} = $value; # add a key-value pair
  } else { $fields{$field} .= " $value"; } # append to that
  key-value pair
}
```

The hash is %fields, the key is \$field, and the value is \$value.

If a mail message begins with these lines:

From: perluser@waite.com

To: joebob@waite.com

then the first \$field will be *From:* and the first \$value will be perluser@waite.com. On the next iteration through the loop, \$field will be *To:* and \$value will be joebob@waite.com:

```
$fields{`From:`} = `perluser@waite.com`  
$fields{`To:`} = `joebob@waite.com`
```

The *else* clause is only triggered if a field is split across more than one line, in which case the current line is appended to the previous \$value.

mailfilt illustrates two hash operations:

- **Adding a new key-value pair:** `$fields{$field} = $value`
- **Modifying a value that's already there:** `$fields{$field} .= " $value"`

The => symbol is almost identical to the comma. The only difference is that => doesn't complain if you leave off the quotes on a bareword under -w:

```
%hash = (key => 6)
```

versus

```
%hash = (`key` => 6)
```

You can create a hash out of any array:

```
%hominid = (`Kingdom`, `Animalia`, `Phylum`, `Chordata`,  
`Class`, `Mammalia`);
```

is equivalent to

```
%hominid = (Kingdom => `Animalia`, Phylum => `Chordata`,  
Class => `Mammalia`);
```

You can even derive hashes directly from arrays:

```
%hash = @array;
```

The even-numbered elements in the array become keys and the odd-numbered elements become values. *movidate*, shown in Listing 5-2, demonstrates the comma notation.

Listing 5-2 *movidate*: The comma notation

```
#!/usr/bin/perl -wl  
%years = ('Yentl', 1983, 'Rebecca', 1940, 'Dave', 1993,  
'Shampoo', 1975, 'Harvey', 1950, 'Forrest Gump', 1995);  
print "Shampoo was released in $years{Shampoo}.";   
print "Rebecca was released in $years{Rebecca}.";
```

The double quotes interpolate \$years{Shampoo} and \$years{Rebecca}, as you'd expect:

```
% movidate  
Shampoo was released in 1975.  
Rebecca was released in 1940.
```


frenchiz (Listing 5-3) contains a very small English-to-French dictionary and uses it to make substitutions.

Listing 5-3 frenchiz: English-to-French dictionary

```
#!/usr/bin/perl -w
%french = (Hello => 'Bonjour',
cup => 'tasse',
'the movies' => 'le cinema',
horseradish => 'raifort');
while (<>) {
s/ \b Hello \b /$french{Hello}/gix;
s/ \b cup \b /$french{cup}/gix;
s/ \b the\ movies \b /$french{'the movies'}/gix;
s/ \b horseradish \b /$french{horseradish}/gix;
print;
}
```

For each line of input, *frenchiz* replaces the keys in its dictionary with the values.

```
% frenchiz
Hello! After the movies, would you like a cup of horseradish?
Bonjour! After le cinema, would you like a tasse of raifort?
[CTRL]-[C]
```

It's a pity that the *while* loop in *frenchiz* needs an explicit line for every hash entry. In the next lesson, you'll see how to avoid this by iterating through a hash.

Here's your first quiz of the chapter. Bonne chance!

Chapter 5, Perl 5 Interactive Course

LESSON 2

MANIPULATING KEYS AND VALUES

This lesson covers the four functions used for manipulating hashes: `keys()`, `values()`, `each()`, and `delete()`.

Retrieving Keys or Values

Looping through regular arrays is easy. Because all the elements are ordered, you can loop through the indices from beginning to end (or, for Perl purists, from `$[` to `$#array`). But hashes aren't linear, so you can't simply loop through them. Instead, you need to extract an array of the keys (with `keys()`) or of the values (with `values()`).

The `keys()` and `values()` Functions

`keys(HASH)` returns a regular array containing all the keys in *HASH*.

`values(HASH)` returns a regular array containing all the values in *HASH*.

```
@keys = keys %hash
```

assigns the keys in `%hash` to `@keys`.

```
@values = values %hash
```

assigns the values in `%hash` to `@values`.

Listing 5-4 shows how to loop through a hash with `keys()`.

Listing 5-4 bodypart: Using `keys()` to iterate through the keys of a hash

```
#!/usr/bin/perl -w
%bodycount = (fingers => 10, bones => 206, muscles => `600+`,
neurons => 1e+11, ribs => 24, `heart chambers` => 4);
foreach $key (keys %bodycount) {
print "Number of $key: $bodycount{$key}\n";
}
```

`keys()` returns an array, which is then used by the *foreach* loop:

```
% bodypart
Number of ribs: 24
Number of heart chambers: 4
Number of neurons: 100000000000
```

```
Number of muscles: over 600
Number of fingers: 10
Number of bones: 206
```

Note that the key-value pairs were printed in a strange order. There doesn't seem to be any relation to the order in which they were assigned to %bodycount. That's because there is no relation--the order of key retrieval depends on the internal function used to store the data. Whenever you modify a hash, the retrieval order can change.

values() is similar to keys(), but returns an array of all the values in the hash, as shown in Listing 5-5.

Listing 5-5 *stars*: Using values() to iterate through the values of a hash

```
#!/usr/bin/perl -w
%roster = (Elizabeth => 'Taylor',
Lana => 'Turner',
Martin => 'Landau',
Kathleen => 'Turner',
Kurt => 'Russell',
Keanu => 'Reeves');
@lastnames = values %roster;
print "@lastnames";
% stars
Taylor Landau Reeves Turner Turner Russell
```

The keys *Lana* and *Kathleen* both have the value *Turner*--values can be duplicated in a hash; keys can't.

Retrieving Keys and Values

You'll often want to access both keys and values simultaneously. For instance, in *bodypart*, the keys are retrieved and then immediately used to access the values. This can be done more efficiently with each().

The each() Function

each(*HASH*) returns a two-element array containing the "next" key-value pair from *HASH*.

each() returns an empty array when it runs out of key-value pairs, so it's often used with a while loop, as opposed to the foreach loops often used with keys() and values():

```
while (($key, $value) = each %hash) {
    print "$key is associated with $value.\n";
}
```

Listing 5-6 uses each() to iterate through the %wines hash:

Listing 5-6 *wines*: The each() function iterates through key-value pairs

```
#!/usr/bin/perl -w
%wines = ( `Cabernet Sauvignon` => `red`, Chardonnay =>
`white`,
`Chenin Blanc` => `white`, `Pinot Noir` => `red`,
Gewurztraminer => `white`, Merlot => `red`,
"Rose`" => `pink`);
while (($key, $value) = each %wines) {
print "I'd like a $value wine...perhaps a $key.\n";
}
```

As with keys() and values(), the retrieval order bears no resemblance to the assignment order:

```
% wines
I'd like a pink wine...perhaps a Rose`
I'd like a red wine...perhaps a Pinot Noir.
I'd like a red wine...perhaps a Merlot.
I'd like a white wine...perhaps a Chenin Blanc.
I'd like a white wine...perhaps a Gewurztraminer.
I'd like a red wine...perhaps a Cabernet Sauvignon.
I'd like a white wine...perhaps a Chardonnay.
```

keys(), values(), and each() all iterate through the hash in the same order. You won't know beforehand what that order is, because it's different for each hash and it can change every time you modify the hash.

But that order won't change unless you modify the hash. As long as you leave the hash unchanged, these functions behave the same way every time they're called. You'll get the same ordering if you simply try to print() a hash:

```
print %hash;
```

which jams all the keys and values together. If you assign it to an array:

```
@array = %hash;
```

then print @array; jams all the keys and values together. \$array[0] contains the first key and \$array[1] the corresponding value. \$array[2] contains the second key, and so on.

However, double-quoted strings don't interpolate hashes:

```
print "%hash"
```

prints "%hash".

One caveat: Don't ever modify a hash while you're iterating through it with each()! Because you change the hash ordering whenever you modify the hash, you could end up seeing a key-value pair twice, or never.

Deleting Key-Value Pairs

You can delete key-value pairs from a hash with `delete()`.

The `delete()` Function

`delete($HASH{KEY})` removes the `KEY`-value pair from `HASH`, returning the deleted value.

`delete()` returns `UNDEF` if `$HASH{KEY}` doesn't exist. So

```
delete $wines{Merlot}
```

deletes the *Merlot-red* key-value pair from `%wines`, returning *red*.

The `delete()` in *wines2* (Listing 5-7) deletes *Chenin Blanc => white* from `%wines`. The value, *white*, is returned (but ignored by the program).

Listing 5-7 *wines2*: The `delete()` function removes elements from a hash

```
#!/usr/bin/perl -w
%wines = ( `Cabernet Sauvignon` => `red`, Chardonnay =>
`white`,
`Chenin Blanc` => `white`, `Pinot Noir` => `red`,
Gewurztraminer => `white`, Merlot => `red`,
"Rose`" => `pink`);
delete $wines{'Chenin Blanc'};
$wines{`Sauvignon Blanc`} = `white`;
while (($key, $value) = each %wines) {
print "I'd like a $value wine...perhaps a $key.\n";
}
```

Note that the order of the wines changes slightly:

```
% wines2
I'd like a pink wine...perhaps a Rose`.
I'd like a white wine...perhaps a Sauvignon Blanc.
I'd like a red wine...perhaps a Pinot Noir.
I'd like a red wine...perhaps a Merlot.
I'd like a white wine...perhaps a Gewurztraminer.
I'd like a red wine...perhaps a Cabernet Sauvignon.
I'd like a white wine...perhaps a Chardonnay.
```

The first and last wines are still in their places, but some of the wines in the middle have moved around.

Chapter 5, Perl 5 Interactive Course

LESSON 3

RANDOMNESS AND TIME

This lesson uses hashes to demonstrate a few fun concepts: random-number generation and determining the time.

Choosing a Random Number

Listing 5-8 presents *trivia*, a trivia game that uses a hash to store question-answer pairs about movies.

Listing 5-8 *trivia*: A movie-trivia game

```
#!/usr/bin/perl -w

# if a trivia.lst file exists, use it.
# (It should contain questions and answers on alternate
lines.)
# Otherwise, use three predefined questions.

if (open (TRIVIA, "trivia.lst")) {
while (<TRIVIA>) {
chomp($question = $_);
chomp($answer = <TRIVIA>);
$questions{$question} = $answer;
}
} else {
%questions = ( `What movie won Best Picture in 1959?` => `Ben
Hur`,
"Who directed `Dances with Wolves`?" => `Kevin Costner`,
"In what year did `Platoon` win Best Picture?" => 1986);
}

# Ask the user all of the questions
while (($question, $answer) = each %questions) {
print $question, "\n";

chomp($input = <>);
if ($input =~ /^$answer$/i) {
print "Correct! \n\n";
} else { print "Sorry. The answer is $answer. \n\n"; }
```

```
}
```

There aren't any new concepts in *trivia*: After finding a list of questions and answers, either from the file *trivia.lst* (available from ftp.waite.com) or from the predefined %questions hash, the program iterates through %questions with each(), printing a question (the key) and seeing whether the user mentions the correct answer (the value) in his or her reply:

```
% trivia
What movie won the Oscar for Best Picture in 1959? Casablanca
Sorry. The answer is Ben Hur.

In what year did 'Platoon' win Best Picture? 1987
Sorry. The answer is 1986.

Who directed 'Dances with Wolves'? Kevin Costner
Correct!
```

If you run *trivia* again, you'll get the same questions in the same order. A better game would mix up the order of the questions.

Computers, in general, can't choose random numbers. Instead, they compute pseudorandom numbers. Your computer uses a simple mathematical function (often just addition followed by multiply and a modulus) to generate a "random" number, which it makes available via a function named something like rand() or random(). It then recycles the output of that function as input to choose the next pseudorandom number. That's used to determine the third number, which in turn determines the fourth, and so on, in a completely predictable fashion.

But all this has to start somewhere: with the seed to the pseudorandom-number generator. The seed determines where this process begins.

You can choose a seed and access the resulting pseudorandom numbers--with srand() and rand().

The srand() and rand() Functions

srand(NUMBER) initializes the random-number generator with the seed *NUMBER*. Use it once, at the beginning of your program.

rand(NUMBER) returns a random floating-point number between 0 and *NUMBER*.

If srand()'s NUMBER is omitted, it uses the current time by default. If rand()'s NUMBER is omitted, it uses 1 by default.

```
srand;
print rand(10);
```

prints a random floating-point number between 0 and 10. The number will be always be less than 10, so

```
int(rand(10))
```

generates a random integer from 0 to 9.

You can choose a random integer between \$x and \$y with

```
int(rand($y - $x + 1)) + $x;
```

The most common mistake when using `rand()` (in fact, just about the only mistake you can make) is forgetting to use `srand()` exactly once, at the beginning of your program. So if you're wondering "How come my Perl program chooses the same random numbers each time?", that's why.

Listing 5-9 shows how to simulate a six-sided die with `srand()` and `rand()`.

Listing 5-9 *diecast*: Using `srand()`

```
#!/usr/bin/perl -w
# Seed the random number generator (with the current time)

srand;
print "Rolled a ", int(rand(6)) + 1, ". \n";
print "Alea iacta est. \n";
```

`rand(6)` is a random fraction between 0 and 6 (excluding 6), so `int(rand(6))` is a random integer between 0 and 5 (including 5) and `int(rand(6))+1` is a random integer between 1 and 6 (including 6).

```
% diecast
Rolled a 4. Alea iacta est.

% diecast
Rolled a 6. Alea iacta est.
```

To pick a random element from an `@array`, you could do this:

```
$number_of_elements = scalar(@array);
$random_index = int(rand($number_of_elements));
$array[$random_index];
```

or you could just do this:

```
$array[rand(@array)]
```

See issues 4 and 6 of *The Perl Journal* for more detailed discussions of random number generation.

Time

If `srand()` seeds the random number generator with the current time, there must be a way for you to access the current time, right? Right.

The `time()` Function

`time()` returns the number of seconds since January 1, 1970, Greenwich Mean Time.

January 1, 1970, is often called the beginning of the "UNIX Epoch" because all UNIX systems measure time from this date. It's used by Perl on all platforms, UNIX and non-UNIX alike.

```
print time
```

might print `821581942`, if you execute it at the right moment.

You may remember the `time()` function from Chapter 3--that's how `stat()` and `lstat()` measure the file modification and access times:


```
$num_seconds = time - (stat($file))[8];
```

prints the number of seconds that have elapsed since \$file was accessed.

Because a minute has 60 seconds, this *while* loop burns cycles for six minutes and then stops:

```
$start_time = time;
while (time < $start_time + 360) {}
```

(Also see the alarm() function in Chapter 9, Lesson 2.)

Here's the shortest Perl script you've seen so far. time() is an okay input for srand(). But because it's rather predictable, it's better to combine time() with the process number, \$\$, as in Listing 5-10.

Listing 5-10 *goodrand*: A reasonably good seed for srand()

```
#!/usr/bin/perl -w
srand(time ^ ($$ + ($$<<15)));
foreach (1..10) {
    print int(rand(10)) + 1, ' ';
}
```

\$\$ is discussed further in Chapter 9, Lesson 2. All you need to know for now is that it's an integer that is different every time someone runs your program.

```
% goodrand
7 9 10 4 4 1 2 8 3 6
```

Listing 5-11 uses this seed to randomize the trivia questions in *trivia*.

Listing 5-11 *trivia2*: Using srand(time ^ \$\$) and rand() to pick a hash key at random

```
#!/usr/bin/perl -w

srand(time ^ ($$ + ($$ << 15)));

if (open (TRIVIA, "trivia.lst")) {
    while (<TRIVIA>) {
        chomp($question = $_);
        chomp($answer = <TRIVIA>);
        $questions{$question} = $answer;
    }
} else {
    %questions = (
        'What movie won Best Picture in 1959?' => 'Ben Hur',
        "Who directed 'Dances with Wolves'?" => 'Kevin Costner',
        "In what year did 'Platoon' win Best Picture?" => 1986);
}

# Ask the user three questions
foreach (1,2,3) {
```

```

@keys = (keys %questions); # Collect all the questions
$index = rand($#keys); # Pick a random question number
$question = $keys[$index]; # Grab that question
$answer = $questions{$question}; # Grab its answer
print $question, ' ';
chomp($input = <>);
if ($input =~ /^$answer$/i) {
print "Correct! \n\n";
} else { print "Sorry. The answer is $answer. \n\n"; }
delete $questions{$question}; # Delete that question from
%questions
}

```

After seeding the random-number generator, *trivia2* picks a random question-answer pair from %questions, poses the question to the user, and then deletes the question/answer pair (with `delete()`) so it won't be asked again.

```

% trivia2
Who directed 'Dances with Wolves'? Kevin Costner
Correct!
What movie won Best Picture in 1959? Gremlins
Sorry. The answer is Ben Hur.
In what year did 'Platoon' win Best Picture? 1987
Sorry. The answer is 1986.

```

Setting aside trivia for the moment, you'll often want your programs to know the time--perhaps for inclusion in a printed report, or for version numbers, or just for the user's benefit. In these cases, the number of seconds since 1970 isn't all that informative.

You can convert `time()`'s output to a more useful format with `localtime()` and `gmtime()`.

The `localtime()` and `gmtime()` Functions

`localtime()` converts the numeric *TIME* to nine time/day/date fields in the local time zone.
`gmtime()` converts *TIME* to the same nine fields in Greenwich Mean Time (GMT).

Both `localtime()` and `gmtime()` use the current time if *TIME* is omitted. In a scalar context, these functions return an English string:

```

$time = localtime;
print $time;

```

prints a string of the form:

```
Sun Sep 22 09:30:42 1996
```

The nine fields returned in a list context are shown in Table 5-2.

Table 5-2 `localtime()` and `gmtime()` fields

Array Index	Meaning
-------------	---------

0	Number of seconds (0..59)
1	Minutes (0..59)
2	Hours (0..23)
3	Day of the month (1..31)
4	Month (0..11) (NOT 1..12!)
5	The year minus 1900
6	Day of the week (0..6) (NOT 1..7!)
7	Day of the year (0-365)
8	Whether it's daylight saving time: 1 if it is, 0 if it isn't.

Functions that convert these fields back into numeric TIMEs can be found in the `Time::Local` module, described in Chapter 8, Lesson 2.

Listing 5-12 grabs the day of the week field from `localtime()`.

Listing 5-12 *weekday*: Accessing the seventh field of `localtime()`

```
#!/usr/bin/perl -wl
@days = ("Sunday", "Monday", "Tuesday", "Wednesday",
"Thursday",
"Friday", "Saturday");
$day = $days[(localtime)[6]];
print "I don't like ${day}s."
```

If you happen to run *weekday* when `(localtime)[6]` is 1, you'll see the following output:

```
% weekday
I don't like Mondays.
```

Because `localtime()` uses the output of `time()` by default, `$days[(localtime)[6]]` is just a shorter way of saying `$days[(localtime(time))[6]]`, which is itself shorthand for `$days[(localtime(time()))[6]]`.

You can use the difference between `localtime()` and `gmtime()` to determine the time difference between your location and London, as shown in Listing 5-13.

Listing 5-13 *timeskew*: Comparing `localtime()` and `gmtime()`

```
#!/usr/bin/perl -w

$hour_here = (localtime)[2];
$hour_there = (gmtime)[2];
$difference = $hour_here - $hour_there;
if ($difference > 0) {

    print "You're $difference hours later than London. \n";

} elsif ($difference < 0) {
```

```
print "You´re ", abs($difference), " hours earlier than London. \n";
} else {
print "You´re in the same time zone as London. \n";
}
```

(localtime)[2] yields the third field returned by localtime(): the current hour. (gmtime)[2] does the same for London.

```
    % timeskew
```

```
    You´re 5 hours earlier than London.
```

This program won't always do the right thing--consider what happens if it's 1 a.m. in London and 11 p.m. locally.

Chapter 5, Perl 5 Interactive Course

LESSON 4

%ENV AND SLEEPING

Whenever a program runs, it inherits an environment from the process that created it. (Processes are covered in Chapter 9.) The environment contains information such as the current directory and the username of the person running the program.

In UNIX, this information is stored in environment variables. Here's an environment, as displayed by the UNIX program *printenv* (or *setenv*, or just plain *env*, depending on what UNIX shell you're using):

```
SHELL=/bin/csh
SHLVL=1
HOME=/u/superman
PATH=/bin:/usr/bin:/usr/local/bin:.
TERMCAP=d0|vt100:do=^J:co#80:li#24 ...
TERM=vt100
USER=superman
HOSTTYPE=MIPSEL
OSTYPE=Ultrix
_=/usr/ucb/printenv
```

As you can see, environment variables are usually uppercase. Perl stores the current environment in the %ENV hash. Non-UNIX users still have access to %ENV; it just doesn't contain quite so much information.

The %ENV Hash

%ENV contains your current environment. You can modify it.

To print your PATH (a list of directories through which your shell searches for executable programs), just type this:

```
% perl -e 'print $ENV{PATH}'
```

Perl provides two other hashes that you'll find useful: %INC and %SIG. You'll learn about them in Chapters 8 and 9, respectively.

You can access and modify %ENV like you would any hash. Listing 5-14 shows how you can print out the contents of your current environment.

Listing 5-14 *my_env*: Accessing your environment with %ENV

```
#!/usr/bin/perl -w
while (($key, $value) = (each %ENV)) {
    print "$key: $value";
}
```

You'll often make use of %ENV to customize your programs or to gather information about the conditions under which your program was launched. The *trivia3* program (Listing 5-15) uses %ENV to find out the name of the user running the program.

Listing 5-15 *trivia3*: Using \$ENV{USER} to access the user's name

```
#!/usr/bin/perl -w

srand(time ^ ($$ + ($$<<15)));

greet();

if (open (TRIVIA, "trivia.1st")) {
    while (<TRIVIA>) {
        chomp($question = $_);
        chomp($answer = <TRIVIA>);
        $questions{$question} = $answer;
    }
} else {
    %questions = (
        'What movie won Best Picture in 1959?' => 'Ben Hur',
        "Who directed 'Dances with Wolves'?" => 'Kevin Costner',
        "In what year did 'Platoon' win Best Picture?" => 1986);
}

foreach (1..3) {
    @keys = keys %questions;
    $index = rand($#keys);
    $question = $keys[$index];
    $answer = $questions{$question};
    print $question, "\n";
    chomp($input = <>);
    if ($input =~ /^$answer$/i) { print "Correct! \n\n"; }
    else { print "Sorry. The answer is $answer. \n\n"; }
    delete $questions{$question};
}

# Use $ENV{USER} if it exists, or the last word
# of $ENV{HOME} otherwise.
```

```

sub greet {
  if ($ENV{USER}) {
    print "\nWelcome to trivia3, $ENV{USER}!\n\n";
  } elsif ($ENV{HOME}) {
    my($name) = ($ENV{HOME} =~ /(\w+)$/);
    print "\nWelcome to trivia3, $name!\n\n";
  }
}

```

There's no guarantee that \$ENV{USER} or \$ENV{HOME} exists. But if either does, the greet() subroutine calls the user by name:

```

% trivia3
Welcome to trivia3, perluser!
In what year did `Platoon` win Best Picture? [CTRL]-[C]

```

Child processes (e.g., your program) inherit their environment from their parent process (e.g., your UNIX shell). When your program modifies %ENV, it changes the current environment, and all processes created by your program inherit that environment. But your program can't modify the environment of its parent. That would be irreverent--and a serious security risk.

Sleeping

What if you want *trivia3* to print a victory message if the user does well? Listing 5-16 presents a program that you could include at the end of *trivia3*. It animates a few ASCII characters.

Listing 5-16 yay: Animating a few ASCII characters

```

#!/usr/bin/perl -w
foreach $i (0..34) {
  print STDERR "\r" . ( ` ` x $i ) . `Y` . ( ` ` x (17 - $i) ) .
    `A`;
  print STDERR ( ` ` x (34 - $i) ) . `Y` . ( ` ` x $i );
}

```

Unless you're running Perl on a sewing machine, *yay* will run far too quickly for you to see the animation. All you'll see is the final result:

```

% yay
YAY

```

You can delay execution with the sleep() function.

The sleep() Function

sleep(NUMBER) makes your program wait for *NUMBER* seconds before resuming execution.

If *NUMBER* is omitted, sleep() sleeps forever.

sleep() returns the number of seconds actually slept (in case your program is woken up prematurely).

sleep()'s resolution is up-to-the-second only; sleep(8) might actually sleep for anywhere between 7 and 9 seconds.

Listing 5-17 shows sleep() in exciting action.

Listing 5-17 *sleeper*: sleep() waits before executing the next statement

```
#!/usr/bin/perl -w
foreach (1..10) {
    sleep 2;
    print "yawn \n";
}
```

Listing 5-18 uses sleep() to slow down the congratulatory message.

Listing 5-18 *yay1*: Using the sleep() command to delay a program

```
#!/usr/bin/perl -w
foreach $i (0..34) {
    print STDERR "\r" . (´´ x $i) . ´Y´ . (´´ x (17 - $i)) .
    ´A´;
    print STDERR (´´ x (34 - $i)) . ´Y´ . (´´ x $i);
    sleep 1;
}
```

Now the printing is far too slow, because there's a one second delay at the end of the loop. A more reasonable delay would be a quarter of a second. But sleep() doesn't let you sleep for a fractional number of seconds. For that, you need to use select(). The yay2 program, shown in Listing 5-19, sleeps for a quarter second during each loop iteration.

Listing 5-19 *yay2*: Using select() to delay a program

```
#!/usr/bin/perl -w
foreach $i (0..34) {
    print STDERR "\r" . (´´ x $i) . ´Y´ . (´´ x (17 - $i)) .
    ´A´;
    print STDERR (´´ x (34 - $i)) . ´Y´ . (´´ x $i);
    select(undef, undef, undef, 0.25);
}
```

Until now, you've used select() only to change the default filehandle. This four-argument use of select() is a different beast altogether and is discussed in Chapter 9, Lesson 6.

[Previous lesson](#)[Quiz](#)[Next lesson](#)

Chapter 5, Perl 5 Interactive Course

LESSON 5

EXISTENCE

Perl performs value judgments: Some values are FALSE, others are undefined, and some don't even exist. This lesson describes the difference.

The defined() Function

defined(VAR) returns TRUE if the variable **VAR** is defined and false otherwise.

A variable is defined() as soon it's given a useful value.

```
$scalar = 8
```

defines \$scalar,

```
@array = (8)
```

defines @array (but @array = () doesn't!), and

```
%hash = (key => "value")
```

defines %hash (but %hash = () doesn't!).

Any variable that has been given a useful value (or subroutine that has been declared) is defined(), which as Listing 5-20 shows, isn't quite the same as being TRUE.

Listing 5-20 *baddef*: Using defined() to determine whether a variable has been set

```
#!/usr/bin/perl -wl  
$number = shift;  
die "I need an argument!" unless $number;  
print "HAPPY!";
```

baddef shifts the first command-line argument into \$number and then dies if \$number isn't TRUE.

```
% baddef  
I need an argument! at baddef line 5
```

That's fine--no number was provided, so the program died as expected. But consider:

```
% baddef 0  
I need an argument! at baddef line 5
```

Not so good. You need to test whether \$number has been defined, not whether it's FALSE (Listing 5-21).

Listing 5-21 *defined*: Using defined() to test whether a variable has been set

```
#!/usr/bin/perl -wl
$number = shift;
die "I need an argument!" unless defined($number);
print "HAPPY!";
```

Because \$number is defined (even though it's FALSE), the program continues:

```
% defined 0
HAPPY!
```

You can also use defined() to see whether arrays or hashes or subroutines exist, as shown in Listing 5-22.

Listing 5-22 *defvar*: Using defined() to test the existence of variables and subroutines

```
#!/usr/bin/perl -wl
print "I have no arguments." unless defined(@ARGV);
print "I have no environment." unless defined(%ENV);
print "There's no my_sub!" unless defined(&my_sub);
```

If *defvar* is called with no arguments, @ARGV won't be defined. %ENV always exists, and there's no my_sub() subroutine declared, so here's what *defvar* prints:

```
% defvar
I have no arguments.
There's no my_sub!
```

defined() won't tell you whether a hash key is valid, however. That's what exists() is for.

The exists() Function

exists(\$HASH{KEY}) is true if **KEY** exists in **HASH**.

Listing 5-23 contrasts existence, definition, and truth.

Listing 5-23 *phillie*: A value can't be TRUE if it's undefined and can't be defined if it doesn't exist

```
#!/usr/bin/perl -w
%phillies = (Kant => `4:00`, Nietzsche => -1,
Popper => 0, Sartre => `` ,
Russell => $the_square_root_of_minus_1);
foreach $phil
(`Camus`, `Russell`, `Popper`, `Sartre`, `Kant`, `Nietzsche`) {
print "$phil: ";
print "$phil is. " if exists $phillies{$phil};
print "$phil is defined. " if defined $phillies{$phil};
```

```
print "$phil is true." if $phillies{$phil};
print "\n";
}
```

For each of six philosophers (note that %phillies only has five key-value pairs; Camus is missing), *phillie* prints whether that philosopher (a) exists, (b) is defined, and (c) is true.

```
% phillie
Camus:
Russell: Russell is.
Popper: Popper is. Popper is defined.
Sartre: Sartre is. Sartre is defined.
Kant: Kant is. Kant is defined. Kant is true.
Nietzsche: Nietzsche is. Nietzsche is defined. Nietzsche is
true.
```

Finally, you can use undef() to "undefine" an expression.

The undef() Function

undef(VAR) undefines the value of *VAR*, which can be a scalar, an array, a hash, or a subroutine. The undefined value (undef) is returned.

If *VAR* is omitted, an undef value is returned. That was used in the last lesson's select() statement:

```
select(undef, undef, undef, 0.25)
```

which is the same as

```
select(undef(), undef(), undef(), 0.25)
```

The undefs mean that you don't care about the first three arguments. They're placeholders, nothing more.

The *genie* program, shown in Listing 5-24, erases its own wish subroutine in an act of self-mutilation.

Listing 5-24 *genie*: Using undef() to erase a subroutine

```
#!/usr/bin/perl
sub wish {
    $wishes++;
    print "You have ", 3-$wishes, " wishes left.\n";
}
while (1) {
    print "\nMake a wish: ";
    chomp($request = <>);
    wish();
    if ($wishes == 3) { undef &wish } # Obliterate the wish()
    subroutine.
}
% genie
Make a wish: Peace on Earth
You now have 2 wishes left.
```

```
Make a wish: Lots o' Money
You now have 1 wishes left.
Make a wish: Good Movies
You now have 0 wishes left.
Make a wish: A cheeseburger!!!!
Undefined subroutine called at ./genie line 10.
```

Chapter 5, Perl 5 Interactive Course

LESSON 6

DATA STRUCTURES

You've now seen the three basic data types offered by Perl: scalars, arrays, and hashes. In this lesson, you'll see how to use them to construct more sophisticated structures.

User-Defined Data Types

Languages like C let you define your own data types as combinations of other data types:

```
struct tree {  
    int age;  
    char *species;  
    char *name;  
    char **friends;  
}
```

Perl doesn't explicitly provide anything like C's *struct*. But as you'll see, scalars, arrays, and hashes are all you need. If you use *references*, you can create arbitrarily sophisticated data structures. (References are formally introduced in Chapter 7, Lesson 2, but they're hiding behind the scenes in this lesson.)

For instance, if you want to store information about many trees, you could mimic the C tree structure using four separate hashes, one for each field:

```
%ages = (tree1 => age1, tree2 => age2);  
%species = (tree1 => species1, tree2 => species2);  
%name = (tree1 => name1, tree2 => name2);  
%friends = (tree1 => friends1, tree2 => friends2);
```

Or you could store everything in the same hash, constructing your own key names by concatenating strings:

```
%trees = (tree1age => age1, tree2age => age2,  
tree1species => species1, tree2species => species2,  
tree1name => name1, tree2name => name2,  
tree1friends => friends1, tree2friends => friends2);
```

Both of these solutions work. But don't bother. Perl 5 provides you with a better way: anonymous variables.

Anonymous Variables

A full discussion of what it means for an array or a hash to be anonymous will have to wait until Chapter 7. In the meantime, look at Listing 5-25.

Listing 5-25 *datatype*: An example of a user-defined data type

```
#!/usr/bin/perl
%elm68 = (name => 'Big Red',
species => 'sequoia',
age => 314,
friends => ['Stilts', 'Beanstalk', 'Lumber Jack']);
print "The age of elm68 is $elm68{age}.\n";
print "Its friends are: @{$elm68{friends}}.\n";
```

elm68 is a hash with four key-value pairs: name, species, age, and friends.

```
% datatype
The age of elm68 is 314.
Its friends are: Stilts Beanstalk Lumber Jack.
```

What's up with that last key-value pair? The key friends is associated with the value

```
['Stilts', 'Beanstalk', 'Lumber Jack']
```

which is a creature you haven't seen before: a reference to an anonymous array. *Anonymous arrays* are just like regular arrays, except that they don't have names, only values. Because that array can be accessed with

```
@{$elm68{friends}}
```

you can access individual elements as follows:

```
${$elm68{friends}}[1]
```

which, as you'll see in Chapter 7, is equivalent to

```
$elm68{friends}[1]
```

Both yield *Beanstalk*.

You can use anonymous arrays to construct multidimensional arrays, as shown in Listing 5-26.

Listing 5-26 *matrix*: Using anonymous arrays to construct a two-dimensional array

```
#!/usr/bin/perl
@matrix = ([ 8, 10, 102 ],
[ 75, 0, 999 ],
[ 12, 200, 1 ]);
print "Third row, second column: $matrix[2][1].\n";
```

The @matrix array contains three elements, each of which is an anonymous array with three elements of its own. Each anonymous array can be thought of as a row of numbers in the matrix.

`$matrix[2]` is the third element of `@matrix`: the anonymous array `[12, 200, 1]`.

`$matrix[2][1]` is the second element of `$matrix[2]`: the scalar `200`.

```
% matrix
Third row, second column: 200.
```

Note that anonymous arrays allow you to create fully multidimensional arrays, not just two-dimensional arrays, because anonymous arrays can themselves contain references to other anonymous arrays. Listing 5-27 creates a multidimensional array.

Listing 5-27 *3Darray*: A three-dimensional array

```
#!/usr/bin/perl -wl
@cube = ([[-4, -3], [2, 1]],
[[-1, -2], [3, 4]]);
print "Height 2, width 1, depth 2: $cube[1][0][1]\n";
```

cube is a three-dimensional array: It contains two anonymous arrays, each of which also contains two anonymous arrays and two numbers.

```
% 3Darray
Height 2, width 1, depth 2: -2
```

You can create anonymous hashes, too, using curly braces. They're useful for naming fields, as you might want to do when building a database:

```
$database{key} = {first_name => "Bob", last_name => "Dobbs"}
```

This lets you retrieve Bob with

```
$database{key}{first_name}
```

Anonymous arrays and hashes can themselves contain anonymous arrays and hashes, which can contain anonymous arrays and hashes, and so on, to your heart's content.

Thanks to symbolic references (Chapter 7, Lesson 2), fields can even name other variables. Here's a glimpse:

```
$left_structure = ["right_structure", 7];
$right_structure = ["left_structure", 6];
print ${$right_structure->[0]}->[1];
```

This prints `7`. You can create circular data structures this way.

Multi-Element Keys

Now that you've learned the *real* way to create multidimensional arrays, you'll learn the bad (Perl 4) way: with hashes and the `$;` special variable.

Perl lets you (seemingly) create multi-element keys, as shown in Listing 5-28.

Listing 5-28 *hashmat*: Creating a multidimensional array with \$;

```
#!/usr/bin/perl -w
$matrix{0, 0} = 1;
$matrix{0, 1} = 2;
$matrix{0, 2} = 3;
$matrix{1, 0} = 4;
$matrix{1, 1} = 5;
$matrix{1, 2} = 6;
foreach $x (0..1) {
  foreach $y (0..2) {
    print "$matrix{$x, $y} ";
  }
  print "\n";
}
```

%matrix appears to be a two-dimensional hash:

```
% hashmat
1 2 3
4 5 6
```

At the beginning of the chapter, we told you that a hash associates one scalar with another. Now we're showing you keys that look an awful lot like full-fledged arrays; for example, `$matrix{1, 2}`. What gives?

The `$hash{scalar1, scalar2}` is just Perl trying very hard to please you. It's equivalent to

```
$hash{ join($;, scalar1, scalar2) }
```

In other words, Perl is converting your array subscripts into a string and using that as the key! It can't just concatenate *scalar1* and *scalar2*, because there's no guarantee that *scalar1scalar2* won't be a key used by the program in its own right. So it has to use a funny sort of glue: the `$;` special variable.

The `$;` Special Variable

`$;` (or `$SUBSEP`, or `$SUBSCRIPT_SEPARATOR`) is the subscript operator for multidimensional array emulation.

By default, `$;` is ASCII 28, an obscure character that doesn't get much use. That's why it's a good choice, because your data is not likely to contain that character. If it does, you can set `$;` to something else that doesn't appear in your key names. But if you're storing binary data, there's no guarantee that there won't be an embedded 28 (or octal 034, or hexadecimal 0x1c; they're all the same). Of course, if your data runs through the entire gamut of ASCII characters, there won't be any safe value--all the more reason to use anonymous arrays instead.

The *Perl Data Structures Cookbook*

Tom Christiansen's *Perl Data Structures Cookbook* (which is available from <http://mox.perl.com/perl/pdsc/>) has a collection of code samples that implement popular data structures.

Arrays of arrays, hashes of arrays, arrays of hashes, hashes of hashes, heterogeneous records, recursive data structures: They're all there.

Chapter 5, Perl 5 Interactive Course

LESSON 7

HASH LEFTOVERS

This lesson addresses four "leftover" hash topics:

- Determining the efficiency of hash storage
- Passing a hash into a subroutine
- How to manipulate symbol tables (they're hashes)
- The misconception of "sorting" a hash

Hash Storage Efficiency

Nothing has been said so far about evaluating hashes, just accessing and modifying them. When you evaluate a hash in a scalar context, it returns FALSE if it contains no key-value pairs. Otherwise, it returns a string containing a fraction. The higher the fraction (up to a maximum of 1), the more efficiently Perl is storing the hash.

Listing 5-29 shows how efficiently Perl stores the wine list.

Listing 5-29 wines3: Calculating the efficiency of hash storage

```
#!/usr/bin/perl -w
%wines = ( `Cabernet Sauvignon` => `red`, Chardonnay =>
`white`,
`Chenin Blanc` => `white`, `Pinot Noir` => `red`,
Gewurztraminer => `white`, Merlot => `red`,
"Rose`" => `pink` );
$efficiency = %wines;
print "Hash storage efficiency: $efficiency.\n";
```

The line `$efficiency = %wines` evaluates the hash in a scalar context, returning (on my computer) the string `5/8`.

```
% wines3
Hash storage efficiency: 5/8.
```

Hashes as Subroutine Arguments

You haven't seen it yet, but, as you'd expect, you can pass a hash into a subroutine. Listing 5-30 counts the number of elements in a hash, using a simple subroutine.

Listing 5-30 wines4: Passing a hash into a subroutine

```
#!/usr/bin/perl -w
%wines = ( `Cabernet Sauvignon` => `red`, Chardonnay =>
`white`,
`Chenin Blanc` => `white`, `Pinot Noir` => `red`,
Gewurztraminer => `white`, Merlot => `red`,
"Rose`" => `pink`);
sub hashcount {
my (%hash) = @_;
scalar(keys %hash);
}
print `We have `, hashcount(%wines), ` wines on our wine
list.`;
```

hashcount(%wines) passes the hash into the subroutine, where it's assigned (through @_, as usual) to the local variable %hash.

```
% wines4
We have 7 wines on our wine list.
```

Quick: If hashcount() modified %hash, would %wines be affected? No, because the my (%hash) = @_ copies the array elements. That makes %hash a local copy of %wines, so %wines won't notice if %hash is modified.

If you want a subroutine to modify a variable, pass a reference to it.

Symbol Tables

Caution: Wizardry ahead!

When you create a plain scalar variable, Perl needs to associate the name of that variable with its value. That sounds like what a hash does and, sure enough, Perl stores your variables in an internal hash called a symbol table.

When you use the *name notation, you're actually referring to a key in the symbol table of the current package. You'll learn more about packages in Chapters 7 and 8; all you need to know now is that every package contains its own symbol table. Every variable you've seen so far is part of the main package, for which the symbol table is

```
%main::
```

which you should think of as a regular hash with a funny six-character name. Because *main* is the default package, an abbreviation is provided:

```
%::
```

`%main::` contains the names and values of the variables in your program. So if your program contains the variable

```
$car = 'Audi'
```

then the "fully qualified name" of `$car` is `$main::{car}`. You can access the value of `$car` through the symbol table hash:

```
$main::{car}
```

But the value of `$main::{car}` isn't *Audi*. It's a type glob:

```
*main::car
```

Why isn't it *Audi*? Because Perl doesn't know that you're looking for a scalar value; you might be asking about `@car` or `%car`. So you can do the usual type glob manipulation:

```
*value = $main::{car};  
print $value;
```

and you'll get your *Audi*. You can also use the shorthand:

```
$main::car
```

or even

```
$::car
```

to access *Audi* directly. These usages are combined in Listing 5-31.

Listing 5-31 *symtest*: Using the symbol table to access a variable

```
#!/usr/bin/perl -w  
$car = 'Audi';  
print 'Symbol table entry: ', $::{car}, "\n";  
print 'Symbol table value: ', $::car, "\n";  
*value = $::{car};  
print 'Symbol table value: ', $value, "\n";  
% symtest  
Symbol table entry: *main::car  
Symbol table value: Audi  
Symbol table value: Audi
```

You can iterate through the symbol table as you would through any hash. Listing 5-32 shows *dumpvar*, which lists the name and value of user-defined variables.

Listing 5-32 *dumpvar*: Printing the contents of the symbol table

```
#!/usr/bin/perl -wl  
$chapter = 5;  
@chapter = ('Associative Arrays', 'Hashes');  
@movies = ('The Associate', 'Hoffa');  
%wines = ('Cabernet Sauvignon' => 'red', 'Chardonnay' =>  
    'white',  
    'Chenin Blanc' => 'white', 'Pinot Noir' => 'red',
```

```

Gewurztraminer => `white`, Merlot => `red`,
"Rose" => `pink`);
while (($Key, $Value) = each(%main::) ) {
    local(*symbol) = $Value;
    next if $Key =~ /^[^a-z]/;
    if (defined $symbol) { # Scalar
        print "\$$Key = $symbol";
    }
    if (defined @symbol) { # Array
        print "@$Key = ", join(`, `, @symbol);
    }
    if (defined %symbol) { # Hash
        print "%$Key = ", join(`, `, %symbol);
    }
}

```

dumpvar iterates through every key-value pair in %main::. The *while* loop skips over any variable whose name has anything but lowercase letters, catching all the special variables like @ARGV and %ENV and \$| and %main::. That's why \$Key and \$Value are capitalized--so they get skipped over, too.

At this point, *dumpvar* doesn't know whether *symbol stands for a scalar, an array, or a hash--or all three. The three if statements catch each one in turn. Because both \$chapter and @chapter are defined, the type glob accesses both:

```

% dumpvar
%wines = Rose`, pink, Pinot Noir, red, Merlot, red, Chenin
Blanc, white, *
Gewurztraminer, white, Cabernet Sauvignon, red, Chardonnay,
white
$chapter = 5
@chapter = Associative Arrays, Hashes
@movies = The Associate, Hoffa

```

For a more thorough variable dumper, use Andreas König's Devel::SymDump module, available from the CPAN--the Comprehensive Perl Archive Network. (URLs are listed in Appendix B.)

Sorting a Hash

Perl novices often ask how to sort a hash. That's an ill-formed question. You shouldn't want to: The whole idea of a hash is that Perl controls the internal structure for maximum efficiency. A better question is "How can you sort the keys of a hash?" or "How do you sort the values?"

Sorting by keys is easy, as shown in Listing 5-33.

Listing 5-33 sortkeys: Sorting a hash by key

```

#!/usr/bin/perl -w
%movies = ( `This is Spinal Tap` => `Rob Reiner`,
`Beverly Hills Cop` => `Martin Brest`,

```

```

Gremlins => `Joe Dante`,
`Raging Bull` => `Martin Scorsese`,
Alien => `Ridley Scott`,
Bugsy => `Barry Levinson`);
foreach $key (sort keys %movies) {
print "$key = $movies{$key}\n";
}

```

That doesn't change the hash at all. It just returns an array of sorted keys, which is used by the *foreach* loop:

```

% sortkeys
Alien = Ridley Scott
Beverly Hills Cop = Martin Brest
Bugsy = Barry Levinson
Gremlins = Joe Dante
Raging Bull = Martin Scorsese
This is Spinal Tap = Rob Reiner

```

Sorting a hash by value is almost as easy. You just need to provide a sorting subroutine that compares `$hash{$a}` and `$hash{$b}`, as shown in Listing 5-34.

Listing 5-34 *sortvals*: Sorting a hash by value

```

#!/usr/bin/perl -w
%movies = (`This is Spinal Tap` => `Rob Reiner`,
`Beverly Hills Cop` => `Martin Brest`,
Gremlins => `Joe Dante`,
`Raging Bull` => `Martin Scorsese`,
Alien => `Ridley Scott`,
Bugsy => `Barry Levinson`);
sub byvalue {
    $movies{$a} cmp $movies{$b};
}
foreach $key (sort byvalue keys %movies) {
print "$key = $movies{$key}\n";
}

```

The sorting subroutine, `byvalue()`, is given the keys `$a` and `$b`. It compares their values by looking them up in `%movies`.

```

% sortvals
Bugsy = Barry Levinson
Gremlins = Joe Dante
Beverly Hills Cop = Martin Brest
Raging Bull = Martin Scorsese
Alien = Ridley Scott
This is Spinal Tap = Rob Reiner

```

Chapter 5, Perl 5 Interactive Course

LESSON 8

QUOTING OPERATORS AND BAREWORDS

In this final lesson, you'll learn about something completely different: *quoting operators*, which give you some new ways of constructing strings (and arrays of strings), and *barewords*, which are strings that have no quotes at all.

q, qq, and qx

You've already learned about single quotes, double quotes, and backquotes. The following quoting operators don't let you do anything you couldn't do before. They just let you do it with different symbols.

When newspaper reporters quote a source, they use double quotes.

```
The official said, "Oof."
```

When they quote that source quoting someone else, they use single quotes inside double quotes.

```
The official said, "The prisoner said, 'Wow.'"
```

What if that person quotes someone else? Then, according to proper written style, you place double quotes inside single quotes inside double quotes:

```
The official said, "The prisoner said, 'He said, "Boy!"'
```

But that's a little too confusing for computers (especially UNIX shells), so Perl provides some alternative ways to mimic quotes.

The q, qq, and qx Operators

q/STRING/ is a way to single-quote *STRING* without actually using single quotes.

qq/STRING/ is a way to double-quote *STRING* without actually using double quotes.

qx/STRING/ is a way to backquote *STRING* without actually using backquotes.

The *q*, *qq*, and *qx* operators let you choose any delimiter you want, as long as it's non-alphanumeric. Or, if the starting delimiter is an open parenthesis (or bracket or brace), then the ending delimiter should be the close parenthesis (or bracket or brace). Listing 5-35 demonstrates these operators.

Listing 5-35 *quotable*: q, qq, and qx behave like single quotes, double quotes, and backquotes

```
#!/usr/bin/perl
$orator = $ENV{USER};
$single = q/$orator said, 'Hi.'\n/;
```

```
$double = qq!$orator said, "Hi."\n!;
$back = qx[date];
print $single, "\n";
print $double, "\n";
print $back, "\n";
```

If this program were run by *perluser* on a computer with the date program, it would print something like the following:

```
% quotable
$orator said, `Hi.`\n
perluser said, "Hi."
Fri Oct 17 11:32:41 EDT 1997
```

qw()

You can quote multiple words with the `qw` operator.

The `qw` Operator

`qw/STRING/` returns an array of the whitespace-separated words in *STRING*.

`qw` is often used with parentheses:

```
@words = qw(hello parlez-vous $hey!)
```

places three strings--*hello*, *parlez-vous*, and *\$hey!*--into `@words`.

`qw(STRING)` is the same as `split(' ', q/STRING/)`, as shown in Listing 5-36.

Listing 5-36 *qwtest*: Using `qw` to quote words

```
#!/usr/bin/perl
@functions = (time, study, rand);
@not_functions = qw(time study rand);
@words = qw (integral)New and improved!(integral);
print `Functions: `, join( ' ', @functions ), "\n";
print `Not functions: `, join( ' ', @not_functions ), "\n";
print `Words: `, join( ' ', @words ), "\n";
```

`@functions` actually calls the functions `time()`, `study()`, and `rand()`. `@not_functions` is assigned three strings: *time*, *study*, and *rand*. `@words` is assigned three words as well: *New*, *and*, and *improved!*.

```
% qwtest
Functions: 878051044, , 0, 0.513870078139007
Not functions: time, study, rand
Words: New, and, improved!
```


quotemeta()

Every once in a while, you'll want to turn a string into a regular expression, but you won't want characters in the string to adopt their metacharacter behaviors. They need to be backslashed--and that's what the `quotemeta()` function does.

The `quotemeta()` Function

`quotemeta(STRING)` returns *STRING* with any regex special characters backslashed.

`quotemeta()` does to strings what `\Q` does to patterns:

```
print quotemeta('.*+b')
```

prints

```
\\.\\*\\+b
```

because `.`, `*`, and `+` are all metacharacters and `b` isn't.

Listing 5-37 shows `quotemeta()` in action.

Listing 5-37 `quotmeta`: `quotemeta()` backslashes anything that can be a metacharacter

```
#!/usr/bin/perl -w
$string = 'Six dots .....';
$pattern = quotemeta($string);
print "Pattern: $pattern \n";
print "String match \n" if 'Six dots !!!!!' =~ $string;
print "Regex match \n" if 'Six dots !!!!!' =~ $pattern;
% quotmeta
Pattern: Six\ dots\ \.\.\.\.\.\.
String match
```

The string

```
Six dots !!!!!
```

matches `$string` but not `$pattern`, because `$string` contains six periods, which match any character (except newlines), whereas `$pattern` contains six backslashed periods, thanks to `quotemeta()`.

Barewords

You don't always need quotes around strings. Consider an equivalent (but stylistically inferior) version of the *complain* program from Lesson 3, in which the days aren't quoted (Listing 5-38).

Listing 5-38 *complain*: Handling a string

```
#!/usr/bin/perl -w
@days = (Sunday, Monday, Tuesday, Wednesday, Thursday,
Friday, Saturday);
```

```
$day = $days[(localtime)[6]];
print "I don't like $day", "s. \n";
```

It'll work. -w won't even complain:

```
% complain
I don't like Mondays.
```

complain could have used *qw*, which knows not to look for quotation marks:

```
@days = qw(Sunday Monday Tuesday Wednesday Thursday Friday
Saturday);
```

But it didn't. So how did Perl know that the members of @days were strings?

If Perl can't interpret a word as anything else, it assumes that the word is a *bareword*: a quoted string. Try to avoid using barewords if possible, because you won't always know whether a word might be interpreted another way. If you say *use strict* (that's a pragma, discussed in Chapter 8, Lesson 1) at the beginning of your program, Perl will treat barewords as syntax errors.

If it's a string, it should look like a string. Use quotes.

Review

Michel: Why are anonymous arrays necessary? Why can't you associate a key with a regular array?

Claudine: Because hashes just map scalars to scalars. And [3,4,5] isn't an anonymous array-it's a reference to an anonymous array, and references are scalars. We'll learn about references in Chapter 7.

Michel: Okay. But I'd like to port a C program, which uses a linked list, to Perl. Each element in the linked list has three fields: a pointer-to-char color, an integer length, and a pointer to the next element. How can I convert that to Perl?

Claudine: There are lots of ways. I'd construct a hash that uses consecutive integers as keys. Every value would be a reference to an anonymous array. That array would contain your two values, as well as the key of the "next" element.

Michel: So each hash value would contain a string for the color, followed by an integer for the length, followed by another integer, which would be the key of the next element. So to get the length of element number 4, I'll be able to say \$table{4}[1]. But suppose I wanted to name the fields, so that I could say \$table{4}{color} and get back "red"?

Claudine: Then you could use a reference to an anonymous hash instead of a reference to an anonymous array.

Chapter 6, Perl 5 Interactive Course



LOOKS ARE EVERYTHING: FORMATS

Every Perl program you've written so far has used `print()` for output, which lets you choose what to print but not where on the screen to print it. This chapter covers *formats*, which allow you to exercise finer control over the appearance of your program's output.

When Larry Wall created Perl, one of his original motivations was report generation: printing documents that adhere to a certain format. Programming languages generally makes you handle the nitty-gritty when formatting documents:

- **The number of lines on a page**
- **Page headers**
- **Document pagination**
- **Multicolumn text**

Perl formats support all these features, allowing you to design the "look and feel" of your documents with ease.

This chapter is not just about the Perl format statement, but about formatting in general: how to arrange data in programs, in files, and on printed pages. To that end, you'll also learn how Perl programs can produce PostScript, a page description language frequently used for printing. In addition, you'll learn how to write binary data, demonstrated with the TIFF image format.

In this book, Chapters 1 through 5 are critical for obtaining a working knowledge of Perl. This chapter isn't quite so important, and formats aren't for everyone. If you're not interested in controlling the appearance of text on pages, just skim this chapter, paying attention only to the following topics: `length()` and `substr()` in Lesson 1, `printf()` and `sprintf()` in Lesson 4, here-strings and literals in Lesson 5, and `pack()` and `unpack()` in Lesson 8.

LESSON 1

INTRODUCTION TO FORMATS

A format is a description of how text should appear on the user's screen. You declare formats just like subroutines: You can place them anywhere in your file, and they won't do anything until invoked. They can only be invoked in one way: with the `write()` function, which is similar to `print()` but uses a declared format to determine what to show and how to show it.

The `write()` Function

`write(FILEHANDLE)` prints formatted text to *FILEHANDLE*, using the format associated with that *FILEHANDLE*.

If *FILEHANDLE* is omitted, `STDOUT` is used.

```
write;
```

prints formatted text to the default filehandle (which is `STDOUT`, unless you've changed it).

```
write FILE;
```

uses the format associated with *FILE* to print to the file associated with *FILE*.

Declaring Formats

Format declarations look a little bizarre. The first line is always **`format name =`**, followed by some other lines, followed by a lone period on a line.

Formats

```
format NAME =
```

```
text1
```

```
text2
```

```
.
```

Formats are used to specify the "look" of your output to the screen or to a file. If *NAME* is omitted, `STDOUT` is assumed.

Normally, Perl ignores spaces and tabs in your programs unless they're part of a quoted string. Formats, however, preserve whitespace: When printed, most of the text in a format declaration will appear exactly as it appears in your program, as shown in Listing 6-1.

Listing 6-1 *verbatim*: Using a format to print verbatim text

```
#!/usr/bin/perl -w
format =
# This is a comment.
"Marie said, 'Let them eat cake!'"
```

```

I've got $12.
Now 50% off!
Print THIS, buddy: !#$$%&*()_-=+={}[ ]|\:;"`<>,.?/\`
# Comments must be "flushed left".
# And don't forget this period:
.
write;

```

verbatim demonstrates the simplest use of formats--for printing characters exactly as shown: spaces, dollar signs, and all. Because the format in *verbatim* has no name, it's associated with the STDOUT filehandle and is therefore triggered by the write(), which prints the format to your screen exactly as shown.

```

% verbatim
"Marie said, `Let them eat cake!`"
I've got $12.
Now 50% off!
Print THIS, buddy: !#$$%&*()_-=+={}[ ]|\:;"`<>,.?/\`

```

Notice that the comments (which begin with # in the first column) weren't displayed. If you call write() twice, the format will be printed twice, as shown in Listing 6-2.

Listing 6-2 *verbatimtwo*: Every write() prints the format

```

#!/usr/bin/perl -w
format =
"Marie said, `Let them eat cake!`"
I've got $12.
Now 50% off!
Print THIS, buddy: !#$$%&*()_-=+={}[ ]|\:;"`<>,.?/\`
.
write;
write;

```

Because there are two write()s, you'll see double:

```

% verbatwo
"Marie said, `Let them eat cake!`"
I've got $12.
Now 50% off!
Print THIS, buddy: !#$$%&*()_-=+={}[ ]|\:;"`<>,.?/\`
"Marie said, `Let them eat cake!`"
I've got $12.
Now 50% off!
Print THIS, buddy: !#$$%&*()_-=+={}[ ]|\:;"`<>,.?/\`

```

Naming Formats

Formats are a data type unto themselves, so you can have a format that's the same name as a scalar, a hash, a subroutine, or a filehandle. In fact, you'll often give a format and a filehandle the same (capitalized) name. When you do, Perl will use that format whenever writing to that filehandle.

So if you have a filehandle named CINEPLEX, you can create a format to be used automatically with that filehandle, as shown in Listing 6-3.

Listing 6-3 *cinplex*: Associating a format with a filehandle

```
#!/usr/bin/perl -w
format CINEPLEX =
Ticket price: $9.75
Matinee: $6.25
.
open (CINEPLEX, ">cinplex.out") or die "Can't write
cinplex.out: $! ";
write CINEPLEX;
close CINEPLEX;
```

The *cinplex* program first declares a format called CINEPLEX and then creates a filehandle called CINEPLEX. Then the **write CINEPLEX** is executed, writing the contents of the CINEPLEX format to the file *cinplex.out*:

```
% cinplex; cat cinplex.out
Ticket price: $9.75
Matinee: $6.25
```

(MS-DOS users should use *type* instead of *cat*.)

What if you want to use the same format for two different filehandles? You can associate any particular format (regardless of its name) with a filehandle using the `$~` special variable.

The `$~` Special Variable

`$~` (or `$FORMAT_NAME`) contains the name of the format associated with the current filehandle.

By default, `$~` is the name of the filehandle itself; that's why **write CINEPLEX** used the CINEPLEX format.

So to set the format for the FILE filehandle to the format BANNER, you'd want something like this:

```
select(FILE);
$~ = "BANNER";
select(STDOUT);
```

logouts, shown in Listing 6-4, first declares a LOGOUT format and then opens two filehandles for appending: USER1 and USER2. Then the LOGOUT format is associated with both filehandles by first selecting each filehandle and then setting `$~`. When *logouts* is run, each `write()` appends *Bye bye!* to the

corresponding file.

Listing 6-4 *logouts*: Assigning a format to a file with the \$~ special variable

```
#!/usr/bin/perl -w

format LOGOUT =
echo "Bye bye!"
.

open(USER1, ">>/users/superman/.logout") or die "Can't
append: $! ";
open(USER2, ">>/users/tankgirl/.logout") or die "Can't
append: $! ";

select(USER1); $~ = `LOGOUT`; # Assign LOGOUT to USER1
select(USER2); $~ = `LOGOUT`; # Assign LOGOUT to USER2 also
select(STDOUT);

write USER1;
write USER2;

close USER1;
close USER2;
```

Formatting Without Formats

Although the bulk of this chapter is about controlling the arrangement of your files with formats, you can also control text placement using a couple of string functions: `length()` and `substr()`.

The `length()` Function

`length(STRING)` returns the number of characters in *STRING*.

If *STRING* is omitted, `$_` is used.

```
$a = length;
```

assigns the number of characters in `$_` to `$a`.

```
print length($string);
```

prints the number of characters in `$string`.

You can create columns with `length()` and a little elbow grease. If you have a series of words to be placed into columns, you can repeat the following procedure for each line:

- **Print the first word.**
- **Print enough spaces to move to the second column.**

- **Print the second word.**
- **Print enough spaces to move to the third column.**

...and so on. *topfive*, shown in Listing 6-5, lines up three arrays, one per column.

Listing 6-5 *topfive*: Using `length()` to print columns of output

```
#!/usr/bin/perl -w
@movies = ('Sybil', 'Philadelphia', 'Ghost', 'Death Be Not
Proud', 'Batman');
@numbers = (1..5);
@years = (1977, 1993, 1990, 1975, 1989);
foreach $i (0..$#movies) {
    print "$numbers[$i]." . ' ' x 3; # up to column 5
    print $movies[$i] . ' ' x (25 - length($movies[$i])); # up to
    column 30
    print "$years[$i]\n";
}
```

The first `print()` displays `$i` at the beginning of each line, followed by a period and three spaces. That moves the "print point" to position 5.

The second `print()` displays a movie title, followed by some spaces. How many? It depends on the length of the title: **25 - `length($movies[$i])`**. That moves the print point to position 30, regardless of the title length.

The final `print()` prints the year, followed by a newline. The results:

```
% topfive
1. Sybil 1977
2. Philadelphia 1993
3. Ghost 1990
4. Death Be Not Proud 1975
5. Batman 1989
```

topfive uses `length()` to build each line from the bottom up, piece by piece.

Another approach is to create a space-filled line first and then overlay your text onto that line in the appropriate places. But to do that, you need to know how to access and modify a substring within a string. Enter the `substr()` function.

The `substr()` Function

`substr(STRING, OFFSET, LENGTH)` accesses a *LENGTH*-character substring out of *STRING* (starting at *OFFSET*).

In Perl, strings are indexed starting from 0, just like arrays. So:

```
substr($word, 14, 1)
```

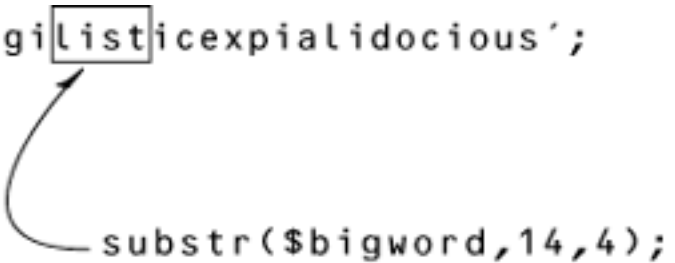

is the 15th character.

```
substr($word, 5, 10)
```

is the ten characters following (and including) the 6th character.

Figure 6-1 shows a four-character substring of \$bigword beginning at position 14 (which is the 15th letter).

```
$bigword = 'supercalifragilisticexpialidocious';
```



```
substr($bigword, 14, 4);
```

Figure 6-1

substr()

If *OFFSET* is negative, the substring starts that far from the end of the string, just like array indices:

```
$array[-2]
```

is the second-to-last element of @array.

```
substr($string, -2, 1)
```

is the second-to-last character.

If *LENGTH* is omitted, the rest of the string is returned:

```
substr($string, 4)
```

is all characters after the fourth.

```
substr($string, -3)
```

is the last three characters.

You can use substr() to perform assignments:

```
substr($string, 3, 2) = 'hi';
```

sets the fourth and fifth characters to *h* and *i*.

```
substr($string, 3, 2) = ``;
```

removes the fourth and fifth characters.

```
substr($string, 8, length($word)) = $word;
```

overlays \$word onto \$string, beginning after the eighth character.

You can use substr() to format columns of text without worrying about those pesky print points, as shown in Listing 6-6.

Listing 6-6 *topfive2*: Using substr() to print columns of output

```
#!/usr/bin/perl -wl
@movies = ( `Grease`, `Footloose`, `Dave`, `Forrest Gump`,
`Doctor Zhivago` );
@numbers = (1..5);
@years = (1978, 1984, 1993, 1994, 1965);
# for each movie, create a string of length 35, and then
# overwrite segments of that string.
foreach $i (0..$#movies) {
$line = ` ` x 35;
substr($line, 0, length($numbers[$i])) = $numbers[$i];
substr($line, 5, length($movies[$i])) = $movies[$i];
substr($line, 30, length($years[$i])) = $years[$i];
print $line;
}
```

For each movie, *topfive2* creates a line filled with 35 spaces. It then overlays the line with the number at position 0, with the movie title at position 5, and the year at position 30. Here's *topfive2* in action:

```
% topfive2
1. Grease 1978
2. Footloose 1984
3. Dave 1993
4. Forrest Gump 1994
5. Doctor Zhivago 1965
```

topfive and *topfive2* both include variables in the formatted text. The next lesson describes how to include variables in formats.

Chapter 6, Perl 5 Interactive Course

LESSON 2

PICTURE FIELDS

The `substr()` and `length()` combination used in *topfive2* let you include variables in your formatted output. Formats let you do that, too, with picture fields. Picture fields are special symbols inside a format that are replaced by variables.

Aligning Variables

Picture fields are a format's way of interpolating variables. How Perl formats these variables (which must be scalars or arrays) depends on which picture field is used. The three most common fields are `@<`, `@|`, and `@>`.

The Picture Fields `@<`, `@|`, and `@>`

`@<<<<` left-flushes the string appearing on the following line of the format.

`@|||` centers the string appearing on the following line of the format.

`@>>>` right-flushes the string appearing on the following line of the format.

The longer the symbol, the longer the field:

```
@<<<<  
$string
```

prints the first five (the `@`, plus four `<`) characters of `$string`, flush left.

```
@|||  
$value
```

prints the first four characters of `$value`, centered.

```
@>  
@array
```

`@array` prints the first two characters of concatenated elements of `@array`, flush right.

A lone `@` lines up single characters.

The length of the picture field determines how many characters are included in the picture field. So `@>>>` (which has a length of 4) means "take the variable on the next line and right-justify the first four characters at `@`." *topfive3*, shown in Listing 6-7, uses four picture fields to line up four variables.

Listing 6-7 topfive3: Using picture fields to line up variables

```
#!/usr/bin/perl -w

@movies = ('Top Gun', 'The Godfather', 'Jaws',
           'Snow White', 'Sphere');

@grosses = ('79,400,000', '86,275,000', '129,549,325',
            '67,752,000', '4,193,500,000');

@years = (1986, 1972, 1975, 1937, 2002);
@prices = ('$5.00', '$2.50', '$4.00', '$.25', '$14.50');

# define a format with four picture fields, each of which
# uses the contents of the variable below.

format TOPFIVE =

@@<<<<<<<<< @<<<<<<<<< @|||   @>>>>>
$movie,          $gross,          $year,$price

.

open(TOPFIVE, ">topfive3.out") or die "Can't write
topfive3.out: $! \n";
foreach $i (0..$#movies) {
    $movie = $movies[$i];
    $gross = $grosses[$i];
    $year = $years[$i];
    $price = $prices[$i];
    write TOPFIVE;
}
}
```

Note that the *topfive* format lines up the variables with their picture fields: \$movie appears directly under @<<<<<<<<<<, \$year appears directly under @|||, and so on. This isn't necessary; the variables could have appeared anywhere because the first picture field grabs the first variable, the second picture field grabs the second variable, and so on. The *topfive3* format could just as well have looked like this:

```
@<<<<<<<<< @<<<<<<<<< @|| @>>>>>>>>
$movie, $gross, $year, $price
```

but not

```
@<<<<<<<<< @<<<<<<<<< @||| @>>>>>
Data $movie, $gross, $year, $price
```

because Perl will try to interpret the entire line as a call to the subroutine `Data` with four arguments: `$movie`, `$gross`, `$year`, and `$price`.

Here's what happens when you run *topfive3* and look at its output:

```
% topfive3; cat topfive3.out
Top Gun          79,400,000 1986 $ 5.00
The Godfather    86,275,000 1972 $ 2.50
Jaws             129,549,325 1975 $ 4.00
Snow White      67,752,000 1937  $ .25
Sphere          4,193,500,000 2002 $14.50
```

Everything looks good--except for the prices. They're right-justified (with @>>>>>) and look a little funny that way. A better choice for dollar values, and all numbers with decimal points, is the @# picture field.

The @# Picture Field

@###.## formats numbers by lining up decimal points under the ".". Extra numbers after the decimal point are rounded.

Suppose a lottery is held in which the first prize of \$5000 is shared between two people and the second prize of \$2000 is shared among three people. When you display the lottery results, the @# field serves two equally important purposes: lining up the decimal points and rounding the prizes to the nearest cent, as shown in Listing 6-8. (You can also round numbers with `printf()`, `sprintf()`, and a multiply-int()-divide sequence.)

Listing 6-8 *lottery*: The @# field lines up numbers on decimal points, rounding if necessary

```
#!/usr/bin/perl -w

$first = 5000 / 2;
$second = 2000 / 3;

%winnings = (Bob => $second, Connie => $first,
             Adam => $first,   Kirsten => $second,
             Fletch => $second);

format =
@<<<<<<<< @###.##
$key,        $value
.

write while (($key, $value) = each %winnings);
```

lottery iterates through each key-value pair of %winnings, setting \$key and \$value and then writing to

STDOUT. The first nine characters of each key are left-justified (overkill, because the longest name is seven characters), and each value is aligned with the `@#` picture field, which lines up all the decimal points. Here's what *lottery* prints:

```
% lottery
Kirsten 666.67
Adam    2500.00
Bob      666.67
Connie   2500.00
Fletch   666.67
```

If the number has more digits before the decimal point than the number field, Perl sacrifices the decimal-point alignment to avoid losing data. If the *lottery* format was

```
format =
@<<<<<<<< @#.##
$key          $value
.
```

then the decimal-point alignment would be sacrificed so that the most significant digits could be preserved:

```
Kirsten 666.6
Adam    2500.
Bob      666.6
Connie   2500.
Fletch   666.6
```

Adding `@#` fields to the *topfive* program is easy, as shown in Listing 6-9.

Listing 6-9 *topfive4*: `@#` fields center numbers on the decimal point

```
#!/usr/bin/perl -w

@movies = ( 'Top Gun', 'The Godfather', 'Jaws',
            'Snow White', 'Sphere' );

@grosses = ( '79,400,000', '86,275,000', '129,549,325',
            '67,752,000', '4,193,500,000' );

@years = (1986, 1972, 1975, 1937, 2002);
@prices = ( '5.00', '2.50', '4.00', '.25', '14.50' );
```

```
format TOPFIVE =
@<<<<<<<<<<< @>>>>>>>>>>>>>>> @||| | @#.##
$movie, $gross, $year, $price
.
```

```
open(TOPFIVE, ">topfive4.out") or die "Can't write
topfive4.out: $! \n";
foreach $i (0..$#movies) {
    $movie = $movies[$i];
    $gross = $grosses[$i];
    $year = $years[$i];
    $price = $prices[$i];

    write TOPFIVE;

}
```

topfive4 lines up its four fields, as before, but also aligns the prices on their decimal points:

```
% topfive4; cat topfive4.out
Top Gun      79,400,000    1986 5.00
The Godfather 86,275,000    1972 2.50
Jaws         129,549,325    1975 4.00
Snow White   67,752,000    1937 0.25
Sphere       4,193,500,000 2002 14.50
```

Chapter 6, Perl 5 Interactive Course

LESSON 3

PAGES

Suppose you want to generate an eight-page report. In addition to the "meat" of your document, you might want your document to keep a few things constant from page to page. Perhaps you want the title at the top of every page, in a header.

Headers

Every filehandle has two formats associated with it: the format you've already seen and a top-of-page format. The *top-of-page* format is printed at the beginning of the document (with the first write()) and at the beginning of every page. You declare top-of-page formats the same way you declare normal formats, and you can explicitly associate a top-of-page format with a filehandle by setting the `$^` special variable.

The `$^` Special Variable

`$^` (or `$FORMAT_TOP_NAME`) is the name of the current top-of-page format. By default, `$^` is the name of the filehandle with `_TOP` appended.

To set the top-of-page format for the filehandle `MARQUEE` to `MARQUEE_TITLE`, you'd do this:

```
select(MARQUEE);
$^ = "MARQUEE_TITLE";
select(STDOUT);
```

If you name the format `MARQUEE_TOP` instead of `MARQUEE_TITLE`, you don't even have to `select()`.

If a top-of-page format exists, its contents are written to the file before the normal format associated with the file.

Now that you have some control over the page-by-page look of your output, Listing 6-10 shows how you might create a simple ASCII newsletter.

Listing 6-10 *variety*: Using a top-of-page format

```
#!/usr/bin/perl -w
%receipts = (Ghost => 9.4e+7, 'Pretty Woman' => 8.19e+7,
'Home Alone' => 8e+7, 'Die Hard 2' => 6.65e+7,
'Total Recall' => 6.5e+7);
format VARIETY_TOP =
```



```

----- E-VARIETY -----
.
format VARIETY =
@<<<<<<<<<< $@#####
$key, $value
.
open (VARIETY, ">variety.out") or die "Can't write
variety.out: $! \n";
write VARIETY while (($key, $value) = each %receipts);
close VARIETY;

```

variety uses a top-of-page format to print the title of the newsletter. Each write() behaves as normal, except that when the first variety format is written, VARIETY_TOP comes along for the ride.

```

% variety
% cat variety.out
----- E-VARIETY -----
Home Alone $80000000
Pretty Woman $81900000
Die Hard 2 $66500000
Ghost $94000000
Total Recall $65000000

```

Of course, that's not enough data to fill a page. Suppose you have a large file containing a list of movie grosses. Each line contains a movie title, followed by a space, followed by a number. If there are enough movies in the list, they won't all fit on one page, and two (or more) headers will be printed, as shown in Listing 6-11.

Listing 6-11 *variety2*: Using top-of-page formats for multipage documents

```

#!/usr/bin/perl -w
format VARIETY_TOP =
----- E-VARIETY -----
.
format VARIETY =
@<<<<<<<<<< $@#####
$key, $value
.
open (INPUT, "variety.lst") or die "Can't find variety.lst:
$! ";
while (<INPUT>) {
/((.*) (\S*))$/;
$receipts{$1} = $2;
}
close INPUT;
open (VARIETY, ">variety.out") or die "Can't write
variety.out: $! ";

```

```
write VARIETY while (($key, $value) = each %receipts);
close VARIETY;
```

The first *while* loop fills the hash %receipts with movie titles (the keys) and their grosses (the values).

Assuming there's more than a page worth of movies, more than one header will be printed:

```
% variety2; cat variety.out
----- E-VARIETY -----
Raiders of t $24240000
Indiana Jone $19490000
Unforgiven $10120000
Terms of End $10840000
Empire Strik $22310000
...<a pageful of movies>
----- E-VARIETY -----
Karate Kid P $11510000
Superman: Th $13420000
Officer and $12980000
Driving Miss $11060000
Dances With $18420000
...
```

Pagination

Pagination is the organization of information into pages: determining what goes where, deciding how to break text across pages, and displaying page numbers. The newsletter could use some page numbers; the \$% special variable (Figure 6-2) can help.



Figure 6-2

The `$%` special variable

The `$%` Special Variable

`$%` (or `$FORMAT_PAGE_NUMBER`) is the current page number.

Caveat: `$%` changes only if you've defined a top-of-page format.

You can access `$%` anywhere in your program, including inside a format statement. `variety3`, shown in Listing 6-12, places the page number in its top-of-page format, using `$%` and a picture field.

Listing 6-12 *variety3*: Using `$%` to print the page number

```
#!/usr/bin/perl -w
format VARIETY_TOP =
----- E-VARIETY ----- page @<
$%
```

```

.
format VARIETY =
@<<<<<<<<<< $@#####
$key, $value
.
open (INPUT, "variety.lst") or die "Can't find variety.lst:
$! \n";
while (<INPUT>) {
/(.*) (\S*)$/;
$receipts{$1} = $2;
}
close INPUT;
open (VARIETY, ">variety") or die "Can't open variety.out: $!
\n";
write VARIETY while (($key, $value) = each %receipts);
close VARIETY;

```

variety3 places the page number in the upper-right corner of each page:

```

% variety3; cat variety.out
----- E-VARIETY ----- page 1
Raiders of t $24240000
Indiana Jone $19490000
Unforgiven $10120000
Terms of End $10840000
Empire Strik $22310000
....<a pageful of movies>
----- E-VARIETY ----- page 2
Karate Kid P $11510000
Superman: Th $13420000
Officer and $12980000
Driving Miss $11060000
Dances With $18420000
....

```

Lines on a Page and Footers

There is no such thing as a bottom-of-page format (a footer). Why not? Because (as you'll see in later lessons) Perl can't always tell how long a format will be, which makes it hard to determine whether a format will fit on the current page. So your programs have to decide explicitly when to write footers.

The `$=` and `$-` special variables (Figure 6-3) help, by telling you the length of your pages and the number of lines left on the current page.

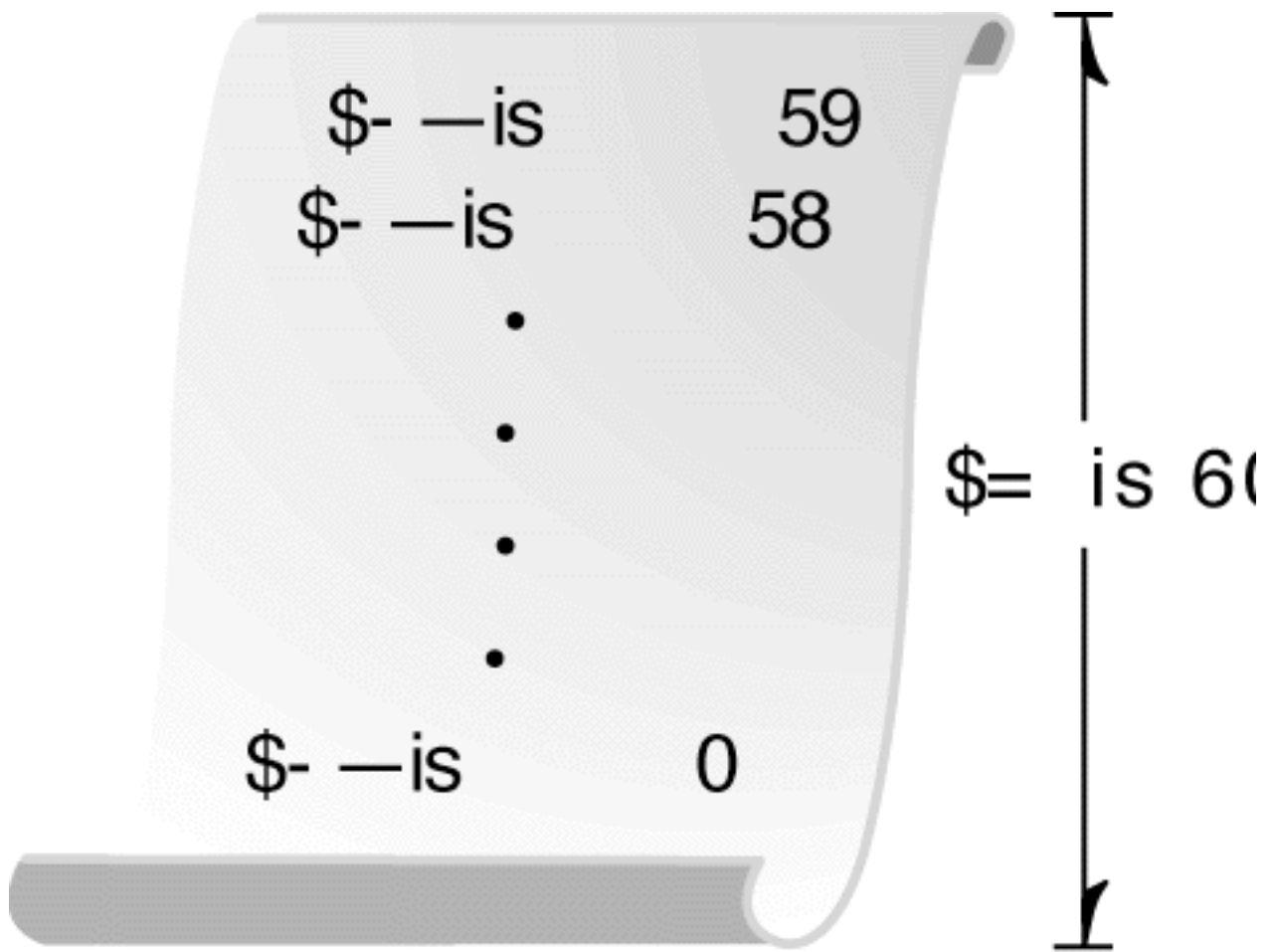


Figure 6-3

The \$= and \$- special variables

The \$= and \$- Special Variables

\$= (or **\$FORMAT_LINES_PER_PAGE**) is the page length (of the current filehandle). By default, **\$=** is 60.

`$-` (or `$FORMAT_LINES_LEFT`) is the number of lines left on the page (of the current filehandle).

To set the page length for `FILE` to 24 lines:

```
select(FILE);
$= = 24;
select(STDOUT);
```

You can assign values for both of these variables: Setting `$=` changes the number of lines on a page, and setting `$-` changes the number of lines left on the page. Setting it to 0 forces the top-of-page format to be printed. Listing 6-13 uses these variables.

Listing 6-13 *bottom*: Emulating a "bottom-of-page" format with `$-`

```
#!/usr/bin/perl -w
format STDOUT_TOP =
Page @
$%
.
format STDOUT =
Lines left: @<, out of @<
$-, $=
.
format MY_BOTTOM =
Uh oh! I'm running out of space!
.
$= = 5;
foreach (1,2,3) {
write;
write while ($- > 1);
$~ = MY_BOTTOM;
write;
$~ = STDOUT;
}
```

After declaring the three formats, *bottom* sets the number of lines per page to 5. Then, for each page, the first `write()` prints the top-of-page format and the first line from the `STDOUT` format. The second `write()` is embedded in a *while* loop that chugs along until there's only one line left on the page. Then the format changes to `MY_BOTTOM` so that the third `write()` prints the footer.

At the top of the *foreach* loop, `$-` is 0, not 5. That's why the first `write()` is necessary--to set `$-` to the proper value before entering the *while* loop in the next line.

```
% bottom
Page 1
Lines left: 0, out of 5
Lines left: 3, out of 5
```

```

Lines left: 2, out of 5
Uh oh! I'm running out of space!
Page 2
Lines left: 0, out of 5
Lines left: 3, out of 5
Lines left: 2, out of 5
Uh oh! I'm running out of space!
Page 3
Lines left: 0, out of 5
Lines left: 3, out of 5
Lines left: 2, out of 5
Uh oh! I'm running out of space!

```

If you're using \$- and \$= on a filehandle other than STDOUT, remember to select() the filehandle first! Also note that print() doesn't decrement \$-, so if you intermix print()s and write()s, you shouldn't rely on it.

The *formfeed* is an ASCII character that instructs your system to start a new page (by clearing the screen on a monitor and by ejecting a page on a printer). Another way to emulate a bottom-of-page format is to redefine the formfeed, conveniently stored in the \$^L special variable.

The \$^L Special Variable

\$^L (or \$FORMAT_FORMFEED) is printed by a format when a formfeed is called for. By default, \$^L is \f.

Normally, the formfeed is an ASCII character (a [CTRL]-[L], or \f in Perl and C). But you can set it to anything you want: a bunch of newlines, a decorative string containing the page number, or even *More on the next page*.

The *formfeed* program, shown in Listing 6-14, uses \$^L to construct a footer.

Listing 6-14 *formfeed*: Using \$^L to define a footer

```

#!/usr/bin/perl -w
format STDOUT_TOP =
Top of Page @<
$%
.
format STDOUT =
All work and no play...
.
$^L = "\n---- More on the next page ----\n\n";
$= = 5;
foreach (1..9) {
write;
}

```

formfeed creates a footer by setting the value of \$^L. When the program is run, the following output is

produced:

```
% formfeed
Top of Page 1
All work and no play...
All work and no play...
All work and no play...
All work and no play...
---- More on the next page ----
Top of Page 2
All work and no play...
All work and no play...
All work and no play...
All work and no play...
---- More on the next page ----
Top of Page 3
All work and no play...
```

Chapter 6, Perl 5 Interactive Course

LESSON 4

FORMATTED STRINGS

Format declarations allow you to control the arrangement of data on a page. But you can control the arrangement of data in strings, too.

Printing a Formatted String

C programmers are familiar with `printf()`, which prints a formatted string by interpolating a list of fields into a template. C's `fprintf()` does the same thing for a filehandle. Perl's `printf()` does both.

The `printf()` Function

`printf(FILEHANDLE TEMPLATE, ARRAY)` prints a formatted string to *FILEHANDLE*.

If *FILEHANDLE* is omitted, `printf()` uses the currently selected filehandle (STDOUT by default).

The *TEMPLATE* is a string that can contain regular text and fields beginning with `%`. These are similar to picture fields, except that instead of adhering to variables on the next line of a format declaration, they adhere to variables in the *ARRAY* that follows.

For instance, the `%d` field expects a decimal number, so

```
printf(FILE "The number is %d", 15);
```

prints *The number is 15* to FILE.

Nothing you couldn't do with a regular `print()`, right? But check this out:

```
printf("Hex 255 is %x, and 1/3 rounded off to 5 places is  
%1.5f", 255, 1/3);
```

prints

```
Hex 255 is ff, and 1/3 rounded off to 5 places is 0.33333.
```

because the `%x` grabs the first value of the *ARRAY* (that's 255), converting it to a hexadecimal number, and the `%1.5f` grabs the second value of the *ARRAY* (that's 0.333...), truncating the result after five decimal places.

As you've seen, the template fields determine how the *ARRAY* values are interpreted. You can think of the template fields as creating contexts: `%d` creates a "decimal" context and `%s` creates a "string" context, so

```
printf("The error number is %d, and the error string is  
%s.\n", $!, $!);
```

prints

```
The error number is 2, and the error string is No such file
or directory.
```

Listing 6-15 demonstrates a simple usage of printf().

Listing 6-15 jeeves: Using printf()

```
#!/usr/bin/perl -w
$name = $ENV{USER};
$myname = $ENV{`_`};
$age = 55;
printf "Hello, %s. I'm %s. I'm %d years old.\n", $name,
$myname, $age;
```

The first %s means "interpret the first field of *ARRAY* as a string"; the second %s means "interpret the second field of *ARRAY* as a string"; and the %d means "interpret the third field of *ARRAY* as a decimal number."

Here's what might happen if *perluser* ran the *jeeves* program:

```
% jeeves
Hello, perluser! I'm jeeves. I'm 55 years old.
```

Table 6-1 shows the printf() template fields.

Table 6-1 printf()'s fields

Field	Expected Value
%s	String
%c	Character
%d	(Decimal) number
%ld	Long decimal number
%u	Unsigned decimal number
%lu	Long unsigned number
%x	Hexadecimal number
%lx	Long hexadecimal number
%o	Octal number
%lo	Long octal number
%f	Fixed-point floating-point number
%e	Exponential floating-point number
%g	Compact floating-point number

The template fields listed in Table 6-1 convert the elements of printf() *ARRAY* to the specified type. You've seen that oct() converts an octal number to a decimal number. printf() does the opposite:

```
printf "%o", 42;
```

prints 52 to STDOUT because decimal 42 is octal 52. The %o field was used to retrieve the umask in Chapter 3:

```
printf "%o", umask;
```

printf() can do another thing print() can't--give you control over the width and precision of fields. Most of the fields listed in Table 6-1 let you specify a width and precision in the form %*w.p*, where *w* is the minimum width of the printed field and *p* is the number of digits after the decimal point (the precision). The particular behavior of width and precision varies from field to field and computer to computer; if you're on a Unix system, read the printf() documentation for further details.

But you can always omit either the width or the precision:

```
printf("%.7f", 4 * atan2(1, 1))
```

prints the first seven digits of $\frac{1}{4}$ after the decimal point.

The width specifier pads the string with spaces, if necessary:

```
printf("%5d", 2)
```

prints four spaces followed by a 2.

When the width and precision conflict, the precision wins, just as with the \$# picture field.

```
printf("%.5f", 987.654321)
```

prints 987.65432.

```
printf "The logarithm of %d is %3.2f ", 100000, log(100000);
```

prints the natural logarithm of 100,000, up to the second digit after the decimal: *11.51*.

Creating a Formatted String

You can construct a formatted string (as opposed to merely printing it) with the sprintf() function. sprintf() behaves like printf(), but returns the string instead of printing it.

The sprintf() Function

sprintf(TEMPLATE, ARRAY) returns a formatted string.

printf(TEMPLATE, ARRAY) is equivalent to print sprintf(TEMPLATE, ARRAY). The TEMPLATE is the same as for printf(), so all the fields in Table 6-1 are valid choices.

You can use sprintf() to convert a decimal number into octal or hexadecimal (the opposite of oct() and hex()). Those uses of sprintf(), and others, are demonstrated in Listing 6-16.

Listing 6-16 *sprintfy*: Using sprintf()

```
#!/usr/bin/perl -w
$octal = sprintf("Octal value: %o", 127);
$hexadecimal = sprintf("Hex value: %x", 48879);
$number = 8;
```

```
$alphabetty = sprintf("The %dth letter of the alphabet is
%c.",
$number, $number | 0x40);
$greeting = sprintf("Greeting, %s.", `perlian`);
print "$octal \n";
print "$hexadecimal \n";
print "$alphabetty \n";
print "$greeting \n";
```

Here's what happens when you run *sprintf*:

```
% sprintfy
Octal value: 177
Hex value: beef
The 8th letter of the alphabet is H.
Greetings, perlian.
```

You probably won't need `printf()` or `sprintf()` very often. They're most often used for converting between numeric formats.

Record Separators

Perl likes newlines. Whenever you open a file for reading, you "read between the newlines"--each line (or record, to use database terminology) is defined as the characters between `\n`'s. The input record separator, `$/`, lets you change that definition. The output record separator, `$\`, allows you to append a certain string whenever you `print()` a piece of data.

The `$/` and `$\` Special Variables

`$/` (or `$RS`, or `$INPUT_RECORD_SEPARATOR`) defines what a "line" is when reading from a filehandle.

`$\` (or `$ORS`, or `$OUTPUT_RECORD_SEPARATOR`) specifies what is printed after every `print()`.

`$/` and `$\` might look like opposites. They aren't. `$\` is about the behavior of `print`; `$/` is about how to interpret data in a file.

The Input Record Separator

By default, `$/` is `\n`, as you'd expect. That's why `$line = <FILE>` reads `FILE` up to and including the next `\n`. As a special treat, if `$/` is set to an empty string (or `\n\n`), files are read paragraph by paragraph (assuming there's a blank line between paragraphs) instead of line by line. And if you `undef($/)`, you'll read the whole file at once.

You can read the entire file into a string by `undef()`ing `$/`, as shown in Listing 6-17.

Listing 6-17 *1string*: Reading an entire file into a string by undef()ing \$/

```
#!/usr/bin/perl -w
undef $/;
open (FILE, "director.lst")
    or die "Can't read director.lst: $! \n";

$_ = <FILE>;
print;
```

1string removes the input record separator by undef()ing \$/. The first read from FILE is therefore also the last because the entire file is treated as a single record.

If your file consists of blocks of text, each separated by a blank line, you can read the text block by block instead of line by line, by setting \$/ to \n\n, as shown in Listing 6-18.

Listing 6-18 *paragraph*: Reading a file by paragraphs instead of lines

```
#!/usr/bin/perl -w
$/ = "\n\n";
open(FILE, "review.txt")
    or die "Can't open review.txt: $! \n";

while (<FILE>) {
    print "paragraph: \n$_\n";
}
```

If you run *paragraph* in a directory with the file *review.txt* (available from the Waite Group FTP site), you'll see the following output: **paragraph:**

"Car" Crashes

paragraph:

Director Lee Franklin has outdone himself again. "My Life As A Car" is a charming coming-of-age story with a twist: instead of a lovelorn pimply teen, the protagonist is a '74 Plymouth Duster (named, of course, "Dusty") whose broken muffler prevents him from making other car friends.

paragraph:

At times, the dialogue is hard to follow, due to the nonstop exhaust noise generated by the muffler-lacking Dusty. When Dusty was finally reunited with his original muffler, lost during an attempt to save a wayward kitten (artfully played by John Candy) from becoming roadkill on I-95, I was happy. Happy for Dusty, happy for the kitten (who had

curiously grown up into a fetching white Honda), and happy that I could finally hear myself think again.

`$/` can be used to separate the header and body of a mail message without looping:

```
$/ = ``; $header = <>; # Read everything up to a blank line
undef $/; $body = <>; # Read everything that's left.
```

The Output Record Separator

By default, `$\` is an empty string, as you'd expect. But if you're tired of always having to append a newline to everything you print, you can save some typing by setting `$\` to `\n`, as shown in Listing 6-19.

Listing 6-19 *noline*: Using `$\` to print a newline after every `print()`

```
#!/usr/bin/perl -w
$\ = "\n";
print `Look ma, no newlines!`; # no newline needed!
print `No pesky commas or double quotes.`;
print `They uglify my code.`;
```

noline sets `$\` to a newline, which has the same effect as the `-l` command-line flag.

```
% noline
Look ma, no newlines!
No pesky commas or double quotes.
They uglify my code.
```

Fun choices for `$\` include a period, a double newline, and a carriage return plus a newline.

Field and List Separators

In addition to the output record separator, there is also an output field separator `$,` and a list separator `$"`. The first has to do with `print()`; the second has to do with array interpolation.

The Special Variables `$,` and `$"`

`$,` (or `$OFS`, or `$OUTPUT_FIELD_SEPARATOR`) is implicitly inserted between arguments to `print()`.

`$"` (or `$LIST_SEPARATOR`) is implicitly inserted between interpolated array elements.

By default, `$,` is an empty string, which is why

```
print 1, 2, "three", 4, 5, 6;
```

jams the numbers together: *12three456*.

`$"` is, by default, a space, which is why

```
@array = (1, 2, "three", 4, 5, 6);
```

```
print "@array"; prints 1 2 three 4 5 6.
```

These all print *1 and-a 2 and-a 3*:

```
$, = ' and-a ';  
@numbers = (1,2,3);  
print @numbers;  
$" = ' and-a ';  
print "@numbers";  
print join(' and-a ', @numbers);
```

There's even a special variable for controlling the numeric output format: `$#`. It's deprecated, and so might vanish in future versions of Perl. Avoid it.

The `$#` Special Variable

`$#` is the output format for numbers.

`$#` is initially undefined, but can be set to a template field. For example, to round all fractional numbers to two decimal places, use `"%.2f"`:

```
$# = "%.2f";
```

`print 1/3;` prints 0.33.

Chapter 6, Perl 5 Interactive Course

LESSON 5

HERE-STRINGS AND SPECIAL LITERALS

This lesson is really about "funny ways to organize your Perl programs." *Here-strings* are a way to bypass standard quoting mechanisms entirely.

The *special literals* (`__DATA__`, `__END__`, `__FILE__`, and `__LINE__`) are tokens with special meanings inside Perl programs.

Here-Strings

A here-string is just another way to quote some text. Here-strings superficially resemble formats, in the sense that they start on a particular line and continue until an identifier appears on a line by itself.

Here-Strings

```
<<IDENTIFIER;
```

```
multiline text
```

```
goes here
```

```
IDENTIFIER
```

A here-string contains a chunk of text delimited by two *IDENTIFIERS*.

You can assign a here-string to a regular scalar:

```
$string = <<END_OF_MY_STRING;  
line1  
line2  
line3  
END_OF_MY_STRING
```

This assigns a three-line string to `$string`, everything between the `END_OF_MY_STRING` identifiers. `$string` now contains

```
line1  
line2  
line3
```

or, in other words, *`$string eq "line1\nline2\nline3\n"`*.

There are two important things to remember about here-string syntax:

- You can't have any space between the `<<` and the identifier.

- You usually need a semicolon after the first identifier (because it's a statement).

Listing 6-20 shows a simple example that uses a here-string to emulate a format.

Listing 6-20 *carshow*: Using a here-string

```
#!/usr/bin/perl -w
$carlist = <<LASTCAR;
Acura
BMW
Chevrolet
Datsun
LASTCAR
print $carlist;
```

carshow places the here-string (delimited by the identifier LASTCAR) into \$carlist, newlines and all.

```
% carshow
Acura
BMW
Chevrolet
Datsun
```

If the first *IDENTIFIER* is quoted somehow, the *TEXT* is treated as though it were inside those quotes:

```
$string = `Revolution`;
$num = 9;
# $string and $num will NOT be interpolated
print <<'END_OF_SINGLE_QUOTED_STRING';
$string
$num
END_OF_SINGLE_QUOTED_STRING
```

prints

```
$string
$num
```

but

```
$string = `Revolution`;
$num = 9;
# $string and $num will be interpolated
print <<"END_OF_DOUBLE_QUOTED_STRING";
$string
$num
END_OF_DOUBLE_QUOTED_STRING
```

prints

```
Revolution
9
```

You can backquote here-strings, too:

```
print <<END_OF_BACKQUOTED_STRING;
ls -lF
END_OF_BACKQUOTED_STRING
```

might print something like

```
-rws--x--x 4 system root 40960 bigprog
-rwxr-xr-- 2 system perluser 109 original
lrwxrwxrwx 1 system perluser 12 symlink -> original
-rwxr--r-- 2 system perluser 109 hardlink
```

If you don't use any sort of quotes, double-quote behavior is used: Backslashed characters take on their special meanings, and variable substitution occurs, as shown in Listing 6-21.

Listing 6-21 *timechek*: By default, here-strings act like double quotes

```
#!/usr/bin/perl -w
$hour = (localtime)[2] + 1;
$minute = (localtime)[1] + 1;
print <<END_OF_TEXT;
At the tone, the time will be:

$hour:$minute...exactly.
\a
END_OF_TEXT
```

Here's what you would see (and hear!) if you ran *timechek* at 1:12 pm.

```
% timechek
At the tone, the time will be:
1:12...exactly.
<beep!>
```

If you don't specify an identifier (in other words, if your identifier is an empty string), the next line becomes the here-string. The following hurls an epithet at you five times:

```
print << x 5;
Go fry asparagus!
```

prints

```
Go fry asparagus!
Go fry asparagus!
Go fry asparagus!
Go fry asparagus!
Go fry asparagus!
```

Special Literals

There are four literals, encased in double underscores, that pertain to the physical file containing your program: `__END__`, `__DATA__`, `__FILE__`, and `__LINE__`.

Special Literals

`__DATA__` and `__END__` can be placed at the end of your program; any text after that point won't be executed and can be read with the DATA filehandle.

`__FILE__` is the current filename.

`__LINE__` is the current line number.

`__END__` is helpful for storing auxiliary data at the end of a file, such as usage instructions, a description, or any raw data that your program might need. That data can be read via the special filehandle DATA. You don't need to open() DATA, so

```
while (<DATA>)
```

iterates through all the data following the `__END__` token.

As of Perl version 5.001m, the `__DATA__` token can be used in place of `__END__`. What's the difference? You can have a `__DATA__` in every package (packages are discussed further in Chapter 8), so that

```
while (<Movie::DATA>)
```

iterates through the text at the end of the Movie package.

meshfile, shown in Listing 6-22, uses `__DATA__` to store a usage string that helps the user run the program.

Listing 6-22 *meshfile*: Using `__FILE__`, `__DATA__`, and DATA

```
#!/usr/bin/perl -w
if (@ARGV != 3) {
    print "Incorrect number of arguments for ", __FILE__, "\n";
    while (<DATA>) { # print all lines following __DATA__
        print;
    }
    exit -1;
}
open (FILE1, "$ARGV[0]") or die "Can't read $ARGV[0]: $! ";
open (FILE2, "$ARGV[1]") or die "Can't read $ARGV[1]: $! ";
open (FILE3, ">$ARGV[2]") or die "Can't write $ARGV[2]: $! ";
@lines1 = <FILE1>;
@lines2 = <FILE2>;
while (@lines1 && @lines2) {
    print FILE3 shift(@lines1), shift(@lines2);
}
__DATA__
```

```
Usage: file1 file2 file3
```

```
This program interleaves file1 and file2 by line,  
writing the result to file3.
```

When *meshfile* is run, the number of arguments (@ARGV in a scalar context) is checked; if the user supplies any number of arguments other than three, two events occur:

- *meshfile* prints its warning: **Incorrect number of arguments for followed by the filename (__FILE__).**
- *meshfile* reads from DATA, accessing the text following **__DATA__**.

Here's what happens when you call *meshfile* with only two arguments:

```
% meshfile file1 file2
```

```
Incorrect number of arguments for ./meshfile
```

```
Usage: file1 file2 file3
```

```
This program interleaves file1 and file2 by line,  
writing the result to file3.
```

Chapter 6, Perl 5 Interactive Course

LESSON 6

ADVANCED FORMATS

So far, everything you've learned about formats has assumed that your strings fit on a single line. But if you want to print, say, an entire paragraph of text, you can't use regular @ picture fields; you need multiline picture fields.

Multiline Picture Fields

The Picture Fields ^<<<, ^|||, and ^>>>

^<<< left-justifies multiline text.

^||| centers multiline text.

^>>> right-justifies multiline text.

These picture fields are similar to their @ counterparts, but they modify the variables on the following line. Every time you call write(), these fields chop off the front of your string. How much? As much as can fit within the field. Listing 6-23 is an example.

Listing 6-23 *textpoet*: Placing multiline text into columns with the ^| picture field

```
#!/usr/bin/perl -w
$martyr = <<EOS;
So there I am, minding my own business
on this page, and then, WHAM! I'm trapped inside
this stupid column of letters. I went from total marginless
freedom (such unfettered raggedness!)
to the smothering constraints of the
paper's edge, to this prison of t-e-x-t
that squeezes the lexical juice from my cellulose veins,
(to coin a mixed meta-4)
EOS
format =
^| | | | | | | | | |
$martyr
.
write while ($martyr);
```

textpoet first creates the multiline \$smarty with a here-string. It then declares a simple format that creates a centered column 13 letters wide (the caret plus 12 vertical bars). *textpoet* produces the following output:

```
% textpoet
```

```
So there I
am, minding
  my own
business on
this page,
  and then,
  WHAM! I´m
    trapped
inside this
stupid column
of letters.
I went from
  total
  marginless
freedom (such
  unfettered
raggedness!)
  to the
  smothering
constraints
  of the
paper´s edge,
  to this
prison of t-
e-x-t that
squeezes the
lexical juice
  from my
  cellulose
veins, (to
coin a mixed
  meta-4)
```

By changing the ^| to a ^>, you can make a ragged-left column:

```
So there I
am, minding
  my own
business on
  this page,
```

and then,
WHAM! I'm
trapped
inside this
stupid column
of letters.
I went from
total
marginless
freedom (such
unfettered
raggedness!)
to the
smothering
constraints
of the
paper's edge,
to this
prison of t-
e-x-t that
squeezes the
lexical juice
from my
cellulose
veins, (to
coin a mixed
meta-4)

There are a few things to note about the behavior of multiline text fields:

- 1. In the second example, the 11th, of letters., line has a space at the end. Just because the text is right-justified doesn't mean that there can't be a space in the final column.**
- 2. If a word is larger than the column, it's included anyway.**
- 3. Note that t-e-x-t was broken across a line. Lines can be broken on whitespace, hyphens, or newlines; the first hyphen in t-e-x-t was interpreted as a word boundary.**

You can change the line-breaking behavior with the \$: special variable.

The \$: Special Variable

\$: (or **\$FORMAT_LINE_BREAK_CHARACTERS**) is the current set of characters after which a string can be broken in a multiline field.

By default, **\$:** is `\n-` (that's a space, a newline, and a hyphen) to break on all whitespace, newlines, and hyphens.

You can modify **\$:** to change the set of characters that can break up lines of text. To prevent textpoet from breaking up t-e-x-t, you can set a new value for **\$:**.

```
$: = " \n"
```

after which multiline strings will break on whitespace or newlines, but not hyphens.

If you want to break on all letters, you could append them all to **\$:**, as demonstrated by Listing 6-24.

Listing 6-24 *birdy*: **\$:** contains the characters that can separate multiline text

```
#!/usr/bin/perl -w  
$: .= join('', 'a'..'z'); # add all lowercase letters to $:  
$string = <<BIRD;  
The spiny red-beaked throat warbler  
is the most peculiar of birds, hatching  
completely cubical eggs, perfect for stacking.  
BIRD  
format =  
^>>> ^||| ^<<<  
$string, $string, $string  
.  
write while ($string);
```

birdy first adds all the lowercase letters (join()ed into a string) to **\$:**. After the here-string is stored in **\$string**, the format is declared; it sets up three multiline fields of four characters each. Every iteration of the *write-while* loop lops 12 characters off **\$string**: four for each field. Here's the result:

```
% birdy  
The spin y re  
d-be aked thro  
at w arbl er i  
s th e mo st p  
ecul iar of b  
irds , ha tchi  
ng c ompl etel  
y cu bica l eg  
gs, perf ect  
for stac king  
.
```

Multiline picture fields sometimes leave unsightly blank lines. You can eliminate them with a lone tilde:

~. If you want your multiline fields to keep writing until the string is used up, use a double tilde: ~~.

Multiline Picture Field Modifiers: ~ and ~~

~ suppresses blank lines.

~~ repeats the line until all the fields are exhausted.

You can place ~ and ~~ anywhere in the picture field line.

Suppose a two-line format is used to write a string ("Hello there!") that's short enough to fit on one line. With ~, you can suppress the extra line, as shown in Listing 6-25.

Listing 6-25 *suppress*: Using ~ in a picture field

[illegible]

If that tilde weren't in the format, the output of *suppress* would be two lines: *Hello there!* and a blank line (because the first picture field writes all of \$string, leaving nothing for the second picture field). But ~ tidies up things:

```
% suppress
Hello there!
<no blank line here!>
```

~ is useful for short strings. ~~ is useful for longer strings, telling Perl to repeat the field over and over until the variable is used up. So you can eliminate that funny-looking *write while (\$string)* in *birdy*, as shown in Listing 6-26.

Listing 6-26 *birdy2*: Using `~~` to repeat a picture field as many times as necessary

```
#!/usr/bin/perl -w
$string = <<BIRD;
The spiny red-beaked throat warbler
is the most peculiar of birds, hatching
completely cubical eggs, perfect for stacking.
BIRD
format =
~~ ^<<<<<<<<<<<<<<<<<<
```

The Picture Field ^###

`^###` behaves just like `@###`, but prints nothing at all (rather than 0) if its value doesn't exist.

Accessing Your Formatted Data Without Printing

Formats provide a panoply of powerful features. Too many, in fact, to be saved just for printing to filehandles. Sometimes you'll want to place the results of a format into a string instead. `sprintf()` lets you place `printf()` data into a string, but there's no way to do the same thing for a format. Instead, you have to use the internal function employed by formats: `formline()`.

The formline() Function and the \$^A Special Variable

formline(*FORMAT*, *ARRAY*) formats *ARRAY* according to *FORMAT* and sticks the result in the special variable \$^A (or \$ACCUMULATOR).

formline() is used for its side effect--setting `$^A`--and not for its return value (which is always true). You must access the contents of `$^A` before a `write()` occurs, because `write()` resets it, as shown in Listing 6-27.

Listing 6-27 *grabform*: Using `formline()` and `$^A`

file:///C:/e-books/Perl 5/6-06.htm (6 of 7) [10/1/1999 12:04:51 AM]

Note that there's no format declaration in *grabform*, just a formline statement that resembles a format, but without any variables below the picture field. But that's okay; the \$string appears at the end of the declaration line, as an additional argument to formline().

```
% grabform
Format contents:
The spiny red-beaked
throat warbler is
the most peculiar of
birds, hatching
completely cubical
eggs, perfect for
stacking.
```

Chapter 6, Perl 5 Interactive Course

LESSON 7

GENERATING POSTSCRIPT

"Format" is one of those overused words that has adopted several different meanings. So far, you've seen two different uses in this chapter:

- **The organization of the data sent to a filehandle: the format declaration**
- **The organization of data placed in a string: `printf()` and `sprintf()`**

In this lesson and the next, you'll learn about two more formats:

- **The graphical appearance of your data: creating simple pictures and "graphical" text with PostScript**
- **The structure of binary data in a file: file formats**

The PostScript Page Description Language

PostScript(TM) (a trademark of Adobe Systems, Inc.) is a language used by graphics programs, printers, and some windowing systems. It's tailored for specifying the look of text and images on a page. Your programs (or you) can construct PostScript programs that, when interpreted by a "device" (e.g., a printer), display the results for you.

In this lesson, you'll build a Perl program that creates a PostScript file (i.e., a .ps file) suitable for feeding to a printer or to a PostScript previewer.

PostScript Headers and Footers

The first step is creating a skeletal PostScript file, one that contains only a header and footer. Normally, there would be PostScript commands between the header and footer; without them, the generated picture will be blank. But it's a good way to test whether the headers and footers are valid for your system.

You can create a blank (but valid) PostScript file with *makeps* (Listing 6-28), which uses here-strings to construct a PostScript header and footer.

Listing 6-28 *makeps*: Creating a valid (but blank) PostScript file

```
#!/usr/bin/perl -w
main();
sub main {
    open(PS, ">$ARGV[0].ps") or die "Can't open $ARGV[0].ps: $!";
    print "Creating $ARGV[0].ps...\n";
    ps_print_header();
    ps_print_footer();
}
sub ps_print_header {
    print PS <<END_PS_HEADER;
    %!PS-Adobe-2.0
    %%DocumentPaperSizes: Letter
    %%Orientation: Portrait
    %%EndComments
    %%EndProlog
    %%BeginSetup
    %%PaperSize: Letter
    %%EndSetup
    %%Page: 1 1
    %%PageBoundingBox: 66 288 546 756
    END_PS_HEADER
}
sub ps_print_footer {
    print PS <<END_PS_FOOTER;
    showpage
    %%PageTrailer
    %%Trailer
    %%Pages: 1 1
    %%showpage
    %%BoundingBox: 66 288 546 756
    END_PS_FOOTER
}
% makeps test
```

creates the file *test.ps*.

After running *makeps*, you should be able to view the PostScript file through most image-viewing utilities or by sending it to a PostScript printer.

This lesson introduces several PostScript (not Perl!) functions, with a Perl subroutine wrapped around each one. Instead of demonstrating each subroutine as it's introduced, we've saved them all until the end and integrated them into the *haveaday* program.

Drawing Lines

PostScript always maintains a *current point*. Like the print point discussed earlier in this chapter, the current point is wherever PostScript is about to draw.

PostScript is a postfix language; when you give it a command (such as *lineto*, which draws a single line), you provide the function name after the arguments:

```
200 100 lineto
```

This draws a line from the current point to (200, 100). PostScript coordinates are measured from the bottom-left corner, so that's 200 "dots" from the left edge and 100 "dots" from the bottom edge.

The *moveto* PostScript command moves the current point, so

```
157 319 moveto
```

moves the current point to 157 dots from the left edge and 319 dots from the bottom.

The `ps_line()` subroutine accepts four numeric arguments (let's call them `x1`, `y1`, `x2`, and `y2`), and draws a line from the point (`x1`, `y1`) to (`x2`, `y2`):

```
sub ps_line {  
  print PS "$_[0] $_[1] moveto \n";  
  print PS "$_[2] $_[3] lineto \n";  
  print PS "stroke";  
}
```

Lines created by *lineto* aren't actually drawn until PostScript receives the *stroke* command.

Here's how you could use `ps_line()` to create a PostScript file with a single diagonal line, from (100, 100) to (200, 200):

```
open (PS, ">$ARGV[0].ps") or die "Can't write $ARGV[0].ps: $!  
";  
ps_print_header();  
ps_line(100, 100, 200, 200);  
ps_print_footer();
```

Drawing Text

You can draw text with the PostScript *show* command. The `ps_text()` subroutine moves the current point, selects a font (Helvetica), makes it larger or smaller (with *scalefont*), and then makes it the default font (with *setfont*). Then the text is drawn with *show* and stroked onto the page, as shown in Listing 6-29.

Listing 6-29 Printing text with `findfont`, `show`, and `stroke`

```
sub ps_text {  
  print PS "$_[0] $_[1] moveto \n";  
  print PS "/Helvetica findfont $_[2] scalefont setfont \n";  
  print PS "$_[0] $_[1] ($_[3]) show \n";  
}
```

```
print PS "stroke \n";
}
```

You could call this from `main()` as follows:

```
ps_text(100, 200, 12, "Oblate spheroid");
```

would print the 12-point text *Oblate spheroid* at the position (100, 200).

Circles, Squares, and Fills

PostScript lets you draw more than just lines and text, of course. The *arc* and *rectstroke* commands let you draw circles and rectangles and the *rectfill* command lets you paint rectangular areas, as demonstrated in Listing 6-30.

Listing 6-30 Circles, rectangles, and fills

```
sub ps_circle { # x y radius
print PS "$_[0] $_[1] $_[2] 0 360 arc \n";
print PS "stroke \n";
}
sub ps_rect { # x y width height
print PS "$_[0] $_[1] $_[2] $_[3] rectstroke \n";
}
sub ps_fill { # x y width height c
print PS "$_[0] $_[1] $_[2] $_[3] rectfill \n";
}
```

These are all pretty simple: They're just Perl wrappers around PostScript commands.

- `ps_circle()` expects the (x, y) position of the center of the circle, followed by its radius.
- `ps_rect()` expects the (x, y) position of the lower-left corner of the rectangle, followed by its width and height.
- `ps_fill()` expects the same arguments as `ps_rect()`.

Listing 6-31 puts this all together into a program that creates a face.

Listing 6-31 *haveaday*: A Perl program that generates a PostScript program that draws a face

```
#!/usr/bin/perl -w
open(PS, ">face.ps") or die "Can't open face.ps: $! ";
ps_print_header();
ps_circle(273, 378, 150); # draw the face
ps_line(203, 300, 343, 300); # draw the mouth
ps_rect(173, 420, 70, 30); # draw the left eyeball
ps_fill(203, 430, 10, 10); # draw the left pupil
ps_rect(303, 420, 70, 30); # draw the right eyeball
```

```

ps_fill(333, 430, 10, 10); # draw the right pupil
ps_text(167, 150, 40, "Have a day.");
ps_print_footer();
sub ps_print_header {
print PS <<END_PS_HEADER;
%!PS-Adobe-2.0
%%DocumentPaperSizes: Letter
%%Orientation: Portrait
%%EndComments
%%EndProlog
%%BeginSetup
%%PaperSize: Letter
%%EndSetup
%%Page: 1 1
%%PageBoundingBox: 66 288 546 756
END_PS_HEADER
}
sub ps_print_footer {
print PS <<END_PS_FOOTER;
showpage
%%PageTrailer
%%Trailer
%%Pages: 1 1
%%BoundingBox: 66 288 546 756
END_PS_FOOTER
}
sub ps_line {
print PS "$_[0] $_[1] moveto \n";
print PS "$_[2] $_[3] lineto \n";
print PS "stroke \n";
}
sub ps_text {
print PS "$_[0] $_[1] moveto \n";
print PS "/Helvetica findfont $_[2] scalefont setfont \n";
print PS "$_[0] $_[1] ($_[3]) show \n";
print PS "stroke \n";
}
sub ps_circle {
print PS "$_[0] $_[1] $_[2] 0 360 arc \n";
print PS "stroke \n";
}
sub ps_rect {
print PS "$_[0] $_[1] $_[2] $_[3] rectstroke \n";
}
sub ps_fill {
print PS "$_[0] $_[1] $_[2] $_[3] rectfill \n";
}

```


}

The result, stored in *face.ps*, is shown in Figure 6-4.

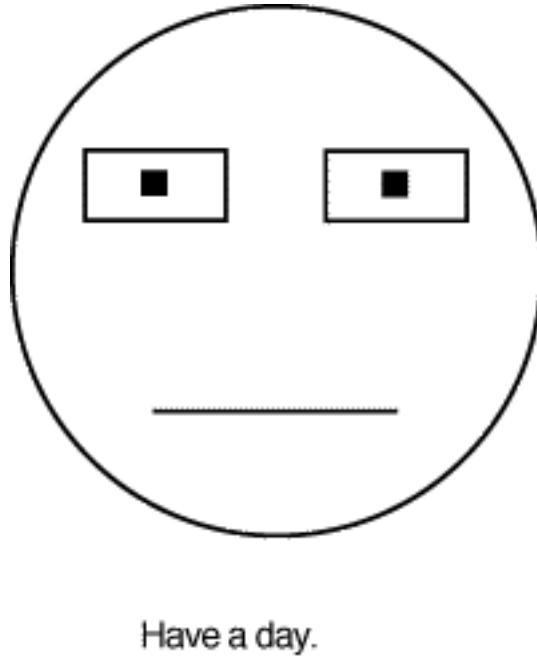


Figure 6-4

***haveaday* PostScript output**

This lesson has covered only the most commonly used PostScript functions, out of the hundreds available. To learn more about PostScript, consult the *PostScript Language Reference Manual*, 2nd edition (San Diego: Addison-Wesley, 1990).

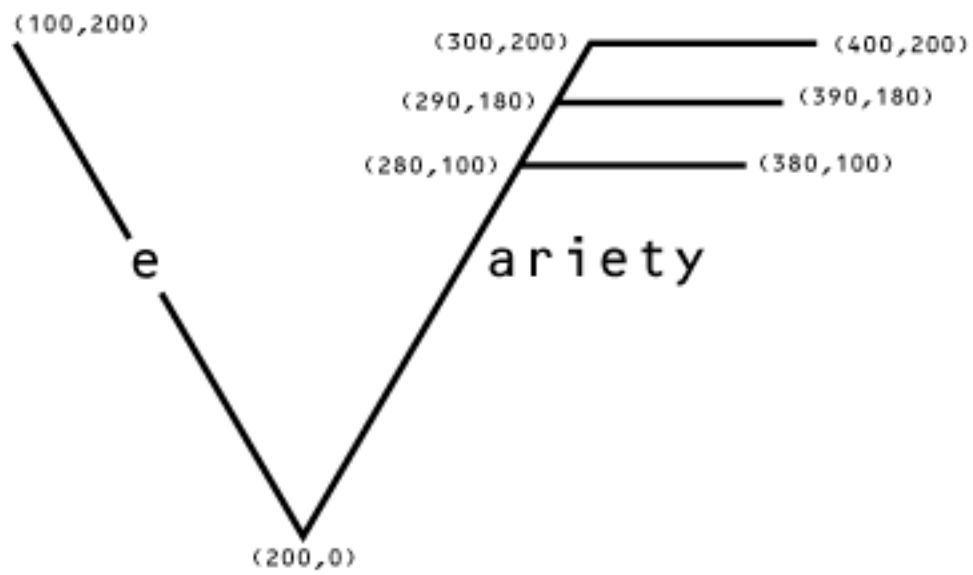


Figure 6-5
The e-Variety logo

Chapter 6, Perl 5 Interactive Course

LESSON 8

BINARY DATA

All the programs you've seen so far in this book create text output. Even the PostScript programs in the last lesson used text to produce graphics. But some file formats are binary, meaning that they contain eight-bit data not meant for reading by humans. Perl gives you two powerful functions for reading and writing binary data: `pack()` and `unpack()`. In this lesson, they'll be used to create a TIFF image that you can view with the graphics package of your choice.

Binary Data

`pack()` and `unpack()` allow you to manipulate binary strings, which are merely scalars that you access by individual bits, just like `vec()`.

The `pack()` and `unpack()` Functions

`pack(TEMPLATE, ARRAY)` turns the *ARRAY* of values (conforming to *TEMPLATE*) into a binary string.

`unpack(TEMPLATE, STRING)` turns the binary *STRING* into an array (conforming to *TEMPLATE*).

Before seeing how the *TEMPLATE* is defined, look at some examples of `pack()` and `unpack()`:

```
$bits = pack("c", 65);  
print $bits;
```

prints `A`, which is ASCII 65.

```
$bits = pack("x");  
print length($bits);  
print ord($bits);  
print $bits;
```

`$bits` is now a null character. This prints `1`, `0`, and nothing at all, in that order.

```
$bits = pack("sai", 255, "A", 47);
```

creates a seven-character string on most computers. The first two bytes comprise the short integer 255, the third byte is ASCII `A`, and the last four bytes are the integer 47.

```
$bits = pack ("sai", 255, "A", 47);  
@array = unpack("sai", $bits);
```

`@array` now contains three elements: 255, `A`, and 47.

```
$bits = unpack("B*", pack("N", $number));
```

converts decimal numbers to binary, and

```
$number = unpack("N", pack("B32", substr("0" x 32 . $bits,
-32)))
```

converts binary numbers to decimal (that's from the *perlfunc* online documentation).

The binary structures created by `pack()` (and parsed by `unpack()`) are useful as compact representations of data. Why would you want that? Well, maybe you're sending messages over the network and you need to minimize transmission time. Or maybe you need to write data in a particular binary format, such as the TIFF format described later in this lesson.

Table 6-2 shows the valid *TEMPLATE* fields. Each field corresponds to one item (i.e., one of the elements in the *ARRAY* to be `pack()`ed or one of the elements in the `unpack()`ed output array).

Table 6-2 `pack()` and `unpack()` characters

Character	<code>pack()</code> Value	<code>unpack()</code> Value
a	ASCII string, null padded	ASCII string
A	ASCII string, space padded	ASCII string, without trailing spaces and nulls
b	Bit string, low bit to high bit	Same as <code>pack()</code> value
B	Bit string, high bit to low bit	Same as <code>pack()</code> value
c	Signed character	Same as <code>pack()</code> value
C	Unsigned character	Same as <code>pack()</code> value
d	Double-precision floating-point number	Same as <code>pack()</code> value
f	Single-precision floating-point number	Same as <code>pack()</code> value
h	Hexadecimal string, low nybble first	Same as <code>pack()</code> value
H	Hexadecimal string, high nybble first	Same as <code>pack()</code> value
i	Signed integer	Same as <code>pack()</code> value
I	Unsigned integer	Same as <code>pack()</code> value
l	Signed long integer	Same as <code>pack()</code> value
L	Unsigned long integer	Same as <code>pack()</code> value
n	Short integer, in network ("big endian") order	Same as <code>pack()</code> value
N	Long integer, in network order	Same as <code>pack()</code> value
p	Pointer to a null-terminated string	Same as <code>pack()</code> value
P	Pointer to a fixed-length string (structure)	Same as <code>pack()</code> value
s	Signed short integer	Same as <code>pack()</code> value
S	Unsigned short integer	Same as <code>pack()</code> value
u	Uuencoded string	Uudecoded string
v	Short integer, in little endian order	Same as <code>pack()</code> value
V	Long integer, in little endian order	Same as <code>pack()</code> value
w	BER compressed integer	Same as <code>pack()</code> value

x	Null byte	Skip forward a byte
X	Back up a byte	Same as pack() value
@	Null fill to absolute position for next field	Go to absolute position in string for next field

B means 1 bit, and *BBBB* means 4 bits, which you can abbreviate as *B4*. If you mean "as many bits as possible," you can say *B**. Check the *perlfunc* documentation for more specific examples.

When unpacking, you can also prefix a field with *%NUMBER* to indicate that you want a *NUMBER*-bit checksum of the bits, rather than the bits themselves. For instance,

```
unpack("%1B*", $bits)
```

is 1 if there are an odd number of ones in *\$bits* and 0 if there are an even number of ones.

TIFF

TIFF (Tagged Image File Format) is a standard format for representing images.

Figure 6-6 shows the organization of a simple TIFF file, one that contains only a single image. (TIFF files can contain multiple images, but we'll keep things simple.)

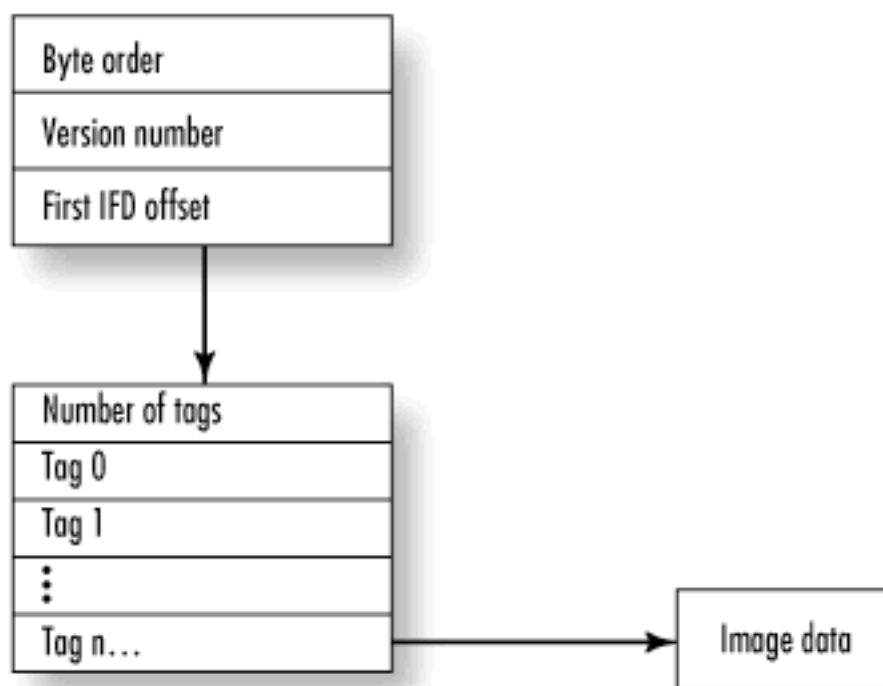


Figure 6-6

The TIFF file format (for a single-image file)

The *maketiff* program will use `pack()` to construct the TIFF data structures:

- **TiffHeader:** a 1-byte identifier, followed by a 1-byte version, followed by a 2-byte IFDOffset
- **TiffImageDirectory:** a 1-byte NumberOfEntries, followed by a TifTag structure, followed by a 2-byte NextImageDirectoryOffset
- **TifTag:** a 1-byte TagID, followed by a 1-byte DataType, followed by a 2-byte DataCount, followed by a 2-byte DataOffset

You don't need to know what all these fields mean to use *maketiff*. But for more information, consult a complete description of the TIFF standard, which can be found in Murray and vanRyper's *Encyclopedia of Graphics File Formats* (O'Reilly & Associates, 1994).

Without further ado, Listing 6-32 presents *maketiff*.

Listing 6-32 *maketiff*: Creating TIFF files with `pack()`

```
#!/usr/bin/perl
$height = 128;
$width = 128;
$bits_per_sample = 8;
$header_size = 8;
$data_size = $height * $width;
$num_tags_size = 2;
$num_tags = 12;
$taglist_size = 12 * $num_tags;
$xres_offset = $header_size = $data_size + $num_tags_size +
$taglist_size;
open (TIFF, ">face.tif") or die "Can't write face.tif: $! ";
generate_data();
print_header();
print_data();
print_num_tags();
tiff_print_taglist();
tiff_print_footer();
# The first few bytes of the TIFF file
sub print_header {
print TIFF pack("N", hex("4d4d002a"));
print TIFF pack("N", $height * $width + 8);
}
# Write the picture data to the TIFF file
sub print_data {
foreach $y (1..$height) {
foreach $x (1..$width) {
print TIFF pack("C", $picture[$x][$y]);
}
}
}
```

```

}
# ...followed by the tag information
sub tiff_print_tag {
my($id, $type, $count, $offset) = @_;
print TIFF pack("n", $id);
print TIFF pack("n", $type);
print TIFF pack("N", $count, 0);
if ($type == 3) {
print TIFF pack("nn", $offset, 0);
} elsif ($type == 4) {
print TIFF pack("N", $offset);
} elsif ($type == 5) {
print TIFF pack("N", $offset, 0);
}
}
sub print_num_tags {
print TIFF pack("n", hex("c"));
}
sub tiff_print_taglist {
tiff_print_tag(256, 3, 1, $width);
tiff_print_tag(257, 3, 1, $height);
tiff_print_tag(258, 3, 1, $bits_per_sample);
tiff_print_tag(259, 3, 1, 1);
tiff_print_tag(262, 3, 1, 0);
tiff_print_tag(273, 4, 1, 8);
tiff_print_tag(277, 3, 1, 1);
tiff_print_tag(279, 4, 1, $width * $height);
tiff_print_tag(282, 5, 1, $xres_offset);
tiff_print_tag(283, 5, 1, $xres_offset + 8);
tiff_print_tag(284, 3, 1, 2);
tiff_print_tag(296, 3, 1, 2);
}
# and finally, write the footer
sub tiff_print_footer {
print TIFF pack("N", 0); # tagnum
print TIFF pack("NN", hex("afc80"), hex("2710"));
print TIFF pack("NN", hex("afc80"), hex("2710"));
}
# generate_data uses make_circle() to create its circles
sub make_circle {
my ($xc, $yc, $radius) = @_;
my ($x, $y, $theta);
for ($theta = 0; $theta < 6.28; $theta += 1/$radius) {
$x = int($xc + $radius * cos($theta));
$y = int($yc + $radius * sin($theta));
$picture[$x][$y] = 255;
}
}

```

```

}
}
sub generate_data {
make_circle(64, 64, 50);
make_circle(50, 40, 10);
make_circle(52, 38, 1);
make_circle(78, 40, 10);
make_circle(76, 42, 1);
make_circle(64, 88, 4);
}

```

The `generate_data()` subroutine contains all the "content" for the picture: six circles. The data is stored in `%picture`, a two-dimensional hash (see Chapter 5, Lesson 6) containing grayscale values ranging from 0 (white) to 255 (black). Each call to `make_circle()` sets a "circle" of values in `%picture` to 255.

`print_data()` is the key subroutine: It cycles through all the pixels in `%picture` and prints them all to *face.tif*. All the other subroutines (`print_header()`, `print_num_tags()`, `tiff_print_taglist()`, and `tiff_print_footer()`) create the necessary baggage so that the picture conforms to the TIFF format.

When you run *maketiff*, it creates a *face.tif* file that looks like Figure 6-7.

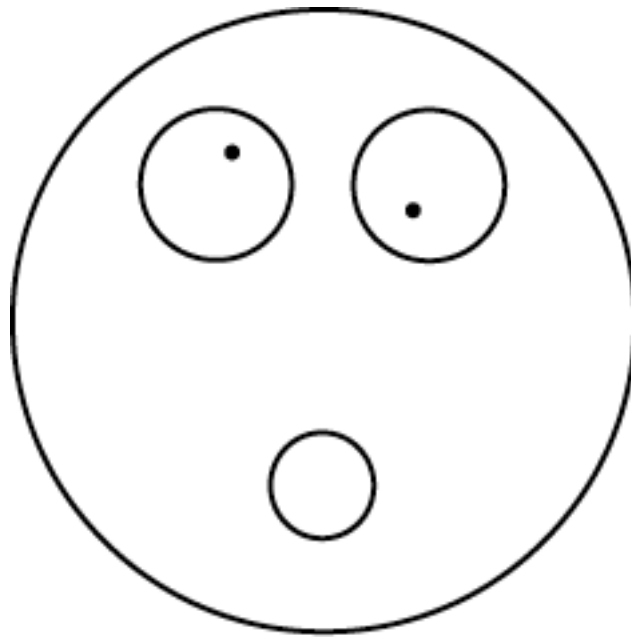


Figure 6-7

face.tif

That's a fitting expression for the end of the chapter.

Review

Jodie: I've played with TIFF images in PhotoShop before, but never PostScript.

Anthony: I've edited PostScript images before, but I've never created my own. I didn't realize they were just text files. And I've certainly never built up TIFF files byte by byte.

Jodie: I was looking at some other PostScript files I had lying around and I discover some other PostScript commands-*setlinewidth* and *curveto*-so I added them to *makeps*.

Anthony: Sounds helpful. I just wish there was a way to combine formats and PostScript-I'd love to be able to write multicolumn text to a PostScript file.

Chapter 7, Perl 5 Interactive Course

HOOPLA: OBJECT-ORIENTED PROGRAMMING

Nearly everything you've learned so far has been part of both Perl 4 and Perl 5. Sure, Perl 5 has a few more functions and operators. And lexical scoping is new, too. But Perl 5 also succumbs to that notorious juggernaut of 1990s computer science: *object-oriented programming*. This chapter covers all of it.

Object-oriented programming (called--unfortunately--OOP, resulting in insipid acronyms and too-cute puns, such as this chapter title) offers three advantages that other programming languages lack:

- **Encapsulation:** The properties (variables, subroutines, etc.) of a thing are contained inside it, so you can use it without knowing how it works.
- **Inheritance:** You can say, "This thing is kind of like that other thang, and uses (inherits) some of its properties."
- **Polymorphism:** A function for this thing can have the same name as a function for that thang.

OOP allows you to think about your programs in a new way: as an interaction between independent "objects" that maintain their own variables and subroutines. OOP also facilitates code *reusability*: the ability to extract objects from one program and include them in others.

LESSON 1

OOP AND PACKAGES

Before you wallow in the hype-laden mud pit of object-oriented programming, it helps to know what all the hoopla is about. This lesson begins with a bit of proselytizing. It's not essential knowledge for writing good Perl programs--that just requires common sense. After the rhetoric, we'll jump directly into packages, which are used to implement one of the basic concepts in OOP: classes.

Object-Oriented Programming

OOP was originally developed to help manage the complexity of programs through abstraction: building self-contained chunks of code that correspond to how you think about your program. Instead of having to think about your program as a succession of statements and loops, you can think about souped-up variables called *objects* that belong to glorified packages called *classes*.

This chapter will illustrate OOP concepts by explaining the three hallmarks of object-oriented programming: encapsulation, inheritance, and polymorphism.

Encapsulation

Encapsulation is separating the what from the how. Objects (and classes of objects) can shield programmers from the details of their implementation. You don't need to know how they work--you merely need to know what they do.

- **Objects encapsulate their own variables, called *instance variables*. The variable's location and storage are "hidden" inside the object.**
- **Classes encapsulate subroutines (called *methods*) that manipulate the objects belonging to that class. The subroutine's implementation is "hidden" inside the class.**

Inheritance

Inheritance lets you say, "This class is kind of like that class." Often, one class inherits methods or variables from another class. This facilitates code reusability: Because different classes use the same parts, you can eliminate redundancy in your programs. *Multiple inheritance* occurs when one class uses the tools from two or more classes. The most common form of inheritance is a superset-subset relationship, in which one class (the *subclass*, or *derived class*) is a more specific version of another (the *superclass*, or *base class*), for example, "All Westerns are Movies."

Polymorphism

Polymorphism just means using the same name to mean different things. The Movie class and the Western class each contain a subroutine named sequel(). The subroutines are different, but the name is the same. You can use polymorphism to create a uniform look and feel for your objects, hiding different behaviors behind the same name.

Until now, you've probably thought of variables and subroutines as completely different beasts. But you can think of a subroutine as a peculiar sort of variable with a name (the subroutine's name) and a value (the subroutine's code). (In Chapter 9, you'll use eval() to change that value while your program is running.) But the notion of subroutines as variables is a valuable one for OOP--in fact, C++ blurs the distinction by referring to both variables and subroutines as simply "members."

OOP is as famous for its jargon as for its usefulness. There's a lot of new terminology to learn in this lesson and throughout this chapter. Table 7-1 shows the four most important terms, their meanings independent of the programming language, and their implementations in Perl.

Table 7-1 OOP terms, their "formal" meanings, and their Perl implementations

Buzzword	Formal OOP Meaning	What It Really Means, in Perl
Reference	Something that "points" to a piece of data	The name (or address) of a variable (as opposed to the variable itself)
Object	A self-contained entity with its own data and routines	A reference that knows to which package it belongs
Method	A function particular to a class	A subroutine operating on either an object or a class
Class	A user-defined data type	A package containing methods for manipulating objects

Packages

Perl's *packages* are used to define object-oriented classes.

Every variable and subroutine you create belongs to a package. In Chapter 5, Lesson 7, you saw that the default package is *main*, which is why

```
$box
```

is the same as

```
$main::box
```

You can create your own packages with the package declaration.

The package Declaration

package *NAME* enters the *NAME* package.

After a package declaration, all variable and subroutine definitions are stored in the symbol table associated with that package. Put another way: The variables and subroutines will belong to the *NAME* package.

Following a

```
package Movie;
```

all variables and subroutines you create will belong to the `Movie` package. So a subsequent

```
$plant = "rhododendron";
```

defines `$Movie::plant`, not `$main::plant`. If you now switch back to the *main* package and print the value of `$plant`:

```
package main;  
print $plant;
```

it'll be undefined, because `$main::plant` and `$Movie::plant` are completely different variables. To access `Movie`'s `$plant` from *main*, you need to tell Perl where to find it:

```
print $Movie::plant;
```

This is why you can't create global variables in Perl--variables are valid only inside a single package. (Well, there are a few true global variables, such as `%ENV`, `%SIG`, and `@ARGV`. But all user-defined variables have only a packagewide scope.)

Figure 7-1 illustrates two packages: the `Movie` package and the `Book` package. The *media* program (Listing 7-1) declares those two packages and defines `$description` and `describe()` in each, as well as in *main*.

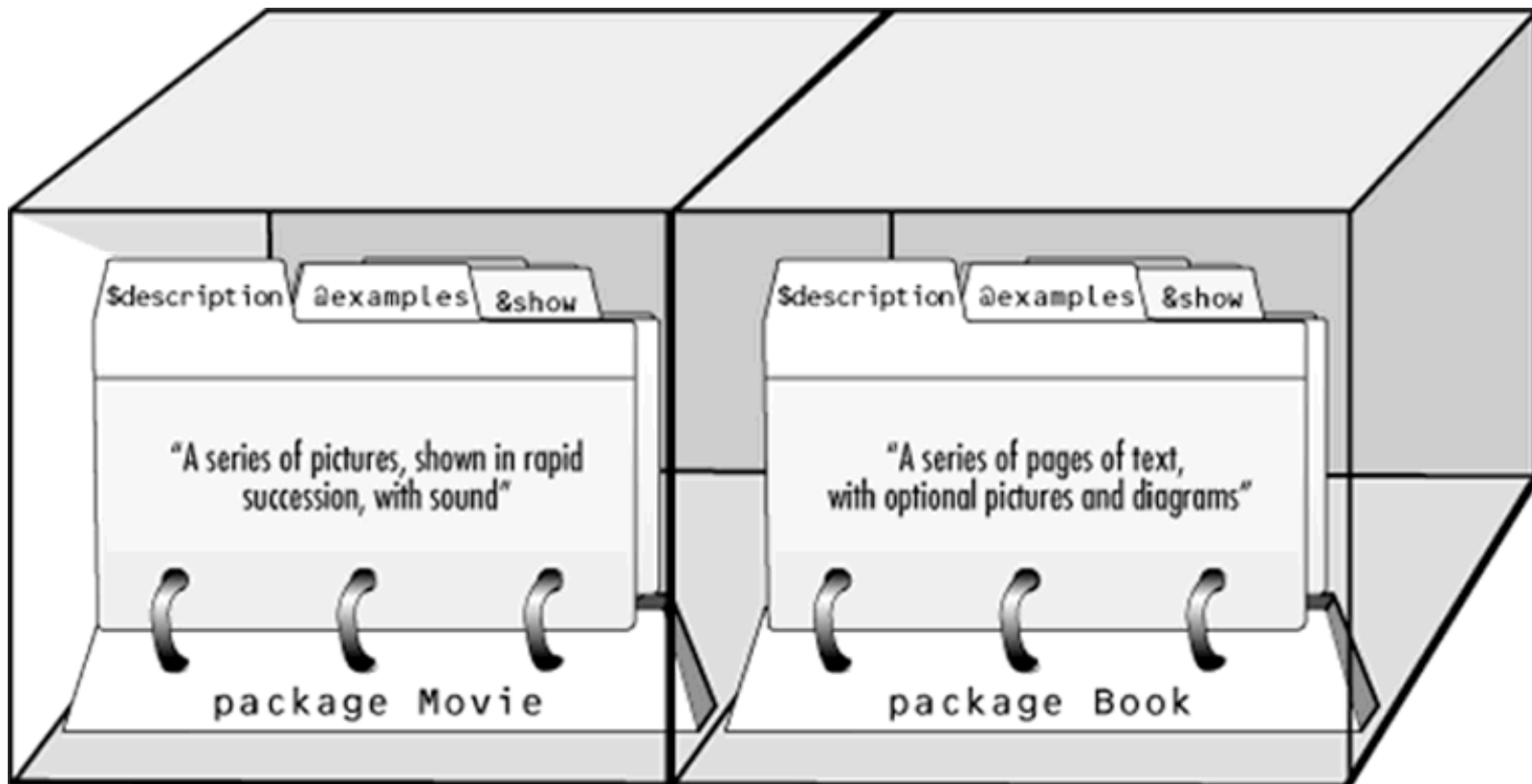


Figure 7-1

Packages provide alternate namespaces

Listing 7-1 *media*: Using a package to switch between packages

```
#!/usr/bin/perl -wl  
# Initially, we're in the main package, as usual.  
$description = "Huh?";  
sub describe { print $description }  
# Leave the main package and enter the Movie package,
```

```

# defining the scalar $Movie::description and the subroutine
# &Movie::describe.
package Movie;
$description = `A series of pictures shown in rapid succession, with sound.`;
sub describe { print $description }
# Leave the Movie package and enter the Book package,
# defining the scalar $Book::description and the subroutine
# Book::describe().
package Book;
$description = `A series of pages of text, with optional pictures.`;
sub describe { print $description }
package main; # Return to the good old main package
print $description; # print main's $description
print $Movie::description; # print Movie's $description
Book::describe(); # invoke Book's describe()

```

It might seem as though *media* is awfully redundant: It defines three \$descriptions and three describe()s. But because each variable and subroutine belongs to its package, there are actually three different scalars and three different subroutines:

```

% media
Huh?
A series of pictures shown in rapid succession, with sound.
A series of pages of text, with optional pictures.

```

As usual, this program begins in the *main* package, where \$description (or, to be precise, \$main::description) is set to *Huh?*. Then the Movie package is entered, and Movie's \$description (or, to be precise, \$Movie::description) is set to a different value. Next, the Book package is entered (and therefore the Movie package is exited) and \$Book::description is set to yet another value. Finally, the program returns to the *main* package: *main*'s \$description and Movie's \$description are printed, and Book's describe() is invoked.

In Lesson 3, packages will be used to define OOP classes. This ability of packages to shroud variables and subroutines from one another will come in handy--that's what makes encapsulation possible.

Chapter 7, Perl 5 Interactive Course

LESSON 2

REFERENCES

A *reference* is a scalar that holds a peculiar kind of value. Instead of storing a number or string, references store the address of another variable. They're similar to pointers in C, but Perl programs typically don't depend on references nearly as much as C programs depend on pointers.

Creating and Accessing References

References and Backslashes

A *reference* is a scalar that points to a value.

`\VAR` is a reference to `VAR`.

Here's a reference that points to a scalar value:

```
$scalarref = \7;
```

`$scalarref` isn't 7, but it points to a scalar whose value is 7.

You can also create a reference that points to an array:

```
@array = (2,3,4);  
$arrayref = \@array;
```

`$arrayref` is still a scalar, but it's also a reference to an array.

Figure 7-2 shows two variables: `$movieref` and `$movie`. `$movieref` is a reference that points, or refers, to the scalar `$movie`. Using the Rolodex analogy, it "knows" which card contains `$movie`.

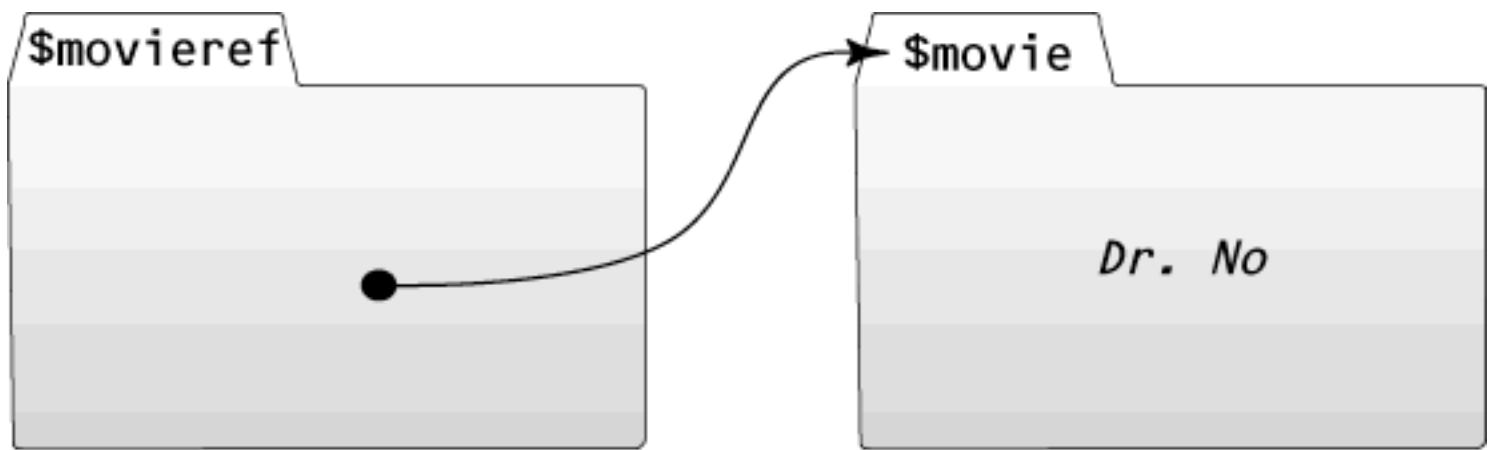


Figure 7-2

References

refintro, shown in Listing 7-2, demonstrates creating a reference with `\`, and accessing the value of the referred-to variable.

Listing 7-2 *refintro*: Creating and accessing a reference

```
#!/usr/bin/perl -wl
$movie = `Dr. No`;
$movieref = \$movie; # Now $movieref is a reference to $movie
print ` $movieref is `, $movieref; # prints the memory address
of $movie
print ` $$movieref is `, $$movieref; # prints the contents of
$movie
```

The line `$movieref = \$movie` creates the reference `$movieref`, which points to `$movie`.

Then, in the last line, `$movieref` is dereferenced, which means that the value of the referred-to variable `$movie` is accessed. `$$movieref` is equivalent to `$movie`, which contains the value `Dr. No`:

```
% refintro
$movieref is SCALAR(0x575b8)
$$movieref is Dr. No
```

Because `$movieref` is just another scalar (that also happens to be a reference), you can print its value. The

result, `SCALAR(0x575b8)`, tells you that `$movieref` is a reference to a scalar variable currently residing at memory address 575b8. (That address will vary from computer to computer and from program to program.)

In a sense, `\` and `$` are opposites. `\` means "name of" or "address of" and creates an extra "layer": It yields a reference to a value rather than the value itself. `$` (as well as `@`, `%`, and `&`) means "contents of" and removes a layer by dereferencing a name. You can add and remove layers freely:

```
# A scalar.  
$movie = 'Dr. No';  
print $movie;
```

prints *Dr. No.*

```
# A reference to a scalar.
$movieref = \$movie;
print $$movieref;
```

prints *Dr. No.*

```
# A reference to a reference to a scalar.
$movierefref = \\$movie;
print $$$movierefref;
```

prints *Dr. No.*

```
# A reference to a scalar that's a reference to a scalar.
$movierefref = \$movieref;
print $$$movierefref;
```

prints *Dr. No.*

```
# A reference to a reference to a reference to a scalar.
$movierefrefref = \\$movie;
print $$$$movierefrefref;
```

prints Dr. No.

```
# A reference to a reference to a reference to a ref...oh  
never mind.  
$movierefrefrefrefrefrefrefrefrefrefref =  
\\\\\\\\\\\\\\\\\\\\$movie;  
print $$$$$$$$$$$$movie$refrefrefrefrefrefrefrefrefref;
```

prints *Dr. No.*

References don't have to refer to scalars. They can refer to any data type. *referee*, shown in Listing 7-3, creates references to a scalar, an array, a hash, and even a subroutine (remember, you can think of subroutines as just another type of variable).

Listing 7-3 *referee*: Creating references to scalars, arrays, hashes, and subroutines

```
#!/usr/bin/perl -wl
$scalar = 1;
```



```

$array = (2, 3, 4);
%hash = (key => "value");
sub routine { $_[0] + $_[1] }
$scalarref = \$scalar;
$arrayref = \@array;
$hashref = \%hash;
$subref = \&routine;
print $scalarref;
print $arrayref;
print $hashref;
print $subref;
print ``;
print $$scalarref;
print $$arrayref[1];
print $$hashref{key};
print &$subref(8, 9);

```

referee creates four references and then dereferences all of them. First \$scalarref is dereferenced, yielding the value of \$scalar. Then \$arrayref is dereferenced, with index 1: \$\$arrayref[1] is equivalent to \$array[1]. Likewise, \$\$hashref{key} is equivalent to \$hash{key}. Finally, &\$subref(8, 9) is equivalent to sub(8, 9). The results are shown here:

```

% referee
SCALAR(0x575e8)
ARRAY(0x57624)
HASH(0x57684)
CODE(0x57720)
1
3
value
17

```

A Word About Syntax

Some people mistakenly read \$\$arrayref[1] as

"the contents of the second element of the array @arrayref"

instead of what it actually means:

"the second element of the contents of the reference \$arrayref."

You can use curly braces to disambiguate dereferences:

```

$array = (2, 3, 4);
$arrayref = \@array;
print $$arrayref[1];

```

prints 3.

```

print ${$arrayref}[1];

```

also prints 3.

Because this notation is a little cumbersome, you can use the `->` shorthand, which should be familiar to C programmers.

The `->` Notation

REF->**INDEX** is shorthand for "get the **INDEX**th element of the thing referred to by **REF**."

Don't confuse `->` with `=>`, which acts like the comma operator.

All of the following are equivalent:

```
print $$arrayref[1]; # this prints 3
print ${$arrayref}[1]; # this also prints 3
print $arrayref->[1]; # this prints 3 too
```

Figure 7-3 shows how the `->` notation can be used to access the contents of an array or a hash through a reference.

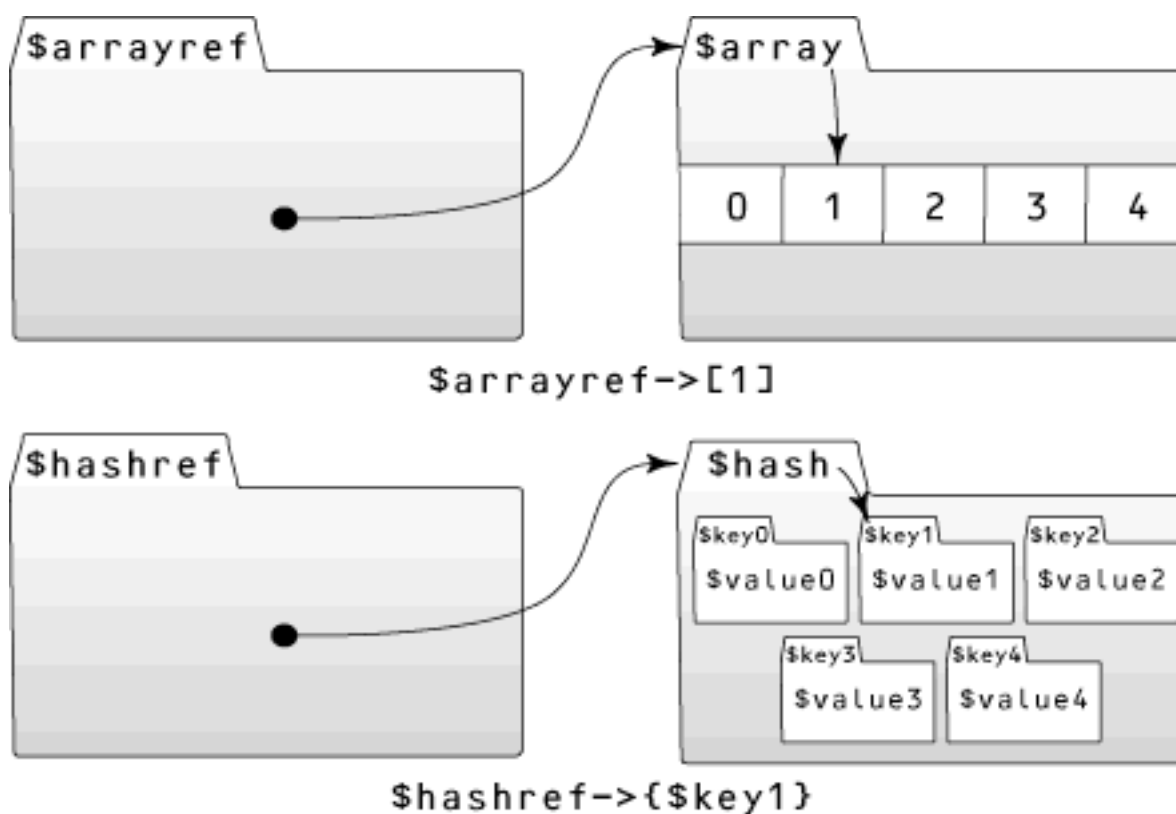


Figure 7-3

The `->` notation

The ref() Function

Programs making heavy use of references can occasionally be tough to understand: When many layers of referencing are used, it'll sometimes be unclear whether you're using the right number and type of dereferencing symbols.

You can find out what a reference refers to with `ref()`.

The `ref()` Function

`ref(REFERENCE)` returns the type of the variable referred to by *REFERENCE*.

`ref()` returns `undef` if *REFERENCE* isn't actually a reference. Typical values returned by `ref()` are `SCALAR`, `ARRAY`, `HASH`, `CODE`, `GLOB`, and `REF` (for a reference to a reference).

So,

```
print ref($arrayref);
```

prints `ARRAY`.

Listing 7-4 is an example that shows all six possible return values of `ref()`.

Listing 7-4 *referee2*: The `ref()` function returns the type of an array

```
#!/usr/bin/perl -wl
$scalar = 1;
@array = (2, 3, 4);
%hash = (key => "value");
sub sub { $_[0] + $_[1] }
$ref = \$scalar;
$scalarref = \$scalar;
$arrayref = \@array;
$hashref = \%hash;
$subref = \&sub;
$globref = \*GLOB;
$refref = \$ref;
print ref($scalarref);
print ref($arrayref);
print ref($hashref);
print ref($subref);
print ref($globref);
print ref($refref);
```

referee2 creates six references, including a reference to a symbol table entry (`$glob-ref`) and a reference to a reference (`$refref`).

```
% referee2
SCALAR
ARRAY
```

HASH
CODE
GLOB
REF

Copying References Versus Copying Values

Remember that a reference is one step removed from a value. Copying a reference is NOT the same as copying the value it points to.

Suppose that `$aref` is a reference to the scalar variable `$a` and `$bref` is a reference to the scalar variable `$b`. Consider the difference between

```
$b = $a;
```

which copies the value of `$a` to `$b`, and

```
$bref = $aref;
```

which copies the value of `$aref`--that is, the address of `$a`--to `$bref`.

`$a = $b` copies the value of `$a` to `$b`, after which `$a` and `$b` are completely unrelated: If you modify one, the other won't notice. But `$bref = $aref` copies the reference, so `$$bref` and `$$aref` are identical even though `$bref` and `$aref` are different.

If that seems confusing, study Listing 7-5, which first copies a scalar and then copies a reference to a scalar.

Listing 7-5 *refcopy*: Copying values versus copying references

```
#!/usr/bin/perl -wl  
$a = 8;  
$b = $a;  
$b++; # This doesn't change $a...  
print $a; # 8  
print $b; # 9  
$a = 77;  
$aref = \ $a;  
$bref = $aref;  
$$bref++; # ...but this changes $$aref  
print $$aref; # 78  
print $$bref; # 78
```

refcopy demonstrates the difference between copying a scalar value and copying a reference to a scalar value. Changing `$b` doesn't affect `$a`, but changing `$$bref` does affect `$$aref` because both refer to the same variable.

```
% refcopy  
8  
9  
78
```

Symbolic References Versus Hard References

It's often handy to think of a reference as containing the name of another variable. But when you print the value of a reference, you see something like *"SCALAR (0x575b8)"*. That's because the references you've seen are *hard* references--they don't really contain names, they contain memory addresses.

A *symbolic* reference is a reference that really does contain a name. If you say

```
$name = "movie";
$$name = "Unforgiven";
```

then the scalar \$movie is created automatically and set to the string *Unforgiven*.

Symbolic references can pop up when you least expect them. Quick--what does this statement do?

```
$$enigma = "riddle";
```

If you answered, "It sets the scalar pointed to by \$enigma to riddle," you might be right. And if you answered, "It sets the scalar named by \$enigma to riddle," you might be right. It all depends on whether that statement followed something like this:

```
$enigma = \ $scalar;
```

which sets \$scalar to *riddle*, or like this:

```
$enigma = "puzzle";
```

which sets \$puzzle to *riddle*.

Does that scare you? It should. That's why you can outlaw symbolic references by saying **use strict 'refs'** at the beginning of your program. (The *strict* pragma is described in Chapter 8, Lesson 1.)

Chapter 7, Perl 5 Interactive Course

LESSON 3

ANONYMOUS VARIABLES AND OBJECTS

There's one last concept you need to learn before tackling objects: *anonymous variables*. Objects use anonymous variables to store instance variables: the individual snippets of data owned by a particular object.

Anonymous Variables (and Anonymous Functions)

An anonymous variable is nothing more than a variable without a name, allowing you to create a reference without actually referring to a named variable. Instead of saying

```
@movies = ('Unforgiven', 'A Fistful of Dollars');  
$arrayref = \@movies;
```

you can combine the two statements into one, using square brackets:

```
$arrayref = ['Unforgiven', 'A Fistful of Dollars'];
```

\$arrayref is now a reference to an anonymous array. There's no @movies required--that's what makes it anonymous. \$arrayref->[1] is *A Fistful of Dollars* and ref(\$arrayref) is ARRAY, as you'd expect.

Hashes and subroutines can be anonymous, too. You can create references to

An anonymous array	with brackets	e.g., ["physics", "chemistry", "biology"]
An anonymous hash	with braces	e.g., {letter => 'y', number => 3}
An anonymous subroutine with a sub	e.g., sub { \$_[0]. 's' }	without a name

So,

```
$hashref = {name => "Connie", age => 32};
```

creates a reference to an anonymous hash, and

```
$hashref->{name}
```

is Connie.

Anonymous references to subroutines work, too:

```
$subref = sub { return $_[0] * $_[1] };
```

They can be invoked with &:

```
&$subref(6, 8)
```

is 48.

You've already seen one good use of anonymous variables: to create multidimensional arrays. Here's the

two-dimensional array `@matrix` from Chapter 5, Lesson 6:

```
@matrix = ([ 8, 75, 102 ],
[ 75, 0, 999 ],
[ 12, 200, 1 ]);
```

`@matrix` is an array consisting of three elements; each element is a reference to an anonymous array. So the third element of the second row is

```
$matrix[1]->[2]
```

which is 999.

Omitting ->

You might recall from Chapter 5 that you can access multidimensional array elements without using `->`:

```
$matrix[1][2]
```

That's because Perl lets you omit the `->` between pairs of braces and brackets. The following are all equivalent:

```
$cube[2]->[0]->[1]
$cube[2][0][1]
${${$cube[2]}[0]}[1]
```

as are these:

```
$structure{key}[$index]
$structure{key}->[$index]
${structure{key}}[$index]
```

Anonymous Subroutines and Closures

As you've seen, anonymous subroutines can be invoked like normal subroutines:

```
$subref = sub { return substr($_[0], 0, -3) . substr($_[1],
-3) };
&$subref("posture", "amateur");
```

returns *poseur*.

You can even have a subroutine that returns a reference to an anonymous subroutine, as shown in Listing 7-6.

Listing 7-6 *closure*: Anonymous subroutines

```
#!/usr/bin/perl -wl
sub make_sub {
my $multiplicand = shift;
return sub { print $multiplicand * shift }
}
$multiply_by_four = make_sub(4);
$multiply_by_five = make_sub(5);
```

```
&$multiply_by_four(10);  
&$multiply_by_five(10);
```

The subroutines returned by `make_sub()` are curiously half-baked: They're able to see `$multiplicand`, even though it's a `my()` variable that is visible inside `make_sub()`. Or so you would think. But the anonymous subroutines can see both `$multiplicand` and the as-yet-unseen variable that is accessed via the second shift:

```
% closure  
40  
50
```

The subroutines created by `make_sub()` are called closures. They're pretty tricky, but occasionally useful.

Creating Objects

By now, you're probably wondering what references have to do with object-oriented programming. Recall the OOP jargon chart (Table 7-1).

Objects and Classes

An *object* is a reference that knows which package it belongs to.

A *class* is a package that provides subroutines (methods) for manipulating objects.

So how do you turn a reference into an object? You `bless()` it into a package.

The `bless()` Function

`bless(REFERENCE, PACKAGE)` tells the value referenced by *REFERENCE* that it's a member of *PACKAGE*, making it an object.

`bless()` returns the *REFERENCE*, which now refers to an object instead of merely a value. If *PACKAGE* is omitted, the current package is used:

```
bless $ref;
```

turns `$ref` into an object in the current package, and

```
bless $ref, Movie;
```

turns `$ref` into a `Movie` (i.e., it turns `$ref` into an object in the `Movie` package).

The `bless()` function takes a reference and blesses its variable (oops, object) into the `Movie` package (oops, class).

How do you know when you've made an object? `ref()` tells you so, as demonstrated by Listing 7-7.

Listing 7-7 *movie*: Assigning the object `$film` (a reference) to the class `Movie` (a package)

```
#!/usr/bin/perl -w  
$film = {}; # an anonymous and empty hash  
bless ($film, "Movie"); # ...which is now an object.
```



```
print `film is a `, ref($film), "\n";
```

movie creates a reference to an anonymous hash called *\$film*, blesses the value of *\$film* into the *Movie* class (which hasn't been declared, but that's okay), and then identifies which class *Movie* belongs to, using *ref()*.

```
% movie
film is a Movie
```

The *\$film* object is an anonymous hash. That's where many objects store instance variables--variables contained within the object. In *movie2* (Listing 7-8), each instance variable is a key-value pair in that hash.

Listing 7-8 *movie2*: Storing variables in an object

```
#!/usr/bin/perl -wl
package Movie; # the Movie class
$film = {title => "Unforgiven"}; # an anonymous hash...
bless ($film); # ...which is now an object.
$cine = {title => "Eraserhead"};
bless ($cine);
print "$film->{title} is a ", ref($film);
print "$cine->{title} is a ", ref($cine);
```

movie2 creates two anonymous hash references, *\$film* and *\$cine*, each of which contains a single key-value pair that associates *title* with a movie title. Then each reference is blessed into the *Movie* class, after which the *=>* notation is used to retrieve the movie titles:

```
% movie2
Unforgiven is a Movie.
Eraserhead is a Movie.
```

In the next lesson, you'll learn how to make your classes more useful by defining methods (subroutines) that can manipulate objects.

Chapter 7, Perl 5 Interactive Course

LESSON 4

METHODS

Methods are subroutines that belong to a class. You'll use them for nearly everything you do with objects: to create and destroy them, to fetch and store their instance variables, and to perform whatever computation they require.

Methods

A *method* is a subroutine that belongs to a class.

All methods expect either an object or the name of a class as their first argument.

Classes often have a method called `new()` that creates objects, as shown in Listing 7-9.

Listing 7-9 *movie3*: Using the `new()` method to create an object

```
#!/usr/bin/perl -wl
package Movie;
sub new {
my $class = shift;
my $self = {};
bless $self;
}
package main;
$film = new Movie;
print 'film is a ', ref($film);
```

movie3 defines a `new()` method, which creates a reference to an anonymous hash and blesses it into the current package, returning the resulting object.

Then `new()` is called with one argument: `Movie`.

```
% movie3
film is a Movie
```

The `new()` method could also have been invoked with the `->` notation:

```
$film = Movie->new;
```

is the same as

```
$film = new Movie;
```

But those aren't quite the same as



```
$film = Movie::new;
```

which is a function call, not a method call. Don't confuse them! Only method calls support inheritance, the subject of the next lesson.

In most OOP languages, methods like `new()` are called *constructors* and have a special syntax. In Perl, however, everything is much more straightforward: The `new()` method is just like any other subroutine. You can call it `initialize()`, or `create()`, or whatever you wish.

Static Versus Virtual Methods

Every method is either static or virtual. *Static methods* expect the name of a class as their first argument; *virtual methods* expect an object as their first argument.

The `new()` method in *movie3* is static because it uses a class name as its first argument: `new Movie`. Virtual methods often `shift()` their first argument--an object--into a local variable:

```
sub virtual_method {  
    my $self = shift;  
    ...  
}
```

Listing 7-10 adds two virtual methods to the `Movie` class.

Listing 7-10 *movie4*: Two virtual methods: `set_name()` and `get_name()`

```
#!/usr/bin/perl  
package Movie;  
sub new { # A static method  
    my $class = shift;  
    my $self = {};  
    bless $self;  
}  
sub set_title { # A virtual method  
    my $self = shift;  
    $self->{title} = shift;  
}  
sub get_title { # Another virtual method  
    my $self = shift;  
    return $self->{title};  
}  
package main;  
$film = new Movie;  
$film->set_title("Unforgiven");  
print "The title of $film is ", $film->get_title, "\n";
```

The `set_title()` and `get_title()` methods both shift the object into `$self`. The `set_title()` method sets the title key in the hash pointed to by `$self`, and the `get_title()` accesses that value and returns it:

```
% movie4
The title of $film is Unforgiven
```

The `set_title()` and `get_title()` methods aren't very complicated. You could just as easily have ignored them entirely: After creating `$film`, you could have said `$film->{title} = "Unforgiven"`. Don't. One of the tenets of OOP is that class data should be accessed with class methods *only*. As you can see, none of the statements below **package main**; acknowledges that the title is stored in a hash. That's the way it should be.

Suppose you were to document your `Movie` class. It's better to be able to say

```
"The title of a Movie can be accessed with the get_title() method;"
```

than

```
"The title of a Movie can be accessed by saying object->{title};"
```

because tomorrow you might want to change how `set_title()` and `get_title()` operate. Users of `get_title()` won't notice the change; users of `object->{title}` will get burned.

The `get_title()` method is an abstraction barrier: It encapsulates the variable inside the object so programmers can still manipulate the variable even after they've forgotten how it's stored. The examples in this chapter use separate methods for storing and fetching object data. But `set_title()` and `get_title()` could have been combined into a single method that performs both operations:

```
sub title {
my $self = shift;
$self->{title} = $_[0] if @_; # If called with arg, act like
get_title()
return $self->{title};
}
```

Passing Arguments to Methods

If you construct your `new()` properly, you can supply instance variables to an object when you create it, as shown in Listing 7-11.

Listing 7-11 *movie5*: Passing values to a constructor

```
#!/usr/bin/perl -wl
package Movie;
sub new {
my $class = shift; # get 'Movie' out of the way...
my $self = {};
%$self = @_; # Set the hash to the rest of the input array
bless $self;
}
```

```

sub display {
my $self = shift; # the object
print ref($self), `:`;
while (($key, $value) = each %$self) {
print "\t $key: $value";
}
}

package main;
$film = new Movie(title => 'Unforgiven',
year => 1992,
director => 'Eastwood');
display $film;

```

Because `new()` is a static method, its first argument is `Movie`; the `shift` gets rid of that. The rest of `@_` contains the three key-value pairs within the parentheses (it's just like passing a hash into a subroutine) so the third line of the `new()` method stores all those values in the anonymous hash pointed to by `$self`.

The `display()` method then prints all the key-value pairs in the anonymous hash. Here it is in action:

```

% movie5
Movie:
title = Unforgiven
year = 1992
director = Eastwood

```

`movie5` passes extra arguments to `new()`, a static method. You can do the same with virtual methods, too: `movie6` (Listing 7-12) provides a `display()` method that lets you select which fields to print.

Listing 7-12 *movie6*: Passing values to a virtual method

```

#!/usr/bin/perl -wl
package Movie;
sub new {
my $class = shift;
my $self = {};
%$self = @_;
bless $self;
}
sub display {
my $self = shift;
my @keys = @_; # ( `year`, `director` )
print ref($self), `:`;
foreach $key (@keys) {
print "\t $key: $self->{$key}";
}
}
package main;
$film = new Movie(title => `Unforgiven`,

```

```
year => 1992,  
director => 'Eastwood');  
$film->display('year', 'director');
```

movie6 demonstrates passing arguments to a virtual method: `display()`. Inside `display()`, all the arguments (after the object itself) are stored in `@keys` and then used to access values in the anonymous hash of `$film`:

```
% movie6  
Movie:  
year: 1992  
director: Eastwood
```

Chapter 7, Perl 5 Interactive Course

LESSON 5

INHERITANCE

In every object-oriented language, classes can use, or *inherit*, methods from other classes. The most common style of inheritance is when one class (say, the class of Western movies) is a more specific type of another (all Movies). Inheritance lets the Western class (the derived class, or subclass) use some of the methods common to all Movies (the base class, or superclass).

Figure 7-4 shows an inheritance tree: At the root is the Entertainment class, which doesn't inherit from anything--it's a base class. Entertainment has three derived classes: Book, Movie, and Music. Each of these derived classes happens to have two derived classes of its own; for instance, Movie has Western and Comedy.

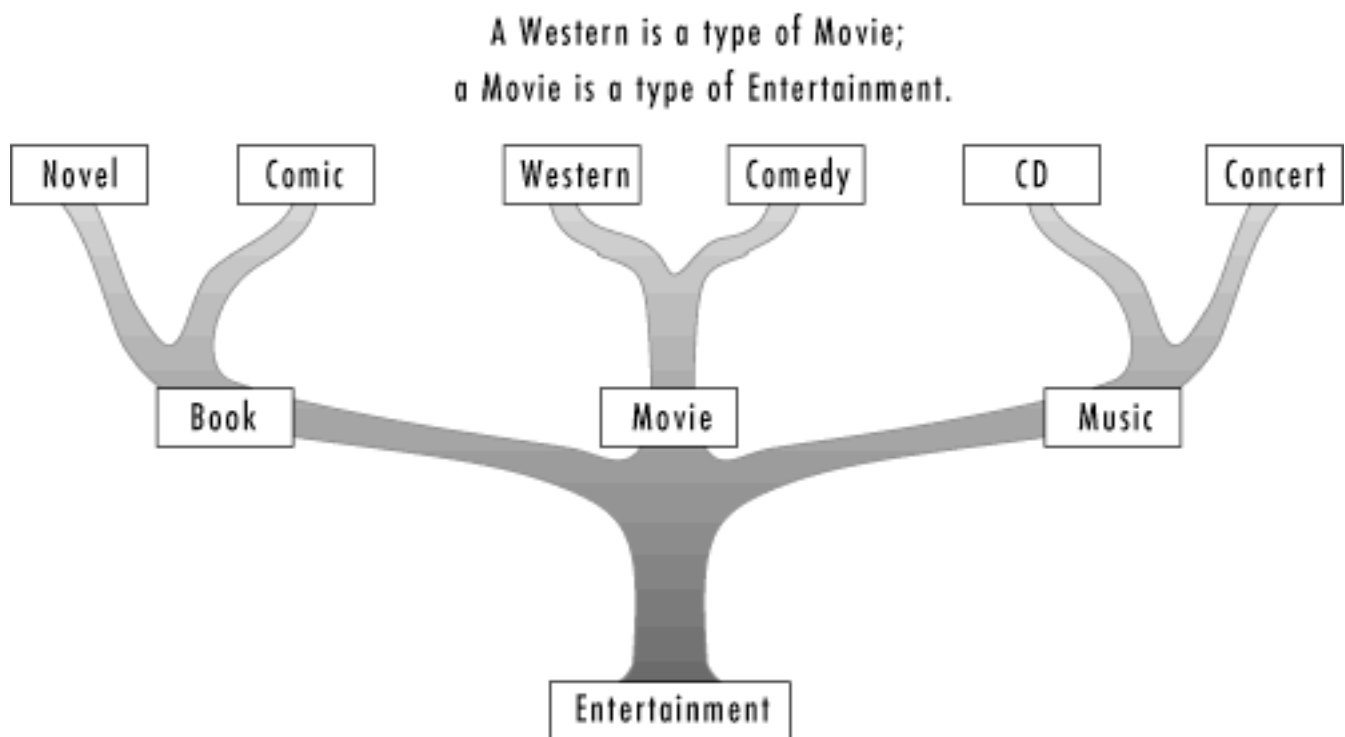


Figure 7-4

Inheritance

The Western class inherits some methods from its base class: Movie. In turn, Movie inherits from its base class: Entertainment.

Choosing a Base Class

You can set the parent of a class with the @ISA array, as in "A Western IS A Movie."

The @ISA array

A *class* inherits from the classes listed in its @ISA array.

A class doesn't have to have an @ISA array. But if it does, that's where Perl looks for any methods you invoke that don't appear in the class.

movie7, shown in Listing 7-13, declares two classes: Western and Movie. Inside the Western class, the @ISA array is set to (Movie), which makes Western a subclass of Movie.

Listing 7-13 *movie7*: Inheritance: A Western IS A Movie

```
#!/usr/bin/perl -w
package Movie;
sub set_title {
my $self = shift;
$self->{title} = shift;
}
sub get_title {
my $self = shift;
return $self->{title};
}
package Western; @ISA = qw(Movie);
sub new {
my $class = shift;
my $self = {};
bless $self;
}
# Westerns have victims, but Movies (in general) do not.
sub set_victims {
my $self = shift;
$self->{victims} = shift;
}
sub get_victims {
my $self = shift;
return $self->{victims};
}
package main;
$film = new Western; # Create a new Western object
$film->set_title("Unforgiven"); # Use a Movie method
$film->set_victims(11); # Use a Western method
print $film->get_title, " is a ", ref($film), ".\n";
print $film->get_victims, " people were killed in ",
```



```
$film->get_title, ".\n";
```

When the `Western` class is declared, its `@ISA` array is set to `(Movie)`, giving `Westerns` access to `Movie` methods. When you call `$film->set_title`, Perl first looks in the `Western` class for the method. Because it's not there, Perl looks in the `Movie` class and finds `set_title()` there.

`set_title()` and `set_victims()` are called in the same way, even though `set_title()` is inherited and `set_victims()` isn't. You don't have to specify where to find the methods; Perl finds them for you.

```
% movie7
Unforgiven is a Western.
11 people were killed in Unforgiven.
```

Inheritance: Where Does It End?

If an invoked method can't be found, Perl uses the `AUTOLOAD()` subroutine (described in Chapter 4, Lesson 6), if one exists. If not, Perl makes a last-ditch attempt to find the method in a class named `UNIVERSAL`. If that doesn't exist, you're out of luck.

The *movie8* program (Listing 7-14) declares four packages: `UNIVERSAL`, `Entertainment`, `Movie`, and `Western`. As you can see from the `@ISA` arrays, `Western` inherits from `Movie` and `Movie` inherits from `Entertainment`. And even though `UNIVERSAL` is not declared explicitly, all classes inherit from it.

Listing 7-14 *movie8*: Invoking methods from base classes and the `UNIVERSAL` class

```
#!/usr/bin/perl -l
package UNIVERSAL;
sub get_class {
my $self = shift;
return ref($self);
}
package Entertainment;
sub set_title {
my $self = shift;
$self->{title} = shift;
}
sub get_title {
my $self = shift;
return "Entertainment---$self->{title}";
}
sub advertise {
print $self->get_title, ": fun for the whole family!";
}
package Movie; @ISA = qw(Entertainment);
sub set_duration {
my $self = shift;
$self->{duration} = shift;
```

```

}
sub get_duration {
my $self = shift;
return $self->{duration};
}
sub get_title {
my $self = shift;
return "The movie $self->{title}";
}
sub advertise {
my $self = shift;
print $self->get_title, ": now playing at a theater near
you.";
}
package Western; @ISA = qw(Movie);
sub new {
my $class = shift;
my $self = {};
%$self = @_;
bless $self;
}
sub set_duration {
my $self = shift;
$self->{duration} = shift;
}
sub get_victims {
my $self = shift;
return $self->{victims};
}
package main;
$film = new Western(victims => 11); # instead of calling
set_victims().
$film->set_duration(90);
$film->set_title("Unforgiven");
print $film->get_victims(); # A Western method
print $film->get_duration(); # A Movie method
print $film->get_title(); # An Entertainment method
print $film->get_class(); # A UNIVERSAL method
$film->advertise(); # A Movie method

```

movie8 creates a `$film`, a `Western` object, initializing `victims` to 11. (That's a bit of an abstraction violation; it would have been more proper to use a `set_victims()` method instead.) Then `set_duration()` is called. That doesn't exist in the `Western` class, so Perl grabs it from `Movie`.

Then `set_title()` is called. Now both `Western` and `Entertainment` contain a `set_title()` method. Which one is used? `Western::set_title()`, because inheritance kicks in only when necessary.

Next, `get_class()` is called. That doesn't exist in any of the "normal" classes, but it does exist in `UNIVERSAL`.

Finally, `advertise()` is called. It doesn't exist in `Western`, but it does exist in both `Movie` and `Entertainment`. Which one is used? `Movie::advertise()`, because it's discovered first. Inheritance uses the first method it finds.

That's why `$film->get_title()` invokes `Movie::get_title()` instead of `Entertainment::get_title()`. Because `get_title()` was found in the `Movie` class, there was no need for Perl to check `Entertainment`.

```
% movie8
11
90
The movie Unforgiven
Western
The movie Unforgiven: now playing at a theater near you.
```

Multiple Inheritance

When a class directly inherits methods from more than one parent class, that's called *multiple inheritance*. Not all object-oriented languages support multiple inheritance, but Perl does. All it means is that you have more than one class in `@ISA`:

```
package Tragicomedy; @ISA = qw(Tragedy Comedy);
```

inherits from both `Tragedy` and `Comedy`.

Figure 7-5 illustrates multiple inheritance: The `Tragicomedy` class inherits from both `Tragedy` and `Comedy`, both of which inherit from `Movie`. The `@ISA` array for `Tragicomedy` might be either `(Tragedy, Comedy)` or `(Comedy, Tragedy)`. The order does matter, however, because `@ISA` is searched depth first.

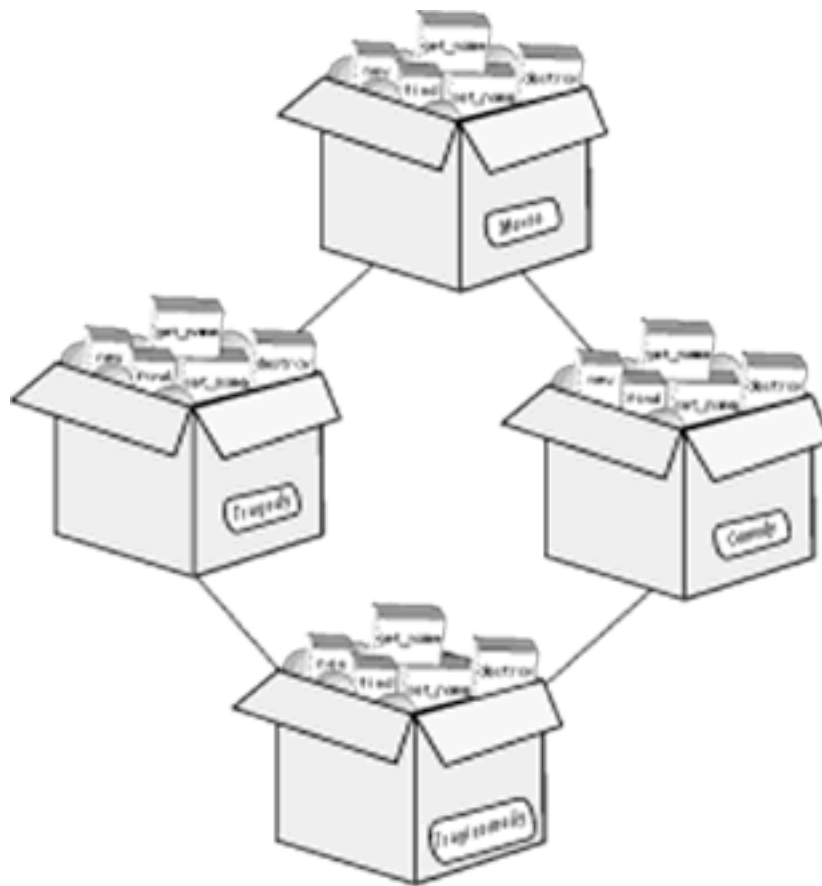


Figure 7-5

Multiple Inheritance

Depth-first search means that as Perl looks through a hierarchy of classes, it checks the parents of a class before it checks the siblings.

The Tragicomedy class uses the @ISA array (Tragedy, Comedy). Because inheritance is depth first, every ancestor of Tragedy is searched before Comedy.

Inheriting Variables

You've seen how one class can inherit methods from another. Classes can inherit instance variables as well. The @ISA array isn't used (that's only for methods); instance variables need to be handled a bit more explicitly, as shown in Listing 7-15.

Listing 7-15 *movie9*: Inheriting the instance variable \$price

```
#!/usr/bin/perl -wl
package Movie;
sub new {
my $class = shift;
my $self = {};
$self->{price} = 8.50;
bless $self;
```

```

}
package Western; @ISA = qw(Movie);
sub new {
my $self = new Movie; # Western->new() calls Movie->new()!
$self->{ending} = `violent`;
bless $self;
}
$film = new Western;
print "film is a ", ref($film);
printf "%1.2f\n", $film->{price};
print $film->{ending};

```

movie9 demonstrates how a subclass can inherit an instance variable from a superclass. The `new()` method for the `Western` class invokes the `new()` method for the `Movie` class, but then blesses the result into the `Western` class. The price variable included in every `Movie` is included in `Westerns` as well.

```

% movie9
film is a Western
8.50
violent

```

You've now learned the basics of object-oriented programming. In the next three lessons, you'll learn advanced OOP techniques and how to put them to use in Perl.

Chapter 7, Perl 5 Interactive Course

LESSON 6

ADVANCED OOP FEATURES

You've now covered the basics of object-oriented programming. Most OOP programs can be written using only the techniques you've seen so far. The next three lessons explore some of the more advanced OOP features, features that you might never need. This lesson covers three: how to distinguish between public and private data, how to access overridden methods, and how to call a method whenever an object is destroyed.

Subpackages

A *subpackage* is a package within a package. Variables like `Movie::price` have shown you that `::` (or, in Perl 4, `'`) means "the thing on the right belongs to the package on the left." That thing can be a package as well as a variable or a subroutine:

```
package Movie;
package Movie::Popcorn;
package Movie::Popcorn::Goobers;
```

The `Movie` package has a `Popcorn` subpackage, which in turn has a `Goobers` subpackage. You can treat subpackages as you would any regular package or class: They can have their own variables and subroutines/methods.

In fact, subpackagehood doesn't imply any relationship at all between the two packages. `Movie` and `Movie::Popcorn` are completely separate packages that happen to have similar names. So what's the point? Well, Perl programmers can use subpackages to declare their intentions about how one class relates to another. And they can "hide" private data (i.e., subroutines and variables) inside a subpackage, as in C++.

In C++, data can be dubbed private or public. In Perl, all methods and variables are public, but you can use subpackages when you want a little privacy.

You can name subpackages anything you want, but if your goal is to keep them hidden from other classes, you could name them "private" to keep C++ programmers happy, as shown in Listing 7-16.

Listing 7-16 *privacy*: Using a subpackage to hide methods and variables

```
#!/usr/bin/perl -l
package Movie;
sub new {
```

```

my $class = shift;
my $self = {};
bless $self;
}
# The Movie::private package is meant for use by the Movie
package ONLY.
package Movie::private;
$release_date = "December 5, 2002";
sub change_title {
my $self = shift;
$self->{title} = shift;
}
package main;
$film = new Movie;

```



```

$film->change_title("2010: Just Eight Years Away"); # undefined
subroutine!

```

As far as the *main* package is concerned, *Movie::private* is distinct from *Movie*, so the *change_title()* method won't be found:

```

% privacy
Can't locate object method "change_title" via package "Movie"
at ./privacy line 23.

```

Perl doesn't prevent other classes from accessing private methods. It merely discourages the practice by forcing you to state your intentions. If you're typing *private* after the name of a foreign class, chances are you're being naughty.

Instead of using the previous line of code, you could have cheated and directly invoked the hidden method:

```

$film->Movie::private::change_title("2010: Just Nine Years
Away");

```

and Perl wouldn't have complained. C++ has the notion of a *friend*, which is a class (or a function) that is permitted to access the private members of another class. If you're not a friend, you have no hope of access. In contrast, Perl lets you access every class from anywhere. As Larry Wall says, "Everyone's a friend in Perl."

Access to Overridden Methods

In *Movie9*, both the *Movie* and *Western* classes contained *new()* methods. When creating a *Western* object, Perl first searches the *Western* class for a *new()* method. Because that search succeeds, Perl never checks the method in the base class: *Movie::new()*. That method is said to be *overridden*.

Every once in a while, you'll want to access an overridden method.

Suppose the `Movie` base class and its derived class `Comedy` each have a `choose()` method. Under ordinary circumstances, a `Comedy` object would use `Comedy::choose()`. But a `Comedy` object can access `Movie::choose()` if it really wants to.

The *suggest* program, shown in Listing 7-17, recommends movies: Given a movie object, it prints the name of another similar movie that the user might enjoy. There are many ways to choose such a movie; two different ways are implemented in `Movie::choose()` and `Comedy::choose()`.

`Movie::choose()` selects and returns another movie with the same director, and `Comedy::choose()` selects and returns another movie with the same lead actor. Here's the catch: If the movie was made before 1970, the director is a better indicator of similarity than the actor. In that event, you want to override the overriding.

Listing 7-17 *movierec*: Accessing an overridden method

```
#!/usr/bin/perl -wl
package Movie;
sub choose { # If it's a pre-1970 movie...
my $self = shift;
print "Maybe another movie by the same director?";
}
package Comedy; @ISA = qw(Movie);
@Comedy::Inherit::ISA = @ISA; # used to access overridden
methods
sub new {
my $class = shift;
my $self = {};
%$self = @_;
bless $self;
}
# Depending on year, choose() either prints a string,
# or invokes the choose() method of its base class.
# Movie::choose() is inherited through the @ISA
# belonging to Comedy::Inherit instead of the @ISA
# belonging to Comedy.
sub choose {
my $self = shift;
if ($self->{year} > 1970) { # If it's a post-1970 movie...
print "Maybe another movie with the same lead actor?";
} else { # otherwise, use Movie::choose.
$self->Comedy::Inherit::choose($self);
}
}
package main;
$film = new Comedy(year => 1977);
$film->choose;
```



```
$cine = new Comedy(year => 1960);
$cine->choose;
```

After the Comedy class is declared, an Inherit subpackage is (implicitly) created, containing a lone @ISA array. Its sole reason for existence is to inherit from base classes.

When Comedy::choose() wants to call Movie::choose(), it calls Comedy::Inherit::choose() instead. Thanks to its @ISA, that resolves to Movie::choose(). (So why didn't Comedy::choose() simply call Movie::choose()? Well, it could have. But proper OOP style says you should let the inheritance mechanism select your methods, in case you change the class relationships later.)

There's nothing magical about what's going on: Inherit could just as well have been called Parents or MyLittleInheritanceTree and the result would have been the same.

```
% movierec
Maybe another movie with the same lead actor?
Maybe another movie by the same director?
```

Destructors

Many OOP languages let you create *destructor methods* that clean up after objects when you're done with them. Languages like C++ place the burden of memory allocation and deallocation upon the programmer. Because Perl takes care of those tasks for you (it performs reference counting--counting the number of references to an object, and destroying it when the count reaches 0), you don't need to call a destructor explicitly. (In C++ there is no reference counting, so you have to remember to call destructors yourself.)

If you want to create your own destructor, you can do so by naming a method DESTROY(). If a DESTROY() method exists, it's called automatically when the object falls prey to undef(), if your program ends, or if anything else obliterates the object.

The DESTROY method is executed as the "last gasp" of an object, as shown in Listing 7-18.

Listing 7-18 *lastgasp*: Using a DESTROY() method

```
#!/usr/bin/perl -wl
package Movie;
# new() creates a Movie and adds its time to the packagewide
%schedule
sub new {
my $class = shift;
my $self = {};
%$self = @_;
$schedule{$self->{title}} = $self->{'time'};
bless $self;
}
# showtimes() (a static method) prints the contents of
%schedule
sub showtimes {
```

```

my $class = shift;
my ($key, $value);
while (($key, $value) = each %schedule) {
    print "$key begins at $value.";
}
}
# DESTROY() deletes the movie time from %schedule
sub DESTROY {
    my $self = shift;
    print 'AAAAAAAAAAAAAGH!';
    delete $schedule{$self->{title}};
}
package main;
$film = new Movie(title => 'The War of the Roses', 'time' =>
'7:30');
$scine = new Movie(title => 'The Bicycle Thief', 'time' =>
'8:15');
$flick = new Movie(title => 'Get Shorty', 'time' => '7:50');
Movie->showtimes();
undef $scine;
Movie->showtimes();

```

In addition to creating a Movie object, new() adds the movie showtime to a package-wide hash: %schedule. DESTROY() removes the showtime. When \$scine is undef()'ed, DESTROY() is automatically called, printing AAAAAAAAAAAAAAGH! and removing the key-value pair for *The Bicycle Thief* from %schedule. When the program finally ends, DESTROY() is called for both remaining Movies: \$film and \$flick.

```

% lastgasp
Get Shorty begins at 7:50.
The Bicycle Thief begins at 8:15.
The War of the Roses begins at 7:30.
AAAAAAAAAAAAAGH!
Get Shorty begins at 7:50.
The War of the Roses begins at 7:30.
AAAAAAAAAAAAAGH!
AAAAAAAAAAAAAGH!

```

Reblessing Objects

The *movie9* program at the end of the last lesson contains the following method inside the Western class:

```

package Western; @ISA = qw(Movie);
sub new {
    my $self = new Movie;
    $self->{ending} = 'violent';
    bless $self;
}

```

The line in bold deserves further scrutiny. `Movie::new()` blesses its object into the `Movie` package, and then the line above reblesses it into the `Western` package.

You can rebless an object as many times as you like:

```
$Crichton = new Book;
print "Crichton's latest ", ref($Crichton), "\n";
prints Crichton's latest Book.

 bless($Crichton, "Movie");
print "Crichton's latest ", ref($Crichton), "\n";
prints Crichton's latest Movie.

 bless($Crichton, "TV_Show");
print "Crichton's latest ", ref($Crichton), "\n";
prints Crichton's latest TV_Show.

 bless($Crichton, "CD_ROM");
print "Crichton's latest ", ref($Crichton), "\n";
prints Crichton's latest CD_ROM.
```

One caveat: If you have a `DESTROY()` method in one class and you rebless an object in that class into another class, you'll need to explicitly call the old `DESTROY()` if you want it to take effect.

Chapter 7, Perl 5 Interactive Course

LESSON 7

OOP DESIGN METHODS

Perhaps future computer languages will be able to identify not just syntax errors but semantic errors, such as flawed logic or questionable program designs. In the meantime, OOP lets you build complex, wonderful programs, but it guarantees only the "complex" part--more organization doesn't necessarily mean better organization. It's harder to understand a program with a misleading structure than a program with no structure whatsoever.

This lesson formalizes some of the techniques for structuring programs: how one class can contain another, how you can maximize the reusability of your code, and how to create classwide variables.

A "Containing" Relationship

So far, you've seen only one style of inheritance: superclass-subclass, in which the derived class is a subset of the base class. That's by far the most common use of inheritance.

That style is used so often that many people mistakenly think that inheritance is synonymous with a superclass-subclass relationship. But there are other ways for one class to use another.

For instance, a class can contain another class. Think about the difference between these two statements:

- **All Westerns are Movies.**
- **Every Movie contains a Script.**

Every Movie object contains a Script object. Neither is a subclass of the other; rather, every Movie has a Script. But there's no need for a @HASA array. All you need is a method call.

The Movie::new() method in contain (Listing 7-19) demonstrates how a Movie might contain a Script.

Listing 7-19 *contain*: A Movie contains a Script

```
#!/usr/bin/perl -wl
package Script;
sub new {
my $class = shift;
my $self;
%$self = @_;
bless $self;
}
```

```

package Movie;
sub new {
my $self;
$self->{actor} = 'John Travolta';
$self->{Script} = new Script(author => 'Tarantino');
bless $self;
}
package main;
$film = new Movie;
print $film->{Script}{author};

```

Every Movie contains its own Script, which is a full-fledged object. Movie::new creates a hash entry (called simply Script) whose value is a Script object. Because \$film->{Script} is an object, its data can be accessed as you might expect: **\$film->{Script}{author}**.

```

% contain
Tarantino

```

That's all there is to it: The "containing" relationship, like everything else in object-oriented Perl, is nothing more than a judicious use of references, packages, and subroutines.

Calling the Right Method

Suppose a method (say, `determine_profit()`) in one class (say, `Movie`) is called by an object from another class (say, `Laserdisc`). Furthermore, suppose `determine_profit()` calls another method, `profit()`:

```

package Movie;
sub determine_profit {
my $self = shift;
return profit();
}

```

Now suppose that `profit()` exists in both classes. Which one is called? As shown, it'll be `Movie::profit()`. But it would be nice if `determine_profit()` chose which `profit()` to call based on the class of `$self`. It's easy enough:

```

sub determine_profit {
my $self = shift;
return $self->profit;
}

```

profit, shown in Listing 7-20, demonstrates how to "call the right method."

Listing 7-20 *profit*: Calling an unanticipated method

```

#!/usr/bin/perl -wl
package Movie;
sub new {
my $self = {};
$self->{box_office} = 100e+6;
$self->{cost} = 50e+6;

```

```

bless $self;
}
sub determine_profit {
my $self = shift;
return $profit = $self->profit
# Movie::profit() or Laserdisc::profit()
}
sub profit {
my $self = shift;
return $self->{box_office} - $self->{cost};
}
package Laserdisc; @ISA = qw(Movie);
sub new {
my $self = {};
$self->{rentals} = 20e+6;
$self->{cost} = 50e+6;
bless $self;
}
sub profit {
my $self = shift;
return $self->{rentals} - $self->{cost};
}
package main;
$film = new Movie;
$rental = new Laserdisc;
print "Film profits: ", $film->determine_profit;
print "Laserdisc profits: ", $rental->determine_profit;

```

Even though `determine_profit()` is inside the `Movie` class, it calls a method that can be found in both the `Movie` and the `Laserdisc` classes. So what happens?

- If `$self` is a `Movie`, `Movie::profit()` is called.
- If `$self` is a `Laserdisc`, `Laserdisc::profit()` is called.

If `determine_profit()` had been called as `profit()` instead of `$self->profit()`, then `Movie::profit()` would have been invoked even if `$self` were a `Laserdisc`. `$self->profit` yields the method most appropriate for the class; it's a more flexible way to structure your code.

```

% profit
Film profits: 50000000
Laserdisc profits: -30000000

```

The `Movie` class is an example of reusable code; you can extract the class and place it in another program, even as a superclass. When the `determine_profit()` method is invoked, it will function correctly regardless of whether the subclass has a `profit()` method. Programs structured this way are easier to understand and cause fewer errors when modified or incorporated into new programs.

A very general rule for creating reusable code is this: Use `->` as frequently as possible and use `::` as

infrequently as possible.

Class Variables

Because a class is just a package and every package has its own symbol table, you can store plain old variables in a class:

```
package Movie;
%cast = (Rocky => `Sylvester Stallone`);
```

All methods in the `Movie` class can use the package variable `%cast`. But they shouldn't. For the most readable and reusable code, methods should use data from either `@_` or from the object's instance variables, NOT from variables that are global to the entire package.

Objects should tell methods where their data is located, as demonstrated by Listing 7-21.

Listing 7-21 *rollcall*: Using objects to find variables

```
#!/usr/bin/perl -wl
package Movie;
%cast = (Ferris => `Matthew Broderick`, Cameron => `Alan
Ruck`,
Rooney => `Jeffrey Jones` , Grace => `Edie McClurg`);
sub new {
my $self = {};
$self->initialize();
bless $self;
}
sub initialize {
my $self = shift;
$self->{cast} = \%cast;
}
sub roll_call {
my $self = shift;
while (($key, $value) = each %{$self->{cast}} ) {
print "$value as $key";
}
}
package Laserdisc; @ISA = qw(Movie);
%cast = (Skywalker => `Mark Hamill`, Solo => `Harrison Ford`,
Leia => `Carrie Fisher`);
sub new {
my $self = {};
$self->initialize();
bless $self;
}
package main;
```

```
$film = new Movie;  
$rental = new Laserdisc;  
$film->roll_call;  
$rental->roll_call;
```

Both classes store a reference to %cast inside their objects, taking the burden off roll_call. Otherwise, \$rental->roll_call() would need to choose between %Movie::cast or %Laserdisc::cast. It's better to remove the ambiguity by storing a reference to the correct variable.

```
% rollcall  
Edie McClurg as Grace  
Matthew Broderick as Ferris  
Jeffrey Jones as Rooney  
Alan Ruck as Cameron  
Carrie Fisher as Leia  
Harrison Ford as Solo  
Mark Hamill as Skywalker
```


Chapter 7, Perl 5 Interactive Course

LESSON 8

OVERLOADING

When you *overload* one of Perl's built-in operators, you give it a new behavior. For instance, if you define a class of numbers that add in a strange way, you'll want to overload the `+` operator so that `$a + $b` will Do The Right Thing. Or, say you define your own String class in which capitalization never matters. You might want to overload `eq` so that your own function or method is called instead, one that will `lc()` both arguments.

Incorporating overloading into a programming language is usually tricky. It's no different with Perl--overloading is not yet an "official" part of Perl 5, but is getting more stable with every release. The first edition of this book explained the overloading mechanism for 5.002. Then overloading was overhauled, and now this edition explains overloading as implemented in 5.003 and 5.004. When in doubt, consult Ilya Zakharevich's overload documentation bundled with the Perl distribution. It's embedded in the `overload.pm` file which you'll find in your `LIB/` directory.

The overload Pragma

To map an operator to a new behavior, you use the *overload* pragma. (Pragmas are formally introduced in the first lesson of Chapter 8.) You can specify the behavior in any of three ways: with the name of a subroutine, with a reference to a subroutine, or as an anonymous subroutine. If you want to define "add," "subtract," and "equals" for your Movie class, you might do this:

```
package Movie;
use overload ("+" => "my_add", "-" => \&subtract, "eq" => sub
{ return 0 });
```

after which you could say things like

```
$oscar = $film + $cine; # Calls the my_add method
$cine = $film - $oscar; # Calls the subtract subroutine
print "eq!" if $film eq $oscar # Evaluates { return 0 }
```

Any operator mapped to a string (as in `"+" => "my_add"` above) will be treated as a method, making possible inheritance of operators.

In this lesson, a Complex number class will be used to explore overloading. Every complex number has two parts: a real number and an imaginary number (which is a real number multiplied by *i*, equal to $\sqrt{-1}$). Here are four complex numbers:

7+5i 8+(-2)i 4.77+0i 0+0.4i

When you add complex numbers, you perform two separate additions.

You add the real parts and the imaginary parts independently:

$$(a+bi) + (c+di) = (a+c)+(b+d)i$$

The *complex* program shown in Listing 7-22 defines a Complex class and overloads the + operator to handle complex addition.

Listing 7-22 *complex*: Using the *overload* pragma to define addition for complex numbers

```
#!/usr/bin/perl -w
package Complex;
use overload ("+" => "complex_add");
sub new {
my $class = shift;
my $self = {};
%$self = @_;
bless $self;
}
sub display {
my $self = shift;
print "$self->{real} + $self->{imag}i\n";
}
sub complex_add {
my ($a, $b) = @_;
my $sum = {real => $a->{real} + $b->{real},
imag => $a->{imag} + $b->{imag}};
bless $sum;
}
package main;
$x = new Complex (real => 2, imag => 5 );
$y = new Complex (real => 3, imag => 4.3 );
display $x;
display $y;
$z = $x + $y;
display $z;
```

complex overloads the + operator so that when Perl sees $\$x + \y in the second-to-last line, it calls `complex_add($x, $y)`. The result:

```
% complex
2 + 4i
3 + 4.3i
5 + 9.3i
```

Overloadable Operators

As you've seen, all methods expect either a class name or an object as the first argument. But suppose you want to add a real number to a complex number, as in $21 + \$x$? You don't want `complex_add()` receiving 21 as its first argument, because every static method wants an object as its first argument. Perl, anticipating this, reverses the order of the arguments for you, so that $\$x$ is first. That's fine for addition, but what if the order of the arguments matters? You'll usually want $21 + \$x$ to be the same as $\$x + 21$, but subtraction isn't commutative: $21 - \$x$ is very different than $\$x - 21$.

That's where the third argument comes in: It tells you whether Perl swapped the arguments. If the third argument is `TRUE`, the arguments were swapped; if it's `FALSE`, they weren't. If the third argument is `UNDEF`, Perl is telling you that your operation (say, `+`) was needed indirectly to compute another operation (say, `+=`).

For unary operators such as `++`, the second argument is always `UNDEF` and the third argument is always `FALSE`.

What about the lone ternary operator `?:`, which wants three arguments all to itself? You can only overload it indirectly, with a special keyword recognized by the *overload* pragma: *bool*. Here's the complete list of overloadable operations, derived from Perl's *overload* documentation.

Basic operators:

`+` `-` `*` `/` `%` `**` `<<` `>>` `x` `.`

Assignment operators:

`+=` `-=` `*=` `/=` `%=` `**=` `<<=` `>>=` `x=` `.=`

If you overload `+` but not `+=`, the latter will invoke the former. That's called *autogeneration*, and it occurs whenever one operator can be expressed in terms of another. As another example, the `-` (subtraction) operator is used to autogenerate methods for `--`, `-=`, *abs*, and "neg" (unary minus).

Comparison operators:

`<` `<=` `<=>` `>` `>=` `==` `!=`
`lt` `le` `cmp` `gt` `ge` `eq` `ne`

If `<=>` (or *cmp*) is overloaded, it's used to autogenerate methods for the related numeric (or string) operators.

Bit operators:

`&` `^` `|` `!` `~` `neg`

"neg" represents unary minus, as in `-7`.

If there's no method for `!`, it's autogenerated with one of the "special conversion operators" below.

Increment and decrement operators:

`++` `--`

Mathematical functions:

`abs` `sqrt` `log` `exp` `sin` `cos` `atan2`

If `abs()` is undefined, it is autogenerated with either `<<` or `<=>`, combined with `-` or `"neg."`

Special conversion operators:

```
0+ \"\\" bool
```

Like `"neg,"` these aren't real Perl operators, but rather names contrived especially for overload. `0+` specifies what should happen when the value is evaluated as a number, `\"\\"` when the value is evaluated as a string, and `bool` when the value is evaluated as a TRUE or TRUE value.

`bool` is used for flow control operators like *if* and *while*, and for `?:`.

Special overloading symbols:

```
fallback nomethod =
```

fallback is used whenever an operator can't be found. If it's UNDEF, then Perl will try to autogenerated a behavior as usual, yielding an error if it can't (and if *nomethod*, below, hasn't been defined). If it's TRUE (e.g., `use overload ("fallback" => 1)`), then Perl will still try to autogenerated a behavior, but won't complain. If it's defined but false, Perl won't attempt autogeneration, relying instead solely on *nomethod*.

nomethod should be a reference to a function with four arguments. It's called whenever Perl can't find a method for an operator: The first three arguments are the regular overloading arguments, and the fourth is the symbol corresponding to the missing operator. So if `-` isn't overloaded, but you've defined a *nomethod* function, then the expression `8 - $x` will invoke that function with the four arguments `$x`, `8`, `1`, and `-`.

If you overload `=`, it'll be used in situations when a mutator is applied to a reference that shares its object with another reference. Consider what happens with the following pair of statements:

```
$y = $x;  
$y++;
```

After `$y = $x`, we have two references that refer to the same object. The object isn't actually copied until the `$y++` statement, and then only if `++` (or `+=`) is overloaded to a method.

Functions Related to Overloading

The *overload* module provides three public functions. You'll probably never need them.

`overload::StrVal($x)` yields the string value of `$x`.

`overload::Overloaded($x)` yields TRUE if `$x` is subject to overloading of any kind.

`overload::Method($x, $op)` yields a reference to the method overloading `$op`, or UNDEF if none exists.

Review

Clint: Thanks to inheritance, I can override methods. And thanks to overloading, I can override operators. What about variables? Why isn't there an easy mechanism to use variables from another class?

Gene: There is-the double colon operator. Of course, if you're being strict about object-oriented methodology, you should give up variables altogether, storing and fetching variables with methods specially designed for this purpose.

Clint: No thanks. Another question: What's the difference between `Class->method()` and `Class::method()`?

Gene: The first is a method invocation, the second is a function call. `Class::method()` won't ever perform inheritance.

Chapter 8, Perl 5 Interactive Course



BROWSING THE LIBRARY: MODULES

All too often, programming books tell you only how to write your programs rather than what programs have already been written. But the Perl community has contributed hundreds of utilities to the public domain, many of which have been incorporated into the standard Perl distribution. Chances are you'll find some of them useful.

Appendix B lists some of the best places to find Perl scripts on the Internet. This chapter is about utilities that you can find on your computer right now. These utilities are files that aren't part of the Perl language itself, but come bundled with Perl anyway. Perl 5 introduces *modules*, self-contained chunks of code usable by your programs. Modules are, well, modular.

This chapter discusses many of the library files that are supplied with the Perl distribution and a few that you can retrieve via the Internet. Much of this information will change as new utilities are integrated into the Perl distribution and old utilities are improved. This chapter is merely a sampling of the goodies out there right now.

Not all library files were written by Larry Wall; other authors are acknowledged where possible. If you create a utility that you think others might find useful, consider making the world a better place and uploading it to the Comprehensive Perl Archive Network (CPAN); visit <http://www.perl.com> or one of the sites listed in Appendix B. Tim Bunce maintains a list of modules that is posted there regularly (and archived at <ftp://rtfm.mit.edu/pub/usenet/news.answers/perl-faq/module-list>). You can acquire new modules from one of the Comprehensive Perl Archive Network (CPAN) sites listed in Appendix B.

After introducing pragmas, library files, and modules (and detailing how you can build your own), this chapter breezes through about three dozen of the most frequently used library files, giving brief descriptions and examples of each.

LESSON 1

THE PERL LIBRARY

In Chapter 3, you saw that you can execute one program from within another using `do()`. This chapter is about a set of programs, collectively called the *Perl library*, that have been collated and incorporated into the Perl distribution (as opposed to the Perl language itself).

Library Files

You have to tell `do()` exactly where your script can be found. That's okay when you have a script that's only used by your programs--after all, you know where to find it. But there's a standard location for Perl utilities on every system and a function smart enough to look there: `require()`.

The `require()` Function

`require FILE` includes the library **FILE** (at runtime), if it hasn't already been included.

`require NUMBER` exits the program unless the current version of Perl is greater than or equal to that number.

If `require()`'s argument is omitted, `$_` is used.

```
require `assert.pl`;
```

includes the library file *assert.pl*.

```
require 5;
```

ensures that Perl 5 (or higher) is running.

`require()` is like `do()`, but it remembers what it has included--`require()` won't include the same file twice, unlike `do()`, which executes the file every time you call it.

Two (easily forgotten) attributes of library files distinguish them from conventional Perl programs:

- **They don't have a `#!/usr/bin/perl` at the beginning because they aren't standalone programs (there are a few exceptions to this, however).**
- **They must return a `TRUE` value as their last statement so that `require()`, `use()`, and `do()` know that the code has been executed successfully.**

Customarily, the second condition is fulfilled by explicitly ending the file with

```
1;
```

which, of course, is always `TRUE`. So where does `require()` look for library files?

@INC and %INC

@INC contains the list of directories where Perl searches for library files. After a library file has been included, **%INC** stores it where it was found.

@INC varies from system to system:

```
% perl -e 'print "@INC"'
```

```
RESULT /usr/local/lib/perl5 /usr/local/lib/perl5/m68-next
```

On an MS-DOS machine, you might see something like this:

```
% perl -e "print qq(@INC)"
```

```
RESULT c:/perl5/msdos c:/perl5/lib .
```

Running Perl with the -V flag will also show the contents of @INC, as well as other choices used to configure Perl on your computer.

One of these directories will contain all the library files bundled with Perl. It's often */usr/local/lib/perl5* on UNIX machines and *c:\perl5\lib* on MS-DOS machines (even though @INC entries contain forward slashes: *c:/perl5/lib*.) We'll call that directory *LIB*, whatever it is.

If you look in your *LIB* directory, you'll see a lot of files ending in *.pl* or *.pm* and a few directories with more of the same.

.pl and .pm Files

A file ending in **.pl** is a Perl library file. A file ending in **.pm** is a Perl module.

If the *FILE* in **require FILE** is a bareword (an unquoted string), **require()** assumes a *.pm* extension.

If your system permits, create a file in one of the @INC directories called *domino.pl*, containing the following two lines:

```
@games = ("Tiddly-Wink", "Matador", "Bergen", "Sebastopol");  
1;
```

Then run *dominos*, shown in Listing 8-1.

Listing 8-1 *dominos*: Using **require()** to include a library file

```
#!/usr/bin/perl -w  
require 'domino.pl'; # brings @games into the current  
namespace  
srand (time ^ ($$ + ($$ < 15)));  
print "A random game: $games[rand(@games)] \n";
```

The **require()** pulls in *domino.pl*, which defines @games:

```
% dominos
```

```
RESULT A random game: Tiddly-Wink
```


Listing 8-2 demonstrates the other feature of `require()`: ensuring that the user's Perl isn't too old.

Listing 8-2 *need5*: Using `require()` to ensure that Perl 5.004 is being used

```
#!/usr/bin/perl
require 5.004;
$greeting = ["Hello", "world"];
print "@$greeting";
```

If *need5* is run by Perl version 5.003 or less, the following error is printed and the program die(s):

```
% need5
```

```
RESULT Perl 5.004 required--this is only version 5.003, stopped at need5
line 2.
```

How did Perl know its version number? It's contained in the special variable `$]`.

The Special Variable `$]`

`$]` (or `$PERL_VERSION`) contains the version number of Perl.

(No, you can't set `$]` to upgrade to Perl 6 automatically.)

Modules

In Chapter 7 you learned that a class is just a package. A module is just a package, too, with a *.pm* extension. Modules:

- Usually contain subroutines or methods for use by other programs (just like *.pl* files)
- Are packages and therefore contain their own symbol tables (just like classes)

By convention, module names are capitalized.

The `use()` Function, and the `-m` and `-M` Command-Line Flags

use *MODULE ARRAY* includes the subroutines and variables mentioned in *ARRAY* from *MODULE*. The `-m` and `-M` flags let you emulate a `use()` statement from the command line.

The *ARRAY* in `use()` is usually omitted, in which case all the *MODULE*'s subroutines and variables are brought (imported) into the current package.

If *MODULE* is a bareword, Perl assumes a *.pm* extension.

```
use English;
```

looks for *English.pm* in the `@INC` directories and imports its subroutines and variables into the current package.

```
use MyModule;
```

does the same for *MyModule.pm*.

```
use MyModule `tetrade`;
```

imports just the `tetrade()` subroutine into the current package.

```
use MyModule qw(wind polish %bow);
```

imports just the `wind()` and `polish()` subroutines and the `%bow` hash.

A `use()` anywhere in your program takes effect as Perl reads your program, as opposed to a `require()`, which is executed only after the program has been compiled. `use()` can also alter (some say "pollute") your symbol table even if the module is a separate package. You'll learn how that works in Lesson 3.

The **-m** and **-M** flags let you `use()` modules from the command line:

```
perl -mMyModule ...
```

and

```
perl -M`MyModule qw(wind polish %bow)` ...
```

behave like **use MyModule** and **use MyModule qw(wind polish %bow)**. See Appendix G for details.

Pragmas

Pragmas are a special kind of module that change how a block of statements is compiled. Three widely used pragmas are discussed in this section: *integer*, *subs*, and *strict*.

You can countermand a pragma with `no()`.

The `no()` Function

no PRAGMA ARRAY undoes a pragma.

If provided, *ARRAY* narrows down the scope of what's undone:

```
no strict;
```

undoes the *strict* pragma, and

```
no strict `subs`
```

undoes the *subs* part of the *strict* pragma.

Pragmas are used to make all-encompassing changes to your program that will affect how Perl compiles it, such as how arithmetic operations are performed.

The integer Pragma

The *integer* pragma (Listing 8-3) tells Perl to use integer arithmetic for all mathematical operations; everything after the decimal point is dropped. On most computers, that will be faster.

Listing 8-3 *nofrac*s: The *integer* pragma

```
#!/usr/bin/perl -wl
use integer; # integer-only arithmetic
print `22/7 is `, 22/7;
no integer; # return to floating-point arithmetic
print `22/7 is `, 22/7;
```

Here's *nofrac*s in action:

```
% nofrac
```

```
RESULT 22/7 is 3
```

```
22/7 is 3.142857142857143
```

If you see an error message complaining *Can't locate integer.pm*, then your Perl distribution hasn't been installed correctly. You'll want to fix that; the rest of this chapter won't be much fun for you otherwise. Find the modules from your Perl distribution and make sure they're in one of the directories in `@INC`. (This is why copying Perl binaries from machine to machine is not a good thing--unless you duplicate the LIB directory as well).

The subs Pragma

The *subs* pragma predeclares subroutines for you. To see why that's sometimes helpful, consider the following behavior under "normal Perl."

You can do this:

```
chew("asteroid"); # OK, Perl can tell that's a function call
sub chew { print substr($_[0], -1, 1) }
```

or this:

```
sub chew { print substr($_[0], -1, 1) }
chew "asteroid"; # OK, Perl now knows about chew()
```

but not this:

```
chew "asteroid"; # ERROR! Unrecognized function!
sub chew { print substr($_[0], -1, 1) }
```

unless you first declare the subroutine before the real definition:

```
sub chew;
chew "asteroid"; # OK, chew() has been predeclared
sub chew { print substr($_[0], -1, 1) }
```

or predeclare `chew()` with the *subs* pragma:

```
use subs 'chew';
chew "asteroid"; # OK, chew() has been predeclared
sub chew { print substr($_[0], -1, 1) }
```

The *subs* pragma tells Perl not to fret if it encounters an as-yet-undeclared function.

Prototypes

Perl 5.002 and later releases support *prototypes*, which let you declare a subroutine (and its arguments) without explicitly defining the subroutine. They notify the Perl compiler that a subroutine definition will appear somewhere later in the file. As shown above,

```
sub chew;
```

has that effect. Thanks to prototypes, you can declare the number and types of arguments to your subroutines:

```
sub one_scalar ($) ;
sub three_scalars ($$$) ;
```

Here, the subroutine `one_scalar()` is declared to take one scalar argument, while `three_scalars()` takes three. If you then call `three_scalars()` with fewer than three arguments, Perl complains:

```
RESULT Not enough arguments for main::three_scalars at ./your_program
```

The prototype can be included with the actual subroutine definition:

```
sub two_scalars ($$) {
my ($a, $b) = @_;
print "My first scalar is $a and my second is $b.\n";
}
```

As you might expect, you can prototype arrays with `@s`:

```
sub one_array (@) ;
```

tells Perl that the subroutine `one_array()` expects a single array. For multiple array arguments, the usual caveat applies: Because Perl merges subroutine arguments into a single array, you need to disambiguate multiple arrays with references. But here's where prototypes pay off: They let the referencing happen implicitly.

```
sub two_arrays (\@\@) {
my ($aref, $bref) = @_;
print "My first array is @$aref and my second is @$bref.\n";
}
@a = (1, 2, 3);
@b = (4, 5, 6);
two_arrays(@a, @b); # No backslashes required!
```

prints

```
RESULT My first array is 1 2 3 and my second is 4 5 6.
```

Optional arguments can be placed after a semicolon:

```
sub one_scalar_and_maybe_another ($;$) {
my ($a, $b) = @_;
print "My mandatory argument is $a and my optional argument
is $b\n";
}
```

And the fun continues. You can specify type globs for subroutines that expect filehandles:

```
sub one_glob(*);
```

and % for hashes, and & for anonymous subroutines:

```
sub one_function_and_an_optional_hash_reference(& \%);
```

Prototypes will be ignored during method invocations, or if you preface the call to your subroutine with &:

```
&two_scalars(4, 5, 6)
```

prints

```
RESULT My first scalar is 4 and my second is 5.
```

while

```
two_scalars(4, 5, 6)
```

results in the error message:

```
RESULT Too many arguments for main::two_scalars
```

For more information about prototypes, consult the *perlsub* documentation bundled with Perl.

The strict Pragma

The *strict* pragma lets you outlaw certain programming constructs:

Symbolic references use strict refs;

Global variables use strict vars;

Barewords use strict subs;

Or all three use strict;

If your program violates a restriction, Perl immediately aborts compilation. *use strict* is handy for complex programs, where there's a lot going on and you want to be absolutely sure that your program does what you intended. Listing 8-4 (*nosymref*) outlaws symbolic references, which sometimes trip up programmers.

Listing 8-4 *nosymref*: Preventing symbolic references

```
#!/usr/bin/perl -w
use strict refs;
$symref = "film";
$$symref = "movie"; # die() rather than set $film to "movie".
```

nosymref uses **strict refs** to prevent the symbolic reference that occurs in line 5, when the derived scalar `$film` is set to **"movie"**. If you run *nosymref*, you'll see this error message:

```
% nosymref
```

```
RESULT Can't use string ("film") as a SCALAR ref while "strict refs" in
use at nosymref line 6.
```

For some programmers, global variables are another no-no. Perl doesn't let you create true global variables. But it does let you create "package-wide" global variables--unless you **use strict vars**, as shown

in Listing 8-5.

Listing 8-5 *noglobal*: Preventing global variables

```
#!/usr/bin/perl -w
use strict "vars";
$film = "movie"; # die() rather than set the global variable
$film.
```

`$film` isn't explicitly scoped--it's not declared with `local()` inside a subroutine, nor is it declared with `my()`, nor is it qualified with a package name (`$main::film`). So **use strict vars** won't let it hurt you:

```
% noglobal
```

RESULT Global symbol "film" requires explicit package name at *noglobal* line 5

```
Execution of noglobal aborted due to compilation errors.
```

It's not hard to pacify **use strict vars**. Just create a scope with `my()`:

```
my $film = "movie";
```

Finally, *nobares* (Listing 8-6) prevents barewords, making you explicitly quote all strings.

Listing 8-6 *nobares*: Preventing barewords

```
#!/usr/bin/perl
use strict "subs";
$film = movie; # die() rather than set $film to bareword
"movie".
```

use strict subs is handy for catching typos: If you declare a `MySub` subroutine and then mistype it as `Mysub` (with no arguments or parentheses), Perl treats `Mysub` as a string, unless you **use strict subs**. If it's a string, it should look like a string--use quotes.

```
% nobares
```

RESULT Bareword "movie" not allowed while "strict subs" in use at *nobares* line 5.

```
Execution of nobares aborted due to compilation errors.
```

Other Pragmas

There are three other pragmas: *sigtrap* (covered in Chapter 9, Lesson 2), *diagnostics* (covered in Chapter 10, Lesson 1), and *less*, which hasn't been implemented yet. When *less* becomes functional, it will let you balance the resources used by your program, such as

```
use less 'memory';
use less 'CPU';
```

Chapter 8, Perl 5 Interactive Course

LESSON 2

ENGLISH, TIME, SYSLOG, AND POD

From now until the end of the chapter, you'll learn about many of the standard modules that come with Perl. This book shows you how to use them, not how they are built--but feel free to peek under the hood and browse around Perl's *LIB* directory.

English

Perl can be wonderfully concise. It can also be maddeningly concise, such as when you forget the meanings of special variables: `$%` or `$-` or `$&` or `$/`. The `English` module lets you substitute some predefined English names instead: `$FORMAT_PAGE_NUMBER` can be used in place of `$%`, `$MATCH` may be used in place of `$&`, and either `$INPUT_RECORD_SEPARATOR` or `$RS` can be used in place of `$/`. A full list of the English names is in Appendix H.

English

readable, shown in Listing 8-7, demonstrates using five English names in place of the more cryptic `$\`, `$0`, `$]`, `$&`, and `$_`.

Listing 8-7 *readable*: Enhancing readability with *English.pm*

```
#!/usr/bin/perl -w
use English;
$OUTPUT_RECORD_SEPARATOR = "\n"; # instead of $\
print "This is $PROGRAM_NAME"; # instead of $0
print "Running under Perl version $PERL_VERSION"; # instead
of $]
$ARG = "squanch"; # instead of $_
m/(qua)/;
print "Matched $MATCH in $ARG." # instead of $& and $_
```

The output is exactly what you'd expect:

```
% readable
```

```
RESULT This is readable
```

```
Running under Perl version 5.004
```

```
Matched qua in squanch.
```

The `English` module is popular. So why aren't there similar modules for other languages? Why not have

an Italian module and a French module and a Norwegian module? Or, because English spelling isn't uniform, a British English module?

If Perl were to let you write in any language you chose, the chances of your understanding programs written elsewhere would plummet. The one-character symbols might be cryptic, but they mean the same thing to everyone. When confronted with a technical paper in a foreign language, mathematicians have an easier time of it than lawyers--because the meanings are expressed more with symbols than with words, the native language of the reader doesn't matter so much.

Time

If you look in your *LIB* directory, you won't see a *Time.pm* or a *Time.pl*. That's because Time is a module set--a directory containing modules with a common theme. At the very least, it should contain *Local.pm*, embodying the Time::Local module. The line:

```
use Time::Local;
```

makes the functions exported by Time::Local available to your program.

Time::Local

This module contains two functions: timelocal() and timegm(). They behave like opposites of localtime() and gmtime(), allowing you to convert a set of time and date fields to UNIX time format (the number of seconds since January 1, 1970). Listing 8-8 uses timelocal().

Listing 8-8 *bigtime*: timelocal() is the opposite of localtime()

```
#!/usr/bin/perl -wl
use Time::Local; # Look for Local.pm in the lib/Time
directory
print "The clock struck ", timelocal(14,13,12,11,10,98);
print "In London, that's ", timegm(14,13,12,11,10,98);
timelocal($sec,$min,$hours,$mday,$mon,$year);
```

bigtime uses timelocal() and timegm() to translate 12:13:14 on November 11, 1998, (remember, months are indexed from 0) into UNIX time:

```
% bigtime
```

```
RESULT The clock struck 910804394
```

```
In London, that's 910786394
```

```
UNIX
```

Syslog

The UNIX syslog facility allows users (and mailers and print spoolers) to append messages to system logs. System logs are plain text files; every time you send mail, a record is (probably) appended to the mail system log. Your Perl programs can append messages to the user system log, which you might find

somewhere beneath the directories */usr/adm* or */var/adm*.

Sys::Syslog

(Module written by Tom Christiansen)

Syslog (and syslogd, the daemon that handles syslog requests) varies from system to system. This code might not run on all platforms; consult your UNIX syslog documentation, as well as the notes in *Sys::Syslog.pm*, for more information. Listing 8-9 shows the *logfile* program.

Listing 8-9 *logfile*: Using Sys::Syslog to append messages to a system log

```
#!/usr/bin/perl
use Sys::Syslog;
# Start a log in the "user" log file
openlog('MyLog', 'cons,pid', 'user');
# Append four different messages to the log file
syslog('info', "Some information");
syslog('notice', "A notice, at time %d", time); # behaves
like printf()
syslog('debug', "Bug found in $0");
syslog('alert', "Something's wrong!");
# syslog() interprets %m as an error string,
# so this message appends
# Problem: No such file or directory
# to the log file.
if (!open(FILE, "/tmp/no_file_here")) {
    syslog('warning', 'Problem: %m');
}
# Close the log file
closelog();
```

The `openlog()` function prepares your computer for log messages sent by the program. In the *logfile* program:

- The first argument (`MyLog`) causes all messages sent to be tagged `MyLog`.
- The second argument (`cons, pid`) causes messages to be sent to the system console if they can't be appended to the log file (`cons`) and tags all messages with the process ID (`pid`). Process IDs are described in Chapter 9, Lesson 2.
- The third argument (`user`) indicates which log file should be used. That will almost always be `user`.

Every `syslog()` call appends another message to the log file. The first argument to `syslog()` indicates the priority of the message; the second argument contains the message itself, which can contain `printf()`-style fields. (That's why the notice message includes the current time.) In addition, `%m` has a special meaning

for syslog(): It contains the system-error string.

Finally, `closelog()` closes the log entry.

You won't see any output when you run *logfile*. But if you look at the end of the user log file on a workstation named Jupiter, you might see something like this:

```
RESULT Oct 5 01:31:52 jupiter MyLog[19548]: Some information
Oct 5 01:31:52 jupiter MyLog[19548]: A notice, at time
882871112
Oct 5 01:31:52 jupiter MyLog[19548]: Bug found in ./logfile
Oct 5 01:31:52 jupiter MyLog[19548]: Something's wrong!
Oct 5 01:31:52 jupiter MyLog[19548]: Problem: No such file or
directory
```

Pod: Plain Old Documentation

If you look inside *LIB/English.pm*, you'll see some strange-looking text:

```
=head1 NAME
English - use nice English (or awk) names for ugly
punctuation variables
=head1 SYNOPSIS
use English;
...
if ($ERRNO =~ /denied/) { ... }
=head1 DESCRIPTION
This module provides aliases for the built-in variables whose
names no one seems to like to read. Variables with side
effects
that get triggered just by accessing them (like $0) will
still
be affected.
For those variables that have an B<awk> version, both long
and short English alternatives are provided. For example,
the C<$/> variable can be referred to either $RS or
$INPUT_RECORD_SEPARATOR if you are using the English module.
See L<perlvar> for a complete list of these.
=cut
```

Like many modules, English contains both Perl code *and* the documentation for that code. The above documentation is pod text, a simple ASCII format (designed by Larry Wall) that also happens to be valid Perl, so it can be freely sprinkled throughout any Perl program.

In addition to modules, the documentation for Perl itself is available as pod files: *perlvar.pod*, *perlfunc.pod*, *perlpod.pod*, and so on. You'll find copies in the pod subdirectory of the Perl distribution. Both pod files and modules with embedded pod text can be converted into other formats with any of the following programs, bundled with Perl:

- ***pod2man*: Man pages**
- ***pod2html*: HTML, viewable with any Web browser**
- ***pod2latex*: LaTeX**
- ***pod2text*: Plain ASCII text**

Details about the pod format can be found (as pod text, of course) in the *perlpod* documentation bundled with Perl.

Chapter 8, Perl 5 Interactive Course

LESSON 3

MAKING MODULES

For a package to aspire to modulehood, it should respect the caller's namespace. Let's take a look at the beginning of the `Time::Local` module.

```
package Time::Local;
require 5.000;
require Exporter;
use Carp;

@ISA = qw(Exporter);
@EXPORT = qw(timegm timelocal);
```

The `@ISA` suggests that `Time::Local` is a class, but it's not. Here's what's going on: First, the `Time::Local` module makes sure that the user is running Perl 5 (or higher). Then the `Exporter` module is `require()`d and its methods are inherited (via `@ISA`). But the important line is the last, which makes the `timegm()` and `timelocal()` functions available to the calling program by including them in the `@EXPORT` array.

Exporter

Modules use `Exporter` to make their variables and subroutines available to programs--that's why you can call `timelocal()` directly from your program rather than having to qualify it with the full package name:

```
Time::Local::timelocal()
```

So what methods are in the `Exporter` module? Just two: `export()` and `import()`. You might never use them yourself, but they help illustrate what happens when you `use()` a module.

`export(@array)` makes the `@array` of module variables and subroutines available for importing by a calling program. It's used inside modules.

`import(@array)` incorporates the `@array` of exported variables and subroutines into the current namespace. It's used by `use()`.

When `use()` is invoked, it first `require()`s the `.pm` file and then `import()`s any variables or subroutines listed in `@EXPORT`. If you want to export a reserved name (such as any Perl built-in function), you can put that in the `@EXPORT_OK` array, which lets the calling program explicitly import the name.

(The built-in function isn't lost forever; you can always access Perl's original definition by prepending `CORE::` to the function name.)

The `Film` module, shown in Listing 8-10, exports `$title`, `summarize()`, and `advertise()` (by placing them in `@EXPORT`). But `close()` is already defined in Perl, so it's placed into `@EXPORT_OK`.

Listing 8-10 film.pm: A sample module

```
package Film;
require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw($title summarize advertise);
@EXPORT_OK = qw(close);
$title = "A Fish Called Wanda";
sub summarize { "a spirited farce about a bank heist in
England." }
sub advertise { "See this movie!" }
sub close { print "It's playing at a cinema near you!" }
1;
```

The *usefilm* program, shown in Listing 8-11, uses the *Film* module twice: once to import all the names in `@EXPORT` and once specifically to get the new `close()`, which overrides Perl's built-in `close()`.

Listing 8-11 *usefilm*: Using `use()` to import the contents of `@EXPORT` and `@EXPORT_OK`

```
#!/usr/bin/perl -w
use Film;
print $title, " is ";
print summarize(), "\n";
print advertise(), "\n";
use Film 'close';
close();
```

When run, *usefilm* prints the following:

```
% usefilm
```

RESULT A Fish Called Wanda is a spirited farce about a bank heist in England.

```
See this movie!
It's playing at a cinema near you!
```

usefilm is equivalent to the following:

```
BEGIN {
require Film;
import Film;
}
```

Here's what that does, step by step:

1. Perl finds the file called *Film.pm* in its LIB directories and loads it.
2. It sees that *Film* inherits from *Exporter*.
3. `import Film` invokes `Exporter::import`, which makes the variables and functions named in `@EXPORT` available to the current package.

All this happens when the program is compiled--before any statement is executed. That's because BEGIN and END, like pragmas, are compiler directives.

The BEGIN and END Declarations

BEGIN BLOCK tells Perl to execute **BLOCK** before running the program.

END BLOCK tells Perl to execute **BLOCK** after running the program.

BEGIN blocks are executed during compilation, so all BEGIN blocks take effect before any "regular" statements.

If there are multiple BEGIN blocks in your program, they are executed in order. If there are multiple END blocks, they are executed in reverse order when your program finishes (including after any exit() or die()). Suppose you want to modify @INC so that it looks in /users/perluser/my_perl_library. This won't do much:

```
push(@INC, "/users/perluser/my_perl_library");
```

because use() performs its magic at compile time. You need to wrap it in a BEGIN block:

```
BEGIN { push(@INC, "/users/perluser/my_perl_library"); }
```

AutoLoader

Because BEGINs are executed at compile time, the entire Film module is loaded into memory. That's not a problem if the module is small, or if you're sure that the calling program uses all the module's subroutines. But for large modules, you might want to keep the size of the running program small by loading in only those parts of the module needed. That's what the AutoLoader module does. (There's also a module by Jack Shirazi called SelfLoader, which is better suited for standalone modules. It's bundled with Perl as of version 5.002; if you have an earlier version of Perl, you'll have to grab a copy from one of the CPAN sites listed in Appendix B.

Automatically loaded modules are organized differently than regular modules. Listing 8-12 shows how *RegFilm.pm* could be converted to an automatically loaded module.

Listing 8-12 *autoFilm.pm*: Using Autoloader

```
package AutoFilm;
use Exporter;
use AutoLoader;
@ISA = qw(Exporter AutoLoader);
@EXPORT = qw($title summarize advertise);
@EXPORT_OK = qw(close);
1;
__END__
$title = "A Fish Called Wanda";
sub summarize { "a spirited farce about a bank heist in
England." }
sub advertise { "See this movie!" }
```

```
sub close { print "It's playing at a theater near you!"}
```

Your program behaves the same regardless of whether it uses the autoloading `AutoFilm` or the loaded-during-compilation `Film`. Programmers using a module shouldn't have to know how a module has been loaded; they just have to know what the module does.

As shown, the `AutoFilm` module isn't quite ready for automatic loading. It needs to be split into separate files, one per function. That's done with another module: `AutoSplit`.

AutoSplit

You can split your module into bite-sized chunks suitable for automatic loading with `AutoSplit`. Then, when your program invokes a function, the `AUTOLOAD()` subroutine (inherited from the `AutoLoader` module) loads the function from the `LIB/auto` directory.

The `autosplit()` Function

`autosplit(MODULE, DIRECTORY)` creates a subdirectory inside ***DIRECTORY*** and splits the functions in ***MODULE*** into individual files in that new directory.

You can `autosplit()` the `AutoFilm` module in one line (well, one-and-a-half):

```
% perl -e "use AutoSplit;
autosplit('/usr/local/lib/perl5/AutoFilm.pm', *
'/usr/local/lib/perl5/auto')"
```

This places each subroutine in its own `.al` file:

```
writing auto/AutoFilm/advertize.al
writing auto/AutoFilm/summarize.al
writing auto/AutoFilm/close.al
```

These `.al` files don't contain any surprises: just the subroutine definitions followed by the `1`; that ensures the last statement is `TRUE`. Autosplitting also creates an `.ix` file containing the subroutine declarations.

```
/usr/local/lib/perl5/auto/AutoFilm/close.al
# NOTE: Derived from /usr/local/lib/perl5/AutoFilm.pm.
Changes made here will be lost.
package AutoFilm;
sub close { print "It's playing at a theater near you!"}
1;
```

See issue 6 of *The Perl Journal* for more details about autoloading.

Chapter 8, Perl 5 Interactive Course

LESSON 4

GETOPT AND ASSERT

The command line is the first aspect of your program experienced by users. Whether it's inviting or confusing is up to you--do you force a particular order of arguments on the command line? Do you ensure that mandatory arguments are present and avoid error messages if optional arguments aren't? Even though it takes time away from creating the "meat" of your program, effective argument parsing is a must for any program that you intend to be used widely.

Some programs might need several arguments on the command line. Some arguments should be mandatory, some optional, and some repeatable. Perhaps the order of the arguments doesn't matter, or perhaps it does, or perhaps it does for some of the arguments but not all.

When you write a program for yourself, you can insist upon any argument order you want. After all, it's your program--you know how to use it. But if other people use it, they might not know the ordering of the arguments. "Does height come before width, or vice versa?"

Enter the Getopt module set, which provides some mechanisms for parsing command-line arguments. This lesson also covers the library *file assert.pl*, which lets you stipulate conditions that your program must fulfill to stay alive.

Getopt

There are many different conventions for supplying command-line arguments to a program. Some programs expect a series of single-character switches. For example,

```
% program -x 10 -y 16
```

conveys to the program that the value of x is 10 and the value of y is 16.

The Getopt module set contains two modules: Getopt::Std and Getopt::Long.

Getopt::Std

The Getopt::Std module contains two functions: getopt() and getopt(), which interpret single-character switches. The getopt() function accesses the -x and -y switches on the command line and stores them in the variables \$opt_x and \$opt_y.

The getopt() and getopt() Functions

getopt(CHARACTERS) processes *CHARACTERS* as single-character command-line switches; each character takes one argument.

`getopts(CHARACTERS)` processes *CHARACTERS* as switches taking arguments; only *CHARACTERS* followed by colons take arguments.

Both functions let the user provide the switches in any order.

Both `getopt()` and `getopts()` place the switch results in a scalar that includes the name of the switch: `$opt_switch`. So,

```
getopt('pq');
```

stores the value following `-p` in `$opt_p`, and the value following `-q` in `$opt_q`.

`allargs` (Listing 8-13) uses `getopt()` to parse three switches with arguments.

Listing 8-13 `allargs`: Using `getopt()` to parse switches with arguments

```
#!/usr/bin/perl -wl
use Getopt::Std;
getopt('isN'); # Look for three switches: i, f, and N.
if ($opt_i) { print "input: $opt_i" }
if ($opt_s) { print "speed: $opt_s" }
if ($opt_N) { print " Name: $opt_N" }
% allargs -i blot -N Winston -s 55
```

RESULT input: blot

```
speed: 55
Name: Winston
```

The user doesn't have to provide all switches; if a switch isn't supplied, the corresponding `$opt_` variable ends up undefined. You can also combine the switches and arguments:

```
% allargs -NWinston -s55
```

RESULT speed: 55

```
Name: Winston
```

If you want some of your switches to take optional arguments, use the `getopts()` function instead, from the same module (Listing 8-14).

Listing 8-14 `someargs`: Using `getopts()` to parse switches, some of which have arguments

```
#!/usr/bin/perl -wl
use Getopt::Std;
getopts('ti:N:'); # i and N take arguments, t doesn't.
if ($opt_i) { print " input: $opt_i" }
if ($opt_N) { print " Name: $opt_N" }
if ($opt_t) { print "toggle: $opt_t" }
% someargs -t -N Winston -i blot
input: blot
```

Name: Winston
toggle: 1

Getopt::Long

(Module written by John Vromans)

getopt() and getopt3() are useful, but not very general:

- All arguments have to be single-character switches.
- You can't repeat arguments.

The Getopt::Long module contains the GetOptions() function, which provides these features and more.

The GetOptions() Function

GetOptions(*ARRAY*) parses the command-line arguments according to the *ARRAY* of strings, each describing a rule for processing one argument.

Table 8-1 shows some sample strings.

Table 8-1 Sample GetOptions() strings

String	Meaning
"arg"	Doesn't take an argument.
"arg!"	Accepts either -arg or -noarg.
"arg=i"	-arg takes a mandatory integer.
"arg=s"	-arg takes a mandatory string.
"arg:s"	-arg takes an optional string.
"arg:f"	-arg takes an optional floating-point number.
"arg=s@"	Accepts multiple -arg strings; places the results in @opt_arg.
"arg urg:i"	Accepts an optional -arg integer; -urg is another name for -arg.

GetOptions() has even more features than those shown in Table 8-1. See the manual page for GetOptions (if your system has it) or read the pod documentation in *LIB/Getopt/Long.pm*, which has the same information.

The *longopts* program, shown in Listing 8-15, supplies GetOptions() with four rules for parsing arguments.

Listing 8-15 *longopts*: Using GetOptions() to perform "extended" parsing

```
#!/usr/bin/perl -l
use Getopt::Long;
GetOptions("car:i", # the car switch takes an optional
integer
"name|nom|namen=s", # name is nom is namen: a mandatory
string
"angry!", # angry or noangry
"kid=s@"); # parse multiple -kid strings
if ($opt_name) { print "Name: $opt_name" }
if ($opt_angry) { print "Angry!" } else { print "Not angry!"
}
if (@opt_kid) { print "Kids: ", join(' ', @opt_kid) }
if (defined($opt_car)) {
if ($opt_car) { print "Number of cars: $opt_car" }
else { print "Has a car." }
}
```

The first rule, `car:i`, looks for a switch named `-car`, which can be followed by an integer but doesn't have to be. The value is stored in `$opt_car`, if provided; `$opt_car` is set to an empty string otherwise.

The second rule, `name|nom|namen=s`, looks for any of three switches: `-name`, `-nom`, or `-namen`, followed by a string. That string is stored in `$opt_name`, regardless of which switch was used.

The third rule, `angry!`, looks for either of two switches: `-angry` or `-noangry`. If `-angry` is supplied, `$opt_angry` is set to 1. If `-noangry` is supplied, `$opt_angry` is set to 0.

The fourth rule, `kid=s@`, looks for multiple `-kid` switches. The values are placed into the array `@opt_kid`, in order.

Here, *longopts* is invoked with the `-nom` flag, the negated `-noangry` switch, the `-car` switch (without the optional integer), and two `-kid` switches:

```
% longopts -nom Jean -noangry -kid Bob -car -kid Connie
```

RESULT Name: Jean

Not angry!

Kids: Bob, Connie

Has a car.

In this example, the `-car` switch is given an integer and an extra `-kid` switch is added:

```
% longopts -kid Yog -namen Johan -angry -kid Bob -car 1 -kid
Connie
```

RESULT Name: Johan

Angry!

Kids: Yog, Bob, Connie

Number of cars: 1

Assert

When you *assert* a statement, you're saying, "This statement better be TRUE. If it's not, something has gone horribly wrong, so exit()." Assertions are helpful for debugging. For instance, if you know that \$card should be positive at some point during your program, you can assert() that condition:

```
assert($card > 0);
```

If \$card isn't greater than 0 when that statement is reached, your program will immediately halt.

assert.pl

(Library file written by Tom Christiansen)

The assert() Function

assert(*CONDITION*) stops the program if *CONDITION* is false.

Listing 8-16, featuring *ensure*, shows assert() in action:

Listing 8-16 *ensure*: Using assert() to ensure that a condition is TRUE

```
#!/usr/bin/perl
require 'assert.pl';
$password = $ARGV[0];
assert('password eq "shibboleth"');
print "You may enter, friend.\n";
```

If the user provides the wrong password on the command line, *ensure* dies when the assert() is reached, printing the expression that failed (as well as some other information, such as the line number, package name, and filename):

```
% ensure sesame
```

```
RESULT panic: ASSERTION BOTCHED: $password eq "shibboleth"
```

(Whoops...there goes the secret \$password.)

You could use assert() to test the result of an open(), as in Listing 8-17.

Listing 8-17 *surefile*: Using assert() to ensure that a file was opened successfully

```
#!/usr/bin/perl
require 'assert.pl';
assert('open(FILE, "/bet_this_file_does_not_exist")');
print FILE "HMMMMMM...I guess it DOES exist!";
```

Because open() returns 0 on failure (that's why the **or die** form works), you can assert() the value of open().

```
% filesure
```

```
RESULT panic: ASSSSERTION BOTCHED: open(FILE,  
"/bet_this_file_does_not_exist")
```

Chapter 8, Perl 5 Interactive Course

LESSON 5

INSPECTING AND TESTING

In this lesson, we'll be covering a number of modules that inspect your Perl environment and test files (for existence and permission bits) and programs (for speed). The six modules covered in this lesson, which all loosely pertain to the theme of learning about your Perl environment, such as your program's current directory and how Perl was configured for your computer.

Config

Installing Perl on a computer can be tricky. The installer has to make lots of choices, ranging from choosing a C compiler, to determining how many bits the `rand()` function uses, to the location of C libraries and a destination directory for the library files. All this information is stored in the `Config` module, which is generated automatically by Perl during installation.

Config

`Config` is unusual among modules because it doesn't provide any functions or methods. Instead, it exports a single hash, `%Config`, containing the configuration options (there are over 400!) used to compile Perl on your system.

Many of the configuration options are obscure. If you want to see all of them, iterate through `%Config` with `each()`. Listing 8-18 is a program that displays some of the more interesting key-value pairs.

Listing 8-18 *showconf*: Using `%Config` to access Perl's configuration

```
#!/usr/bin/perl -wl
use Config;
print "I'm running on $Config{osname}";
print "Perl was configured for this platform by
$Config{cf_by}";
print "The byte ordering on this machine is
$Config{byteorder}";
if ($Config{osname} eq `msdos`) {
print "Libraries were installed in $Config{lib}";
} else {
print "Libraries were installed in $Config{installprivlib}";
}
```

Naturally, *showconf* will display different results on different systems. Here's what it might say on an

MS-DOS machine:

```
% showconf
```

RESULT I'm running on msdos

```
Perl was configured for this platform by darrylo
Libraries are stored in c:/perl5/lib
The byte ordering on this machine is 1234
```

Env

Env is another example of a module that doesn't provide any new functions. It merely makes access to the %ENV hash more convenient.

Env

(Module written by Chip Salzenberg)

In Chapter 5, you learned that your current environment is stored in %ENV. The Env module converts the contents of %ENV to plain variables, saving you some typing: Instead of accessing the user's path through \$ENV{PATH}, you can use plain \$PATH instead. Those two variables are linked to one another; you can't change one without changing the other.

The *useenv* program in Listing 8-19 uses the Env and Config modules to modify the user's path.

Listing 8-19 *useenv*: Using the Env module to turn environment variables into Perl variables

```
#!/usr/bin/perl -wl
use Env;
use Config;
if ($Config{osname} eq `msdos`) {
    $PATH .= `C:\PERL5;`; # Add C:\PERL5 to the DOS path
} else {
    $PATH = "$PATH:"; # Add . to the UNIX path
}
print "$PATH is the same as $ENV{PATH}";
```

The *use Env* statement links the contents of %ENV to corresponding scalar variables:

```
% useenv
```

RESULT /usr/bin:/usr/local/bin:. is the same as
/usr/bin:/usr/local/bin:.

Cwd

The Cwd module gives you access to the current working directory of your program. By default, that's whatever directory you were in when you ran the program, but it might change if you *chdir()* to a different directory.

Cwd

(Module written by Brandon Allbery and John Bazik)

Cwd provides three functions: the slow but safe `getcwd()`, the fast but unsafe `fastcwd()`, and a new `chdir()` function that sets `$ENV{PWD}` (because Perl's built-in `chdir()` doesn't).

Listing 8-20, *currrdir*, demonstrates how `$ENV{PWD}` can be fooled if Cwd's `chdir()` isn't used.

Listing 8-20 *currrdir*: Using Cwd to access the current directory

```
#!/usr/bin/perl -l
use Cwd;
$dir = fastcwd;
print "You're in $dir.";
chdir "..";
$dir = getcwd;
print "You're in $dir.";
print "But %ENV thinks you're in $ENV{PWD}.";
use Cwd 'chdir'; # Keep $ENV{PWD} up to date
chdir "/tmp";
$dir = getcwd;
print "You're in $dir.";
print "%ENV thinks you're in $ENV{PWD}.";
```

Here's what would happen if you ran *currrdir* from the `/perl5` directory:

```
% currrdir
```

```
RESULT You're in /perl5.
```

```
You're in /.
```

```
But %ENV thinks you're in /perl5.
```

```
You're in /tmp.
```

```
%ENV thinks you're in /tmp.
```

In case you're wondering why `fastcwd()` is unsafe, it's because `fastcwd()` itself uses `chdir()` and it might (in very unusual circumstances) `chdir()` out of a directory and be unable to `chdir()` back.

Benchmark

The Benchmark module lets you measure the speed of a chunk of Perl code.

That's not as simple as merely timing the code--you could do that with a couple of calls to `time()`. Most modern computers can do more than one thing at a time; if your computer is busy running other tasks in the background or serving multiple users simultaneously, that's going to affect how much attention the CPU can devote to your program. So you want to look at the amount of time the CPU devotes to your program, not the "real-time" duration. And you don't want to run the program just once, but hundreds or thousands of times, and determine the average time.

Benchmark

(Module written by Jarkko Hietaniemi and Tim Bunce)

The Benchmark module provides a Benchmark class used to determine the amount of system and user resources used by your code. To benchmark a chunk of code, create a new Benchmark before the code starts and another new Benchmark after the code finishes. Then use the `timestr()` and `timediff()` functions (exported by the module) to create a human-readable string containing the amount of user, system, and CPU time used by the code. The code in Listing 8-21 shows how this is done.

Listing 8-21 *longshot*: Using Benchmark to time programs

```
#!/usr/bin/perl -wl
use Benchmark;
srand (time ^ ($$ + ($$<15)));
$start = new Benchmark;
for ($i = 0; $i < 1000000; $i++) {
    if (int(rand(1000000)) == 777) { print "One in a million
    chance!\n" }
}
$end = new Benchmark;
print "That took ", timestr(timediff($end, $start));
```

longshot picks 1 million random numbers. The `$start` and `$end` objects contain the amount of system time and user time used by the *for* loop in between.

```
% longshot
```

```
RESULT One in a million chance!
```

```
That took 22 secs (20.73 usr 0.07 sys = 20.80 cpu)
```

The total execution time was 22 seconds, but the code required only 20.8 seconds of CPU time. (That's the important number.)

The Benchmark module provides some additional functions: `timethis()`, `timethese()`, and the lower-level function `timeit()`.

- **`timethis(NUMBER, CODE)` executes *CODE* for *NUMBER* iterations.**
- **`timethese(NUMBER, HASH)` executes the chunks of code named in *HASH* for *NUMBER* iterations.**
- **`timeit(NUMBER, CODE)` executes *CODE* for *NUMBER* iterations.**

timers, shown in Listing 8-22, uses these functions.

Listing 8-22 *timers*: Using Benchmark's timethis(), timethese(), and timeit() functions

```
#!/usr/bin/perl -wl
use Benchmark;
sub sqrt_pi { sqrt(4 * atan2(1,1)) }
timethis(1000000, "rand");
timethese(100000, { "Random" => `rand`, "Listing" => `ls
/tmp `});
$t = timeit(1000000, `sqrt_pi`);
print "One million sqrt_pi() calls took ", timestr($t);
% timers
timethis 1000000: 5 secs ( 5.27 usr 0.00 sys = 5.27 cpu)
Benchmark: timing 10000 iterations of Listing, Random...
Listing: 684 secs ( 6.80 usr 147.98 sys + 55.55 cusr 370.07
csys = 580.40 cpu
)
Random: 0 secs ( 0.07 usr 0.00 sys = 0.07 cpu)
(warning: too few iterations for a reliable count)
One million sqrt_pi() calls took 61 secs (44.82 usr 0.05 sys
= 44.87 cpu)
```

As you can see, ``ls /tmp``, a filesystem operation, takes quite a bit more time than anything else.

Sys

The Sys module set contains utilities for gathering information about your computer. At present, it contains two modules: Sys::Syslog (which was discussed in Lesson 2) and Sys::Hostname.

UNIX

Sys::Hostname

(Module written by David Sundstrom)

Every UNIX computer has a hostname that identifies it to other computers. Unfortunately, there's no single technique for determining the hostname that works on all (or even most) computers. Enter the Sys::Hostname module, which rolls several techniques into a single function: `hostname()`, demonstrated in Listing 8-23.

Listing 8-23 *whereami*: Using Sys::Hostname to determine the name of the computer

```
#!/usr/bin/perl -wl
use Sys::Hostname;
print "I'm on ", hostname;
```

If you ran *whereami* on a computer named zanzibar in the waite.com domain, you might see this:

```
% whereami
```

```
RESULT I'm on zanzibar.waite.com
```

dumpvar

The *dumpvar* library file displays all the variables in a package. You could iterate through the package variables yourself using the *dumpvar* program from Chapter 5, Lesson 7, but the library file is a bit nicer: It indents hash key-value pairs and it sorts the keys for you.

dumpvar.pl

`dumpvar(PACKAGE)` function pretty-prints the variables in *PACKAGE*, as shown in Listing 8-24.

Listing 8-24 *showvars*: Using `dumpvar()`

```
#!/usr/bin/perl
package Solo;
$alone = 1;
%solitary = (only => 'one', uni => 'single');
package main;
require 'dumpvar.pl';
dumpvar('Solo');
```

showvars uses `dumpvar()` to print the variables in the Solo package:

```
% showvars
```

```
RESULT %solitary = {
  only 'one'
  uni 'single'
}
$alone = '1';
```

Andreas König's Devel::Symdump (available from the CPAN) is a more sophisticated version of *dumpvar.pl* and will probably supersede it in a future version of Perl.

Chapter 8, Perl 5 Interactive Course

LESSON 6

FILES

This lesson describes several modules for finding files, navigating filesystems, and controlling filehandles. The modules covered in the lesson make up a useful set of utilities for manipulating files and filesystems.

File

The File module set provides four modules for inspecting and finding files.

File::Find

The Find module exports two functions: `find()` and `finddepth()`.

- **`find(SUBROUTINE, ARRAY)` searches recursively through the *ARRAY* of directories. For each file in those directories, the *SUBROUTINE* is executed.**
- **`finddepth(SUBROUTINE, ARRAY)` behaves like `find()`, but performs a depth-first search.**

Both `find()` and `finddepth()` make certain variables available to the *SUBROUTINE*:

- **`$File::Find::dir` contains the current directory during the search.**
- **`$_` contains the current filename.**
- **`$name` contains `$File::Find::dir/$_`.**

Listing 8-25 uses `File::Find`.

Listing 8-25 *findcore*: Using `File::Find` to find files named `core`

```
#!/usr/bin/perl -wl
use File::Find;
sub findcore { print $name if ($_ eq 'core') }
find(&findcore, '/');
% findcore
/users/perluser/core
/users/perluser/project/core
```

Remember how the `@ISA` hierarchy is searched depth first? `finddepth()` searches depth-first through

your filesystem. It'll usually behave the same as `find()`; it just encounters files in a different order. *delzeros* (Listing 8-26) demonstrates `finddepth()` as well as passing in an anonymous subroutine.

Listing 8-26 *delzeros*: Using `finddepth()`

```
#!/usr/bin/perl -w
use File::Find;
finddepth sub { -z && unlink $_ }, ".";
```

delzeros deletes all empty files in the current directory.

`File::Find` is a Perl implementation of the UNIX *find* command. If you have a particular UNIX *find* invocation that you wish to convert to Perl, consider using the *find2perl* program, discussed in Chapter 14, Lesson 4.

File::CheckTree

(Module written by Brandon S. Allbery)

The `File::Checktree` module exports the `validate()` subroutine, which accepts a multiline string of file tests to be applied to files and directories. Each line can contain a filename, some whitespace, and a file test (see Chapter 3, Lesson 6). The file test can be

- **Negated, by preceding it with an `!`.**
- **Followed by `|| warn ...` or `|| die ...`, in which case `$file` will contain the name of the file.**
- **`cd`, which causes succeeding filenames to be interpreted relative to that directory.**
- **Combined with other tests: `-rw` tests both read and write access.**

`validate()` (Listing 8-27) returns the number of tests that fail.

Listing 8-27 *checksys*: Using `File::CheckTree` to perform file tests

```
#!/usr/bin/perl
use File::CheckTree;
$warnings = validate( q{/tmp -d || warn "No $file!\n";
/tmp -rwx
/usr/bin cd
perl -x || warn "No executable perl.\n"});
$warnings = 0 unless $warnings;
print "There were $warnings warnings.\n"
```

The first line of the `validate()` string checks to make sure that `/tmp` is a valid directory. If it's not, a warning occurs (for example, *No /tmp!*). The second line checks `/tmp` to make sure it's readable, writable, and executable. If not, `$warnings` is incremented by 1. The third line changes the `validate()` directory to `/usr/bin`, affecting the path for the subsequent lines. The fourth line tests to make sure `/usr/bin/perl` is executable, and `warn()`s if it is not.

File::Basename

(Module written by Charles Bailey)

The File::Basename module provides three functions: `fileparse()`, which splits a pathname into pieces; `basename()`, which extracts the filename from directory/filename; and `dirname()`, which extracts the directory from directory/filename. Listing 8-28 shows how this module is used.

Listing 8-28 *getnames*: Using File::Basename to extract the pieces of a pathname

```
#!/usr/bin/perl -wl
use File::Basename;
print "Dirname is ",
dirname("/usr/local/lib/perl5/English.pm");
print "Basename is ",
basename("/usr/local/lib/perl5/English.pm");
($name, $path, $extension) =
fileparse("/usr/local/lib/perl5/English.pm", `\\.*.`);
print "Name: $name";
print "Path: $path";
print "Extension: $extension";
```

getnames splits up the string `/usr/local/lib/perl5/English.pm` into directory, basename, filename, path, and extension:

```
% getnames
```

```
RESULT Dirname is /usr/local/lib/perl5
```

```
Basename is English.pm
```

```
Name: English
```

```
Path: /usr/local/lib/perl5
```

```
Extension: .pm
```

Because operating systems have different schemes for representing pathnames, you choose one with `fileparse_set_fstype()`:

`fileparse_set_fstype("UNIX")` For UNIX-flavored systems

`fileparse_set_fstype("MSDOS")` For MS-DOS systems

`fileparse_set_fstype("MacOS")` For Macintosh systems

`fileparse_set_fstype("VMS")` For VMS systems

If you don't call `fileparse_set_fstype()`, Perl calls it for you with the appropriate value.

FileHandle

The FileHandle module has two purposes. First, it permits a quasi-object-oriented interface to filehandle access methods, providing functions that correspond to many of the per-filehandle special variables. Second, the `cacheout()` function lets you keep more files open than the operating system would normally permit.

FileHandle

The FileHandle module provides the functions shown in Table 8-2, each of which does the same as `select()`ing a filehandle, setting the corresponding special variable, and `select()`ing back to the original filehandle.

Table 8-2 FileHandle functions corresponding to per-filehandle special variables

Special Variable	English Name	Corresponding Method
<code>\$ </code>	<code>\$OUTPUT_AUTOFLUSH</code>	<code>autoflush()</code>
<code>\$,</code>	<code>\$OFS</code> , <code>\$OUTPUT_FIELD_SEPARATOR</code>	<code>output_field_separator()</code>
<code>\$\</code>	<code>\$ORS</code> , <code>\$OUTPUT_RECORD_SEPARATOR</code>	<code>output_record_separator()</code>
<code>\$/</code>	<code>\$RS</code> , <code>\$INPUT__RECORD_SEPARATOR</code>	<code>input_record_separator()</code>
<code>\$.</code>	<code>\$NR</code> , <code>\$INPUT_LINE_NUMBER</code>	<code>input_line_number()</code>
<code>\$%</code>	<code>\$FORMAT_PAGE_NUMBER</code>	<code>format_page_number()</code>
<code>\$=</code>	<code>\$FORMAT_LINES_PER_PAGE</code>	<code>format_lines_per_page()</code>
<code>\$-</code>	<code>\$FORMAT_LINES_LEFT</code>	<code>format_lines_left()</code>
<code>\$~</code>	<code>\$FORMAT_NAME</code>	<code>format_name()</code>
<code>^</code>	<code>\$FORMAT_TOP_NAME</code>	<code>format_top_name()</code>
<code>\$:</code>	<code>\$FORMAT_LINE_BREAK_CHARACTERS</code>	<code>format_line_break_characters()</code>
<code>^</code>	<code>\$FORMAT_FORMFEED</code>	<code>format_formfeed()</code>

The *ezfile* program demonstrates two FileHandle functions: `autoflush()` and `output_record_separator()`, as shown in Listing 8-29.

Listing 8-29 ezfile: Using FileHandle functions instead of the special variables

```
#!/usr/bin/perl -w
use FileHandle;
open (FILE, ">ezfile.out") or die "Can't write ezfile.out: $!";
# same as select()ing FILE, setting $| to 1, and then
select()
```

```
# STDOUT again (presuming that was the previous default
filehandle)
FILE->autoflush();
# Ditto for $\.
FILE->output_record_separator("\r\n");
print FILE "This will be flushed to the file immediately.";
```

To flush STDOUT, just say

```
STDOUT->autoflush();
```

Most operating systems have some maximum number of file descriptors that can be opened simultaneously. Often, this number is 32 or 64; if you want to have more, you can circumvent that limit with the `cacheout()` function. Use `cacheout()` where you would normally use `open()` for writing, as seen in Listing 8-30.

Listing 8-30 *cacheout*: Using FileHandle to open a large number of files

```
#!/usr/bin/perl -w
use FileHandle;
foreach $i (1..100) {
    $path = "/tmp/wheelie.$i";
    cacheout($path); # open $path for writing
    print $path "This is file $i.";
}
```

Flush

The *flush.pl* library file lets you explicitly flush a file without bothering with `$|` or `close()` or even `FILE->autoflush()`.

flush.pl

The *flush.pl* contains two functions: `flush(FILE)` and `printflush(FILE, EXPRESSION)`. `flush()` writes any data remaining in *FILE*'s buffer; `printflush()` prints the *EXPRESSION* and then flushes the buffer, as shown in Listing 8-31.

Listing 8-31 *flusher*: Using *flush.pl* to avoid buffer delays

```
#!/usr/bin/perl -w
require "flush.pl";
open (FILE, ">flusher.out") or die "Can't write flusher.out:
$! ";
print FILE "This will be flushed after the next statement.
\n";
flush(FILE);
printflush(FILE, "This will be flushed immediately.");
```



```
close FILE;
```

stat

Like the `Env` module, the *stat* library file doesn't provide any functions. It merely places the values returned by `stat()` into plain scalars.

stat.pl

The names of those scalars correspond to the UNIX `stat` structure: `$st_dev`, `$st_ino`, `$st_mode`, `$st_nlink`, `$st_uid`, `$st_gid`, `$st_rdev`, `$st_size`, `$st_atime`, `$st_mtime`, `$st_ctime`, `$st_blksize`, `$st_blocks`. The function that performs this translation is `Stat()`, demonstrated in Listing 8-32.

Listing 8-32 *statvars*: Using *stat* to access file statistics as plain variables

```
#!/usr/bin/perl -w
require "stat.pl";
Stat("statvars");
print "This file is $st_size bytes long.\n";
% statvars
```

RESULT This file is 100 bytes long.

stat.pl may eventually find its way into the `Sys` module set, after which you'll be able to use `Sys::Stat`.

Chapter 8, Perl 5 Interactive Course

LESSON 7

TEXT, SEARCH, AND MATH

This lesson explores three module sets: Text, Search, and Math.

Text

The Text module contains four modules: Text::Abbrev, Text::Soundex, Text::Tabs, Text::Wrap, and Text::ParseWords.

Text::Abbrev

The Abbrev module lets you take a group of strings and find minimal, mutually exclusive abbreviations for them. It's useful for programs in which you want to let users type short versions of commands: allowing *li* or *list* for listing, but disallowing *l* if it might also mean *logout*.

The abbrev() function fills a hash with substring-string pairs mapping abbreviations to expansions, as demonstrated by *breviate*, in Listing 8-33.

Listing 8-33 *breviate*: Using Abbrev to create a hash of abbreviations

```
#!/usr/bin/perl -w
use Abbrev;
abbrev(*hash, qw( mail more finger edit emacs ));
while (($key, $value) = each %hash) {
    print "$key expands to $value \n";
}
```

Because *mail* and *more* both begin with *m*, there's no key for *m* in the hash, but there are keys for *ma* and *mo*, as well as all possible longer abbreviations (*mor*, *mai*, *more*, and *mail*).

```
% breviate
```

```
RESULT em expands to emacs
```

```
edi expands to edit
ema expands to emacs
more expands to more
mail expands to mail
fing expands to finger
emac expands to emacs
edit expands to edit
f expands to finger
mai expands to mail
emacs expands to emacs
ed expands to edit
```

```
finge expands to finger
finger expands to finger
mo expands to more
fin expands to finger
ma expands to mail
mor expands to more
fi expands to finger
```

If you're daring, you could combine this with AUTOLOAD() to create a module that corrects typos in subroutine names. (See the exercise for Chapter 9, Lesson 5.)

Text::Soundex

(Module written by Mike Stok)

Soundex is a simple algorithm that maps words (such as last names) to brief codes that correspond very roughly to the pronunciation of the word. Every Soundex code is four characters long: a letter (the first letter of the word) and three digits. The soundex() function returns the codes, as shown in Listing 8-34.

Listing 8-34 *announce*: Using Text::Soundex to compare pronunciations

```
#!/usr/bin/perl -wl
use Text::Soundex;
print soundex("Perl");
print soundex("Pearl");
print soundex("Wall");
if ($ARGV[1]) {
    if (soundex($ARGV[0]) eq soundex($ARGV[1])) {
        print "$ARGV[0] and $ARGV[1] are pronounced similarly.";
    } else {
        print "$ARGV[0] and $ARGV[1] are pronounced differently.";
    }
}
```

announce first prints the Soundex codes for *perl*, *pearl*, and *wall*. The first two yield the same Soundex code: P640. *Wall* yields W400, and if you run the program with arguments *movie* and *moovy*, you'll see that they result in the same Soundex code.

```
% announce movie moovy
```

```
RESULT P640
```

```
P640
```

```
W400
```

```
movie and moovy are pronounced the same.
```

Text::Tabs

(Module written by David Muir Sharnoff)

The Text::Tabs module provides two functions: `expand()`, which converts tabs to spaces, and `unexpand()`, which converts spaces to tabs. Listing 8-35 uses these functions.

Listing 8-35 *fliptabs*: Using Text::Tabs to convert tabs to spaces and back

```
#!/usr/bin/perl
use Text::Tabs;
$hastabs = "Column \tLine \tAisle \tQueue ";
$notabs = "Series Row Ridge Succession";
print join(' ', expand($hastabs)), "\n";
print join(' ', unexpand($notabs)), "\n";
% fliptabs
Column Line Aisle Queue
Series Row Ridge Succession
```

Text::Wrap

(Module written by David Muir Sharnoff)

If you have a chunk of text that you want to format into a certain number of columns, you can use the `wrap()` and `fill()` functions in the `Text::Wrap` module. Listing 8-36 demonstrates.

Listing 8-36 *parawrap*: Using Text::Wrap to wrap paragraphs

```
#!/usr/bin/perl
use Text::Wrap
$string = "To promote the Progress of Science and useful Arts, by
securing for limited Times to Authors the exclusive Right to their
respective Writings and Discoveries.";
$Text::Wrap::columns = 50;
print wrap("> ", "> ", $string);
% parawrap
```

```
RESULT > To promote the Progress of Science and useful
> Arts, by securing for limited Times to Authors
> the exclusive Right to their respective
> Writings and Discoveries.
```

The `fill()` function behaves like `wrap()`, but allows you to wrap a series of paragraphs instead of just one.

Text::ParseWords

(Module written by Hal Pomeranz)

The `Text::ParseWords` module splits a line (or array of lines) into words in the same way that a shell would. Leading whitespace is discarded, and the words can be extracted with either of these functions:

- `shellwords(STRING)` splits *STRING* into words, ignoring any delimiters in the words.
- `quotewords(DELIM, KEEP, STRING)` breaks *STRING* up into a list of quoted words separated by *DELIM*. If *KEEP* is `TRUE`, any quoting delimiters are kept.

Listing 8-37 demonstrates these functions.

Listing 8-37 *shellify*: Using Text::ParseWords to split strings into words

```
#!/usr/bin/perl -l
use Text::ParseWords;
# There are two "words" here: hi there and "goodbye".
$string = qq!"hi\ there" `goodbye`!;
print join(---, shellwords($string));
$string = qq!"red yellow" blue!;
print join(---, quotewords(``, 0, $string)); # join red yellow
and blue
print join(---, quotewords(``, 1, $string)); # join "red yellow"
and blue
$string = qq!"alpha `omega`!; # Oops! Missing a double-quote
@words = shellwords($string); # shellwords() will complain, so
print "Reached the end!"; # this will never be reached
```

shellify inserts a triple-hyphen between the words that it extracts from the strings:

```
% shellify
```

```
RESULT hi there---"goodbye"
```

```
red yellow---blue
```

```
"red yellow"---blue
```

```
Unmatched quote at shellify line 13
```

The *STRING* can actually be an array of strings, in which case `shellwords()` and `quotewords()` combine them before the extraction.

Search

As bundled with Perl, the Search module set contains only one module at present: `Search::Dict`, which performs a binary lookup on an alphabetically ordered file. (See the Hard exercise at the end of Chapter 3.)

Search::Dict

`Search::Dict` provides the `look()` function, which searches through an alphabetically ordered file and sets the file pointer to the first line to come after a given string:

```
look(*FILE, $string, $dict, $fold);
```

If `$dict` is nonzero, dictionary ordering (using only letters, digits, and whitespace) is used for comparisons. Otherwise, all characters are used.

If `$fold` is nonzero, capitalization is ignored when making comparisons. This function is used in Listing 8-38.

Listing 8-38 *startat*: Using Search::Dict

```
#!/usr/bin/perl
use Search::Dict;
open(DICT, "/usr/dict/words") or die "Can't read /usr/dict/words:
$! ";
look(*DICT, "perl", 1, 1);
$nextline = <DICT>;
print "The next line after `perl` is ", $nextline;
```



```
% precise
```

```
RESULT +11557273494230422767386509211207482352238E-40
```

bigrat.pl

(Library file written by Mark Biggar)

bigrat.pl enables infinite-precision arithmetic on rational numbers (fractions), as shown in Listing 8-41.

Listing 8-41 *fraction*: Using *bigrat.pl*

```
#!/usr/bin/perl
require "bigrat.pl";
print rsub("1/3", "1/2");
```

One-third minus one-half is negative one-sixth:

```
% fraction
```

```
RESULT -1/6
```

Math::Complex

(Module written by David Nadler)

Chapter 7 used a class of complex numbers to demonstrate overloading the + operator. The *Math::Complex* module does quite a bit more. Listing 8-42 is a simple example that adds two complex numbers.

Listing 8-42 *imagino*: Using *Math::Complex*

```
#!/usr/bin/perl
use Math::Complex;
$x = new Math::Complex(2, 3); # 2 + 3i
$y = new Math::Complex(4, 7.5); # 4 + 7.5i
$z = $x + $y;
print $z;
% imagino
6+10.5i
```

Chapter 8, Perl 5 Interactive Course

LESSON 8

UNIT 8

DOTSH.PL, TERM, AND POSIX

This final lesson covers a couple of modules that don't fit anywhere else in this chapter (or this book). The *dotsh.pl* library file allows you to execute an external shell script, just like `do()` lets you execute external Perl programs. The Term module set includes facilities for interacting with a user's terminal--the interface between the user's screen and the operating system. Finally, the POSIX module contains a large number of functions that allow you to build POSIX-compliant Perl programs.

dotsh.pl

In UNIX, the `."` (or "source") command executes a shell script in your current environment. *dotsh.pl* lets you do the same from a Perl program. Why is this necessary? Well, backquotes and `system()` both create a new process and therefore a new environment. So if you `system()` or backquote a shell script, it'll modify the new environment (say, by adding a directory to your PATH) but not the environment of your Perl program.

dotsh.pl

(Library file written by Chuck Collins) You don't need *dotsh.pl* to modify your environment; you can do that easily enough with `%ENV`. But if you've been handed a shell script that modifies its environment and you want it to affect your program's environment (but you don't want to convert it to Perl), *dotsh.pl* is handy. The simple *sh* script *addpath.sh* adds the current directory (`.`) to the user's path and echoes a string, which is printed directly to standard output:

```
#!/bin/sh
PATH=$PATH:/tmp/books # Add /tmp/books to user's PATH
echo "I'm in a shell script!"
```

Now let's see a Perl program that "dots" *addpath.sh* and imports the PATH into the current environment (Listing 8-43).

Listing 8-43 *use_sh*: Using *dotsh.pl* to execute a shell script

```
#!/usr/bin/perl -w
require 'dotsh.pl';
print "Path is ", $ENV{PATH};
```



```
# Run addpath.sh from the shell, which imports $PATH
dotsh(`addpath.sh`);
print "Path is now ", $ENV{PATH};
```

When *addpath.sh* is run, the echo prints *I'm in a shell script!* directly to standard output. The shell script adds */tmp/books* to the PATH environment; because it shares an environment with the Perl program, the result is visible in \$ENV{PATH}:

```
% use_sh
RESULT Path is /usr/bin:/usr/local/bin:/usr/ucb
I'm in a shell script!
Path is now /usr/bin:/usr/local/bin:/usr/ucb:/tmp/books
```

Term

In UNIX parlance, your terminal is the connection between you and your computer. Your terminal configuration affects that connection: for example, how many lines you have on your screen, how newlines are sent, and the speed at which the bits flow from your keyboard to the computer.

Term::Cap

The *Term::Cap* module allows you to manipulate your termcap, which is a collection of terminal configuration options. For more information, see the termcap manual page on your UNIX system. Listing 8-44 is a sample of the terminal characteristics you can control via the *Term::Cap* module.

Listing 8-44 *termplay*: Using *Term::Cap*

```
#!/usr/bin/perl
# These three lines determine $ospeed.
# If they don't work for you, try 19200.
require `sys/ioctl.ph`;
ioctl(TTY, $TIOCGGETP, $sgtty);
($ispeed, $ospeed) = unpack(`cc`, $sgtty);
require Term::Cap;
$term = Tgetent Term::Cap { TERM => undef , OSPEED => $ospeed
};
print $term->{_cl};
print "This line is normal.\n";
print $term->{_md};
print "This line might be highlighted.\n";
print $term->{_mr};
print "This line might be reversed.\n";
```

```

print $term->{_me};
print $term->{_us};
print "This line might be underlined.\n";
print $term->{_me};
print "Your terminal is configured for ", $term->{`_co`}, "
columns";
print " and ", $term->{`_li`}, " lines.\n";

```

termplay changes the user's terminal just by printing. For instance, **print \$term-> {_md}** switches the terminal into highlighting mode, and **print \$term->{_me}** switches the terminal back to normal. Your programs can use this to highlight or underline text on certain systems.

```
% termplay
```

```
RESULT This line is normal.
```

```
This line might be highlighted.
```

```
This line might be underlined.
```

```
Your terminal is configured for 80 columns and 24 lines.
```

Term::Complete

(Module written by Wayne Thompson) The Term::Complete module allows someone using your program to hit [TAB] to complete a word or [CTRL]-[D] to see all the options. Listing 8-45 uses this module.

Listing 8-45 *guess*: Using Term::Complete

```

#!/usr/bin/perl -w
use Term::Complete;
@list = qw( abrogate continue );
$input = Complete(`Choice? `, @list);

```

The Complete() function prompts the user for a Choice?, but permits the user to use [TAB] to complete any abbreviation that matches a word in @list:

```
% guess
```

```
RESULT Choice? a[TAB]brogate
```

and allows the user to see the contents of @list with [CTRL]-[D]: % guess

```
RESULT Choice? [CTRL]-[D]
```

```
abrogate
continue
```

POSIX

POSIX is an international standard for UNIXish operating systems, used to enhance portability among applications. Toward that goal, it specifies how a lot of operating system functions (e.g., `mkdir()` and `sleep()`) should be called: the number, order, and type of their arguments. If an operating system aspires to POSIX compliance, it has to implement those functions in exactly the manner specified by the POSIX standard. That's awfully constraining for Perl, which has functions that support different numbers and types of arguments. But if you convert a POSIX-compliant application from C to Perl, it will expect system functions to behave in a POSIX-compliant manner. That's when the POSIX module comes in handy. POSIX also implements some useful functions that aren't built into Perl, such as `ceil()`, `floor()`, and `tan()`.

POSIX

POSIX cries out for autoloading. It contains over 100 functions (many of which simply call their Perl counterparts and juggle the arguments a little); any given Perl program is likely to use just a small subset of those. Thanks to autoloading, you pay only for the functions you need. Listing 8-46 is a short example that loads in the POSIX definition of `sleep()`. POSIX's `sleep()` requires an argument. Perl's `sleep()` doesn't, so an error is raised.

Listing 8-46 *posixnap*: Using POSIX

```
#!/usr/bin/perl -w
use POSIX `sleep`;
sleep;
% posixnap
```

RESULT Usage: POSIX::`sleep(seconds)` at `./posixnap` line 5

You can always access Perl's built-in functions through the CORE package. If you want to use both POSIX `sleep()` and Perl's `sleep()`, you can import the POSIX definition as above and then refer to Perl's built-in `sleep()` as `CORE::sleep()`. Perl doesn't have `ceil()` or `floor()` functions. POSIX does, as shown in Listing 8-47.

Listing 8-47 *rounder*: Using POSIX::`ceil()` and POSIX::`floor()`

```
#!/usr/bin/perl -wl
use POSIX qw( ceil floor );
print "ceil (99.7) is ", ceil 99.7;
print "floor(99.7) is ", floor 99.7;
print "int (99.7) is ", int 99.7;
print "ceil (-99.7) is ", ceil -99.7;
print "floor(-99.7) is ", floor -99.7;
print "int (-99.7) is ", int -99.7;
```

Above zero, `int()` behaves like `floor()`. Below zero, `int()` behaves like `ceil()`:

```
% rounder
```

RESULT ceil (99.7) is 100

floor(99.7) is 99

int (99.7) is 99

ceil (-99.7) is -99

floor(-99.7) is -100

int (-99.7) is -99

Review

Kevin: We've skimmed through a lot of modules; I'm not going to remember how half of them work. And future versions of Perl are going to come with even more modules, right?

Jamie Lee: Right, and there are hundreds of modules floating around on the Net, many of them concentrated in the CPAN. The important thing is knowing which modules exist, not knowing how they work, because they usually come in their own *pod* documentation.

Kevin: The file utilities seem especially useful. I'm always forgetting where I put my files, so I coded up a one-liner that searches through my home directory for files containing a certain string.

Jamie Lee: Neat! I'd like something similar for my e-mail. By the way, I just realized that the ID on my driver's license uses the Soundex code for my last name, C632!

Chapter 9, Perl 5 Interactive Course



PUSHING THE ENVELOPE: PROCESS MANAGEMENT

Programs are static. They're just documents on disks. But when you run a program, it becomes a dynamic entity, a *process* that grows, shrinks, and inevitably dies. It might receive signals from users or the operating system itself. It might even talk to other processes via *interprocess communication*.

This chapter is all about processes: how you create them, how you can control them, and how processes running on the same computer can talk to one another. When processes running on different computers talk to one another, that's *networking*, the subject of Chapter 13.

In this chapter, you'll learn about some functions that are relatively close to the operating system. High-level functions like `grep()` and `study()` manipulate Perl data structures; they have little to do with the operating system itself. Other functions, like `print()` and `open()`, are a bit closer to the operating system, but Perl still insulates you somewhat from the internal structure. Some of the functions in this chapter are at an even lower level, at the junction between your programs and the UNIX kernel. Virtually all of this chapter is UNIX-specific; if you're running Perl on a computer that doesn't have UNIX-style processes, you can stop after Lesson 1.

These low-level functions generally have very simple behavior. The difficult part is knowing when and how to use them, so pay close attention to the examples!

This chapter begins with a Perl function, `eval()`, that has nothing to do with processes. But `eval()` is similar to processes in spirit; it lets your program assemble and execute Perl statements on the fly.

eval()

`eval()` lets you evaluate a self-contained string or block of Perl code, so that your program can assemble its own statements, expressions, blocks, subroutines, and even formats. When you call `eval()`, you're saying, "Here's some Perl code. Pretend it's a little self-contained program and execute it."

The `eval()` Function

`eval(EXPRESSION)` evaluates the Perl code in *EXPRESSION*.

If *EXPRESSION* is omitted, `$_` is used.

```
eval { $var = 7 };
```

evaluates the block `{ $var = 7 }`, which sets `$var` to 7. The argument to `eval()` can be a full-fledged block with a return value:

```
$a = eval { $var = 7; return $var * $var }
```

sets `$a` to 49. The expression `$var = 7` could have been built from scratch into a string rather than a block:

```
$string = '$' . 'var' . '=' . '7';
```

`$string` is now `'$var=7'`.

```
eval $string;
```

sets `$var` to 7.

"But wait!" you say. "If you already know you want `$var` to be 7, why not just do it? Why bother with the `eval()`?" Well, you might not always know beforehand what code you want to execute. Listing 9-1 shows an example, a tiny Perl interpreter that lets you execute Perl statements simply by typing them.

Listing 9-1 *pesh*: Using `eval()` to emulate a shell

```
#!/usr/bin/perl
do {
$return = eval;
print "\n";
print $return;
print $@;
print "\n\nperl> ";
} while (<>);
```

What's going on here? It's not as bad as it looks. Every line typed by the user is assigned to `$_` (by the `while (<>)`). Then `eval()` evaluates `$_`, and the result is placed in `$return`. `$return` is printed and then `$@` is printed (don't worry about that yet), followed by the `perl>` prompt.

```
% pesh
perl> sqrt 2
1.4142135623731 # $return is 1.414.
```

```
perl> print (integral)Square the Circle(integral)
Square the Circle # this is just a side effect of print
1 # this is the actual return value
perl> ;ls -l; # or dir on MS-DOS machines

-rwxr-xr-x 1 perluser 100 113 May 15 02:05 pesh
perl> open (FILE, "/usr/dict/words");
1 # open() returns 1 on success
perl> seek(FILE, 10000, 0);
1 # seek() returns 1 also
perl> <FILE>; $a = <FILE>; # evaluating two statements at a
time
arcsine
perl> 6 / 0 # creating a run-time error...
Illegal division by zero at (eval 3) line 3, <> line 4.
perl> time # ...but we're still alive
906816065
perl> <CTRL-D> # leave pesh
%
```

Wait a minute...normally, dividing by zero makes your program immediately exit()! How come *pesh* didn't halt? Because each `eval()` is its own little program and that program died, not your shell. That's the other advantage of `eval()`: trapping fatal errors.

Trapping Fatal Errors

You can use `eval()` to trap otherwise fatal errors in a section of code because a fatal error merely kills the `eval()` and not your program. The error string is placed into the `$@` special variable.

The `$@` Special Variable

`$@` (or `$EVAL_ERROR`) contains the error message from the last `eval()`.

If the last `eval()` caused no errors, `$@` will be empty (but not undefined!).

Listing 9-2 shows a program that divides the first command-line argument by the second. But instead of checking the second divisor to make sure it's not 0, the *safediv* program blithely divides the first argument by the second inside an `eval()`.

Listing 9-2 *safediv*: Using `$@` to trap fatal errors

```
#!/usr/bin/perl -l
eval { $result = $ARGV[0] / $ARGV[1] };
warn $@ if $@;
print "Answer: $result";
```

When you run *safediv* with a nonzero denominator, you get the first argument divided by the second:


```
% safediv 7 2
```

```
RESULT Answer: 3.5
```

But when you run *safediv* with the second argument 0, `eval()` traps the run-time error and places the error message in `$@`. The program continues. (You can tell because *Answer:* is still printed.)

```
% safediv 7 0
```

```
RESULT Answer:
```

```
Illegal division by zero at safediv line 3.
```

Assembling Functions

You can assemble and evaluate your own subroutines, too. The *twinkle* program shown in Listing 9-3 assembles a subroutine called `allstar()`.

Listing 9-3 *twinkle*: Using `eval()` to assemble functions

```
#!/usr/bin/perl -wl
$num_stars = 22;
$func = `sub allstar { print "*" x $num_stars }`;
eval $func;
allstar();
$num_stars *= 2;
allstar();
```

`allstar()` prints *\$num_stars* stars. It's a normal subroutine, so when *\$num_stars* doubles, it notices:

```
% twinkle
```

```
RESULT *****
```

```
*****
```

twinkle could also have been written without `eval()`, using anonymous subroutines, as shown in Listing 9-4.

Listing 9-4 *twinkle2*: Using anonymous subroutines

```
#!/usr/bin/perl -wl
$num_stars = 22;
$func = sub { print "*" x $num_stars };
&$func();
$num_stars *= 2;
&$func();
```

You can even `eval()` formats, say, by reading a format from a file into a string. Or you could assemble a format at runtime, modifying elements as you see fit. Both define the format just as if there were a format declaration in your program. To Perl, there's no difference.

Many compiled languages (including C) have no equivalent of `eval()`. That's one of the reasons that a

Perl-to-C translator is hard to write: If your program contains an `eval()`, the corresponding C program will need to include the entire Perl interpreter. If you can see why this computes the factorial of `$number`, you'll know that you know that you know `eval()`.

```
$number = 5;
$factor = 1;
$string = '!$number ? $factor : ($factor = eval { $number-- *
eval $string })';
print eval $string;
```

Chapter 9, Perl 5 Interactive Course

LESSON 2

UNIT 2

SIGNALS

Signals are (more or less) software interrupts. When you [CTRL]-[C] a process or when you use the UNIX *kill* command, you're sending a particular signal to that process (INT and KILL, respectively). Operating systems can send signals, too; if you've ever seen a program complain about a bus error, that was the operating system sending the bus signal to the program. This lesson is about how you can trap those signals so you can define what they do and how you can generate signals so you can send them to other programs. A full list of all the signals is in Appendix C; a partial list is shown below in Table 9-1. Most UNIX machines use roughly the same names for signals (and numbers--every signal has an associated number). But there are a few differences between systems. You can find out which signals you have and what their numbers are with the program shown in Listing 9-5.

Table 9-1 Some signals that might be sent to your program

Signal	Description
ALRM	An alarm timed out.
BUS	Bus error (hardware fault).
CHLD	A child process has stopped.
CONT	Continue the stopped process.
INT	Interrupt [CTRL]-[C].
KILL	Terminate the process.
PIPE	Process tried writing to a pipe with no readers.
TSTP	Stop the process [CTRL]-[Z]; can be continued later.
WINCH	The terminal window size has changed.

Listing 9-5 whatsigs: Checking signals

```
#!/usr/bin/perl -wl
use Config;
@signals = split(``, $Config{sig_name} );
foreach (0..$#signals) { print "$_: $signals[$_]" }
```

(POSIX has its own ideas about what constitutes proper signals. If you're writing POSIX-compliant applications, use the signal handling provided by the POSIX module.) Some of the signals shown in

Table 9-1 cause the running program to terminate (e.g., INT, KILL); others don't (e.g., CHLD, WINCH). But all signals (except for KILL and STOP) can be trapped, meaning that your program can countermand whatever action would normally be taken.

Trapping Signals

Whenever a signal is sent to your program, UNIX takes a particular action that depends on the signal. For some signals (like INT), that action is to terminate your program. If you want to intercept that signal so that you decide what happens, you need to trap that signal with a *signal handler*. [CTRL]-[C] can be trapped by a signal handler that traps the INT signal. eval() is handy because it lets you trap fatal errors. But it has a drawback: You have to know beforehand where the errors might occur. Signals can arrive at any time, not just when a particular operation is executed. Therefore, any mechanism that can catch signals must be "awake" during your entire program. This mechanism is made available through the %SIG hash, one of the few truly global variables in Perl.

The %SIG Hash

%SIG maps signals to subroutines; when the signal occurs, the subroutine is executed.

The keys in %SIG are the names of signals: INT, TSTP, and so on. The values can be names of subroutines, or an anonymous subroutine, or either of two special strings: 'IGNORE' (which throws the signal away) or 'DEFAULT' (which lets the operating system decide what to do). Two signals can never be caught: KILL and STOP. For instance, to trap INT signals, you could do any of the following:

```
$SIG{INT} = sub { print "<CTRL-C> has been disabled\n"; }
```

or

```
$SIG{INT} = 'IGNORE';
```

or

```
$SIG{INT} = \&morbid
```

which executes the morbid() subroutine instead of terminating your program, as illustrated in Listing 9-6. (You could also have said \$SIG{INT} = 'morbid', but it's always better to use references.)

Listing 9-6 *nobreak*: Using %SIG to trap signals

```
#!/usr/bin/perl -w
sub handler {
    print "\nPlease. I'm not done yet.\n";
}
$SIG{INT} = \&handler;
$| = 1;
foreach (1..50) {
    print ".";
    sleep 1;
}
```

```
% nobreak
```

```
RESULT [CTRL]-[C]
```

```
Please. I´m not done yet.
```

```
[CTRL]-[C]
```

```
Please. I´m not done yet.
```

```
[CTRL]-[Z]
```

```
Stopped
```

[CTRL]-[C] didn't work, so the user sent a different signal to the process:[CTRL]-[Z], which (on UNIX systems) suspends the current process. The name of that signal is TSTP; with a small change, you can modify your program so it's trapped by the handler() as well. You'll also want the handler() to identify which signal triggered it. Luckily, signal handlers are provided with a single argument: the name of the signal, as shown in Listing 9-7.

Listing 9-7 *nobreak2*: Trapping two signals to the same subroutine

```
#!/usr/bin/perl -w
sub handler {
my $sig = shift;
if ($sig eq `INT`) {
print "\n ^C won´t work! \n";
} elsif ($sig eq `TSTP`) {
print "\n ^Z won´t work either! Ha ha ha! \n";
}
}
$SIG{INT} = \&handler;
$SIG{TSTP} = \&handler;
$| = 1;
foreach (1..50) {
print ".";
sleep 1;
}
% nobreak2
```

```
RESULT [CTRL]-[C]
```

```
^C won´t work!
```

```
[CTRL]-[Z]
```

`^Z won't work either! Ha ha ha!`

`.....`

To be nice, you could allow the user to terminate the program if he or she presses [CTRL]-[C] twice. To do that, you could restore the original meaning of `$SIG{INT}`, which you can do by setting its value to `'DEFAULT'`, as shown in Listing 9-8.

Listing 9-8 *nobreak3*: Restoring the original behavior of a signal

```
#!/usr/bin/perl -w
sub handler {
    print "\nIf you really meant to do that, press ^C again.\n";
    $SIG{INT} = 'DEFAULT';
}
$SIG{INT} = 'handler';
$| = 1;
foreach (1..50) {
    print ".";
    sleep 1;
}
% nobreak3
```

```
RESULT .....[CTRL]-[C]
If you really meant to do that, press ^C again.
.....[CTRL]-[C]
```

If you wish simply to ignore a signal, you can do so by setting its `%SIG` entry to `'IGNORE'`, as shown in Listing 9-9.

Listing 9-9 *nobreak4*: Ignoring signals

```
#!/usr/bin/perl -w
$SIG{INT} = 'IGNORE';
$SIG{TSTP} = 'IGNORE';
$| = 1;
foreach (1..20) {
    print ".";
    sleep 1;
}
% nobreak4
```

```
RESULT ..[CTRL]-[C]...[CTRL]-[Z].....
```

\$SIG{__WARN__} and \$SIG{__DIE__}

You can also use %SIG to trap two Perl events: warn()s and die()s. Because those aren't actual signals, they're encased in double underscores to distinguish them from conventional UNIX signals. But you can use them the same way:

```
$SIG{__WARN__} = sub { print "You have have been warned." }  
  
warn "Error! Bad input!";
```

prints *You have been warned.* instead of *Error! Bad input!*.

The *sigtrap* Pragma

The *sigtrap* pragma traps 11 signals (QUIT, ILL, TRAP, ABRT, EMT, FPE, BUS, SEGV, SYS, PIPE, and TERM) and invokes its own handler (sigtrap::trap()). *trapper*, shown in Listing 9-10, uses *sigtrap* to catch all 11 signals and notify the user. That's helpful, because UNIX often doesn't provide the courtesy of telling you when a signal has been sent or what statement was being executed when the signal arrived.

Listing 9-10 *trapper*: Using the sigtrap program to trap unexpected signals

```
#!/usr/bin/perl  
use sigtrap; # Note lowercase, because it's a pragma.  
sleep;
```

trapper immediately falls asleep. On UNIX systems, you can make *trapper* run in the background by appending an &:

```
% trapper &
```

```
RESULT [1] 24518
```

This tells you that your trapper process has process ID (PID) 24518, which you can use as input to the UNIX *kill* command. (You could also have found out the PID with the *ps* command.) To send the QUIT signal to PID 24518, do the following:

```
% kill -QUIT 24518
```

```
RESULT Caught a sigquit at ./trapper line 3
```

The UNIX *kill* command doesn't print anything; that *Caught a SIGQUIT...* message came from sigtrap::trap(), which subsequently died. \$SIG{__DIE__} could then have been used to prevent the program from exiting.

Sending Signals

Sometimes programs run amok and need to be put out of their misery. Or maybe you're feeling malevolent and you want to kill someone else's process. Either way, you'll want to kill().

The kill() Function

kill(SIGNAL, ARRAY) sends the *SIGNAL* to the *ARRAY* of processes, returning the number of processes successfully signaled.

The *SIGNAL* can either be the signal name ('KILL', 'INT', 'TSTP', etc.) or number (from Table 9-1 and Appendix C). *ARRAY* is a list of process IDs, described below. Negative signals kill() process groups instead of processes. (Process groups are described in Lesson 4.) Listing 9-11 is a program that uses open()'s "command|" form combined with kill() to kill all programs with a specified name.

Listing 9-11 *killname*: Using kill() to send a signal to a process

```
#!/usr/bin/perl -w
$martyr = shift;
# The UNIX ps program lists process names and IDs.
open(PS, "ps |");
# For each line of ps output, try to extract
# the process ID and process name.
# This won't work on all computers.
while (<PS>) {
    ($name) = /[A-Za-z]\w{2,}/;
    ($pid) = /(\d{3,5})/;
    push(@deathrow, $pid) if ($name eq $martyr);
}
kill 9, @deathrow;
```

killname assumes that the ps command returns two columns: the name of the process followed by a number--its PID. Those numbers are placed in @deathrow, which becomes the *ARRAY* provided to kill(). Every UNIX process has a PID, no matter who (or what) creates it. Inside a process, you can access the PID with the \$\$ special variable.

The \$\$ Special Variable

\$\$ (or \$PID, or \$PROCESS_ID) contains the PID (process ID).

The operating system assigns a unique PID to each process. This means that \$\$ will usually contain a different number each time your process is run (Listing 9-12). Chapter 5, Lesson 3 uses \$\$ as an input to srand() because that's the one variable that is most likely to change if you run your program twice in succession.

Listing 9-12 *soundoff*: Accessing the PID with \$\$

```
#!/usr/bin/perl -w
print "I'm process number $$.\n";
% soundoff
```

```
I'm process number 18755.
% soundoff
I'm process number 18756.
.
```

Listing 9-13 shows how a process can kill itself.

Listing 9-13 *suicide*: Using kill to send a signal to a process

```
#!/usr/bin/perl -w
print "Goodbye...\n";
kill 'INT' $$;
print "I'll never live to see this line."
```

suicide doesn't print anything after *Goodbye...* because the INT signal terminates the process before the print() is reached:

```
% suicide
```

```
RESULT Goodbye...
```

Alarm

The ALRM signal occurs when a user-set timer expires. You can set that timer with the alarm() function.

The alarm() Function

alarm(NUMBER) generates an ALRM signal after *NUMBER* seconds.

If *NUMBER* is omitted, the previous alarm() is canceled. Be careful when mixing alarm()s and sleep()s. They won't do what you expect on some systems. You can trap the ALRM signal if you wish; if you don't, your process will terminate. *robocop*, shown in Listing 9-14, uses alarm() to force the user to enter the password (well, any password) within 5 seconds.

Listing 9-14 *robocop*: Using alarm() to terminate the process after time expires

```
#!/usr/bin/perl
print "Enter the password. You have five seconds to comply.
";
alarm(5); # send an ALRM signal in five seconds
chomp($password = <>);
print "Very good.\n";
alarm(); # cancel the previous alarm()
% robocop
```

```
RESULT Enter the password. You have five seconds to comply.
```


<wait 5 seconds>

Alarm clock

Chapter 9, Perl 5 Interactive Course

LESSON 3

UNIT **exec() AND fork()**

You create a process whenever you run a program. But your programs can create processes of their own, too: Backquotes and `system()` are examples. Each creates a process, waits for it to finish, and then returns control to your program. In contrast, `fork()` creates a new process but doesn't wait around for it to finish. `exec()` replaces your program with another one without launching a new process at all.

exec()

Sometimes you'll want to run another program instead of the current one. For that, you can use `exec()`, which is just like `system()` except that it terminates the current program and passes control to another.

The exec() Function

`exec(PROGRAM, ARGS)` terminates the Perl program and runs ***PROGRAM*** in its place.

So

```
system("ls");  
  
print "There ya go."
```

executes an `ls` command and then prints *There ya go.*, whereas

```
exec("ls");  
  
print "There ya go."
```

terminates the Perl program and runs `ls` instead. The `print()` statement is never reached.

Although both `system()` and `exec()` execute a command in the user's shell, `system()` returns after the command completes, whereas `exec()` terminates the current program as it begins the command.

Suppose you are designing a login program that gives users a choice of programs to execute: a mail program, a spreadsheet, and so on. When a user makes a selection, the program should terminate, launching the chosen program. That's a perfect use for `exec()`, as illustrated in Listing 9-15.

Listing 9-15 login: Using exec() to replace one program with another

```
#!/usr/bin/perl -w  
  
@programs = ("mail", "text editor", "calculator");  
  
@commands = ("/usr/bin/mail", "/usr/local/bin/emacs",
```

```

"/usr/bin/X11/xcalc");

foreach $i (0..$#programs) { print "$i. $programs[$i]\n" }

print "Choose a program: ";

chomp($choice = <>);

exec $commands[$choice];

print "This line will never be executed.!!";

```

login lets users choose a program; that program is then `exec()`uted, replacing *login*.

```
% login
```

RESULT 0. MAIL

```
1. text editor
```

```
2. calculator
```

```
Choose a program: 0
```

```
Mail version 5.3 2/18/88. Type ? for help.
```

```
"/usr/spool/mail/perluser": 12 messages 10 new 10 unread
```

```
U 7 earhart_ame@afrotc.mil Fri Oct 17 18:26 65/2309 "The
Bermuda Trian"
```

```
U 8 james_hoffa@stadium.underg Fri Oct 17 18:13 60/2547
"Would anyone mind"
```

```
U 9 vanilla_ice@rhapsody.org Fri Oct 17 18:47 77/3715 "Got a
great bassl"
```

and so on.

Special Characters in Perl Versus Special Characters in UNIX Shells

When you use `system()` and `exec()` and backquotes, remember that Perl sees your command before the shell does. So if you're using double quotes or backquotes, variable interpolation will occur before the shell sees the command. Backslash when appropriate:

```
$shell_name = `echo \$$SHELL`;
```

properly sets `$shell_name` to `/bin/csh` (or whatever your shell happens to be), but

```
$shell_name = `echo $$SHELL`;
```

won't work: `$shell_name` will be set to a newline, because the backquotes interpolate `$$SHELL`, yielding an empty string.

Forking Processes

`system()` starts another process and waits for it to terminate; `exec()` starts another program and terminates itself. `fork()` starts another process and continues to execute. `fork()` is tricky. It launches a new process identical (well, nearly identical) to the old process.

The `fork()` Function

`fork()` splits a process in two, returning the PID of the child process.

`fork()` creates a new process identical to the old process in all respects but one: The PID returned by `fork()` is visible only to the old process. Listing 9-16 shows how `fork()` is used. Stare at *tryfork* for a while.

Listing 9-16 *tryfork*: Using `fork()` to split a process in two

```
#!/usr/bin/perl -wl

$pid = fork;

if ($pid) { print "I'm the parent." }

else { print "I'm the child. " }
```

The `$pid = fork` statement creates a new process, called the child. (The original process is called the parent.) In the parent process and ONLY the parent process, `$pid` contains the PID of the child. That's how the child knows it's a child: `$pid` is false.

```
% tryfork
```

```
RESULT I'm the child.
```

```
I'm the parent.
```

tryfork might print the lines in the opposite order, because once the `fork()` occurs, there are really two copies of *tryfork*, both of which try to print at the same time. *tryfork* can be improved in a couple of ways. First, the `fork()` might fail, in which case `$pid` will be undefined. It's also good practice to have the child `exit()` so that it doesn't execute code later in the program that's meant only for the parent. *tryfork2* (Listing 9-17) remedies both problems.

Listing 9-17 *tryfork2*: Improving *tryfork*

```
#!/usr/bin/perl -wl

$pid = fork;

if ($pid) { print "I'm the parent." }

elsif (defined $pid) { print "I'm the child."; exit; }

else { die "Couldn't fork: $! " }

print "The child has now exited, so this will be seen by the
```

```
parent only.";
```

Why (defined \$pid) rather than (\$pid == 0)? Because undefined values are still == 0.

```
% tryfork2
```

```
RESULT I´m the parent.
```

```
I´m the child.
```

The child has now exited, so this will be seen by the parent only.

Use fork() whenever your program needs to do two things at once. It'll help you write programs that serve multiple users, avoid pauses in interactive programs, and provide uninterrupted network service, as you'll see in Chapter 13.

If both parent and child are printing to STDOUT, it's good practice to unbuffer STDOUT before the fork(), as *tryfork3* (Listing 9-18) does.

Listing 9-18 *tryfork3*: Using fork() to create a new process

```
#!/usr/bin/perl -wl

$| = 1; # necessary because both child and parent compete for
STDOUT

$pid = fork;

if ($pid) {

    print "I´m the parent.";

    print "While you were reading this sentence, my child made a
    directory.";

} elsif (defined $pid) {

    print "\t\t I´m a child.";

    mkdir ("/tmp/newdir", 0777);

    print "\t\t Bye bye!";

    exit;

} else {

    die "Couldn´t fork: $! ";

}

print "All done.";
```

In *tryfork3*, the child creates a directory while the parent prints some lines for the user to read. When the child is done, it exit()s, to avoid executing code meant only for the parent.

```
% tryfork3
```

```
RESULT I'm a child.
```

```
While you were reading this sentence, my child made a  
directory.
```

```
I'm the parent.
```

```
All done.
```

```
%
```

```
RESULT Bye Bye!
```

The child's output is indented. Notice that when this program ran, the parent finished before the child--that's why the shell prompt (%) appears before the child is done printing.

Chapter 9, Perl 5 Interactive Course

LESSON 4

OBJECTIVE

PROCESS ATTRIBUTES

Now that you've learned how to create processes, it's time to learn how to modify them. This lesson covers five functions, all pertaining to process attributes: process names, process priorities, and process groups.

Changing the Process Name

Suppose you want your program to call itself with different arguments. You can use `exec()`, but `exec()` needs to know which program to execute. It could use either the `__FILE__` literal or the `$0` special variable.

The \$0 Special Variable

`$0` (or `$PROGRAM_NAME`) contains the name of the running program.

Listing 9-19 is an example of when you might want your program to call itself:

Listing 9-19 *obsolete*: Using `system()` to make a program call itself

```
#!/usr/bin/perl -wl

if ($ARGV[0] eq "-y") {
    print "The -y switch is obsolete.";
    print "Use -x instead.";
    print "Here---I'll use it for you.";
    print "Next time, use -x! \n";
    exec $0, '-x'; # call myself with -x
}

print "Thank you for using the -x option.";
```

When *obsolete* is called with `-x`, only the last `print()` triggers:

```
% obsolete -x
```

RESULT Thank you for using the `-x` option.

But when the `-y` option is used, the *if* statement triggers and the program executes itself with the correct option:

```
% obsolete -y
```

RESULT The `-y` switch is obsolete.

```
Use -x instead.
```

```
Here---I'll use it for you.
```

```
Next time, use -x!
```

```
Thank you for using the -x option.
```

For added fun, you can set `$0` to change the name of your process. When you use the UNIX *ps* command to look for your process, you'll see the value of `$0`, as demonstrated by *oldname*, shown in Listing 9-20.

Listing 9-20 *oldname*: Using `$0` to change the process name

```
#!/usr/bin/perl
$0 = "Hello!";
sleep;
% oldname &
% ps
PID TT STAT TIME COMMAND
10523 p2 SW 0:00 csh
10529 p2 T 0:06 emacs
10567 p2 SW 0:00 Hello!
```

Not all computers let you change process names, however.

Getting the PID of the Parent Process

It's easy for a parent to know the PID of its child, because that's what `fork()` returns. A child can identify its parent with the `getppid()` function.

The `getppid()` Function

`getppid()` returns the parent PID.

If the parent of a child dies, the orphaned process becomes a ward of the state and is adopted by *init*, the generic foster parent. *init* has a PID of 1. `getppid()` is demonstrated in Listing 9-21.

Listing 9-21 *stork*: Using `getppid()` to determine the parent's PID

```
#!/usr/bin/perl -wl

print "Parent: I´m $$";

$pid = fork;

if ($pid) {
    print "My child is ", $pid;
} elsif (defined $pid) {
    print "\t Child: I´m $$";
    print "\t My parent is ", getppid;
    exit;
} else {
    die "Couldn´t fork: $! ";
}
```

Now each process knows both PIDs. The child's `print()`s are indented:

```
% stork
```

```
RESULT Parent: I´m 1051
```

```
My child is 1052
```

```
Child: I´m 1052
```

```
My parent is 1051
```

Every user process has a parent, even if it wasn't `fork()`ed. Any program you run directly from your UNIX shell will have that shell as its parent.

```
% echo $$
```

```
RESULT 29284
```

prints the PID of your shell.

Typing the following command launches a new process (`perl`) that prints the PID of its parent. Because the UNIX shell is the parent, the same PID is printed: 29284.

```
% perl -e 'print getppid'
```

```
RESULT 29284
```

Process Priorities

The *priority* of a process is a number that designates how important that process is. In the socially questionable world of UNIX, that's the proportion of system resources grabbable by the process (at the expense of other processes).

On most UNIX computers, processes have a priority of 0 by default. The priority numbers are like golf scores: The lower the priority (ranging down to -20 on many computers), the more important the process and the more attention devoted to it by the CPU. A process with a priority of -10 runs faster than a process with a priority of 10.

You can get and set the priority of a process with `getpriority()` and `setpriority()`.

The `getpriority()` and `setpriority()` Functions

`getpriority(WHAT, WHO)` returns the *PRIORITY* of *WHO*.

`setpriority(WHAT, WHO, PRIORITY)` sets the current *PRIORITY* of *WHO*.

WHAT can be a process (0), a process group (1, you'll learn about those later), or a user (2). *WHO* can be a PID or 0 to denote the current process. You can find the priority of the current process as follows:

```
print "I am running at priority ", getpriority(0, 0);
```

celerate, shown in Listing 9-22, demonstrates demoting a process by increasing its priority to 10 and then promoting it by decreasing its priority to -10. (UNIX doesn't usually let normal users speed up their own programs; you'll probably need to run this program as the superuser to see any difference.)

Listing 9-22 *celerate*: Using `getpriority()` and `setpriority()` to get and set process priorities

```
#!/usr/bin/perl -wl

print "Running at priority ", getpriority(0, 0); # probably 0
for ($i = 0; $i < 100000; $i++) { $x = rand; }
setpriority(0, 0, 10);

print "Now I'm running slower, at priority ", getpriority(0, 0);

for ($i = 0; $i < 100000; $i++) { $x = rand; }
setpriority(0, 0, -10);

print "Now I'm running faster, at priority ", getpriority(0, 0);

for ($i = 0; $i < 100000; $i++) { $x = rand; }

print "All done.";
```

The difference in loop execution speeds might be noticeable.

```
% celerate
```

```
RESULT I am running at priority 0
```

```
Now I'm running slower, at priority 10
```

```
Now I'm running faster, at priority -10
```

All done.

A child inherits the priority of its parent, as demonstrated by the *priority* program, shown in Listing 9-23.

Listing 9-23 *priority*: Children inherit the priorities of their parents

```
#!/usr/bin/perl -wl

setpriority(0, 0, 10); # Slow down the parent to 10

$pid = fork; # Create a child

if (defined $pid && !$pid) {

    print "\t\t I´m the child."; # inherits the priority

    print "\t\t My priority is ", getpriority(0, 0);

    sleep;

}

print "Child priority is ", getpriority(0, $pid); # (the
parent can tell)

% priority

I´m the child.

My priority is 10
Child priority is 10
```

Process Groups

A *process group*, or PGRP, is a number designating a collection of processes. Every process belongs to a process group and every process group has a group leader, which has the distinction of having its PID equal to the process group ID. You can get and set the process group with `getpgrp()` and `setpgrp()`.

The `getpgrp()` and `setpgrp()` Functions

`getpgrp(PID)` returns the current process group for the *PID*.

`setpgrp(PID, PGRP)` sets the current process group of *PID* to *PGRP*.

As with `getpriority()` and `setpriority()`, you can use 0 for the current process if you're too lazy to type `$$`. What can you do with process groups? Well, you can send a signal to all processes in the same group with `kill()` by prefacing the signal with a minus sign. An instance of when you might want to do this: the *chatter* program in Chapter 13, Lesson 5, a network service that spawns a process for every user connected to it. If you want to stop *chatter*, you'd have to either identify and kill every individual process or kill the process group, as shown in Listing 9-24. (Or you could reboot your computer.)

Listing 9-24 *procgrp*: Assigning process groups with `getpgrp()` and `setpgrp()`

```
#!/usr/bin/perl -wl

print "Parent is $$";

$pid1 = fork;

if (defined $pid1 && !$pid1) { # is $pid1 0 but defined?
    print "\t Child: my pgrp is ", getpgrp(0); # inherits
    from parent

    $pid2 = fork;

    if (defined $pid2 && $pid2) {
        print "\t\t Grandchild: I´m ", getpgrp(0); #
        inherits from grandparent

        sleep 4;

        setpgrp(0, getppid); # changes to child

        print "\t\t Grandchild: now I´m ",
        getpgrp(0);

        sleep;
    }

    sleep 2;

    print "\t Child: setting group to $$";

    setpgrp(0, $$); # Make group leader

    print "\t Child: now my group is $$";

    sleep;
}

print "Parent: my process group is ", getpgrp(0);

sleep 10;

print "Parent: my process group is still ", getpgrp(0);

kill -9, $pid1; # kill process group
```

procgrp choreographs the following sequence of actions (to keep things simple, *procgrp* doesn't attempt to catch `fork()` errors):

1. The parent is created (when the program is launched); it's the process leader of its one-member process group.
2. A child is created, inheriting the PGRP of its parent.
3. A grandchild is created, inheriting the PGRP of its parent, which is also the PGRP of its grandparent.
4. The child's PGRP is changed to its own PID, making it a process leader.
5. The grandchild's PGRP is changed to its parent, making it a member of the child's process.
6. A kill signal is sent to the child. That kills the grandchild, too, because it's a member of the same PGRP.

Here's *procgrp* in action:

```
% procgrp
```

```
RESULT Parent is 21297
```

```
Parent: my process group is 21297
```

```
Child: my pgrp is 21297
```

```
Grandchild: I'm 21297
```

```
Child: setting group to 21298
```

```
Child: now my process group is 21298
```

```
Grandchild: now my process group is 21298
```

```
Parent: my process group is still 21297
```

Chapter 9, Perl 5 Interactive Course

LESSON 5

UNIT 1

wait(), pipe(), Open2, AND Open3

Lessons 5, 6, and 8 of this chapter focus on the relationship between processes on your computer. This lesson focuses on how parent processes can wait for children to finish and how they can communicate with other processes through a pipe.

Waiting for Another Process to Finish

You've learned that the `system()` function launches another process and then returns control to the calling program. Here's how it works: It `fork()`s itself and then waits for the child process to finish. Imagine you are implementing your own version of `system()`. How would you know when the process has completed? You'd use the `wait()` function.

The `wait()` and `waitpid()` Functions

`wait()` waits for any child process to terminate, returning the process **PID**.

`waitpid(PID, FLAGS)` waits for the child process with **PID** to terminate.

Upon success, `wait()` returns the PID of the child process that just finished and `waitpid()` returns true. `wait()` and `waitpid()` both place the return value of the child process in `$?`, and both return -1 if there aren't any child processes.

```
wait;
```

is used when there's only one process to wait for or when any process will do.

```
waitpid(25768, 0)
```

is used to wait for a specific process: the one with PID 25768. The **FLAGS** vary from system to system; check `/usr/include/sys/wait.h` (or its equivalent on your computer) or the man page for `waitpid()`. The most important **FLAG** is **WNOHANG**, shown later. Listing 9-25 demonstrates `wait()`, which halts the current process until another process finishes.

Listing 9-25 *waiter*: Using `wait()`

```
#!/usr/bin/perl -wl  
  
$pid = fork;  
  
if ($pid) {
```

```

    print "\t\t I am the parent. Waiting for my child to
    die...";

    wait;

    print "\t\t Now I'm all alone."; # only printed after
    the child dies
} elsif (defined $pid) {
    print "I am a child.";

    print "Enter some text. Press CTRL-D when done.";

    while (<>) {}

    exit;
} else { die "Couldn't fork: $! " }

```

Once created, the child process prompts the user for text while the parent process wait()s for the child process. When the user is done, the child process exits and the parent process continues, printing *Now I'm all alone*.

```
% waiter
```

```
RESULT I am a child.
```

```
I am the parent. Waiting for my child to die...
```

```
Enter some text. Press CTRL-D when done.
```

```
Frothy, flaky, crusty, and cakey
```

```
Mothy, snaky, dusty, and shaky
```

```
[CTRL]-[D] Now I'm all alone.
```

waitpid(), demonstrated in Listing 9-26, waits for a particular process, in this case the process with PID \$pid2.

Listing 9-26 *mywaiter*: Using waitpid() to wait for a particular process to exit

```

#!/usr/bin/perl -wl

print "Parent pid: $$";

$pid1 = fork;

if (defined $pid1 && !$pid1) {
    print "\t Firstborn child: $$";

    sleep 5;

    print "\t Firstborn child dying...";
}

```

```

        exit;
    }

    $pid2 = fork;

    if (defined $pid2 && !$pid2) {
        print "\t\t Secondborn child: $$";

        sleep 10;

        print "\t\t Secondborn child dying...";

        exit;
    }

    sleep 2;

    print "Waiting for secondborn child to die...";

    waitpid ($pid2, 0);

    print "Secondborn child died.\n";
% mywaiter

```

RESULT Parent pid: 21225

Firstborn child: 21226

Secondborn child: 21227

Waiting for secondborn child to die...

Firstborn child dying...

Secondborn child dying...

Secondborn child died.

If the parent process dies before the child, the UNIX *init* program inherits the child process. *init* is pretty good at reaping orphan processes.

But if the child process dies before the parent, there's a problem. If the child completely disappears, the parent is unable to fetch its termination status. The kernel has to keep a certain amount of information about every process so that it can provide that information to `wait()` or `waitpid()`. When a process terminates but its parent hasn't waited for it, it becomes a zombie process.

You can avoid zombie processes by ensuring that parents always wait for their children inside a handler for the `CHLD` signal, which is sent to a parent whenever its child dies:

```

use POSIX `WNOHANG`; # If Perl complains about POSIX, say
"sub WNOHANG { 1 }"

$SIG{CHLD} = sub {
    my($pid);

    while (1) {

```



```

        $pid = waitpid(-1, &WNOHANG);

        last if $pid < 1;
    }
};

```

If the WNOHANG flag is given to waitpid(), it returns 0 if there aren't any reapeable child processes. It won't wait around for a process to die: It returns immediately, regardless. That's a nonblocking operation.

What's a Blocking Operation?

A *blocking I/O operation* waits for something, such as input from the user or for a file lock to be released. system() is a good example--it waits for its command to finish before returning. A nonblocking operation never waits: If it succeeds, it succeeds immediately and if it fails, it fails immediately. For example, opening a file is nonblocking: If open() fails, it immediately returns a value. If open() were to wait until the file could be opened, that would be a blocking operation.

Pipes

So far, you've learned how to create processes, how to kill them, and how to wait for their death. But none of these functions lets processes talk to one another. The pipe() function does.

The pipe() Function

pipe(READHANDLE, WRITEHANDLE) opens a pair of connected pipes.

In most flavors of UNIX, pipes are half-duplex--the information flows only in one direction. So a process can either talk or listen; it can't do both. *1waypipe*, shown in Listing 9-27, demonstrates a pipe through which a parent sends text to a child.

Listing 9-27 *1waypipe*: Using pipe() to communicate between parent and child

```

#!/usr/bin/perl -w

use FileHandle; # only needed for WRITE->autoflush()

pipe(READ, WRITE); # The parent writes to WRITE
# and the child reads from READ

WRITE->autoflush();

$pid = fork;

if ($pid == 0) {
    print "\t\t\t I'm the child. Be nice to me! \n";

    close WRITE; # Be sure to close this end
}

```

```

        while (<READ>) {
            print "\t\t\t Parent said: $_";
        }
    } else {
        print "I'm the parent. Please yell at my child. \n";
        close READ; # and close this end too!

        while (<>) {
            print WRITE $_;
        }
    }
}

```

The filehandles READ and WRITE are connected to each other; anything written to WRITE can be subsequently read from READ. (They're called filehandles even though they don't correspond to actual files. To the UNIX kernel, they're all the same.)

```
% 1waypipe
```

```
RESULT I'm the parent. Please yell at my child.
```

```
I'm the child. Be nice to me!
```

```
Hello kiddie!
```

```
Parent said: Hello kiddie!
```

```
How are you?
```

```
Parent said: How are you?
```

In *1waypipe*, the parent can talk to the child but not vice versa, because pipes are only one way. *2waypipe* (Listing 9-28) creates a full-duplex connection by using two pipes: one pipe for communication from the parent to the child, the other for communication from the child to the parent.

Listing 9-28 *2waypipe*: Creating a full-duplex connection with two pipes

```

#!/usr/bin/perl -w

use FileHandle;

pipe(READ_FROM_PARENT, WRITE_TO_CHILD);
pipe(READ_FROM_CHILD, WRITE_TO_PARENT);

WRITE_TO_PARENT->autoflush();
WRITE_TO_CHILD->autoflush();

$pid = fork;

if ($pid == 0) {

```

```

close READ_FROM_CHILD;

close WRITE_TO_CHILD;

print "\t\t\t I'm the child. \n";

print WRITE_TO_PARENT "Mommy? \n";

while (<READ_FROM_PARENT>) {
    print "\t\t\t Parent said: $_ \n";

    sleep 1;

    print WRITE_TO_PARENT "Can I have another
    cookie? \n";
}
} else {
    close READ_FROM_PARENT;

    close WRITE_TO_PARENT;

    print "I'm the parent. \n";

    print WRITE_TO_CHILD "Hi there! \n";

    while (<READ_FROM_CHILD>) {
        print "Child said: $_ \n";

        print WRITE_TO_CHILD "Here you go. \n";
    }
}

% 2waypipe

```

RESULT I'm the parent.

I'm the child.

Child said: Mommy?

Parent said: Hi there!

Child said: Can I have another cookie?

Parent said: Here you go.

Child said: Can I have another cookie?

Parent said: Here you go.

Child said: Can I have another cookie?

Parent said: Here you go.

Child said: Can I have another cookie?

Parent said: Here you go.

[CTRL]-[C]

Notice that both processes write before they try to read. If they had both tried to read before writing, they would have become deadlocked--each waiting for the other.

When a process writes to a pipe with no readers, a PIPE signal is sent to that process. That kills the process, unless `$SIG{PIPE}` is trapped.

Open2 and Open3

The `IPC::Open2` and `IPC::Open3` modules (written by Tom Christiansen and Marc Horowitz) let you pipe data to and from an external program. That's NOT the same as using `pipe()` to send data between processes. You use `IPC::Open2` when you mean this:



```
open(PIPEBOTHWAYS, "| $command |");
```

which isn't legal Perl but is a reasonable thing to want to do. For instance, suppose you want to write data to *bc* (a UNIX command-line calculator) and read its output. Listing 9-29 shows how you'd do it.

Listing 9-29 *opencalc*: Using `open2()` to access standard input and output of an external program

```
#!/usr/bin/perl

use IPC::Open2;

$pid = open2('OUTPUT', 'INPUT', 'bc');

print INPUT "2 * 7 \n";

print "bc said: ", scalar(<OUTPUT>); # grab a single line of
output

print INPUT "5 ^ 6 \n";

print "bc said: ", scalar(<OUTPUT>);
```

opencalc creates two filehandles: `OUTPUT`, which reads the standard output of the program, and `INPUT`, which writes to the standard input of the program:

```
% opencalc
```

```
RESULT bc said: 14 <
```

```
> bc said: 15625
```

`Open3` behaves the same as `IPC::Open2`, but provides an extra filehandle for the `STDERR` of the external program, for example, `open3('OUTPUT', 'INPUT', 'ERROR', 'bc')`.

Chapter 9, Perl 5 Interactive Course

LESSON 6

UNIT 1

ADVANCED I/O

This lesson discusses three of the most intimidating functions in UNIX: `select()`, `fcntl()`, and `ioctl()`. These functions are normally used by C programmers to manipulate low-level characteristics of files, file descriptors, and terminals. What follows are brief descriptions of their syntax and simple examples of these functions so that you can see how they're used. For more detailed information, consult the online manual pages (for *select*, *fcntl*, or *ioctl*) or a book on UNIX programming, such as W. Richard Stevens' *Advanced Programming in the Unix Environment* (Reading, MA: Addison Wesley, 1992).

select()

Suppose you want a program to act as an intermediary between the user and some other program. You want to catch all the user's input (from `<>`) and send it to that other program. And you want to catch all that program's output (say, from the filehandle `<READ>`) and print it to `STDOUT`. Sounds like a job for pair of pipes, right? Well, yes, but there's a problem: A process can't listen to two filehandles at the same time, so it can read from either `<>` or `<READ>` but not both.

The `select()` function lets you get around this problem by telling you which filehandles are available for reading.

The `select()` Function

`select(RBITS, WBITS, EBITS, TIMEOUT)` lets you multiplex I/O channels.

You use `select()` all the time with one argument to change the default filehandle. This four-argument `select()` is a completely different beast. It lets you inspect three characteristics of filehandles: whether they're readable, writable, or have raised a UNIX exception. Sounds simple, right? There's a catch: You can't simply supply `select()` with the name of a filehandle. Instead, you have to provide a bit vector of file descriptors. That makes it possible to inspect many filehandles simultaneously, but it also makes `select()` more awkward to call because you have to assemble the *RBITS*, *WBITS*, and *EBITS* arguments yourself. *RBITS*, *WBITS*, and *EBITS* are bit vectors in which each bit corresponds to a filehandle:

- ***RBITS*: Readable filehandles**
- ***WBITS*: Writable filehandles**
- ***EBITS*: Filehandles that have raised UNIX exceptions (rarely used)**

The *nth* bit corresponds to the *nth* file descriptor. So if the fourth bit of *RBITS* is 1, `select()` checks whether file descriptor 4 is readable. `select()` modifies its arguments; it places the result in *RBITS* as well, so if the fourth bit of *RBITS* is 1 afterward, you'll know that the filehandle is ready for reading. (That's doesn't mean that data is present, only that there won't be an error if you try to read from the filehandle.) You can construct the bit vectors with `vec()` and `fileno()`.

The `fileno()` Function

`fileno(FILEHANDLE)` returns the file descriptor associated with *FILEHANDLE*.

The file descriptor is usually a small integer: `STDIN`, `STDOUT`, and `STDERR` are 0, 1, and 2, respectively. User-defined filehandles will have higher numbers.

So

```
vec($rbits, fileno(STDIN), 1) = 1;
```

sets the `STDIN` bit in `$rbits` to 1, and

```
vec($wbits, fileno(BERRY), 1) = 1;
```

sets the `BERRY` bit in `$wbits` to 1. Now you can `select()` to see whether `STDIN` is ready for reading and/or whether `BERRY` is ready for writing:

```
select($rbits, $wbits, undef, 0);
```

`$rbits` and `$wbits` now tell you what you want to know:

```
if (vec($rbits, fileno(STDIN), 1)) { print "STDIN is ready
for reading!" }
if (vec($wbits, fileno(BERRY), 1)) { print "BERRY is ready
for writing!" }
```

Because `select()` modifies its arguments, you'll find this form convenient:

```
select($rbits_out = $rbits, $wbits_out = $wbits, ...
```

which places the modified bits in `$rbits_out` and `$wbits_out`, leaving `$rbits` and `$wbits` unchanged. `select()` returns a two-element array: the number of ready filehandles and (when the operating system permits) the number of seconds remaining before *TIMEOUT* would have elapsed.

- If *TIMEOUT* is set to 0, the `select()` function polls, which means that it checks the filehandles and returns immediately.
- If *TIMEOUT* is `undef`, the `select()` function blocks, waiting until some file-handle becomes ready.

Suppose you want to check whether the `CASTOR` filehandle is ready for reading or writing and you're willing to wait no more than 12 seconds for an answer. Here's how you'd do it:

```
vec($rbits, fileno(CASTOR), 1) = 1;
vec($wbits, fileno(CASTOR), 1) = 1;
```

Each `vec()` statement sets a single bit to 1. Which bit? The one corresponding to the file descriptor for `CASTOR`, probably number 3 or thereabouts.

```
($ready, $timeleft) = select($rbits_out = $rbits,
                             $wbits_out = $wbits,
```

```
undef,
```

```
12);
```

EBITS is set to UNDEF because you don't care whether any descriptors have raised exceptions.

```
if ($ready == 0) { print "CASTOR is ready for neither reading  
nor writing." }
```

```
if ($ready == 1) { print "CASTOR is ready for reading OR  
writing." }
```

```
if ($ready == 2) { print "CASTOR is ready for reading AND  
writing." }
```

```
if ($timeleft) { print "There were $timeleft seconds left  
before TIMEOUT." }
```

```
if (vec($rbits_out, fileno(CASTOR), 1)) {print "CASTOR is  
ready for reading."}
```

```
if (vec($wbits_out, fileno(CASTOR), 1)) {print "CASTOR is  
ready for writing."}
```

selector, shown in Listing 9-31, tests five filehandles: three defined by the user and the two system filehandles STDIN and STDOUT.

Listing 9-31 *selector*: Using `select()` to check on several files

```
#!/usr/bin/perl -l

open (LOG, ">/tmp/logfile") or die "Can't write /tmp/logfile:
$! ";

open (WORDS, " cat /usr/dict/words |");

# If your computer doesn't have diction,
# any program that reads from standard input will do.

open (DICTION, "| diction");

@filenames = qw( LOG WORDS DICTION STDIN STDOUT );

@filenumbers = (fileno(LOG), fileno(WORDS), fileno(DICTION),
    fileno(STDIN), fileno(STDOUT));

foreach $i (@filenumbers) {
    vec($rbits, $i, 1) = 1; # check whether it's readable
    vec($wbits, $i, 1) = 1; # check whether it's writable
}
```

```
# Assigning $rbits_out to $rbits leaves $rbits intact.

# Likewise for $wbits_out, which indicates whether
filehandles are writable.

select($rbits_out = $rbits, $wbits_out = $wbits, undef, 0);

foreach $i (0..$#filenames) {
    print "$filenames[$i] is: ";

    print "\t readable" if vec($rbits_out,
        $filenumbers[$i], 1);

    print "\t writable" if vec($wbits_out,
        $filenumbers[$i], 1);
}
```

The answers might not be what you expect:

```
% selector
```

```
RESULT LOG is:
```

```
readable
```

```
writable
```

```
WORDS is:
```

```
writable
```

```
DICTION is:
```

```
writable
```

```
STDIN is:
```

```
writable
```

```
STDOUT is:
```

```
writable
```

You might be surprised that STDIN is writable but not readable. There's nothing preventing you from writing to STDIN; you just won't see the output, that's all. STDIN isn't readable because there's nothing to read.

If you send something to the standard input of selector, select() will notice.

```
% echo "This goes to STDIN" | selector
```

makes STDIN readable:

```
RESULT STDIN is:
```

```
readable
```

```
writable
```

As you saw in Chapter 4, select() can also be used to sleep for a fractional number of seconds, in which case *RBITS*, *WBITS*, and *EBITS* can all be UNDEF:

```
select(undef, undef, undef, 0.1);
```


sleeps for a tenth of a second.

File Control with fcntl()

Sometimes you'll want finer control of a file descriptor than `open()` permits. You might want to duplicate the file descriptor, or change a file descriptor from write-only to read-only without closing and reopening the file, or perform file locking on a system that doesn't support `flock()`. `fcntl()` gives you this control.

The fcntl() Function

fcntl(FILEHANDLE, REQUEST, ARGUMENT) applies **REQUEST** to **FILEHANDLE**.

If your system doesn't support `flock()`, Perl 5.001m and later versions will emulate it for you, using `fcntl()`. Table 9-2 shows the ten requests recognized by `fcntl()`.

Table 9-2 `fcntl()` requests

fcntl() Request	Description
F_DUPFD	Duplicates an existing descriptor
F_GETFD	Gets file descriptor flags
F_SETFD	Sets file descriptor flags
F_GETFL	Gets file status flags
F_SETFL	Sets file status flags
F_GETOWN	Gets asynchronous I/O ownership
F_SETOWN	Sets asynchronous I/O ownership
F_GETLK	Gets file lock
F_SETLK	Sets file lock
F_SETLKW	Waits to set file lock

For an explanation of what each `REQUEST` does and the `ARGUMENT` expected by each, consult your system's *fcntl* documentation. The program shown in Listing 9-32 demonstrates the use of one command: `F_GETFL`.

Listing 9-32 *filectrl*: Using `fcntl()` to get the file status flags

```
#!/usr/bin/perl -l
    use Fcntl; # This defines F_GETFL, O_RDONLY, etc.
open (FILE, "/tmp");

$val = fcntl(FILE, F_GETFL, 0);

print "/tmp is ";

print_status_flags($val);
```

```

open (FILE, "/tmp/sprocket");
$val = fcntl(FILE, F_GETFL, 0);
print "/tmp/sprocket is ";
print_status_flags($val);
open (FILE, "+</tmp/celluloid");
$val = fcntl(FILE, F_GETFL, 0);
print "/tmp/celluloid is ";
print_status_flags($val);
sub print_status_flags {
    my ($val) = $_[0];
    if ($val == O_RDONLY ) { print "\t read only " }
    elsif ($val == O_WRONLY ) { print "\t write only " }
    elsif ($val == O_RDWR ) { print "\t read + write " }
    if ($val & O_NONBLOCK) { print "\t non-blocking " }
    if ($val & O_APPEND ) { print "\t append " }
}

```

filectrl uses `fcntl()` to grab the status flags from three filehandles. Each time, the result is placed into `$val`, which is then passed to `print_status_flags()`, which contains the necessary magic needed to coax `$val` into English:

```

% filectrl
RESULT /tmp is
        read only
/tmp/sprocket is
        read only
/tmp/celluloid is
        read + write

```

I/O Control with `ioctl()`

`ioctl()` is the meat loaf of I/O operations. It doesn't have any single purpose other than as a catchall function for operations that you can't perform any other way.

The `ioctl()` Function

`ioctl(FILEHANDLE, REQUEST, ARG)` applies *REQUEST* to the I/O channel associated with *FILEHANDLE*.

The available *REQUESTs* vary from system to system; they generally affect attributes such as file I/O, terminal I/O, socket I/O, magnetic tape I/O, and disk labels. For more information, check the man page on your local system. Listing 9-33 shows an example of using `ioctl()` for terminal I/O.

Listing 9-33 *ioctlr*: Using `ioctl()` to fetch the window size

```
#!/usr/bin/perl -wl

# This file necessary to define &TIOCGWINSZ.
# If sys/ioctl.ph isn't found,
# you might want to do this as superuser:
# cd /usr/include; h2ph * sys/*
# See the h2ph documentation for more detail.
require "sys/ioctl.ph";

$SIG{WINCH} = `stat_window`;

sleep;

sub stat_window {
    $ret = ioctl(STDIN, &TIOCGWINSZ, $bytes) or die
        "ioctl failed: $! ";

    @size = unpack("ccccc", $bytes);

    print "Your window is $size[0] characters long,";
    print " and $size[2] characters high.";
}
```

Whenever the user changes the dimensions of the terminal window, a WINCH signal is sent to the program, which causes `stat_window()` to be called. `stat_window()` calls `ioctl()` with the request `&TIOCGWINSZ` (Terminal Input Output Change Window Size, if you must know) and argument `$bytes`, which is an empty string ready to be filled by `ioctl()`. The `unpack()` on the subsequent line breaks that string into its individual elements: four characters and a short integer. The first and third elements contain the length and width of the window. (That detail, and many more, should be in the `ioctl()` documentation for your system.)

```
% ioctlr
```

```
RESULT Your window is 80 characters long,
        and 24 characters high.
```

```
<lengthen window>
```

```
Your window is 80 characters long,
```

and 56 characters high.

It's quite possible that no one has run h2ph on your ioctl.h file or that h2ph didn't properly create &TIOCGWINSZ. In that case, consider using the Term::ReadKey module.

Term::ReadKey

If you want to read single characters or perform a nonblocking read of STDIN or just about any other task you'd use ioctl() for, check out Kenneth Albanowski's Term::ReadKey module, available from the CPAN (see Appendix B).

Term::ReadLine

Most shells let users manipulate their command lines. Various control characters will let you jump backward by a word or retrieve a previously typed command or delete everything until the end of the line. If you want your Perl scripts to let users do the same, use the Term::ReadLine module by Ilya Zakharevich and Jeffrey Friedl, available from the CPAN. Despite the names, this Term::ReadLine isn't quite the same thing as the Term::ReadLine module bundled with the Perl distribution, which acts as a front end to packages like Term::ReadLine. See the Term::ReadLine installation instructions for details.

Chapter 9, Perl 5 Interactive Course

LESSON 7

UNIT 1

syscall() AND COMPILING PERL

You've learned that `ioctl()` is a catchall for leftover I/O operations. This lesson is a catchall for this chapter. Two unrelated functions are covered: `syscall()` and `dump()`.

System Calls

Every flavor of UNIX has a kernel--the core of the operating system that ensures that programs behave. You don't want two programs allocating the same chunk of memory for personal use. You don't want unauthorized processes romping through the filesystem either. The kernel makes sure that such crises never occur.

Whenever a program requests "services" (such as memory allocation) from the kernel, it does so through a system call. Most of the time, you're never exposed to system calls; programming languages wrap them inside higher-level functions. Whenever Perl needs to, say, allocate memory (e.g., to create an array for you), it does so by making a system call. System calls are the lowest level operations that a program can perform. (They have nothing to do with Perl's `system()` function.)

Every once in a while, if you're very unlucky, you'll need to speak to the kernel directly with a (nonportable!) `syscall()`. As with most of the functions in this chapter, check your local documentation (e.g., *man 2 syscall*) for details.

The `syscall()` Command

`syscall(COMMAND, ARGS)` calls the system call *COMMAND*.

The *ARGS* depend on which *COMMAND* is used. The available commands vary from system to system and are (we hope) explained online. The arguments can be flags, strings, or byte structures that are filled by `syscall()` (in the same way that `ioctl()` fills `$bytes` in the *ioctl* program). If an *ARG* is a number, it is passed as a C integer. Otherwise, a pointer to the string value of *ARG* is passed.

Using `syscall()` requires a certain commitment on your part. For instance, if you allocate memory with the *sbrk* system call, you have to remember to deallocate that memory--Perl can't do it for you. Or if you provide a `pack()`ed string *ARG* that you want `syscall()` to fill, you have to ensure that the string is long enough. C programmers are used to such hassles; Perl programmers probably aren't.

Strictly speaking, none of Perl's functions is a system call. But some are closer to system calls than others: Perl's `print()` uses the C `fwrite()` function, which uses the `write()` system call, whereas Perl's

syswrite() allows you to access write() directly.

If you found Perl's mkdir() unacceptable for some reason, you could use the mkdir() system call instead, as shown in Listing 9-34.

Listing 9-34 sysmkdir: Using syscall() to access to the UNIX kernel directly

```
#!/usr/bin/perl -w

require "sys/syscall.ph";

$string = "newdir";

syscall(&SYS_mkdir, $string); # Create a directory named
newdir

# &SYS_mkdir is 136 on most UNIX systems.
```

Assuming that your system has *sys/syscall.ph* installed in the proper place and that your system supports the mkdir() system call, the require() should make &SYS_mkdir available. That function probably returns 136, which is passed to syscall() so it knows which function to perform. You might find a table of syscall() values in */usr/include/sys/syscall.h*.

```
% sysmkdir

% ls -ld newdir

d----- 2 newuser 1024 May 14 17:16 newdir
```

Compiling Perl

C is a *compiled language*: You must first compile a C program into an executable binary file before you can run it. LISP is a good example of an interpreted language: Expressions are evaluated as soon as you type them--you don't need a separate file containing your source code.

Perl isn't a good example of either. The program */usr/bin/perl* (or its equivalent on your system) reads your file and compiles it into a bytecode image in memory. No executable binary is created. Whenever you run a Perl program, you must first wait for */usr/bin/perl* to be loaded into memory and then wait for it to compile your program. By compiling your programs yourself into either an executable binary, or an intermediate form more palatable to Perl, you can save a little time.

The Perl Compiler

A Perl compiler written by Malcolm Beattie is now available as part of the CPAN. It produces, at your discretion, either platform-independent bytecode or C source code. The bytecode can be read by your */usr/bin/perl* much quicker than normal Perl programs, with the aid of a special module included with the compiler kit. The C source code can be compiled into an executable binary with a regular C compiler. However, you'll sacrifice portability if you do this: A binary compiled for one architecture won't run on another.

The Perl compiler is the new, good way to speed up your programs. The old way is with `dump()`.

The `dump()` Command and the `-u` Command-Line Flag

`dump(LABEL)` creates a core file containing the contents of the process.

`-u` causes the program to **`dump()`** as soon as it has been compiled.

The resulting core file can--sometimes--be coaxed into an executable binary. If `LABEL` is supplied, the binary will restart from `LABEL`. Otherwise, it will restart from the beginning of the program.

`dump()` kills a dynamic process and leaves a static corpse: a core file on your disk. The corpse can be resurrected, but there's a catch: You need an architecture-dependent program called *undump* to turn the core file into an executable binary. *undump* is NOT supplied with Perl, but you might be able to find a copy in the TeX distribution. Check <ftp://ftp.cis.ufl.edu/pub/perl/doc> for more information.

Listing 9-35 shows a program that uses `dump()`.

Listing 9-35 coredump: Using `dump()` to create a core file

```
#!/usr/bin/perl -w

use English;
use FileHandle;

print "Morituri te salutamus\n";

dump HERE;

print "This won't ever be printed.";

HERE: print "Eight lives left...";

% coredump

Morituri te salutamus

IOT/Abort trap (core dumped)

% ls -l core

-rw----- 1 perluser 100 2654208 May 15 02:52 core
```

If you can *undump* that core file and turn it into an executable binary on your system, you might find that the program starts up more quickly. When it restarts, it resumes execution at the `HERE` label, not where the `dump()` occurred.

`dump()` freezes your process, not the entire operating system. When it restarts, it'll be a completely separate program--in particular, any open file descriptors in your program will be lost.

Chapter 9, Perl 5 Interactive Course

LESSON 8

SYSTEM V INTERPROCESS COMMUNICATION

You've already seen some ways that different processes can communicate with one another: `pipe()` is an example. But pipes only facilitate communication between processes with a common parent--if two users run programs on the same computer, they can't simply connect pipes to one another. System V interprocess communication (IPC) routines let any two processes on the same computer communicate. They don't even have to know one another's PIDs.

System V IPC is really three distinct styles of interprocess communication: message queues, semaphores, and shared memory. (Why is it called System V? That's a particular flavor of UNIX, but that hasn't kept other operating systems from adopting the routines.) Everything in this lesson depends on these routines. If your system doesn't support them (few non-UNIX systems do), you're done with this chapter.

In the future, these functions may migrate to an `IPC::SysV` module.

Message Queues

A *message queue* is a list of in-memory messages stored by a process. Think of it as a bulletin board--one process can tack a note to the bulletin board and other processes can read that note.

The `msgget()`, `msgsnd()`, `msgrcv()`, and `msgctl()` Functions

`msgget(KEY, FLAGS)` gets or creates a new message queue, returning a *QUEUE ID*.

`msgsnd(ID, MSG, FLAGS)` sends a message to *QUEUE*, returning TRUE if successful.

`msgrcv(QUEUE, VAR, SIZE, TYPE, FLAGS)` receives a message from *QUEUE*, returning TRUE if successful.

`msgctl(ID, COMMAND, ARG)` performs some *COMMAND* on *QUEUE*.

To demonstrate message queues, we'll use two separate programs: *talker* and *listener*. *talker* creates a message queue and places a message in it for *listener* to read.

Both programs agree on a numeric \$key, in this case 999, used to identify the message queue. First, *talker* creates the message queue by calling `msgget()` with a bitwise OR of `IPC_CREAT` (which tells UNIX to allocate a message queue) and octal 777 (which tells UNIX that anyone is allowed to write to the queue). Then *talker* (Listing 9-36) stores two messages: *Riddles and rainbows* and *Brambles and barrows*.

Listing 9-36 *talker*: Creating a queue and storing two messages in it

```
#!/usr/bin/perl -wl

# These might be replaced by IPC::SysV someday
require `sys/ipc.ph`;
require `sys/msg.ph`;

# Use these next two lines if the previous require()s fail.
eval `sub IPC_CREAT {0001000}` unless defined &IPC_CREAT;
eval `sub IPC_NOWAIT {0004000}` unless defined &IPC_NOWAIT;

$key = 999; # any number will do

$queue = msgget($key, &IPC_CREAT | 0777); # create the queue

$message = pack("La*", 0, "Riddles and rainbows");

msgsnd($queue, $message, &IPC_NOWAIT) or die "Can't send
message: $! ";

$message = pack("La*", 0, "Brambles and barrows");

msgsnd($queue, $message, &IPC_NOWAIT) or die "Can't send
message: $! ";

print "Good night!";

sleep;
```

Now that *talker* has stored the messages, *listener* (Listing 9-37) can receive them with one call to `msgrcv()` for each slot. But first it has to determine where the queue is located. That information comes from `msgget()`, invoked with no flags.

Listing 9-37 *listener*: Reading from a queue and deleting it

```
#!/usr/bin/perl -l

# These might be replaced by IPC::SysV someday
require `sys/ipc.ph`;
require `sys/msg.ph`;

# Use these next two lines if the previous require()s fail.
eval `sub IPC_NOWAIT {0004000}` unless defined &IPC_NOWAIT;
eval `sub IPC_RMID {0}` unless defined &IPC_RMID;
```

```

$key = 999; # use the same as talker

$queue = msgget($key, 0); # find out the queue id

# Grab the first message

msgrcv($queue, $var, 80, $slot, &IPC_NOWAIT) or die "Can't
rcv message: $! ";

print $var;

# Grab the second message

msgrcv($queue, $var, 80, $slot, &IPC_NOWAIT) or die "Can't
rcv message: $! ";

print $var;

# Delete the queue

msgctl($queue, &IPC_RMID, 0);

```

Finally, when *listener* has read the messages, it invokes `msgctl()` to delete the message queue by supplying `IPC_RMID` (Remove ID) as an argument.

```
% talker &
```

```
RESULT Good night!
```

```
% listener
```

```
RESULT Riddles and rainbows
```

```
Brambles and barrows
```

In spite of their names, *talker* and *listener* can both send and receive messages--the communication need not be one way.

Semaphores

Ever been to a meeting where everyone wanted to talk at once? Here's a solution: Grab some object and declare that only the person with the object can speak. When that person is done speaking, he or she hands it to someone else.

That's how semaphores work. A *semaphore* is a counter that a collection of processes uses to determine which process can access some resource. A process that wants the "rights" to that resource waits until the semaphore counter reaches a particular value, at which time it locks the semaphore by changing the counter to an intermediate value. Then it manipulates the data; when it finishes, it releases the lock by setting the counter to some other value.

The `semget()`, `semop()`, and `semctl()` Functions

`semget(KEY, VALUE, SIZE, FLAGS)` gets or creates a semaphore, returning a *SEMAPHORE* ID.

`semop(KEY, OPERATION)` applies the *OPERATION* to the *KEY* semaphore.

`semctl(SEMAPHORE, VALUE, COMMAND, ARG)` performs some *COMMAND* on *SEMAPHORE*.

`semget()` and `semctl()` are similar to their message queue analogs. `semop()` is used to change the value of the semaphore counter.

Like message queues, semaphores are best demonstrated with two separate programs, each using a shared key, in this case 1066. Below, *left* and *right* alternate printing to STDOUT. *left* creates the semaphore with `semget()`. Then it waits until the semaphore counter becomes 0, at which time it prints *Left!* and then sets the semaphore counter to 2, as shown in Listing 9-38.

Listing 9-38 *left*: Creating a semaphore and incrementing its value

```
#!/usr/bin/perl -wl

# These might be replaced by IPC::SysV someday

require `sys/ipc.ph`;
require `sys/sem.ph`;

# Use these next two lines if the previous require()s fail.
eval `sub IPC_CREAT {0001000}` unless defined &IPC_CREAT;
eval `sub IPC_EXCL {0002000}` unless defined &IPC_EXCL;
eval `sub IPC_RMID {0}` unless defined &IPC_RMID;

$key = 1066; # any number will do <

$| = 1;

$num = 0;

$flags = 0;

# Create the semaphore.

$id = semget($key, 1, &IPC_EXCL|&IPC_CREAT|0777) or die
"Can't semget: $! ";

foreach (1..5) {
    $op = 0;

    $operation = pack("s*", $num, $op, $flags);

    semop($id, $operation) or die "Can't semop: $! ";

    print "Left!";

    sleep 1;
```

```

    $op = 2;

    $operation = pack("s*", $num, $op, $flags);
# Add 2 to the semaphore (now 2)
    semop($id, $operation) or die "Can't semop: $! ";
}

semctl($id, 0, &IPC_RMID, 0); # delete semaphore

```

right (Listing 9-39), meanwhile, gains control once *left* sets the semaphore to 2. It prints *Right!* and resets the semaphore to 0. This continues until *left* finishes its loop, after which it destroys the semaphore with `semctl()`.

Listing 9-39 *right*: Printing *Right!* and resetting the semaphore

```

#!/usr/bin/perl -wl

$key = 1066; # use the same number as left
$| = 1;
$num = 0;
$flags = 0;

# Identify the semaphore created by left
$id = semget($key, 1, 0) or die "Can't semget: $! ";
    foreach (1..5) {
        $op = -1;

        $operation = pack("s*", $num, $op, $flags);
# Add -1 to the semaphore (now 1)
        semop($id, $operation) or die "Can't semop: $! ";

        print "Right!";

        sleep 1;

        $operation = pack("s*", $num, $op, $flags);
# Add -1 to the semaphore (now 0)
        semop($id, $operation) or die "Can't semop: $! ";
    }
% left &
RESULT Left!
% right

```

RESULT Right!

Left!

Right!

Left!

Right!

Left!

Right!

Left!

Right!

left and *right* pass control back and forth with a binary semaphore. But because System V semaphore counters can be set to any value, you can create more complex behaviors, as you might if you were organizing communication between more than two processes.

Shared Memory

System V shared memory lets multiple processes utilize the same memory segment. This form of IPC is faster than the other methods in this chapter because no data needs to be copied from one process to another.

The `shmget()`, `shmread()`, `shmwrite()`, and `shmctl()` Functions

`shmget(KEY, SIZE, FLAGS)` gets or creates a shared memory *SEGMENT* and returns it.

`shmwrite(SEGMENT, STRING, POSITION, SIZE)` writes the *SIZE*-byte *STRING* to the *SEGMENT* at *POSITION*.

`shmread(SEGMENT, VAR, POSITION, SIZE)` reads a *SIZE*-byte *STRING* from the *SEGMENT* starting at *POSITION*.

`shmctl(SEGMENT, COMMAND, ARGUMENT)` performs some *COMMAND* on *SEGMENT*.

The *shm* functions are similar to their msg counterparts. These four functions are demonstrated with *writer* (Listing 9-40) and *reader* (Listing 9-41). *writer* first creates an area of shared memory, retrievable by the predefined \$key, 80. It then places a message in that memory segment with `shmwrite()` and goes to sleep for 20 seconds. Upon awakening, it deletes the memory segment by calling `shmctl()`.

Listing 9-40 *writer*: Creating a shared memory segment and writing to it

```
#!/usr/bin/perl -w

# These might be replaced by IPC::SysV someday

require `sys/ipc.ph`;
```

```

require `sys/shm.ph`;

# Use these next two lines if the previous require()s fail.
eval `sub IPC_CREAT {0001000}` unless defined &IPC_CREAT;
eval `sub IPC_RMID {0}` unless defined &IPC_RMID;

$key = 12345;

$size = 80; # Larger than necessary, but that's okay.

$message = "Pennyfarthingale.";

# Create the shared memory segment

$key = shmget($key, $size, &IPC_CREAT | 0777) or die "Can't
shmget: $! ";

# Place a string in it

shmwrite($id, $message, 0, 80) or die "Can't shmwrite: $! ";

sleep 20;

# Delete it.

shmctl($id, &IPC_RMID, 0) or die "Can't shmctl: $! ";

```

reader (Listing 9-41) retrieves the memory segment corresponding to `$key` and reads its contents using `shmread()`.

Listing 9-41 *reader*: Reading from a shared memory segment

```

#!/usr/bin/perl

$key = 12345;

$size = 80;

# Identify the shared memory segment

$id = shmget($key, $size, 0777) or die "Can't shmget: $! ";

# Read its contents into a string

shmread($id, $var, 0, $size) or die "Can't shmread: $! ";

print $var;

% writer &

% reader

```

RESULT Pennyfarthingale.

As with all the techniques in this lesson, the communication need not be one way. Both *writer* and *reader* can read from and write to the shared memory segment.

Review

Susan: Okay, now I've got a seperate prcess for my monster, using `fork()` and `pipe()`.

Geena: You've prbably got two pipes, right? One for the monster process to send data to the parent process, and one for your game to sendtat to the monster process.

Susan: Right. And when the player kills the monster, the game `kill()`s the monster process.

Geena: And then the game `wait()`s for the monster to die?

Susan: Yup! My game's been pretty popular around here. Now I want everyone on the Internet to play it.

Geena: Then you'll need the networking techniques in Chapter 13.

Chapter 10, Perl 5 Interactive Course



WHEN GOOD PROGRAMS GO BAD: DEBUGGING

No one writes perfect programs on the first try. Creating a robust program is an interactive experience: You write a chunk of code, try it out, write a little more, fix a bug, test it, and repeat until satisfied. One of Perl's strengths is the speed with which you can write and test programs (because there aren't any lengthy compile or link phases); another is its debugging facilities, which help you track down programming errors.

In this chapter, you'll find a guide to help you decide where to begin debugging--an experience that is often arduous, sometimes educational, and always humbling. This chapter is all about how to make your programs better: how to remove bugs, how to handle error conditions, and how to speed up execution. You'll learn about the symbolic debugger supplied with Perl and about a compilation

option that allows you to trace the execution of your program step by step.



In this chapter, you'll look at lots of buggy code. To distinguish this lethal code from the good stuff, it's marked with a skull and crossbones icon, just like the one appearing in the margin here.

When all else fails, post a plea for help to the Usenet newsgroup `comp.lang.perl.misc`. The Perl community has plenty of altruists who enjoy answering questions.

Of course, if you never make mistakes, you can skip this chapter.

HOW TO FIX IT

There are two debugging tools that are a snap to use and that uncover more bugs than you might expect: the **-w** command-line flag and the *strict* pragma (described in Chapter 8, Lesson 1). And don't underestimate the value of `print()` statements for tracking changes in variables and showing you when subroutines are entered and exited.

Warnings

If you learn nothing else from this chapter, learn this: **Use the -w flag**. It tells Perl to complain about code that looks wrong but doesn't necessarily merit a fatal error message. **-w** will uncover about three-quarters of the bugs you'll make and is the first tool you should turn to when debugging.

The -w Flag

The **-w** flag warns you about suspicious-looking code.

-w catches run-time errors, that is, errors triggered as your program runs.

Suppose you type `$array` when you really mean `@array`.

```
% perl -e `@array = (1,2,3); print $array`
```

won't print anything, but

```
% perl -we `@array = (1,2,3); print $array`
```

complains.

RESULT Use of uninitialized variable at -e line 1.

How many errors can you find in *warnme*, shown in Listing 10-1?

Listing 10-1 *warnme*: Using -w to find possible errors



```
#!/usr/bin/perl -w
$switch = "on";
$ARGV[0] != "string";
```

The **-w** switch finds four errors.

```
% warnme
```

RESULT Useless use of numeric ne in void context at warnme line 5.

Identifier "main::switch" used only once: possible typo at warnme line 3.

Argument "string" isn't numeric for numeric ne at warn line 5.

Use of uninitialized value at warn line 5.

Inside a program, you can determine whether the **-w** flag is present with the `$$W` special variable.

The `$$W` Special Variable

`$$W` (or `$WARNING`) is TRUE if the **-w** flag is used.

Also consider using the *strict* pragma, described in Chapter 8, Lesson 1. If you find it too restrictive (many programmers do), use whichever portions you can.

- **use strict subs flags barewords that might be interpreted as either strings or subroutine calls. It's almost always worthwhile.**
- **use strict refs outlaws symbolic references. Use it unless you're planning on using symbolic references.**
- **use strict vars makes you scope all your variables with `my()` or `local()`. It can mean a lot of extra typing, but if you use it, you'll never make scoping mistakes.**

Remember that you can **use strict** at the beginning of your program and then turn it off inside a block. So if you use symbolic references inside a subroutine but nowhere else, place **use strict** at the top of your program and **no strict refs** at the top of the subroutine.

Unfortunately, a few of the standard modules supplied with some Perl distributions don't adhere to **strict**. If you **use strict** and one of those modules, you'll see a flood of unsuppressible warnings from code that you didn't write and that you can't modify. But that's changing; soon all standard Perl modules will obey **strict**.

So where do you go from here? You can use Figure 10-1 as a roadmap for this chapter.

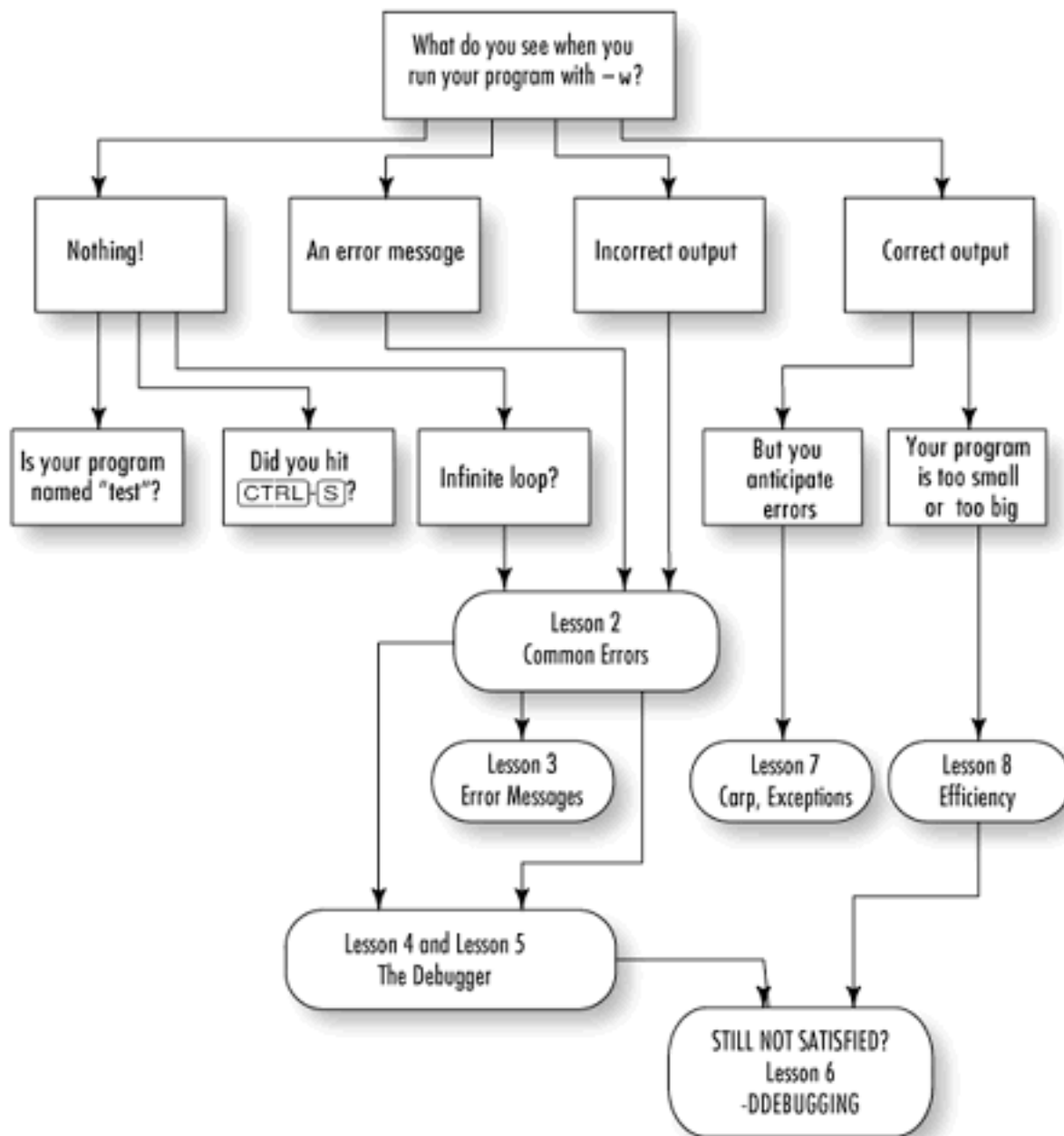


Figure 10-1

How to fix it

Checking Syntax

The -c Flag

The **-c** flag compiles your Perl script and checks its syntax.

You might think, "Why go to all the trouble of using the **-c** option? If there's a syntax error, Perl will tell me." True, but it will also run your script, which you might not want. The **-c** flag *compiles* your script (which executes *use* statements and **BEGIN** blocks) but exits immediately after compilation.

For instance, you don't want to test-drive programs that perform some irreversible action, such as generating electronic mail, as shown in Listing 10-2.

Listing 10-2 *syntax1*: Using -c to check the syntax without actually running the program



```
#!/usr/bin/perl -c
BEGIN { print "I´m printed during compilation. \n" }
open (MAIL, "| /usr/lib/sendmail -t -fperluser");
print MAIL "To: boss\@waite.com \n";
print MAIL "From: perluser \n\n";
print MAIL "I´m running the program, boss! Ribbit! \n";
close MAIL;
print "Mail sent! \n";
```

The -c flag uncovers the error without embarrassing the programmer. (But note that the BEGIN block was executed, printing *I'm printed during compilation*.)

```
% syntax1
```

RESULT I´m printed during compilation.

```
syntax error in file ./syntax1 at line 6, next token "print "
./syntax1 had compilation errors.
```

There's a semicolon missing. Let's add it (Listing 10-3).

Listing 10-3 *syntax2*: Adding a semicolon to fix the error

```
#!/usr/bin/perl -c
BEGIN { I´m printed during compilation. \n" }
open (MAIL, "| /usr/lib/sendmail -t -fperluser");
print MAIL "To: boss\@waite.com \n";
print MAIL "From: perluser \n\n";
print MAIL "I´m running the program, boss! Ribbit! \n";
close MAIL;
print "Mail sent! \n";
```

Even after the semicolon is added, -c plays it safe. The program is compiled (executing the BEGIN block) without being run.

```
% syntax2
```

RESULT I´m printed during compilation.

```
syntax2 syntax OK
```

Diagnostics

Tom Christiansen has written a **diagnostics** pragma (available from <http://mox.perl.com/pub/perl/ext/diagnostics.shar.gz> and the CPAN) that elaborates on Perl's error messages. The pragma (and its associated standalone program, **splain**) displays the explanations from the

perldiag documentation in addition to the standard error messages. So if you use `$my_var` only once in your program, you'll see the usual message:

```
RESULT Identifier "main::my_var" used only once: possible typo at
./my_prog line 7
```

but you'll also get

```
RESULT (W) Typographical errors often show up as unique identifiers. If
you had a good reason for having a unique identifier, then just
mention it again somehow to suppress the message.
```

Bugs in Perl

Perl isn't perfect. As proof, there's a list of bugs at <http://www.perl.com/perl/bugs/>. If you think you've found another one, send mail to perlbugs@perl.com so that the bug can be fixed in future releases.

Chapter 10, Perl 5 Interactive Course

LESSON 2

PERL PITFALLS

This lesson covers Perl pitfalls, organized by category and from most common to least common within each category. Consult the *perltrap* documentation for a more comprehensive list.

Lists Versus Scalars

- **`($x) = (3,4,5)` sets `$x` to 3, but `$x = (3,4,5)` sets `$x` to 5.**

The first statement is evaluated in a list context, so `=` grabs as many elements from its right side as it can fit into the left side: one.

The second statement is evaluated in a scalar context, so `=` grabs the last array element.

- **`$x[0]` is a scalar; `@x[0]` is a list.**

If it starts with an `@`, it's a list. Period. If it starts with a `$`, it's a scalar. Period.

- **It's `$#array`, not `#$array`.**

If it starts with a `$`, it's a scalar. Period. If it's a scalar, it starts with `$`. Period.

Functions Seek Out Argument Lists

Like most functions, `print()` gobbles up as many arguments as it can:

```
print 3, 4, 5, 6;
```

prints 3456, although

```
print (3), 4, 5, 6
```

prints just 3, since the parentheses around (3) make it look like an argument list. That's why

```
print 3, reverse 4, 5, 6
```

prints 3654, whereas

```
print 3, reverse (4, 5), 6
```

prints 3546.

Functions

- **chop()** returns the chopped-off character, not the string.

Use **chop(\$answer = <>)**, not **\$answer = chop(<>)**. Better yet, use **chomp()**.

- **srand()** once so you can **rand()** many times.

srand() seeds (initializes) the random number generator. You should do it once, at the beginning of any program that uses **rand()**.

- **print()** doesn't want a comma after the filehandle.

```
print STDOUT 'hi', 'there'
```

is correct;

```
print STDOUT, 'hi', 'there'
```

is not.

- **write()** is not the opposite of **read()**.

(Though **syswrite()** is the opposite of **sysread()**.) **write()** is for formats.

Syntax and Logic

- Every data type has its own namespace.

\$var has nothing to do with **@var**, or **%var**, or **var**. And **\$var** has nothing to do with **\$var[1]**, either. ***var** indirectly refers to all of them, however.

- Library files should always return **TRUE**.

That's why they often have this statement at the end:

```
1;
```

- **print (4+5), 6 prints 9; print 4+5, 6 prints 96.**

The parentheses make **4+5** look like the sole argument for **print()**.

- Semicolons are required after all statements, except at the end of a block.

But it never hurts to add them: **{ ;;; print "hi";;;; print "there";;;;;; };;;**

- Arrays and strings start at **0**, not **1**.

(You can change that with \$[, but you shouldn't, because most programmers assume that arrays and strings always begin at 0.)

- **foreach \$key (@array) works; foreach \$key @array doesn't.**

foreach needs the parentheses.

- **0 and ' ' are FALSE.**

buggy1 (Listing 10-4) doesn't print *17021* as you might expect:

Listing 10-4 *buggy1*: Defined values aren't always TRUE



```
#!/usr/bin/perl
@array = (17, 0, 21);
while ($array[0]) {
    print $array[0];
    shift @array;
}
```

because when \$array[0] becomes 0, the loop exits. Just because something is defined doesn't mean it's TRUE.

```
% buggy1
```

```
RESULT 17
```

- **&Class->method and Class->method and Class::method are different.**

Class->method is the proper way to invoke a method. Class::method and &Class->method are function calls, NOT method calls!

Operators

- **Don't use == when you mean *eq*, or != when you mean *ne*.**

```
"eq" eq "eq"
```

and

```
7 == 7
```

but

```
"six" == "seven"
```

because strings evaluate to 0 in a numeric context.

```
"6" != "7"
```

...unless they're obviously numbers.

- **Don't use = when you mean ==.**

= assigns; == tests.

● **Don't use & when you mean &&, or | when you mean ||.**

& and | are bitwise operators; && and || are logical operators.

● **Regexes use =~, not ~=.**

There's no such operator as ~=.

● **Don't use = when you mean =~.**

`($result) = ($string =~ /(\w)\s(\w)/)` matches the scalar \$string against the regular expression and sticks the first \w in \$result.

● **1..9 and a..z work as you'd expect, but 1..z doesn't.**

The .. operator performs an implicit increment unrelated to ASCII ordering. Only letters and numbers can be incremented. If you want an array of all the ASCII characters between 1 and z, do this:

```
map(chr, ord('1')..ord('z'))
```

● **Here-strings need semicolons and spaces.**

```
print <<END; # Neither "<< END;" nor "<<END" work.
hello, world!
END
print "\n";
```

Just because that first print() has a here-string doesn't mean it's not a statement. You need to tell Perl where the statement ends by putting a semicolon after the *first* END.

● **qw expects a string, not a list.**

Don't use commas unless you mean it:

```
qw( word1 word2 word3 )
```

is probably what you want, not

```
qw(word1, word2, word3)
```

which retains the commas.

Regular Expressions

● **/\w*/ matches every string.**

* means "zero or more." Every string has zero or more \w's (and zero or more \d's, and zero or more bananas...). Use the anchors ^ and \$ whenever possible. (Your matches will be faster, too.)

- **Dot (.) means any character EXCEPT \n.**

```
($greeting) = ("hello\n" =~ /(.*))/;
```

.sets \$greeting to hello, not hello\n. If you want . to match \n, use the \s modifier.

- **Use backslashes only when appropriate.**

Backslash weird characters to get their normal meaning; backslash normal characters to get their weird meaning. Except inside brackets.

- **Symbols have different meanings inside brackets.**

For instance, [^g] means "anything that's not a g." And [.] means "period," not "any character."

Precedence

- **-2 ** 2 is -4, not 4.**

** binds more tightly than unary minus (e.g., -2). That's why

```
-2 ** 2
```

is equivalent to

```
-(2 ** 2)
```

- **2 ** 3 ** 2 is 512, not 64.**

** is right-associative, so

```
2 ** 3 ** 2
```

is equivalent to

```
2 ** (3 ** 2)
```

- **1 || 0 && 0 is TRUE.**

&& has higher precedence than ||, so 1 || 0 && 0 is equivalent to 1 || (0 && 0).

- **Whitespace never affects precedence.**

```
1 || 0 && 0
```

is misleading. This expression is still equivalent to 1 || (0 && 0).

Scoping

- **local() variables are visible to subroutines; my() variables aren't.**

local() variables are *dynamically scoped*: they're defined not just in the current subroutine, but in "deeper" subroutines as well. my() variables are *lexically scoped*: visible within only the current subroutine.

- **my(\$_) is illegal.**

Only alphanumeric variables can be lexically scoped.

- **my \$x, \$y declares only one variable.**

```
my $x = 4
```

is okay, but

```
my $x, $y = (4, 8)
```

isn't. That's equivalent to

```
my($x), $y = (4, 8)
```

Sorting

- **sort() uses ASCII ordering by default.**

ASCII ordering is *not* the same as numeric ordering: *1000* is alphabetically less than *9*.

- **Use \$a and \$b, not \$_[0] and \$_[1].**

Sort subroutines are handled differently than other subroutines. For efficiency reasons, they don't use input arrays, but instead compare \$a and \$b.

- **The first argument to sort() should be the name of a subroutine or a subroutine definition.**

These are okay:

```
sort mysub @array;  
sort { $a <=> $b } @array;
```

These are not:

```
sort mysub() @array;  
sort sub { $a <=> $b } @array;
```

Chapter 10, Perl 5 Interactive Course

LESSON 3

ERROR MESSAGES

The *perldiag* documentation has a comprehensive list of error messages. In this lesson, you'll see a couple dozen of the error messages that programmers encounter the most, along with sample programs that produce the errors, so that you can avoid them!

Array found where operator expected...



```
#!/usr/bin/perl
foreach $key @array {
    print $key;
}
RESULT Array found where operator expected at program line 2,
near "$key @array"
```

You need parentheses around @array.

Can't bless non-reference value...



```
#!/usr/bin/perl
package Movie;
sub new {
    my @array = (1,2,3,4);
    bless @array;
}
package main;
$film = new Movie;
RESULT Can't bless non-reference value at program line 5.
```

You can't transform a plain old array into an object, only referenced values:

```
my $arrayref = [1, 2, 3, 4];
bless $arrayref;
```

Can't coerce...



```
#!/usr/bin/perl
*glob += 1;
```

RESULT Can't coerce GLOB to number in addition at program line 2.

This means that Perl can't pretend that `*glob` is actually a number, so it can't add 1 to `*glob`.

Can't find string terminator...



```
#!/usr/bin/perl
$var .= "all the way`
```

RESULT Can't find string terminator `` anywhere before EOF at program line 2.

Every quoting symbol needs its twin: Every double quote needs another double quote, every single quote needs another single quote, and every backquote needs another backquote.

Can't locate...



```
#!/usr/bin/perl
require `somefile.ph`;
```

RESULT Can't locate somefile.ph in @INC (did you run h2ph?) at program line 2.

Perl couldn't find *somefile.ph* in any of your *LIB* directories (they're listed in `@INC`). *h2ph* is a program bundled with Perl that tries to convert a C header file to Perl. (It doesn't always do a great job, however.)

In most Perl distributions, *h2ph* needs to be run after Perl is installed, but often isn't run, leading to these error messages. The online documentation for *h2ph* tells you to do this:

```
cd /usr/include; h2ph * sys/*
```

That converts all the *.h* files in */usr/include* and */usr/include/sys* to their *.ph* equivalents and stores them in your *LIB* directory.

Can't locate...



```
#!/usr/bin/perl
use ThisDoesNotExist;
```

RESULT Can't locate ThisDoesNotExist.pm in @INC at program line 2.

BEGIN failed--compilation aborted at program line 2.

Perl can't find the *ThisDoesNotExist* module because it's not in any of the directories mentioned in `@INC`. Assuming you didn't mistype the module name, your library files probably have been installed

somewhere else, or not at all. (Maybe the Perl binary was copied from a system with a different *LIB* directory.) Check all the possible *LIB* directories on your filesystem: */usr/local/lib/perl5*, */opt/perl5*, */usr/gnu/lib/perl5*, and so on. If you find the files, link them to one of the *@INC* directories. If you can't find them, consider reinstalling the latest version of Perl.

You can add directories to *@INC* if you wish, but only inside a *BEGIN* block.

Can't modify constant item...



```
#!/usr/bin/perl
6 = 7;
```

RESULT Can't modify constant item in scalar assignment at program line 2, near "7;"

You can't set 6 equal to 7. That would have grave consequences for all of mathematics.

Can't return outside a subroutine...



```
#!/usr/bin/perl
foreach $i (1..10) {
return if $i == 7;
}
```

RESULT Can't return outside a subroutine at program line 3.

Only subroutines can return. You probably meant *last*.

Can't write to temp file for -e

```
perl -e `print 100/8`;
```

RESULT Can't write to temp file for -e.

If you see this message, you were probably trying to execute a Perl statement from the command line with the *-e* flag. Perl then tried to create a temporary file to run your program, but couldn't. Your disk is probably full.

Deep recursion on subroutine...



```
#!/usr/bin/perl -w
sub recurse { recurse($_[0]) }
recurse(8);
```

RESULT Deep recursion on subroutine "recurse" at program line 2.

The `recurse()` subroutine calls itself forever; the warning is Perl's way of saying, "Hey! I've called myself over and over and over again. You sure this isn't an infinite loop?" It shows up only if you use the `-w` flag.

...follows nothing in regexp



```
#!/usr/bin/perl
$string = /?*/;
```

RESULT `/?*/: ?+*{}` follows nothing in regexp at program line 2.

If you mean zero or more question marks, use a backslash: `/\?*/`.

Infinite Loop 1 (not an actual error message)



```
#!/usr/bin/perl -w
for ($i = 1; $i != 10; $i += 2) {
}
```

`$i` never becomes equal to 10 because its values follow the sequence 1,3,5,7,9,11,13...

Infinite Loop 2 (not an actual error message)



```
#!/usr/bin/perl -w
while ($i = 7) {
    $i++;
}
```

Perhaps you mean `$i == 7`.

`$i = 7` always evaluates to TRUE, so the loop never exits.

Infinite Loop 3 (not an actual error message)



```
#!/usr/bin/perl -w
$_ = "volvox";
while (./g) {
    $_ .= '!';
}
```

The `./g` match returns TRUE and lops off the first character. Then the loop body tacks on another character.

Infinite Loop 4 (not an actual error message)



```
#!/usr/bin/perl -w
do `program`;
```

If this program were called *program*, it would *do* (read and execute) itself, which would *do* itself, which would *do* itself, which would...

Infinite Loop 5 (not an actual error message)



```
#!/usr/bin/perl -w
$a = `eval $a`;
eval $a;
```

\$a is evaluated, which evaluates \$a, which evaluates \$a, which evaluates \$a...

Literal array now requires backslash at program line 2, within string



```
#!/usr/bin/perl
print "user@waite.com";
RESULT Literal @waite now requires backslash at program line
2, within string
```

To a double-quoted string, @waite looks like an array, not an e-mail address. You should either backslash the @ or use single quotes instead.

Modification of a read-only value...



```
#!/usr/bin/perl
sub routine { $_[0] = 7; }
routine(6);
RESULT Modification of a read-only value attempted at program
line 2.
```

routine() tries to set 6 equal to 7. It should probably be using a scoped variable (with my() or local()) instead.

Nested *?+ in regexp...



```
#!/usr/bin/perl
$string =~ /\w?*/;
RESULT /\w?*/: nested *?+ in regexp at program line 2.
```

Maybe you meant `\w*?`, which performs nongreedy matching.

Not enough arguments...



```
#!/usr/bin/perl
splice();
RESULT Not enough arguments for splice at program line 2, near
"splice()".
```

You can't always omit arguments!

Number found where operator expected...



```
#!/usr/bin/perl
if (6 7 8) { print "Gentle Giant" }
RESULT Number found where operator expected at program line 2,
near "6 7".
```

That 7 should be an operator, such as `==`.

...Permission denied



```
#!/usr/bin/perl
RESULT csh: ./program: Permission denied.
```

Did you remember to make your program executable with `chmod +x program`?

Precedence problem...



```
#!/usr/bin/perl
open FILE || die;
RESULT Precedence problem: open FILE should be open(FILE) at
program line 2.
```

This is one of the few expressions valid in Perl 4, but not in Perl 5. Use

```
open(FILE) || die
```

or

open FILE or die

Recompile Perl with -DDEBUGGING to use -D switch



```
#!/usr/bin/perl -D
```

RESULT Recompile Perl with -DEBUGGING to use -D switch.

Most perls (that is, executables such as */usr/bin/perl*) don't have -DDEBUGGING compiled. But don't worry--the -DDEBUGGING compilation option isn't the same as the symbolic debugger (which is inside every perl). See Lessons 4 through 6 in this chapter for more details.

Sort subroutine didn't return a numeric value...



```
#!/usr/bin/perl
sub sorta { "1"; }
```

```
print sort sorta ("nothing", "but", "Set");
```

RESULT Sort subroutine didn't return a numeric value at program line 3.

Every sort subroutine should return a positive integer if \$a is larger than \$b, a negative integer if \$a is smaller than \$b, and 0 for "don't care." (Of course, your definitions of "larger" and "smaller" depend on how you want to sort the array.)

substr outside of string...



```
#!/usr/bin/perl -w
$_ = 'chevron';
```

```
print substr($_, 3400);
```

RESULT substr outside of string at program line 3.

You tried to access the 3401st character of \$_ and it only had seven. You need the -w flag to see this message.

syntax error...



```
#!/usr/bin/perl
while sleep;
RESULT syntax error at program line 2, near "while sleep".
```

Error messages try to tell you on what line the error occurred. (They're not always correct, though.)



```
#!/usr/bin/perl
$var = "string"
print 7;
RESULT syntax error at program line 8, near "print".
```

The mistake isn't really near print() at all. It's on line 2, where the semicolon is missing.

Type of arg to keys must be hash...



```
#!/usr/bin/perl
@array = (1,2,3,4,5,6);
foreach (keys @array) {}
RESULT Type of arg 1 to keys must be hash (not array deref) at
program line 3, near "@array)".
```

Only hashes have keys.

Undefined sort subroutine...



```
#!/usr/bin/perl
print sort numerically (6,7);
RESULT Undefined sort subroutine "main::numerically" called at
program line 2.
```

You need to define the numerically() subroutine.

Undefined subroutine...



```
#!/usr/bin/perl
package Movie;
sub create { print "hi" }
package main;
create();
RESULT Undefined subroutine &main::create called at program
line 5.
```

This probably means that you mistyped the name of the subroutine. But in the example above, the error was raised because the subroutine was called from the wrong package. `Movie::create()` should have been used instead.

...unmatched [] in regexp...



```
#!/usr/bin/perl
$string =~ /%.[/;
```

RESULT `/%.[/:` unmatched `[]` in regexp at program line 2.

Every left bracket needs a right bracket inside a regex. If you're actually trying to match a left bracket in `$string`, you should backslash it (`\[`) because it's a special character. See the `quotemeta()` function.

Unquoted string...may clash with future reserved word



```
#!/usr/bin/perl -w
hello
```

RESULT Unquoted string "hello" may clash with future reserved word.

Perl couldn't figure out what to do with *hello*. It might be a function call, but there's no function named `hello()`. It might be a string, but if so, it's not used for anything.



```
#!/usr/bin/perl -w
$var <<END
```

```
This
is
not
a
string.
END
```

RESULT Unquoted string "is" may clash with future reserved word.

Another misleading error message: The `<<END` needs a semicolon.

Use of uninitialized value...



```
#!/usr/bin/perl -w
$var = 7;
print $vat;
```

RESULT Use of uninitialized value at program line 3.

There's a typo: \$vat should be \$var. This warning shows up only if you use the -w flag.

Use of...is deprecated...



```
#!/usr/bin/perl -w,br> $* = 1;
```

RESULT Use of \$* is deprecated at program line 2.

This means that \$* is obsolete. It'll still work, but you should avoid it because either (a) there's a better alternative in the version of Perl you're running or (b) there's a chance that it won't exist in future versions of Perl. This warning shows up only if you use the -w flag.

(For \$* in particular, you should be using the /s or /m modifiers instead.)

Useless use...



```
#!/usr/bin/perl -w
4;
```

RESULT Useless use of a constant in void context at program line 2.

The statement 4, although legal, doesn't do anything, so it can't possibly be useful.

Chapter 10, Perl 5 Interactive Course

LESSON 4

USING THE DEBUGGER

C programmers need separate utilities (such as *dbx*, *gdb*, or grander "debugging environments") to debug their programs. Perl's debugger is part of the language itself, so it's always available.

When you enter the debugger, you can step line by line through the execution of your program, check the values of certain variables, or identify when a particular subroutine is called.

You can access the debugger by adding the **-d** flag to your script. The Perl debugger "oversees" program execution.

The debugger is useful even when you have nothing to debug. It provides a Perl shell, similar to the *pesh* program from Chapter 9, Lesson 1. The **-d** flag pops you into the debugger, and *debug1* (Listing 10-5) demonstrates the use of this flag.

The -d Command-Line Flag

-d pops you into the debugger.

perl -de 0 can be used to start up an interactive debugging session. Try it!

Listing 10-5 *debug1*: Using the -d flag to invoke the debugger

```
#!/usr/bin/perl -wd
foreach (1..6) {
    print "$_\n";
}
```

debug1 launches an interactive session with the debugger. (If your text editor is Emacs, you might be able to type *M-x perl5db* to run a program inside the debugger.)

The debugger provides you with a prompt, **DB<n>**, at which you can execute commands that let you trace program execution. In the example below, we use just two debugger commands: *h* (for help) and *q* (for quit).

```
% debug1
```

RESULT Stack dump during die enabled outside of evals.

```
Loading DB routines from perl5db.pl version 1
```

```
Emacs support available.
```

```
Enter h or `h h` for help.
```

```
Sequence is: check for breakpoint, print line if necessary,
```

do action, prompt user if breakpoint or step, evaluate line.	
main::(./debug1:3):	foreach (1..6) {
DB<1> h	
T	Stack trace.
s [expr]	Single step [in expr].
n [expr]	Next, steps over subroutine calls [in expr].
<CR>	Repeat last n or s command.
r	Return from current subroutine.
c [line sub]	Continue; optionally inserts a one-time-only breakpoint at the specified position.
l min+incr	List incr+1 lines starting at min.
l min-max	List lines min through max.
l line	List single line.
l subname	List first window of lines from subroutine.
l	List next window of lines.
-	List previous window of lines.
w [line]	List window around line.
.	Return to the executed line.
f filename	Switch to viewing filename. Must be loaded.
/pattern/	Search forwards for pattern; final / is optional.
?pattern?	Search backwards for pattern; final ? is optional.
L	List all breakpoints and actions.
S [[!] pattern]	List subroutine names [not] matching pattern.
t	Toggle trace mode.
t expr	Trace through execution of expr.
b [line] [condition]	Set breakpoint; line defaults to the current execution line; condition breaks if it evaluates to true, defaults to '1'.
b subname [condition]	Set breakpoint at first line of subroutine.
b load filename	Set breakpoint on `require'ing the given file.
b postpone subname [condition]	Set breakpoint at first line of subroutine after it is compiled.
b compile subname	Stop after the subroutine is compiled.
d [line]	Delete the breakpoint for line.

D	Delete all breakpoints.
a [line] command	Set an action to be done before the line is executed.
A	Delete all actions.
V [pkg [vars]]	List some (default all) variables in package (default current). Use ~pattern and !pattern for positive and negative regexps.
X [vars]	Same as "V currentpackage [vars]".
x expr	Evals expression in array context, dumps the result.
m expr	Evals expression in array context, prints methods callable on the first element of the result.
m class	Prints methods callable via the given class.
O [opt[=val]] [opt"val"] [opt?]....	Set or query values of options. val defaults to 1. opt can be abbreviated. Several options can be listed.
recallCommand, ShellBang:	chars used to recall command or spawn shell;
pager:	program for output of " cmd";
tkRunning:	run Tk while prompting (with ReadLine);
signalLevel warnLevel dieLevel:	level of verbosity;
inhibit_exit	Allows stepping off the end of the script.
The following options affect what happens with V, X, and x commands:	
arrayDepth, hashDepth:	print only first N elements (`` for all);
compactDump, veryCompact:	change style of array and hash dump;
globPrint:	whether to print contents of globs;
DumpDBFiles:	dump arrays holding debugged files;
DumpPackages:	dump symbol tables of packages;
quote, HighBit, undefPrint:	change style of string dump;
OptionPrintRet	affects printing of return value after r command,
frame	affects printing messages on entry and exit from subroutines.
AutoTrace	affects printing messages on every possible breaking point.
maxTraceLen	gives maximal length of evals/args listed in stack trace.
ornaments	affects screen appearance of the command line.
	During startup options are initialized from \$ENV{PERLDB_OPTS} . You can put additional initialization options TTY, noTTY, ReadLine, and NonStop there (or use `R' after you set them).

< command	Define Perl command to run before each prompt.
<< command	Add to the list of Perl commands to run before each prompt.
> command	Define Perl command to run after each prompt.
>> command	Add to the list of Perl commands to run after each prompt.
{ commandline	Define debugger command to run before each prompt.
{{ commandline	Add to the list of debugger commands to run before each prompt.
! number	Redo a previous command (default previous command).
! -number	Redo number'th-to-last command.
! pattern	Redo last command that started with pattern. See 'O recallCommand' too.
!! chd	Run chd in a subprocess (reads from DB::IN, writes to DB::OUT) See 'O shellBang' too.
H -number	Display last number commands (default all).
p expr	Same as "print {DB::OUT} expr" in current package.
 dbchd	Run debugger command, piping DB::OUT to current pager.
 dbchd	Same as dbchd but DB::OUT is temporarily select()ed as well.
= [alias value]	Define a command alias, or list current aliases.
command	Execute as a perl statement in current package.
v	Show versions of loaded modules.
R	Pure-man-restart of debugger, some of debugger state and command-line options may be lost. Currently the following settings are preserved: history, breakpoints and actions, debugger Options and the following command-line options: -w, -I, -e.
h [db_command]	Get help [on a specific debugger command], enter h to page.
h h	Summary of debugger commands.
q or ^D	Quit. Set \$DB::finished to 0 to debug global destruction.

DB<1> q

The *q* quits the debugging session, returning you to your shell.

Now let's take *debug1* a bit farther, using the *s* command to step through the program instruction-by-instruction and then the *c* command to continue program execution without stepping.

```
% debug1
Stack dump during die enabled outside of evals.

Loading DB routines from perl5db.pl version 1

Emacs support available.
Enter h or `h h` for help.

main:.(./debug1:3):foreach (1..6) {
    DB<1> s

main:.(./debug1:4): print "$_\n";
    DB<1> s

1
main:.(./debug1:4): print "$_\n";
    DB<1> s

2
main:.(./debug1:4): print "$_\n";
    DB<1> s

3
main:.(./debug1:4):print "$_\n";
    DB<1> c

4
5
6
```

Listing 10-6 shows *debug2*, a program that's a bit more bugworthy.

Listing 10-6 *debug2*



```
#!/usr/bin/perl -w
sub get_nth_number {
my ($string, $n) = @_;
my $count = 0;
while (/ (\d+) /g) {
if ($count == $n) { print $&, "\n"; }
$count++;
}
```

```

}
}
print get_nth_number($ARGV[0], $ARGV[1]);

```

debug2 is meant to find the *nth* number in a string. The user supplies the string as the first command-line argument and *n* as the second. But the program doesn't work.

```
% debug2 "There 4 you are 1 2." 1
```

should print 4, but actually prints nothing at all! There aren't even any error messages, so it's time to use the debugger by adding the **-d** flag and rerunning the program. This time we'll use the *p* command to print the command-line arguments and then step through the program to see what it does.

```
% debug2 'There 4 you are 1 2' 1
```

```

RESULT Stack dump during die enabled outside of evals.
Loading DB routines from perl5db.pl version 1
Emacs support available.
Enter h or `h h` for help.
main:.(./debug2:3): print get_nth_number($ARGV[0], $ARGV[1]);
DB<1> p $ARGV[0]
There 4 you are 1 2.
DB<2> p $ARGV[1]
1
DB<3> s
main::get_nth_number(./debug2:4): my ($string, $n) = @_;
DB<3> <ENTER>
main::get_nth_number(./debug2:5): my $count = 0;
DB<3> <ENTER>
main::get_nth_number(./debug2:6): while (/ (\d+) /g) {
DB<3> l get_nth_number
3: sub get_nth_number {
4: my ($string, $n) = @_;
5: my $count = 0;
6: while (/ (\d+) /g) {
7: if ($count == $n) { print $&, "\n"; }
8: $count++;
9: }
10: }
DB<4> <ENTER>
%

```

Hmmm...the program finished without printing anything! The while condition must have evaluated to FALSE: `/ (\d+) /` must not have matched anything in `$_`...oops, there's the bug...*debug2* was testing `$_` when it should have been testing `$string`. Let's fix that and take another look, as shown in Listing 10-7.

Listing 10-7 *debug3*



```
#!/usr/bin/perl -w
sub get_nth_number {
    my ($string, $n) = @_;
    my $count = 0;
    while ($string =~ /(\d+)/g) {
        if ($count == $n) { print $&, "\n"; }
        $count++;
    }
}
print get_nth_number($ARGV[0], $ARGV[1]);
% debug3 "There 4 you are 1 2." 1
```

RESULT 1

Well, it's an improvement over *debug2*. But it's the wrong answer: The first number is 4, not 1. Let's add the **-d** flag and try the debugger again with a new command, *t*, which prints every statement.

```
% debug3 "There 4 you are 1 2." 1
...
main::(./debug3:12): print get_nth_number($ARGV[0],
$ARGV[1]);
DB<1> t
Trace = on
DB<1> c
main::get_nth_number(./debug3:4): my ($string, $n) = @_;
main::get_nth_number(./debug3:5): my $count = 0;
main::get_nth_number(./debug3:6): while ($string =~ /(\d+)/g) {
main::get_nth_number(./debug3:7): if ($count == $n) { print
$&, "\n"; }
main::get_nth_number(./debug3:8): $count++;
main::get_nth_number(./debug3:7): if ($count == $n) { print
$&, "\n"; }
main::get_nth_number(./debug3:7): if ($count == $n) { print
$&, "\n"; }
1
main::get_nth_number(./debug3:8): $count++;
%
```

The *t* (for trace) command shows you the order in which statements are executed. After toggling Trace to on, the program is "continued," displaying Perl's path through the subroutine.

Here you can see that the loop executed twice, which is odd because we wanted only the first occurrence. If you go back to the debugger and

```
p $count
```

inside the loop, you'll verify the bug: \$count starts at 0, so if \$n is 1, the second occurrence is printed, not the first. That's fixed in *debug4* (Listing 10-8).

Listing 10-8 *debug4*



```
#!/usr/bin/perl -w
sub get_nth_number {
    my ($string, $n) = @_;
    my $count = 1; # Start $count off at 1!
    while ($string =~ /(\d+)/g) {
        if ($count == $n) { print $&, "\n"; }
        $count++;
    }
}
print get_nth_number($ARGV[0], $ARGV[1]);
% debug4 "There 4 you are 1 2." 1
```

RESULT 4

Perfect! Let's make sure it works for other values.

```
% debug4 "There 4 you are 1 2." 2
```

RESULT 1

Good, good...

```
% debug4 "There 4 you are 1 2." 3
```

Uh oh. *debug4* should have printed 2, but it didn't print anything at all. Back to the debugger...

```
% debug4 "There 4 you are 1 2." 3
```

RESULT ...

```
main::(./debug4:12): print get_nth_number($ARGV[0],
$ARGV[1]);
DB<1> s
main::get_nth_number(./debug4:4): my ($string, $n) = @_;
DB<1> <ENTER>
main::get_nth_number(./debug4:5): my $count = 1;
DB<1> @array = ($string =~ /(\d+)/g)
DB<2> print join(' ', @array)
4, 1
DB<3> q
```

By evaluating `$string =~ /(\d+)/g` in a list context, you can see all the matches in \$string--and 2 isn't there. The debugger can give you the *what* but not the *why*; at this point, you have to stare at the regex until you realize that 2 doesn't match because it has a period on its right instead of a space (or better yet, a `\b`). Fix that, and you're all set.

Chapter 10, Perl 5 Interactive Course

LESSON 5

ADVANCED DEBUGGING

As you saw from the list of commands in the last lesson, there's a lot you can do with Perl's symbolic debugger. In this lesson, you'll learn about how you can make your program stop (using *breakpoints*) at certain lines or subroutines and execute actions (such as `print()` statements) automatically.

Tricky debugging techniques are best demonstrated with tricky programs. *base2base*, shown in Listing 10-9, converts numbers from one base to another. The user provides three arguments on the command line: an input base, a number in that base, and a desired output base. The program converts the number from the input base to the output base and prints the result.

Listing 10-9 *base2base*: Converting numbers from one base to another



```
#!/usr/bin/perl

sub numberize { # Convert a letter to a number, e.g., F to
# 15.
my ($char) = @_;
return $char if ($char =~ /\d/);
$char = uc($char);
ord($char) - 55;
}

sub letterize { # Convert a number to a letter, e.g., 15 to
# F.
my ($number) = @_;
return $number if ($number <= 9);
chr($number + 55);
}

sub base2decimal { # Convert the input number to base 10.
my ($ibase, $string) = @_;
my ($num, $i);
for ($i = 0; $i < length($string); $i++) {
$digit = substr($string, $i, 1);
$digit = numberize($digit);
```

```

$num += ($ibase ** (length($string) - $i - 1)) * $digit;
}
$num;
}

sub decimal2base { # Convert the base 10 number to the output
# base.
my ($outbase, $num) = @_;
my ($result);
my ($i) = int(log($num) / log($outbase));
while ($i >= 0) {
    use integer;
    $digit = $num / ($outbase ** $i);
    $result .= letterize($digit);
    $num -= ($outbase ** $i) * $digit;
    $i--;
}
$result;
}

sub base2base { # the meat of the program.
my ($ibase, $string, $outbase) = @_;
my ($num);
$num = base2decimal($ibase, $string);
decimal2base($outbase, $num);
}

print base2base($ARGV[0], $ARGV[1], $ARGV[2]);

```

Let's give *base2base* a simple run-through.

```
% base2base 5 14 2
```

```
RESULT 1001
```

The input base is 5, the number is 14, and the output base is 2. 14 in base 5 is 9 in base 10 (because $1*5^1 + 4*5^0$ is 9) and 9 in base 2 is 1001 (because $1*2^3 + 0*2^2 + 0*2^1 + 1*2^0$ is 9).

So **base2base 5 14 2** printed the correct result. But perhaps you won't be surprised if a judicious choice of inputs uncovers a bug. Try converting 255 in base 10 to hexadecimal (base 16). The answer should be *FF*.

```
% base2base 10 255 16
```

```
RESULT PP
```

Hmmm...*PP* isn't right. How do you start looking for the bug? A reasonable plan might be to check the value after each line of the main subroutine, *base2base()*. So let's add the **-d** flag and enter the debugger. This example introduces the *n* ("next") command.

```
% base2base 10 255 16
```

```
RESULT ...
```

```

main::(/base2base:48): print base2base($ARGV[0], $ARGV[1],
$ARGV[2]);
DB<1> s
main::base2base(/base2base:42): my ($ibase, $string,
$outbase) = @_;
DB<1>
main::base2base(/base2base:43): my ($num);
DB<1>
main::base2base(/base2base:44): $num = base2decimal($ibase,
$string);

```

Because you're interested in only the output of `base2decimal`, you can use the `n` debugger command to skip to the next line instead of probing deeper into the subroutine. (Note that you can't print `$num` yet--it doesn't exist, because `base2decimal()` hasn't been called just yet.)

```

DB<1> n
main::base2base(/base2base:45): decimal2base($outbase,
$num);
DB<1> print $num
255

```

`base2decimal(10, 255)` is 255, as it should be. Skipping over `decimal2base()` takes you out of the program because it's the last statement executed.

```

DB<2> n
PP

```

The problem is in the `decimal2base()` subroutine. Run the debugger again.

```
% base2base 10 255 16
```

RESULT ...

```

main::(/base2base:48): print base2base($ARGV[0], $ARGV[1],
$ARGV[2]);
DB<1> s

```

Now you're stepping through the execution line by line. The `r` debugger command returns from the subroutine.

```

main::base2base(/base2base:42): my ($ibase, $string,
$outbase) = @_;
DB<1>
main::base2base(/base2base:43): my ($num);
DB<1>
main::base2base(/base2base:44): $num = base2decimal($ibase,
$string);
DB<1>
main::base2decimal(/base2base:17):
17: my ($ibase, $string) = @_;

```

Because we think that the bug is somewhere in `decimal2base()`, we can return immediately with `r`.

```
DB<1> r
```



```

main::base2base(./base2base:45):decimal2base($outbase, $num);
DB<1>
main::decimal2base(./base2base:28):
28: my ($outbase, $num) = @_;
DB<1>
main::decimal2base(./base2base:29):
29: my ($result);
DB<1>
main::decimal2base(./base2base:30):
30: my ($i) = int(log($num) / log($outbase));
DB<1>
main::decimal2base(./base2base:31):
31: while ($i >= 0) {
DB<1>
main::decimal2base(./base2base:33):
33: $digit = $num / ($outbase ** $i);
DB<1>
main::decimal2base(./base2base:34):
34: $result .= letterize($digit);
DB<1> print $digit
15

```

Looks good so far: That's the correct value. So what's \$result? You can't print it yet because the line `$result .= letterize($digit)` hasn't been executed just yet. The next s takes you into that subroutine.

```

main::letterize(./base2base:11):my ($number) = @_;
DB<1>
main::letterize(./base2base:12):return $number if ($number <=
9);
DB<1>
main::letterize(./base2base:13):chr($number + 65);
DB<2> print chr($number + 65)
P

```

There's the problem. But was chr() fed the right value?

```

DB<3> print $number
15

```

It was. So the 65 must be wrong. (It should be 55.)

```

DB<4> q

```

At this point, you can ascertain that numberize() should be changed too because it also uses 65. Pretend you don't know that.

If you switch the input and output from base2base 10 255 16 (which should produce *FF*) to **base2base 16 FF 10** (which should produce 255), you can uncover the numberize() bug.

```

% base2base 16 FF 10

```

```

RESULT 85

```

Now use the debugger again, using a breakpoint to skip directly to `numberize()`. You can think of a breakpoint as a bookmark in the debugger. When the debugger executes your program, it stops whenever it sees the breakpoint.

There are three breakpoint commands.

Breakpoint Debugger Commands

b *SUBROUTINE CONDITION* creates a breakpoint at the first line of *SUBROUTINE*.

b *LINE CONDITION* creates a breakpoint. If *LINE* is omitted, the current execution line is used. The breakpoint triggers on *CONDITION*, which is true by default.

d *LINE* deletes the breakpoint at *LINE*.

D deletes all breakpoints.

Now execute *base2base* with the same arguments, setting a breakpoint at the `numberize()` subroutine.

```
% base2base 16 FF 10
```

```
RESULT ...
```

```
main::(./base2base.1:48): print base2base($ARGV[0], $ARGV[1],  
$ARGV[2]);  
DB<1> b numberize
```

This creates a breakpoint at `numberize()`. If the program is continued at this point, it stops automatically when it reaches that subroutine.

```
DB<2> c  
main::numberize(./base2base.1:4): my ($char) = @_;  
DB<2> s  
main::numberize(./base2base:5): return $char if ($char =~  
/\d/);  
DB<2> p $char  
F  
DB<3> s  
main::numberize(./base2base:6): $char = uc($char);  
DB<3>  
main::numberize(./base2base:7): ord($char) - 65;  
DB<3> p $char  
F  
DB<4> p ord($char) - 65  
5  
DB<5> q
```

You've uncovered the same problem in `numberize()` as `letterize()`. But don't fix it yet. The above debugging session assumes that your reasoning was as follows: "There was a bug in `letterize()`, so a twin bug probably infests `numberize()`."

But suppose you thought, "There was a bug in the input to `chr()`, so there's probably a twin bug in the input to `ord()`, the opposite of `chr()`." Inside the debugger, you'd want to be able to search for occurrences of `ord()` in the program and then trigger an action when that point is reached.

The following debugging commands help:

```
/pattern/ Search forwards for pattern; final / is optional.
?pattern? Search backwards for pattern; final ? is optional.
a [line] command
Set an action to be done before the line is executed.
Sequence is: check for breakpoint, print line if necessary,
do action, prompt user if breakpoint or step, evaluate line.
A Delete all actions.
% base2base 16 FF 10
```

RESULT ...

```
main::(./base2base.1:48): print base2base($ARGV[0], $ARGV[1],
$ARGV[2]);
```

The opposite of `chr()` is `ord()`. Because you're looking for essentially the same bug as before, you can search for the character string *ord* in your code by enclosing it in slashes: */ord/*. Slashes search forward; question marks (e.g., *?ord?*) search backward.

```
DB<1> /ord/
7: ord($char) - 65;
```

ord occurs on line 7: `ord($char) - 65`

```
DB<2> a 7 print $char; print ' ' ; print ord($char) - 65;
print "\n"
```

The *a* debugging command creates an action to be executed before the specified line (in this case, line 7). Every time line 7 is executed, the four *print* statements are triggered. Now continue the program with the *c* command.

```
DB<3> c
F 5
F 5
85
```

There's the bug: `$char` is `F`, as it should be, but it's being numberized to 5, when it should be numberized to 15 instead. As before, the 65 needs to be changed to 55.

You can also define commands that are executed before (or after) every debugger prompt. The *blueleak* program in Listing 10-10 has no bugs and does nothing useful, but illustrates the `<` and `>` debugger commands.

```
< command Define command before prompt.
> command Define command after prompt.
```

Listing 10-10 *blueleak*

```
#!/usr/bin/perl -wd
$string = `porphyria`;
$string =~ s/[aeiou]/ /g;
$string =~ tr/a-z/b-za/;
$string =~ tr/a-z/b-za/;
```

```
$string =~ tr/a-z/b-za/;
print $string, "\n";
```

Now the < command can be used to print the value of \$string after each statement.

```
% blueleak
```

```
RESULT ...
```

```
main::(/blueleak:3): $string = "porphyria";
DB<1> < print "$string\n";
DB<2> s
main::(/blueleak:5): $string =~ s/[aeiou]/ /g;
porphyria
DB<2>
main::(/blueleak:7): $string =~ tr/a-z/b-za/;
p rphyr
DB<2>
main::(/blueleak:8): $string =~ tr/a-z/b-za/;
q sqizs
DB<2>
main::(/blueleak:9): $string =~ tr/a-z/b-za/;
r trjat
DB<2>
main::(/blueleak:11): print "$string\n";
DB<2>
s uskbu
```

You've now seen enough debugger commands to help you detect just about any bug you might find in your programs. But as you've seen from the h debugger command, there are quite a few more, which you should explore at your convenience.

The guts of the debugger is stored in the library file *perl5db.pl*. If you don't like it, make your own: Copy *LIB/perl5db.pl* to another file (say, *LIB/mydebug.pl*) and change that. (Detailed instructions can be found in the *perldebug* documentation. You'll also find instructions for customizing the existing debugger by creating a *.perl5db* file in your home directory.)

Chapter 10, Perl 5 Interactive Course

LESSON 6

-DDEBUGGING

You now have several weapons in your debugging arsenal, from *print* statements to the **-w** flag and **use strict** to the debugger itself. There's one final step you can take, but it's a big one: You can recompile Perl (in other words, create a brand new */usr/bin/perl*) with the **-DDEBUGGING** compilation option. This creates a Perl that's a bit more self-aware: It can track the execution of your programs a bit better, for instance. (If you recompile Perl on a computer used by other people, consider naming the new Perl something different, such as */usr/bin/perl.d*.)

It's possible, but unlikely, that the Perl installed on your system is already compiled with **-DDEBUGGING**. Here's how you can tell:

```
perl -D -e 'print "Hello world.";'
```

If you see this error message:

Recompile perl with -DDEBUGGING to use -D switch

then this lesson won't be of much use to you. But there's a consolation prize: Your programs will run faster. By default, the **-DDEBUGGING** flag isn't used when Perl is configured because it makes Perl pay closer attention to the programs as it executes them. That scrutiny makes it easier for you to figure out precisely how your programs are being interpreted, but it also makes them run slower.

If your Perl has **-DDEBUGGING** built in, you can do any of the things listed in Table 10-1. These options are explained in the *perlrun* documentation.

Table 10-1 The **-DDEBUGGING** options, from *perlrun*

Character	Numeric Equivalent	Function
p	1	Tokenizing and parsing
s	2	Stack snapshots
l	4	Label stack processing
t	8	Trace execution
o	16	Operator node construction
c	32	String/number conversions
P	64	Print preprocessor command for -P
m	128	Memory allocation
f	256	Format processing

r	512	regex parsing
x	1024	Syntax tree dump
u	2048	Tainting checks
L	4096	Memory leaks (no longer supported)
H	8192	Hash dump (usurps values())
X	16384	Scratch pad allocation
D	32768	Cleaning up

Explaining all these flags is beyond the scope of this book, but some of the more useful flags (*s*, *t*, *r*, and *x*) are demonstrated in this lesson.

If you specify the -DDEBUGGING flags by letter, you can concatenate them: -DsIt shows stack snapshots and label processing and traces the execution of your program. If you specify the flags as numbers, you can add them: -D14 and -DsIt behave the same.

In addition to helping you debug your programs, the -DDEBUGGING flags are useful for *optimizing* (making your program run faster) because you can see exactly how Perl executes the code. For instance, you can use the "stack snapshot" option to display the operands on the stack (an internal data structure used by Perl to store temporary values).

```
% perl -Ds -e 'print 3 + 4 * 5'
```

```
RESULT =>
```

```
=> IV(4)
=> IV(4) IV(5)
=>
=> IV(3)
=> IV(3) IV(20)
EXECUTING...
=>
=>
=>
=> *
=> * IV(23)
=> SV_YES
23
```

That stack snapshot shows you that $3 + 4 * 5$ is interpreted as $3 + (4 * 5)$. You could also have determined that with the *t* option, which traces execution.

```
% perl -Dt -e 'print 3 + 4 * 5'
```

```
RESULT (-e:1) constant item(IV(4))
(-e:1) constant item(IV(5))
(-e:1) multiplication
(-e:1) constant item(IV(3))
(-e:1) constant item(IV(20))
```

```
(-e:1) addition
EXECUTING...
(-e:1) block entry
(-e:1) next statement
(-e:1) pushmark
(-e:1) constant item(IV(23))
(-e:1) print
(-e:1) block exit
23
```

You can see how a regex is compiled with the *r* option.

```
% perl -Dr -e '"aaabbb" =~ /^a.+b$/'
```

RESULT first 14 next 97 offset 4

```
rarest char b at 0
1:BRANCH(33)
5:BOL(9)
9:EXACTLY(15) <a>
15:PLUS(23)
19:ANY(0)
23:EXACTLY(29) <b>
29:EOL(33)
33:END(0)
start `a` anchored must have "b" back -1 minlen 3
EXECUTING...
1:BRANCH <aaabbb>
5:BOL <aaabbb>
9:EXACTLY <aaabbb>
15:PLUS <aabbb>
23:EXACTLY <b>
29:EOL <>
33:END <>
```

The first chunk (everything before "EXECUTING...") shows you how Perl parses the regex. The second chunk shows you the result of matching *aaabbb* against that regex.

You can also print the syntax tree of your program, which shows (in graphic detail) how Perl distinguishes expressions, statements, subroutine calls, and so on, as shown in *syntree* (Listing 10-11).

Listing 10-11 *syntree*: Using -Dx to print the syntax tree

```
#!/usr/bin/perl -wDx
print "Hello, world! \n";
```

syntree just prints *Hello, world!*, but here's what happens under the hood:

```
% syntree
```

RESULT {
6 TYPE = block exit ==> DONE

```

FLAGS = (SCALAR,KIDS,PARENS)
{
1 TYPE = block entry ==> 2
}
{
2 TYPE = next statement ==> 3
FLAGS = (SCALAR)
LINE = 3
}
{
5 TYPE = print ==> 6
FLAGS = (SCALAR,KIDS)
{
3 TYPE = pushmark ==> 4
FLAGS = (SCALAR)
}
{
4 TYPE = constant item ==> 5
FLAGS = (SCALAR)
SV = PV("Hello, world!")
}
}
}
SUB DynaLoader::boot_DynaLoader = (xsub 0x2009e4f0 0)
EXECUTING...
Hello, world!

```

Should you ever want your program to know which -DDEBUGGING flags it was called with, you can access the numeric value through `$_D`, demonstrated in Listing 10-12.

The `$_D` Special Variable

`$_D` (or `$DEBUGGING`) contains the numeric value of the debugging flags.

Listing 10-12 debugnum: Using `$_D` to access the -DDEBUGGING flags

```

#!/usr/bin/perl -DLur
print "The numeric value of the debugging flags is $_D.\n";

```

The debugging flags *Lur* are equivalent to $(4096 + 2048 + 512)$, which is 6656. So that's what you'll see in `$_D`.

```
% debugnum
```

RESULT EXECUTING...

```
The numeric value of the debugging flags is 6656.
```

Chapter 10, Perl 5 Interactive Course

LESSON 7

ERROR HANDLING

Bulletproof programs don't wait for Perl to find their errors for them. They generate and resolve their own errors. A subroutine deep inside your program can signal a subroutine "closer to the surface," which can decide for itself what to do about the error. Depending on the circumstances, it might decide to warn the user, restart some loop, or kill the program entirely.

caller()

Suppose an error occurs on line 196, inside a subroutine that's called several different times in your program. Knowing that line number is useful, but it's not always as helpful as knowing where that subroutine was called from. That's what `caller()` tells you.

The `caller()` Function

`caller(NUMBER)` returns the environment of the current subroutine call.

If *NUMBER* is supplied, `caller()` returns some extra information that the debugger uses to print a stack trace. The value of *NUMBER* indicates how many levels of subroutines to "pop."

In a scalar context, `caller()` simply returns `TRUE` if invoked inside a subroutine and `FALSE` otherwise.

In an array context, `caller()` returns a three-element array: (*\$package*, *\$filename*, *\$line*), indicating that the current line of code was called from line number *\$line* in the file *\$filename* in the package *\$package*.

Listing 10-13, *mooview*, contains one subroutine, `view()`, that doesn't do much other than display how it was called.

Listing 10-13 *mooview*: Using `caller()` to identify from where the current subroutine was called

```
#!/usr/bin/perl -wl
package Movie;
sub view {
    if (caller) { print "I'm in a subroutine." } # scalar context
    ($package, $filename, $line) = caller; # list context
    print " This subroutine was called from line $line";
    print " in the package $package, in file $filename.";
    view() if $package eq `main`; # Recurse if called from main
}
```

```

}
package main;
Movie::view();

```

When the program begins, it calls `view()` with the statement `Movie::view()` (at line 15 and in the *main* package). Then, after displaying some information about its caller, `view()` calls itself again. Only this time, that call comes from within the *Movie* package, at line 11.

```
% mooview
```

```
RESULT I'm in a subroutine. # first call to view()
```

```

This subroutine was called from line 15
in the package main, in file ./mooview.
I'm in a subroutine. # second call to view()
This subroutine was called from line 10
in the package Movie, in file ./mooview.

```

By setting *NUMBER* equal to 0, a subroutine can access information about itself, as shown in Listing 10-14.

Listing 10-14 *mooview2*: Using `caller()` to access parent subroutine names

```

#!/usr/bin/perl -wl
sub view {
    print "This subroutine is ", (caller(0))[3];
    print "which was called from ", (caller(1))[3];
    print "which was called from ", (caller(2))[3];
}
sub two {
    view();
}
sub moo {
    two();
}
moo();

```

Inside the `view()` subroutine, `caller(0)` refers to the current subroutine (i.e., `main::view()`), `caller(1)` refers to the "parent" subroutine, and `caller(2)` refers to the "grandparent" subroutine.

```
% mooview2
```

```
RESULT This subroutine is main::view
which was called from main::two
which was called from main::moo
```

Exceptions

An exception is an error message generated by the programmer instead of Perl, designed to be caught by another part of the program. The *exceptions.pl* library file, written by Tom Christiansen, lets you use `throw()` to raise exceptions and `catch()` to catch them.

The `throw()` and `catch()` Functions

`throw(STRING)` raises an exception containing the *STRING*.

`catch(CODE, LIST)` evaluates *CODE* and returns an error *STRING* if any regex in *LIST* matches the exception raised by **`throw()`**.

Exceptions provide a way to signal an error without using the subroutine's return value. If an exception isn't caught, the program `die()`s.

The *lob* program, shown in Listing 10-15, contains an `input()` subroutine that can raise any of three exceptions: *zero*, *negative*, and *complex*. Calls to `input()` are invoked inside a `catch()`, which traps the *zero* and *negative* exceptions.

Listing 10-15 *lob*: Using `catch()` and `throw()` to send and receive exceptions

```
#!/usr/bin/perl -l
require 'exceptions.pl' ;
sub input {
  chomp($answer = <>);
  throw('zero') if $answer =~ /^0$/;
  throw('negative') if $answer =~ /^-/;
  throw('complex') if $answer =~ /i/;
  $answer;
}
$i = 1;
while (1) {
  if ($error = catch('$return = &input()', 'negative', 'zero'))
  {
    print "Hey! That number was $error!\n";
  } else {
    print "OK. That number was $return.\n";
  }
}
```

The `input()` subroutine `throw()`s an exception if the user types a solitary 0, anything beginning with a hyphen, or anything containing an *i*. Otherwise, the user's string is returned and `$return` is set as usual.

Only two of the three possible exceptions are caught: *negative* and *zero*; the *complex* exception therefore terminates the program.

% lob

RESULT 8

```
OK. That number was 8.  
6.022e+23  
OK. That number was 6.022e+23.  
0  
Hey! That number was zero!  
4.669201  
OK. That number was 4.669201.  
-18  
Hey! That number was negative!  
8+2i  
EXCEPTION: complex  
%
```

(These are *program* exceptions specific to Perl. UNIX lets processes create exceptions as well. That's what the "exception bits" in the `select()` call are for.)

Carp

The Carp module (runner-up for "best name in Perl") lets one package complain to another. It's especially useful inside modules.

There are three different levels of complaining, all of which use `caller()` to identify the line from which the subroutine was called.

- **`carp()` is similar to `warn()`, but provides the `caller()`'s line number instead.**
- **`croak()` is similar to `die()`, but provides the `caller()`'s line number instead.**
- **`confess()` is similar to `die()`, but provides both the `caller()`'s line number and the actual line number on which the error occurred.**

`croak()` and `confess()` both provide backtraces, so you can see the chain of subroutines that resulted in the complaint. `croak()` shows only subroutines belonging to the current package; `confess()` shows all the subroutines, regardless of their package, as shown in Listing 10-16.

Listing 10-16 *complain*: Using `carp()` to warn, and `croak()` and `confess()` to die

```
#!/usr/bin/perl -w  
package Complain;  
use Carp;  
sub input {  
    chomp($answer = <>);  
    warn " warning: zero" if $answer =~ /0/;  
    carp " carping: zero" if $answer =~ /0/;  
    croak " croaking: negative number" if $answer =~ /^-/;
```

```
confess " confessing: fractional number" if $answer =~ m!/!;  
}  
package main;  
while (1) {  
  Complain::input; # this is line 17  
}
```

Here are four inputs to *complain*: 0 makes it warn() and carp(), -18 makes it croak(), and 1/4 makes it confess():

```
% complain
```

```
RESULT 56
```

```
0  
warning: zero at ./complain line 8, <> line 2.  
carping: zero at ./complain line 17  
12  
-18  
croaking: negative number at ./complain line 17  
% complain  
1/4  
confessing: fractional number at ./complain line 11  
Complain::input called at ./complain line 17
```

Chapter 10, Perl 5 Interactive Course

LESSON 8

EFFICIENCY

So far, this chapter has discussed techniques for diagnosing and correcting errors. But that's only half the battle: A program that produces the right output might take an unacceptably long time to execute, or might hog all your computer's memory, grinding it to a halt. In this lesson, you'll learn some ways to

- **Speed up your code (minimizing time)**
- **Make the compiled program smaller (minimizing space)**

Sometimes you have to trade off one for the other, sacrificing execution speed for a smaller executable or letting the program grow in size to make it run faster.

Parts of the Perl language itself embody this compromise. Consider the library files bundled with Perl--at one extreme, they could be eliminated and placed in the `/usr/bin/perl` executable. Then you wouldn't need `require()` or `use()` to read them off your disk and into your program. But that has a hefty price: The Perl binary would be much bigger and your processes would soak up oodles more memory. At the other extreme, some of Perl's built-in functions could be relegated to library files, resulting in a smaller `/usr/bin/perl`, but slower execution whenever you wanted to use those functions.

You can make similar trade-offs in your programs. The two sections that follow list some techniques for minimizing time and space. Some are compromises between the two; others are simply optimizations.

Minimizing Time

Several rules can make your program run faster. If you're in doubt about whether some modification will speed up or slow down your code, try it both ways and use the `Benchmark` module (described in Chapter 9) to see which runs faster.

Easy Choices

- **Several simple statements are usually better than a single complex statement.**
- **Pre-extend your arrays with `$#array` and your strings with `substr()`.**

- Avoid conditionals when you can get by with `&&`, `||`, *and*, and *or*. Example:

```
if ($var == 7) { print "hi" }
```

isn't as fast as

```
$var == 7 && print "hi"
```

- Avoid regex alternators when you can get by with `||`. Example:

```
/\w|\s/
```

is slower than

```
/\w/ || /\s/
```

- Use arrays instead of hashes (unless your program often searches for a particular key).
- Use hashes instead of arrays (if your program often searches for a particular key).
- Use `push()` and `pop()` where possible. `unshift()` is slower than `push()`.
- Avoid `$&`, `$'`, and `$``. If Perl sees them in your program, it will store the results of all regex matches explicitly.
- Use the *integer* pragma to avoid floating-point arithmetic.
- When printing strings to be concatenated, use commas instead of concatenation operators:

```
print $greeting, " world"
```

is faster than

```
print $greeting . " world"
```
- Use `tr///` instead of `s///` when possible.
- `study()` a large string if you're going to be performing many pattern matches on it.
- Avoid `getc()`.

Hard Choices

- Use the Benchmark module to determine which programming constructs are the fastest.
- Don't get your data from files: Store it after the `__DATA__` or `__END__` token in your program. Better yet, place it directly into whatever data structure you need.

- If you must read data from a file and need all of it in memory at once, slurp it directly into an array (`@array = <>`) because it's faster to access memory than disk. (But if your file is larger than available memory, things will slow down considerably when your process starts "swapping out" to disk.)
- Rather than making your program compute a nasty function, consider creating a lookup table containing precomputed values. For instance, if your program needs to convert degrees to radians thousands of times, and only for integral degree values, you can precompute a 360-element array:


```
$deg_to_rad[0] = 0;
$deg_to_rad[1] = 0.0174532925199433;
$deg_to_rad[2] = 0.0349065850398866;
$deg_to_rad[3] = 0.0523598775598299;
```
- and so on. Then, if you want to convert 34° to radians, you can simply access `$deg_to_rad[34]` instead of computing `34 * (4 * atan2(1,1)) / 180`.

Minimizing Space

Sometimes it's difficult to identify how much memory your program uses. Utilities such as *ps* or *top* might tell you, but they can be misleading because most operating systems don't immediately reclaim memory freed by your program.

Easy Choices

- Declare variables with `my()`. Use the smallest scope possible, minimizing the number of package-wide variables. Use `undef()` when possible.
- Don't use `1..100000` to loop from 1 to 100000; that constructs a 100,000-element array. Use a loop variable instead: `for ($i = 1; $i <= 100000; $i++)`.
- Keep as much data on disk as possible, rather than storing it in variables. (The more variables you have, the larger your process.)
- `vec()` your bitstrings.
- Avoid `tr///`, which stores a translation table containing 256 short integers.
- Use `each()` instead of `keys()`.
- Use a database manager (such as SDBM or Berkeley DB) to store large hashes. (Database managers are discussed in Chapter 11.)
- Use arrays instead of hashes.
- `pack()` your data.

Hard Choices

- **Avoid recursion when you can.**
- **Use numbers instead of strings; they take fewer bytes to store.**

Profiling Your Code

A profiler tells you how much time your program spends in each of its subroutines and how often it calls each one. There are a few Perl profilers available; consider using Dean Roehrich's `Devel::DProf` module, available from the CPAN.

Review

Patricia: My Perl doesn't have doesn't have `-DDEBUGGING`, so I skipped Lesson 6. How about you?

Al: My Perl didn't either, so I grabbed a new Perl distribution from the CPAN and complied my own version with `-DDEBUGGING` turned on, so I could play around. That "trace execution" flag is pretty nifty.

Patricia: I'm surprised you have enough disk space to compile Perl. It's pretty big-5 megabytes.

Al: Well, it wouldn't fit inside my home directory, so I stored parts of it on a different filesystem. That made the compilation take a little bit longer, I suppose, because the compiler had to retrieve files over the network.

Patricia: So you saved some disk space, at the cost of a little extra compilation time. You traded time for space, just like the compromises discussed in Lesson 8.

Chapter 11, Perl 5 Interactive Course



LOCK AND KEY: SECURITY AND DATABASES

Computer security comes in several different forms. *Physical* security means that intruders can't get at your data because there's a concrete wall or an armed guard in their way. *Network* security means that an unauthorized user on the Internet (or a local network) can't access your data. *Access control* is used to withhold a file on a computer from another user on that same computer. *Encryption* is used to make sure that even if someone manages to get the file, he or she won't be able to read it.

UNIX has some built-in mechanisms for access control and encryption; Perl lets you use them to help keep your data away from prying eyes. Perl also makes it easy to construct your own encryption programs. In the first three lessons of this chapter, you'll learn about Perl functions relating to UNIX users and groups,

security issues with UNIX setuid scripts, and methods for data encryption.

If your goal is to enhance (rather than hinder) access to your data, you can use a *database engine* that structures your data and stores it on disk. Database engines let you access data in multiple ways: by row, by column, by page, by type, or by key. The Perl 4 interfaces to these database engines have to be built into the actual Perl executable; that's not necessary in Perl 5, which lets you tie variables (such as a database stored in a hash) to packages that can store and fetch variables from a database engine. Lessons 5 through 8 describe means of interacting with database engines through Perl.

LESSON 1

UNIX USERS AND GROUPS

Every file and every process in UNIX has an owner. That owner might correspond to an actual user or it might be one of the system accounts, such as *root* or *daemon* or *nobody*. Information about these accounts is stored in a database (often called the *passwd* file) somewhere on the local network; each entry contains information about an account, such as its name, encrypted password, home directory, and *group*, which is a collection of users that have joint ownership.

Many UNIX systems store account information in a text file called */etc/passwd*. Other UNIX systems use different mechanisms, such as NIS/Yellow Pages or NetInfo. Never fear: Perl provides you with a uniform interface for account information, so your programs will behave properly no matter what mechanism is used.

To demonstrate what these functions do, let's assume that account information is stored in */etc/passwd*:

```
% cat /etc/passwd
```

```
RESULT root:bvrvcapnuxmn:0:1:system PRIVILEGED Account,,,:/bin/csh
nobody:*:65534:65534:anonymous NFS user:/:
daemon:*:1:1:system background account:/:
sys:*:2:2:::/bin/csh
bin:*:3:3::/bin:
uucp:*:4:8::/var/spool/uucppublic
ingres:*:7:7::/usr/ingres/bin/csh
perluser:wwFl2U703IYY2:1014:20:Perry
Luser,Rm.409,555-9112:/users/perluser:/bin/csh
joebob:mxSo.3KRNTkpE:1015:20:Joe Bob:/users/joebob:/bin/tcsh
deadcaml:TyLtKlDivDOms:1016:21:Dolly
Lama:/users/deadcaml:/bin/bash
```

What does all this mean? Take a look at the eighth line, which is the password file entry for the *perluser* account. This line consists of seven fields, separated by colons:

1. The user's username: *perluser*
2. The user's encrypted password: *wwFl2U703IYY2*
3. The user's "user ID" (UID): *1014*
4. The user's default "group ID" (GID): *20*
5. The user's name, office, and telephone number: *Perry Luser,Rm.409,555-9112*
6. The user's home directory: */users/perluser*
7. The user's shell: */bin/csh*

Every user has one (and exactly one) user ID (UID), just as every process has one, and exactly one,

process ID (PID).

No two users on a computer can have the same UID. But multiple users can share a *group ID* or GID. And each user can belong to more than one group (on most systems).

So what's a group? You learned in Chapter 3 that the `chmod()` function lets you specify three classes of file permissions, as shown in Figure 11-1.

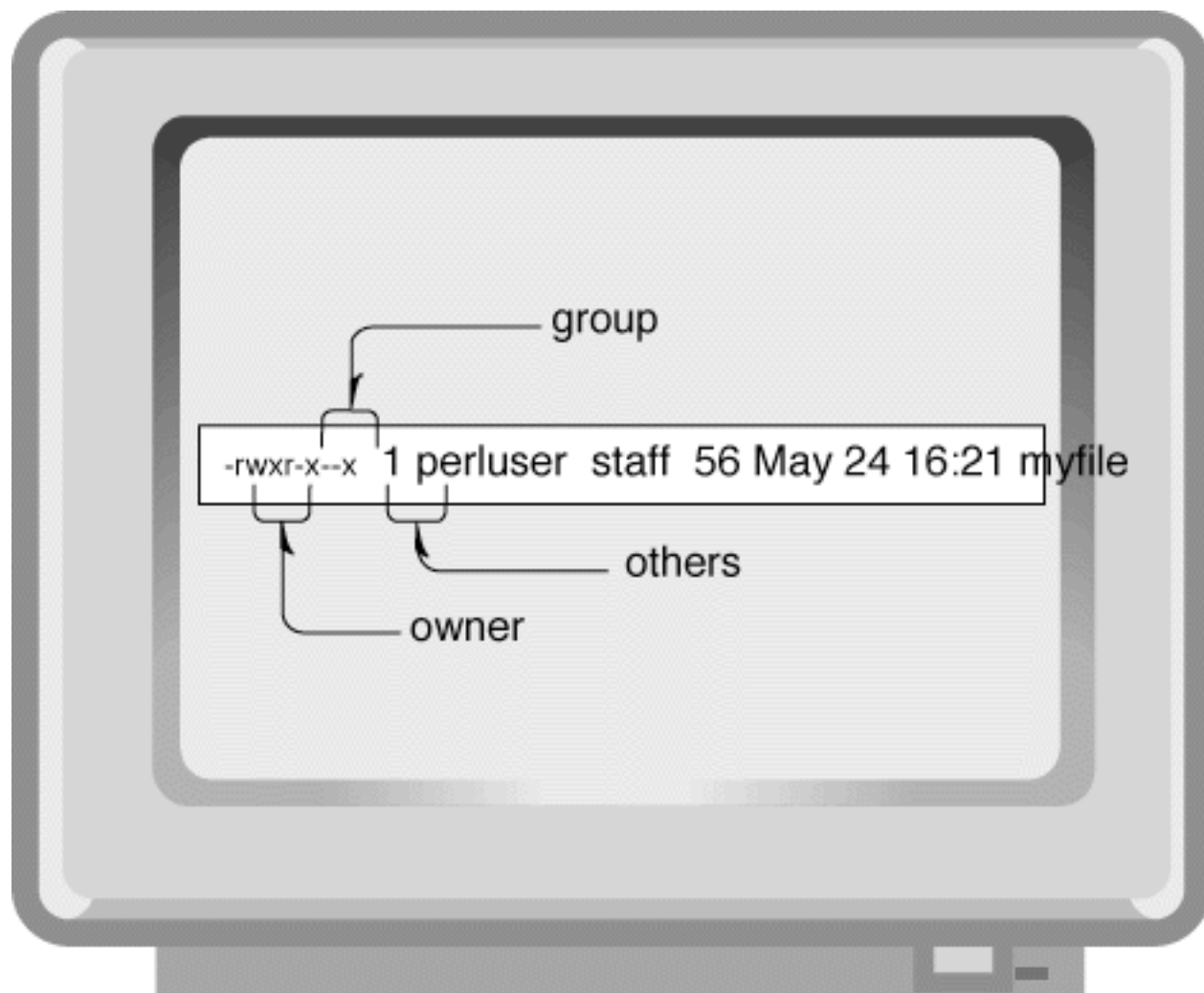


Figure 11-1

File permissions

Groups lie halfway between a "single user" and "everyone": Members of the group with GID 20 can read and execute the file, whereas others can only execute the file and the owner can read, write, and execute the file. (Some improvements have been made to the UNIX group system, notably by Transarc's AFS filesystem, which lets users create more flexible access control lists.)

Every computer maintains a list of the valid groups and their members, most often in an `/etc/group` file. Here's part of one:

```
system:*:0:root
```

```
daemon:*:1:daemon
uucp:*:2:uucp
tty:*:7:root
sysadmin:*:16:
ingres:*:74:ingres
staff:*:20:perluser,joebob,deadcaml
managers:*:21:deadcaml
teambblue:*:22:perluser,joebob
```

You can find out which groups you belong to with either the `groups` or `id` UNIX commands.

Before you encounter Perl's functions for extracting user information from the *passwd* and *group* databases, let's see an easy (but often inaccurate) way to determine who's running your program: `getlogin()`.

The `getlogin()` Function

`getlogin()` returns the current login from the */etc/utmp* file, if your computer has one.

`getlogin()` returns `FALSE` if there's no */etc/utmp* file or if the user isn't mentioned in it, as shown in Listing 11-1.

Listing 11-1 *whobeme*: Using `getlogin()` to print the user's name

```
#!/usr/bin/perl -w
print getlogin;
```

Here's what happens if *perluser* runs *whobeme*:

```
% whobeme

perluser
```

The UNIX *su* (substitute user) command, which lets you become another user, won't affect the return value of `getlogin()`.

```
% su - root

Password: <enters root password>

# whobeme

perluser
```

even though it might affect `$ENV{USER}`:

```
# perl -e'print $ENV{USER}'

root
```

The `getpwnam()` and `getgrnam()` functions provide a convenient interface to the password and group files. They mean "Get Password by Name" and "Get Group by Name"; you'll see corresponding functions for "Get Password by UID" and "Get Group by GID" soon afterward.

The `getpwnam()` and `getgrnam()` Functions

`getpwnam(`*USER*`)` returns the password entry for the user named *USER*.

`getgrnam(`*GROUP*`)` returns the group entry for the group named *GROUP*.

In a scalar context, `getpwnam()` returns the UID and `getgrnam()` returns the GID.

In a list context, `getpwnam()` returns a nine-element array (`$username`, `$encrypted_password`, `$UID`, `$GID`, `$disk_quota`, `$comment`, `$information`, `$home_directory`, `$shell`)--all the elements for a single password entry.

In a list context, `getgrnam()` returns a four-element array (`$group_name`, `$encrypted_password`, `$GID`, `$group_members`). `$group_members` is a space-separated list of the members in the group.

The *biograph* program (see Listing 11-2) invokes `getpwnam(getlogin)`, which returns the password fields for the user running the program.

Listing 11-2 *biograph*: Using `getpwnam()` to retrieve information about the user

```
#!/usr/bin/perl

($username, $encrypted_password, $UID, $GID, $disk_quota,
$comment, $information,
$home_directory, $shell) = getpwnam(getlogin);

print <<END;

$username, whose encrypted password is $encrypted_password,
has user id $UID
and default group $GID.

${username}'s home directory is $home_directory (quota
$disk_quota),
and ${username}'s login shell is $shell.

In real life, $username is $information.

END

% biograph
RESULT perluser, whose encrypted password is akb4skhus87mt, has user id
1014
and default group 20.
```

The user's home directory is `/users/perluser` (quota 0), and the user's login shell is `/bin/csh`.

In real life, `perluser` is **Perry Luser**, Rm.409, 555-9112.

As promised, here are the nearly identical `getpwuid()` and `getgrgid()` functions, which behave the same as `getpwnam()` and `getgrnam()` except they expect an ID instead of a name.

The `getpwuid()` and `getgrgid()` Functions

`getpwuid(UID)` returns the password entry for the user with user ID *UID*.

`getgrgid(GID)` returns the group entry for the group with group ID *GID*.

`getpwuid()` and `getgrgid()` return the same arrays as `getpwnam()` and `getgrnam()`.

In a scalar context, `getpwuid()` returns the username and `getgrgid()` returns the group name.

```
print getpwuid(0)
```

prints the password fields for *root*, because `print()` forces a list context and *root* has UID 0. In a scalar context, `getpwuid(0)` yields the name of the account: "root."

```
(getpwuid($<))[0]
```

yields the user's username. `$<` is covered in the next lesson, but you should use this instead of `getlogin()`.

```
print (getgrgid(20))[3]
```

prints the names of the people in group 20: *perluser joebob deadcaml*.

In addition to accessing individual entries, you can also iterate through the password and group databases with `getpwent()` and `getgrent()`.

The `getpwent()` and `getgrent()` Functions

`getpwent()` iterates through the password database.

`getgrent()` iterates through the group database.

`getpwent()` and `getgrent()` return the same arrays as `getpwnam()` and `getgrnam()`.

In a scalar context, `getpwent()` returns the username and `getgrent()` returns the group name.

Not all computers store the password and group databases in human-readable text files. On those systems, it's convenient to use `getpwent()` and `getgrent()` for tasks that require gathering some snippet of data from every user or group. The *roster* program (Listing 11-3) iterates through each database.

Listing 11-3 *roster*: Using `getpwent()` and `getgrent()` to grab all password and group entries

```
#!/usr/bin/perl -w1

while (@passwd_fields = getpwent) {
    print "User $passwd_fields[0] has UID
    $passwd_fields[2]";
```



```

}

print ``;

while (@group_fields = getgrent) {
    print "Group $group_fields[0] has GID
    $group_fields[2]";
}

% roster

```

RESULT User root has uid 0

```

User nobody has UID 65534
User daemon has UID 1
User sys has UID 2
User bin has UID 3
User uucp has UID 4
User ingres has UID 7
User perluser has UID 1014
User joejob has UID 1015
User deadcaml has UID 1016

```

```

Group system has GID 0
Group daemon has GID 1
Group uucp has GID 2
Group tty has GID 7
Group sysadmin has GID 16
Group ingres has GID 74
Group staff has GID 20
Group managers has GID 21

```

The setpwent(), setgrent(), endpwent(), and endgrent() Functions

setpwent() rewinds **getpwent()** to the beginning of the password database.

setgrent() rewinds **getgrent()** to the beginning of the group database.

endpwent() closes the user database.

endgrent() closes the group database.

setpwent() and **setgrent()** reset **getpwent()** and **getgrent()** so that the next invocation retrieves the first entry from each database. **endpwent()** and **endgrent()** close the files.

In spite of their names, **setpwent()** and **setgrent()** don't actually change the databases--they merely change Perl's "pointer" to them, like **seek()** does for files. If you want to modify the databases, look at the C/UNIX functions **putpwent()** and **putgrent()**, which have no Perl equivalents.

These four functions aren't used very often. When they are used, it's usually in tortuous system administration applications that keep the user and/or group databases open for a long time. The *pwrewind* program (Listing 11-4) isn't exactly tortuous, but it does show how **setpwent()** and **getpwent()** behave.

Listing 11-4 *pwrewind*: Using `setpwent()` and `endpwent()`

```
#!/usr/bin/perl -wl

while (@password_fields = getpwent) {
    print join(':', @password_fields);
    if ($password_fields[0] eq `perluser`) {
        setpwent; # reset getpwent
        @password_fields = getpwent;
        print join(':', @password_fields);
        endpwent; # close the database
        exit;
    }
}
```

The `getpwent()` iterates through the password database. When it encounters *perluser*, it resets the database pointer, so that the next `getpwent()` retrieves the root entry:

```
% pwrewind
```

```
RESULT root:bvRVcaonpuxMn:0:1:System PRIVILEGED Account,,,:/bin/csh
nobody*:65534:65534:anonymous NFS user:/:
daemon*:1:1:system background account:/:
sys*:2:2:./bin/csh
bin*:3:3:./bin:
uucp*:4:8:./var/spool/uucppublic
ingres*:7:7:./usr/ingres/bin/csh
perluser:wwF12U703IYY2:1014:20:Ms.
Perl,Rm.409,555-9112:/users/perluser:/bin/csh
root:bvRVcaonpuxMn:0:1:System PRIVILEGED
Account,,,:/bin/csh
```

Chapter 11, Perl 5 Interactive Course

LESSON 2

UNIT 11

UNIX SETUID SCRIPTS AND THE SAFE MODULE

Suppose you distribute a program that provides a questionnaire to users. You want users to fill out the questionnaire, and then you want the results to be written to a file in your home directory.

Here's the problem: You don't want to make your home directory world-writable. Nor do you want to create a huge entry in */etc/group* with the name of everyone who might possibly answer your questionnaire.

Here's a solution: Make your program *setuid*. That means that the UID of the program is set to the *owner's* UID rather than the *user's* UID. The result: UNIX thinks that the person running that program is you, so it lets the user write the file into your home directory. You can do the same thing for groups, too: You can make a *setgid* script that uses the owner's GID.

Real Versus Effective UIDs

That temporary UID is called the *effective UID*, as opposed to the *real UID*.

You can determine the real and effective UIDs of a process with the `$<` and `$>` special variables.

The `$<` and `$>` Special Variables

`$<` (or `$UID` or `$REAL_USER_ID`) is the real UID of the user running the program.

`$>` (or `$EUID` or `$EFFECTIVE_USER_ID`) is the effective UID of the user running the program.

In most cases, `$<` and `$>` will be the same. They'll only be different inside *setuid* programs.

To determine *your* UID, do this:

```
perl -e 'print $<'
```

That number should match the entry in your *passwd* file (or equivalent). That's why this is an acceptable (and even preferred) substitute for `getlogin()`:

```
(getpwuid($<))[0]
```

The special variables `$()` and `$>` do the same for groups. They'll be different only inside *setgid* programs.

The `$()` and `$>` Special Variables

`$()` (or `$GID` or `$REAL_GROUP_ID`) is the real GID of the user running the process.

`$`) (or `$EGID` or `$EFFECTIVE_GROUP_ID`) is the effective GID of the user running the process.

If your computer lets users belong to multiple groups, `$(` and `$)` will be a space-separated list of all of them.

Setuid Bits

Setuid (or setgid) programs are identified by the setuid (or setgid) bits in their permissions. Any Perl program can be made setuid, as shown in Listing 11-5.

Listing 11-5 *insecure*: Making a program setuid

```
#!/usr/bin/perl -wl

print "My real UID is $<";
print "My effective UID is $>";
```

To make *insecure* both executable and setuid:

```
% chmod +sx insecure
```

If you run *insecure* under your account, you won't notice anything out of the ordinary:

```
% insecure
```

```
RESULT My real UID is 1014
```

```
My effective UID is 1014
```

But if someone else runs the program:

```
% su joebob
```

```
RESULT Password: <enter joebob's password>
```

it'll have your effective UID and can create and delete files owned by you.

```
% insecure
```

```
RESULT My real UID is 1015
```

```
My effective UID is 1014
```

You might not see a difference between the regular and setuid behaviors, even after you've used `su` to become someone else. That might be because your computer ignores the setuid bit. Or your computer might outlaw setuid programs altogether for security reasons. If your computer merely ignores the setuid bit, Perl can sometimes emulate setuid behavior for you. Here's what happens: Your `/usr/bin/perl` (or its equivalent on your system) executes a different Perl interpreter, `/usr/bin/suidperl`, in its place.

Tainted Variables

What makes an operation unsafe? Well, suppose that a program prompts the user for the name of a file to create:

```
chomp($file = <>)
```

To be safe, the program runs inside a particular directory so users won't be able to modify any important

files.

But the user could enter, say, */etc/passwd* and overwrite the */etc/passwd* file with one of his or her own choosing. Not good.

When Perl is running a `setuid` or `setgid` program, it prevents "tainted" variables (such as `$file` above) from being used for operations like `open()`, `chmod()`, `unlink()`, `kill()`, or anything else that might pose a threat. Listing 11-6 shows an example.

Listing 11-6 *sudanger*: A tainted script

```
#!/usr/bin/perl -w

print `Enter a filename to delete: `;
chomp($file = <>); # $file is tainted!

unlink $file; # ...so this is unsafe.
print "OK!\n";
```

If you run this program as is, Perl won't complain. After all, the program isn't doing anything you couldn't do from the command line. But if you make it `setuid`, using `chmod()`:

```
% chmod +sx sudanger
```

and then run it, you might see this complaint:

```
% sudanger
```

```
RESULT YOU HAVEN'T DISABLED SET-ID SCRIPTS IN THE KERNEL YET!
```

```
FIX YOUR KERNEL, PUT A C WRAPPER AROUND THIS SCRIPT, OR USE
-u AND UNDUMP!
```

or this one:

```
% sudanger
```

```
RESULT Enter a filename to delete: myfile
```

```
Insecure dependency in unlink while running setgid at
./sudanger line 6, <> line 1.
```

If you can derive an untainted variable from `$file` by matching and removing any suspicious characters, Perl will notice and let your program proceed, as shown in Listing 11-7.

Listing 11-7 *suhaven*: Untainting a variable

```
#!/usr/bin/perl -w

print `Enter a filename to delete: `;
chomp($file = <>); # $file is tainted

$file =~ /^\\W*(\\w*)/;
$good_file = $1; # Untaint $file.

unlink $good_file; # This is now a safe operation.
```

```
print "OK!\n";
```

Notice that *suhaven* parenthesizes the "safe" part of \$file (that's the \w* part) and then sets \$file to \$1. You have to set \$file to that value explicitly--s/// wouldn't have worked--to help Perl recognize the new \$file as untainted.

```
% suhaven
```

```
RESULT Enter a filename to delete: newfile
```

```
OK!
```

To check whether your script uses tainted variables, use the **-T** flag.

The -T Command-Line Flag

-T checks for tainted variables.

-T is honored by Perl even if the script isn't running setuid.

The Safe Module

Proponents of the Java language tout its ability to execute code in a quarantined environment, so that you can run someone's code (downloaded, say, from a Web site) without any danger of losing your precious files or compromising security. In Chapter 12, you'll learn about Penguin, an alternative to Java that lets you program in Perl. Penguin's security stems from the Safe module by Malcolm Beattie and Tim Bunce, available from the CPAN.

The Safe module lets you create self-contained compartments for evaluating Perl code, similar to eval(), but with security in mind. First, each compartment has an operator mask that specifies a list of operations (for example, system(), open(), chown(), and shmget()) that could be used to compromise your computer or network. Any attempt to use those operations results in the program failing during compilation. Second, each Safe compartment contains its own namespace--main:: is changed to a different package so that code evaluated in the compartment can't refer to variables in other packages.

Chapter 11, Perl 5 Interactive Course

LESSON 3

ENCRYPTION

When you *encrypt* data, you encode it so that only people holding a certain key (such as a secret password) can read it. This lesson describes a few different techniques for encrypting your data from within a Perl program.

crypt()

UNIX never stores unencrypted passwords. That would be a monumental security risk. Instead, it stores *encrypted* passwords--those bizarre strings of characters after the usernames in */etc/passwd*.

The crypt() Function

crypt(PASSWORD, SALT) encrypts *PASSWORD* (using the help of *SALT*), returning the encrypted password.

There's no known way to decrypt the result other than guessing a password and seeing whether you get the same result when you crypt() it. That's how most password-crackers work--they start with a dictionary of words and encrypt each one, seeing whether it matches the user's encrypted password.

crypt() encrypts only the first eight characters of PASSWORD because UNIX passwords are assumed to be no longer than that length.

The salt is a pair of characters used to perturb the encryption algorithm, after which the salt is prepended to the encrypted string. When you choose a new password, the password program picks a salt at random.

```
crypt("sesame", "HO")
```

encrypts the password sesame with the salt HO and returns the encrypted password:

```
HOqHlINI8THzY
```

The first two characters of the encrypted password are the salt. That's how the operating system can verify that you are who you say you are. When you log in, you type your password and the login program extracts your salt from the password database so that both arguments to crypt() are correct.

Why use salt? Well, remember that crypt() isn't random. It behaves the same way on every computer. So if you want to crack password files, you could take a list of likely passwords, encrypt each word once, and then waltz around from computer to computer seeing whether any of the encrypted passwords match. You can still do this, but the salt makes it 4,096 times harder, because you have to encrypt every password for every possible salt. Listing 11-8 shows *verify*, a program that checks to see whether the user's password is what he or she says it is.

Listing 11-8 *verify*: Using crypt() to verify the user's password

```
#!/usr/bin/perl -w
print "What's your password? ";
chomp($password = <>);

# Grab the user's salt from the password database
$encrypted_password = (getpwuid($<))[1];
$salt = substr($encrypted_password, 0, 2);

if (crypt($password, $salt) eq $encrypted_password) {
    print "Correct! Now I know your password, sucker.\n";
} else {
    print "That's not your password.\n";
}
```

In general, you shouldn't be typing your password into random programs that you find lying around. But *verify* is simple enough: It encrypts the user's plain text password with the salt from the password file and tests to see whether the result matches the user's actual encrypted password.

Listing 11-9 presents a program that enhances security on your system by identifying easily "crackable" passwords, using `getpwent()`, `crypt()`, and a list of possible passwords (*/usr/dict/words*).

Warning! If you run this program, certain multinational corporations may prosecute you.

Listing 11-9 *cracker*: Using crypt() to check for insecure passwords

```
#!/usr/bin/perl -w

open (WORDS, ` /usr/dict/words `) or die "Can't read
/usr/dict/words: $! \n";
while (@passwd_entries = getpwent) {
    $user = $passwd_entries[0];
    print "Checking $user...\n";
    $encrypted_password = $passwd_entries[1]; # get the
    encrypted password
    next if $encrypted_password =~ /\*/;
    $salt = substr($encrypted_password, 0, 2); # extract
    the salt
    seek (WORDS, 0, 0);
# for every dictionary word, see whether it's a match.
    while (<WORDS>) {
        chomp;
        if (crypt($_, $salt) eq $encrypted_password)
        {
```

```

        print "$user\'s password is $_\n";
        last;
    }
}
}

```

DES

The Data Encryption Standard (DES) is a public-domain encryption standard developed by the U.S. government. `crypt()` uses a variant of DES.

Eric Young developed a DES package for Perl; you can find a copy at <ftp://sunsite.unc.edu/pub/languages/perl/scripts/des.Z>.

The DES algorithm encrypts data in 8-byte chunks, so to use it for larger blocks of text, you need to break up your text into 8-byte sized pieces. The *wrapdes* program (Listing 11-10) demonstrates this for you.

Listing 11-10 *wrapdes*: Using the DES routines to encrypt and decrypt text

```

#!/usr/bin/perl -w

require `des` ; # available from ftp://sunsite.unc.edu
sub des_encrypt {
    my ($chunk, $i);
    print "\nEncrypted text: \n";
    # encrypt every 8 character chunk
    for ($i = 0; $i < length($text)/8; $i++) {
        $chunk = substr($text, $i*8, 8);
        $chunk .= ' ' x (8 - length($chunk)); # 1
        means "encrypt"
        $encrypted_text .= des_ecb_encrypt(*key, 1,
        $chunk);
    }

    print $encrypted_text, "\n";
}
sub des_decrypt {
    my ($decrypted_text, $i);
    print "\nDecrypted text: \n";
    # decrypt every 8-character chunk
    for ($i = 0; $i < length($encrypted_text)/8; $i++) {
        $decrypted_text = des_ecb_encrypt(*key, 0,

```



```

        substr($encrypted_text, $i*8, 8));
        print map(chr, unpack("C8",
        $decrypted_text));
    }
}
# choose your secret 8 characters
@key = des_set_key(pack("C8", 0x12, 0x34, 0x45, 0x67, 0x89,
0xab, 0xcd, 0xef));
print `Enter the text to be DES-encrypted: `;
chop($text = <>);
des_encrypt;
des_decrypt;

```

Assuming you've retrieved and installed the DES package, you can encrypt any length of text with *wrapdes*.

```
% wrapdes
```

RESULT Enter the text to be DES-encrypted: Squeamish Ossifrage

Encrypted text:

```
*'| |i..*~+0c']$*2ay,a~©j.a`a~4a~
```

RESULT Decrypted text:

Squeamish Ossifrage

Unlike `crypt()`, the DES algorithm produces 8-bit ASCII, which is why the encrypted text looks funny.

RSA

The Rivest Shamir Adelman (RSA) algorithm is a patented technique providing *public key* encryption. In public key encryption, every user has a private key and a public key. You encrypt your data with your secret private key; others use your public key to verify that you were the one who did the encrypting.

RSA encryption is used by Pretty Good Privacy (PGP), a popular program for encrypting electronic mail. Adam Back (and others) has condensed an implementation of the RSA algorithm to four lines of Perl code (or three, depending on how you count), as shown in Listing 11-11.

Listing 11-11 An implementation of RSA encryption

```

#!/bin/perl -sp0777i<X+d*1MLa^*1N%0]dsXx++1M1N/dsM0<j]dsj
$/=unpack(`H*`,$_);$_=`echo 16dio\U$k"SK$/SM$n\EsnOp[1N*1
1K[d2%Sa2/d0$^Ixp"|dc`;s/\W//g;$_=pack(`H*`,/((...)*$/)

```

Sometimes encryption can be pretty cryptic. To use this script, you need to generate your own private and public key pair. That's beyond the scope of this book; consult Simson Garfinkel's *PGP: Pretty Good Privacy* (O'Reilly & Associates, 1995) or *Bruce Schneier's Applied Cryptography* (2nd ed., John Wiley & Sons, 1995). To test the script, you can use short keys:

```

% echo My secret message | rsa -e -k=4f -n=d09 | rsa -d
-k=3fb -n=d09

```

My secret message

or long keys (from Adam Back's .sig file, visible from <http://dcs.ex.ac.uk/~aba/>):

```
% echo squeamish ossifrage | rsa -e -k=3 -n=7537d365 | rsa -d  
-k=4e243e33 -n=7537d365  
squeamish ossifrage
```

Chapter 11, Perl 5 Interactive Course

LESSON 4

TIED VARIABLES

You probably think that you know how scalars, arrays, and hashes operate. But what if you could *redefine* how they operate? For instance, what if you could define an "implementation" for a hash that automatically saves the key-value pairs to disk every time you modify them? Presto--an instant database manager.

Of course, you don't actually want to redefine hashes. That would confuse Perl, because it expects hashes to behave in certain ways. Instead, you can construct your own class that describes how your hash should behave. You want your hashes to behave like regular hashes (so you can say `$hash{key} = 'value'`, and `keys %hash`, etc.), but you also want your class methods to be triggered automatically when your program stores or fetches values.

That's what the `tie()` function does.

The `tie()` and `untie()` Functions

`tie(VARIABLE, CLASS, LIST)` makes *VARIABLE* behave by the rules of *CLASS*.

`untie(VARIABLE)` makes *VARIABLE* normal again.

You can't `tie()` a variable to just any class. It has to contain certain methods with predefined names. If your class redefines a scalar, there are four methods you should include:

- **`FETCH()`**: This method is invoked automatically when you access the variable.
- **`STORE()`**: This method is invoked automatically when you store a value in the variable.
- **`DESTROY()`**: This method is invoked automatically when Perl wants to destroy the variable, such as when your program ends.
- **`TIESCALAR()`**: This method is invoked automatically when you call `tie()` with a scalar variable.

A class implementing an array needs `FETCH()`, `STORE()`, `DESTROY()`, and `TIEARRAY()` methods, and a class implementing a hash needs `FETCH()`, `STORE()`, `DESTROY()`, `TIEHASH()`, `DELETE()`, `EXISTS()`, `FIRSTKEY()`, and `NEXTKEY()`. The `SubstrHash` module, which implements a hash table with fixed-length keys and values, can be used as a starting point if you want to construct your own `tie()`able classes.

You can `untie` that variable from the class, which makes it a normal variable again--the special methods you've defined will no longer be called for that variable.

myarray1, shown in Listing 11-12, defines a new type of array called MyArray. MyArrays don't do anything special other than print when array fetches and stores occur.

Listing 11-12 *myarray1*: Using tie() and untie() to define an "implementation" for @x

```
#!/usr/bin/perl -wl
package MyArray;
sub TIEARRAY {
    print "TYING";
    bless [];
}
sub DESTROY {
    print "DESTROYING";
}
sub STORE {
    my ($self, $index, $value) = @_;
    print "STORING $value at index $index";
    $self[$index] = $value;
}
sub FETCH {
    my ($self, $index) = @_;
    print "FETCHING the value at index $index";
    return $self[$index];
}
package main;
$object = tie @x, MyArray; # @x is now a MyArray array.
print "object is a ", ref($object);
$x[0] = 'value'; # This will call STORE().
print $x[0]; # This will call FETCH().
print $object->FETCH(0); # (You can call FETCH() too, if you
want.)
untie @x; # Now @x is a normal array again.
```

When @x is tied to the MyArray package, \$object is returned. That's the actual class object, but you don't care. Here's what matters:

```
$x[0] = 'value'
```

calls STORE(), and

```
print $x[0]
```

calls FETCH().

Because these are normal methods (with special names), you can call them as you normally would:

```
$object->FETCH(0)
```

Finally, @x is untie()d, making it a normal array again with all its values intact.

Because these methods contain print()s, you can see when they're being called:

```
% myarray1
```

```
RESULT TYING
```

```
object is a MyArray
STORING value at index 0
FETCHING the value at index 0
value
FETCHING the value at index 0
value
DESTROYING
```

The CountArray class, featured in Listing 11-13, remembers how many times an element has been accessed. When you tie() an array to the CountArray package, you create a double-sized array. For each normal array element, a CountArray object uses two elements in *its* array: The first holds the real value and the second holds the number of accesses.

When you set, say, the fourth element of an array that's been tied to CountArray, STORE() translates that to the *eighth* element of its array; the ninth element is then reserved for the number of accesses.

When you access the fourth element, FETCH() returns the eighth element of its array so you get the value you expect. It also increments the number of accesses stored in the ninth element.

Listing 11-13 *myarray2*: Using tie() to count accesses

```
#!/usr/bin/perl -wl
package CountArray;
sub TIEARRAY {
    return bless [];
}
sub STORE {
    my ($self, $index, $value) = @_;
    $self[2 * $index] = $value;
}
sub FETCH {
    my ($self, $index) = @_;
    $self[2 * $index + 1] ++; # there's another one!
    return $self[2 * $index]; # return the value expected
    at @x[$index]
}
sub how_many_fetches {
    my ($self, $index) = @_;
    return $self[2 * $index + 1]; # return how many times
    $index was accessed.
```

```
}  
package main;  
$object = tie @x, CountArray;  
$x[4] = `schnittzel`;  
print $x[4];  
print "x[4] has been fetched ", $object->how_many_fetches(@x,  
4), " times.";   
print $x[4];  
print "x[4] has been fetched ", $object->how_many_fetches(@x,  
4), " times.";   
print $x[4];  
print "x[4] has been fetched ", $object->how_many_fetches(@x,  
4), " times.";   
untie @x;  
% myarray2
```

RESULT schnitzel

```
x[4] has been fetched 1 times.  
schnittzel  
x[4] has been fetched 2 times.  
schnittzel  
x[4] has been fetched 3 times.
```

Chapter 11, Perl 5 Interactive Course

LESSON 5

DATABASE MODULES

In the next three lessons, you'll learn about how your programs can interface with database engines. These systems are external to Perl; in general, these lessons describe how you can use Perl modules (or Perl 4 *extensions*, which are Perl utilities that need to be compiled into your Perl 4 executable) to manage database engines that have been installed separately on your computer.

This lesson deals with public domain *database managers* (DBMs), which manage the storage and retrieval of large collections of key-value pairs, allowing you to manipulate the data in a variety of ways. It's a stretch, but you can think of a DBM database as a hash that can be written to disk (as long as you don't try to store references!). More powerful database engines are discussed in Lessons 7 and 8.

Perl 4 uses `dbmopen()` and `dbmclose()` to manage databases. In Perl 5, those functions are superseded by the more generic `tie()` and `untie()`. `dbmopen()` and `dbmclose()` are still around, but only for backward compatibility.

Perl "knows" about five public domain database managers.

- **NDBM:** Comes bundled with many UNIX systems; not FTPable
- **ODBM:** Comes bundled with many UNIX systems; not FTPable
- **GDBM:** The GNU database manager, available from <ftp://prep.ai.mit.edu/pub/gnu>
- **Berkeley DB:** The most commonly used DBM, available from <ftp://ftp.cs.berkeley.edu/ucb/4bsd>
- **SDBM:** A simple, small database manager included in the Perl distribution

See the Perl `AnyDBM_File` documentation (embedded in the module of the same name) for more information on the differences between these systems. Of the five, only SDBM is guaranteed to be on your system, so that's what most of the examples in this lesson use. Listing 11-14 shows a hash tied to a database manager.

Listing 11-14 *storedb*: Using a DBM to store a hash on disk

```
#!/usr/bin/perl -w
use POSIX; # needed for O_RDWR and O_CREAT
use SDBM_File;
# Create an SDBM hash
```

```
tie(%hash, SDBM_File, 'movies-sdbm', O_RDWR | O_CREAT, 0666);
$hash{'The Seven Samurai'} = 1954;
$hash{'City Lights'} = 1931;
$hash{'The Third Man'} = 1949;
untie %hash; # not necessary, but good form
```

storedb first ties the hash %hash to the SDBM_File package. As items are stored in the hash, they are written to the database files *movies-sdbm.pag* and *movies-sdbm.dir* (the extensions have a special meaning to SDBM and are appended automatically). Those files are created because the O_CREAT flag was used; they will be readable and writable by everyone because the 0666 mode was used.

You can access the contents of the *movies-sdbm* files using tie() without the O_CREAT argument, as shown in Listing 11-15.

Listing 11-15 *fetchdb*: Accessing a DBM disk file

```
#!/usr/bin/perl -wl
use POSIX;
use SDBM_File;
tie(%hash, SDBM_File, 'movies-sdbm', O_RDONLY, 0666);
print "City Lights was released in $hash{'City Lights'}.";
untie %hash;
% storedb
% ls -l movies*
-rw-r--r-- 1 perluser staff 0 May 27 00:37 movies-sdbm.dir
-rw-r--r-- 1 perluser staff 1024 May 27 00:37 movies-sdbm.pag
% fetchdb
City Lights was released in 1931.
```

The interfaces to all the DBMs are similar, even if the database managers themselves aren't. Berkeley DB, for instance, uses only one file to store the database, instead of the *.dir* and *.pag* organization used by SDBM and NDBM. It's also the only DBM that stores its data in a byte-order-independent format.

The DB_File documentation (embedded pod-style in *DB_File.pm*) contains some useful information for using the Berkeley DB engine from Perl. Berkeley DB supports three different types of databases, identified by the last argument to tie(): \$DB_HASH (standard key-value pairs), \$DB_BTREE (key-value pairs stored in a sorted and balanced binary tree), and \$DB_RECNO (key-value pairs for flat text files).

Listing 11-16 *hashberk*: Using DB_File to store a regular hash

```
#!/usr/bin/perl -w
use DB_File;
use Fcntl; # needed to define O_RDWR and O_CREAT
tie %hash, DB_File, "hashberk.db", O_RDWR | O_CREAT, 0666,
$DB_HASH;
$hash{'Pulp Fiction'} = 1994;
$hash{"Schindler's List"} = 1993;
$hash{'The Prince of Tides'} = 1949;
```



```

$hash{`The Seven Samurai`} = 1954;
$hash{`City Lights`} = 1931;
$hash{`The Third Man`} = 1949;
untie %hash;

```

hashberk (Listing 11-16) behaves like *fetchdb*, but note the extra argument provided to *tie()*: `$DB_HASH`. With this argument, `DB_File` stores your hash in that efficient but unpredictable hash order. If you want to specify your own ordering, you can store your hash (now it really should be called an associative array) as a btree by using the `$DB_BTREE` argument instead of `$DB_HASH`, as shown in Listing 11-17.

Listing 11-17 *treeberk*: Using `DB_File` to store a btree

```

#!/usr/bin/perl -w
use DB_File;
use Fcntl;
sub by_length {
    my ($key_a, $key_b) = @_;
    length($key_a) - length($key_b);
}
$DB_BTREE->{compare} = 'by_length';
tie(%btree, DB_File, "treeberk.db", O_RDWR | O_CREAT, 0666,
$DB_BTREE);
$btree{`Pulp Fiction`} = 1994;
$btree{"Schindler`s List"} = 1993;
$btree{`The Prince of Tides`} = 1949;
$btree{`The Seven Samurai`} = 1954;
$btree{`City Lights`} = 1931;
$btree{`The Third Man`} = 1949;
foreach $key (keys %btree) { # Keys will be retrieved by
length!
    print $key, "\n";
}
untie %btree;

```

When `%btree` is tied to a `DB_File` as a btree, the `DB_File` checks to see whether `$DB_TREE{compare}` has been defined. If it hasn't, `DB_File` uses standard string comparison to construct the binary tree. But *treeberk* does define its own comparison function, `by_length()`, which returns a positive value if the first key is longer than the second key. The result: The associative array is ordered by the length of its keys, from shortest to longest.

```
% treeberk
```

```

RESULT City Lights
Pulp Fiction
The Third Man
Schindler`s List

```

The Seven Samurai The Prince of Tides

Suppose you want to organize the btree by year. Here's what you DON'T want to do:



```
sub by_value {  
    my ($key_a, $key_b) = @_;  
    $hash{$key_a} - $hash{$key_b};  
}
```

because `$hash{$key_a}` is itself a database access, which needs to call `by_value()`, which needs to call `$hash{$key_a}`, which needs to call `by_value()`, and so on. You'll fare better if you copy the array (say, from `%oldhash` to `%hash`) before the `tie()` so you can do this:

```
sub by_value {  
    my ($key_a, $key_b) = @_;  
    $oldhash{$key_a} - $oldhash{$key_b};  
}
```

Chapter 11, Perl 5 Interactive Course

LESSON 6

DBI, THE DATABASE INTERFACE

Commercial database products (Oracle, Sybase, Informix, Ingres, and so on) can do a lot of things that the various DBMs can't. Typically, they support such nifty features as file locking and a built-in query language for retrieving selected portions of your database. Of course, there's a downside: They aren't free. (Well, some are: Old versions of Ingres, and its successor Postgres, are available from <ftp://s2k-ftp.CS.Berkeley.edu/pub>.)

Some benevolent programmers have constructed Perl interfaces for these products so that you can access the product from within Perl *if the product is installed on your system*. But if you haven't bought an Oracle database engine, don't bother with either of the Oracle interfaces described in this chapter.

In many cases, these interfaces were developed for Perl 4, which isn't able to load extensions dynamically--they have to be compiled into your Perl 4 executable.

Enter the Perl Database Interface, or DBI. The DBI is an effort to provide a uniform, object-oriented interface to all database engines so that a script developed for Oracle will be (mostly) portable to Ingres. The DBI is a *switch* (it's also been called a *driver manager*) that resides between your Perl program and a *database driver*, or DBD. Each DBD knows how to talk to one database package. At publication time, there are eleven DBDs: Oracle, Informix, Sybase, mSQL, Solid, Qbase, DB2, Fulcrum, Ingres, and PostgreSQL. More are being built all the time.

Using DBI

Before you use any DBD, you first need the DBI, which intercepts your function calls and routes them to the appropriate DBD. You can retrieve it from the CPAN; see Appendix B.

It's hard to demonstrate the DBI without assuming the existence of a particular database system on your computer. Luckily, the leader of the DBI effort, Tim Bunce, has provided the ExampleP DBD, which is an interface to *fstat* (you can think of your filesystem as a database, in which each file is a record with certain attributes: name, size, access time, and so on).

testDBI, shown in Listing 11-18, demonstrates only a tiny portion of the DBI features. For more information, consult the DBI API Specification (available from <ftp://ftp.demon.co.uk/pub/perl/db/DBI/>) or join the mailing list by sending mail to perldb-interest-request@fugue.com.

Listing 11-18 *testDBI*: Using DBI with the ExampleP DBD

```
#!/usr/bin/perl -w
use DBI;
# which databases exist?
print "You have the following drivers (DBDs): ";
print "\t ", join(' ', DBI->available_drivers), "\n";
$driver_handle = DBI->install_driver('ExampleP'); # ExampleP
is always there
$database_handle = $driver_handle->connect(); # Begin a
session
# Construct a query
$cursor = $database_handle->prepare('select name,atime from
?');
print "There are $cursor->{NUM_OF_FIELDS} fields.\n";
# Apply query to /usr/spool
$cursor->execute('/usr/spool');
# Retrieve query results
while (@row = $cursor->fetchrow) {
    print "$row[0] was last accessed at ",
    scalar(localtime($row[1])), "\n";
}
$driver_handle->disconnect();
```

The first bold segment in Listing 11-18,

DBI->available_drivers

returns a list of the DBDs you have on your system. Assuming DBI has been installed correctly, it should always have at least one element: ExampleP. The *testDBI* program creates two handles: a *driver* handle and a *database* handle. The driver handle is used to create the database handle via its `connect()` method. (The driver handles will vary somewhat, but they all support `connect()`.)

The `prepare()` method is used to convert the query `'select name,atime from ?'` into a cursor object, which will have a `fetchrow()` method defined. `fetchrow()` returns an array of the `prepare()`d fields: name and atime (access time).

% testDBI

RESULT You have the following drivers (DBDS):

```
ExampleP There are 2 fields.
. was last accessed at Wed May 15 12:35:54 1996
.. was last accessed at Wed May 15 2:01:18 1996
cron was last accessed at Tue May 14 4:40:01 1996
mail was last accessed at Wed May 15 12:08:46 1996
mqueue was last accessed at Wed May 15 12:31:49 1996
lpd was last accessed at Mon May 13 7:33:55 1996
locks was last accessed at Mon May 13 7:33:55 1996
uucp was last accessed at Mon May 15 7:33:55 1996
```

uucppublic was last accessed at Mon May 15 7:33:55 1996

lpd.lock was last accessed at Mon Jan 8 9:11:00 1996

DBI Methods

No matter what the database engine, the DBD should supply the following core functions that interact with the database engine:

- **connect()** begins every session with a database.
- **disconnect()** ends the session (without saving anything).
- **prepare()** takes a command to be executed by the database system and returns a statement handle (call it a cursor) suitable for feeding to **execute()**.
- **execute()** executes the command **prepare()**d for the cursor.
- **fetchrow()** returns an array consisting of the fields in a particular row.
- **finish()** releases all resources associated with a cursor.
- **commit()** performs a transaction (if the system supports transactions).
- **rollback()** undoes a transaction (if the system supports transactions).

For more information about DBI, see *CPAN/modules/DBI* or issue 5 of *The Perl Journal*.

Chapter 11, Perl 5 Interactive Course

LESSON 7

NON-DBI INTERFACES, PART I

Many existing database interfaces haven't yet been integrated into the DBI. Enough to fill the next two lessons, in fact. All these interfaces (with the exception of one; see the Sybase discussion later in this lesson) are Perl 4 extensions that you use to create an entirely new Perl executable. In general, you won't be able to install (or even compile) these interfaces if you're using Perl 5; you need Perl 4, because these extensions require the *upperl.o* file in the Perl distribution that has since been supplanted by dynamic loading.

Perl 4 Extensions

Every Perl 4 extension requires creating a completely new Perl executable. Perl 4 extensions are not compatible with Perl 5.

These Perl 4 interfaces are often called *database Perls*, not to be confused with *dbPerl*, which is an old name (one of many) for the DBI.

Oracle

The most commonly used database Perl is Kevin Stock's OraPerl, which is compatible with versions of Oracle up to Oracle 7. Oracle 7 features row- and table-level locking, clustered tables, a shared SQL cache, binary data support, database triggers, and parallel backup and recovery of files.

OraPerl makes a number of functions available, corresponding to the C API for Oracle:

- **ora_login(), ora_logoff()**
- **ora_open(), ora_close()**
- **ora_do(), ora_fetch(), ora_bind()**

The latest version of OraPerl, OraPerl 2.4, can be obtained from the CPAN under *modules/dbperl/perl4/*. Check *oraperl.FAQ* and *README.oracle7* in that directory. But you might not need it: The Perl 5 DBD::Oracle module can emulate OraPerl.

Sybase

Things get a bit confusing with Sybase. There's a Perl 4 Sybase interface (Sybperl, version 1), and a Perl 5 module set (Sybperl, version 2), both by Michael Peppler. Sybase itself offers page- and table-level locking, database triggers, and binary data support.

- **Sybase::DBlib adds a subset of Sybase's DB Library API.**
- **Sybase::CTlib adds a subset of the CT Library API.**
- **Sybase::Sybperl lets you run the Perl 4 Sybperl scripts with Perl 5.**

Sybperl 2 can be obtained from the CPAN. Sybperl 1.011 can be obtained from the CPAN or <ftp://ftp.demon.co.uk/pub/perl/db/perl4/>.

Informix

Isqperl is a Perl 4 interface to Informix databases by Bill Hails. Informix features page-, row-, and table-level locking; programmable read isolation levels; and binary data support. Isqperl supports the following functions:

- **isql_open(), isql_close(), isql_shutdown()**
- **isql_execute(), isql_statement(), isql_fetch()**
- **isql_columns(), isql_titles(), isql_databases()**
- **isql_type(), isql_msg(), isql_ca()**

You can find Isqperl 1.2 on the CPAN or <ftp://ftp.demon.co.uk/pub/perl/db/perl4/>.

Ingres

A stable Ingres DBD now exists, but older applications made use of Tim Bunce's *ingperl* (a rewrite of Ted Lemon's *sqlperl*), which contains the following functions:

- **sql('connect dbname [-sqloptions]') and sql('disconnect')**
- **sql_exec(), sql_fetch(), sql_close()**
- **sql_types(), sql_ingtypes(), sql_lengths(), sql_nullable(), sql_names()**

Ingres offers page- and table-level locking, database triggers, and clustered tables.

You can obtain *ingperl* from the CPAN or <ftp://ftp.demon.co.uk/pub/perl/db/perl4/>.

Chapter 11, Perl 5 Interactive Course

LESSON 8

NON-DBI INTERFACES, PART II

The last lesson covered the "big four" database engines. This lesson briefly describes six other database Perls.

Interbase

interperl is a Perl 4 extension for interfacing with Interbase databases by Buzz Moschetti. It provides functions such as

- **dbready(), dbprepare(), dbstarttrans(), dbcommit(), dbfinish()**
- **dbopencursor(), dbclosecursor()**

You can get *interperl* from the CPAN or <ftp://ftp.demon.co.uk/pub/perl/db/perl4/>.

UNIFY

uniperl is a Perl 4 extension for UNIFY 5.0 by Rick Wargo. It has far too many functions to list here, and its syntax is quite different from the other database Perls. It implements all UNIFY 5.0 HLI routines.

uniperl 1.0 can be obtained from the CPAN or <ftp://ftp.demon.co.uk/pub/perl/db/perl4/>.

Postgres

pgperl is a Perl 4 extension for Postgres 2.0 by Igor Metz. Postgres (which uses a query language called PostQuel) began as a successor to Ingres (which uses the more common SQL as a query language); both were developed at the University of California at Berkeley, home of Berkeley DB. Both Ingres and Postgres have now been commercialized, but noncommercial versions are available from <ftp://s2k-ftp.cs.berkeley.edu>.

pgperl can be obtained from <ftp://ftp.demon.co.uk/pub/perl/db/perl4/pgperl/>. The home page for PostgreSQL, which is free and replaces PostQuel with SQL, is <http://www.postgresql.org/>.

C-Tree

The Faircom C-tree libraries are made available as a Perl 4 extension through John Conover's *ctreeperl*. The C-tree engine is designed for file indexing; it supports networked queries and retrieval and also has facilities for report generation. Some of its low-level functions are

- **InitCTree(), OpenCtFile(), CloseCtFile()**
- **CreateIndexFile(), CreateDataFile(), NewData(), ReleaseData()**
- **GetKey(), NextKey(), LastKey()**
- **LockCtData(), ReadData(), WriteData()**

You can retrieve *ctreeperl* from the CPAN or <ftp://ftp.demon.co.uk/pub/perl/db/perl4/>.

C-ISAM

cisamperl, a Perl 4 extension by Mathias Koerber, provides an interface to version 3 of the INFORMIX C-ISAM library. There are actually two separate executables available: *cisamperl*, which contains the calls for C-ISAM access, and *rocisperl* (read-only *cisperl*), which is the same except that all subroutines that create, modify, or delete data are disabled.

cisamperl functions look like this:

- **isbegin(), isopen(), isclose()**
- **isread(), iswrite(), islock(), isrelease()**
- **iscluster(), isaudit()**
- **iscommit(), isrollback(), isrecover()**

plus many, many more.

cisamperl/rocisperl can be found on the CPAN or <ftp://ftp.demon.co.uk/pub/perl/db/perl4/>.

X.500

The X.500 Directory User Agent system isn't a true database, but it does store information about users: phone numbers, addresses, and so on. X.500 directories can be accessed via a Perl 4 extension--*duaperl*, written by Eric Douglas. The current release of *duaperl* is 1.0a3; it contains the following functions:

- **dua_open(), dua_close()**
- **dua_add(), dua_delete(), dua_modattr(), dua_modrtn()**
- **dua_find(), dua_show(), dua_moveto(), dua_settmout()**

duaperl can be obtained from <ftp://gaudi.csufresno.edu/pub/x500>. The distribution contains a man page. Provisions for database-like systems are part of the DBI specification, so *duaperl* may someday be superseded by an X.500 DBD.

Review

Ally: We don't have any databases on our system besides the SDBM that comes with Perl, which is a little simplistic for our needs.

Matthew: Did you try getting Berkeley DB from the FTP site?

Ally: Yes, but I'd like to be able to query our database. The DBMs don't provide any way of accessing the stored data other than reading it all into memory. Ingres came with our workstations, so we'd like to use that.

Matthew: Oh. Well, you could use *ingperl*, the Ingres Perl 4 extension. Or you could check out the latest version of the Ingres DBD at ftp.demon.co.uk.

Chapter 12, Perl 5 Interactive Course

SURFING ON OYSTERS: PERL AND THE WORLD WIDE WEB

"Perl has dominated Web programming so completely that some people think the terms Perl script and Common Gateway Interface (CGI) program are synonyms."

--Jon Udell, Byte Magazine, April 1997

Much of Perl's reputation as a concise and powerful language is due to the Web. "Active" Web pages that let surfers toggle buttons and enter text have become commonplace, due to the ease of programming in Perl and the wealth of freely available utilities.

Not just Web pages are written in Perl. Web servers, Web browsers, HTML conversion utilities, Web usage analyzers, PerlScript (an embedded scripting language for the Web), and even robots that traverse the Internet looking for interesting Web pages: all contribute to Perl's popularity. In this chapter, we'll explore all of these, in particular how you can create active Web pages with Perl.

LESSON 1

A WORLD WIDE WEB PRIMER

This lesson is just about the Web--your Perl skills (which are quite extensive by now, if you've read through Chapter 11) won't be required. But you will be creating Web pages, so you'll need a browser with which to view them, such as Netscape Navigator, Internet Explorer, Mosaic, or Lynx. A list of FTP sites for these browsers (and others) is in Appendix B.

The Web (World Wide Web, or WWW) is a collection of computers that speak a particular language (the Hypertext Transfer Protocol--HTTP) to one another. A Web server on one computer provides Web pages to other users on the Internet, who view the pages with their browsers. This server-client relationship should be familiar to anyone who's retrieved files via FTP or used a database: Clients request documents; servers provide them.

If you want to see what the Web has to offer, grab a browser (they're freely available and available for free) and surf away--most browsers start you off at some interesting sites. Appendix B has some good starting points for the Web and for Perl in general.

HTML

If you want to create your own Web page for others to browse, you need to learn some Hypertext Markup Language (HTML). HTML documents are plain ASCII text files that include tags that tell Web browsers how to display the text graphically. Here's a sample:

```
<B> Here's an HTML snippet. </B>
```

The string Here's an HTML snippet is surrounded by two tags: `` and ``. Those tags tell the Web browser to display the enclosed text in **bold**. Read the `` as "enter bold mode" and the `` as "exit bold mode."

That's all there is to it. HTML is just an assortment of tags that describe how your document should look.

In this chapter, most HTML tags will be shown in boldface to distinguish them from the document text. They'll be shown in uppercase, even though the tags themselves are not case-sensitive: `` is the same as ``, and `</EMPHASIZE>` is the same as `</emphasize>`.

This lesson and the next two show you perhaps four-fifths of the features used by four-fifths of the HTML authors on the Web. But there's a lot that's not covered; for more detail, you should consult the WWW references listed in Appendix B. If you'd prefer a book, consult Ian S. Graham's *The HTML Sourcebook* (John Wiley & Sons, 1995) or Lincoln Stein's *How to Set Up and Maintain a World Wide Web Site: The Guide for Information Providers* (Addison-Wesley, 1995).

When you type in (or download) the Web pages in this chapter, you'll be able to view them with your browser. But other Internet users won't be able to see them unless you or your site is running a Web server. If there's no server at your site, Lesson 6 will help you get one and set it up. If

your site is running a Web server and you don't know where to put your pages, contact your Webmaster to find out.

Let's create the first Web page. The standard filename extension for HTML documents is *.html*, so call the page *page1.html*. (On MS-DOS computers, *html* needs to be shortened to *htm*.) You can supply whatever text you like, but the tags (in bold) should be typed as shown in Listing 12-1.

Listing 12-1 *page1.html*: An HTML document

```
<TITLE> Page One </TITLE>
<H1> Ye Olde Webbe Page </H1>
<HR>
Whither thou goest, there thou be.
```

page1.html contains three HTML commands: `<TITLE>...</TITLE>`, which contains the title of the Web page; `<H1>...</H1>`, which makes the browser display the "level 1" header *Ye Olde Webbe Page* (probably in a huge, bold font); and `<HR>`, which makes the browser display a horizontal rule (i.e., a line).

Now point your WWW browser at *page1.html*. If your browser of choice happens to be Netscape, just type

```
% netscape page1.html
```

What you see should resemble Figure 12-1. Each browser renders HTML in its own way, so expect subtle variations in the appearance. HTML isn't a page description language like PostScript; to the chagrin of graphic designers, browsers have some freedom to decide what to show and where to show it.



Figure 12-1

Page 1

If you change *page1.html*, the browser won't reflect the change until you reload the image. That's because HTTP--the language spoken between Web servers and browsers--is a connectionless protocol: The browser sends a request and retrieves a document, and that's that--the connection closes. That's in contrast to telnet or FTP, for which the server and client can remain connected for a long time.

The `<TITLE>...</TITLE>` string isn't part of the page itself. It's extra information intended for the browser (e.g., to include in a hot list). In "strict" HTML, tags such as `<TITLE>...</TITLE>` that aren't graphical instructions need to be grouped inside `<HEAD>...</HEAD>` tags:

```
<HEAD>
<TITLE>Page One</TITLE>
</HEAD>
```

Everything else should be enclosed in `<BODY>...</BODY>` tags:

```
<BODY>
<H1>Ye Olde Webbe Page</H1>
<HR>
</BODY>
```

Furthermore, the entire HTML file should be enclosed in `<HTML>...</HTML>` tags:

```
<HTML>
<HEAD>
<TITLE>Page One</TITLE>
</HEAD>
```

```
<BODY>
<H1>Ye Olde Webbe Page</H1>
<HR>
</BODY>
</HTML>
```

In practice, no browser insists on these tags. They'll be omitted in this chapter to reduce the visual clutter. Listing 12-2 shows *page2.html*, which bends over backward to use as many tags as possible.

Listing 12-2 *page2.html*: Lots of HTML tags

```
<TITLE> Page Two </TITLE>
```

```
<H1> This document demonstrates some nifty tags. </H1>
<H2> A second-level heading. </H2>
<H3> A third-level heading. </H3>
<H4> A fourth-level heading. </H4>
<H5> A fifth-level heading. </H5>
<H6> A sixth-level heading. </H6>
Here is a line break: <BR>
It's kinda like a \n. <BR>
Now I'll start a new paragraph (otherwise, all text gets strung together):<P>

This text is <B>bold</B>; this text is <I>italicized</I>.
Beneath this text is a line (a "horizontal rule"):
<HR>
Here's an ordered list:
<OL>
<LI> One </LI>
<LI> Two </LI>
<LI> Three </LI>
</OL>
<BLOCKQUOTE>
This is a BLOCKQUOTE, which might be used for a long quotation, or
for poetry, or for a chunk of text that you wish to isolate from the
rest of the document.
</BLOCKQUOTE>
Here's an unordered list:
<UL>
<LI> Red </LI>
<LI> Green </LI>
<LI> Blue </LI>
</UL>
<P ALIGN="center">
This text is centered on the page. Whoooo!
</P>
<PRE>
This text is preformatted-it's in a monospace font.
It's good for displaying Perl code, e.g.
if ($filename =~ /\.html$/) {
print "That's an HTML file.\n";
}
</PRE>
<CODE> print "You could also use the CODE tag.\n"; </CODE>
<P>
<STRIKE> These characters are stricken out. </STRIKE>
<P>
```

Most of the tags are self-explanatory; look at Figure 12-2 to see them in action. Note that you can (under certain conditions) place some tags inside others: The ordered and unordered lists have nested `...` tags to denote list elements. Also, some tags have attributes: You can use either `<P>` to begin an ordinary paragraph or `<P ALIGN="center">` to begin a paragraph that is centered on the page. (Unfortunately, not all browsers support centering.)

This document demonstrates some nifty tags.

A second-level heading.

A third-level heading.

A fourth-level heading.

A fifth-level heading.

A sixth-level heading.

Here is a line break:

It's kinda like a \n.

Now I'll start a new paragraph (otherwise, all text gets strung together):

This text is **bold**; this text is *italicized*. Beneath this text is a line (a "horizontal rule"):

Here's an ordered list:

1. One
2. Two
3. Three

This is a BLOCKQUOTE, which might be used for a long quotation, or for poetry, or for a chunk of text that you wish to isolate from the rest of the document.

Here's an unordered list:

- Red
- Green
- Blue

This text is centered on the page. Whoooo!

This text is pre-formatted- it's in a monospace font.
It's good for displaying Perl code, e.g.

```
if ($filename =~ /\.html$/) {  
    print "That's an HTML file.\n";  
}  
  
print "You could also use the CODE tag.\n";
```

~~These characters are stricken out.~~

Figure 12-2

Page 2

The <P> tag inserts a blank line between chunks of text; the
 tag is equivalent to a \n, starting a new line at that point. As you'll see in the next lesson, Web browsers ignore \n's; if you want to end a line, use
 or <P>. This is a big "gotcha" for Web beginners.

Figure 12-2 shows how *page2.html* appears when viewed with Netscape. Remember that HTML commands are really just suggestions to browsers describing how the author intended the text to be viewed. If your browser doesn't display graphics, or if you resize the window to a square inch, you will see something very different than Figure 12-2. HTML is not a page-description language (if it were, it would describe exactly how the document should appear, and browsers such as the text-only Lynx couldn't be HTML-compliant).

Note that the <CODE> tag and the <PRE> tag seem to do the same thing. The decision of how to render a tag is not part of the HTML language--it's up to the particular browser. For instance, some browsers render <CODE> the same as <PRE>, <EMPHASIZE> the same as <I>, and the same as .

Links and URLs

So far, all the tags you've seen affect the appearance of your document. But hypertext documents can include links to other documents--that's what makes them hypertext. Those links are specified with the anchor tags <A>...:

 Click here!

The <A> tag contains the attribute **HREF="http://www.waite.com/other_page.html"**. The uniform resource locator (URL) http://www.waite.com/other_page.html is an address of a document (*other_page.html*) elsewhere on the Net (www.waite.com), to be retrieved via HTTP (the protocol spoken by Web servers and Web clients to one another) via port 80 (often this number is omitted; 80 is the default). Don't worry if you don't know what ports are--they will be covered in the next chapter.

page3.html, shown in Listing 12-3, has several anchors. The first anchor is a partial URL, which is used to refer to a document "close to" the current document (on the same local network). The other anchors are all complete URLs, each specifying a different network protocol: news, http, ftp, telnet, gopher, wais, and mailto. (mailto isn't actually a network protocol; it's a little bit of Web magic that launches a local mail

program.)

Listing 12-3 *page3.html*: Creating links to other documents

```
<TITLE>Page Three</TITLE>
<H1>Some links</H1>
Here's a document in the same directory: <A HREF="page2.html">page 2</A>.
<P>
Here's the USENET newsgroup <A HREF="news:comp.lang.perl.misc">comp.lang.perl</A>.
<P>
Here are the <A HREF="http://www.metronet.com/0/perlinfo/perl5/manual/perl.html"> Perl 5
manual pages</A>.
<p>
Here's the <A HREF="ftp://rtfm.mit.edu/pub/usenet/comp.lang.perl.misc">Perl Frequently
Asked Questions list (FAQ)</A>
<P>
Log in to <A HREF="telnet://lambda.parc.xerox.com:8888">LambdaMoo</A>, or <A
HREF="telnet://localhost">your own computer</A>.
<P>
Dig in to <A
HREF="gopher://gopher.tc.umn.edu:70/11/Other%20Gopher%20and%20Information%20Servers/all">all
of the gopher servers in the world</A>
<P>
Swim through the WAIS (Wide Area Information Server) of WAISes at
<AHREF="wais://wais.com:210/">wais.com</A>
<P>
Send me e-mail at <A HREF="mailto:perluser@waite.com">perluser@waite.com</A>.
```

As you can see in the finished page, shown in Figure 12-3, the links all look pretty much the same, even though each URL uses a different service.

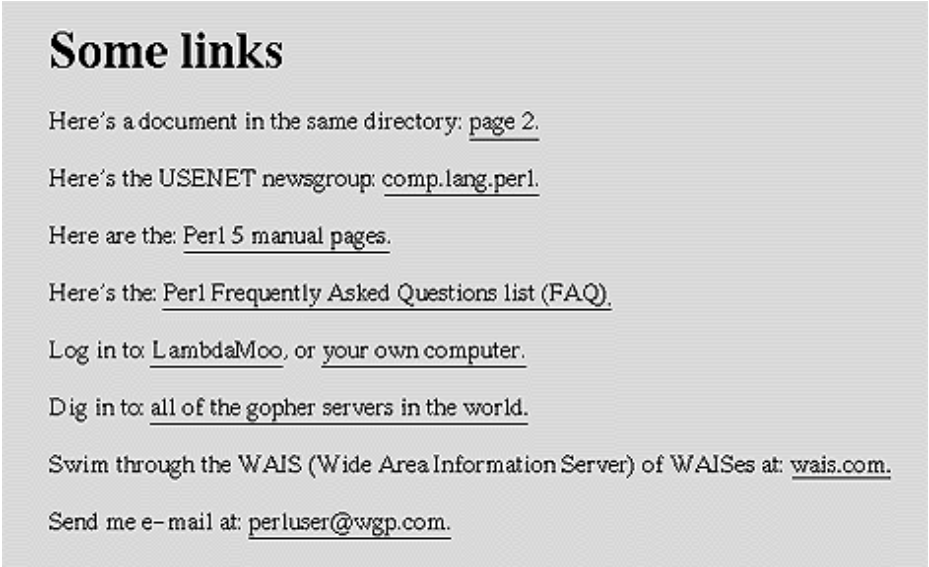


Figure 12-3
Page 3

All but the last are actual links, so click away! You might find that you can't get to certain links; that might be because the URL is no longer valid or because your browser (or your computer) doesn't support the protocol.

"My browser followed a URL to a WWW site, where it used HTTP to retrieve an HTML document, which might have been generated with a CGI program." Enough jargon for you? Table 12-1 presents a simplified glossary.

Table 12-1 A simple glossary of HTML-related terms

Term	Definition
WWW	World Wide Web: the collection of Internet computers providing hypertext documents via HTTP.

HTTP	Hypertext transport protocol: the language spoken between servers and clients.
HTML	Hypertext Markup Language: the language in which you write Web pages.
URL	Uniform resource locator: an address of something on the Net--a WWW site, a file, an FTP site, a person (an e-mail address), and so on. URLs look like this: http://www.waite.com .
CGI	Common gateway interface: a specification of how information is passed between Web servers and programs that generate Web pages.

Welcome to the world of froggies!

ribbit **RIBBIT** ribbit **RIBBIT**

Click [here](#) to turn me into a prince.

Figure 12-4

The frog home page

Chapter 12, Perl 5 Interactive Course

LESSON 2

GENERATING WWW PAGES

Building Web pages is as addictive as writing Perl programs, and for the same reason: You get immediate results. But all the Web pages so far have been static--once written, they won't change.

To write dynamic pages (such as a page that prints the results of a user query or a page that presents the user with different links depending on the time of day), you need two things: a program that generates HTML automatically and a Web server that executes the program when the user visits the page. That's where CGI comes in.

CGI

The common gateway interface (CGI) is a bridge between a Web server and CGI scripts--programs that generate HTML. The CGI itself isn't a program--just a specification of how data is passed between Web servers and programs such as the Perl scripts you'll be writing.

That information can be passed in three ways: through command-line arguments, through standard input, and through environment variables. Environment variables are the most useful.

With a CGI "gateway," data passes from the Web server to a Perl program that generates HTML documents. These programs are often called *CGI scripts* or *gateway programs*.

Getting CGI to Find Your Perl Scripts

Your Web site may expect all your CGI scripts to reside in a single directory (or set of directories), for reasons of security and simplicity. That directory is often called *cgi-bin* or *htbin*; its precise location will vary from system to system.

One popular Web server (alas, not written in Perl) is CERN's httpd. It uses a configuration file, *httpd.conf*, to specify at startup time where CGI scripts can be found. Here's the relevant line from that file. It tells the Web server on, say, www.waite.com to translate URLs that looks like http://www.waite.com/cgi-bin/my_program to a particular directory, say */project/WWW/scripts*, which should contain the program *my_program*.

```
#
# URLs starting with /cgi-bin/ will be looked for in
/project/WWW/scripts
#
Exec /cgi-bin/* /project/WWW/scripts/*
```

On a UNIX machine, CGI scripts usually run as *nobody* (that's a particular account, not an insult). *nobody* can't do very much; in particular, it (he? she?) can't execute your Perl scripts unless you make them world executable.

Listing 12-4 shows an HTML file that calls the Perl script *gbook1* when the user clicks on a particular link.

Listing 12-4 *page4.html*: Using CGI to call a Perl program from a Web page

```
<TITLE> Page Four </TITLE>
<H1> Gateway to the Guest Book program </H1>
How many people have clicked <A
  HREF="http://your.web.server/cgi-bin/gbook1">here</A>?
```



Throughout this chapter, you should replace *your.web.server* with your Internet address, for example, *www.waite.com*, and you should replace *users/perluser* with your Web directory.

Listing 12-5 shows *gbook1*.

Listing 12-5 *gbook1*: A CGI gateway script--Perl code that generates HTML

```
#!/usr/bin/perl
print "Content-type: text/html\r\n\r\n";
print `<TITLE> Guest Book I </TITLE>`;
print `<H3> Guest Book I </H3>`;
if (open (FILE, "/tmp/access.num")) {
    chomp($num = <FILE>);
    $num++;
}
print "Number of visits to this page: <H1> $num </H1><P>";
open (FILE, ">/tmp/access.num") or print "Can't write
/tmp/access.num: $!<P>";
print FILE $num, "\n";
print `Back to <A
  HREF="http://your.web.server/users/perluser/page4.html">Page
*
  Four</A>.`;
close FILE;
```

gbook1 prints HTML commands to the standard output; from the browser's point of view, it's just another HTML file. The **print "Content-type: text/html\r\n\r\n"** line at the beginning is important; that's what tells the browser that what follows is an HTML file (as opposed to plain text or an image).

When users visit *page4.html*, they'll see the page shown in Figure 12-5.

Gateway to the Guest Book program

How many people have clicked [here](#)?

Figure 12-5

Page 4

If the user clicks on [here](#), *gbook1* is launched and it generates the Web page shown in Figure 12-6.

Guest Book I

Number of visits to this page:

41

Back to [Page Four](#).

Figure 12-6

Visiting Guest Book I

If you see an error message, there are three possibilities: Your Web server doesn't know where to find your scripts, your Perl script isn't world executable, or there's an error in the script that prevents it from being run. If in doubt, try running the script yourself and check your server's error log. If you return to the Guest Book page, you'll notice a change, as shown in Figure 12-7.

Guest Book I

Number of visits to this page:

42

Back to [Page Four](#).

Figure 12-7

Suppose you want your guest book to record where people are from. You could simply ask them (using forms, the subject of the next lesson). But it's easier just to grab the hostname through the CGI interface.

CGI data is made available to gateway programs via environment variables. For Perl scripts, they show up in %ENV--the client's hostname appears in \$ENV{REMOTE_HOST}.

page5.html, shown in Listing 12-6, contains a link to *gbook2*, which accesses some of the CGI values stored in %ENV.

Listing 12-6 *page5.html*: Accessing CGI variables

```
<TITLE> Page Five </TITLE>
<H1> Gateway to the Guest Book II </H1>
Where are <A
  HREF="http://your.web.server/cgi-bin/gbook2">y'all</A> from?
```

Figure 12-8 shows Page 5, which prompts visitors to click on the link. That appends the hostname and the username (if present) to the book.

Gateway to the Guest Book II

Where are y'all from?

Figure 12-8

Page 5

Here's a sampling of the CGI variables that might be available to your Perl script. The keys are standard, but the values will be different for every combination of server, browser, and user.

```
$ENV{SERVER_SOFTWARE} = CERN/3.0
$ENV{GATEWAY_INTERFACE} = CGI/1.1
$ENV{REMOTE_ADDR} = 172.16.121.85
$ENV{SERVER_PROTOCOL} = HTTP/2.0
$ENV{REMOTE_HOST} = thrasher.plastic.edu
$ENV{REQUEST_METHOD} = GET
$ENV{HTTP_REFERER} =
http://your.web.server/users/perluser/page5.html
$ENV{REMOTE_USER} = joebob
$ENV{HTTP_USER_AGENT} = Mozilla/1.1N (X11; I; OSF1 V2.0
alpha)
$ENV{QUERY_STRING} =
$ENV{PATH} = /sbin:/usr/sbin:/usr/bin
$ENV{HTTP_ACCEPT} = */*, image/gif, image/x-xbitmap,
image/jpeg
$ENV{AUTH_TYPE} =
$ENV{SCRIPT_NAME} = /cgi-bin/gbook2
$ENV{SERVER_NAME} = your.web.server
$ENV{PATH_INFO} =
$ENV{SERVER_PORT} = 80
$ENV{REFERER_URL} =
http://your.web.server/users/perluser/page5.html
$ENV{PATH_TRANSLATED} =
$ENV{REMOTE_IDENT} =
```

`$ENV{REMOTE_HOST}` contains the name of the computer visiting the Web site, and `$ENV{REMOTE_USER}` contains the username of the visitor (sometimes!), so *gbook2* (Listing 12-7) can use that to construct a Guest Book of sorts.

Listing 12-7 *gbook2*: Using CGI environment variables to record who is connecting and from where

```
#!/usr/bin/perl
print "Content-type: text/html\r\n\r\n";
print "<TITLE> Guest Book II </TITLE>";
print "<H1> Guest Book II </H1>";
while (($key, $value) = each %ENV) {
    print "$key = $value<P>";
}
print "<H2>This site has been graced by:</H2>";
if (open (FILE, "/tmp/gbook2.log")) {
    while (<FILE>) {
        print "$_<P>";
    }
}
open (FILE, ">>/tmp/gbook2.log") or print "Can't add to the
guest book: $!<P>";
if ($ENV{REMOTE_USER}) {
    print FILE "$ENV{REMOTE_USER}\@$ENV{REMOTE_HOST}\n";
} else {
    print FILE "someone\@$ENV{REMOTE_HOST}\n";
}
print "Back to <A
HREF='http://your.web.server/users/perluser/page5.html'>Page
*
Five</A>.";
```

When the user clicks on the Page 5 link, he or she will see everyone who clicked on the link previously (Figure 12-9) and because that user clicked on the link, the hostname and username will be appended to the end of the list for all future visitors to see, as shown in Figure 12-10.

Guest Book II

This site has been graced by:

joebob@wgp.com

thraggle@state.edu

tater@corp.com

da_prez@whitehouse.gov

someone@arpa.mil

Back to [Page Five](#).

Figure 12-9

Visiting Guest Book II

Guest Book II

This site has been graced by:

joebob@wgp.com

thraggle@state.edu

tater@corp.com

da_prez@whitehouse.gov

someone@arpa.mil

perluser@corp.com

Back to [Page Five](#).

Figure 12-10

Revisiting Guest Book II

Chapter 12, Perl 5 Interactive Course

LESSON 3

HTML FORMS

If you want your Web visitors to be able to enter text in your page, press buttons, or select from pulldown menus, you need FORM tags. Thanks to the CGI, FORM data is made available to Perl scripts through an environment variable: `$ENV{QUERY_STRING}`. If the user toggles a checkbox or types his or her name into a TEXTAREA or presses a RADIO button (more about all of those soon), that information then becomes available to your Perl scripts through `$ENV{QUERY_STRING}`.

The scripts in this lesson use the `CGI::Request` module, which might already be in your *LIB* directory, but probably isn't. `CGI::Request` is discussed in more detail in Lesson 5, but to run the program in this chapter, you should get it now. You can obtain it from <ftp://www.perl.com/pub/perl/ext/CGI> and the CPAN.

Radio Buttons

Our first example of a `<FORM>` tag uses radio buttons, so named because they're like the buttons on a car radio. The selections are exclusive--when you select one, you "de-select" all the others. They're demonstrated in Listing 12-8 inside a multiline `<FORM>...</FORM>` tag.

Listing 12-8 *page6.html*: Using HTML forms to collect input from visitors to your Web page

```
<TITLE> Guest Book III </TITLE>
<H1> Visitor Feedback </H1>
Pleez fill out this questionnaire so we can make our site
better.
<FORM ACTION="http://your.web.server/cgi-bin/getradio">
I am:
<MENU>
<LI> <INPUT TYPE="radio" NAME="mood" VALUE="very happy"
CHECKED >very happy
<LI> <INPUT TYPE="radio" NAME="mood" VALUE="happy" >somewhat
happy
<LI> <INPUT TYPE="radio" NAME="mood" VALUE="satisfied"
>mildly satisfied
<LI> <INPUT TYPE="radio" NAME="mood" VALUE="annoyed" >rather
annoyed
</MENU>
```


with this site.

<P>

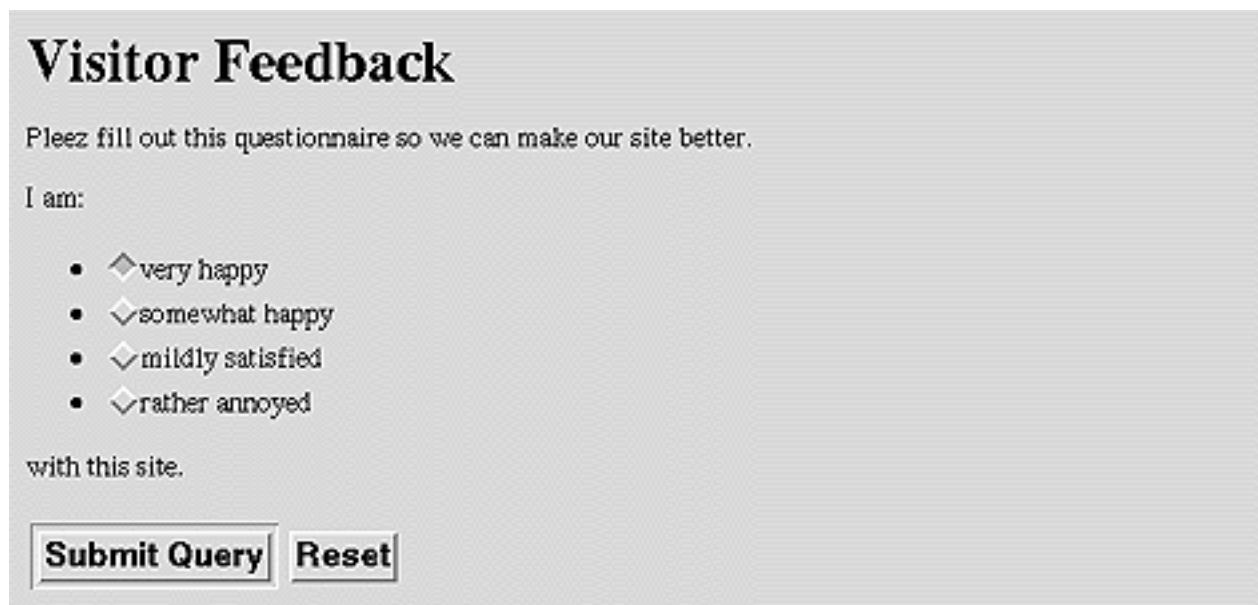
<INPUT TYPE="submit"> <INPUT TYPE="reset">

</FORM>

Here's what the different tags mean:

- **<FORM>...</FORM>:** a user-input area; the user's actions are sent to the CGI program *getradio*, located in the *cgi-bin* directory.
- **<MENU>...</MENU>:** a list of menu items, each designated by an **** tag.
- **<INPUT TYPE="radio"...>:** a radio button. There are four; the user's selection determines which **VALUE** (very happy, happy, satisfied, or annoyed) is passed (under the identifier **mood**) to *getradio*. The **CHECKED** attribute in the first list item makes very happy the default selection.
- **<INPUT TYPE="submit">:** a **SUBMIT** button, which sends the form data to *getradio* when pressed.
- **<INPUT TYPE="reset">:** a **RESET** button, which resets the form data to its original state (with the very happy button **CHECKED**).

Figure 12-11 shows the result.



The screenshot shows a web form titled "Visitor Feedback" in a large, bold, serif font. Below the title is a line of text: "Pleez fill out this questionnaire so we can make our site better." followed by "I am:". Below this is a list of four radio buttons, each followed by a text label: "very happy", "somewhat happy", "mildly satisfied", and "rather annoyed". The first radio button is selected. Below the list is the text "with this site." At the bottom of the form are two buttons: "Submit Query" and "Reset".

Figure 12-11

Page 6: Radio buttons

The HTML generated by the *getradio* program, in Listing 12-9, depends on the **<FORM>** input: which button the visitor pressed. It could also store the form data in an external file, send you mail, or compute the square root of 12. CGI scripts are just regular Perl programs that happen to produce HTML.

Listing 12-9 *getradio*: Determining which buttons the visitor pressed

```
#!/usr/bin/perl
use CGI::Request;
$request = GetRequest();
$happiness = $request->param('mood');
print PrintHeader();
print "<TITLE> Feedback </TITLE>";
print "<H1> Feedback </H1>";
print "<HR>";
if ($happiness eq `very happy`) {
print "Glad you enjoyed it!<BR>";
} elsif ($happiness eq `happy`) {
print "OK. If you can think of a way to improve our site, let
us know.<BR>";
} elsif ($happiness eq `satisfied`) {
print "Hmmm. What didn't you like about our site?<BR>";
} else {
print "Hey, it's not like we ASKED you to visit our site,
pal.<BR>";
}
```

The CGI::Request module provides a full-fledged object-oriented interface to QUERY_STRING data. But *getradio* uses the simpler GetRequest() function instead, which parses the QUERY_STRING data for you. That's fairly handy; because the types of information that might be created in a <FORM> vary so much, the QUERY_STRING format is somewhat arcane. CGI::Request does the dirty work for you.

In particular, you can access the radio button selection (which has the attribute **NAME="mood"**) with `$request->param("mood")`.

Checkboxes

Another <FORM> is the checkbox type, which resembles radio buttons except that you can select several at the same time. *page7.html*, shown in Listing 12-10, is similar to *page6.html*, but uses checkboxes instead of radio buttons.

Listing 12-10 *page7.html*: Using HTML forms to enable nonexclusive choices

```
<TITLE> Guest Book III </TITLE>
<H1> Visitor Feedback Part II </H1>
Let our marketing droids know what junk mail to send you:
<HR>
<FORM ACTION="http://your.web.server/cgi-bin/getcheck">
I use the following computers:
<P>
```

```

<INPUT TYPE="checkbox" NAME="computers" VALUE="UNIX
workstation"> UNIX workstation
<P>
<INPUT TYPE="checkbox" NAME="computers" VALUE="IBM clone" >
IBM-compatible machine running DOS or Windows
<P>
<INPUT TYPE="checkbox" NAME="computers" VALUE="Macintosh"
SELECTED> Macintosh (or compatible)
<P>
<INPUT TYPE="checkbox" NAME="computers" VALUE="Appliance" >
CPU-enhanced coffee pot (or dishwasher)
<P>
<INPUT TYPE="submit"> <INPUT TYPE="reset">
<HR>
</FORM>

```

page7.html visitors see the Macintosh checkbox SELECTed; they can turn that off and others on if they wish. In Figure 12-12, the UNIX and Macintosh buttons have been selected by the visitor.

getcheck (Listing 12-11) is similar to *getradio*, but because visitors can select more than one checkbox, a string won't do: `$request->param("computers")` returns an array of `<FORM>` data.

Visitor Feedback Part II

Let our marketing droids know what junk mail to send you:

I use the following computers:

☒ UNIX workstation

☐ IBM-compatible machine running DOS or Windows

☒ Macintosh (or compatible)

☐ CPU-enhanced coffee pot (or dishwasher)

Figure 12-12

Listing 12-11 *getcheck*: Determining which checkboxes the user selected

```
#!/usr/bin/perl
use CGI::Request;
print "Content-type: text/html\r\n\r\n";
print "<TITLE> Feedback </TITLE>";
print "<H1> Feedback </H1>";
print "<HR>";
$request = new CGI::Request;
@list_of_computers = $request->param('computers');
```

Each array element in @list_of_computers is a VALUE from the <INPUT...> tag; in the preceding example, the first array element would be *UNIX Workstation* and the second would be *Macintosh*.

<SELECT>

The <SELECT> form lets visitors choose single options from a scrollable list. Every <SELECT>...</SELECT> contains a list of text <OPTION>s framed inside a box. If one of the selections is tagged <OPTION SELECTED> instead of <OPTION>, that selection becomes the default choice and is highlighted, as shown in Listing 12-12.

Listing 12-12 *page8.html*: Letting the user select among several options

```
<TITLE> Guest Book III </TITLE>
<H1> Mad Libs </H1>
<HR>
<FORM ACTION="http://your.web.server/cgi-bin/getsell">
Choose one from Column A:
<P>
<SELECT NAME="Column A" MULTIPLE SIZE = 5>
<OPTION SELECTED> Bulbous
<OPTION> Zealous
<OPTION> Dangerous
<OPTION> Marvelous
<OPTION> Scabrous
</SELECT>
<P> and one from Column B:
<SELECT NAME="Column B" SIZE = 4>
<OPTION> Argument
<OPTION> Tournament
<OPTION> Testament
<OPTION SELECTED> Filament
<OPTION> Firmament
</SELECT>
```

```

<P>
Ready?
<P>
<INPUT TYPE="submit" VALUE="Yup"> <INPUT TYPE="reset"
VALUE="Nope">
<HR>
</FORM>

```

Figure 12-13 shows Page 8.

Mad Libs

Choose one from 'Column A':

- Bulbous
- Zealous
- Dangerous
- Marvelous
- Scabrous

and one from 'Column B':

- Argument
- Tournament
- Testament
- Filament

Ready?

Figure 12-13

Page 8

Now the *getsel* program (Listing 12-13) can access the visitor's selections. As you'd expect, the visitor's selections are made available via `$req->param('Column A')` and `$req->param('Column B')`.

Listing 12-13 *getsel*: Determining which options the user selected

```

#!/usr/bin/perl
use CGI::Request;
print "Content-type: text/html\r\n\r\n";
print "<TITLE> Feedback </TITLE>";
print "<H1> Feedback </H1>";
print "<HR>";
$req = new CGI::Request;
$adjective = $req->param('Column A');

```

```
$noun = $req->param( 'Column B' );
```

<TEXTAREA>

If you want visitors to type text on your Web page, you can supply a typeable <TEXTAREA> box, demonstrated in Listing 12-14.

Listing 12-14 *page9.html*: Using <TEXTAREA> to create a text-input area

```
<TITLE> Guest Book III </TITLE>
<H1> Visitor Feedback Part III </H1>
<HR>
Regale us with your wit:
<FORM ACTION="http://your.web.server/cgi-bin/gettext">
<TEXTAREA NAME="wit" COLS=60 ROWS=6>This is "default" text.
Press I Mean It! to submit.</TEXTAREA>
<P>
Your e-mail address: <INPUT TYPE="text" NAME="e-mail"
VALUE="perluser@waite.com" SIZE=30>
<P>
Your password for that account (so we can delete your files):
<INPUT TYPE="password" NAME="password" VALUE="BOBOLink"
SIZE=8>
<P>
Press either <INPUT TYPE="submit" VALUE="I Mean It!"> or
<INPUT TYPE="reset" VALUE="Just Kidding">
<HR>
</FORM>
```

page9.html creates a window sixty characters long and six lines high in which users can type, as shown in Figure 12-14.

Visitor Feedback Part III

Regale us with your wit:

This is "default" text.

Your e-mail address:

Press either or

Figure 12-14

Letting the user enter text

gettext, shown in Listing 12-15, plops the contents of each text box into a scalar.

Listing 12-15 *gettext*: Grabbing text from the user

```
#!/usr/bin/perl
use CGI::Request;
print "Content-type: text/html\r\n\r\n";
print "<TITLE> Feedback </TITLE>";
print "<H1> Feedback </H1>";
print "<HR>";
$req = new CGI::Request;
$email = $req->param('e-mail');
$text = $req->param('wit');
$password = $req->param('password');
```

You can mix and match any of the FORMs in this lesson, of course. You can also weave in audio, images, and video; that's the subject of the next lesson.

Chapter 12, Perl 5 Interactive Course

LESSON 4

IMAGES, AUDIO, AND VIDEO

Websurfing without pictures is like reading a text-only transcript of the Sunday funnies. Pictures (and, to a lesser extent, audio and video) shape the tone of Web pages in a way that text can't. As often as not, the graphical style of a Web page separates the memorable pages from unmemorable ones. This lesson briefly addresses how you can add a splash of multimedia to your Web pages (with or without Perl).

Multimedia has always been a Tower of Babel. There are hundreds of file formats for still images, audio, and video--no single browser can display all of them, and most can display only a few. Browsers can pass the buck to helper applications, such as graphics programs that can display TIFF files, sound players that can play *.aiff* sound files, or MPEG players that play MPEG movies. Of course, your browser won't necessarily know what applications you have on your computer; you have to tell it. Check the documentation for your particular browser.

Images

In a few cases, browsers won't need to invoke helper applications because they'll be able to handle the job themselves. That's generally true only for images, particularly GIF images, X bitmaps, and X pixmaps. Those images can be inlined (mixed with text), rather than requiring a helper application (a nonWeb-related program, such as xv). Inlined images are displayed with the tag:

```
<IMG SRC="my_pics/me.gif" ALT="[Picture of me]">
```

The SRC attribute contains a filename identifying where the image can be found. It's mandatory, unlike the other three IMG attributes:

- **ALIGN="bottom"** (or "middle" or "top") specifies where the image is placed relative to the adjoining text. (You can't get text to flow around the image, so don't try.)
- **ALT="some text"** specifies a text alternative that nongraphical browsers can display instead of the image.
- **ISMAP** designates the image as an active picture: Clicking in different areas causes different actions. You then need to anchor the picture to an imagemap program, which you can get from
<ftp://mox.perl.com/pub/perl/ext/CGI/tools/imagemap> or
<http://hoohoo.ncsa.uiuc.edu/docs/tutorials/imagemapping.html>.

If you don't want to construct your own images, you can use one of the stock gifs provided by the repository of GIF icons at <http://www-pcd.stanford.edu/gifs/>. Want a key, an arrow, or a dollar sign? They're all there.

If you want to generate your GIFs automatically, two Perl scripts can help you create a .gif file, which you can then include with .

- Jeffrey Friedl's *gif.pl* (<http://www.wg.omron.co.jp/~jfriedl/perl/gif.pl>), which supplies three functions: `gif::start()` to create a GIF header, `gif::bits()` to append data to the GIF, and `gif::end()` to finish writing the GIF data.
- Lincoln Stein's *GD.pm* (<http://www.genome.wi.mit.edu/ftp/pub/software/WWW/GD.pm.tar.Z>), which requires the C *gd* library, available at <http://www.boutell.com/gd/>. *GD.pm* defines three classes--`GD::Image`, `GD::Font`, and `GD::Polygon`--and contains a host of functions that let you create text, arcs, rectangles, fills, and more. *GD.pm* also lets you create interlaced images.

Audio, Video, and Noninlined Images

Web pages themselves don't contain audio or video, only anchors to audio and video. It's up to your browser to invoke the appropriate helper application. When you click on an audio or video clip, your browser launches an application (such as *xv* to show JPEG or TIFF images, *mpeg_play* to play MPEG movies, RealAudio to play sound files).

When a browser requests a URL, the server places an appropriate MIME (Multipurpose Internet Mail Extensions) header at the beginning of the data sent to the client. In a CGI script, you have to do that manually. That's why you include this line at the beginning of your script:

```
print "Content-type: text/html\r\n\r\n";
```

This tells the browser that it's receiving data of type `text` and subtype *html*. MIME headers are used in both directions: When the browser sends POST data, it prepends `Content-type: application/x-www-form-urlencoded`.

If you want to send *gif.pl* output without ever creating a GIF file on disk, include this line instead:

```
print "Content-type: image/gif\r\n\r\n";
```

Other type/subtype combinations that might come in handy include the following:

- `image/jpeg`
- `image/tiff`
- `image/rgb`
- `audio/basic`
- `audio/x-aiff`
- `audio/x-wav`
- `text/richtext`
- `video/mpeg`

- **video/quicktime**

There are many more type/subtype pairs. For more information about MIME, consult

- **HTML:** <http://www.oac.uci.edu/indiv/ehood/MIME/MIME.html>
- **Postscript:** <http://ds.internic.net/rfc/rfc1521.ps>
- **Plain text:** <http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/rfc1521.txt>

Chapter 12, Perl 5 Interactive Course

LESSON 5

PERL/WEB RESOURCES

There are plenty of nifty utilities for building a great Web site with Perl. Of course, all those utilities are on the Web, and new utilities are developed every day. No book can be completely current because everything changes so quickly. Knowledge about how to find new resources is much more useful. For that, you need a meta-resource: a listing of resources that is more current than any book. Appendix B contains a listing of meta-resources, as well as pointers to Web sites, FTP sites, and newsgroups for both Perl and the Web. There's an online meta-resource as well, specifically tailored to Perl and the Web: perlWWW.

CGI.pm and the CGI Module Set

The *CGI.pm* module (<http://www-genome.wi.mit.edu/ftp/pub/software/WWW/cgi-docs.html>), written by Lincoln Stein, is the most popular Perl utility of all. It's incredibly useful--not only does *CGI.pm* allow you to parse CGI requests, but it includes methods for automatically generating HTML for FORMs, for example:

```
use CGI;
$query = new CGI;
print "<P>What's your favorite color? ",
$query->popup_menu(...)
```

As of Perl version 5.004, *CGI.pm* is finally bundled with the standard Perl distribution. For further information, see Lincoln's extensive online documentation, or his regular column in *The Perl Journal*.

The CGI Module Set

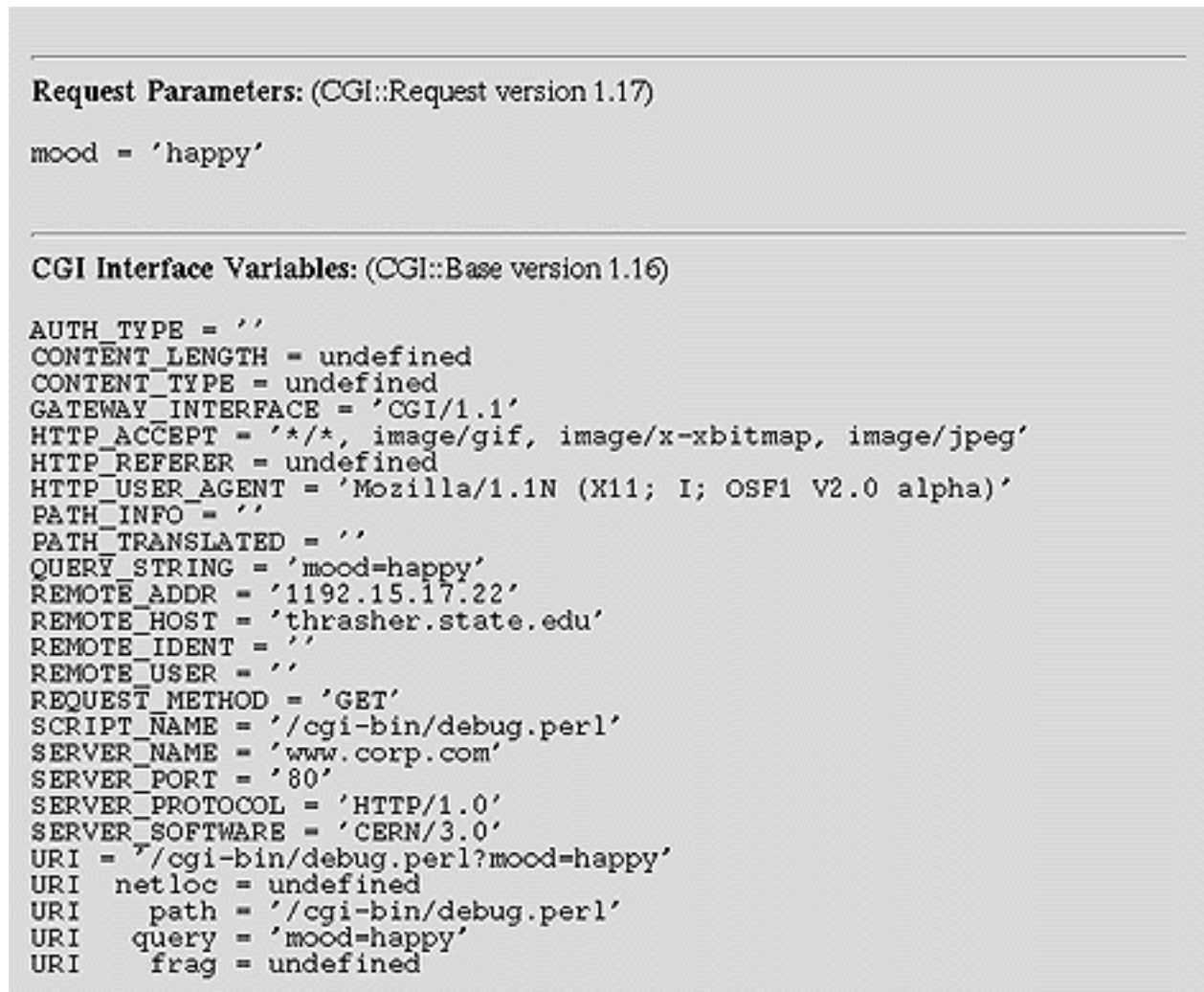
The CGI module set is Lincoln's comprehensive alternative to *CGI.pm*. It contains six modules: Request, Base, BasePlus, MiniSvr, Form, and Carp.

CGI::Request implements a Request object, which uses a CGI::Base object to parse \$ENV{QUERY_STRING}; it's what was used in Lesson 3. This module also contains the helpful as_string() method, which comes in handy for debugging CGI scripts--it prints the raw and parsed CGI information. For instance, suppose the radio button example in Lesson 3 were to link to the CGI script shown in Listing 12-16.

Listing 12-16 *cgidebug*: Using CGI::Request to debug a CGI script

```
#!/usr/bin/perl -w
use CGI::Request;
print "Content-Type: text/html\r\n\r\n";
print "<TITLE> as_string() demo </TITLE>";
$request = new CGI::Request;
print $request->as_string;
```

The visitor would see the information that the browser provided to the server, as shown in Figure 12-15.



```
Request Parameters: (CGI::Request version 1.17)

mood = 'happy'

CGI Interface Variables: (CGI::Base version 1.16)

AUTH_TYPE = ''
CONTENT_LENGTH = undefined
CONTENT_TYPE = undefined
GATEWAY_INTERFACE = 'CGI/1.1'
HTTP_ACCEPT = '*/*, image/gif, image/x-xbitmap, image/jpeg'
HTTP_REFERER = undefined
HTTP_USER_AGENT = 'Mozilla/1.1N (X11; I; OSF1 V2.0 alpha)'
PATH_INFO = ''
PATH_TRANSLATED = ''
QUERY_STRING = 'mood=happy'
REMOTE_ADDR = '1192.15.17.22'
REMOTE_HOST = 'thrasher.state.edu'
REMOTE_IDENT = ''
REMOTE_USER = ''
REQUEST_METHOD = 'GET'
SCRIPT_NAME = '/cgi-bin/debug.perl'
SERVER_NAME = 'www.corp.com'
SERVER_PORT = '80'
SERVER_PROTOCOL = 'HTTP/1.0'
SERVER_SOFTWARE = 'CERN/3.0'
URI = '/cgi-bin/debug.perl?mood=happy'
URI    netloc = undefined
URI    path = '/cgi-bin/debug.perl'
URI    query = 'mood=happy'
URI    frag = undefined
```

Figure 12-15

The results of the CGI::Request as_string method

CGI::Base implements a CGI object. These objects export (but do not parse) the environment variables available to your CGI scripts. Higher level processing of the exported data is done by the CGI::Request module. CGI::BasePlus behaves the same, but is tailored for MIME multipart/form data.

CGI::MiniSvr lets you create (well, fork(), really) a mini HTTP server for maintaining client information throughout a "session" of Web page accesses.

CGI::Form (it's not part of the module list but is included with the CGI package) provides an interface for creating FORMs and parsing their contents.

CGI::Carp is a substitute for Perl's Carp module that contains routines for writing more informative error messages to the HTTP error log.

The LWP Bundle (libwww-perl)

There are several (partially redundant) efforts to help Perl programmers manipulate URLs, the CGI, and other features of the WWW. The granddaddy of them all is Steven Brenner's *cgi-lib.pl* (available from the perlWWW page), which contains a ReadParse() subroutine that translates GET and POST data into Perl data structures. The CGI::Request module used in Lesson 3 is loosely based on *cgi-lib.pl*, as is *CGI.pm*.

CGI::Request is intended for use by novices. It's good enough for most purposes. But the definitive WWW library is LWP (libwww-perl), a set of Perl 5 libraries available from the CPAN.

Technical discussion (for developers only!) of these modules takes place on a couple of mailing lists; if you're interested in helping out, send mail to libwww-perl-request@ics.uci.edu or CGI-perl-request@webstorm.com. The rest of this chapter briefly describes each module set in the LWP.



To make full use of LWP, you'll need to install three additional packages from the CPAN: IO, MD5, and the libnet bundle.

The URI Module Set

A universal resource identifier (URI) is the name of an object on the Net. A URL is a particular type of URI containing an address that describes how the object can be retrieved via network protocols. The distinction between URIs and URLs is often blurred.

There's a URI module set, which contains two modules: URI::URL. This defines a Perl URL object. These objects should be thought of as mere strings; they don't "know" about the Internet. You can store and access the individual parts of a URL (such as the different chunks of the string *http://www.waite.com:8001/cgi-bin/program*), but to use the URLs over the Internet, you'll need the HTTP module set.

The URI::Escape module provides two functions, uri_escape() and uri_unescape(), that let you use whatever characters you wish without having to worry about the official URI specification (RFC 1738; refer to the RFC entry in Appendix B).

The HTTP Module Set

This module set helps your programs speak the HTTP protocol more fluently. For instance, the HTTP::Headers module provides an object-oriented interface to HTTP message headers; the HTTP::Status module contains a list of all the status codes (for example, code 404 is "Not Found") that a Web server might send to browsers.

If you want to write your own Web client or server, look here first.

The HTML Module Set

The HTML modules provide various mechanisms for manipulating HTML. For instance, the `HTML::QuickCheck` module performs fast validation of HTML, ensuring that your Web pages don't have any obvious bloopers, and the `HTML::Base` module provides an object-oriented mechanism for Web page construction.

The WWW Module Set

The WWW module set contains only one module at present: `WWW::RobotRules`, which parses the `robots.txt` file specified by the Robot Exclusion Standard.

The MIME Module Set

The MIME module set contains two modules: `MIME::Base64` and `MIME::QuotedPrint`. Both help you encode and decode MIME messages--helpful for e-mail as well as Web pages.

The LWP Module Set

This module set provides an interface to the `libwww-perl` library. It contains easy-to-use functions such as

```
mirror($url, $file)
```

which replaces a local `$file` if there's a more recent version at `$url`.

perlWWW

The `perlWWW` page, located at <http://www.oac.uci.edu/indiv/ehood/perlWWW>, is an index of Perl utilities for the WWW maintained by Earl Hood. The page is divided into eight categories; taken together, they give a pretty good idea of what you can do with the Web in Perl. All the links within these categories lead you to Perl programs, and the `perlWWW` page is likely to be updated more frequently than this book. The categories are as follows:

- **Applications:** Web navigation tools and Plexus, a Web server (described in Lesson 6).
- **Browsers:** Perl programs used by WWW browsers, such as programs for manipulating hot lists (No one has written a browser in Perl yet, unfortunately. But if someone does, it'll probably use `Perl/Tk`, described in Chapter 14, Lesson 1.)
- **Converters:** Perl programs that convert between various formats (PostScript, man, plain text) to HTML and back.
- **Development:** Perl libraries for developing WWW software--includes a link to `libWWW-perl`, discussed below.
- **Graphics:** Perl programs that manipulate GIF images (the image format of choice on the Web).
- **HTML:** Programs that process HTML documents, including the popular `htmlchek`, which checks the syntax of HTML files.

- **Miscellaneous:** Includes WebLog, a guestbook program.
- **Utilities:** Perl utilities that help Webmasters, including Page-stats, which reports how many times a page was accessed.

Chapter 12, Perl 5 Interactive Course

LESSON 6

RUNNING A WWW SERVER

You want to run CGI programs, but there's no Web server running at your site. No problem: Just FTP a server and run it. It's not as hard as it sounds. You don't even need superuser access; all you need is some disk space and an Internet connection.

There are dozens of Web servers available; rather than attempt to describe them all, this book will restrict itself to three servers, two of which are written in Perl: Plexus, a big, powerful, popular Web server, and `phttpd`, a small and simple Web server. Whether your Web server is written in Perl has no bearing on what CGI scripts may be called; you can execute Perl scripts from a non-Perl Web server (e.g., CERN `httpd`) and you can execute non-Perl programs (such as shell scripts or C programs) from Perl Web servers. The third Web server is Apache, a production-quality, free Web server of special interest to Perl programmers.

If you set up a Web site, you should make sure that a *webmaster* account exists. Many visitors will send e-mail to that account to report problems or make suggestions. Also, consider setting up an alternative hostname for your Web site beginning with *www*, for example, `www.waite.com`, `www.state.edu`, and so on. That way you can move your Web server from computer to computer without visitors noticing.

Another choice is which port your Web server listens to. Most Web sites run on port 80; many UNIX systems restrict low-numbered ports (less than 1024) to superusers. Plexus and `phttpd` both let you choose a different port: By default, nonsuperusers who run Plexus use port 8001, and nonsuperusers who run `phttpd` use port 4040. (Ports are covered in the first lesson of the next chapter.)

Web servers like to have their own top-level directory so they can refuse to serve files outside of that directory tree. As yet, there is no convention for what that directory should be named; both Netscape and Plexus servers suggest `/usr/local/www`.

Plexus

Plexus (`ftp://ftp.earth.com/plexus`), written by Tony Sanders, is a popular Web server that has plenty of nifty features (some that you won't find on any other server) while being easy to install and maintain. Among its features are

- **Load limiting:** You can specify the maximum number of visitors that can be connected at a time.
- **Authentication:** You can drop in privacy enhanced mail (PEM) modules.
- **Access control:** You can filter out requests from particular hosts or domains to particular directories or using certain methods.

- **File locking:** This is useful if two WWW visitors try to modify the same file at the same time.
- **Plexus searches directory trees using Perl regexes.**
- **Plexus allows multiple servers on the same port.**
- **Plexus allows simultaneous requests on different ports.**

The Plexus server handles ISMAP tags, letting you create multiple clickable areas in an image. Plexus provides a native gateway interface in addition to CGI: It can run your Perl CGI scripts without creating a new process. That means that your CGI scripts will start up faster and won't use as much memory.

Plexus comes with four sample gateway programs, each providing a Web interface to a particular UNIX program. Each is just a few lines of Perl code:

- **Finger**
- **Calendar**
- **Date and Time**
- **Fortune**

Plexus gateway programs are easy to write. Listing 12-17 shows one that uses `rand()` to print a random number between 0 and some number typed by the user.

Listing 12-17 *randpage*: A native gateway program

```
# Note - no /usr/bin/perl!
sub gimme_rand {
my($num) = $_[0] if $_[0]; # The upper bound for rand()
MIME_header('ok', 'text/html'); # Print the MIME header
(required!)
print "Enter the upper bound for rand() in the box below:\n";
print "<ISINDEX>\n<TITLE>Random Number
Page</TITLE>\n\n<P>\n";
srand(time ^ ($$ + ($$<<15)));
$num =~ s/\D//g; # Remove non-digits
if (defined($num)) {
print "The result from your last request:";
print "<H2>rand(", $num, ") = ", rand $num,
"</H2>\n</BODY>\n";
}
}
1; # Loaded files should always be true
```

Here are the two lines that need to be added to `server/local.conf` so that Plexus will load *rand* when it starts up, enabling URLs of the form `http://www.waite.com:8001/random`. (Plexus runs on port 8001 by

default; you can choose a different port with the -p option.)

```
load rand
map random rand gimme_rand($query)
```

phttpd

phttpd (<http://eewww.eng.ohio-state.edu/~pereira/software/phttpd>) is a lean-and-mean Web server written by Pratap Pereira. Pereira notes that phttpd was never meant to compete with Plexus: Think of it as a no-frills template for building your own Web server. It's quick to set up and easy to maintain, making it a good choice for single-purpose tasks (such as reading your own mail), for learning about how a server can be built, or for setting up a quasi-secret Web server on an arbitrary port.

Installation is a snap: Use *edit setup.pl* to choose a server directory and port number, and then run phttpd. By default, phttpd runs on port 4040.

One caveat: phttpd does not handle directory structures, nor does it understand POST queries.

Apache

Unlike Plexus and phttpd, Apache (<http://www.apache.org>) is a production-quality Web server in use by more than 400,000 Internet sites. It is fast, well maintained, easy to install, and free.

Worth special mention is the *mod_perl* plugin for Apache. In Chapter 9, you learned about `fork()` and how it creates a new process. Normally, when you execute a CGI script from your Web server, a new process is forked every time that script is executed. That takes time and memory. Doug MacEachern's solution was to embed the Perl interpreter *inside* Apache, so that no new processes have to be created. The result: much faster execution of all your CGI scripts.

You can download Doug's *mod_perl* plugin from the CPAN, in the Apache module category. Once you've installed *mod_perl*, you can use various Apache/Perl modules; Doug maintains a list on his Apache/Perl page at <http://www.osf.org/~dougma/apache/>.

Chapter 12, Perl 5 Interactive Course

LESSON 7

ADVANCED WEBBERY

You've got your Web server up and running and you've still got a few questions:

- "How safe is my Web server?" (Security)
- "Can a Web server make a browser do anything besides display a Web page?" (dynamic documents and cookies)
- "How can I retrieve Web pages from the command line?" (webget and url_get)
- "How can I verify that my Web page is bug-free?" (htmlchek, linkcheck, verify_links)
- "How can I make mail archives available via the Web?" (Mail)

Security

First, you need to clarify what sort of security you're talking about. In addition to ensuring that network users don't gain access to the files on your server (consider the methods discussed in the last chapter for preventing use of variables tainted by user input), three concerns are raised by communication and commerce over the Internet:

- **Authentication:** thwarting impostors
- **Encryption:** thwarting eavesdroppers
- **Protection:** thwarting vandals

A server can perform authentication by itself by prompting for a password, but both server and browser must cooperate to achieve encryption and protection.

The notion of secure HTTP has spawned several schemes and discussion groups; you can find a list at http://www.yahoo.com/Computers/World_Wide_Web/HTTP/Security/. Some efforts are noncommercial, such as the Internet Engineering Task Force (IETF) HTTP Security Working Group (<http://www-ns.rutgers.edu/www-security/https-wg.html>); other efforts are proprietary and will work only with certain servers or browsers. One exception is Netscape's SSL protocol.

Netscape, SSL, and HTTPS

Netscape Communications (whose security handbook at <http://home.netscape.com> discusses its approach to the three aims highlighted above) provides a server that uses its nonproprietary secure sockets layer (SSL) protocol to verify users, encrypt HTTP communications, and prevent interception of HTTP communications. URLs for sites secured with SSL begin with `https://` instead of `http://`.



Don't send credit card numbers over the Net unless you're using a secure version of HTTP!

Firewalls

Some sites have a firewall computer that protects their network from the rest of the Internet. Unfortunately, like real walls, firewalls don't just prevent outsiders from getting in: They also hinder insiders from getting out. Local users behind a firewall won't be able to browse the Net unless their firewall is configured as a proxy server that follows URLs on a browser's behalf.

Dynamic Documents

HTTP is designed for sporadic, user-driven accesses: The Web surfer points the browser at page x, then jumps to page y, then downloads document z--the client chooses what to do and when. But what if a server wants to "push" data to the client without waiting to be asked? What if you want the browser's computer to run a program? Enter three mechanisms for creating dynamic documents: Sun's Java language, Felix Gallo's Penguin extension for Perl 5, and the much less powerful Netscape mechanisms *server push* and *client pull*.

Java™

Java™ (<http://java.sun.com>) lets a Web page run programs on a browser's computer. These programs, called *applets*, are written in a language called Java and are automatically downloaded from the server to the browser's computer. The idea is a good one. Unfortunately, the Java source code is proprietary; neither Java itself nor Java binaries are public domain--licensing and nondisclosure agreements are required, and sometimes money as well. Several security problems have been discovered in Java, too. But the biggest problem with Java is that it makes you program in Java, which resembles an interpreted C++ sans pointers. Perl is more fun. If you want to ship secure Perl programs around the Internet, you should be using Penguin.

Penguin

Felix Gallo's Penguin is a completely free, public domain Perl 5 module that lets you

- **Send encrypted and digitally signed Perl programs to remote machines.**
- **Receive Perl programs, and, depending on the sender, automatically execute them in an appropriate compartment secured with the Safe module (described in Chapter 11, Lesson 2).**

Penguin's security model is simple. Unlike Java, which requires a separate pass through the program to

verify that maybe possibly it won't hurt your computer too badly, Penguin just uses Safe to outlaw certain operations. Which operations? Your choice. And you can have different levels of security for different users, so that your computer can grant friends more latitude than strangers. For instance, you might want neither friends nor strangers to erase your files, but you wouldn't mind if friends ran computationally intensive programs when you're not around. With Penguin, it's easy to make distinctions between friends and strangers, and between threats to your disk and threats to your system load.

The Penguin distribution (available on the CPAN) includes a server (called *penguind*) and a client (called *client*). The server, which you can run from the command line, is what executes the Perl programs sent to your computer. It consults a customizable "rightsfile" to determine whom to trust and how much. The client is what you use to send your program to another computer for execution: It wraps the program in your digital signature so that the remote penguind can verify your identity.

Server Push and Client Pull

You can't animate Web pages because HTTP is a connectionless protocol. Right? Well, almost. Check out http://home.netscape.com/assist/net_sites/pushpull.html (or thereabouts) for information about server push and client pull, which lets a Web server coerce a browser into reloading the page.

- **Server push allows the Web server to keep the HTTP connection open indefinitely. The server sends data to the browser whenever it wants.**
- **Client pull allows the Web server to send two types of commands to the browser:**

"Reload this page in n seconds"

and

"Load this other page in n seconds."

Client pull is easy. Just create a document that looks like Listing 12-18.

Listing 12-18 *pulltest.html*: Using client pull to make a client reload an image every 2 seconds

```
<META HTTP-EQUIV="Refresh" CONTENT=2>
<TITLE> Test of Client Pull </TITLE>
<H1> Let me repeat myself. </H1>
```

pulltest.html reloads itself every 2 seconds for the rest of eternity. (You could set a maximum number of reloads by embedding *pulltest.html* in a Perl CGI script.) Your script can make the browser load another document, as *question.html* does in Listing 12-19.

Listing 12-19 *question.html*: Using client pull to load another URL after 10 seconds

```
<META HTTP-EQUIV="Refresh" CONTENT="10;
URL=http://your.web.server/answer.html">
<TITLE> Test 2 of Client Pull </TITLE>
```

<H1> Can you name an animal with naturally blue blood? </H1>

Server push is a little more complicated because you have to speak MIME. The server has to send the *Content-type: text/html\r\n\r\n* line over at the beginning of every page and has to prepare the browser by beginning the connection with a string that looks like this:

Content-type:

multipart/x-mixed-replace;boundary=---HeyNewPage---

where *HeyNewPage* is a string that you'll use to mark new pages.

pushtest, shown in Listing 12-20, pushes data to the browser whenever it finds a new prime number.

Listing 12-20 *pushtest*: Using server push to force new data to the browser

```
#!/usr/bin/perl
use integer;
@primes = (2, 3);
print "HTTP/1.0 200\n";
print "Content-type:
multipart/x-mixed-replace;boundary=---HeyNewPage---\r\n\r\n";
print "---HeyNewPage---\r\n";
while (1) {
    print "Content-type:text/html\r\n\r\n";
    print "<H2> The next prime number: </H2>";
    print "<H2> $primes[$#primes] </H2>";
    print ' ' x 64;
    print "---HeyNewPage---\n";
    find_prime;
}
sub find_prime {
    my $num = $primes[$#primes] + 2;
    my $i;
    for ( ; ; $num += 2) {
        last unless $num % $i;
        if ($i > sqrt($num)) {
            push (@primes, $num);
            return;
        }
    }
}
```

Cookies

Suppose you're browsing through an online Web store. As you click around from page to page, you place items you want to buy in a virtual shopping basket. Now, HTTP is a connectionless protocol: When you visit a page, the Web server has no knowledge of where you've been previously.

Or does it? If your Web server and browser both support *cookies*, then your browser can forward a few bits of information to the server. With a Web store, that would include what's in your shopping basket.

A cookie is just a name-value pair, just like the named parameters in the CGI query string. Lincoln Stein's *CGI.pm* module contains a `cookie()` method that you can include in your server's CGI scripts. When you activate the method, it tells the browser to remember the values of whatever attributes you choose. The next time the browser visits your site, the cookies will be served up, and your server can use them to personalize its behavior to that particular visitor.

See Lincoln Stein's article in issue 3 of *The Perl Journal*, or download his cookie code from <http://www.tpj.com>.

weblint, htmlchek, linkcheck, and verify_links

Nothing is more frustrating to a Web surfer than seeing one's browser chug across the network (and often across oceans) to reach a faraway Web site only to meet defeat 45 seconds later: *Can't connect to host or Invalid URL* or *Document contains no data*. Some of those errors can be detected or prevented with the LWP library or any of these four utilities.

weblint

weblint (<http://www.cre.canon.co.uk/~neilb/weblint.html>) is a Perl script that finds errors (and non-errors worth griping about) in HTML files: mismatched tags, unknown attributes, missing ALTs in tags, and much more.

htmlchek

Henry Churchyard's *htmlchek* (<ftp://ftp.cs.buffalo.edu/pub/htmlchek/>) does a good job of identifying errors (both stylistic and syntactic) in HTML files. It's available as a Perl program (as well as sh and awk).

linkcheck

David Sibley's Perl program *linkcheck* (<ftp://ftp.math.psu.edu/pub/sibley>) checks the URLs (HTTP, FTP, and gopher only) in a Web page to make sure they exist.

verify_links

If you want to bulletproof your Web site, try Enterprise Information Technologies' *verify_links* (<ftp://ftp.eit.com/pub/eit/wsk>), part of their Webmaster's Starter Kit. It verifies a URL, even going to the trouble of pressing all the buttons for you, entering text, and so on.

Always try out your Perl CGI scripts from the command line before you install them.

Retrieving Web Documents

Several people have written programs that follow URLs from the command line. The *lwp-rget* and *lwp-download* programs in LWP do this, as does *webget*.

In addition to *gif.pl* (described in Lesson 4), Jeffrey Friedl also wrote *webget* (<http://www.wg.omron.co.jp/~jfriedl/perl/webget>), which fetches an HTTP or FTP document from the command line. *webget* is the successor to Friedl's widely used *httpget*.

```
% webget www.netscape.com
<TITLE>Welcome to Netscape</TITLE>
<CENTER>
<A HREF="/misc/home.map">
...
```

webget has an especially clever feature: the *-update* option, which grabs the latest version of *webget* and reports whether it's newer than your version.

Mail

Two programs convert mail archives to a Web-palatable format: *MHonArc* and *Hypermail*.

MHonArc

Earl Hood's Perl *MHonArc* (<http://www.oac.uci.edu/indiv/ehood/mhonarc.html>), the successor to *mail2html*, turns mail (or USENET news) into searchable HTML pages, complete with links to authors and to any URLs mentioned in the text. It follows subject threads and understands MIME (so you can use it to read MIME messages via your Web browser).

Hypermail

Enterprise Information Technologies' Hypermail (<http://www.eit.com/software/hypermail>, written by Tom Gruber in Lisp; rewritten by Kevin Hughes in C) takes a file of mail messages in UNIX mailbox format (one big file) and breaks it into HTML files, one file per message, including links to any other messages in the archive, as well as e-mail addresses and URLs mentioned in the message.

Chapter 12, Perl 5 Interactive Course

LESSON 8

NON-WEB INTERNET SERVICES

Many people confuse the Web with the Internet. The Web is a collection of servers speaking a particular language (HTTP) on the Internet. (Every computer on the Web is also on the Internet, but not all computers on the Internet are on the Web.) In this lesson, you'll learn about Perl utilities for three non-Web network services: FTP, IRC, and WAIS. (Utilities for connecting to any network service, *comm.pl* and *chat2.pl*, are discussed in the next chapter.)

FTP

The file transfer protocol (FTP) is used to download files from one computer to another. FTP is spoken by nearly every browser; if you want your Perl programs to do the same, get Graham Barr's `Net::FTP` module, available as part of the `libnet` bundle on the CPAN. Locations such as `ftp://ftp.metronet.com/pub/perl/scripts/ftpstuff/` have lots of other Perl/FTP utilities. For example, you'll find *ftpget* and *ftpmget*, which log into a remote host and retrieve a particular file (or, with *ftpmget*, multiple files).

Listing 12-21 features `Net::FTP`, using it to download files with FTP.

Listing 12-21 *getirc*: Using `Net::FTP` to download files using FTP

```
#!/usr/bin/perl
use Net::FTP
$ftp = Net::FTP->new("cs-pub.bu.edu");
$ftp->login('anonymous') or die "Can't connect to
cs-pub.bu.edu!";
$ftp->cwd('irc/clients');
$ftp->get('sirc-2.101.tar.gz');
$ftp->close;
```

getirc opens an FTP connection to `cs-pub.bu.edu`, changes the directory to `irc/clients`, retrieves the file *sirc-2.101.tar.gz*, and then closes the connection.

IRC

Internet relay chat (IRC) is a cross between USENET and a telephone chat line. An IRC server passes messages between individual users or from one user to many. Users connect to IRC servers with IRC clients. There's a Perl client at `ftp://cs-pub.bu.edu/irc/clients/sirc-2.101.tar.gz`, but if you ran the program shown above, you should already have it by now.

General information about IRC can be found at <http://www.kei.com/irc.html>.

WAIS

Wide area information servers (WAIS) index large text databases and serve them to Internet users, fielding complex queries (including queries expressed in simple English) and retrieving matching documents.

The Wais module

Ulrich Pfeifer's Wais module (available from the CPAN) provides access to the freeWAIS-sf libraries (available from <ftp://ftp.germany.eu.net/pub/infosystems/wais/Unido-LS6/freeWAIS-sf/>). It's a client that you can use to search for and retrieve documents from any WAIS server.

wais.pl

wais.pl (ftp://ftp.ncsa.uiuc.edu/Web/httpd/Unix/ncsa_httpd/cgi/wais.tar.Z), written by Tony Sanders and Rob McCool, is a CGI WAIS gateway; use it if you want Web visitors to be able to access your database using the WAIS protocol. It indexes files on your computer with the help of a WAIS server such as freeWais (<ftp://sunsite.unc.edu/pub/packages/infosystems/wais>).

Review

Rex: I've got this great video game that I wrote in Perl/Tk, which is covered in Chapter 14. I'd like to have people play it over the Web. Can server push, client pull, or HotJava™ help?

Debbie: Probably not, unless your game is really slow. Because HTTP is a connectionless protocol, it's not suited for anything requiring real-time interaction. You should define your own protocol that runs over TCP/IP instead.

Rex: So I'll have two programs: The first will be a server program that runs on my workstation and the other will be a client that connects to my server over some other port. I'll keep the server to myself and distribute the client program over FTP and the Web.

Debbie: Good idea-just make sure you avoid port numbers used by other services.

Rex: How can I tell whether my port is unreserved? Is there a universal registry of port numbers?

Debbie: Nope.

Chapter 13, Perl 5 Interactive Course



SPREADING THE WORD: NETWORKING

As the ranks of Internet-ready computers grow, so does the attraction of linking them together. The Web is fine for applications that don't require a continuous flow of data between information providers and information consumers. But you wouldn't want to run a real-time simulation over the Web, or a video game, or anything that depends on moving data around quickly. For those tasks, you need to dig a bit deeper into your computer's operating system so that your programs can speak the Internet protocols necessary to communicate with other computers.

Networking merely means transmitting data between two different computers. In a sense, it's a generalization of the interprocess communication techniques covered in Chapter 9, which move data between two processes running on the same computer. Networking just removes the "same computer" restriction so that computers

anywhere on the Internet can exchange data. Whenever you use e-mail, *finger*, *telnet*, FTP, or the Web, you're using a networked service. Of course, users don't need to know precisely how these services work. But if you want to develop your own application with its own methods of transferring data between two computers, you have to get your hands dirty--and that's just what this chapter does.

Perl's networking facilities use a computational abstraction called sockets. Most of the programs in this chapter assume that your computer uses them, too. If you have a UNIX computer, or WinSock for Microsoft Windows, there's a good chance you have them; otherwise, you probably don't, and you can stop reading after Lesson 2. Lesson 3 leads you through the creation of several network services, starting with a simple server-client pair, building up to an interactive, multiuser service in Lesson 5, and techniques for making it robust in Lesson 6.

HOW THE INTERNET WORKS

Contrary to common sense and common usage, the Internet is not a computer network. It's a *network of networks*. A "computer on the Internet" is merely able to communicate with other computers on the Internet. But what does "communicate" mean? Does it imply a continuous connection, allowing you to telnet or use real-time chat programs such as IRC? Or will a *store-and-forward* connection suffice, allowing you to send electronic mail or read Usenet? Depends on who you ask. To nail down what we mean when we talk about networks, it helps to visualize the infrastructure of Internet services: At the highest level, you can think of the Internet as consisting of applications used by people; at the lowest level, you can think of it as copper wire and fiber-optic cable. So what's in between?

Protocol Layers

One way to conceptualize the different strata is the *OSI reference model*, shown in Figure 13-1, which divides network communication into seven layers, each built on top of the one below. There's nothing "official" about this model; there are several other ways it could be drawn. But the OSI (Open Systems Interconnection) model has become quite popular, in part because it demystifies the layers of computation making up the Internet.

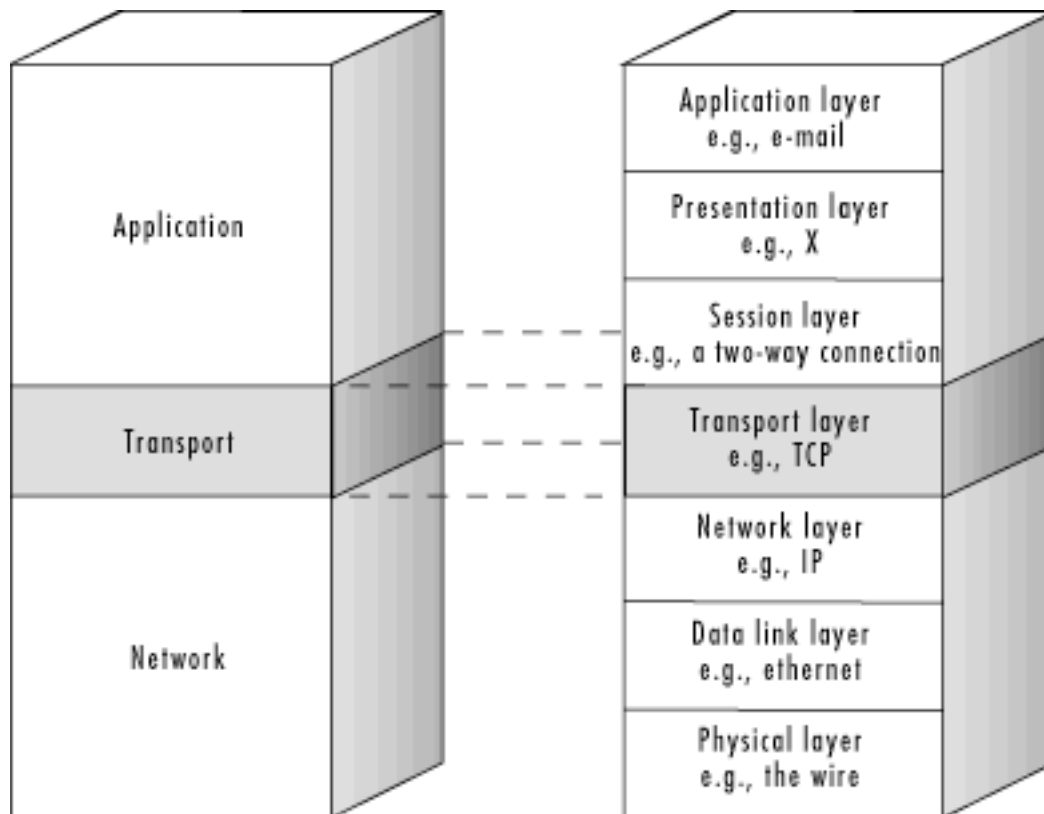


Figure 13-1

The OSI reference model

At the bottom of the model lies the *physical layer*, which consists of the actual hardware connecting two systems: for example, coaxial cable, a microwave transmitter, fiber optics, satellites, and so forth--the tangible technologies responsible for moving the individual bits from sender to receiver. Just above the physical layer is the *data link layer*, which ensures that the bits are moved around accurately (error correction), cleanly (divided into packets), and fairly (so that one computer doesn't hog the entire network).

Computers no longer need to be connected over a single wire, thanks to devices such as routers that make up the *network layer*. Both the data link layer and the network layer operate in either a connection-oriented mode (in which data passes along a semipermanent channel) or a connectionless mode (in which data is sent in unordered spurts); the *transport layer* hides this distinction (as well as other attributes, such as maximum packet size) from higher layers.

Arguably, the *session layer* doesn't merit "layerhood." All it does is oversee the integrity of the packets exchanged between computers, identifying when data might need to be retransmitted but without actually performing the transmission. The *presentation layer* defines the representation of the *application data* for the highest level, the application layer, which concerns what data is transmitted and to where. In well-designed applications, users need not be cognizant of any happenings below the application layer.

The most common method of networking Internet computers is with a pair of protocols: the Transmission Control Protocol (TCP) and the Internet Protocol (IP). Nearly every UNIX computer is bundled with TCP/IP software; from Figure 13-1, you can see that TCP is at level 4 and IP is at level 3 of the OSI network model. (This is a bit of a fib; OSI, Inc. has its own notion of "official" OSI compliance that excludes TCP and IP, but we'll let that slide.)

TCP is connection oriented, in contrast to User Datagram Protocol (UDP), which is connectionless. What's the difference? TCP is like a telephone call: One computer dials another and waits until the connection is established. Then both parties communicate until someone closes the connection. UDP is like mailing a letter: One computer throws a message at another and hopes that it arrives.

Ports

Every computer with TCP/IP lets users and programs connect to different ports. Each port is the home of a different network service: FTP, e-mail, *finger*, *who*, Usenet, and so on (Figure 13-2). Whoever coined the term "port" presumably had an image of a computer as a large city, with multiple places into which a ship could sail. It's not a bad analogy.

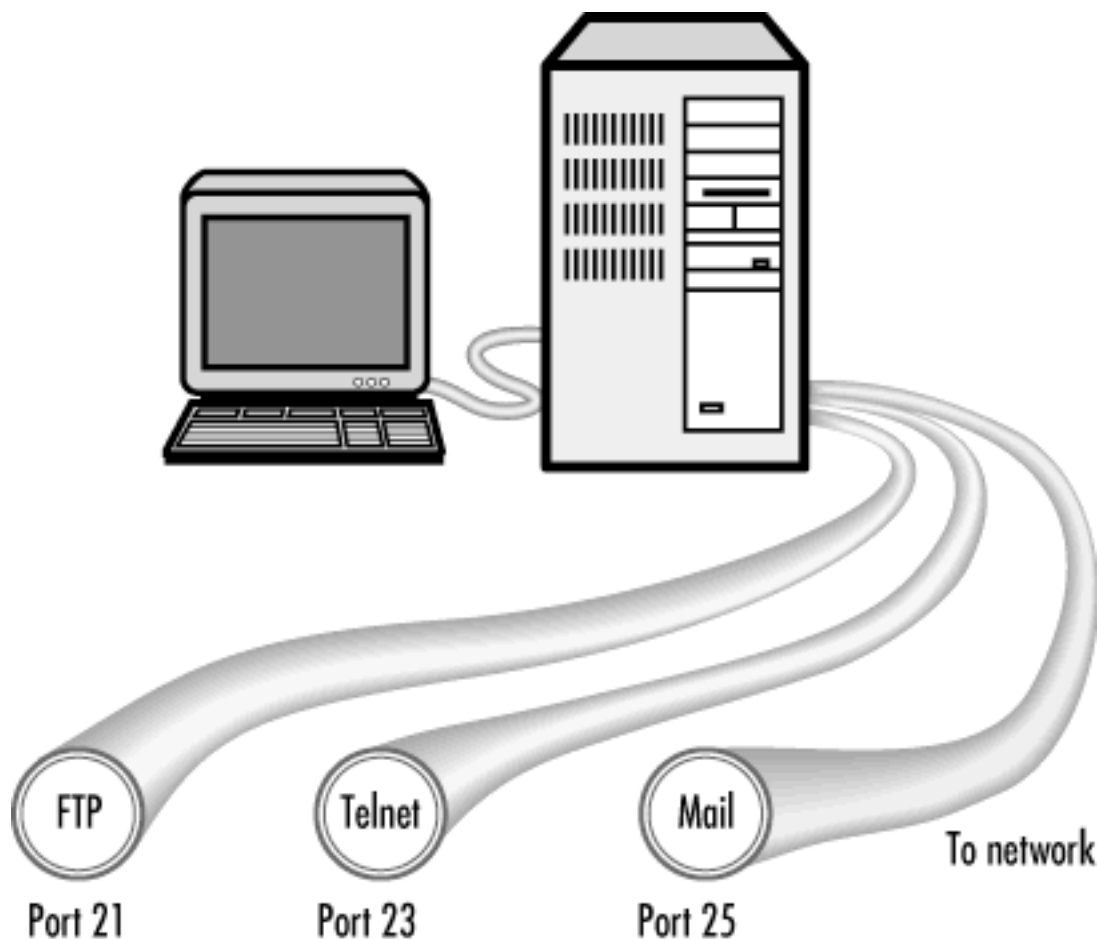


Figure 13-2

Ports

You could also think of a computer as a business, with an IP address for its phone number and each port as the extension of the person whom you want to call. Each person corresponds to a network service; for instance, if your computer sends mail to waite.com, it connects to port 25 (the mail port) at 192.85.206.12 (waite.com's IP address).

When you telnet or FTP or send e-mail, some program on your computer is connecting to a particular port on another computer. Each port is uniquely associated with a network service: FTP requests always arrive at port 21, and anyone or anything connecting to that port is expected to speak the appropriate language: the File Transfer Protocol.

On most UNIX computers, an */etc/services* file tells the computer how to map ports to services. An excerpt from an */etc/services* file is shown in Listing 13-1.

Listing 13-1 Selected lines from an */etc/services* file

```
# Network services
#
ftp 21/tcp
telnet 23/tcp
smtp 25/tcp mail
domain 53/tcp
```

```

finger 79/tcp
nntp 119/tcp readnews untp # Usenet News Transfer Protocol
#
# UNIX services
#
who 513/udp
#
# Local services
#
monty 12345/tcp

```

This */etc/services* file shows you that the finger daemon, which dispenses *finger* information to remote computers, expects *finger* requests on port 79. It also shows you that mail can be received with the Simple Mail Transfer Protocol (SMTP) on port 25.

Most of the entries will be the same from system to system because computers need to agree on port numbers: If one computer suddenly starts accepting FTP requests on port 22 instead of port 21, other computers have no way of knowing. The port numbers are often specified in documents called Requests for Comments (RFCs); there's a link to all the RFCs in Appendix B.

Because the UNIX *telnet* program lets you specify a port, you can connect directly to these services without using intermediaries such as *finger* or mail programs.

```
% telnet waite.com 79
```

```
RESULT TRYING 192.85.206.12...
```

```

Connected to waite.com.
Escape character is '^]'.
perluser
Login name: perluser In real life: Perry Luser
Office: Room 428 Home Phone: (800) 555-1212
Directory: /users/perluser Shell: /bin/csh
On since Jun 2 13:13:28
Learning Perl for fun and profit

```

```

____
/ / / / / / / /
/ /_/ / / / / /
/_ / / / / _/
/ / / / / / ____
/_/ /_/ /_/ /_/

```

```
Connection closed by foreign host.
```

When you connect to port 79 on some computer, you're pretending to be another computer executing the *finger* command. Given a user and a host, *finger* connects to that host at port 79 and does exactly what you did above: It sends the name of the user and prints the result.

Because mail is exchanged on port 25, you can impersonate a mail program and forge your own mail messages.

```
% telnet waite.com 25
```

RESULT Trying 140.174.162.10...

Connected to waite.com.

Escape character is '^]'.
220 waite.com Sendmail 5.65 (1.1.3.6) Wed, 18 Dec 1996

20:06:35 -0500

MAIL FROM: kilroy

250 kilroy... Sender ok

RCPT TO: perluser

250 perluser... Recipient ok

DATA

354 Enter mail, end with "." on a line by itself

From: Your_conscience@nowhere

To: perluser

Subject: Bad programmer! Bad, bad programmer!

I'm coming to get yooooooooooooooooooooooooooooou...

.

250 OK

QUIT

221 waite.com closing connection

Connection closed by foreign host.

perluser will receive a strange message in his or her mailbox, appearing to be from Your_conscience@nowhere. When you connect to port 25, waite.com can't tell whether you're a computer or a person. As long as you provide the fields it needs (MAIL FROM: and RCPT TO:), you can put whatever you want in the message.

Chapter 13, Perl 5 Interactive Course

LESSON 2

NETWORKING UTILITIES

You can create a network service without actual network programming--that is, without using sockets, which are seen by many as arcane and difficult to use. Before we delve into sockets in Lesson 3, you should know about three higher level utilities that let you send and receive messages over a network: the UNIX program *rsh*, the Perl library files *Comm.pl* and *chat2.pl*, and the UNIX "superserver," *inetd*.

rsh

rsh allows to execute commands on a remote computer. You need an account on that computer, and that account must have an *.rhosts* file in your home directory containing the name of the computer you're connecting from (unless the systemwide *hosts.equiv* file is used).

To execute the program *ls* with the argument *-l* on the computer *server.waite.com*, you can type the following:

```
% rsh server.waite.com ls -l
```

RESULT

```
drwxr-xr-x 4 perluser staff 8192 Apr 18 14:44 Library
-rw-r--r-- 1 perluser staff 174 May 7 15:01 junk.txt
drwx----- 1 0 perluser staff 8192 May 7 17:12 Mail
```

rsh has nothing to do with Perl (other than the fact that you can `system()` or `exec()` or backquote an *rsh* command), but it can be a perfectly good substitute for simple networking tasks. Never use a battering ram when a hammer will do.

Comm.pl and *chat2.pl*

Eric Arnold's *Comm.pl* and Randal Schwartz's *chat2.pl* let your Perl program manipulate interactive services in a method similar to the *expect* library. They let you express behaviors like this: "Connect to an FTP server, and when it says *Ready*, send the string *cd pub/perl/scripts* and then *get INDEX.html*, but if it says *Login:*, send the string *anonymous* and wait 5 seconds at most for a reply." In essence, you're preprogramming a conversation into your Perl script. *Comm.pl* and *chat2.pl* are handy for programs that converse with network services such as FTP, or for any other service in which your program has to wait an indeterminate time for the remote computer to respond. *Comm.pl* is available from the CPAN, and *chat2.pl* was bundled with the Perl distribution prior to 5.004. For that reason, the following examples use *chat2.pl*, but *Comm.pl* is newer, and in several ways superior.

They operate in much the same way. After connecting to a network service, a program using *chat2.pl* typically alternates between sending data to the remote service with `chat::print()` and waiting for results with `chat::expect()`. These four functions are the most important:

- **`chat::open_port($server, $port)` opens a connection to the server `$server` on port `$port`, returning a `$handle`.**
- **`chat::print($handle, @data)` sends `@data` to the network service.**
- **`chat::close($handle)` closes the connection.**
- **`chat::expect($handle, $timeout, $pattern1, $body1, $pattern2, $body2...)` expects certain regexes from the network service. If `$pattern1` matches the text returned by the service, `$body1` is evaluated. If `$pattern2` occurs, `$body2` is evaluated, and so on. If `$pattern` is `'TIMEOUT'`, the corresponding `$body` is executed if `$timeout` seconds elapse. If `$pattern` is `'EOF'`, the `$body` is executed if the connection ends before other patterns are seen.**

Three additional functions are included in *chat2.pl*:

- **`chat::select(timeout, @handles)` returns the subset of `@handles` that are ready for I/O, in case your program maintains more than one connection.**
- **`chat::open_proc("command", "argument", "argument2")` opens a `/bin/sh` on a pseudo-tty.**
- **`chat::open_listen()` opens a TCP port on the current machine for listening to incoming connections.**

For more details, see the documentation (currently embedded in the *chat2.pl* file itself).

Because *chat2.pl* is often used for fairly complex interactions with network services, useful examples can be lengthy. Before you see one of those examples, take a look at the simple-minded pinkie in Listing 13-2, which uses *chat2.pl* to finger *pepsi* at *www.cc.columbia.edu*.

Listing 13-2 *pinkie*: Using *chat2.pl* to interact with a network service

```
#!/usr/bin/perl
require `chat2.pl`;
$handle = chat::open_port("www.cc.columbia.edu", 79); #
create connection
chat::print($handle, "pepsi\n"); # send "pepsi\n"
foreach (1..20) {
    chat::expect($handle, 8, `(.+)`, `print "$1\n"`); # print any
    result
}
```

The `chat::open_port()` establishes a connection to *www.cc.columbia.edu* on port 79, the finger port. Then `chat::print()` sends the string *pepsi* to the port, and `chat::expect()` waits eight seconds at most for *www.cc.columbia.edu* to respond. Any string matching `.+` (that is, any text at all) is printed. Here's the result:

```
% pinkie
```

```
RESULT pepsi Pepsi Machine Last seen Mar 24 17:03 on cunixd from ttypl  
(cs.dal.ca)
```

```
Project: All-night caffeine stop.  
Mail forwarded to jte,thor,janey  
Location: 8th Floor, Watson Hall  
Last filled: Probably within the last week or two.  
Inventory:  
-----  
Pepsi [Full] Lemon/Lime Seltzer [Full]  
Diet Pepsi [Full] Lipton Iced Tea [Full]  
7up [Full] Apple Juice [Full]  
Cream Soda [Full] Mountain Dew [Full]  
URL--> http://www.cc.columbia.edu/~pepsi/
```

(Of course, you didn't really need *chat2.pl* just to finger `pepsi@www.cc.columbia.edu` and print the result. You could have done it in one line: `perl -e 'print `finger pepsi@www.cc.columbia.edu`'`.)`

The next *chat2.pl* example, shown in Listing 13-3, does quite a bit more. *randnews* connects to a Usenet news server, requests a list of all of the newsgroups carried by the server, and picks one at random. Then it determines the range of available articles, picks the most recently uploaded article, and displays it for the user.

Listing 13-3 *randnews*: Interacting with a network service (a news server) using *chat2.pl*

```
#!/usr/bin/perl  
require `chat2.pl`;  
srand(time ^ ($$ + ($$<<15)));  
# This isn't a real machine. Use your news server instead.  
$server = `news.waite.com`;  
$port = 119; # The NNTP (Usenet) port.  
$handle = chat::open_port($server, $port);  
die "There's no news server running on $server, port  
$port!\n" unless $handle;  
chat::expect($handle, 5,  
'200 (.*)', 'print "Established connection.\n"');  
chat::print ($handle, "list\n");  
$return = 1;  
while ($return) {  
$return = chat::expect($handle, 5,  
'^\s*215 (.+)', 'print "$server listing groups."';  
return 1; ',  
'^\s*(\S+)\s.*', 'push (@groups, $1); return 1; ',  
'TIMEOUT', 'print "TIMEOUT!\n"; return 0; ');  
}
```

```

print "There are ", scalar(@groups), " news groups.\n";
$group = $groups[rand @groups];
print "Random group is: $group\n";
chat::print ($handle, "group $group\n");
chat::expect($handle, 5, '^\\s*211 \\d+ \\d+ (\\d+)', '$article = $1');
print "Fetching article number: $article\n";
chat::print ($handle, "article $article\n");
$return = 1;
while ($return) {
$return = chat::expect($handle, 5,
'^\\.\\n$', 'return 0',
'(.+)', 'print "$1\\n"; return 1',
'TIMEOUT', 'print "TIMEOUT!\\n"; return 0');
}
chat::close($handle);

```

After the connection to the appropriate port (NNTP, port 119) on the computer named news.waite.com, *randnews* waits for a string beginning with 200, which signals that the news service is ready to begin a conversation. *randnews* then `chat::print()`s the string *list*\n. The news server should reply with the response code 215, followed by a list of all the newsgroups it carries. When the server prints the response code 215, *randnews* prints *news.waite.com listing groups*. In succeeding lines, the group name is extracted and pushed onto @groups.

That can take a while because some servers carry more than 8,000 newsgroups. Once all the group names have been collected, the five-second TIMEOUT strikes, the while loop exits, and a random newsgroup is chosen. *randnews* then selects that group with `chat::print($handle, "group $group\n");` the news server should reply with the response code 211 followed by the numeric ranges of article IDs. *randnews* is interested only in the last value, which is the latest article added to that newsgroup. It then fetches that article with `chat::print($handle, "article $article\n")` and prints any resulting characters, stopping when a lone period appears on a line (the NNTP convention to signal the end of an article).

One run of *randnews* yields the following output. (A few lines have been modified for clarity.)

```
% randnews
```

```
RESULT Established connection.
```

```
news.waite.com listing groups.
```

```
There are 6106 news groups.
```

```
Random group is <alt.tv.sn1>
```

```
Fetching article number 535.
```

```
220 535 <3r60he$1fj@pion.rutgers.edu> Article retrieved; head
and body follow.
```

```
From: nebus@pion.rutgers.edu (Joseph Nebus)
```

```
Newsgroups: alt.tv.sn1
```

```
Subject: Blues Brothers 2000
```

```
Organization: Rutgers University
```

```
I meant to post this last week, but kept forgetting.
```

Anyway, the (NJ) Star-Ledger reports that Dan Aykroyd, John Goodman, and Jim Belushi are set to do a follow-up to the Blues Brothers movie, now entitled "Blues Brothers 2000." It starts after Elwood is released from jail, with the explanation that Jake died while in prison. Anyway, if there's any justice in cult movies the title would indicate that Aykroyd, Goodman and Belushi death race across the country in souped-up vehicles trying to kill pedestrians for "points" while plotting to overthrow a fascist dictator in control of America. But that probably won't be the story. (Also, and this is tangential but bothers me, there was NEARLY a sequel to Strange Brew: The Adventures of Bob & Doug McKenzie made for this summer but Dave Thomas and Rick Moranis weren't able to get everything together in time. A shame. Back at my campus, which is in upstate New York, which I feel the need to point out is close to Canada, hockey, beer and the McKenzies are major parts of the campus culture.)
Joseph Nebus

If it's Usenet news you're after, don't use *chat2.pl*. Use the Net::NNTP module (available from the CPAN), which helps your Perl programs talk to news servers. You could write a news "spammer" (a program that posts to thousands of newsgroups) in just a few lines with this module. Please don't.

inetd

inetd is a server server. That's not a typo--*inetd* (think "Internet daemon") is a program that connects incoming service requests to the appropriate server. This "superserver" listens for connections on certain ports (often listed in */etc/services*) and transfers those connections to the appropriate service (listed in the file */etc/inetd.conf* on most UNIX systems).

Why is this helpful? Because it lets you create an Internet service without having to worry about listening to ports and without spawning new processes to deal with every new client who connects to your service. *inetd* does all the dirty work.

inetd clones is a program for each user who connects, turning your Perl program into an instant network service.

Here's an excerpt from a sample */etc/inetd.conf* file:

```
ftp    stream tcp  nowait root /usr/sbin/ftpd    ftpd
telnet stream tcp  nowait root /usr/sbin/telnetd telnetd
finger stream tcp  nowait root /usr/sbin/fingerd fingerd
time   dgram udp  wait   root internal         time
```

Whenever `inetd` sees a connection to a port mentioned in */etc/services*, it checks to see whether there's a corresponding entry in */etc/inetd.conf*. If there is, it plugs the input and output of that connection into the program specified in the sixth column. So if five people are FTPing files from your computer, you'll see five *ftpd* processes running on your machine.

You can add your own services to */etc/inetd.conf*, sparing yourself the agony of having to create sockets that listen to ports. It's a quick way to turn a program into an Internet service in just a few seconds. See the UNIX `inetd` documentation for further details. However, if you want to do anything tricky, such as have one process deal with multiple users or have client processes directly communicate with one another, you'll want to use sockets, the subject of the next lesson.

Chapter 13, Perl 5 Interactive Course

LESSON 3

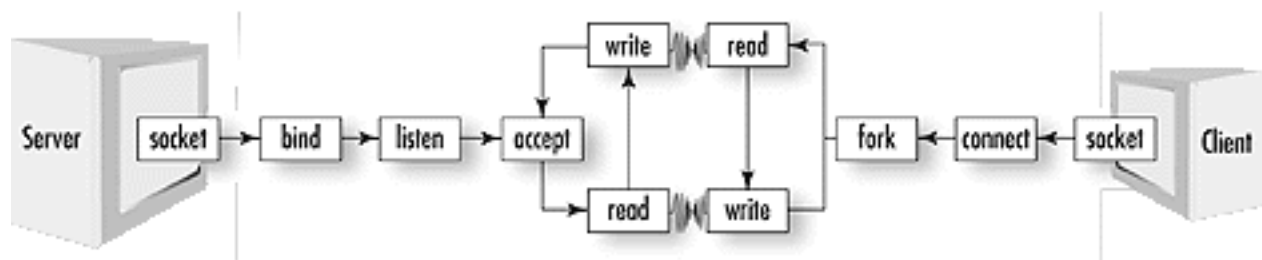
SOCKETS

A *socket* is, roughly, a thing that you plug into a network connection. Sockets are often called "communication endpoints." If that sounds vague, there's a good reason: Sockets can be used for many different styles of communication. In this chapter, we'll be focusing on one in particular: TCP/IP.

In the last lesson, *inetd* demonstrated how ports can be used to turn a program into a network service. In this lesson, you'll use sockets to build your own network service from the ground up, eliminating the need for *inetd* and affording greater control over the connections.

Here's how a typical TCP/IP server works: The server creates a socket (with `socket()`) and binds it to a port on that machine (with `bind()`). Then it listens for incoming connections on that socket (with `listen()`); when it finds one, it accepts that connection (with `accept()`).

A typical client talks to that server by creating a socket (again with `socket()`) and then connecting to the server (with `connect()`) on the agreed-upon port. Figure 13-3 shows the stages in a typical TCP connection.



13-06

Figure 13-3

Stages in establishing a TCP connection

So how do you use these functions? Sockets have a bad reputation, not because the concepts involved are inherently difficult, but because sockets exist at such a high level of abstraction. They handle not just multiple protocols, but multiple *families* of protocols, one of those families being "the Internet." This generality makes sockets harder to use--even though most socket programs are used exclusively for TCP/IP communication, you still have to declare that explicitly when you create them. And instead of

using the name of the protocol, you specify the *type* of service you want. Sound confusing? It is.

The first step in establishing a network connection is creating a socket, with the `socket()` function.

The `socket()` Function

`socket(SOCKET, DOMAIN, TYPE, PROTOCOL)` creates a *SOCKET*.

`socket()` creates the *SOCKET*. The other three arguments are integers, which should have the following values for TCP/IP connections:

- ***DOMAIN* should be `PF_INET` ("Protocol Family for INternET"). It's probably 2 on your computer.**
- ***TYPE* should be `SOCK_STREAM`. It's the type of network service: Does the server speak in streams (TCP, connection-based) or datagrams (UDP, connectionless)? `SOCK_STREAM` is often 1--except on some System V architectures, including Solaris.**
- ***PROTOCOL* should be `(getprotobyname('tcp'))[2]`. It's the particular protocol (such as TCP) to be spoken over the socket. This function is discussed in greater detail in Lesson 4. The value is probably 6 on your system.**

You can think of sockets as being like filehandles:

- You open them, but with `socket()` or Lesson 7's `socketpair()` instead of `open()`.
- You read from them with angle brackets, just like regular filehandles.
- You write to them with `print()`, or `syswrite()`, or Lesson 7's `send()`.
- You close them with `close()` or Lesson 7's `shutdown()`.

Calls to `socket()` usually look like this:

```
use Socket; # defines PF_INET and SOCK_STREAM
socket(SOCKET, PF_INET, SOCK_STREAM,
       (getprotobyname('tcp'))[2]);
```

Don't worry about `getprotobyname()`--it's covered in Lesson 8. As with all of Perl's socket functions, you'll need a *use Socket* to import the correct definitions for `PF_INET` and `SOCK_STREAM`.

The `bind()` Function

`bind(SOCKET, ADDRESS)` binds a network *ADDRESS* to a *SOCKET*.

The *SOCKETs* created by `socket()` are innocent; they haven't yet been polluted by a computer hostname or a port number. `bind()` fleshes out the *SOCKET* with these values. Servers use `bind()` to specify the port at which they'll be accepting connections from clients.

ADDRESS is a socket address, which (for TCP/IP) is a packed string containing three elements:

1. The address family (for TCP/IP, that's `AF_INET`, probably 2 on your system)
2. The port number (for example, 21)

3. The Internet address (for example, 199.177.202.1)

Usually, `bind()` is used by a server, which doesn't need to know its own address, so the argument list looks like this:

```
$port = 12345; # the port our service will use
bind(SOCKET, pack('Sna4x8', AF_INET, $port, "\0\0\0\0"))
or die "Can't bind to port $port! \n";
```

The *or die* clause is very important--if a server dies with outstanding connections, the port won't be immediately reusable unless you take the `SO_REUSEADDR` precaution described in Lesson 6. Successive calls to `bind()` will then fail, as they will if you try to run two servers simultaneously.

The `listen()` Function

`listen(SOCKET, QUEUESIZE)` prepares *SOCKET* for receiving multiple (up to *QUEUESIZE*) connections.

All servers need to make their *SOCKETs* passive, implying that they will be used to `accept()` connections rather than to initiate them with `connect()`, as clients do.

`listen()` is usually invoked like this:

```
listen(SOCKET, 5)
```

which prepares *SOCKET* for incoming requests on its port. (Which port? The one specified during the previous call to `bind()`.)

QUEUESIZE is the maximum number of outstanding connection requests allowed. Generally, `listen()` is used in an infinite loop: As soon as one connection arrives, the server deals with it and then goes back to `listen()`ing for more connections. Usually, connection requests arriving even a few milliseconds apart are handled seamlessly. But it's possible that if a connection request arrives right on the heels of another, it will be lost unless it can be deferred to a queue. *QUEUESIZE* is the number of sockets that can be stored in that queue before the connection requests are irrevocably lost. On many computers, *QUEUESIZE* can't be greater than 5.

The `accept()` Function

`accept(NEW_SOCKET, SOCKET)` creates a *NEW_SOCKET* when a new process connects to *SOCKET*.

Because the server's *SOCKET* is so busy listening for new connections, it doesn't have time to dally about with already established connections. `accept()` is used to create a *NEW_SOCKET* for each new connection; all future communication between client and server then takes place over *NEW_SOCKET*, and *SOCKET* returns to what it does best: `listen()`ing for new connections. Upon success, `accept()` returns the packed address of *NEW_SOCKET*. Upon failure, `accept()` returns `FALSE`.

You'll often see `accept()` used inside a *while* loop.

```
while (1) {
    accept(NEW_SOCKET, SOCKET);
    ...
}
```



```
}
```

So how does the client connect to the server? `bind()` just prepares the socket for the connection; `connect()` actually does the dirty work.

The `connect()` Function

`connect(SOCKET, ADDRESS)` attempts to connect to a remote socket.

ADDRESS is a socket address just like the argument to `bind()`, except that it contains the IP address of the remote server. You can convert Internet addresses (such as `server.waite.com`) to IP addresses (such as `140.174.162.10`) with `gethostbyname()`, which is covered in Lesson 8. For now, a sample invocation will do.

```
$server = "ftp.waite.com";
$port = 21;
connect(SOCKET, pack('Sna4x8', AF_INET, $port,
(gethostbyname($server))[4]))
or die "Can't connect to $server on port $port!";
```

Let's put this all together. First, here's a client that uses `socket()` and `connect()` to establish a connection to the server at `server.waite.com` on port 12345. When *telnet* or FTP connects to a remote site, each performs these two steps as well (and a whole lot more, including *option negotiation*, which involves getting the server and client to agree on characteristics of the connection, such as how clients can delete characters or specify newlines).

client, in Listing 13-4, is fairly straightforward: After creating a socket and connecting to the server, it `fork()`s: The parent process reads server output and prints the result; the child process reads what the user types and writes it to the server.

Listing 13-4 *client*: A sample Internet client, using `socket()` and `connect()`

```
#!/usr/bin/perl -w
use Socket;
use FileHandle;
# Command-line arguments: the server's port and the server's
name.
($port, $server) = @ARGV;
$port && $server or die "Please supply both port and
server.";
socket (SOCKET, PF_INET, SOCK_STREAM,
(getprotobyname('tcp'))[2]);
connect (SOCKET, pack('Sna4x8', AF_INET, $port,
(gethostbyname($server))[4]))
|| die "Can't connect to server $server on port $port.\n";
SOCKET->autoflush();
$pid = fork;
if ($pid == 0) { print STDOUT while (<SOCKET>); }
```

```
# Read from the server
else { print SOCKET while (<STDIN>); close SOCKET; exit}
# Write to the server
```

When you run *client*, be sure to specify a port number and server hostname on the command line. You can use 25 as your \$port and your mailhost as \$server to send yourself e-mail, or use 8888 as the \$port and lambda.parc.xerox.com as the \$server to connect to a MUD.

Because clients aren't much use without servers to talk to, Listing 13-5 shows a simple server. You learned in the last chapter that non-superusers can't run Web servers on low-numbered ports (less than 1024)--that's true of servers in general, which is why the servers in this chapter use \$port = 12345.



Writing servers is easy. Writing bulletproof servers is hard. The three servers in this lesson are minimal; they're designed to be educational rather than robust.

Techniques for avoiding common problems are discussed starting in Lesson 6.

Listing 13-5 *server*: A sample Internet server that reverses messages from the client

```
#!/usr/bin/perl
use strict;
use Socket;
use FileHandle;
my $port = shift || 12345;
socket (SOCKET, PF_INET, SOCK_STREAM,
(getprotobyname('tcp'))[2]);
bind (SOCKET, pack('Sna4x8', AF_INET, $port, "\0\0\0\0"))
|| die "Can't bind to port $port!";
listen (SOCKET, 5);
NEW_SOCKET->autoflush();
SOCKET->autoflush();
while (1) {
accept(NEW_SOCKET, SOCKET);
while (<NEW_SOCKET>) {
chomp;
print NEW_SOCKET scalar(reverse($_)), "\r\n";
}
}
```

After creating the socket, binding it to the port provided on the command line, and then making it listen for new connections from clients, *server* enters an infinite loop. `accept()` blocks until a client tries to connect, at which time a `NEW_SOCKET` will be created for that client. As long as that client sends data (that's the inner *while* loop), *server* sends each line back to the client, backward.

```
your computer server.waite.com
% % server 12345
%
%
```

```
% client 12345 server.waite.com
```

```
RESULT hello!
```

```
!olleh
```

```
sedona
```

```
anodes
```

```
quit
```

```
tiuq
```

```
spools dna
```

```
and sloops
```

```
[CTRL]-[D]
```

```
% client 12345 server.waite.com
```

```
RESULT a man, a plan, a canal- Panama!
```

```
!amanaP -lanac a ,nalp a ,nam a
```

```
(-:
```

```
:-)
```

Chapter 13, Perl 5 Interactive Course

LESSON 4

MULTIUSER SERVICES

One problem with our *server* program is that it can handle only one client at a time. One solution is for the server to `fork()` a new process for every client. The original parent process remains--it's needed to continue listening to the port for new requests. But whenever a new client connects, the parent creates a new socket and then `fork()`s a child process to manage it so that the parent can return to its *while* loop to `accept()` the next client.

We'll extend *server* to do this. Listing 13-6 shows *backward*, a multiuser server. Only one line of *backward* has anything to do with printing lines backward; the rest is generic code that you can adapt for your own uses.

Listing 13-6 *backward*: A multiuser server, which forks a new process for every client

```
#!/usr/bin/perl
use strict;
use Socket;
use FileHandle;
my $port = shift || 12345;
socket(SOCKET, PF_INET, SOCK_STREAM,
(getprotobyname('tcp'))[2]);
bind (SOCKET, pack('Sna4x8', AF_INET, $port, "\0\0\0\0"))
|| die "Can't bind to port $port!\n";
listen(SOCKET, 5);
my $connection = 1; # Incremented for every client
my ($rbits, $pid) = ('', 0); # To keep -w happy.
NEW_SOCKET->autoflush();
SOCKET->autoflush();
while (1) {
    print "Listening for connection $connection\n";
    accept(NEW_SOCKET, SOCKET); # Wait until someone connects.
    print "Someone new just connected!\n";
    print NEW_SOCKET "Welcome to the Internet backwards
server!\r\n";
    $connection++;
    # Now fork() a child process just to deal with the client.
    # The while loop spends most of its time in the select()
```

```

call,
# which blocks until the client sends a message.
if (! ($pid = fork)) {
while (1) {
vec($rbits, fileno(NEW_SOCKET), 1) = 1;
select($rbits, undef, undef, undef);
$_ = <NEW_SOCKET>;
tr/\040-\177//cd;
s/\s+$/;/;
last if /^quit$/i;
print NEW_SOCKET scalar(reverse($_)), "\r\n";
}
print NEW_SOCKET "Come back again soon!\r\n";
close NEW_SOCKET;
exit;
}
}

```

So how is *backward* better than *server*? There are five differences. In decreasing order of importance:

1. *backward* forks a new process for each connection; if there are three clients connected, there will be four processes running on the server's computer: the parent process (which listens for incoming connections) and the child processes (one per client).
2. There's a *quit* command available to users. If a user types *quit*, the inner *while* loop (which is inside the child process) terminates, closing the socket connection.
3. All messages sent to clients are terminated with `\r\n` rather than `\n`. That's because some operating systems (notably Macintoshes) treat `\r` as "move the cursor all the way to the left" and `\n` as "move the cursor down a line," so you need both to construct a proper newline.
4. The current connection number is kept in `$connection`.
5. The client input is "cleaned up" a bit: Undesirable characters are removed (`tr/\040-\177//cd`) and excess whitespace at the beginning of the input is removed (`s/\s+$/;/`).

The line **last if /^quit\$/i** exits the inner *while* loop when the user types *quit*, which prints the salutation *Come back again soon*, closes `NEW_SOCKET`, and `exit()`s. The `exit()` is very important--as with all `fork()`s, it keeps the child process from executing code intended solely for the parent. In this case, it also closes the socket.

We can dispense with *client* and *telnet* directly to the right port on the right machine. The three parallel columns in Table 13-1 illustrate what two clients and a backward server might print during a lesson.

Table 13-1 Using the backward server

client1.state.edu	client2.bureau.org	server.waite.com
%	% backward 12345	
%	% Listening for connection 1.	

% **telnet server.waite.com 12345**

Trying 140.174.162.10
Connected to server.waite.com.

Escape character is '^']'.

Welcome to the Internet
backward server!

yo

oy

redivider

redivider

programmer

remmargorp

10 + 22 = 32

23 = 22 + 01

/^.+./

/.+.^/

quit

Come back again soon!

Listening for
connection 2.

% **telnet server.
waite.com 12345**

Trying 140.174.162.10

Connected to server.
waite.com.

Escape character is '^']'.

Welcome to the
Internet backward
server!

Plastics.

.scitsalP

quit

Come back again
soon!

Listening for connection 3.

Someone new just
connected!

Someone new just
connected!

Listening for
connection 3.

Listening for
connection 3.

Chapter 13, Perl 5 Interactive Course

LESSON 5

INTERACTIVE SERVICES

Now that you have a multiuser service, the next logical step is to let users communicate with one another--maybe to talk, maybe to share data, or maybe to play a game. Or maybe your users aren't humans at all: Perhaps your service lets computers exchange pictures or databases or programs with one another.

Because the *backward* program creates a new process for each user, the IPC techniques in Chapter 9 could be used: For instance, the processes could share a segment of memory and write their messages to that segment. Or you could take an entirely different approach: Keep all users in the same process and just have that process `select()` over a potentially large number of sockets--one per user.

This lesson extends *backward* in a slightly different way: The server acts as a switchboard for information sent between clients or, to be precise, between child processes running on the server. In *backward*, the server process simply spawned new child processes and forgot about them. But the *chatterly* server in this lesson supports a *say* command that lets clients pass messages back and forth. Only the server "knows" about all the clients: who's connected, who's not, and what their process IDs (PIDs) are. So all messages need to pass through the server, which forwards messages from one client to another.

A server acts in the same way as a switchboard--all messages between clients pass through the server (and, in particular, through the parent process in the server), even though it appears that they're talking directly to each other. This scheme is implemented in *chatterly* (Listing 13-7), an extension of *backward* that supports this interclient communication. *chatterly* lets users identify themselves to the server and then lets them speak to other users with the *say* command:

```
% telnet server.waite.com 12345
```

```
RESULT Trying 140.174.162.19...
```

```
Connected to server.waite.com.
```

```
Escape character is '^]`.
```

```
Welcome to the Internet backwards server!
```

```
Who are you?
```

```
Bob
```

```
Bob has arrived.
```

```
say hi to Connie
```

```
Message sent.
```

whereas Connie, who has already connected, sees

```
RESULT Bob has arrived.
```

Bob says, "hi"

Beware! *chatter* is the largest program in this book. The reason it's so much larger than *backward* is that a full-duplex connection needs to be established between each child process and the parent process. *chatter* has the same *while* loop as *backward*, but four subroutines have been added.

1. **manage_child()**, which the child executes on behalf of each client. It handles the "meat" of the server: reversing words and notifying the parent when the client sends a say command or logs out.
2. **wait_for_clients()**, which is used by the parent to identify either when a new connection is made or when a child has something to say to the parent.
3. **login()**, which is used by the parent to map a user's login name to his or her PID (which is used to identify the sockets for reading and writing).
4. **say()**, which is used by the parent to ferry messages from one client to another.

Listing 13-7 *chatter*: Extending backward to support interclient communication

```
#!/usr/bin/perl
# For a more robust version of this server, make the three
# modifications shown in Lesson 6, or use the chatter
# program available from ftp.waite.com.
use strict qw(subs vars);
use Socket;
use FileHandle;
$port = shift || 12345;
socket(SOCKET, PF_INET, SOCK_STREAM,
(getprotobyname('tcp'))[2]);
bind (SOCKET, pack('Sna4x8', AF_INET, $port, "\0\0\0\0"))
|| die "Can't bind to port $port!\n";
listen(SOCKET, 5);
NEW_SOCKET->autoflush();
SOCKET->autoflush();
my $connection = 1;
my (%who, %readpipes, %writepipes);
my ($rbits, $line, $child, $key, $thispipe);
my ($childread, $childwrite, $parentread, $parentwrite);
while (1) {
    print "Listening for connection $connection\n";
    wait_for_clients(); # child processes OR
    # new connections
    $rbits = ``;
    vec($rbits, fileno(SOCKET), 1) = 1;
    if (select($rbits, undef, undef, 0)) { # was it someone
        new...
```



```

print "Someone new just connected!\n";
accept(NEW_SOCKET, SOCKET);
print NEW_SOCKET "Welcome to the Internet backwards server!
\r\n";
print NEW_SOCKET "Who are you? \r\n";
$parentread = "PARENTREAD" . $connection; # parent reads from
child
$childread = "CHILDREAD" . $connection; # child reads from
parent
$parentwrite = "PARENTWRITE" . $connection; # parent writes
to child
$childwrite = "CHILDWRITE" . $connection; # child writes to
parent
$connection++;
pipe ($childread, $parentwrite); # full-duplex
pipe ($parentread, $childwrite);
$parentwrite->autoflush();
if (! ($child = fork)) {
close $parentread;
close $parentwrite;
manage_child($childread, $childwrite);
close $childread;
close $childwrite;
exit;
}
close NEW_SOCKET;
close $childread;
close $childwrite;
$parentwrite->autoflush();
$readpipes{$child} = $parentread; # "switchboard" hash
$writepipes{$child} = $parentwrite; # "switchboard" hash
} else { # ...or someone old?
foreach $key (keys %readpipes) { # which client spoke?
$rbits = ``;
vec($rbits, fileno($readpipes{$key}), 1) = 1;
if (select($rbits, undef, undef, 0) > 0) {
$thispipe = $readpipes{$key};
if ($line = <$thispipe>) {
$line =~ tr/\040-\177//cd; # was it something for the
# switchboard?
if ($line =~ /^SERVER: login (.*)/) { login($key, $1) }
elsif ($line =~ /^SERVER: say (.*)/) { say($key, $1) }
elsif ($line =~ /^SERVER: quit (.*)/) {
delete $who{$1};
delete $readpipes{$key};
delete $writepipes{$key};

```



```

}
$rbits = ``; vec($rbits, fileno($childread), 1) = 1;
if (select($rbits, undef, undef, 0) > 0) {
$_ = <$childread>;
s/\r//g;
s/\n/\r\n/g;
print NEW_SOCKET;
}
}
}
}
# Since all clients should be notified when someone new logs
in, the
# parent needs to call login(), which iterates through
%writepipes
# (there's one per client) and prints an announcement to
each.
# login() also stores the user's name in %who.
sub login {
my ($key, $name) = @_;
my ($pipe, $other);
$who{$name} = $key;
$pipe = $writepipes{$key};
foreach $other (keys %writepipes) {
$pipe = $writepipes{$other};
print $pipe "$name has arrived.\n";
}
print "$name logged on (process $key).\n";
}
# User-to-user communication passes through the parent; the
parent
# calls say() to find the correct socket for writing (by
comparing
# the recipient's name with the names stored in %who).
sub say {
my ($key, $msg) = @_;
my ($from, $to, $what, $recip, $pipe, $found);
($from, $what, $to) = ($msg =~ /^(.*) say (.*) to (.*)$/);
foreach $recip (keys %who) {
if ($recip eq $to) {
$found = 1;
$pipe = $writepipes{$who{$recip}};
next unless $pipe;
print $pipe "$from says, ", ``', $what, ``', "\n";
$pipe = $writepipes{$key};
print $pipe "Message sent.\n";
last;

```

```

}
}
if (!$found) {
$pipe = $writepipes{$key};
print $pipe "No such recipient!\n";
}
}

```

The main *while* loop at the beginning of *chattery* spends most of its time waiting for either of two events:

- A new connection, which creates a child process with its own socket (*backward* and *server* do this, too)
- A message from a child process, which will be *SERVER: login*, *SERVER: say*, or *SERVER: quit*

For every connection, the parent launches a child process, which creates a `NEW_SOCKET` specifically for dealing with its client, just like *backward* and *server*. So why are the two pipes between parent and child required? Because the parent process (the one accepting connections) and the child processes (one per client) occasionally need to talk to one another.

- The parent needs to talk to children when either an arrival message is sent (*Bob has arrived*) or one client talks to another (*Connie says, "Hi!"*). The parent uses two hashes to send Connie's message to Bob: First, it looks up Bob in the `%who` hash, which yields Bob's PID. Then it looks up that PID in the `%writepipes` hash to find the name of the pipe to the appropriate child process.

A child needs to talk to the parent upon login (so the parent knows that Bob has logged in) or when one client wants to send a message to another. Only the parent has a current copy of `%writepipes`; that's why all communication between clients must pass through the parent. (The `%readpipes` hash maps PIDs to pipes used for reading data from the children.) Special messages intended for use solely by the parent process are prefaced with *SERVER:* to distinguish them from normal client messages.

Table 13-2 shows a sample *chattery* session in which two clients connect to the server and send a few messages to one another.

Table 13-2 A *chattery* session

client1.state.edu	client2.bureau.org	server.waite.com
%	%	% chattery 12345
%	%	Listening for connection 1.
%	%	
% telnet server.waite.com 12345		
Trying 140.174.162.19...		
Connected to server.waite.com.		Someone new just connected!

Escape character is '^']'.		Listening for connection 2.
Welcome to the Internet backward server!		
Who are you?	%	
Bob	%	Bob logged on (process 8052).
Bob has arrived.	%	Listening for connection 2.
strap on	%	
no parts	%	
PORT-555	%	
555-PORT	% telnet server.waite.com	
fh38vbm	Trying 140.174.162.19...	
mbv83hf	Connected to server.	Someone new just waite.com.connected!
Al, Al! Art!	Escape character is '^']'.	Listening for connection 3.
!trA !lA ,lA		Welcome to the Internet backward server!
frog	Who are you?	
gorf	Connie	Connie logged on (process 8129).
Connie has arrived.	Connie has arrived.	
say hi to Connie	ssssh	Bob sent a message to Connie.
Message sent.	Bob says, "Hi."	Listening for connection 3.
hoo-boy	say hello to Bob	Connie sent a message to Bob.
Connie says, "Hello."	Message sent.	Listening for connection 3.
quit		
Connection closed by foreign host.		Listening for connection 3.
	say are you there? to Bob	Connie sent a message to Bob.
	No such recipient!	Listening for connection 3.
quit		

Connection closed by
foreign host.

Listening for
connection 3.

chattery isn't foolproof. For instance, clients might leave *zombie* processes, which can appear in the output of *ps* as **<defunct>**. And the server exits if it tries to write to a pipe whose read handle has been closed. These problems are solved in the next lesson.

Chapter 13, Perl 5 Interactive Course

LESSON 6

BULLETPROOFING SERVERS

The *chatterly* program from the last lesson certainly isn't perfect. There are three improvements that will make it more robust:

1. The server can wait for each child to exit, removing zombie (<defunct>) processes.
2. The server can reuse local addresses (so you don't have to wait to use the port again if your server crashes).
3. When launched, the server should check to see whether other servers are running.

Zombie Processes

In UNIX, a *zombie* process is a process that has exited but hasn't been wait()ed for. If you run *server*, *backward*, or *chatterly*, clients sometimes leave unkillable zombie processes. If you run *ps*, you might see entries like these:

```
% ps
```

```
RESULT PID TT STAT TIME COMMAND
```

```
18646 p3 W 0:00 <defunct>
18648 p3 W 0:00 <defunct>
18650 p3 W 0:00 <defunct>
18652 p3 W 0:00 <defunct>
18654 p3 W 0:00 <defunct>
```

The solution is to have your server wait() for the children to exit. How does it know when a child exits? Easy--the CHLD signal is sent to the parent. That signal can be trapped and fed to a handler that wait()s for children to die. Chapter 9, Lesson 5, provides a generic signal handler; now you want to embellish it somewhat so that it cleans up after the child by deleting all traces of the child from the server's memory: %who, %readports, and %writeports. Add the following to all your servers that resemble *server* or *backward* or *chatterly*:

```
use POSIX `WNOHANG`; # If Perl complains here, try "sub
WNOHANG { 1; }"
$SIG{CHLD} = sub {
my($pid);
while (1) {
$pid = waitpid(-1, WNOHANG);
last if $pid < 1;
# The next three lines are specifically for chatterly.
```

```

delete $who{$pid};
delete $readports{$pid};
delete $writeports{$pid};
}
};

```

On some UNIX systems, you can get away with `$SIG{CHLD} = 'IGNORE'`, but the above solution is more portable.

Reusing Local Addresses

When you run a server multiple times in succession, you'll often have to wait awhile before you can use the same port again. That's annoying--wouldn't it be nice if you could reuse the port immediately?

You can do that using `setsockopt()`, which lets you control low-level socket options.

The `getsockopt()` and `setsockopt()` Functions

`getsockopt(SOCKET, LEVEL, OPTION)` returns the socket *OPTION* requested.

`setsockopt(SOCKET, LEVEL, OPTION, VALUE)` sets *OPTION* to *VALUE*.

`getsockopt()/setsockopt()` returns/sets the value of the socket *OPTION*; both return UNDEF if there's an error.

LEVEL is an integer associated with the protocol being used. (Use `SOL_SOCKET`.)

Some *OPTIONS* are shown in Table 13-3.

Table 13-3 Reusing socket addresses

Socket Option	What It Does
<code>SO_REUSEADDR</code>	If set, will reuse addresses, avoiding that <i>Can't bind to port error</i> .
<code>SO_KEEPALIVE</code>	If set, keeps connections active (for periodic transmissions).
<code>SO_LINGER</code>	If set, the socket will be prevented from closing if there are still messages to be sent.
<code>SO_DEBUG</code>	Enables or disables debugging in the underlying protocol modules.
<code>SO_BROADCAST</code>	Permits sending of broadcast messages.
<code>SO_SNDBUF</code>	Sets the size of the "send" buffer.
<code>SO_RCVBUF</code>	Sets the size of the "receive" buffer.

To reuse socket addresses, just add this line to your server after the call to `socket()`:

```

setsockopt(SOCKET, SOL_SOCKET, SO_REUSEADDR, 1);

```


Server Exclusivity

You usually want just one copy of a server to be running at a time. Often, a *bind* or *die* statement prevents additional copies from running. But what if your server chooses the port dynamically or it decides to wait until the port becomes free?

It's better for the server itself to ensure that no other server is running before it starts to operate. There are many ways to do this; the best way is through file locking. Your server should attempt to lock the lockfile when it begins; if it can't, it knows that another server is running. If you wanted your lockfile to be */tmp/chattery.lock*, here's how you might implement file locking:

```
$LOCK_EX = 2;
$LOCK_NB = 4;
open(LOCK, ">/tmp/chattery.lock");
if (!flock(LOCK, $LOCK_EX | $LOCK_NB)) {
    die "There's another server running! \n";
}
print LOCK $$;
```

By asking for a nonblocking (*\$LOCK_NB*) lock, *flock()* won't wait for the file to become free. If it fails, it fails immediately, telling you that another server is already running. The method is fairly foolproof--if the server crashes or the computer is rebooted, the lock will be automatically released.

The last line of the code snippet above, *print LOCK \$\$*, places the server's PID in the lock file. That's not necessary--the contents of the lock are irrelevant--but it's often handy to be able to find the PID of your server should you need to kill your server in a hurry.

If you want to modify *chattery* for your own purposes, make the three modifications suggested in this lesson or grab the *chatter* program from ftp.waite.com.

Chapter 13, Perl 5 Interactive Course

LESSON 7

ADVANCED SOCKETS

This lesson describes eight socket functions that aren't used in *server*, *backward*, or *chatterly*. Should you wish to customize the servers in this chapter for your own use, you might find them useful.

Using `socketpair()` to Create a Two-Way Pipe

Remember why you're going to all the trouble of specifying address families and protocol families? There wouldn't be any reason to pass `AF_INET` and `PF_INET` as arguments if those were the only possible values. There are alternatives, such as `AF_UNIX` and `PF_UNIX`, which you can use for intra-computer communication.

In Chapter 9, you saw that `pipe()` creates a half-duplex pipe (on most computers). To create a full-duplex connection, you have to cobble one together out of two half-duplex pipes, as *chatterly* does. But on some computers with sockets, you can use `socketpair()` to create two connected sockets.

The `socketpair()` Function

`socketpair(SOCKET1, SOCKET2, DOMAIN, TYPE, PROTOCOL)` creates a pair of connected sockets.

The *DOMAIN*, *TYPE*, and *PROTOCOL* arguments are all similar to `socket()` calls except that you'll want to use `AF_UNIX` (the address family for UNIX domain sockets) as your *DOMAIN* if your goal is to pass information between two sockets on the same computer. `socketpair()` returns `TRUE` if successful.

Listing 13-8 shows *tube*, a short program that demonstrates using `socketpair()` to pass data both ways through a socket. After creating the pair of connected sockets, *tube* forks; the parent closes one socket and the child closes the other.

Listing 13-8 *tube*: Using `socketpair()` to create a bidirectional pipe

```
#!/usr/bin/perl -w
use Socket;
use FileHandle;
socketpair(SOCKET1, SOCKET2, AF_UNIX, SOCK_STREAM, 0) or die
"socketpair()";
SOCKET1->autoflush();
SOCKET2->autoflush();
$pid = fork;
if ($pid) { # parent: reads and writes to SOCKET1
```

```

close SOCKET2;
print SOCKET1 "220 volts!\n";
print "The child wants ", scalar(<SOCKET1>);
} elsif (defined $pid) {
close SOCKET1; # child: reads and writes to SOCKET2
print SOCKET2 "110 volts!\n";
print "The parent wants ", scalar(<SOCKET2>);
} else {
die "Couldn't fork: $! ";
}

```

The parent and child write to each other at (approximately) the same time. Then they both try to read what the other has written.

```
% tube
```

```
RESULT The parent wants 220 volts!
```

```
The child wants 110 volts!
```

Not all computers with sockets support `socketpair()`. If yours doesn't, Perl will complain as your program is being compiled.

Socket Addresses

You've seen that `bind()` and `connect()` stuff socket addresses into a socket. `getsockname()` and `getpeername()` extract socket addresses for you.

The `getsockname()` and `getpeername()` Functions

`getsockname(SOCKET)` returns the packed address of the local end of the *SOCKET* connection.

`getpeername(SOCKET)` returns the packed address of the remote end of the *SOCKET* connection.

`getsockname()` and `getpeername()` return the socket address (the same data structure passed to `bind()` and `connect()`) of the local/remote end of a socket. The return value is an array (`$family`, `$port`, `@address`): the address family, the port number, and the Internet address.

`getpeername()` is used much more often than `getsockname()` because your program usually knows the local address. Listing 13-9 demonstrates this.

Listing 13-9 *peername*: Using `getsockname()` to extract socket address information

```

#!/usr/bin/perl -w
use Socket;
$server = "ftp.waite.com";
$port = 21;
socket (SOCKET, PF_INET, SOCK_STREAM,
(getprotobyname('tcp'))[2]);
connect(SOCKET, pack('Sna4x8', AF_INET, $port,
(gethostbyname($server))[4])) or die "Can't connect to

```

```
$server!";
($family, $port, @address) = unpack('SnC4',
getpeername(SOCKET));
print "Family: $family \n";
print "Port: $port \n";
print "Address: ", join('.', @address), "\n";
% sockname
Family: 2
Port: 21
Address: 140.174.162.12
```

getsockname() behaves identically to getpeername() except that it's used from the server side instead of the client side. If you replace the connect() in *sockname* with bind() and run the resulting program on ftp.waite.com, you'll see identical results.

Socket Control

So far, you've treated sockets just like regular filehandles: sending data across them with print(), reading data from them with <SOCKET>, and closing them with close(). You can exert finer control with send(), recv(), and shutdown().

shutdown()

You can partially close a socket so that it becomes a one-way connection with shutdown().

The shutdown() Function

shutdown(SOCKET, ALLOW) shuts down a socket connection.

shutdown() offers finer control than close() (Figure 13-4); it terminates a socket connection in different ways, depending on the value of *ALLOW*.

- If *ALLOW* is 0, no more receives will be allowed.
- If *ALLOW* is 1, no more sends will be allowed.
- If *ALLOW* is 2, neither will be allowed.

Listing 13-10 shows *earplug*, a client that stops receiving (but not sending) data when the server sends any string with three consecutive s's (as in *ssssh!*).



`shutdown (socket,0)`



`shutdown (socket,1)`



`shutdown (socket,2)`

Figure 13-4

`shutdown()`

Listing 13-10 *earplug*: Using `shutdown()` to close a socket connection partially

```
#!/usr/bin/perl -w
use Socket;
use FileHandle;
($port, $server) = @ARGV;
socket (SOCKET, PF_INET, SOCK_STREAM,
(getprotobyname('tcp'))[2]);
connect(SOCKET, pack('Sna4x8', AF_INET, $port,
(gethostbyname($server))[4])) or die "Can't connect to
$server! \n";
SOCKET->autoflush();
$pid = fork;
if ($pid) { # parent; handles messages from the server
while (<SOCKET>) {
last if /sss/;
print;
}
print "I won't read any more. \n";
```

```

shutdown(SOCKET, 0); # stop listening
print while <SOCKET>; # this will never print anything!
print "There's no more data! \n";
} elsif (defined $pid) {
print SOCKET while <STDIN>;
exit;
} else { die "Couldn't fork: $! " }

```

earplug appears to be a normal client until the magic characters *sss* are sent across the socket.

```
% earplug 12345 your.computer.here
```

RESULT Welcome to the Internet backwards server!

Who are you?

Bob

speed

deeps

LeMac

caMeL

ssssh!

I won't read any more.

There's no more data!

send() and *recv()*

send() and *recv()* send and receive messages across sockets, similar to *syswrite()* and *sysread()* or *print()* and *read()*. All these functions work with sockets; *send()* and *recv()* are the lowest level of the bunch.

The *send()* and *recv()* Functions

send(SOCKET, MESSAGE, FLAGS, TO) sends *MESSAGE* over a *SOCKET*.

recv(SOCKET, SCALAR, LENGTH, FLAGS) receives a message on a *SOCKET*.

send() sends the *MESSAGE* string on *SOCKET*. If the socket provided to *send()* is unconnected, you can specify a destination socket with *TO*. (Otherwise, you can omit the *TO*.)

recv() tries to read *LENGTH* bytes from *SOCKET*, placing the result in *SCALAR*.

send() returns the number of characters sent, and *recv()* returns the address of the sender. Both *send()* and *recv()* return *UNDEF* if an error occurs, setting *\$!*.

The available *FLAGS* vary from computer to computer; typically, *send()* can use *MSG_OOB* to send out-of-band data and *MSG_DONTROUTE* to avoid using the routing tables that direct packets from one place to another. *recv()* can also use *MSG_OOB* and *MSG_PEEK*, which grabs the message but treats it as unread so that the next *recv()* accesses the same message.

On most computers, *send()* and *recv()* are system calls; consult your online documentation for more detail (such as which flags are available on your system).

sendrecv, in Listing 13-11, demonstrates using *send()* and *recv()* in a client to transmit a message and receive a response.

Listing 13-11 *sendrecv*: Using `send()` and `recv()` to transmit data through a socket

```
#!/usr/bin/perl
use Socket;
($port, $server) = @ARGV;
$port = $port || 12345;
$server = $server || "server.waite.com";
socket (SOCKET, PF_INET, SOCK_STREAM,
(getprotobyname('tcp'))[2]);
connect(SOCKET, pack('Sna4x8', AF_INET, $port,
(gethostbyname($server))[4])) or die "Can't connect to
$server!\n";
send(SOCKET, "perluser\n", 0); # send a message (with no
flags)
recv(SOCKET, $string, 10, 0); # receive a message (with no
flags)
print "Received: <$string>.\n";
close SOCKET;
```

`send(SOCKET, "perluser\n", 0)` has the same effect as typing *perluser* when telnetting to the server. If *chattery* is running on *server.waite.com*, it greets the client with its usual salutation, *Welcome to the Internet backwards server!*. The first ten bytes from the socket stream (*Welcome to*) are then placed in `$string`.

```
% sendrecv
```

```
RESULT Received: <Welcome to>.
```


Chapter 13, Perl 5 Interactive Course

LESSON 8

NETWORK DATABASES

The client and servers in this chapter have used a few unexplained functions, such as `getprotobyname()` and `gethostbyname()`. There are many similar functions that you won't use often, but they're part of Perl and therefore merit discussion, so get ready for a barrage.

Chapter 11 discussed *passwd* and *group* databases, each of which contains UNIX account information. That chapter also mentioned a number of functions for extracting specific information from the databases (e.g., `getpwuid()`, `getgrname()`, `getgrgid()`) as well as for iterating through them (e.g., `getpwent()`, `getgrent()`, `endpwent()`). UNIX networking employs four databases with similar access methods: *hosts*, *services*, *protocols*, and *networks*. The databases aren't always stored in plain files (often */etc/hosts*, */etc/services*, */etc/protocols*, and */etc/networks*), but when they are, they contain one entry per line, just like */etc/passwd* and */etc/group*.

- Each *host* entry contains information about the address of a networked computer. If your computer has an */etc/hosts* file, it's probably full of lines that look like this:

```
18.85.0.2    media-lab.media.mit.edu
18.85.6.208  fahrenheit-451.media.mit.edu fahr celsius-233
18.85.13.106 hub.media.mit.edu
127.0.0.1    localhost.media.mit.edu localhost
```

The first column contains the Internet (IP) address, the second column contains the name of the host, and any remaining columns contain aliases for that host; here you see that *fahrenheit-451.media.mit.edu* has the aliases *fahr* and *celsius-233*.

- Each *service* entry contains information about a particular network service, such as FTP, *finger*, or SMTP (mail). The */etc/services* file was discussed in Lesson 1.
- Each *protocol* entry contains information about an Internet protocol. Your */etc/protocols* file might look like this:

```
ip    0  IP    # internet protocol
icmp  1  ICMP  # internet control message protocol
ggp   3  GGP   # gateway-gateway protocol
tcp   6  TCP   # transmission control protocol
pup   12 PUP   # PARC universal packet protocol
udp   17 UDP  # user datagram protocol
```


● Each *network* entry contains information about an Internet or local network. Sometimes, */etc/networks* contains only one line looking something like this:

loop 127 loopback

The first column contains the name of the network (loop), the second column contains the network number (127, often the first value of the IP address), and any remaining columns contain other names for the network (loopback).

As with *passwd* and *group*, each database has two functions that extract specific entries and three functions that iterate through the database, for a total of 20 in all, as shown in Table 13-4.

Table 13-4 UNIX and Perl networking functions

Hosts	Services	Protocol	Networks
Extractors			
gethostbyname()	getservbyname()	getprotobyname()	getnetbyname()
gethostbyaddr()			getnetbyaddr()
		getprotobynumber()	
	getservbyport()		
Iterators			
gethostent()	getservent()	getprotoent()	getnetent()
sethostent()	setservent()	setprotoent()	setnetent()
endhostent()	endservent()	endprotoent()	endnetent()

Hosts

The *host* functions allow you to extract information about computer hostnames from */etc/hosts* (or its equivalent on your system).

The gethostbyname() and gethostbyaddr() Functions

gethostbyname(*NAME*) returns host information for the host named *NAME*.

gethostbyaddr(*ADDRESS*, *TYPE*) returns host information for the host with address *ADDRESS*.

Both `gethostbyname()` and `gethostbyaddr()` return a five-element array: (*\$name*, *\$aliases*, *\$address_type*, *\$length*, *@addresses*). *TYPE* is the address type: 2 for Internet addresses.

hostinfo (Listing 13-12) first decides whether to use `gethostbyname()` or `gethostbyaddr()`. It uses `gethostbyname()` if the command-line argument contains a letter--in other words, if it looks like `ftp.waite.com` and not `18.85.6.208`.

Listing 13-12 *hostinfo*: Using `gethostbyname()` and `gethostbyaddr()` to obtain information about Internet hosts

```
#!/usr/bin/perl -wl
if ($ARGV[0] =~ /[a-zA-Z]/) {
    ($name, $aliases, $type, $length, @addresses) =
    gethostbyname($ARGV[0]);
} else {
    ($name, $aliases, $type, $length, @addresses) =
    gethostbyaddr(pack('C4', split('\.', $ARGV[0])), 2);
}
print "Name: $name";
print "Aliases: $aliases";
print "Address type: $type";
print "Length: $length";
print "Address: ", join('.', unpack('C4', $addresses[0]));
% hostinfo ftp.waite.com
```

RESULT Name: hamilton.dnai.com

```
Aliases: ftp.waite.com
Address type: 2
Length: 4
Address: 140.174.162.12
% hostinfo 18.85.6.208
```

RESULT Name: fahrenheit-451.media.mit.edu

```
Aliases:
Address type: 2
Length: 4
Address: 18.85.6.208
```

The hosts database can be accessed with three additional functions: `gethostent()`, `sethostent()`, and `endhostent()`.

The `gethostent()`, `sethostent()`, and `endhostent()` Functions

`gethostent()` retrieves the host information from the "next line" in the hosts database.

`sethostent(STAYOPEN)` (re)opens the hosts database and moves to the beginning.

`endhostent()` closes the hosts database.

`gethostent()`, `sethostent()`, and `endhostent()` work just like their `passwd` counterparts `getpwent()`, `setpwent()`, and `endpwent()`, with one difference: If a nonzero *STAYOPEN* argument is given to `sethostent()`, the hosts database remains open even after calls to the extractor functions `gethostbyname()` and `gethostbyaddr()`. Listing 13-13 demonstrates the `gethostent()` function.

Listing 13-13 *allhosts*: Using `gethostent()` to iterate through the hosts database

```
#!/usr/bin/perl -wl
while (@fields = gethostent) {
    print "Name: $fields[0]";
    print "Aliases: $fields[1]";
    print "Address type: $fields[2]";
    print "Length: $fields[3]";
    print "Address: ", join('.', unpack('C4', $fields[4]));
}
```

Some computers are *nameservers*, providing host information about the rest of the Internet to a local network. On these computers, `gethostent()` may attempt to grab information about machines from all across the Internet, so it's likely that `gethostent()` has been deactivated.

Services

The *serv* functions allow you to extract information about network services from */etc/services* (or its equivalent on your system).

The `getservbyname()` and `getservbyport()` Functions

`getservbyname(NAME, PROTOCOL)` returns information about the service *NAME*.

`getservbyport(PORT, PROTOCOL)` returns information about the service at *PORT*.

`getservbyname()` and `getservbyport()` extract information from the services database, returning a four-element array: (*\$name*, *\$aliases*, *\$port*, *\$protocol*). Because some services use multiple protocols (e.g., you can access the daytime service via TCP or UDP), there's a *PROTOCOL* argument ("tcp" or "udp") to specify which ones you want. Listing 13-14 demonstrates the `getservbyname()` and `getservbyport()` functions.

Listing 13-14 *servinfo*: Using `getservbyname()` and `getservbyport()` to obtain information about network services on your computer

```
#!/usr/bin/perl -wl
if ($ARGV[0] =~ /[a-zA-Z]/) {
    ($name, $aliases, $port, $protocol) = getservbyname($ARGV[0],
    'tcp');
} else {
    ($name, $aliases, $port, $protocol) = getservbyport($ARGV[0],
    'tcp');
}
print "Information about service $ARGV[0]:";
print "Name: $name";
print "Aliases: $aliases";
```

```
print "Port: $port";
print "Protocol: $protocol";
```

If *servinfo*'s command-line argument contains a letter, `getservbyname()` is used; otherwise, `getservbyport()` is used.

```
% servinfo 119
Information about service 119:
Name: nntp
Aliases: usenet
Port: 119
Protocol: tcp
% servinfo smtp
Information about service smtp:
Name: smtp
Aliases: mail
Port: 25
Protocol: tcp
```

The *services* database can be accessed with three additional functions: `getservent()`, `setservent()`, and `endservent()`.

The `getservent()`, `setservent()`, and `endservent()` Functions

`getservent()` retrieves the host information from the "next line" in the services database.

`setservent(STAYOPEN)` (re)opens the services database and moves to the beginning.

`endservent()` closes the services database.

These functions behave just like their *passwd* and *host* counterparts described earlier; the *STAYOPEN* argument is identical to `sethostent()`'s *STAYOPEN*. `getservent()` is demonstrated in Listing 13-15.

Listing 13-15 *all/servs*: Using `getservent()` to iterate through the services database

```
#!/usr/bin/perl -wl
while (@fields = getservent) {
    print "Name: $fields[0]";
    print "Aliases: $fields[1]";
    print "Port: $fields[2]";
    print "Protocol: $fields[3]";
}
```

Protocols

The *protocol* functions allow you to extract information about network protocols from */etc/protocols* (or its equivalent on your system).

The `getprotobyname()` and `getprotobynumber()` Functions

getprotobyname(*NAME*) returns protocol information for the protocol named *NAME*.

getprotobynumber(*NUMBER*) returns protocol information for protocol number *NUMBER*.

Both `getprotobyname()` and `getprotobynumber()` return a three-element array: (*\$name*, *\$aliases*, *\$protocol*).

If *protinfo*'s (Listing 13-16) command-line argument contains a letter, it's assumed to be a protocol name and `getprotobyname()` is used. Otherwise, `getprotobynumber()` is used.

Listing 13-16 *protinfo*: Using `getprotobyname()` and `getprotobynumber()` to obtain information about Internet protocols

```
#!/usr/bin/perl -wl
if ($ARGV[0] =~ /[a-zA-Z]/) {
    ($name, $aliases, $protocol) = getprotobyname($ARGV[0]);
} else {
    ($name, $aliases, $protocol) = getprotobynumber($ARGV[0]);
}
print "Information about protocol $ARGV[0]:";
print "Name: $name";
print "Aliases: $aliases";
print "Protocol: $protocol";
% protinfo tcp
```

RESULT Information about protocol TCP:

```
Name: tcp
Aliases: TCP
Protocol: 6
% protinfo 0
```

RESULT Information about protocol 0:

```
Name: ip
Aliases: IP
Protocol: 0
```

The local protocols database can be accessed with three additional functions: `getprotoent()`, `setprotoent()`, and `endprotoent()`.

The `getprotoent()`, `setprotoent()`, and `endprotoent()` Functions

getprotoent() retrieves the protocol information from the "next line" in the protocols database.

setprotoent(*STAYOPEN*) (re)opens the protocols database and moves to the beginning.

endprotoent() closes the protocols database.

These functions behave just like their *passwd* and *host* counterparts described earlier; the *STAYOPEN* argument is identical to `sethostent()`'s *STAYOPEN*. `getprotoent()` is featured in Listing 13-17.

Listing 13-17 *allprots*: Using `getprotoent()` to iterate through the protocols database

```
#!/usr/bin/perl -wl
while (@fields = getprotoent) {
    print "Name: $fields[0]";
    print "Aliases: $fields[1]";
    print "Protocol: $fields[2]";
}
```

Networks

The *net* functions allow you to extract information about networks from */etc/networks* (or its equivalent on your system).

The `getnetbyname()` and `getnetbyaddr()` Functions

`getnetbyname(NAME)` returns information about the network *NAME*.

`getnetbyaddr(ADDRESS, TYPE)` returns information about the network at *ADDRESS*.

Both `getnetbyname()` and `getnetbyaddr()` return a four-element array: (*\$name*, *\$aliases*, *\$address_type*, *\$network*). *TYPE* is the address type: 2 for Internet addresses. These functions are demonstrated in Listing 13-18.

Listing 13-18 *netinfo*: Using `getnetbyname()` and `getnetbyaddr()` to obtain information about Internet protocols

```
#!/usr/bin/perl -wl
if ($ARGV[0] =~ /[a-zA-Z]/) {
    ($name, $aliases, $type, $network) = getnetbyname($ARGV[0]);
} else {
    ($name, $aliases, $type, $network) = getnetbyaddr($ARGV[0],
2);
}
print "\nInformation about network $ARGV[0]:";
print "Name: $name";
print "Aliases: $aliases";
print "Address type: $type";
print "Network: $network";
```

If *netinfo*'s command-line argument contains a letter, *netinfo* assumes it's a name and uses `getnetbyname()`. Otherwise, it uses `getnetbyaddr()`.

```
% netinfo loopback
```

RESULT Information about network loopback:

```
Name: loopback
```

```
Aliases: loopback-net software-loopback-net
```

```
Address type: 2
Network: 127
% netinfo 10
```

RESULT Information about network 10:

```
Name: arpanet
Aliases: arpa
Address type: 2
Network: 10
```

The *networks* database can be accessed with three additional functions: `genetent()`, `setnetent()`, and `endnetent()`.

The `genetent()`, `setnetent()`, and `endnetent()` Functions

`genetent()` retrieves the network information from the "next line" in the *networks* database.

`setnetent(STAYOPEN)` (re)opens the *networks* database and moves to the beginning.

`endnetent()` closes the *networks* database.

These functions behave just like their *passwd* and *host* counterparts described earlier; the *STAYOPEN* argument is identical to `sethostent()`'s *STAYOPEN*. `genetent()` is demonstrated in Listing 13-19.

Listing 13-19 *allnets*: Using `genetent()` to iterate through the *networks* database

```
#!/usr/bin/perl -l
while (@fields = genetent) {
    print "Name: $fields[0]";
    print "Aliases: $fields[1]";
    print "Address type: $fields[2]";
    print "Network: $fields[3]";
}
```

Review

Susan: *chatter* is huge. I'm surprised that the difference in size is between *chatter* and *backward* is so much larger than the difference between *backward* and *server*—I would have thought that once a server can handle multiple users, it wouldn't be that hard to add communication between them. I was wrong.

Tommy Lee: That surprised me, too. Servers can get pretty complex. I'm anxiously awaiting the `Server::module` set, which I hear will be available in the CPAN.

Susan: I bet those servers kill zombie processes, too. I had 41 defunct processes on my computer until I got to Lesson 6 and learned how to kill them. Now I'm using the *sigtrap* pragma to trap all those pesky signals. It's helpful—I hadn't realized how often servers receive PIPE signals.

Tommy Lee: PIPE signals occur whenever your server writes to a pipe whose client has disconnected. If

you just avoid trapping them, they'll kill the client process, which raises another signal: `CHLD`. When the server notices that, it can then remove all traces of that client from its data structure.

Chapter 14, Perl 5 Interactive Course



TOWER OF BABEL: PERL AND OTHER LANGUAGES

In spite of its humble origins as a text-processing and system administration tool, Perl is now being used for tasks once reserved for special-purpose languages, from mathematical computation to scientific visualization and animation. You can create graphical applications in X or *Curses* (ASCII graphics); this chapter includes some sample X applications (including an animation) written in Perl, and a menu program and animated game written in *Curses*. That should convince you that Perl can be used for a lot more than "mere" text processing.

Perl can also be used for a lot less: *awk*, *sed*, and *find* scripts can be converted to Perl with the *a2p*, *s2p*, and *find2perl* programs, all bundled with the Perl distribution.

Many people complain that Perl isn't as fast as C, and rightfully so: C sacrifices portability for speed.

Many C applications are inherently faster than their Perl counterparts because C lets you program "closer" to machine language and because your program doesn't have to be interpreted before it can be run. But you can have the best of both worlds: Perl programs can use C subroutines (and libraries), C programs can use Perl subroutines, and as of Perl 5.002, you can compile your Perl programs into C source code, or into a fast-loading, platform-independent bytecode.

Here's a road map for the chapter:

- **Lessons 1, 2, 3: Controlling graphical output with Perl/Tk and Curses**
- **Lesson 4: Porting *awk*, *sed*, and *find* programs to Perl**
- **Lessons 5, 6, 7, 8: The interface between Perl and C**

X

When writing graphical applications, designers have a lot of choices about appearance (how an image or a window should look) and about behavior (what should happen if the user clicks a mouse button, presses some key, or resizes the window). There are too many decisions to make: The higher the flexibility of a system, the harder that system is to configure. That's why graphics programming, and in particular X programming, is difficult.

Some interface builders and toolkits make some of those decisions for you. That's both good and bad: good because you can develop your applications more quickly, bad because the application will always have the "look and feel" imposed by the toolkit.

Tcl, Tk, X: What's the Difference?

Tk is a Tcl-based toolkit for programming in X.

The X Window System allows programmers to manipulate multiple windows on multiple displays. X comes bundled with most UNIX computers and is available for most other platforms as well. In accordance with UNIX philosophy, X is very general: Every display (monitor) has an *X server* that uses the computer's hardware to draw on the display and to record *events* such as mouse clicks and key presses. *Window managers* allow users to manipulate windows (moving, resizing, iconifying, and so on) that have been launched by X applications, which talk to X servers via TCP/IP. That's why X applications can use displays anywhere on the Internet.

X doesn't specify a particular look and feel for your windows. That's why even a simple X program can be hard to develop--you have to provide the look and feel yourself. X toolkits such as Motif and Tk let programmers write X applications without worrying about the low-level details.

Tk allows programmers to create X applications in Tcl (pronounced "tickle," short for "tool command language"). One of the oldest Tcl applications is *expect*, which is similar in spirit to Perl's *Comm.pl* and *chat2.pl* (see Chapter 13, Lesson 2). But Tk is by far the most popular Tcl application--so popular, in fact, that people often confuse the two. This lesson and the next show you how to use Tk from Perl, bypassing the Tcl layer altogether. (If you're interested in using Tcl from Perl, use Malcolm Beattie's Tcl extension, which lets you create Tcl interpreters as Perl objects. It's available from the CPAN.)

General Perl/Tk Information

There are a few different names of "Tk for Perl" floating around: Perl/Tk, pTk, nTk, and tkperl (the Perl 4 statically linked extension). The most recent effort, led by Nick Ing-Simmons, is called Perl/Tk, but the extension itself is named simply Tk and can be retrieved from the CPAN under *modules/by-module/Tk*. (In *modules/by-module*, you'll also find *Sx* and *Qt*, smaller and simpler packages for using X from Perl.)

Of course, to use any of these packages, you need to be running X on your computer. (If you're not already, you probably don't want to start now. But for the sake of completeness, you can retrieve a copy

of the X window system from <ftp://ftp.x.org>.)

There's a newsgroup for discussion about Perl/Tk: <comp.lang.perl.tk>. There's a mailing list as well; send subscription requests to pTk-request@wpi.edu.

Documentation is available from several sources. The *doc* directory inside the Perl/Tk distribution contains dozens of HTML files. Once installed, the *demos* directory inside *lib/Tk* contains some very helpful sample programs (start with the *widget* demo). Peter Prymmer's [ptkFAQ](#), on the CPAN, is an excellent source of information. Finally, there is a Usenet newsgroup (<comp.lang.perl.tk>), mailing list (pTk-request@wpi.edu), and column in *The Perl Journal* (by Steve Lidie), all dedicated to Perl/Tk.

Simple Tk Programs

Once you have Tk installed, writing X programs is a snap. Listing 14-1 shows *tktest*, a minimal Tk program.

Listing 14-1 *tktest*: A minimal Tk program

```
#!/usr/bin/perl
use Tk;
$stop = new MainWindow; # create a window
MainLoop(); # process X events
```

This creates a blank window. The `MainLoop()` (`Tk::MainLoop()`, to be precise) subroutine handles X events; when you click your mouse or resize the window, `Mainloop()` notices and invokes any callbacks in your program. A *callback* is a subroutine that specifies what should happen when an X event occurs: for instance, when the user presses a button. Let's add one in Listing 14-2.

Listing 14-2 *tkbutton*: Creating a window with a button

```
#!/usr/bin/perl -w
use Tk;
$stop = new MainWindow;
# create a button...
$button = $stop->Button(-text => 'Push here',
    -command => sub { print STDERR "Ouch!\n" } );
# ...and place it on the right side.
$button->pack(-side => 'right');
MainLoop;
```

The `MainWindow Button()` method can accept many instance variables--for a full list, consult the documentation that comes with Perl/Tk. *tkbutton* just uses two: **-text**, which holds the button's label, and **-command**, a reference to a subroutine that will be executed when the button is pressed.

The button's `pack()` method has nothing to do with Perl's `pack()`. For every widget you create (a button is a widget; so are checkboxes and menus and text boxes), you'll use Tk's `pack` to arrange it on the display, for example:

```
$button->pack(-side => 'right')
```

If you resize the window, you'll see that it stays on the right, as in Figure 14-1 (a) and (b).

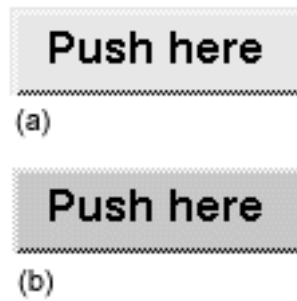


Figure 14-1

A Tk button as it appears (a) when launched and (b) when the window is resized and the cursor is placed over the button

% tkbutton

When you click on the button, Ouch! is printed to the screen from which you ran the program. That's a very simple callback--a more complicated callback might start or stop an animation, pop up a dialog window, or send mail. It's up to you.

Pull-down menus are just buttons, too--see the bounce or widget demos in the lib/Tk/demos directory for a demonstration.

Chapter 14, Perl 5 Interactive Course

LESSON 2

MORE X

So far, you've seen two *widgets*: windows and buttons. This lesson demonstrates a few more widgets in two examples. The first demonstrates a text widget and a dialog box; the second shows how you can animate objects inside a window.

Text Widgets and Dialog Boxes

texteval (Listing 14-3) is an X program that lets users type inside a window. That window is managed by a text widget, which lets you cut and paste text just as you can with any X terminal program. The text widget has an associated *scrollbar widget* that lets users scroll up to view previous input. When users click on the Eval button, whatever text they've typed is `eval()`ed by Perl and the results are printed in the window.

You might remember that `eval()` is a good way to catch errors because a flawed expression merely sets `$@` rather than killing your program. If this happens, *texteval* pops up a dialog box alerting you to the error. If you try to divide by zero, *texteval* displays a different canned message.

Listing 14-3 *texteval*: Creating a text widget and a dialog box

```
#!/usr/bin/perl
use Tk;
use Tk::Dialog;
$stop = MainWindow->new;
$quit_button = $stop->Button(-text => 'Quit', -command =>
\&exit);
$quit_button->pack(-side => 'left');
$eval_button = $stop->Button(-text => 'Eval', -command =>
\&evaluate);
$eval_button->pack(-side => 'left');
$text = $stop->Text(-relief => 'raised', -bd => '2', -setgrid
=> 'true');
$scrollbar = $stop->Scrollbar(-command => ['yview', $text]);
$text->configure(-yscrollcommand => ['set', $scrollbar]);
$scrollbar->pack(-side => 'right', -fill => 'y');
$text->pack (-side => 'right', -expand => 'y', -fill =>
'both');
$text->insert ('1.0', "Type an expression after the prompt:
```

```

\n> ");
MainLoop();
# evaluate() is invoked whenever the user clicks on the Eval
button.
sub evaluate {
my ($string, $backup) = (`, 0);
while ($string !~ /\S/) { # if the user presses [ENTER]
instead of Eval.
$string = $text->get("insert linestart - $backup lines",
`end`);
$backup++;
}
$string =~ s/^\s*>\s+//;
$string =~ s/\s+$//;
$return = eval $string;
if ($@) # There was an error!
if ($@ =~ /division by zero/) {
$dialog = $top->Dialog(-title => `DivZero`, -text =>
`Dividing by zero is a Class III math felony`);
$dialog->show(`-global`);
} else {
$dialog = $top->Dialog(-title => `Error`, -text => "Error:
$@");
$dialog->show(`-global`);
}
} # Place the result in the text window.
$text->insert(`end`, "\n$return\n> ");
}

```

After the main window is created, *texteval* adds two buttons on the left: Quit, which `exit()`s the program, and Eval, which invokes the `evaluate()` subroutine. Then the text and scrollbar widgets are created and placed in the main window. (For a full explanation of the arguments to these and all other widgets, consult the documentation in Tk's *doc* directory--there's only room for a demonstration here.)

When the user's expression (accessed via `$text->get`) is fed to `eval()`, any error causes a dialog box to appear, preventing the user from evaluating any more expressions until he or she clicks on the OK button. (*texteval* doesn't explicitly create that button; it's provided with every dialog box.)

Figure 14-2 shows what *texteval* looks like when operating normally.

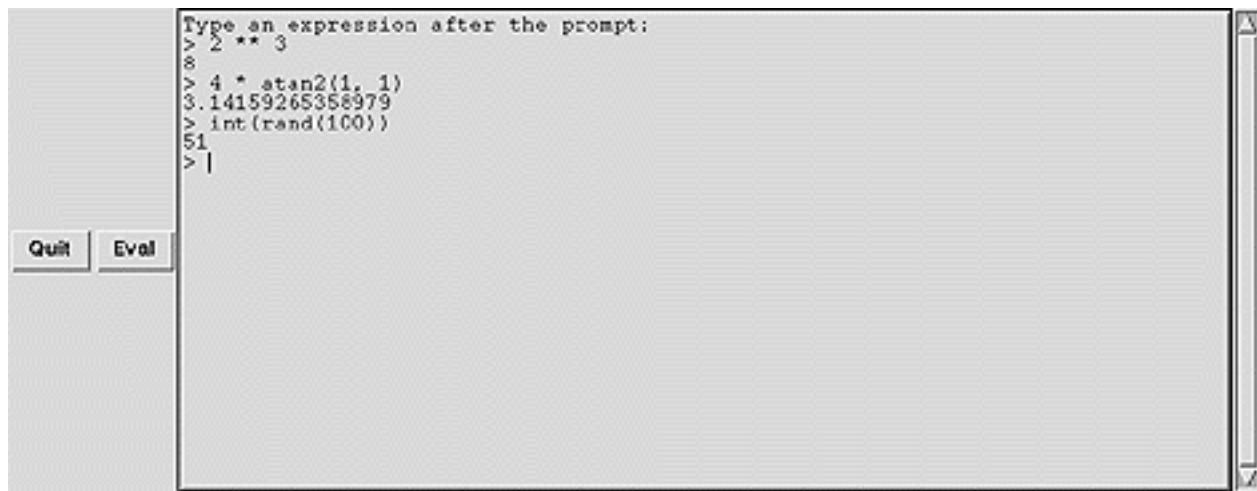


Figure 14-2

texteval: Two buttons, a text widget, and a scrollbar

% texteval

Later (note the scrollbar on the right!), when the user tries to divide by zero (Figure 14-3), a dialog box pops up, shown in Figure 14-4.

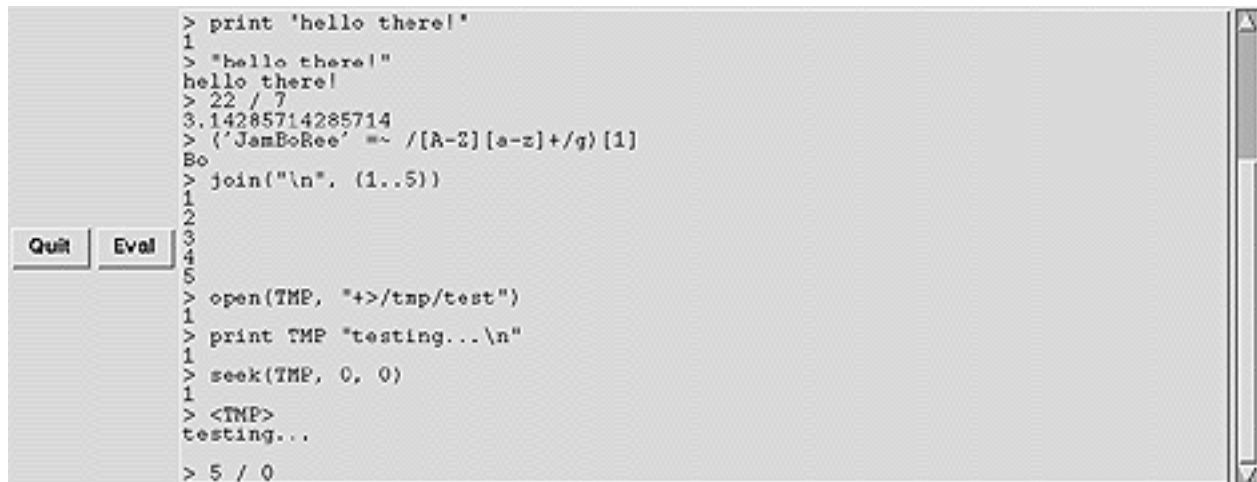


Figure 14-3

texteval, a little while later

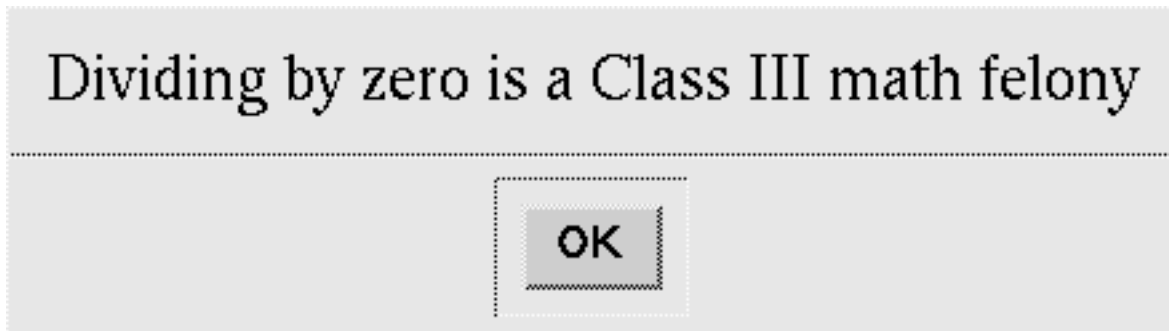


Figure 14-4

A dialog box, raised when an error occurs

Animation

The widgets you've seen are all passive--they change only in response to user actions. Animations, on the other hand, move by themselves. A Tk program that animates objects needs to abandon the `MainLoop()` that sits around waiting for X events so that it can act rather than react.

However, your animation still needs a way to handle X events. The main loop of any animation should be a *while* loop that calls `Tk::DoOneEvent(1)` frequently. That lets the program check for X events when it's not busy animating.

This is demonstrated in the *bugeyes* program (Listing 14-4), based on Gurusamy Sarathy's *bounce* code from *lib/Tk/demos*. *bugeyes* creates an animated face that rolls its eyes and gnashes its teeth. It does this by defining a class, `Eyeteeth`, that provides methods for random motion.

Listing 14-4 *bugeyes*: Animation in X

```
#!/usr/bin/perl -w
use strict;
use Tk;
srand(time ^ ($$ + ($$<15)));
package Eyeteeth;
# @Eyeteeth is a list of all the objects created.
my @Eyeteeth = ();
# Create a new Eyeteeth object.
# Each object is an oval with some color and (x,y) position.
sub new {
my($type, $canvas, $color, $x1, $y1, $x2, $y2) = @_;
my($faceobj, $Eyetooth, $eyetooth_obj);
$Eyetooth = $canvas->create('oval', $x1, $y1, $x2, $y2, -fill
=> $color);
$eyetooth_obj = {eyetooth => $Eyetooth, canvas => $canvas,
color => $color, xpos => ($x1 + $x2) / 2,
ypos => ($y1 + $y2) / 2};
push(@Eyeteeth, $eyetooth_obj);
return bless $eyetooth_obj;
}
# Move the object/oval around.
# $deltax and $deltay specify the movement in the x and y
directions
# The movement rules depend on the color of the oval.
sub move {
my ($eyetooth_obj) = @_;
my ($eyetooth, $canvas, $eyetooth_x, $eyetooth_y, $deltax,
$deltay);
$eyetooth = $eyetooth_obj->{eyetooth};
$canvas = $eyetooth_obj->{canvas};
```



```

$eyetooth_x = $eyetooth_obj->{xpos};
$eyetooth_y = $eyetooth_obj->{ypos};
if ($eyetooth_obj->{color} eq 'white' or
$eyetooth_obj->{color} eq 'yellow') {
$deltax = 0;
$deltay = int(rand(100)) ? 0 : (int(rand(2))) ? 1 : -1;
} elsif ($eyetooth_obj->{color} eq 'gray') {
$deltax = int(rand(5)) - 2;
$deltay = int(rand(3)) - 1;
} elsif ($eyetooth_obj->{color} eq 'red') {
if ($Eyeteeth[0]->{xpos} != $eyetooth_x) {
$deltax = (($Eyeteeth[0]->{xpos} - $eyetooth_x) > 0) ? 1 :
-1;
} else { $deltax = int(rand(7)) - 3; }
if ($Eyeteeth[0]->{ypos} != $eyetooth_y) {
$deltay = (($Eyeteeth[0]->{ypos} - $eyetooth_y) > 0) ? 1 :
-1;
} else { $deltay = int(rand(7)) - 3; }
} elsif ($eyetooth_obj->{color} eq 'blue') {
if ($Eyeteeth[1]->{xpos} != $eyetooth_x) {
$deltax = (($Eyeteeth[1]->{xpos} - $eyetooth_x) > 0) ? 1 :
-1;
} else { $deltax = int(rand(7)) - 3; }
if ($Eyeteeth[1]->{ypos} != $eyetooth_y) {
$deltay = (($Eyeteeth[1]->{ypos} - $eyetooth_y) > 0) ? 1 :
-1;
} else { $deltay = int(rand(7)) - 3; }
}
$canvas->move($eyetooth, $deltax, $deltay);
$eyetooth_obj->{xpos} = $eyetooth_x + $deltax;
$eyetooth_obj->{ypos} = $eyetooth_y + $deltay;
}
# Move all Eyeteeth
sub moveAll {
my ($eyetooth);
foreach $eyetooth (@Eyeteeth) { $eyetooth->move;
Tk::DoOneEvent(1); }
}
package main;
my $stop = new MainWindow;
$stop->title('BugEyes');
# Create the drawing area
my $drawarea = $stop->Frame();
$drawarea->pack(-side => 'top', -fill => 'both');
# Create the canvas
my $canvas = $drawarea->Canvas(-relief => 'ridge', -width =>

```

```

300,
-height => 200,-borderwidth => 4);
$canvas->pack(-side => 'left');
# Create the facial features
my $Face = $canvas->create('oval', 100, 50, 200, 150, -fill
=> 'pink');
my $Mouth = $canvas->create('oval', 127, 120, 173, 140, -fill
=> 'black');
Eyeteeth->new($canvas, 'gray', 120, 80, 140, 100);
Eyeteeth->new($canvas, 'gray', 155, 80, 175, 100);
Eyeteeth->new($canvas, 'red', 125, 85, 135, 95);
Eyeteeth->new($canvas, 'blue', 160, 85, 170, 95);
my $i;
for ($i = 129; $i <= 165; $i += 4) { # the teeth
if (rand() < 0.2) { # make a few teeth yellow
Eyeteeth->new($canvas, 'yellow', $i, 124, $i + 5, 134);
} else { # and the rest white
Eyeteeth->new($canvas, 'white', $i, 124, $i + 5, 134);
}
}
# The main loop
while (1) {
DoOneEvent(1); # wait for other X events
Eyeteeth->moveAll(); # move all the @Eyeteeth.
}

```

There are two big differences between *bugeyes* and previous Tk programs. First, *bugeyes* has scuttled `MainLoop()` in favor of its own *while* loop at the end, which calls `DoOneEvent()`. Second, *bugeyes* defines an `Eyeteeth` class for the facial features that move: the eyes and teeth.

Suppose that *bugeyes* had a menu at the top, allowing the user to drag the cursor down a list of options. `DoOneEvent(1)` lets the animation continue simultaneously. Had `DoOneEvent(0)` been called instead, merely moving the cursor into the menu would have suspended the animation until the next X event arrived (the menu selection).

After the drawing area and canvas are created, *bugeyes* creates a face and a mouth. These don't move, so they don't need to be objects: They're constructed with calls to `$canvas->create()`. Then 14 animated objects are created (with `Eyeteeth->new()`): two eyeballs, two pupils, and ten teeth. When `Eyeteeth->moveAll()` is called from the main *while* loop, each object is moved with `Eyeteeth->move()`. (The motion rules happen to be based solely on the color of the object, to demonstrate how the color attribute can be accessed.) Figure 14-5 shows some snapshots from *bugeyes*.

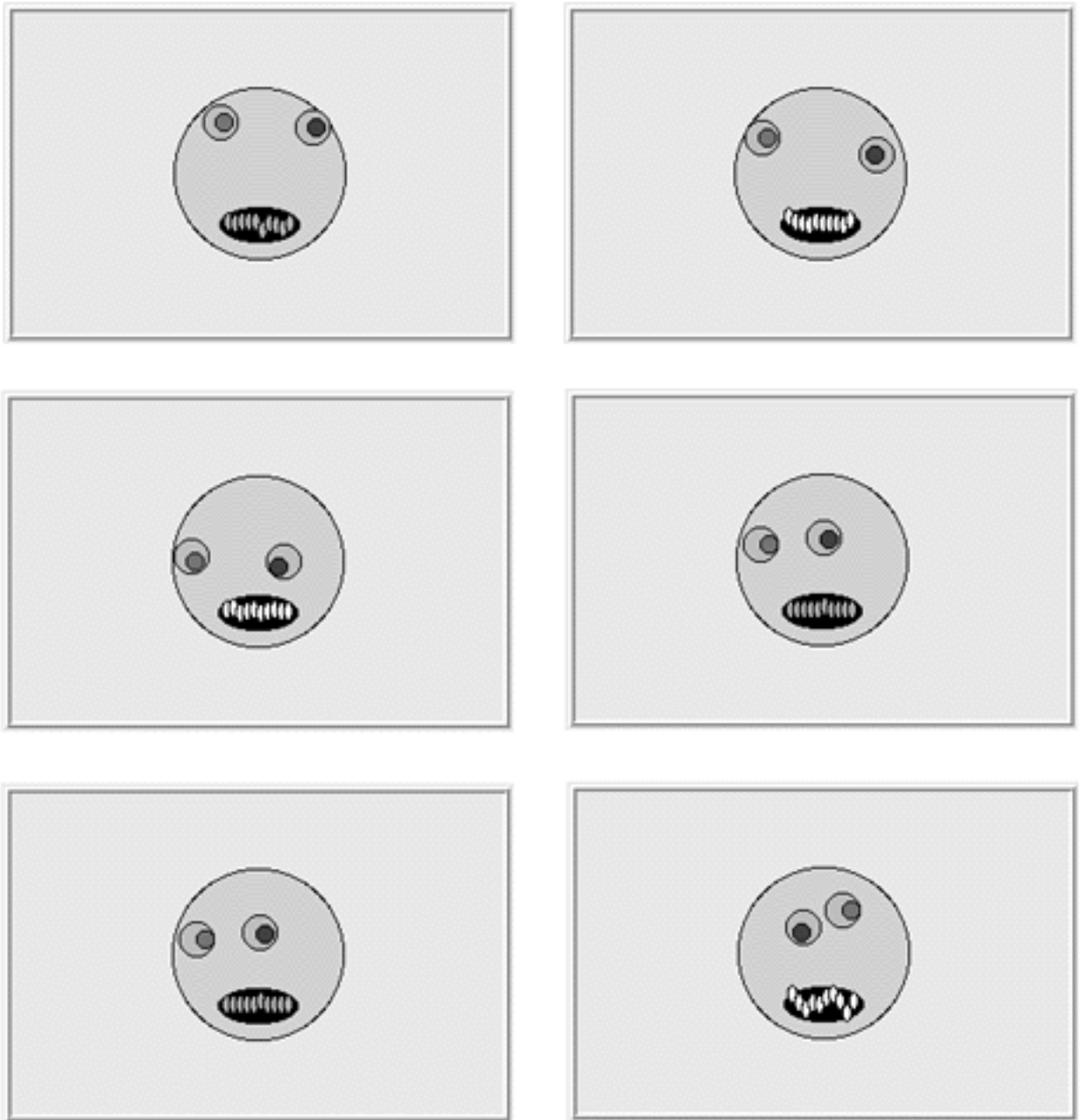


Figure 14-5

Some snapshots from bugeyes

```
% bugeyes
```

Tk Utilities

tkpsh is an interactive shell for Tk applications. Bundled with the Tk distribution, it pops up a window and enters an associated shell. You can then type Perl/Tk commands into the shell and immediately see their effect on the window. A short example:

```
% tkpsh
```

RESULT tkpsh> \$button = \$stop->Button(-text => înow!î) # \$stop is defined for you

```
tkpsh> $button->pack # places button in window
```

tkpsh>

A similar but more powerful utility is *rmt*, in the *lib/Tk/demos* directory. *rmt* attaches a shell to a currently running Tk application of your choice, and lets you apply Tk commands to the application while it's running.

Chapter 14, Perl 5 Interactive Course

LESSON 3

CURSES

William Setzer's *Curses* extension lets you manipulate onscreen ASCII graphics, letting you fetch and store characters at arbitrary places on the user's screen. As you'll see, *Curses* can be used to create ASCII menus, character-based pictures, or even crude animations.

The *Curses* extension is actually an interface to the *Curses* library, written in C. *Curses* comes bundled with most UNIX distributions, but unfortunately there is no "standard" *Curses* library; the available functions vary from system to system, and identical functions sometimes have different names on different systems. The two examples in this lesson play it safe: They use only the most common *Curses* functions. Consult your online documentation (e.g., *man Curses*) to see which functions are supported on your computer, or look at the *Curses.h* file generated automatically by the extension.

Curses represents screen positions height first, with (0,0) in the upper left corner: The position (20, 5) is 20 rows down from the top of the screen and 5 columns across from the left side.

The Perl 4 version of *Curses* was called *curseperl*; you can use *curseperl* scripts with the *Curses* module if you add this line to the beginning of your program:

```
BEGIN { $Curses::OldCurses = 1; }
```

Listing 14-5 shows *maze*, a simple *Curses* program that lets users draw on the screen by maneuvering the cursor left, right, up, and down. All the work is done through straightforward function calls; however, there is an object-oriented interface, should you care to use it.

Listing 14-5 *maze*: Using Curses to print characters on the screen

```
#!/usr/bin/perl -w

use Curses;

initscr; # initialize the screen

$b = subwin(40, 40, 3, 3); # Create a 40x40 window

noecho; # don't print user keystrokes

cbreak; # grab input by characters, not lines

clear; # clears the screen

$x = $y = 20;
```

```

while ($ch != 5) {
move($y, $x); # move the cursor
addch('#'); # add a character
refresh; # show the result
$ch = getch; # get the next character typed
if ($ch == 2) { $y++ }
elseif ($ch == 4) { $x-- }
elseif ($ch == 6) { $x++ }
elseif ($ch == 8) { $y-- }
}

echo; # Let the user see his keystrokes again.

```

When you run *maze*, the cursor starts in the middle of the Curses window, created by `subwin()`. After readying the screen with `noecho()`, `cbreak()`, and `clear()`, the *while* loop prints a # in the middle of the subwindow at position (20, 20), shows the result with `refresh()`, and then waits for users to press a key. If that key is 2, 4, 6, or 8, the cursor is moved and a new hash mark is printed at the beginning of the next iteration. Users can thus move around the screen, merrily leaving a trail of hash marks, until they press 5, which exits the *while* loop and turns `echo()`ing back on (Figure 14-6)

```

% maze
#####
#                                     #
#                                     #
#                                     #
#                                     #
#####                               #####
#                                     #
#                                     #
#                                     #
#                                     #
#                                     #
#                                     #
#                                     #
#                                     #
#                                     #
#                                     #
#####                               #####

```

Figure 14-6

Sample output from maze

If users exit *maze* by any means other than pressing 5 (such as interrupting the program with [CTRL]-[C]), the final `echo()` won't be triggered and users won't be able to see anything they

subsequently type, even after *maze* exits. In that case, users should type **stty echo** from the UNIX prompt to turn echoing back on, or you should modify the program so that `$SIG{INT}` executes an `echo()` before exiting (see Chapter 9, Lesson 2).

You can place an ASCII string on the screen with `addstr()`, which is just like `addch()` except it takes a string as an argument instead of a single character. You can make strings or characters appear to move by erasing them and then printing them in an adjacent position.

tetris0, shown in Listing 14-6, contains some building blocks for a Tetris-like game that animates ASCII graphics. A `select()` with a timeout has the program wait for a fraction of a second (how long is determined by `$speed`) before moving the current *tetris0* piece toward the bottom of the screen.

tetris0 is really a pale approximation of a true Tetris game. Lines can't be "completed" and every shape is a line of four stars (but that's the shape you're always looking for anyway). You can move the shape left with 4 or right with 6, and you can flip (from horizontal to vertical and back) with 5.

Note that *tetris0* uses *no* arrays or hashes to represent the board. In fact, there's no explicit representation of the board at all! This was done to demonstrate `inch()`, which returns the character at a given point on the screen. In effect, *tetris0* uses the screen itself as a data structure.

Listing 14-6 *tetris0*: Using Curses to "query" the screen

```
#!/usr/bin/perl

use Curses;

initscr;

$b = subwin(40, 40, 0, 20);

$x = 20;

$horiz = 1;

# set $speed to $ARGV[0] if it exists, 10 otherwise
$speed = shift or 10;

# let users print after [CTRL]-[C]
$SIG{INT} = sub {echo; exit};

noecho;

cbreak;

clear;

move(40, 0);

addstr( `*****` );

# Each iteration of this outer loop starts a new shape at the
```

```

top.

while (! $end_of_game) {

$y = 0;

# Each iteration of this inner loop moves the shape closer to
the bottom.

while ($y < 40) {

$y++;

vec($rbits, fileno(STDIN), 1) = 1;

# Wait for 1/$speed seconds.

if (select($rbits, undef, undef, 1 / $speed)) {

$ch = getch;

if ($ch eq `4`) { $x-- }

elsif ($ch eq `6`) { $x++ }

elsif ($ch eq `5`) { $horiz = (($horiz == 1) ? 0 : 1) }

}

last if add_shape(); # if add_shape() returns 1,
} # the top was reached-
} # Game Over!

echo;

sub add_shape {

my ($i, $j);

if ($erase) { # Erase previous line
if ($old_horiz) {
move($old_y, $old_x - $length / 2);
addstr(' ');
} else {
for ($i = -2; $i < 2; $i++) {
move($old_y + $i, $old_x);
addch(' ');

```



```

}
}
}
$erase = 0; # Add current line
if ($horiz) { # Is the line horizontal?
move($y, $x - $length/2);
addstr('****');
foreach $i (($x - $length/2)..($x - $length/2)+3) {
if (inch($y+1, $i) eq '*') { # Collision! ($erase not set)
$end_of_game = 1 if $y < 2;
return 1;
}
}
} else { # The line is vertical.
for ($i = -2; $i < 2; $i++) {
move($y + $i, $x);
addch('*'); # add the line, star by star
}
if (inch($y+2, $x) eq '*') { # Collision! ($erase not set)
$end_of_game = 1 if ($y < 2);
return 1;
}
}
refresh;
$erase = 1;
$old_y = $y;
$old_x = $x;
$old_horiz = $horiz;

```

```
return 0; # return 0 if no collision.
```

```
}
```

Note the signal handler at the beginning of *tetris0*. That's to ensure that [CTRL]-[C]s don't leave the user unable to type.

The two nested *while* loops manage the bulk of the program: Every iteration of the outer *while* loop starts a new shape at the top of the screen, and every iteration of the inner *while* loop moves that shape down one line by calling the *add_shape()* subroutine. *add_shape()* first overwrites the old shape with a string of spaces--usually--and then prints the new shape. It also checks to see whether the current shape has collided with another shape, in which case it returns 1. When that happens, *\$erase* is left at 0 so that the shape isn't erased on the next call to *add_shape()*, and the outer *while* loop launches a new shape from the top of the screen. When the shapes pile up high enough, *\$end_of_game* is set and *tetris0* exits (Figure 14-7).

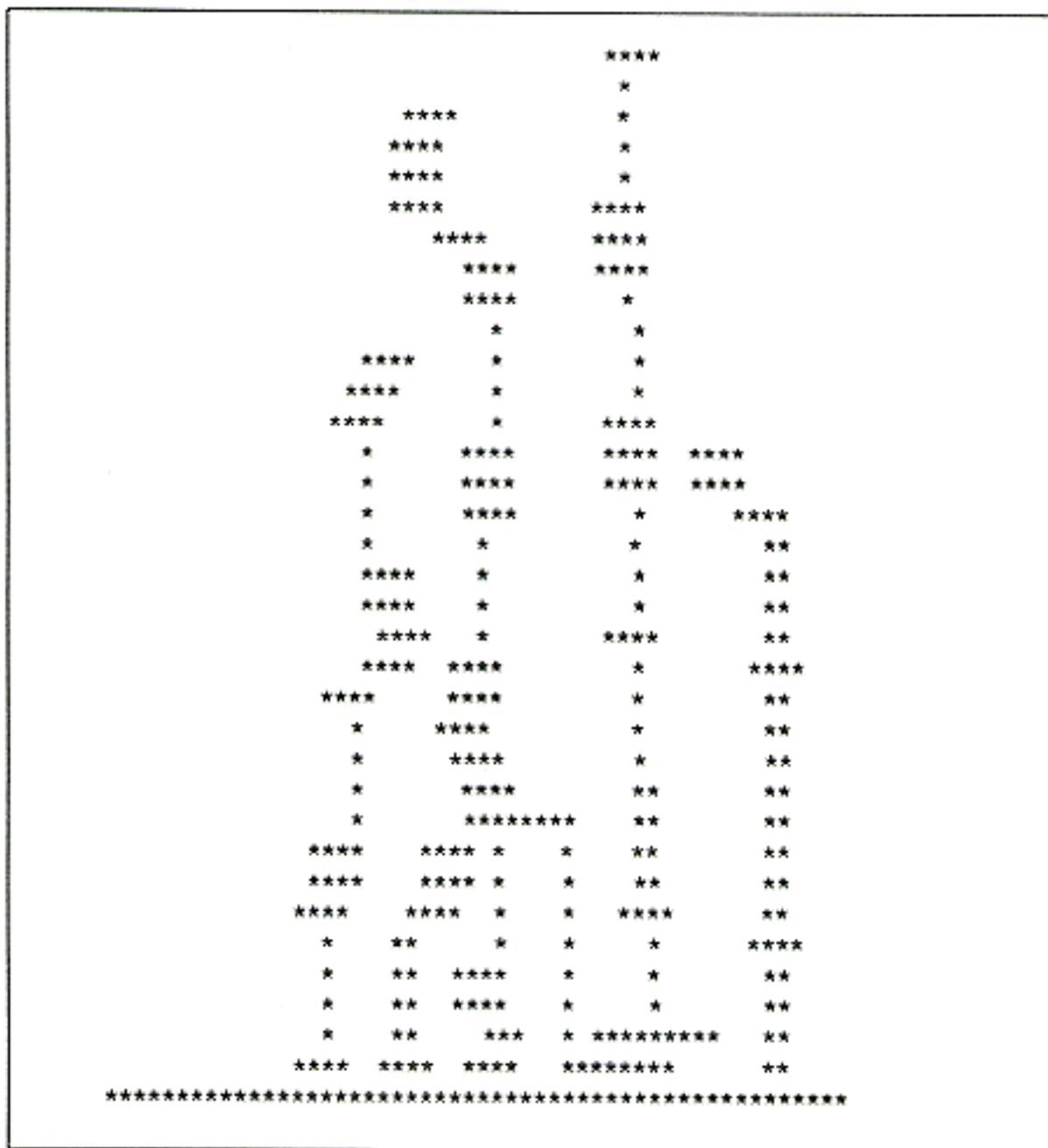


Figure 14-7

The end of a tetris() game

The *menu.pl* package (available from the CPAN) automates *Curses*' full-screen menus. It's an "old-style" *Curses* file, which means that you need to include the magic BEGIN to use it (see Listing 14-7).

Listing 14-7 *menu*: Using the menu.pl library file with Curses

```
#!/usr/bin/perl -w

BEGIN { $Curses::OldCurses = 1; } # because it's a curseperl
program

use Curses;

require 'menu.pl';

# Create the menu

menu_init(1, "What are your display requirements?");

# Add three items.

menu_item("No graphics, low bandwidth, extreme portability",
"text");

menu_item("Clunky graphics, low bandwidth, easy to develop",
"Curses");

menu_item("Nice graphics, high bandwidth, hard to develop",
"X");

# Display the menu and prompt the user

$sel = menu_display("Pick one!");

print "You selected $sel.\n";
```

menu produces a keyboard-navigable menu.

```
% menu
```

RESULT What are your display requirements?

```
-> 1) No graphics, low bandwidth, extreme portability
```

```
2) Clunky graphics, low bandwidth, easy to develop
```

```
3) Nice graphics, high bandwidth, hard to develop
```

```
(All) Pick one!
```

You'll be able to maneuver the arrow up and down with the arrow keys on your keyboard. If you hit down four times (the arrow wraps around) and then hit [ENTER], you'll see

RESULT You selected Curses.

Chapter 14, Perl 5 Interactive Course

LESSON 4

UNIX AWK, SED, AND FIND

The UNIX utilities *awk* and *sed* are becoming less popular every day. Every now and again, you'll encounter an old UNIX diehard who claims that if something can't be done in *awk*, *sed*, or *find*, it isn't worth doing. Chances are, someone like that has written quite a few programs, some of which you might actually find useful. In that case, you can convert the programs to Perl. There's no programming in this lesson--just demonstrations of the conversion programs *a2p*, *s2p*, and *find2perl*, all of which come bundled with the Perl distribution in the directory *x2p*. If Perl has been installed correctly at your site, there should be online documentation available for *a2p* and *s2p*. If this is the first time you've seen *awk* and *sed* code, you'll recognize some traits that have influenced Perl's design.

awk

awk is a utility for finding lines, or parts of lines, in data. There are several different flavors of *awk*, and *a2p* doesn't always work on some of the more unusual features. Here's how you could use *awk* to print all lines in *tetris0* that are longer than 60 characters:

```
% awk 'length > 60' tetris0
```

For an *awk* task this simple, you wouldn't have any trouble rewriting it in Perl. But let's run it through *a2p* and see what happens (Listing 14-8).

Listing 14-8 *a2p* output

```
% echo 'length > 60' | a2p
#!/usr/bin/perl
eval "exec /usr/bin/perl -S $0 $*"
if $running_under_some_shell;
# this emulates #! processing on NIH machines.
# (remove #! line above if indigestible)
eval `$.=$1.$2;` while $ARGV[0] =~ /^[A-Za-z_0-9]+=)(.*)/
&& shift;
# process any FOO=bar switches
while (<>) {
```

```

chop; # strip record separator

print $_ if length($_) > 60;

}

```

Hmmm...what's up with that second `eval()`? When *awk* is invoked from the command line, it lets you assign values to variables that might be used later in the script. This example doesn't do that, but *a2p* tacks it on anyway. And in case you were wondering, yes, those comments were generated automatically. Listing 14-9 shows a more complicated *awk* program: *awksum*. It expects a series of lines, each containing two fields separated by a comma. *awksum* then inverts the order of the fields, printing the second field first and first field second, and then adds up the first column, printing the sum and the average.

Listing 14-9 *awksum*: An *awk* script

```

BEGIN { FS = "," }

{ print $2, $1 }

{ s += $1 }

END { print "sum is", s, "average is", s/NR }

```

You can apply *a2p* directly to this file, as shown in Listing 14-10.

Listing 14-10 *a2p* output

```

% a2p awksum

#!/usr/bin/perl

eval "exec /usr/bin/perl -S $0 $*"

if $running_under_some_shell;

# this emulates #! processing on NIH machines.

# (remove #! line above if indigestible)

eval `$. $1. $2;` while $ARGV[0] =~ /^([A-Za-z_0-9]+=)(.*)/
&& shift;

# process any FOO=bar switches

$, = ` `; # set output field separator

$\ = "\n"; # set output record separator

$FS = ` `;

while (<>) {

```

```

($F1d1,$F1d2) = split(/[,\\n]/, $_, 9999);
print $F1d2, $F1d1;
$S += $F1d1;
}

print `sum is`, $S, `average is`, $S / $.;

```

The online documentation for *a2p* has some additional tips for speeding up the output programs.

sed

The *sed* (short for "stream editor") utility modifies and prints lines. It's often used to make replacements in a chunk of text. The most common *sed* operator is *s///*, which inspired Perl's *s///*.

```
% sed 's/l/u/g' textfile
```

replaces all of the l's in *textfile* with u's, printing the result to standard output. That *sed* command can be converted to Perl with *s2p* (Listing 14-11).

Listing 14-11 s2p output

```

% echo 's/l/u/g' | s2p

#!/usr/bin/perl

eval `exec /usr/bin/perl -S $0 ${1+"$@"}`
if $running_under_some_shell;
while ($ARGV[0] =~ /^-/) {
    $_ = shift;
    last if /^--/;
    if (/^-n/) {
        $nflag++;
    }
    next;
}
die "I don't recognize this switch: $_\\n";
}
$printit++ unless $nflag;
$_ = "\\n"; # automatically add newline on print
LINE:

```

```

while (<>) {
    chop;
    s/l/u/g;
    print if $printit;
}

```

Listing 14-12 shows a more complicated *sed* script that joins every line ending in a backslash to the following line.

Listing 14-12 *sedjoin*: A sed script

```

:join
/\$/{N
s/\\n//
b join
}

```

When processed by *s2p*, *sedjoin*'s *join* label becomes a Perl label named *Join*, as shown in Listing 14-13.

Listing 14-13 *s2p* output

```

% s2p sedjoin
#!/usr/bin/perl
eval `exec /usr/bin/perl -S $0 ${1+"$@"}`
if $running_under_some_shell;
while ($ARGV[0] =~ /^-/) {
    $_ = shift;
    last if /^--/;
    if (/^-n/) {
        $nflag++;
    }
    next;
}
die "I don't recognize this switch: $_\n";
}

```



```

$printit++ unless $nflag;

$\ = "\n"; # automatically add newline on print

LINE:
while (chop($_ = <>)) {
Join:
    if (/\\$/ ) {
        $_ .= "\n";
        $len1 = length;
        $_ .= <>;
        chop if $len1 < length;
        s/\\$//;
        redo LINE;
    }
    print if $printit;
}

```

s2p is itself a Perl script, so you can see how Larry Wall programs in Perl--and how his programs program.

find

As you saw in Chapter 8, the `File::Find` module lets you search recursively through a directory for files that fulfill some condition, mimicking the UNIX *find* command. To convert find expressions to Perl, use the *find2perl* program. Here's a simple use of find:

```
% find . -name core -print
```

This detects and prints all files named *core* in (or beneath) the current directory. (As a Perl programmer, it's your right to sneer at C programmers whose *core* files clutter your filesystem.)

Listing 14-14 shows what happens when you use *find2perl* to convert that line to Perl. Unlike *a2p* and *s2p*, you don't use quotes on the command line.

Listing 14-14 *find2perl* output

```

% find2perl /users -name core -print

#!/usr/bin/perl

eval `exec /usr/bin/perl -S $0 ${1+"$@"}`

```

```

if $running_under_some_shell;
require "find.pl";
# Traverse desired filesystems
&find(`.`);
exit;

sub wanted {
/^core$/ &&
print("$name\n");
}

```

Fairly straightforward: *find2perl* just feeds `.` to `find()`. Note that it uses *find.pl* instead of `File::Find`; that's to ensure Perl 4 compatibility. The `find()` in *find.pl* uses a `wanted()` subroutine; *find2perl* defines `wanted()` so that it prints the complete file (including the pathname) if the file is named *core*. Listing 14-15 shows a more sophisticated *find* expression. It removes all the files named *a.out* or ending in *.o* that haven't been accessed for a week and aren't mounted via *nfs*. (That would be our friendly C programmers dirtying up the filesystem again.)

Listing 14-15 *find2perl* output

```

% find2perl / \( -name a.out -o -name '*.o' \) -atime +7
-exec rm {} \; -o *
-fstype nfs -prune

#!/usr/bin/perl

eval `exec /usr/bin/perl -S $0 ${1+"$@"}`

if $running_under_some_shell;
require "find.pl";
# Traverse desired filesystems
&find(`/`);
exit;

sub wanted {
{
/^a\.out$/
||
/^.*\.o$/
}
}

```

```
} &&

($nlink || (($dev,$ino,$mode,$nlink,$uid,$gid) = lstat($_)))
&&

(int(-A _) > 7) &&

(unlink($_) || warn "$name: $!\n")

||

($nlink || (($dev,$ino,$mode,$nlink,$uid,$gid) = lstat($_)))
&&

($dev < 0) &&

($prune = 1);

}
```

Chapter 14, Perl 5 Interactive Course

LESSON 5

USING C FROM PERL

C is fast. And there's a lot of C code out there. Wouldn't it be nice if you could put *use C* at the top of your Perl script and then freely intermix C and Perl code? Unfortunately, life isn't that simple. Perl and C are different in so many ways that there needs to be a language, XS (eXternal Subroutines), that bridges them. The bulk of this lesson and the next discusses the interface between C and Perl. A working knowledge of C is assumed.

The Perl Compiler

Before we discuss the C/Perl interface, there are a couple of topics that advanced Perl programmers should know about. The first is Malcolm Beattie's Perl compiler, which can be retrieved from the CPAN under *modules/by-module/B*. Not to be confused with Perl's *internal* compiler, which is what interprets and runs your Perl program, Malcolm's compiler is an answer to criticisms of the form "Why isn't Perl as fast as C?" and "Is there a Perl-to-C translator?" The compiler can be used in three ways:

- Converting Perl programs into C source code
- Converting Perl programs into a platform-independent bytecode
- Converting Perl programs into a platform-dependent machine code

We'll demonstrate one use of the compiler with this simple Perl script, called *alphabet.pl*:

```
#!/usr/bin/perl
for ( 'a' .. 'z' ) { print $_ }
```

First, we'll translate the code into C source code and redirect the output into the new file *alphabet.c*:

```
% perl -MO=C alphabet.pl > alphabet.c
```

which creates *alphabet.c*, a three-hundred-plus line C program. You can then compile that program into an executable binary with the *cc_harness* utility provided with the compiler distribution:

```
% perl cc_harness -o alphabet.out alphabet.c
```

(With some non-ANSI C compilers, you may have to add *-DBROKEN_STATIC_REDECL* between *alphabet.out* and *alphabet.c*.)

Finally the speed of C can be unleashed in all its glory:

```
% alphabet.out
```

```
RESULT abcdefghijklmnopqrstuvwxyz
```

For further details about the compiler, see the documentation included with the distribution, or Malcolm's article in issue 2 of *The Perl Journal*.

The C Preprocessor

cpp, the UNIX C preprocessor, is applied to most C programs as the first stage of compilation, using preprocessor directives (`#if`, `#define`, and so forth) to define macros and include (or exclude) sections of code. It's often used to manage nonportable portions of C programs. There's not much need for a separate preprocessing stage in Perl, as Perl already has BEGIN blocks and the Config module, and Perl is substantially more portable than C anyway. But just so you know, the `-P` and `-I` command-line flags let you embed preprocessor directives in your Perl programs.

The `-P` and `-I` Command-Line Flags

- `-P` runs your Perl program through the C preprocessor.
- `-I` tells the C preprocessor where to look for include files.

Because both C preprocessor commands and Perl comments begin with `#`, you should avoid starting comments with anything that might be mistaken for a preprocessor directive, such as `#if` or `#else`.

The *whichsys* program (Listing 14-16) uses the C preprocessor to identify the operating system (although the Config module does a much better job).

Listing 14-16 *whichsys*: Using the `-P` command-line flag

```
#!/usr/bin/perl -wPl
#ifdef unix
print "I'm on a UNIX machine.";
#elif VMS
print "I'm on a VMS machine.";
#elif MSDOS || _Windows || _WINDOWS || OS2
print "I'm on a PC, probably.";
#endif
#ifdef unix
#ifdef sparc
print "...in particular, a SparcStation.";
#elif linux
print "...in particular, a Linux box.";
#endif
#endif
```

On a Sparc, the C preprocessor defines the `sparc` token.

```
% whichsys
```

RESULT I'm on a UNIX machine.

```
...in particular, a SparcStation.
```

The C preprocessor can also be invoked as a source filter.

Source Filtering

There's one last topic before we tackle XS: source filtering, which lets you filter your Perl program through something else before it's compiled. That "something else" can be another program, or a Perl subroutine of your own choosing.

One of the powers of UNIX is the ability to pipe data through a succession of programs. For instance, if you have a compressed file (called, say, *data.Z*), you can apply programs in succession, cascading their input and output.

```
% cat data.Z | uncompress | sort | uniq
```

sends *data.Z* as input to *uncompress*, which sends its output to *sort*, which sends its output to *uniq*. You do the same thing whenever you use `|` with Perl's `open()`.

Perl's source filtering, developed by Paul Marquess and Tim Bunce, lets you pass source code (your Perl script) through another program (which might be a UNIX utility, a C program of your own design, or another Perl program) before reaching Perl's internal compiler. You invoke source filters just like regular modules.

```
#!/usr/bin/perl
use Filter::decrypt;
...
```

invokes the `decrypt` source filter. There are currently six source filters:

- **Filter::decrypt**, which lets you encrypt your source code (using an encryption algorithm you provide or a simple algorithm provided by `Filter::decrypt`)
- **Filter::tee**, which emulates the UNIX `tee` program, copying your Perl program to an external file
- **Filter::call**, which runs your Perl program through a Perl subroutine in your program
- **Filter::exec**, which runs your Perl program through an external program
- **Filter::cpp**, which (like the `-P` flag) runs your Perl program through the C preprocessor
- **Filter::sh**, which passes your Perl program through an `sh` (shell) command

Each source filter is its own module, which includes embedded pod documentation. More source filters are currently being developed and will trickle into future Perl releases.

Using C Functions and Libraries

The remainder of this chapter investigates using XS to manipulate C functions from Perl and Perl functions from C. It assumes you can write C programs and compile them on your computer.

Both Perl/Tk and *Curses* are examples of using C from Perl. You might not have known, because XS insulated you from the fact that you were using C libraries instead of bona fide Perl modules. This lesson and the next show some examples of how you can link C code into Perl. It's a tricky subject; this chapter is only an introduction. For more information, consult Dean Roehrich's *perlxs* documentation and Jeff

Extensions

Extensions are modules that let you use C code from Perl. They can be *dynamically* linked, which means that you can load in compiled C code (as Perl subroutines) on the fly, or *statically* linked, which means that you create a new */usr/bin/perl* that includes the compiled C code. (Perl 4 supports only static linking, using the now-obsolete file *upperl.o*.)

Static linking is more cumbersome because you need to recompile all of Perl whenever you want to add another function. But it results in a faster start-up time.

In Perl 5, creating an extension begins with the *h2xs* program.

```
% h2xs -A -n Your_Extension_Name
```

This creates a subdirectory for your extension containing mostly empty files ready for customization. This lesson (and the next) describes how to flesh them out to create your extension.

Once you've created your extension, you (and anyone else) can install it as follows.

```
% cd Your_Extension_Name
```

The *Your_Extension_Name* directory can be anywhere you wish. (Earlier versions of Perl required that they be inside the Perl distribution. That's no longer the case.)

```
% perl Makefile.PL
```

creates a platform-dependent Makefile for the extension. (This is accomplished by the MakeMaker utility bundled with Perl.)

```
% make
```

compiles your extension, choosing dynamic linking if your system supports it and static linking otherwise. You can override Perl's choice by saying *make dynamic* or *make static* instead.

```
% make test
```

tests the compiled code--assuming the extension's author created a test suite.

```
% make install
```

installs the extension. (If you're unlucky, you might need to *make install* inside your Perl distribution as well.)

This lesson guides you through an extension that makes a factorial function, written in C, available to your Perl programs. The first step is creating an XS subroutine that describes the C function to Perl.

XS

XS is a simple language bridging C and Perl. If you want to use a C function from Perl, you first create an XSUB, which is an XS subroutine that includes information such as the parameter types and return values of the corresponding C function. Most XSUBs are short because they don't replace C functions--they just tell Perl how to call them.

For examples of XS code, consult the *ext* directory in your Perl distribution--any file whose name ends in

`.xs` contains XS code. In particular, `ext/POSIX/POSIX.xs` contains a wealth of tiny XSUBs.

Linking a C Function into Perl

The first step in making our C factorial function available to Perl is creating an empty Factorial extension by running `h2xs`.

```
% h2xs -A -n Factorial
```

```
RESULT Writing Factorial/Factorial.pm
```

```
Writing Factorial/Factorial.xs
```

```
Writing Factorial/Makefile.PL
```

```
Writing Factorial/test.pl
```

```
Writing Factorial/Changes
```

```
Writing Factorial/MANIFEST
```

Then `cd` to the newly created Factorial directory and look at *Factorial.xs*. Eventually, it will contain your C code for the factorial function, but when created, it just looks like Listing 14-17.

Listing 14-17 The XS template

```
#ifdef __cplusplus
extern "C" {
#endif
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
#ifdef __cplusplus
}
#endif
MODULE = Factorial PACKAGE = Factorial
```

C functions can be linked in as entire libraries, or simply inserted just after the `#include` statements in the XS template. After the `MODULE` line, add the three lines exactly as shown in bold in Listing 14-18. (Make sure there's no whitespace between the `n` and its parentheses on the second-to-last line.)

Listing 14-18 *Factorial.xs*: The *Factorial.xs* file, which now contains an XSUB for factorial()

```
#ifdef __cplusplus
extern "C" {
#endif
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
#ifdef __cplusplus
}
#endif
```



```

static int factorial (int n) {
if (n < 2) return 1;
return n * factorial(n - 1);
}
MODULE = Factorial PACKAGE = Factorial
int
factorial(n)
int n

```

You've now written an XSUB, which tells Perl how to pass an argument to factorial (that's the *int n* on the last line) and what to expect in return (an integer; that's the first *int* on a line by itself). When there are multiple arguments, each is specified on its own line, for example:

```

float
addtwo(x, y)
float x
float y

```

There's a lot more to XS than is shown here; the next lesson delves into some more features, including techniques for manipulating the stack. (In this context, a *stack* is a last-in, first-out queue used to store memory addresses and variable values. It contains the values passed between Perl subroutines and their corresponding C functions.)

But let's follow through with our Factorial extension. The next step is to modify the *Factorial.pm* file, which will eventually be installed in the LIB directory for everyone to use. Listing 14-19 shows the *Factorial.pm* template created by *h2xs*.

Listing 14-19 The *Factorial.pm* template

```

package Factorial;
use strict;
use vars qw($VERSION @ISA @EXPORT);
require Exporter;
require DynaLoader;
@ISA = qw(Exporter DynaLoader);
# Items to export into callers namespace by default. Note: Do
# not export
# names by default without a very good reason. Use EXPORT_OK
# instead.
# Do not simply export all your public
# functions/methods/constants.
@EXPORT = qw(
);
$VERSION = '0.01';
bootstrap Factorial $VERSION;
# Preloaded methods go here.
# Autoload methods go after __END__, and are processed by the
# autosplit program

```

```

1;
__END__
# Below is the stub of documentation for your module. You
better edit it!
=head1 NAME
Factorial - Perl extension for blah blah blah
=head1 SYNOPSIS
use Factorial;
blah blah blah
=head1 DESCRIPTION
Stub documentation for Factorial was created by h2xs. It
looks like the
author of the extension was negligent enough to leave the
stub
unedited.
Blah blah blah.
=head1 AUTHOR
A. U. Thor, a.u.thor@a.galaxy.far.far.away
=head1 SEE ALSO
perl(1)
=cut

```

As written, the *Factorial.pm* template doesn't export any functions or create any classes, so it's not very useful. Because you've just defined a `factorial()` XSUB, you can make it accessible to your Perl programs by adding it to `@EXPORT`, as shown in Listing 14-20.

Listing 14-20 *Factorial.pm*, which now exports `factorial()`

```

package Factorial;
use strict;
use vars qw($VERSION @ISA @EXPORT);
require Exporter;
require DynaLoader;
@ISA = qw(Exporter DynaLoader);
# Items to export into callers namespace by default. Note: Do
not export
# names by default without a very good reason. Use EXPORT_OK
instead.
# Do not simply export all your public
functions/methods/constants.
@EXPORT = qw( factorial );
$VERSION = '0.01';
bootstrap Factorial $VERSION;
# Preloaded methods go here.
# Autoload methods go after __END__, and are processed by the

```

```
autosplit program.  
1;  
__END__  
...
```

There's also a *Makefile.PL* file inside the *ext/Factorial* directory, which creates a Makefile tailored for compiling the Factorial extension on your system. You're now ready to use it.

```
% perl Makefile.PL
```

```
RESULT Checking if your kit is complete...
```

```
Looks good
```

```
Writing Makefile for Factorial
```

Once that Makefile has been created, type *make* to compile your *.xs* code into C code:

```
% make
```

Then (you might need to be a superuser to do this)

```
% make install
```

to copy *Factorial.pm* to the *LIB* directory at your site.

Now you'll be able to use the C *factorial()* function from your own Perl scripts. (Curious readers may want to look at the C code generated by the XS compiler as *Factorial.c*.)

Listing 14-21 shows a sample program to verify that your Factorial extension has been successfully installed.

Listing 14-21 *facttest*: Testing the Factorial extension

```
#!/usr/bin/perl  
use Factorial;  
print factorial(6);
```

Lo and behold:

```
% facttest
```

```
RESULT 720
```

For more information on the topics covered in this lesson (and the next), consult the *perlxstut* and *perlxs* documentation bundled with Perl.

Linking a C Library into Perl

factorial.xs is self-contained: It defines the single C function exported by *Factorial.pm*. But the original motivation behind XS was to let Perl programmers use entire libraries of C code, not just individual functions.

Suppose you want to use the standard C math library. C has a built-in *tan()* function; Perl doesn't (although you can always just divide *sin()* by *cos()*, or use *POSIX::tan()*). But for the sake of argument, let's say only C's *tan()* will do. We'll create an extension, *CMath*, that lets you access C's *tan()* from your

Perl programs. First, invoke *h2xs*, supplying the *math.h* header file on your system (wherever it may be) as an argument.

```
% h2xs -O -n CMath < /usr/include/math.h
```

This creates a (large!) *CMath.pm* file that exports every constant and every function declared in *math.h* on your system. Because you're interested only in *tan()*, you can delete everything else from *@EXPORT*, leaving only

```
@EXPORT = qw( tan );
```

If you wanted to use the entire C math library from Perl, you wouldn't remove anything--though you might want to move functions whose names overlap with Perl (such as *sin()* and *cos()*) from *@EXPORT* to *@EXPORT_OK*. Extensions can provide as many functions as you wish; you just have to be willing to write XSUBs for all of them.

You'll need to add *#include <math.h>* to your *CMath.xs* immediately after the other preprocessor directives at the top of the file, and the following XSUB at the end:

```
double  
tan(x)  
double x
```

You also need to tell Perl to link in the library containing the definition of *tan()*. On UNIX systems, that library is called simply *m*; add it to the *Makefile.PL* file as shown in bold:

```
use ExtUtils::MakeMaker;  
# See lib/ExtUtils/MakeMaker.pm for details of how to  
influence  
# the contents of the Makefile that is written  
WriteMakefile(  
  'NAME' => 'CMath',  
  'VERSION_FROM' => 'CMath.pm', # Finds $VERSION  
  'LIBS' => ['-lm'], # e.g., '-lm'  
  'DEFINE' => '', # e.g., '-DHAVE_SOMETHING'  
  'INC' => '', # e.g., '-I/usr/include/other'
```

Now you can install your new *CMath* extension following the same procedure as *Factorial*:

```
% perl Makefile.PL  
% make  
% make install
```

and then *tan()* away:

```
% perl -e 'use CMath; print tan 0'
```

```
RESULT 0
```

```
% perl -e 'use CMath; print tan(atan2(1,1))'
```

```
RESULT 1
```

Chapter 14, Perl 5 Interactive Course

LESSON 6

ADVANCED XS

The XSUBs for `factorial()` and `tan()` are simple--one numeric input, one numeric output. This lesson shows some XS files that

- **Modify an input argument.**
- **Contain strings.**
- **Contain more than one XSUB.**
- **Use a C struct.**

Modifying Input Arguments and Passing Strings to an XSUB

If an XSUB modifies one of its arguments, it's polite (and generally necessary) to notify *xsubpp* with the `OUTPUT` keyword, as shown in *Letter.xs* (Listing 14-22).

Listing 14-22 *Letter.xs*: Using the `OUTPUT` keyword to identify a modified argument

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
static void change_letter(char *s) {
    s[0] = 'c';
}
MODULE = Letter PACKAGE = Letter
void
change_letter(s)
char * s
OUTPUT:
s
```

`change_letter()` is declared as `void` because it has no output. But it does modify its input string by setting the first character to `c`. Perl needs to know this, so the XSUB notifies it with the `OUTPUT` keyword, followed by the parameter that's being modified: `s`.

After installation, you can change *bar* to *car* with the following:

```
% perl -e 'use Letter; $word = "bar"; change_letter($word);  
print $word'
```

RESULT car

OUTPUT is one of a slew of XS keywords that you can use to customize XSUBs. Here are the currently available keywords (more are on the way: INPUT, PREINIT, INIT, and ALIAS). You've already seen a few of them.

- **MODULE** specifies the name of the module. It must be the first keyword in every XS file.
- **PACKAGE** specifies the name of the package inside the module. It's used to divide functions into separate packages.
- **PREFIX** designates a common prefix of the C function names that should be removed from the corresponding Perl functions. For instance, if you're accessing a C library called **Account** that declares an `account_read()` function, Perl users shouldn't need to invoke `Account::account_read()` when `Account::read()` is just as informative.
- **CODE** is used to define particularly tricky segments of C code. Often, CODE fragments push arguments onto the stack and pop return values off the stack.
- **PPCODE** is an alternative form of CODE used when the XSUB needs specific instructions for controlling the argument stack. For instance, if an XSUB needs to return a list of values, PPCODE is used to push the list of values onto the stack explicitly.
- **RETVAL** is the return value of the C function. All C functions (except those returning void) have an implicit RETVAL; if you don't explicitly provide one, xsubpp provides one for you. RETVAL is often used in tandem with CODE to specify exactly what is being returned. By default, that return value is placed in ST(0), the top of the stack.
- **OUTPUT** indicates that the listed parameters are modified by the function.
- **NO_INIT** indicates that a function parameter is used only as an output value.
- **BOOT** adds code to the extension's bootstrap function, which is generated by xsubpp.
- **CLEANUP** identifies code to be executed after any CODE, PPCODE, or OUTPUT blocks.

Consult the *perlxs* documentation for more detailed information on these keywords.

Multiple XSUBs

Perl's `stat()` does everything that the various C `stat()`s do. But let's create an interface to C's `stat()` anyway.

```
% h2xs -A -n MyStat sys/stat.h
```

We'll add two of our own functions to *MyStat.xs*: `block_size()`, which returns the number of bytes in each I/O block, and `file_size()`, which returns the number of bytes in a file. Add `block_size()` and `file_size()` to *MyStat.pm*'s @EXPORT and add the text in bold to *MyStat.xs*, as shown in Listing 14-23.

Listing 14-23 *MyStat.xs*: Adding C functions to an XS file

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
#include <sys/stat.h>
static int block_size() {
return S_BLKSIZE;
}
static int file_size(char *filename) {
struct stat b;
stat(filename, &b);
return b.st_size;
}
MODULE = MyStat PACKAGE = MyStat
int
block_size()
int
file_size(filename)
char * filename
```

Note that `block_size()` uses the C constant `S_BLKSIZE` and `file_size()` uses the C `stat` structure, even though neither is visible to the Perl program.

```
% perl -e 'use MyStat; print block_size'
```

RESULT 512

```
% perl -e 'use MyStat; print file_size("chatter")'
```

RESULT 5546

The CODE Keyword

The `CODE` keyword lets you specify how the argument stack should be manipulated. Listing 14-24 is a short example that computes the square of a number using XS code without ever invoking a true C function. Because XS functions can return only one type, we'll restrict the example to integers.

Listing 14-24 *Square.xs*: Using `CODE` to manipulate stack values

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
MODULE = Square PACKAGE = Square
int
square(number)
int number
CODE:
{
```



```
ST(0) = sv_newmortal();
sv_setiv(ST(0), number * number);
}
```

The `sv_setiv()` function ("set scalar value to integer value;" see the *perlguts* documentation) is used to place **number * number** directly onto the stack: `ST(0)`. (If you wanted to return a floating-point number, you'd use `sv_setnv()` instead--see Lesson 8.)

```
% perl -e 'use Square; print square 5'
```

```
RESULT 25
```

```
% perl -e 'use Square; print square 7.8'
```

```
RESULT 49
```

Types and the typemap

The `MyStat` extension contains the `file_size()` function, which uses C's `stat` structure. But what about C's actual `stat()` function? If Perl had no `stat()`, you'd want to create an XSUB for C's `stat()` that returns a C `stat` structure. But Perl doesn't have explicit structs like C. (In Chapter 13, you saw that Perl emulated C's `sockaddr` structure with a `pack()`ed string.)

In XS, you need to specify explicitly how C structures should be translated into Perl. A file called the *typemap* (*lib/ExtUtils/typemap*, to be precise) does most of the work for you--that's what tells Perl that the thing called an "int" is an integer (and therefore a scalar) and that a "`*char`" is a string (and therefore also a scalar).

Of course, it's not always clear how to map a C type into Perl. For instance, how should an `*int` be translated? As a reference? As an array? Should the `stat` structure be a packed string? An array? An object? A reference? It's up to you. You can specify a mapping between C and Perl types in a custom *typemap* that you create inside your extension directory. *xsubpp* then consults your *typemap* before it uses the default *typemap*. Consult the *perlxs* and *perlxstut* documentation for more information.

In the default *typemap*, you'll see that an `int` becomes a `T_IV` (an integer; a type of scalar), a `*char` becomes a `T_PV` (a string; also a scalar), and a `*void` becomes a `T_PTR` (a pointer; yet another scalar).

Future versions of *xsubpp* will probably support C structures and arrays in a more straightforward manner and allow Perl programs to manipulate individual fields of the structure (or elements of the array).

Chapter 14, Perl 5 Interactive Course

LESSON 7

USING PERL FROM C

Lessons 5 and 6 discussed how you can use C from Perl. This lesson and the next discuss the complementary scenario: how to use Perl from C. There are all sorts of reasons to do this: regular expression (or even string) manipulation, on-the-fly evaluation with `eval()`, formats, or any of the other features Perl possesses that C lacks. See the *perlembed* documentation, by Doug MacEachern and Jon Orwant, bundled with the Perl distribution.

At long last, there is now an external Perl compiler that can generate C source code (see the section on "The Perl Compiler" in Lesson 5). Why did it take so long that Tom Christiansen offered a free laptop to its creator? Part of the problem is that Perl and C are really very different languages. Consider Perl's `eval()`--any program that uses `eval()` needs to contain the entire Perl interpreter.

But that's not quite as ghastly as it sounds. You can embed Perl in C, creating a Perl interpreter that you can use to evaluate Perl expressions and to call Perl subroutines. It's not a full-blown `/usr/bin/perl`, however: The communication between your C program and the Perl interpreter is quite constrained.

Let's look at a minimal program, *minterp*, (Listing 14-25) that creates and uses a Perl interpreter.

Listing 14-25 *minterp.c*: A Perl interpreter embedded in a C program

```
#include <stdio.h>
#include <EXTERN.h>
#include <perl.h> /* perl.h is in the Perl distribution. */
static void xs_init _((void));
static PerlInterpreter *my_perl; /* the interpreter */
int main (int argc, char **argv, char **env)
{
    int status;
    my_perl = perl_alloc();
    perl_construct (my_perl);
    status = perl_parse (my_perl, NULL, argc, argv, (char**)
    NULL);
    if (status) exit(status);
    status = perl_run (my_perl);
    perl_destruct (my_perl);
    perl_free (my_perl);
    exit(status);
}
```

```
static void xs_init() {}
```

To compile *minterp*, you need to link in the Perl library *libperl.a*, which was created when your copy of Perl was compiled. There should be a copy in *LIB/your_architecture/CORE*. If that directory happens to be */usr/local/lib/perl5/alpha-dec_osf/CORE* (as it might be if you're running Perl on a DEC Alpha), you could compile *minterp* as follows:

```
% cc -o minterp minterp.c
-L/usr/local/lib/perl5/alpha-dec_osf/CORE
-I/usr/local/lib/perl5/alpha-dec_osf/CORE -lperl -lm
```

You may need to supply additional flags to your C compiler on other operating systems. But if all goes well, you'll have a working Perl interpreter:

```
% minterp
```

```
RESULT print "3 * 4 IS ", 3 * 4
```

```
[CTRL]-[D]
```

```
3 * 4 is 12
```

When you run *minterp*, you'll be able to type any Perl expression you like. It's just like typing plain old **perl** from your shell: When you end standard input, the interpreter evaluates what you've typed.

You could read Perl expressions from a file as well by placing the filename in **argv[1]** before the call to `perl_run()`. Or, to call a Perl subroutine you've defined, you can bypass `perl_run()` entirely and use `perl_call_argv()` instead.

In *hello.c* (Listing 14-26), we'll use `perl_call_argv()` to feed a subroutine from an external Perl file (*world.pl*) into our interpreter.

Listing 14-26 *hello.c*: Making an external Perl script available to a Perl interpreter

```
#include <stdio.h>
#include <EXTERN.h>
#include <perl.h>
static void xs_init _((void));
static PerlInterpreter *my_perl;
int main (int argc, char **argv)
{
    int status;
    my_perl = perl_alloc();
    perl_construct( my_perl );
    status = perl_parse(my_perl, xs_init, argc, argv, NULL);
    if (status) exit(status);

    /* call the Perl greet() subroutine */
    perl_call_argv("greet", G_DISCARD | G_NOARGS, argv);
    perl_destruct( my_perl );
    perl_free( my_perl );
}
```

```
exit(status);
}
static void xs_init() {}
```

hello.c expects a `greet()` subroutine in whichever Perl file is supplied on the command line. Let's oblige with the last *Hello world!* program of the book, shown in Listing 14-27.

Listing 14-27 world.pl: A Perl subroutine to be embedded in C

```
print "I will NOT be printed.";
sub greet {
print "Hello world!\n";
}
```

Note that *world.pl* does not have a `#!/usr/bin/perl` line. There doesn't have to be because `/usr/bin/perl` isn't needed--the interpreter is embedded in the C program.

```
% cc -o hello hello.c
-L/usr/local/lib/perl5/your_architecture/CORE
-I/usr/local/lib/perl5/your_architecture/CORE -lperl -lm
% hello world.pl
```

RESULT Hello world!

Note that the string *I will NOT be printed.* was truthful. That's because the interpreter wasn't executing *world.pl*--it just made the `greet()` function available to the C program.

The `greet()` function accepted no arguments and returned no values--those tasks require manipulating the stack. You saw a glimpse of that in the last lesson; the next (and final!) lesson shows you how to do it from C.

Chapter 14, Perl 5 Interactive Course

LESSON 8

USING THE PERL STACK FROM C

The previous lesson showed you how to pass data from C into a Perl interpreter. But it didn't cover the return trip: how to access Perl's return values from C. In this lesson, you'll see how to call Perl subroutines directly using Perl's argument stack. For more thorough coverage, consult Jeff Okamoto's *perlguts* and Paul Marquess' *perlcall* documentation.

Playing with the Stack

The Perl library provides a number of functions (described in *perlguts*) for creating stack variables and for pushing them onto and popping them off the stack.

To Perl, integers, floats, and strings are all just scalars. But to C, they're all distinct types, which is why there are three separate functions for converting C data types to scalars:

```
SV* newSViv(IV integer)
SV* newSVnv(double number)
SV* newSVpv(char *string, int length)
```

Each function creates a Perl scalar value out of a typed C value: an integer, a double, and a fixed-length string, respectively. (An IV is a special type defined to be large enough to hold both an integer and a pointer.) These functions all return a pointer to a newly allocated SV, which is Perl's internal representation for a scalar. To access the contents of an SV, use the macros:

```
SvIV(SV*)
SvNV(SV*)
SvPV(SV*, STRLEN length)
```

to retrieve scalar values as integers, doubles, or *chars, respectively.

You can change the values of an already allocated SV with the functions

```
void sv_setiv(SV* scalar, IV integer)
void sv_setnv(SV* scalar, double number)
void sv_setpv(SV* scalar, char* string)
void sv_setpvn(SV* scalar, char* string, int new_length)
```

What about arrays and hashes? They have their own types: AV and HV. Perl arrays can be created with either of two functions:

```
AV* newAV( )
AV* av_make(I32 num_scalars, SV **scalars)
```

`newAV()` creates an empty array; `av_make()` creates an array filled with the scalars provided in the `SV**` scalars. Once these are created, you can manipulate arrays with several functions, such as

```
void av_push(AV* array, SV* scalar)
SV* av_pop(AV* array)
SV** av_fetch(AV* array, I32 key, I32 lvalue)
SV** av_store(AV* array, I32 key, SV* value)
```

All these functions do what you'd expect except for `av_fetch()`, which stores an undef value at index `key` if `lvalue` is nonzero.

Hashes are a bit trickier. There's a `newHV()` for creating hashes, an `hv_store()` for modifying hash values, and an `hv_fetch()` for accessing stored values. Consult the *perlguts* documentation for more information.

internal.c, shown in Listing 14-28, uses some of the SV and AV functions. Normally, these functions are used to manipulate stack values with a Perl interpreter; here, they're demonstrated in isolation.

Listing 14-28 *internal.c*: Accessing Perl's argument stack from C

```
#include <stdio.h>
#include <EXTERN.h>
#include <perl.h>
int main (int argc, char **argv, char **env)
{ /* Create 6 scalars and 1 array */
SV *num, *real, *string, *output0, *output1, *output2;
AV *array;
num = newSViv(5); /* Create a new integer scalar */
printf ("Scalar value: %d\n", SvIV(num));
sv_setiv(num, 7); /* Set it to 7 */
printf ("\t.....set to %d\n", SvIV(num));
array = newAV(); /* Create a new (empty) array */

num = newSViv(88888); /* Create a new integer scalar */
real = newSVnv(4.669201); /* Create a new float scalar */
string = newSVpv("Lucid", 5); /* Create a new string scalar
*/
av_push(array, num); /* Push them all onto array */
av_push(array, real);
av_push(array, string);
output2 = av_pop(array); /* Pop them all off */
output1 = av_pop(array);
output0 = av_pop(array);
/* Access all the values. "na" is passed to SvPV() to
indicate that the */
/* length of the string doesn't matter. */
printf("Popped %s\n", (char *) SvPV(output2, na));
printf("Popped %f\n", (double) SvNV(output1));
printf("Popped %d\n", (int) SvIV(output0));
```

```
}
```

The *internal.c* program is straightforward once you get used to the notion that you have to use Perl's internal functions to fetch and store all Perl variables and that integers, floating-point numbers (Perl always uses doubles, not floats), and strings are all treated differently, even though the distinction is hidden from you in Perl.

Because C null-terminates its strings, you often need to specify the length of a string when retrieving it via SvPV(). In *internal*, the global variable *na* is passed to SvPV() to indicate that you don't care about the string length.

```
% cc -o internal internal.c -L/usr/source/perl  
-I/usr/source/perl -lperl -lm
```

```
RESULT % internal
```

```
Scalar Value: 5  
.....set to 7  
Popped Lucid  
Popped 4.669201  
Popped 88888
```

The perl_call_ Functions

Okay, now that you've got your variables, how can you pass them into Perl subroutines? There are four functions (discussed in *perlcall*) that let you call Perl subroutines from C:

- **perl_call_sv(SV *sv, I32 flags)**
- **perl_call_pv(char *sub_name, I32 flags)**
- **perl_call_method(char *method_name, I32 flags)**
- **perl_call_argv(char *sub_name, I32 flags, register char **argv)**

All these functions return an I32, which is (more or less) a 32-bit integer. You probably won't want to use perl_call_sv() unless you're doing some pretty heavy wizardry; the other three functions (which all invoke perl_call_sv() themselves) are easier to use because they let you specify Perl subroutines by name.

The flags argument can contain any of the five following flags, ORed together:

- **G_SCALAR** calls the subroutine in a scalar context.
- **G_ARRAY** calls the subroutine in a list context.
- **G_DISCARD** throws away the return value.
- **G_NOARGS** reuses old stack values rather than supplying new values.
- **G_EVAL** wraps an eval() around the subroutine, for trapping fatal errors.

Because C has no built-in function for integer exponentiation, let's make Perl's ****** operator available to a C program. First, create a stub function in Perl, as shown in Listing 14-29.

Listing 14-29 power.pl: A stub function

```
sub expo {  
my($a, $b) = @_;  
return $a ** $b;  
}
```

Listing 14-30 shows *power.c*, derived from *interp1.c* with the addition of one subroutine, `PerlPower()`, that manipulates the stack values. Explaining all the stack functions is beyond the scope of this book; see *perlcalls*.

Listing 14-30 power.c: Using a Perl subroutine from a C program

```
#include <stdio.h>  
#include <EXTERN.h>  
#include <perl.h>  
static void xs_init __((void));  
static PerlInterpreter *my_perl;  
static void  
PerlPower(int a, int b)  
{  
dSP; /* initialize stack pointer */  
ENTER; /* enter a Perl block: "{" */  
SAVETMPS; /* create scope for temp variables */  
PUSHMARK(sp); /* remember the stack pointer */  
XPUSHs(sv_2mortal(newSViv(a))); /* push the base onto the  
stack */  
XPUSHs(sv_2mortal(newSViv(b))); /* push the exponent onto the  
stack */  
PUTBACK; /* make local stack pointer global */  
perl_call_pv("expo", G_SCALAR); /* call the function */  
SPAGAIN; /* refresh stack pointer */  
/* POP the return Integer */  
printf ("%d to the %dth power is %d.\n", a, b, POPI);  
PUTBACK; /* refresh stack pointer after POP */  
FREETMPS; /* free temporary variables */  
LEAVE; /* leave Perl block: "}" */  
}  
int main (int argc, char **argv, char **env)  
{  
my_perl = perl_alloc();  
perl_construct( my_perl );  
perl_parse(my_perl, xs_init, argc, argv, env);
```



```

PerlPower(3, 4); /* Compute and print 3 ** 4 */
perl_destruct(my_perl);
perl_free(my_perl);
}
static void xs_init() {}

```

Now the Perl subroutine `expo()` can be invoked via the C subroutine `PerlPower()`.

When the arguments *a* and *b* are converted to SVs, they are made "mortal" with `sv_2mortal()` so that FREETMPS can deallocate their memory. Otherwise, your C program would leak memory with every call to `PerlPower()`.

```

% cc -o power power.c -L/usr/source/perl -I/usr/source/perl
-lperl -lm
% power power.pl

```

RESULT 3 to the 4th power is 81.

The call to `perl_call_pv()` hardcodes the name of the subroutine: "expo." If the subroutine were to be chosen dynamically, `perl_call_sv()` would be required.

More information is available in *my perlembed* documentation, bundled with the Perl distribution. *perlembed* is divided into nine sections:

- **Compiling your C program**
- **Adding a Perl interpreter to your C program**
- **Calling a Perl subroutine from your C program**
- **Evaluating a Perl statement from your C program**
- **Performing Perl pattern matches and substitutions from your C program**
- **Fiddling with the Perl stack from your C program**
- **Maintaining a persistent interpreter**
- **Maintaining multiple interpreter instances**
- **Using Perl modules, which themselves use C libraries, from your C program**

Here's your last quiz. Enjoy!

Review

Meryl: I'm satisfied-I finally have the ammunition to answer that frequent criticism about Perl: "It's not as fast as C." Well, now I can just link into C libraries when I feel the need for speed.

Jeremy: Or you can translate your Perl programs into C with the Perl compiler. Think we'll ever see similar linkages between Perl and other languages?

Meryl: Possibly, but I wouldn't count on it. Perl, UNIX, and C are all deeply intertwined; the connection between Perl and FORTRAN or Pascal or LISP is more tenuous. I'd expect a stringer bridge between Perl and C++ built first.