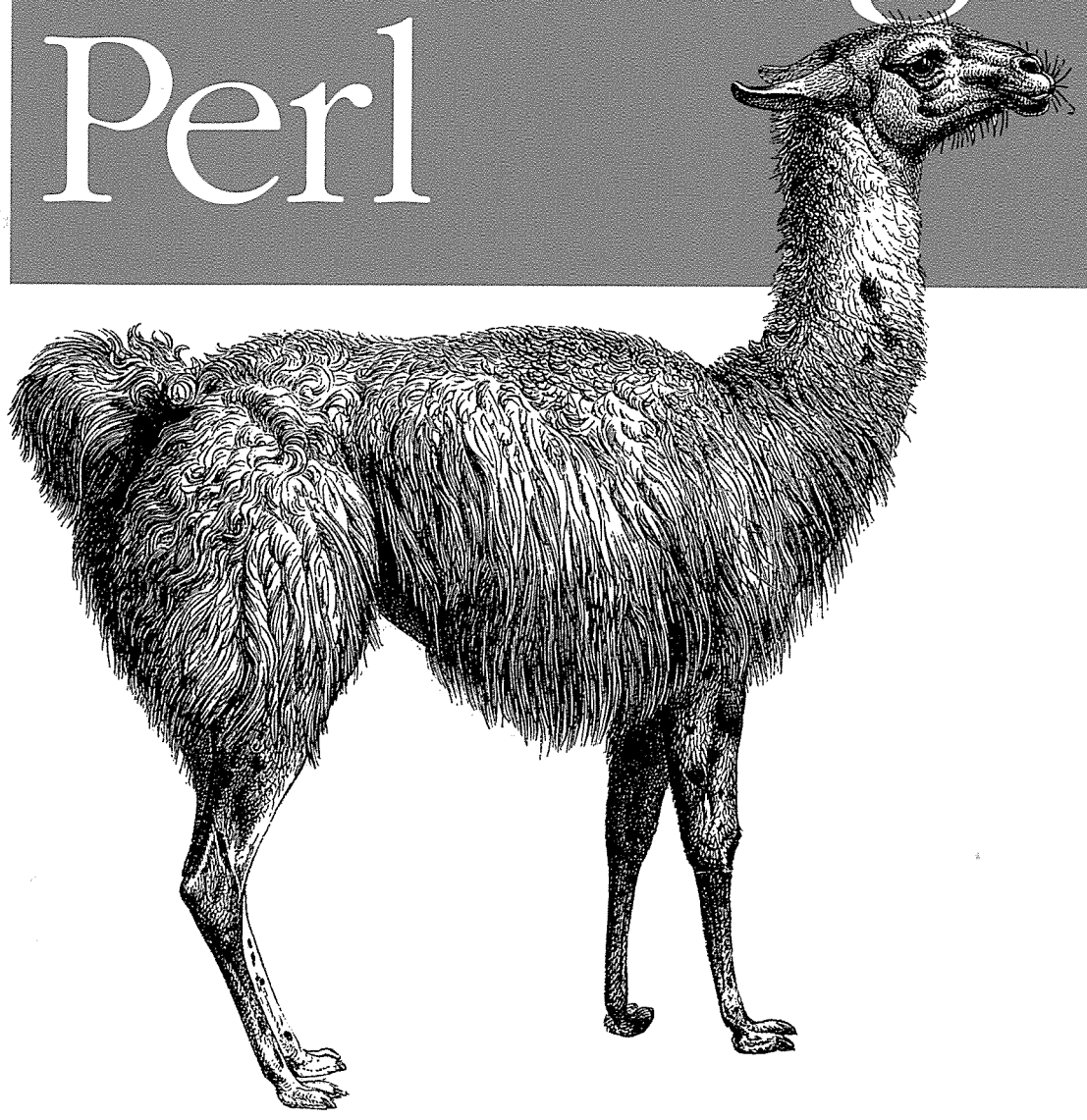
UNIX Programming

2nd Edition

Learning Perl



Randal L. Schwartz & Tom Christiansen
Foreword by Larry Wall

O'REILLY®

Learning Perl

Second Edition

Randal L. Schwartz
and Tom Christiansen

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Paris · Sebastopol · Taipei · Tokyo

Learning Perl, Second Edition
by Randal L. Schwartz and Tom Christiansen

Published by O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472.

Editor: Steve Talbott

Production Editor: Mary Anne Weeks Mayo

Printing History:

November 1993:	First Edition.
April 1994:	Minor corrections.
August 1994:	Minor corrections.
July 1997:	Second Edition.

ISBN: 1-56592-284-0 [3/00]

[M]

Table of Contents

In this chapter:
- *History of Perl*
- *Purpose of Perl*
- *Availability*
- *Basic Concepts*
- *A Stroll Through Perl*

1

Introduction

History of Perl

Perl is short for "*P*ractical *E*xtraction and *R*eport *L*anguage," although it has also been called a "*P*athologically *E*clectic *R*ubbish *L*ister." There's no point in arguing which one is more correct, because both are endorsed by Larry Wall, Perl's creator and chief architect, implementor, and maintainer. He created Perl when he was trying to produce some reports from a Usenet-news-like hierarchy of files for a bug-reporting system, and *awk* ran out of steam. Larry, being the lazy programmer that he is, decided to over-kill the problem with a general-purpose tool that he could use in at least one other place. The result was the first version of Perl.

After playing with this version of Perl a bit, adding stuff here and there, Larry released it to the community of Usenet readers, commonly known as "the Net." The users on this ragtag fugitive fleet of systems around the world (tens of thousands of them) gave him feedback, asking for ways to do this, that, or the other, many of which Larry had never envisioned his little Perl handling.

But as a result, Perl grew, and grew, and grew, at about the same rate as the UNIX operating system. (For you newcomers, the entire UNIX kernel used to fit in 32K! And now we're lucky if we can get it in under a few meg.) It grew in features. It grew in portability. What was once a little language now had over a thousand pages of documentation split across dozens of different manpages, a 600-page Nutshell reference book, a handful of Usenet newsgroups with 200,000 subscribers, and now this gentle introduction.

Larry is no longer the sole maintainer of Perl, but retains his executive title of chief architect. And Perl is still growing.

This book was tested with Perl version 5.0 patchlevel 4 (the most recent release as I write this). Everything here should work with 5.0 and future releases of Perl. In fact, Perl 1.0 programs work rather well with recent releases, except for a few odd changes made necessary in the name of progress.

Purpose of Perl

Perl is designed to assist the programmer with common tasks that are probably too heavy or too portability-sensitive for the shell, and yet too weird or short-lived or complicated to code in C or some other UNIX glue language.

Once you become familiar with Perl, you may find yourself spending less time trying to get shell quoting (or C declarations) right, and more time reading Usenet news and downhill snowboarding, because Perl is a great tool for leverage. Perl's powerful constructs allow you to create (with minimal fuss) some very cool one-up solutions or general tools. Also, you can drag those tools along to your next job, because Perl is highly portable and readily available, so you'll have even *more* time there to read Usenet news and annoy your friends at karaoke bars.

Like any language, Perl can be "write-only"; it's possible to write programs that are impossible to read. But with proper care, you can avoid this common accusation. Yes, sometimes Perl looks like line noise to the uninitiated, but to the seasoned Perl programmer, it looks like checksummed line noise with a mission in life. If you follow the guidelines of this book, your programs should be easy to read and easy to maintain, but they probably won't win any obfuscated Perl contests.

Availability

If you get

```
perl: not found
```

when you try to invoke Perl from the shell, your system administrator hasn't caught the fever yet. But even if it's not on your system, you can get it for free (or nearly so).

Perl is distributed under the GNU Public License,* which says something like, "you can distribute binaries of Perl only if you make the source code available at no cost, and if you modify Perl, you have to distribute the source to your modifications as well." And that's essentially free. You can get the source to Perl for the cost of a blank tape or a few megabytes over a wire. And no one can lock Perl

* Or the slightly more liberal Artistic License, found in the distribution sources.

up and sell you just binaries for their particular idea of "supported hardware configurations."

In fact, it's not only free, but it runs rather nicely on nearly everything that calls itself UNIX or UNIX-like and has a C compiler. This is because the package comes with an arcane configuration script called *Configure* that pokes and prods the system directories looking for things it requires, and adjusts the include files and defined symbols accordingly, turning to you for verification of its findings.

Besides UNIX or UNIX-like systems, people have also been addicted enough to Perl to port it to the Amiga, the Atari ST, the Macintosh family, VMS, OS/2, even MS/DOS and Windows NT and Windows 95—and probably even more by the time you read this. The sources for Perl (and many precompiled binaries for non-UNIX architectures) are available from the Comprehensive Perl Archive Network (the CPAN). If you are web-savvy, visit *http://www.perl.com/CPAN* for one of the many mirrors. If you're absolutely stumped, write *bookquestions@oreilly.com* and say "Where can I get Perl?!?!"

Basic Concepts

A shell script is nothing more than a sequence of shell commands stuffed into a text file. The file is then "made executable" by turning on the execute bit (via *chmod +x filename*) and then the name of the file is typed at a shell prompt. Bingo, one shell program. For example, a script to run the *date* command followed by the *who* command can be created and executed like this:

```
% echo date >somescript
% echo who >>somescript
% cat somescript
date
who
% chmod +x somescript
% somescript
[output of date followed by who]
%
```

Similarly, a Perl program is a bunch of Perl statements and definitions thrown into a file. You then turn on the execute bit* and type the name of the file at a shell prompt. However, the file has to indicate that this is a Perl program and not a shell program, so you need an additional step.

Most of the time, this step involves placing the line

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

* On UNIX systems, that is. For directions on how to render your scripts executable on non-UNIX systems, see the Perl FAQ or your port's release notes.

as the first line of the file. But if your Perl is stuck in some nonstandard place, or your system doesn't understand the `#!` line, you'll have a little more work to do. Check with your Perl installer about this. The examples in this book assume that you use this common mechanism.

Perl is mostly a free-format language like C—whitespace between tokens (elements of the program, like `print` or `+`) is optional, unless two tokens put together can be mistaken for another token, in which case whitespace of some kind is mandatory. (Whitespace consists of spaces, tabs, newlines, returns, or formfeeds.) There are a few constructs that require a certain kind of whitespace in a certain place, but they'll be pointed out when we get to them. You can assume that the kind and amount of whitespace between tokens is otherwise arbitrary.

Although nearly any Perl program can be written all on one line, typically a Perl program is indented much like a C program, with nested parts of statements indented more than the surrounding parts. You'll see plenty of examples showing a typical indentation style throughout this book.

Just like a shell script, a Perl program consists of all of the Perl statements of the file taken collectively as one big routine to execute. There's no concept of a "main" routine as in C.

Perl comments are like (modern) shell comments. Anything from an unquoted pound sign (`#`) to the end of the line is a comment. There are no C-like multiline comments.

Unlike most shells (but like *awk* and *sed*), the Perl interpreter completely parses and compiles the program into an internal format before executing any of it. This means that you can never get a syntax error from the program once the program has started, and that the whitespace and comments simply disappear and won't slow the program down. This compilation phase ensures the rapid execution of Perl operations once it is started, and it provides additional motivation for dropping C as a systems utility language merely on the grounds that C is compiled.

This compilation does take time; it's inefficient to have a voluminous Perl program that does one small quick task (out of many potential tasks) and then exits, because the run-time for the program will be dwarfed by the compile-time.

So Perl is like a compiler and an interpreter. It's a compiler because the program is completely read and parsed before the first statement is executed. It's an interpreter because there is no object code sitting around filling up disk space. In some ways, it's the best of both worlds. Admittedly, a caching of the compiled object code between invocations, or even translation into native machine code, would be nice. Actually, a working version of such a compiler already exists and

is currently scheduled to be bundled into the 5.005 release. See the Perl FAQ for current status.

A Stroll Through Perl

We begin our journey through Perl by taking a little stroll. This stroll presents a number of different features by hacking on a small application. The explanations here are extremely brief; each subject area is discussed in *much* greater detail later in this book. But this little stroll should give you a quick taste for the language, and you can decide if you really want to finish this book rather than read some more Usenet news or run off to the ski slopes.

The "Hello, World" Program

Let's look at a little program that actually *does* something. Here is your basic "Hello, world" program:

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

The first line is the incantation that says this is a Perl program. It's also a comment for Perl; remember that a comment is anything from a pound sign to the end of that line, as in many interpreter programming languages. Unlike all other comments in the program, the one on the first line is special: Perl looks at that line for any optional arguments. In this case, the `-w` switch was used. This very important switch tells Perl to produce extra warning messages about potentially dangerous constructs. You should always develop your programs under `-w`.

The second line is the entire executable part of this program. Here we see a `print` function. The built-in function `print` starts it off, and in this case has just one argument, a C-like text string. Within this string, the character combination `\n` stands for a newline character. The `print` statement is terminated by a semicolon (`;`). As in C, all simple statements in Perl are terminated by a semicolon.*

When you invoke this program, the kernel fires up a Perl interpreter, which parses the entire program (all two lines of it, counting the first, comment line) and then executes the compiled form. The first and only operation is the execution of the `print` function, which sends its arguments to the output. After the program has completed, the Perl process exits, returning back a successful exit code to the parent shell.

Soon you'll see Perl programs where `print` and other functions are sometimes called with parentheses, other times without them. The rule is simple: in Perl,

* The semicolon can be omitted when the statement is the last statement of a block or file or `eval`.

parentheses for built-in functions are never required nor forbidden. Their use can help or hinder clarity, so use your own judgment.

Asking Questions and Remembering the Result

Let's add a bit more sophistication. The `Hello, world` greeting is a touch cold and inflexible. Let's have the program call you by your name. To do this, we need a place to hold the name, a way to ask for the name, and a way to get a response.

One kind of place to hold values (like a name) is a *scalar variable*. For this program, we'll use the scalar variable `$name` to hold your name. We'll go into more detail in Chapter 2, *Scalar Data*, about what these variables can hold, and what you can do with them. For now, assume that you can hold a single number or string (sequence of characters) in a scalar variable.

The program needs to ask for the name. To do that, we need a way to prompt and a way to accept input. The previous program showed us how to prompt: use the `print` function. And the way to get a line from the terminal is with the `<STDIN>` construct, which (as we're using it here) grabs one line of input. We assign this input to the `$name` variable. This gives us the program:

```
print "What is your name? ";
$name = <STDIN>;
```

The value of `$name` at this point has a terminating newline (`Randal` comes in as `Randal\n`). To get rid of that, we use the `chomp` function, which takes a scalar variable as its sole argument and removes the trailing newline (record separator), if present, from the string value of the variable:

```
chomp ($name);
```

Now all we need to do is say `Hello,` followed by the value of the `$name` variable, which we can do in a shell-like fashion by embedding the variable inside the quoted string:

```
print "Hello, $name!\n";
```

As with the shell, if we want a dollar sign rather than a scalar variable reference, we can precede the dollar sign with a backslash.

Putting it all together, we get:

```
#!/usr/bin/perl -w
print "What is your name? ";
$name = <STDIN>;
chomp ($name);
print "Hello, $name!\n";
```

Adding Choices

Now, let's say we have a special greeting for Randal, but want an ordinary greeting for anyone else. To do this, we need to compare the name that was entered with the string `Randal`, and if it's the same, do something special. Let's add a C-like *if-then-else* branch and a comparison to the program:

```
#!/usr/bin/perl
print "What is your name? ";
$name = <STDIN>;
chomp ($name);
if ($name eq "Randal") {
    print "Hello, Randal! How good of you to be here!\n";
} else {
    print "Hello, $name!\n"; # ordinary greeting
}
```

The `eq` operator compares two strings. If they are equal (character-for-character, and have the same length), the result is true. (There's no comparable operator* in C or C++.)

The `if` statement selects which *block* of statements (between matching curly braces) is executed; if the expression is true, it's the first block, otherwise it's the second block.

Guessing the Secret Word

Well, now that we have the name, let's have the person running the program guess a secret word. For everyone except Randal, we'll have the program repeatedly ask for guesses until the person guesses properly. First the program, and then an explanation:

```
#!/usr/bin/perl -w
$secretword = "llama"; # the secret word
print "What is your name? ";
$name = <STDIN>;
chomp $name;
if ($name eq "Randal") {
    print "Hello, Randal! How good of you to be here!\n";
} else {
    print "Hello, $name!\n"; # ordinary greeting
    print "What is the secret word? ";
    $guess = <STDIN>;
    chomp ($guess);
    while ($guess ne $secretword) {
        print "Wrong, try again. What is the secret word? ";
        $guess = <STDIN>;
        chomp ($guess);
  }
}
```

* Well, OK, there's a standard `libc` subroutine. But it's not an operator.

First, we define the secret word by putting it into another scalar variable, `$secretword`. After the greeting the (non-Randal) person is asked (with another `print`) for the guess. The guess is compared with the secret word using the `ne` operator, which returns true if the strings are not equal (this is the logical opposite of the `eq` operator). The result of the comparison controls a `while` loop, which executes the block as long as the comparison is true.

Of course, this is not a very secure program, because anyone who is tired of guessing can merely interrupt the program and get back to the prompt, or even look at the source to determine the word. But, we weren't trying to write a security system, just an example for this section.

More than One Secret Word

Let's see how we can modify this to allow more than one valid secret word. Using what we've already seen, we could compare the guess repeatedly against a series of good answers stored in separate scalar variables. However, such a list would be hard to modify or read in from a file or compute based on the day of the week.

A better solution is to store all possible answers in a data structure called a *list*, or (preferably) an *array*. Each *element* of the array is a separate scalar variable that can be independently set or accessed. The entire array can also be given a value in one fell swoop. We can assign a value to the entire array named `@words` so that it contains three possible good passwords:

```
@words = ("camel","llama","alpaca");
```

Array variable names begin with `@`, so they are distinct from scalar variable names. Another way to write this so that we don't have to put all those quote marks there is with the `qw()` operator, like so:

```
@words = qw(camel llama alpaca);
```

These mean exactly the same thing; the `qw` makes it as if we had quoted each of three strings.

Once the array is assigned, we can access each element using a subscript reference. So `$words[0]` is `camel`, `$words[1]` is `llama`, and `$words[2]` is `alpaca`. The subscript can be an expression as well, so if we set `$i` to 2, then `$words[$i]` is `alpaca`. (Subscript references start with `$` rather than `@` because they refer to a single element of the array rather than the whole array.) Going back to our previous example:

```
#!/usr/bin/perl -w
@words = qw(camel llama alpaca);
print "What is your name? ";
$name = <STDIN>;
chomp ($name);
```

```
if ($name eq "Randal") {
    print "Hello, Randal! How good of you to be here!\n";
} else {
    print "Hello, $name!\n";          # ordinary greeting
    print "What is the secret word? ";
    $guess = <STDIN>;
    chomp ($guess);
    $i = 0; # try this word first
    $correct = "maybe";               # is the guess correct or not?
    while ($correct eq "maybe") {     # keep checking til we know
        if ($words[$i] eq $guess) { # right?
            $correct = "yes";         # yes!
        } elsif ($i < 2) {            # more words to look at?
            $i = $i + 1;              # look at the next word next time
        } else {                      # no more words, must be bad
            print "Wrong, try again. What is the secret word?";
            $guess = <STDIN>;
            chomp ($guess);
            $i = 0;             # start checking at the first word again
        }
    } # end of while not correct
} # end of "not Randal"
```

You'll notice we're using the scalar variable `$correct` to indicate that we are either still looking for a good password or that we've found one.

This program also shows the `elsif` block of the `if-then-else` statement. This exact construct is not present in all programming languages; it's an abbreviation of the `else` block together with a new `if` condition, but without nesting inside yet another pair of curly braces. It's a very Perl-like thing to compare a set of conditions in a cascaded `if-elsif-elsif-elsif-else` chain. Perl doesn't really have the equivalent of C's "switch" or Pascal's "case" statement, although you can build one yourself without too much trouble. See Chapter 2 of *Programming Perl* or the *perlsyn*(1) manpage for details.

Giving Each Person a Different Secret Word

In the previous program, any person who comes along could guess any of the three words and be successful. If we want the secret word to be different for each person, we'll need a table that matches up people with words:

Person	Secret Word
Fred	camel
Barney	llama
Betty	alpaca
Wilma	alpaca

Notice that both Betty and Wilma have the same secret word. This is fine.

The easiest way to store such a table in Perl is with a *hash*. Each element of the hash holds a separate scalar value (just like the other type of array), but the hashes are referenced by a *key*, which can be any scalar value (any string or number, including noninteger and negative values). To create a hash called `%words` (notice the `%` rather than `@`) with the keys and values given in the table above, we assign a value to `%words` (much as we did earlier with the array):

```
%words = qw(
    fred        camel
    barney      llama
    betty       alpaca
    wilma       alpaca
);
```

Each pair of values in the list represents one key and its corresponding value in the hash. Note that we broke this assignment over many lines without any sort of line-continuation character, because whitespace is generally insignificant in a Perl program.

To find the secret word for Betty, we need to use Betty as the key in a reference to the hash `%words`, via some expression such as `$words{"betty"}`. The value of this reference is `alpaca`, similar to what we had before with the other array. Also as before, the key can be any expression, so setting `$person` to `betty` and evaluating `$words{$person}` gives `alpaca` as well.

Putting all this together, we get a program like this:

```
#!/usr/bin/perl
%words = qw(
    fred        camel
    barney      llama
    betty       alpaca
    wilma       alpaca
);
print "What is your name? ";
$name = <STDIN>;
chomp ($name);
if ($name eq "Randal") {
    print "Hello, Randal! How good of you to be here!\n";
} else {
    print "Hello, $name!\n";      # ordinary greeting
    $secretword = $words{$name}; # get the secret word
    print "What is the secret word? ";
    $guess = <STDIN>;
    chomp ($guess);
    while ($guess ne $secretword) {
        print "Wrong, try again. What is the secret word? ";
        $guess = <STDIN>;
        chomp ($guess);
    }
}
```

Note the lookup of the secret word. If the name is not found, the value of `$secretword` will be an empty string,* which we can then check for if we want to define a default secret word for everyone else. Here's how that looks:

```
[... rest of program deleted ...]
    $secretword = $words{$name}; # get the secret word
    if ($secretword eq "") {     # oops, not found
        $secretword = "groucho"; # sure, why a duck?
    }
    print "What is the secret word? ";
[... rest of program deleted ...]
```

Handling Varying Input Formats

If I enter `Randal L. Schwartz` or `randal` rather than `Randal`, I'm lumped in with the rest of the users, because the `eq` comparison is an exact equality. Let's look at one way to handle that.

Suppose I wanted to look for any string that began with `Randal`, rather than just a string that was equal to `Randal`. I could do this in *sed*, *awk*, or *grep* with a regular expression: a template that defines a collection of strings that match. As in *sed*, *awk*, or *grep*, the regular expression in Perl that matches any string that begins with `Randal` is `^Randal`. To match this against the string in `$name`, we use the match operator as follows:

```
if ($name =~ /^Randal/) {
    ## yes, it matches
} else {
    ## no, it doesn't
}
```

Note that the regular expression is delimited by slashes. Within the slashes, spaces and other whitespace are significant, just as they are within strings.

This almost does it, but it doesn't handle selecting `randal` or rejecting `Randall`. To accept `randal`, we add the *ignore-case* option, a small `i` appended after the closing slash. To reject `Randall`, we add a *word boundary* special marker (similar to *vi* and some versions of *grep*) in the form of `\b` in the regular expression. This ensures that the character following the first `l` in the regular expression is not another letter. This changes the regular expression to be `/^randal\b/i`, which means "`randal` at the beginning of the string, no letter or digit following, and OK to be in either case."

When put together with the rest of the program, it looks like this:

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

* Well, OK, it's the `undef` value, but it looks like an empty string to the `eq` operator. You'd get a warning about this if you used `-w` on the command line, which is why we omitted it here.

```
%words = qw(
    fred        camel
    barney      llama
    betty       alpaca
    wilma       alpaca
);
print "What is your name? ";
$name = <STDIN>;
chomp ($name);
if ($name =~ /^randal\b/i) {
    print "Hello, Randal! How good of you to be here!\n";
} else {
    print "Hello, $name!\n"; # ordinary greeting
    $secretword = $words{$name}; # get the secret word
    if ($secretword eq "") { # oops, not found
        $secretword = "groucho"; # sure, why a duck?
    }
    print "What is the secret word? ";
    $guess = <STDIN>;
    chomp ($guess);
    while ($guess ne $secretword) {
        print "Wrong, try again. What is the secret word? ";
        $guess = <STDIN>;
        chomp ($guess);
    }
}
```

As you can see, the program is a far cry from the simple `Hello, world`, but it's still very small and workable, and does quite a bit for being so short. This is The Perl Way.

Perl provides every regular expression feature found in every standard UNIX utility (and even some nonstandard ones). Not only that, but the way Perl handles string matching is about the fastest on the planet, so you don't lose performance. (A *grep*-like program written in Perl often beats the vendor-supplied* C-coded *grep* for most inputs. This means that *grep* doesn't even do its one thing very well.)

Making It Fair for the Rest

So, now I can enter `Randal` or `randal` or `Randal L. Schwartz`, but what about everyone else? Barney still has to say exactly `barney` (not even `barney` followed by a space).

To be fair to Barney, we need to grab the first word of whatever's entered, and then convert it to lowercase *before* we look up the name in the table. We do this

* GNU *egrep* tends to be much faster than Perl at this.

with two operators: the *substitute* operator, which finds a regular expression and replaces it with a string, and the *translate* operator, to put the string in lowercase.

First, the substitute operator: we want to take the contents of `$name`, find the first nonword character, and zap everything from there to the end of the string. `/\W.*/` is the regular expression we are looking for: the `\W` stands for a nonword character (something besides a letter, digit, or underscore), and `.*` means any characters from there to the end of the line. Now, to zap these characters away, we need to take whatever part of the string matches this regular expression and replace it with nothing:

```
$name =~ s/\W.*//;
```

We're using the same `=~` operator that we did before, but now on the right we have a substitute operator: the letter `s` followed by a slash-delimited regular expression and string. (The string in this example is the empty string between the second and third slashes.) This operator looks and acts very much like the substitutions of the various editors.

Now, to get whatever's left into lowercase, we translate the string using the `tr` operator.* It looks a lot like a UNIX *tr* command, taking a list of characters to find and a list of characters to replace them with. For our example, to put the contents of `$name` in lowercase, we use:

```
$name =~ tr/A-Z/a-z/;
```

The slashes delimit the searched-for and replacement character lists. The dash between `A` and `Z` stands for all the characters in between, so we have two lists that are each 26 characters long. When the `tr` operator finds a character from the string in the first list, the character is replaced with the corresponding character in the second list. So all uppercase A's become lowercase a's, and so on.†

Putting that together with everything else results in:

```
#!/usr/bin/perl
%words = qw(
    fred        camel
    barney      llama
    betty       alpaca
    wilma       alpaca
);
print "What is your name? ";
$name = <STDIN>;
chomp ($name);
```

* This doesn't work for characters with accent marks, although the `uc` function would. See the *perllocale*(1) manpage first distributed with the 5.004 release of Perl for details.

† Experts will note that we could have also constructed something like `s/(\S*).*/\L$1/` to do this all in one fell swoop, but experts probably won't be reading this section.

```
$original_name = $name; #save for greeting
$name =~ s/\W.*//; # get rid of everything after first word
$name =~ tr/A-Z/a-z/; # lowercase everything
if ($name eq "randal") { # ok to compare this way now
    print "Hello, Randal! How good of you to be here!\n";
} else {
    print "Hello, $original_name!\n"; # ordinary greeting
    $secretword = $words{$name}; # get the secret word
    if ($secretword eq "") { # oops, not found
        $secretword = "groucho"; # sure, why a duck?
    }
    print "What is the secret word? ";
    $guess = <STDIN>;
    chomp ($guess);
    while ($guess ne $secretword) {
        print "Wrong, try again. What is the secret word? ";
        $guess = <STDIN>;
        chomp ($guess);
    }
}
```

Notice how the regular expression match for `Randal` became a simple comparison again. After all, both `Randal L. Schwartz` and `Randal` become `randal` after the substitution and translation. And everyone else gets a fair ride, because `Fred` and `Fred Flintstone` both become `fred`; `Barney Rubble` and `Barney, the little guy` become `barney`, and so on.

With just a few statements, we've made the program much more user-friendly. You'll find that expressing complicated string manipulation with a few keystrokes is one of Perl's many strong points.

However, hacking away at the name so that we could compare it and look it up in the table destroyed the name that was entered. So, before the program hacks on the name, it saves it in `$original_name`. (Like C symbols, Perl variable names consist of letters, digits, and underscores and can be of nearly unlimited length.) We can then make references to `$original_name` later.

Perl has many ways to monitor and mangle strings. You'll find out about most of them in Chapter 7, *Regular Expressions*, and Chapter 15, *Other Data Transformation*.

Making It a Bit More Modular

Now that we've added so much to the code, we have to scan through many detailed lines before we can get the overall flow of the program. What we need is to separate the high-level logic (asking for a name, looping based on entered secret words) from the details (comparing a secret word to a known good word). We might do this for clarity, or maybe because one person is writing the high-level part and another is writing (or has already written) the detailed parts.

Perl provides *subroutines* that have *parameters* and *return values*. A subroutine is defined once in a program, and can be used repeatedly by being invoked from within any expression.

For our small-but-rapidly-growing program, let's create a subroutine called `good_word` that takes a name and a guessed word, and returns *true* if the word is correct and *false* if not. The definition of such a subroutine looks like this:

```
sub good_word {
    my($somename,$someguess) = @_; # name the parameters
    $somename =~ s/\W.*//; # get rid of everything after first word
    $somename =~ tr/A-Z/a-z/; # lowercase everything
    if ($somename eq "randal") { # should not need to guess
        return 1; # return value is true
    } elsif (($words{$somename} || "groucho") eq $someguess) {
        return 1; # return value is true
    } else {
        return 0; # return value is false
    }
}
```

First, the definition of a subroutine consists of the reserved word `sub` followed by the subroutine name followed by a block of code (delimited by curly braces). This definition can go anywhere in the program file, though most people put it at the end.

The first line within this particular definition is an assignment that copies the values of the two parameters of this subroutine into two local variables named `$somename` and `$someguess`. (The `my()` defines the two variables as private to the enclosing block—in this case, the entire subroutine—and the parameters are initially in a special local array called `@_`.)

The next two lines clean up the name, just like the previous version of the program.

The `if-elsif-else` statement decides whether the guessed word (`$someguess`) is correct for the name (`$somename`). `Randal` should not make it into this subroutine, but even if it does, whatever word was guessed is OK.

A return statement can be used to make the subroutine immediately return to its caller with the supplied value. In the absence of an explicit return statement, the last expression evaluated in a subroutine is the return value. We'll see how the return value is used after we finish describing the subroutine definition.

The test for the `elsif` part looks a little complicated; let's break it apart:

```
($words{$somename} || "groucho") eq $someguess
```

The first thing inside the parentheses is our familiar hash lookup, yielding some value from `%words` based on a key of `$somename`. The operator between that

value and the string `groucho` is the `||` (logical-or) operator similar to that used in C and *awk* and the various shells. If the lookup from the hash has a value (meaning that the key `$somename` was in the hash), the value of the expression is that value. If the key could not be found, the string of `groucho` is used instead. This is a very Perl-like thing to do: specify some expression, and then provide a default value using `||` in case the expression turns out to be false.

In any case, whether it's a value from the hash, or the default value `groucho`, we compare it to whatever was guessed. If the comparison is true, we return 1, otherwise we return 0.

So, expressed as a rule, if the name is `randal`, or the guess matches the lookup in `%words` based on the name (with a default of `groucho` if not found), then the subroutine returns 1, otherwise it returns 0.

Now let's integrate all this with the rest of the program:

```
#!/usr/bin/perl
%words = qw(
    fred        camel
    barney      llama
    betty       alpaca
    wilma       alpaca
);
print "What is your name? ";
$name = <STDIN>;
chomp ($name);
if ($name =~ /^randal\b/i) { # back to the other way :-)
    print "Hello, Randal! How good of you to be here!\n";
} else {
    print "Hello, $name!\n"; # ordinary greeting
    print "What is the secret word? ";
    $guess = <STDIN>;
    chomp ($guess);
    while (! good_word($name,$guess)) {
        print "Wrong, try again. What is the secret word? ";
        $guess = <STDIN>;
        chomp ($guess);
    }
}
[... insert definition of good_word() here ...]
```

Notice that we've gone back to the regular expression to check for `Randal`, because now there's no need to pull apart the first name and convert it to lowercase, as far as the main program is concerned.

The big difference is the `while` loop containing the subroutine `good_word`. Here, we see an invocation of the subroutine, passing it two parameters, `$name` and `$guess`. Within the subroutine, the value of `$somename` is set from the first

parameter, in this case `$name`. Likewise, `$someguess` is set from the second parameter, `$guess`.

The value returned by the subroutine (either 1 or 0, recalling the definition given earlier) is logically inverted with the prefix `!` (logical not) operator. This operator returns true if the expression following is false, and returns false if the expression following is true. The result of this negation controls the `while` loop. You can read this as "while it's not a good word...". Many well-written Perl programs read very much like English, provided you take a few liberties with either Perl or English. (But you certainly won't win a Pulitzer that way.)

Note that the subroutine assumes that the value of the `%words` hash is set by the main program.

Such a cavalier approach to global variables doesn't scale very well, of course. Generally speaking, variables not created with `my` are global to the whole program, while those `my` creates last only until the block in which they were declared exits. Don't worry: Perl does in fact support a rich variety of other kinds of variables, including those private to a file (or package), as well as variables private to a function that retain their values between invocations, which is what we could really use here. However, at this stage in your Perl education, explaining these would only complicate your life. When you're ready for it, check out what *Programming Perl* has to say about scoping, subroutines, modules, and objects, or see the online documentation in the *perlsub*(1), *perlmod*(1), *perlobj*(1), and *perltoot*(1) manpages.

Moving the Secret Word List into a Separate File

Suppose we wanted to share the secret word list among three programs. If we store the word list as we have done already, we will need to change all three programs when Betty decides that her secret word should be `swine` rather than `alpaca`. This can get to be a hassle, especially if Betty changes her mind often.

So, let's put the word list into a file and then read the file to get the word list into the program. To do this, we need to create an I/O channel called a *filehandle*. Your Perl program automatically gets three filehandles called `STDIN`, `STDOUT`, and `STDERR`, corresponding to the three standard I/O channels in most programming environments. We've already been using the `STDIN` handle to read data from the person running the program. Now, it's just a matter of getting another handle attached to a file of our own choice.

Here's a small chunk of code to do that:

```
sub init_words {
    open (WORDSLIST, "wordslist");
    while ($name = <WORDSLIST>) {
```

```
        chomp ($name);
        $word = <WORDSLIST>;
        chomp ($word);
        $words{$name} = $word;
    }
    close (WORDSLIST);
}
```

We're putting it into a subroutine so that we can keep the main part of the program uncluttered. This also means that at a later time (hint: a few revisions down in this stroll), we can change where the word list is stored, or even the format of the list.

The arbitrarily chosen format of the word list is one item per line, with names and words, alternating. So, for our current database, we'd have something like this:

```
fred
camel
barney
llama
betty
alpaca
wilma
alpaca
```

The `open` function initializes a filehandle named `WORDSLIST` by associating it with a file named `wordslist` in the current directory. Note that the filehandle doesn't have a funny character in front of it as the three variable types do. Also, filehandles are generally uppercase—although they aren't required to be—for reasons detailed later.

The `while` loop reads lines from the `wordslist` file (via the `WORDSLIST` filehandle) one line at a time. Each line is stored into the `$name` variable. At the end of the file, the value returned by the `<WORDSLIST>` operation is the empty string,* which looks false to the `while` loop, and terminates it. That's how we get out at the end.

If you were running with `-w`, you would have to check that the return value read in was actually defined. The empty string returned by the `<WORDSLIST>` operation isn't merely empty: it's `undef` again. The `defined` function is how you test for `undef` when this matters. When reading lines from a file, you'd do the test this way:

```
while ( defined ($name = <WORDSLIST>) ) {
```

But if you were being that careful, you'd probably also have checked to make sure that `open` returned a true value. You know, that's probably not a bad idea

* Well, technically it's `undef`, but close enough for this discussion.

either. The built-in `die` function is frequently used to exit the program with an error message in case something goes wrong. We'll see an example of it in the next revision of the program.

On the other hand, the normal case is that we've read a line (including the newline) into `$name`. First, off comes the newline using the `chomp` function. Then, we have to read the next line to get the secret word, holding that in the `$word` variable. It, too, gets the newline hacked off.

The final line of the `while` loop puts `$word` into `%words` with a key of `$name`, so that the rest of the program can access it later.

Once the file has been read, the filehandle can be recycled with the `close` function. (Filehandles are automatically closed anyway when the program exits, but we're trying to be tidy. If we were really tidy, we'd even check for a true return value from `close` in case the disk partition the file was on went south, its network filesystem became unreachable, or some other catastrophe occurred. Yes, these things really do happen. Murphy will always be with us.)

This subroutine definition can go after or before the other one. And we invoke the subroutine instead of setting `%words` in the beginning of the program, so one way to wrap up all of this might look like:

```
#!/usr/bin/perl
init_words();
print "What is your name? ";
$name = <STDIN>;
chomp $name;
if ($name =~ /^randal\b/i) { # back to the other way :-)
    print "Hello, Randal! How good of you to be here!\n";
} else {
    print "Hello, $name!\n"; # ordinary greeting
    print "What is the secret word? ";
    $guess = <STDIN>;
    chomp ($guess);
    while (! good_word($name,$guess)) {
        print "Wrong, try again. What is the secret word? ";
        $guess = <STDIN>;
        chomp ($guess);
    }
}
## subroutines from here down
sub init_words {
    open (WORDSLIST, "wordslist") ||
                                die "can't open wordlist: $!";
    while ( defined ($name = <WORDSLIST>)) {
        chomp ($name);
        $word = <WORDSLIST>;
        chomp $word;
        $words{$name} = $word;
    }
```

```
    close (WORDSLIST) || die "couldn't close wordlist: $!";
}
sub good_word {
    my($somename,$someguess) = @_; # name the parameters
    $somename =~ s/\W.*//;         # delete everything after
                                   # first word
    $somename =~ tr/A-Z/a-z/;      # lowercase everything
    if ($somename eq "randal") {   # should not need to guess
        return 1;                  # return value is true
    } elsif (($words{$somename} || "groucho") eq $someguess) {
        return 1;                  # return value is true
    } else {
        return 0;                  # return value is false
    }
}
```

Now it's starting to look like a full grown program. Notice the first executable line is an invocation of `init_words()`. The return value is not used in a further calculation, which is good because we didn't return anything remarkable. In this case, it's guaranteed to be a true value (the value 1, in particular), because if the `close` had failed, the `die` would have printed a message to `STDERR` and exited the program. The `die` function is fully explained in Chapter 10, *Filehandles and File Tests*, but because it's essential to check the return values of anything that might fail, we'll get into the habit of using it right from the start. The `$!` variable (also explained in Chapter 10), contains the system error message explaining why the system call failed.

The `open` function is also used to open files for output, or open programs as files (demonstrated shortly). The full scoop on `open` comes much later in this book, however, in Chapter 10.

Ensuring a Modest Amount of Security

"That secret word list has got to change at least once a week!" cries the Chief Director of Secret Word Lists. Well, we can't force the list to be different, but we can at least issue a warning if the secret word list has not been modified in more than a week.

The best place to do this is in the `init_words()` subroutine; we're already looking at the file there. The Perl operator `-M` returns the age in days since a file or filehandle has last been modified, so we just need to see whether this is greater than seven for the `WORDSLIST` filehandle:

```
sub init_words {
    open (WORDSLIST, "wordslist") ||
                                die "can't open wordlist: $!";
    if (-M WORDSLIST >= 7.0) { # comply with bureaucratic policy
        die "Sorry, the wordslist is older than seven days.";
    }
```

```
        while ($name = <WORDSLIST>) {
            chomp ($name);
            $word = <WORDSLIST>;
            chomp ($word);
            $words{$name} = $word;
        }
        close (WORDSLIST) || die "couldn't close wordlist: $!";
    }
```

The value of `-M WORDSLIST` is compared to seven, and if greater, bingo, we've violated policy.

The rest of the program remains unchanged, so in the interest of saving a few trees, I won't repeat it here.

Besides getting the age of a file, we can also find out its owner, size, access time, and everything else that the system maintains about a file. More on that in Chapter 10.

Warning Someone When Things Go Astray

Let's see how much we can bog down the system by sending a piece of email each time someone guesses their secret word incorrectly. We need to modify only the `good_word()` subroutine (thanks to modularity) because we have all the information right there.

The mail will be sent to you if you type your own mail address where the code says "YOUR_ADDRESS_HERE." Here's what we have to do: just before we return 0 from the subroutine, we create a filehandle that is actually a process (*mail*), like so:

```
sub good_word {
    my($somename,$someguess) = @_; # name the parameters
    $somename =~ s/\W.*//;         # get rid of stuff after
                                   # first word
    $somename =~ tr/A-Z/a-z/;      # lowercase everything
    if ($somename eq "randal") {   # should not need to guess
        return 1;                  # return value is true
    } elsif (($words{$somename}||"groucho") eq $someguess) {
        return 1;                  # return value is true
    } else {
        open MAIL,"|mail YOUR_ADDRESS_HERE";
        print MAIL "bad news: $somename guessed $someguess\n";
        close MAIL;
        return 0;                  # return value is false
    }
}
```

The first new statement here is open, which has a pipe symbol (|) at the beginning of its second argument. This is a special indication that we are opening a command rather than a file. Because the pipe is at the beginning of the command, we are opening a command so that we can write to it. (If you put the pipe at the end rather than the beginning, you can read the output of a command instead.)

The next statement, a `print`, shows that a filehandle between the `print` keyword and the values to be printed selects that filehandle for output, rather than `STDOUT`.* This means that the message will end up as the input to the *mail* command.

Finally, we close the filehandle, which starts *mail* sending its data merrily on its way.

To be proper, we could have sent the correct response as well as the error response, but then someone reading over my shoulder (or lurking in the mail system) while I'm reading my mail might get too much useful information.

Perl can also open filehandles, invoke commands with precise control over argument lists, or even fork off a copy of the current program, and execute two (or more) copies in parallel. Backquotes (like the shell's backquotes) give an easy way to grab the output of a command as data. All of this gets described in Chapter 14, *Process Management*, so keep reading.

Many Secret Word Files in the Current Directory

Let's change the definition of the secret word filename slightly. Instead of just the file named `wordslist`, let's look for anything in the current directory that ends in `.secret`. To the shell, we say

```
echo *.secret
```

to get a brief listing of all of these names. As you'll see in a moment, Perl uses a similar wildcard-name syntax.

Pulling out the `init_words()` definition again:

```
sub init_words {
    while ( defined($filename = glob("*.secret")) ) {
    open (WORDSLIST, $filename) ||
                              die "can't open wordlist: $!";
        if (-M WORDSLIST < 7.0) {
            while ($name = <WORDSLIST>) {
                chomp $name;
                $word = <WORDSLIST>;
```

* Well, technically, the currently selected filehandle. That's covered much later, though.

```
                chomp $word;
                $words{$name} = $word;
            }
        }
        close (WORDSLIST) || die "couldn't close wordlist: $!";
    }
}
```

First, we've wrapped a new `while` loop around the bulk of the routine from the previous version. The new thing here is the `glob` function. This is called a *filename glob*, for historical reasons. It works much like `<STDIN>`, in that each time it is accessed, it returns the next value: successive filenames that match the shell pattern, in this case `*.secret`. When there are no additional filenames to be returned, the filename glob returns an empty string.*

So if the current directory contains `fred.secret` and `barney.secret`, then `$filename` is `barney.secret` on the first pass through the `while` loop (the names come out in alphabetically sorted order). On the second pass, `$filename` is `fred.secret`. And there is no third pass because the glob returns an empty string the third time it is called, perceived by the `while` loop to be false, causing an exit from the subroutine.

Within the `while` loop, we open the file and verify that it is recent enough (less than seven days since the last modification). For the recent-enough files, we scan through as before.

Note that if there are no files that match `*.secret` and are less than seven days old, the subroutine will exit without having set any secret words into the `%words` array. That means that everyone will have to use the word `groucho`. Oh well. (For *real* code, I would have added some check on the number of entries in `%words` before returning, and `die`'d if it weren't good. See the `keys` function when we get to hashes in Chapter 5, *Hashes*.)

Listing the Secret Words

Well, the Chief Director of Secret Word Lists wants a report of all the secret words currently in use and how old they are. If we set aside the secret word program for a moment, we'll have time to write a reporting program for the Director.

First, let's get all of the secret words, by stealing some code from the `init_words()` subroutine:

```
while ( defined($filename = glob("*.secret")) ) {
open (WORDSLIST, $filename) || die "can't open wordlist: $!";
if (-M WORDSLIST < 7.0) {
```

* Yeah, yeah, `undef` again.

```
            chomp ($name);
            $word = <WORDSLIST>;
            chomp ($word);
    ### new stuff will go here
            }
        }
        close (WORDSLIST) || die "couldn't close wordlist: $!";
}
```

At the point marked "new stuff will go here," we know three things: the name of the file (in `$filename`), someone's name (in `$name`), and that person's secret word (in `$word`). Here's a place to use Perl's report generating tools. We define a format somewhere in the program (usually near the end, like a subroutine):

```
format STDOUT =
@<<<<<<<<<<<<<< @<<<<<<<<<< @<<<<<<<<<<<
$filename, $name, $word
.
```

The format definition begins with `format STDOUT =`, and ends with a single period. The two lines between are the format itself. The first line of this format is a *field definition line* that specifies the number, length, and type of the fields. For this format, we have three fields. The line following a field definition line is always a *field value line*. The value line gives a list of expressions that will be evaluated when this format is used, and the results of those expressions will be plugged into the fields defined in the previous line.

We invoke this format with the `write` function, like so:

```
#!/usr/bin/perl
while ( defined($filename = glob("*.secret")) ) {
    open (WORDSLIST, $filename) || die "can't open wordlist: $!";
    if (-M WORDSLIST < 7.0) {
        while ($name = <WORDSLIST>) {
            chomp ($name);
            $word = <WORDSLIST>;
            chomp ($word);
            write; # invoke format STDOUT to STDOUT
        }
    }
    close (WORDSLIST) || die "couldn't close wordlist: $!";
}
format STDOUT =
@<<<<<<<<<<<<<< @<<<<<<<<<< @<<<<<<<<<<<
$filename, $name, $word
.
```

When the format is invoked, Perl evaluates the field expressions and generates a line that it sends to the `STDOUT` filehandle. Because `write` is invoked once each time through the loop, we'll get a series of lines with text in columns, one line for each secret word entry.

Hmm. We haven't labeled the columns. That's easy enough. We just need to add a top-of-page format, like so:

```
format STDOUT_TOP =
Page @<<
$%

Filename          Name       Word
================ ========== ============
.
```

This format is named `STDOUT_TOP`, and will be used initially at the first invocation of the `STDOUT` format, and again every time 60 lines of output to `STDOUT` have been generated. The column headings here line up with the columns from the `STDOUT` format, so everything comes out tidy.

The first line of this format shows some constant text (`Page`) along with a three-character field definition. The following line is a field value line, here with one expression. This expression is the `$%` variable,* which holds the number of pages printed—a very useful value in top-of-page formats.

The third line of the format is blank. Because this line does not contain any fields, the line following it is not a field value line. This blank line is copied directly to the output, creating a blank line between the page number and the column headers below.

The last two lines of the format also contain no fields, so they are copied as is directly to the output. So this format generates four lines, one of which has a part that changes from page to page.

Just tack this definition onto the previous program to get it to work. Perl notices the top-of-page format automatically.

Perl also has fields that are centered or right-justified, and supports a *filled paragraph area* as well. More on this when we get to formats in Chapter 11, *Formats*.

Making Those Old Word Lists More Noticeable

As we are scanning through the `*.secret` files in the current directory, we may find files that are too old. So far, we are simply skipping over those files. Let's go one step more: we'll rename them to `*.secret.old` so that a directory listing will quickly show us which files are too old, simply by name.

Here's how the `init_words()` subroutine looks with this modification:

```
sub init_words {
```

* More mnemonic aliases for these predefined scalar variables are available via the English module.

```
    while ( defined($filename = glob("*.secret")) ) {
        open (WORDSLIST, $filename) ||
                                die "can't open wordlist: $!";
        if (-M WORDSLIST < 7.0) {
            while ($name = <WORDSLIST>) {
                chomp ($name);
                $word = <WORDSLIST>;
                chomp ($word);
                $words{$name} = $word;
            }
        } else { # rename the file so it gets noticed
            rename ($filename,"$filename.old") ||
                die "can't rename $filename to $filename.old: $!";
        }
        close (WORDSLIST) || die "couldn't close wordlist: $!";
    }
}
```

Notice the new `else` part of the file age check. If the file is older than seven days, it gets renamed with the `rename` function. This function takes two parameters, renaming the file named by the first parameter to the name given in the second parameter.

Perl has a complete range of file manipulation operators; anything you can do to a file from a C program, you can also do from Perl.

Maintaining a Last-Good-Guess Database

Let's keep track of when the most recent correct guess has been made for each user. One data structure that might seem to work at first glance is a hash. For example, the statement

```
$last_good{$name} = time;
```

assigns the current time in internal format (some large integer above 800 million, incrementing one number per second) to an element of `%last_good` that has the name for a key. Over time, this would seem to give us a database indicating the most recent time the secret word was guessed properly for each of the users who had invoked the program.

But, the hash doesn't have an existence between invocations of the program. Each time the program is invoked, a new hash is formed. So at most, we create a one-element hash and then immediately forget it when the program exits.

The `dbmopen` function* maps a hash out into a disk file (actually a pair of disk files) known as a *DBM*. It's used like this:

* Or using the more low-level `tie` function on a specific database, as detailed in Chapters 5 and 7 of *Programming Perl*, or in the *perltie*(1) and *AnyDBM_File*(3) manpages.

```
dbmopen (%last_good,"lastdb",0666) ||
                                die "can't dbmopen lastdb: $!";
$last_good{$name} = time;
dbmclose (%last_good) || die "can't dbmclose lastdb: $!";
```

The first statement performs the mapping, using the disk filenames of `lastdb.dir` and `lastdb.pag` (these names are the normal names for a DBM called `lastdb`). The file permissions used for these two files if the files must be created (as they will the first time through) is `0666`.* This mode means that anyone can read or write the files. If you're on a UNIX system, file permission bits are described in the *chmod*(2) manpage. On non-UNIX systems, *chmod*() may or may not work the same way. For example, under MS-DOS, files have no permissions, whereas under WindowsNT, they do. See your port's release notes about this if you're unsure.

The second statement shows that we use this mapped hash just like a normal hash. However, creating or updating an element of the hash automatically updates the disk files that form the DBM. And, when the hash is later accessed, the values within the hash come directly from the disk image. This gives the hash a life beyond the current invocation of the program—a persistence of its own.

The third statement disconnects the hash from the DBM, much like a file `close` operation.

Although the inserted statements maintain the database just fine (and even create it the first time), we don't have any way of examining the information yet. To do that, we can create a separate little program that looks something like this:

```
#!/usr/bin/perl -w
dbmopen (%last_good,"lastdb",0666) ||
                                die "can't dbmopen lastdb: $!";
foreach $name (sort keys (%last_good)) {
    $when = $last_good{$name};
    $hours = (time() - $when) / 3600; # compute hours ago
    write;
}

format STDOUT =
User @<<<<<<<<<<<<: last correct guess was @<<< hours ago.
$name, $hours
.
```

We've got a few new operations here: a `foreach` loop, sorting a list, and getting the keys of an array.

First, the `keys` function takes a hash name as an argument and returns a list of all the keys of that hash in some unspecified order. For the `%words` hash defined

* The actual permissions of the files will be the logical AND of 0666 and your process's current umask.

earlier, the result is something like `fred`, `barney`, `betty`, `wilma`, in some unspecified order. For the `%last_good` hash, the result will be a list of all users who have guessed their own secret word successfully.

The `sort` function sorts the list alphabetically (just as if you passed a text file through the *sort* command). This makes sure that the list processed by the `foreach` statement is always in alphabetical order.

Finally, the Perl `foreach` statement is a lot like the C-shell `foreach` statement. It takes a list of values and assigns each one in turn to a scalar variable (here, `$name`) executing the body of the loop (a block) once for each value. So, for five names in the `%last_good` list, we get five passes through the loop, with `$name` being a different value each time.

The body of the `foreach` loop loads up a couple of variables used within the `STDOUT` format and invokes the format. Note that we figure out the age of the entry by subtracting the stored system time (in the array) from the current time (as returned by `time`) and then divide that by 3600 (to convert seconds to hours).

Perl also provides easy ways to create and maintain text-oriented databases (like the Password file) and fixed-length-record databases (like the "last login" database maintained by the *login* program). These are described in Chapter 17, *User Database Manipulation.*

The Final Programs

Here are the programs from this stroll in their final form so you can play with them.

First, the "say hello" program:

```
#!/usr/bin/perl
init_words();
print "what is your name? ";
$name = <STDIN>;
chomp($name);
if ($name =~ /^randal\b/i) { # back to the other way :-)
    print "Hello, Randal! How good of you to be here!\n";
} else {
    print "Hello, $name!\n"; # ordinary greeting
    print "What is the secret word? ";
    $guess = <STDIN>;
    chomp $guess;
    while (! good_word($name,$guess)) {
        print "Wrong, try again. What is the secret word? ";
        $guess = <STDIN>;
        chomp $guess;
    }
}
```

```
dbmopen (%last_good,"lastdb",0666);
$last_good{$name} = time;
dbmclose (%last_good);
sub init_words {
    while ($filename = <*.secret>) {
        open (WORDSLIST, $filename)||
                                die "can't open $filename: $!";
        if (-M WORDSLIST < 7.0) {
            while ($name = <WORDSLIST>) {
                chomp ($name);
                $word = <WORDSLIST>;
                chomp ($word);
                $words{$name} = $word;
            }
        } else { # rename the file so it gets noticed
            rename ($filename,"$filename.old") ||
                        die "can't rename $filename: $!";
        }
        close WORDSLIST;
    }
}
sub good_word {
    my($somename,$someguess) = @_; # name the parameters
    $somename =~ s/\W.*//; # delete everything after first word
    $somename =~ tr/A-Z/a-z/; # lowercase everything
    if ($somename eq "randal") { # should not need to guess
        return 1; # return value is true
    } elsif (($words{$somename} || "groucho") eq $someguess) {
        return 1; # return value is true
    } else {
        open (MAIL, "|mail YOUR_ADDRESS_HERE");
        print MAIL "bad news: $somename guessed $someguess\n";
        close MAIL;
        return 0; # return value is false
    }
}
```

Next, we have the secret word lister:

```
#!/usr/bin/perl
while ($filename = <*.secret>) {
    open (WORDSLIST, $filename) ||
                            die "can't open $filename: $!";

    if (-M WORDSLIST < 7.0) {
        while ($name = <WORDSLIST>) {
            chomp ($name);
            $word = <WORDSLIST>;
            chomp ($word);
            write; # invoke format STDOUT to STDOUT
        }
    }
    close (WORDSLIST);
}
format STDOUT =
```

```
@<<<<<<<<<<<<<<< @<<<<<<<<<< @<<<<<<<<<<<
$filename, $name, $word
.
format STDOUT_TOP =
Page @<<
$%

Filename          Name        Word
================ ========== ============
.
```

And finally, the last-time-a-word-was-used display program:

```
#!/usr/bin/perl
dbmopen (%last_good,"lastdb",0666);
foreach $name (sort keys %last_good) {
    $when = $last_good{$name};
    $hours = (time - $when) / 3600; # compute hours ago
    write;
}

format STDOUT =
User @<<<<<<<<<<<: last correct guess was @<<< hours ago.
$name, $hours
.
```

Together with the secret word lists (files named *something*`.secret` in the current directory) and the database `lastdb.dir` and `lastdb.pag`, you'll have all you need.

Exercise

Most chapters end with some exercises, for which answers are found in Appendix A, *Exercise Answers*. For this stroll, the answers have already been given above.

1. Type in the example programs, and get them to work. (You'll need to create the secret-word lists as well.) Consult your local Perl guru if you need assistance.

14

Process Management

In this chapter:

Using system and exec

When you give the shell a command line to execute, the shell usually creates a new process to execute the command. This new process becomes a child of the shell, executing independently, yet coordinating with the shell.

Similarly, a Perl program can launch new processes, and like most other operations, has more than one way to do so.

The simplest way to launch a new process is to use the `system` function. In its simplest form, this function hands a single string to a brand new */bin/sh* shell to be executed as a command. When the command is finished, the `system` function returns the exit value of the command (typically 0 if everything went OK). Here's an example of a Perl program executing a *date* command using a shell:*

```
system("date");
```

We're ignoring the return value here, but it's not likely that the *date* command is going to fail anyway.

Where does the command's output go? In fact, where does the input come from, if it's a command that wants input? These are good questions, and the answers to these questions are most of what distinguishes the various forms of process-creation.

For the `system` function, the three standard files (standard input, standard output, and standard error) are inherited from the Perl process. So for the *date*

* This doesn't actually use the shell: Perl performs the operations of the shell if the command line is simple enough, and this one is.

command in the previous example, the output goes wherever the `print STDOUT` output goes—probably the invoker's display screen. Because you are firing off a shell, you can change the location of the standard output using the normal */bin/sh* I/O redirections. For example, to put the output of the *date* command into a file named *right_now*, something like this will work just fine:

```
system("date >right_now") && die "cannot create right_now";
```

This time, we not only send the output of the *date* command into a file with a redirection to the shell, but also check the return status. If the return status is true (nonzero), something went wrong with the shell command, and the `die` function will do its deed. This is backwards from normal Perl operator convention: a nonzero return value from the `system` operator generally indicates that something went wrong.

The argument to `system` can be anything you would feed */bin/sh*, so multiple commands can be included, separated by semicolons or newlines. Processes that end in & are launched and not waited for, just as if you had typed a line that ends in an & to the shell.

Here's an example of generating a *date* and *who* command to the shell, sending the output to a filename specified by a Perl variable. This all takes place in the background so that we don't have to wait for it before continuing with the Perl script:

```
$where = "who_out.".++$i; # get a new filename
system "(date; who) >$where &";
```

The return value from `system` in this case is the exit value of the shell, and would thus indicate whether the background process had launched successfully, but not whether the *date* and *who* commands executed successfully. The double-quoted string is variable interpolated, so `$where` is replaced with its value (by Perl, not by the shell). If you wanted to reference a shell variable named `$where`, you'd have to backslash the dollar sign or use a single-quoted string.

A child process inherits many things from its parent besides the standard filehandles. These include the current umask, current directory, and of course, the user ID.

Additionally, all environment variables are inherited by the child. These variables are typically altered by the *csh setenv* command or the corresponding assignment and *export* by the */bin/sh* shell. Environment variables are used by many utilities, including the shells, to alter or control the way that utility operates.

Perl gives you a way to examine and alter current environment variables through a special hash called `%ENV` (uppercase). Each key of this hash corresponds to the name of an environment variable, with the corresponding value being, well, the

corresponding value. Examining this hash shows you the environment handed to Perl by the parent shell; altering the hash affects the environment used by Perl and by its child processes, but not parents.

For example, here's a simple program that acts like *printenv*:

```
foreach $key (sort keys %ENV) {
    print "$key=$ENV{$key}\n";
}
```

Note the equal sign here is not an assignment, but simply a text character that the `print` is using to say stuff like `TERM=xterm` or `USER=merlyn`.

Here's a program snippet that alters the value of `PATH` to make sure that the *grep* command run by `system` is looked for only in the normal places:

```
$oldPATH = $ENV{"PATH"};                  # save previous path
$ENV{"PATH"} = "/bin:/usr/bin:/usr/ucb"; # force known path
system("grep fred bedrock >output");      # run command
$ENV{"PATH"} = $oldPATH;                  # restore previous path
```

That's a lot of typing. It'd be faster just to set a local value for this hash element.

Despite its other shortcomings, the `local` operator can do one thing that `my` cannot: it can give just one element of an array or a hash a temporary value.

```
{
    local $ENV{"PATH"} = "/bin:/usr/bin:/usr/ucb";
    system "grep fred bedrock >output";
}
```

The `system` function can also take a list of arguments rather than a single argument. In that case, rather than handing the list of arguments off to a shell, Perl treats the first argument as the command to run (located according to the `PATH` if necessary) and the remaining arguments as arguments to the command without normal shell interpretation. In other words, you don't need to quote whitespace or worry about arguments that contain angle brackets because those are all merely characters to hand to the program. So, the following two commands are equivalent:

```
system "grep 'fred flintstone' buffaloes";   # using shell
system "grep","fred flintstone","buffaloes"; # avoiding shell
```

Giving `system` a list rather than giving it a simple string saves one shell process as well, so do this when you can. (Actually, when the one-argument form of `system` is simple enough, Perl itself optimizes away the shell invocation entirely, calling the resulting program directly as if you had used the multiple-argument invocation.)

Here's another example of equivalent forms:

```
@cfiles = ("fred.c","barney.c");          # what to compile
```

```
@options = ("-DHARD","-DGRANITE");          # options
system "cc -o slate @options @cfiles";      # using shell
system "cc","-o","slate",@options,@cfiles; # avoiding shell
```

Using Backquotes

Another way to launch a process is to put a */bin/sh* shell command line between backquotes. Like the shell, this fires off a command and waits for its completion, capturing the standard output as it goes along:

```
$now = "the time is now ".`date`; # gets text and date output
```

The value of `$now` winds up with the text `the time is now` along with the result of the *date*(1) command (including the trailing newline), so it looks something like this:

```
the time is now Fri Aug 13 23:59:59 PDT 1993
```

If the backquoted command is used in a list context rather than a scalar context, you get a list of strings, each one being a line (terminated in a newline*) from the command's output. For the *date* example, we'd have just one element because it generated only one line of text. The output of *who* looks like this:

```
merlyn      tty42    Dec  7 19:41
fred        tty1A    Aug 31 07:02
barney      tty1F    Sep  1 09:22
```

Here's how to grab this output in a list context:

```
foreach $_ (`who`) { # once per text line from who
    ($who,$where,$when) = /(\S+)\s+(\S+)\s+(.*)/;
    print "$who on $where at $when\n";
}
```

Each pass through the loop works on a separate line of the output of *who*, because the backquoted command is evaluated within a list context.

The standard input and standard error of the command within backquotes are inherited from the Perl process.† This means that you normally get just the standard output of the commands within the backquotes as the value of the backquoted-string. One common thing to do is to merge the standard error into the standard output so that the backquoted command picks up both, using the `2>&1` construct of the shell:

```
die "rm spoke!" if `rm fred 2>&1`;
```

---

* Or whatever you've set `$/` to.

† Actually, it's a bit more complicated that this. See the question in Section 8 of the Perl FAQ on "How can I capture STDERR from an external command?" If you're running Perl version 5.004, the FAQ is distributed as a normal manpage—*perlfaq8*(1) in this case.

Here, the Perl process is terminated if *rm* says anything, either to standard output or standard error, because the result will no longer be an empty string (an empty string would be false).

Using Processes as Filehandles

Yet another way to launch a process is to create a process that looks like a filehandle (similar to the *popen*(3) C library routine if you're familiar with that). We can create a process-filehandle that either captures the output from or provides input to the process.* Here's an example of creating a filehandle out of a *who*(1) process. Because the process is generating output that we want to read, we make a filehandle that is open for reading, like so:

```
open(WHOPROC, "who|"); # open who for reading
```

Note the vertical bar on the right side of `who`. That bar tells Perl that this `open` is not about a filename, but rather a command to be started. Because the bar is on the right of the command, the filehandle is opened for reading, meaning that the standard output of *who* is going to be captured. (The standard input and standard error remain shared with the Perl process.) To the rest of the program, the `WHOPROC` handle is merely a filehandle that is open for reading, meaning that all normal file I/O operators apply. Here's a way to read data from the *who* command into an array:

```
@whosaid = <WHOPROC>;
```

Similarly, to open a command that expects input, we can open a process-filehandle for writing by putting the vertical bar on the left of the command, like so:

```
open(LPR,"|lpr -Pslatewriter");
print LPR @rockreport;
close(LPR);
```

In this case, after opening `LPR`, we write some data to it and then close it. Opening a process with a process-filehandle allows the command to execute in parallel with the Perl program. Saying `close` on the filehandle forces the Perl program to wait until the process exits. If you don't close the filehandle, the process can continue to run even beyond the execution of the Perl program.

Opening a process for writing causes the command's standard input to come from the filehandle. The process shares the standard output and standard error with Perl. As before, you may use */bin/sh*-style I/O redirection, so here's one way to simply discard the error messages from the *lpr* command in that last example:

* But not both at once. See Chapter 6 of *Programming Perl* or *perlipc*(1) for examples of bidirectional communication.

```
open(LPR,"|lpr -Pslatewriter >/dev/null 2>&1");
```

The `>/dev/null` causes standard output to be discarded by being redirected to the null device. The `2>&1` causes standard error to be sent to where the standard output is sent, resulting in errors being discarded as well.

You could even combine all this, generating a report of everyone except Fred in the list of logged-on entries, like so:

```
open (WHO,"who|");
open (LPR,"|lpr -Pslatewriter");
while (<WHO>) {
    unless (/fred/) { # don't show fred
        print LPR $_;
    }
}
close WHO;
close LPR;
```

As this code fragment reads from the `WHO` handle one line at a time, it prints all of the lines that don't contain the string `fred` to the `LPR` handle. So the only output on the printer is the lines that don't contain `fred`.

You don't have to open just one command at a time. You can open an entire pipeline. For example, the following line starts up an *ls*(1) process, which pipes its output into a *tail*(1) process, which finally sends its output along to the `WHOPR` filehandle:

```
open(WHOPR, "ls | tail -r |");
```

Using fork

Still another way of creating an additional process is to clone the current Perl process using a UNIX primitive called `fork`. The `fork` function simply does what the *fork*(2) system call does: it creates a clone of the current process. This clone (called the child, with the original called the parent) shares the same executable code, variables, and even open files. To distinguish the two processes, the return value from `fork` is zero for the child, and nonzero for the parent (or `undef` if the system call fails). The nonzero value received by the parent happens to be the child's process ID. You can check for the return value and act accordingly:

```
if (!defined($child_pid = fork())) {
    die "cannot fork: $!";
} elsif ($child_pid) {
    # I'm the parent
} else {
    # I'm the child
}
```

To best use this clone, we need to learn about a few more things that parallel their UNIX namesakes closely: the `wait`, `exit`, and `exec` functions.

The simplest of these is the `exec` function. It's just like the `system` function, except that instead of firing off a new process to execute the shell command, Perl replaces the current process with the shell. (In UNIX parlance, Perl `exec`'s the shell.) After a successful `exec`, the Perl program is gone, having been replaced by the requested program. For example,

```
exec "date";
```

replaces the current Perl program with the *date* command, causing the output of the *date* to go to the standard output of the Perl program. When the *date* command finishes, there's nothing more to do because the Perl program is long gone.

Another way of looking at this is that the `system` function is like a `fork` followed by an `exec`, as follows:

```
# METHOD 1... using system:
system("date");

# METHOD 2... using fork/exec:
unless (fork) {
    # fork returned zero, so I'm the child, and I exec:
    exec("date"); # child process becomes the date command
}
```

Using `fork` and `exec` this way isn't quite right though, because the *date* command and the parent process are both chugging along at the same time, possibly intermingling their output and generally mucking things up. What we need is a way to tell the parent to wait until the child process completes. That's exactly what the `wait` function does; it waits until the child (any child, to be precise) has completed. The `waitpid` function is more discriminating: it waits for a specific child process to complete rather just any kid:

```
if (!defined($kidpid = fork())) {
    # fork returned undef, so failed
    die "cannot fork: $!";
} elsif ($kidpid == 0) {
                # fork returned 0, so this branch is the child
    exec("date");
                # if the exec fails, fall through to the next statement
    die "can't exec date: $!";
} else {
                # fork returned neither 0 nor undef,
                # so this branch is the parent
    waitpid($kidpid, 0);
}
```

If this all seems rather fuzzy to you, you should probably study up on the *fork*(2) and *exec*(2) system calls in a traditional UNIX text, because Perl is pretty much just passing the function calls right down to the UNIX system calls.

The `exit` function causes an immediate exit from the current Perl process. You'd use this to abort a Perl program from somewhere in the middle, or with `fork` to execute some Perl code in a process and then quit. Here's a case of removing some files in */tmp* in the background using a forked Perl process:

```
unless (defined ($pid = fork)) {
    die "cannot fork: $!";
}
unless ($pid) {
    unlink </tmp/badrock.*>;      # blast those files
    exit;                         # the child stops here
}
                                  # Parent continues here
waitpid($pid, 0);                 # must clean up after dead kid
```

Without the `exit`, the child process would continue executing Perl code (at the line marked `Parent continues here`), and that's definitely not what we want.

The `exit` function takes an optional parameter, which serves as the numeric exit value that can be noticed by the parent process. The default is to exit with a zero value, indicating that everything went OK.

Summary of Process Operations

Table 14-1 summarizes the operations that you have for launching a process.

Table 14-1. Summary of Subprocess Operations

Operation	Standard Input	Standard Output	Standard Error	Waited for?
`system()`	Inherited from program	Inherited from program	Inherited from program	Yes
Backquoted string	Inherited from program	Captured as string value	Inherited from program	Yes
`open()` command as filehandle for output	Connected to filehandle	Inherited from program	Inherited from program	Only at time of `close()`
`open()` command as filehandle for input	Inherited from program	Connected to filehandle	Inherited from program	Only at time of `close()`
`fork`, `exec`, `wait`, `waitpid`	User selected	User selected	User selected	User selected

The simplest way to create a process is with the `system` function. Standard input, output, and error are unaffected (they're inherited from the Perl process). A backquoted string creates a process, capturing the standard output of the process as a string value for the Perl program. Standard input and standard error are unaffected. Both these methods require that the process finish before any more code is executed.

A simple way to get an asynchronous process (one that allows the Perl program to continue before the process is complete) is to open a command as a filehandle, creating a pipe for the command's standard input or standard output. A command opened as a filehandle for reading inherits the standard input and standard error from the Perl program; a command opened as a filehandle for writing inherits the standard output and standard error from the Perl program.

The most flexible way of starting a process is to have your program invoke the `fork`, `exec`, and `wait` or `waitpid` functions, which map directly to their UNIX system call namesakes. Using these functions, you can select whether you are waiting or not, and configure the standard input, output, and error any way you choose.*

Sending and Receiving Signals

One method of interprocess communication is to send and receive signals. A signal is a one-bit message (meaning "this signal happened") sent to a process from another process or from the kernel. Signals are numbered, usually from one to some small number like 15 or 31. Some signals have predefined meanings and are sent automatically to a process under certain conditions (such as memory faults or floating-point exceptions); others are strictly user-generated from other processes. Those processes must have permission to send such a signal. Only if you are the superuser or if the sending process has the same user ID as the receiving process is the signal permitted.

The response to a signal is called the signal's *action*. Predefined signals have certain useful default actions, such as aborting the process or suspending it. Other signals are completely ignored by default. Nearly all signals can have their default action overridden, to either be ignored or else *caught* (invoking a user-specified section of code automatically).

So far, this is all standard stuff; here's where it gets Perl-specific. When a Perl process catches a signal, a subroutine of your choosing gets invoked asynchronously and automatically, momentarily interrupting whatever was executing.

* Although it might also help to know about `open(STDERR,">&STDOUT")` forms for fine tuning the filehandles. See the open entry in Chapter 3 of *Programming Perl*, or in *perlfunc*(1).

When the subroutine exits, whatever was executing resumes as if nothing had happened (except for the actions performed by the subroutine, if any).

Typically, the signal-catching subroutine will do one of two things: abort the program after executing some cleanup code, or set some flag (such as a global variable) that the program routinely checks.*

You need to know the signal names to register a signal handler with Perl. By registering a signal handler, Perl will call the selected subroutine when the signal is received.

Signal names are defined in the *signal*(2) manpage, and usually also in the C include file */usr/include/sys/signal.h*. Names generally start with `SIG`, such as `SIGINT`, `SIGQUIT`, and `SIGKILL`. To declare the subroutine `my_sigint_catcher()` as the signal handler to deal with the `SIGINT`, we set a value into the magic `%SIG` hash. In this hash, we set the value of the key `INT` (that's `SIGINT` without the `SIG`) to the name of the subroutine that will catch the `SIGINT` signal, like so:

```
$SIG{'INT'} = 'my_sigint_catcher';
```

But we also need a definition for that subroutine. Here's a simple one:

```
sub my_sigint_catcher {
    $saw_sigint = 1; # set a flag
}
```

This signal catcher sets a global variable and then returns immediately. Returning from this subroutine causes execution to resume wherever it was interrupted. Typically, you'd first zero the `$saw_sigint` flag, set this subroutine up as a `SIGINT` catcher, and then do your long-running routine, like so:

```
$saw_sigint = 0;                    # clear the flag
$SIG{'INT'} = 'my_sigint_catcher'; # register the catcher
foreach (@huge_array) {
                                    # do something
                                    # do more things
                                    # do still more things
    if ($saw_sigint) {              # interrupt wanted?
                                    # some sort of cleanup here
        last;
    }
}
$SIG{'INT'} = 'DEFAULT'; # restore the default action
```

* In fact, doing anything more complicated than this is likely to mess things up; most of Perl's inner workings do not like to be executed in the main program and from the subroutine at the same time. Neither do your system libraries.

The trick here is that the value of the flag is checked at useful points during the evaluation and is used to exit the loop prematurely, here also handling some cleanup actions. Note the last statement in the preceding code: setting the action to `DEFAULT` restores the default action on a particular signal (another `SIGINT` will abort the program immediately). Another useful special value like this is `IGNORE`, meaning to ignore the signal (if the default action is not to ignore the signal, like `SIGINT`). You can make a signal action `IGNORE` if no cleanup actions are required, and you don't want to terminate operations early.

One of the ways that the `SIGINT` signal is generated is by having the user press the selected interrupt character (like CTRL-C) on the terminal. But a process can also generate the `SIGINT` signal directly using the `kill` function. This function takes a signal number or name, and sends that signal to the list of processes (identified by process ID) following the signal. So sending a signal from a program requires determining the process IDs of the recipient processes. (Process IDs are returned from some of the functions, such as `fork` and—when opening a program as a filehandle—`open`). Suppose you want to send a signal 2 (also known as `SIGINT`) to the processes 234 and 237. It's as simple as this:

```
kill(2,234,237); # send SIGINT to 234 and 237
kill ('INT', 234, 237); #same
```

For more about signal handling, see Chapter 6 of *Programming Perl* or the *perlipc*(1) manpage.

Exercises

See Appendix A for answers.

1. Write a program to parse the output of the *date* command to get the current day of the week. If the day of the week is a weekday, print `get to work`, otherwise print `go play`.
2. Write a program that gets all of the real names of the users from the */etc/passwd* file, then transforms the output of the *who* command, replacing the login name (the first column) with the real name. (Hint: create a hash where the key is the login name and the value is the real name.) Try this both with the *who* command in backquotes and opened as a pipe. Which was easier?
3. Modify the previous program so that the output automatically goes to the printer. (If you can't access a printer, perhaps you can send yourself mail.)
4. Suppose the `mkdir` function were broken. Write a subroutine that doesn't use `mkdir`, but invokes `/bin/mkdir` with `system` instead. (Be sure that it works with directories that have a space in the name.)
5. Extend the routine from the previous exercise to employ `chmod` to set the permissions.