BONUS CHAPTER 44

Using Perl

IN THIS CHAPTER

Perl (the *Practical Extraction and Report Language*, or the *Pathologically Eclectic Rubbish Lister*, depending on who you speak to) is a powerful scripting tool that enables you to manage files, create reports, edit text, and perform many other tasks. Perl is included with and installed in Ubuntu by default and could be considered an integral part of the distribution because Ubuntu depends on Perl for many types of software services, logging activities, and software tools.

Perl is not the easiest of programming languages to learn because it is designed for flexibility. This chapter shows how to create and use Perl scripts on your system. You learn what a Perl program looks like, how the language is structured, and where you can find modules of prewritten code to help you write your own Perl scripts. This chapter also includes several examples of Perl used to perform a few common functions on a computer system.

Using Perl with Linux

Although originally designed as a data-extraction and report-generation language, Perl appeals to many Linux system administrators because they can use it to create utilities that fill a gap between the capabilities of shell scripts and compiled C programs (see Chapter 13, "Automating Tasks and Shell Scripting," and Chapter 37, "Using Programming Tools for Ubuntu"). Another advantage of Perl over other UNIX tools is that it can process and extract data from binary files, whereas `sed` and `awk` cannot.

NOTE

In Perl, "there is more than one way to do it." This is the unofficial motto of Perl, and it comes up so often that it is usually abbreviated as TIMTOWTDI.

You can use Perl at your shell's command line to execute one-line Perl programs, but most often the programs (usually ending in `.pl`) are run as a command. These programs generally work on any computer platform because Perl has been ported to nearly every operating system.

Perl programs are used to support a number of Ubuntu services, such as system logging. For example, if you install the `logwatch` package, the logwatch.pl program is run every morning at 6:25 a.m. by the `crond` (scheduling) daemon on your system. Other Ubuntu services supported by Perl include the following:

- Amanda for local and network backups
- Fax spooling with the `faxrunqd` program
- Printing supported by Perl document-filtering programs
- Hardware sensor monitoring setup using the `sensors-detect` Perl program

Perl Versions

Perl is installed by default.

You can download the code from www.perl.com and build the newest version from source if you want to, although a stable and quality release of Perl is already installed by default in Ubuntu and most (all?) Linux and UNIX-like distributions, including Mac OS X. Updated versions might appear in the Ubuntu repositories, but they're generally only security fixes that can be installed by updating your system. See Chapter 9, "Managing Software," to see how to quickly get a list of available updates for Ubuntu.

You can determine what version of Perl you installed by typing `perl -v` at a shell prompt. If you are installing the latest Ubuntu distribution, you should have the latest version of Perl that was available when the software for your Ubuntu release was gathered and finalized.

A Simple Perl Program

This section introduces a very simple Perl program example to get you started using Perl. Although trivial for experienced Perl hackers, a short example is necessary for new users who want to learn more about Perl.

To introduce you to the absolute basics of Perl programming, Listing 44.1 illustrates a simple Perl program that prints a short message.

LISTING 44.1 A Simple Perl Program

```
#!/usr/bin/perl
print 'Look at all the camels!\n';
```

Type that in and save it to a file called `trivial.pl`. Then make the file executable using the `chmod` command (see the following sidebar) and run it at the command prompt.

COMMAND-LINE ERROR

If you get the message `bash: trivial.pl: command not found` or `bash: ./trivial.pl: Permission denied`, you have either typed the command line incorrectly or forgotten to make `trivial.pl` executable with the `chmod` command, as shown here:

```
matthew@seymour:~$ chmod +x trivial.pl
```

You can force the command to execute in the current directory as follows:

```
matthew@seymour:~$ ./trivial.pl
```

Or you can use Perl to run the program like this:

```
matthew@seymour:~$ perl trivial.pl
```

The sample program in the listing is a two-line Perl program. Typing in the program and running it (using Perl or making the program executable) shows how to create your first Perl program, a process duplicated by Linux users around the world every day.

NOTE

`#!` is often pronounced `she-bang`, which is short for `sharp` (the musicians name for the `#` character), and `bang`, which is another name for the exclamation point. This notation is also used in shell scripts. See Chapter 13, "Automating Tasks and Shell Scripting," for more information about writing shell scripts.

The `#!` line is technically not part of the Perl code at all. The `#` character indicates that the rest of the screen line is a comment. The comment is a message to the shell, telling it where it should go to find the executable to run this program. The interpreter ignores the comment line.

Exceptions to this practice include when the `#` character is in a quoted string and when it is being used as the delimiter in a regular expression. Comments are useful to document your scripts, like this:

```
#!/usr/bin/perl
# a simple example to print a greeting
print "hello there\n";
```

A block of code, such as what might appear inside a loop or a branch of a conditional statement, is indicated with curly braces ({}). For example, here is an infinite loop:

```
#!/usr/bin/perl
# a block of code to print a greeting forever
while (1) {
        print "hello there\n";
};
```

Perl statements are terminated with a semicolon (;). A Perl statement can extend over several screen lines because Perl is not concerned about whitespace.

The second line of the simple program prints the text enclosed in quotation marks. `\n` is the escape sequence for a newline character.

TIP

Using the `perldoc` and `man` commands is an easy way to get more information about the version of Perl installed on your system. To learn how to use the `perldoc` command, enter the following:

```
matthew@seymour:~$ perldoc perldoc
```

To get introductory information on Perl, you can use either of these commands:

```
matthew@seymour:~$ perldoc perl
matthew@seymour:~$ man perl
```

For an overview or table of contents of Perl's documentation, use the `perldoc` command, like this:

```
matthew@seymour:~$ perldoc perltoc
```

The documentation is extensive and well organized. Perl includes a number of standard Linux manual pages as brief guides to its capabilities, but perhaps the best way to learn more about Perl is to read its `perlfunc` document, which lists all the available Perl functions and their usage. You can view this document by using the `perldoc` script and typing `perldoc perlfunc` at the command line. You can also find this document online at http://perldoc.perl.org/.

Perl Variables and Data Structures

Perl is a *weakly typed* language, meaning that it does not require that you declare a data type, such as a type of value (data) to be stored in a particular variable. C, for example, makes you declare that a particular variable is an integer, a character, a structure, or whatever the case may be. Perl variables are whatever type they need to be and can change type when you need them to.

Perl Variable Types

Perl has three variable types: *scalars, arrays,* and *hashes*. A different character is used to signify each variable type, so you can have the same name used with each type at the same time.

Scalar variables are indicated with the `$` character, as in `$penguin`. Scalars can be numbers or strings, and they can change type from one to the other as needed. If you treat a number like a string, it becomes a string. If you treat a string like a number, it is translated into a number if it makes sense to do so; otherwise, it usually evaluates to `0`. For example, the string `"76trombones"` evaluates as the number `76` if used in a numeric calculation, but the string `"polar bear"` evaluates to `0`.

Perl arrays are indicated with the `@` character, as in `@fish`. An `array` is a list of values referenced by index number, starting with the first element numbered `0`, just as in C and `awk`. Each element in the array is a scalar value. Because scalar values are indicated with the `$` character, a single element in an array is also indicated with a `$` character.

For example, `$fish[2]` refers to the third element in the `@fish` array. This tends to throw some people off but is similar to arrays in C in which the first array element is `0`.

Hashes are indicated with the `%` character, as in `%employee`. A *hash* is a list of name and value pairs. Individual elements in the hash are referenced by name rather than by index (unlike an array). Again, because the values are scalars, the `$` character is used for individual elements.

For example, `$employee{name}` gives you one value from the hash. Two rather useful functions for dealing with hashes are `keys` and `values`. The `keys` function returns an array containing all the keys of the hash, and `values` returns an array of the values of the hash. Using this approach, the Perl program in Listing 44.2 displays all the values in your environment, much like typing the `bash` shell's `env` command.

LISTING 44.2 Displaying the Contents of the `env` Hash

```
#!/usr/bin/perl
foreach $key (keys %ENV)  {
  print "$key = $ENV{$key}\n";
}
```

Special Variables

Perl has a variety of special variables, which usually look like punctuation—`$_`, `$!`, and `$]`—and are all extremely useful for shorthand code. `$_` is the default variable, `$!` is the error message returned by the operating system, and `$]` is the Perl version number.

`$_` is perhaps the most useful of these. You will see that variable used often in this chapter. `$_` is the Perl default variable, which is used when no argument is specified. For example, the following two statements are equivalent:

```
chomp;
chomp($_);
```

The following loops are equivalent:

```
for $cow (@cattle) {
        print "$cow says moo.\n";
}
for (@cattle)        {
        print "$_ says moo.\n";
}
```

For a complete listing of the special variables, see the `perlvar` man page.

Operators

Perl supports a number of operators to perform various operations. There are *comparison* operators (used to compare values, as the name implies), *compound* operators (used to combine operations or multiple comparisons), *arithmetic* operators (to perform math), and special string constants.

Comparison Operators

The comparison operators used by Perl are similar to those used by C, `awk`, and the `csh` shells, and are used to specify and compare values (including strings). A comparison operator is most often used within an `if` statement or loop. Perl has comparison operators for numbers and strings. Table 44.1 shows the numeric comparison operators and their meanings.

TABLE 44.1 Numeric Comparison Operators in Perl

Operator	Meaning
`==`	Is equal to
`<`	Less than
`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to
`<=>`	Returns `-1` if less than, `0` if equal, and `1` if greater than
`!=`	Not equal to
`..`	Range of `>=` first operand to `<=` second operand

Table 44.2 shows the string comparison operators and their meanings.

TABLE 44.2 String Comparison Operators in Perl

Operator	Meaning
`eq`	Is equal to
`lt`	Less than
`gt`	Greater than
`le`	Less than or equal to
`ge`	Greater than or equal to
`ne`	Not equal to
`cmp`	Returns -1 if less than, 0 if equal, and 1 if greater than
`=~`	Matched by regular expression
`!~`	Not matched by regular expression

Compound Operators

Perl uses compound operators, similar to those used by C or `awk`, which can be used to combine other operations (such as comparisons or arithmetic) into more complex forms of logic. Table 44.3 shows the compound pattern operators and their meanings.

TABLE 44.3 Compound Pattern Operators in Perl

Operator	Meaning
`&&`	Logical `AND`
`||`	Logical `OR`
`!`	Logical `NOT`
`()`	Parentheses; used to group compound statements

Arithmetic Operators

Perl supports a variety of math operations. Table 44.4 summarizes these operators.

TABLE 44.4 Perl Arithmetic Operators

Operator	Purpose
`x**y`	Raises `x` to the `y` power (same as `x^y`)
`x%y`	Calculates the remainder of `x/y`
`x+y`	Adds `x` to `y`
`x-y`	Subtracts `y` from `x`

Operator	Purpose
x*y	Multiplies x times y
x/y	Divides x by y
-y	Negates y (switches the sign of y); also known as the unary minus
++y	Increments y by 1 and uses value (prefix increment)
y++	Uses value of y and then increments by 1 (postfix increment)
—y	Decrements y by 1 and uses value (prefix decrement)
y—	Uses value of y and then decrements by 1 (postfix decrement)
x=y	Assigns value of y to x. Perl also supports operator-assignment operators (+=, -=, *=, /=, %=, **=, and others)

You can also use comparison operators (such as == or <) and compound pattern operators (&&, ||, and !) in arithmetic statements. They evaluate to the value 0 for false and 1 for true.

Other Operators

Perl supports a number of operators that do not fit any of the prior categories. Table 44.5 summarizes these operators.

TABLE 44.5 Other Perl Operators

Operator	Purpose
~x	Bitwise not (changes 0 bits to 1 and 1 bits to 0)
x & y	Bitwise and
x | y	Bitwise or
x ^ y	Bitwise exclusive or (XOR)
x << y	Bitwise shift left (shifts x by y bits)
x >> y	Bitwise shift right (shifts x by y bits)
x . y	Concatenate y onto x
a x b	Repeats string a for b number of times
x, y	Comma operator—evaluates x and then y
x ? y : z	Conditional expression (If x is true, y is evaluated; otherwise, z is evaluated.)

Except for the comma operator and conditional expression, you can also use these operators with the assignment operator, similar to the way addition (+) can be combined with assignment (=), giving +=.

Special String Constants

Perl supports string constants that have special meaning or cannot be entered from the keyboard.

Table 44.6 shows most of the constants supported by Perl.

TABLE 44.6 Perl Special String Constants

Expression	Meaning
`\\`	The means of including a backslash
`\a`	The alert or bell character
`\b`	Backspace
`\cC`	Control character (like holding the Ctrl key down and pressing the C character)
`\e`	Escape
`\f`	Formfeed
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\xNN`	Indicates that `NN` is a hexadecimal number
`\0NNN`	Indicates that `NNN` is an octal (base 8) number

Conditional Statements: `if/else` and `unless`

Perl offers two conditional statements, `if` and `unless`, which function opposite one another. `if` enables you to execute a block of code only if certain conditions are met so that you can control the flow of logic through your program. Conversely, `unless` performs the statements when certain conditions are not met.

The following sections explain and demonstrate how to use these conditional statements when writing scripts for Linux.

`if`

The syntax of the Perl `if/else` structure is as follows:

```
if (condition) {
  statement or block of code
} elsif (condition) {
  statement or block of code
} else {
  statement or block of code
}
```

`condition` is a statement that returns a `true` or `false` value.

Truth is defined in Perl in a way that might be unfamiliar to you, so be careful. Everything in Perl is true except `0` (the digit zero), `"0"` (the string containing the number `0`), "" (the empty string), and an undefined value. Note that even the string `"00"` is a true value because it is not one of the four false cases.

The statement or block of code is executed if the test condition returns a true value.

For example, Listing 44.3 uses the `if/else` structure and shows conditional statements using the `eq` string comparison operator.

LISTING 44.3 `if/elsif/else`

```
if  ($favorite eq "chocolate") {
      print "I like chocolate too.\n";
} elsif ($favorite eq "spinach") {
      print "Oh, I do not like spinach.\n";
} else {
     print "Your favorite food is $favorite.\n";
}
```

`unless`

`unless` works just like `if`, only backward. `unless` performs a statement or block if a condition is false:

```
unless ($name eq "Rich")        {
  print "Go away, you're not allowed in here!\n";
}
```

NOTE

You can restate the preceding example in more natural language, like this:

```
print "Go away!\n" unless $name eq "Rich";
```

Looping

A *loop* is a way to repeat a program action multiple times. A simple example is a countdown timer that performs a task (waiting for one second) 300 times before telling you that your egg is done boiling.

Looping constructs (also known as *control structures*) can be used to iterate a block of code as long as certain conditions apply, or while the code steps through (evaluates) a list of values, perhaps using that list as arguments.

Perl has four looping constructs: `for`, `foreach`, `while`, and `until`.

for

The `for` construct performs a *statement* (block of code) for a set of conditions defined as follows:

```
for (start condition; end condition; increment function) {
  statement(s)
}
```

The `start` condition is set at the beginning of the loop. Each time the loop is executed, the `increment` function is performed until the end condition is achieved. This looks much like the traditional `for/next` loop. The following code is an example of a `for` loop:

```
for ($i=1; $i<=10; $i++) {
      print "$i\n"
}
```


foreach

The `foreach` construct performs a statement block for each element in a list or array:

```
@names = ("alpha","bravo","Charlie");
foreach $name (@names) {
  print "$name sounding off!\n";
}
```

The loop variable (`$name` in the example) is not merely set to the value of the array elements; it is aliased to that element. That means if you modify the loop variable, you're actually modifying the array. If no loop array is specified, the Perl default variable `$_` may be used:

```
@names = ("alpha","bravo","Charlie");
foreach (@names) {
  print "$_ sounding off!\n";

}
```

This syntax can be very convenient, but it can also lead to unreadable code. Give a thought to the poor person who'll be maintaining your code. (It will probably be you.)

NOTE

`foreach` is frequently abbreviated as `for`.

while

while performs a block of statements as long as a particular condition is true:

```
while ($x<10) {
     print "$x\n";
     $x++;
}
```

Remember that the condition can be anything that returns a true or false value. For example, it could be a function call:

```
while ( InvalidPassword($user, $password) )        {
        print "You've entered an invalid password. Please try again.\n";
        $password = GetPassword;
}
```

until

until is the exact opposite of the while statement. It performs a block of statements as long as a particular condition is false (or, rather, until it becomes true):

```
until (ValidPassword($user, $password))  {
  print "YSdpgm_m
Sdpgm_m
You have entered an invalid password. Please try again.\n";
Sdpgm_m
  $password = GetPassword;
}
```

last and next

You can force Perl to end a loop early by using a last statement. last is similar to the C break command; the loop is exited. If you decide you need to skip the remaining contents of a loop without ending the loop itself, you can use next, which is similar to the C continue command. Unfortunately, these statements do not work with do ... while. However, you can use redo to jump to a loop (marked by a label) or inside the loop where called:

```
$a = 100;
while (1) {
  print "start\n";
  TEST: {
    if (($a = $a / 2) > 2) {
      print "$a\n";
      if (—$a < 2) {
        exit;
      }
```

```
      redo TEST;
    }
  }
}
```

In this simple example, the variable `$a` is repeatedly manipulated and tested in a loop. The word *start* will be printed only once.

`do ... while` and `do ... until`

The `while` and `until` loops evaluate the conditional first. The behavior is changed by applying a `do` block before the conditional. With the `do` block, the condition is evaluated last, which results in the contents of the block always executing at least once (even if the condition is false). This is similar to the C language `do ... while (conditional)` statement.

Regular Expressions

Perl's greatest strength is in text and file manipulation, which is accomplished by using the *regular expression (regex)* library. Regexes, which are quite different from the wildcard-handling and filename-expansion capabilities of the shell (see Chapter 13, "Automating Tasks and Shell Scripting"), allow complicated pattern matching and replacement to be done efficiently and easily.

For example, the following line of code replaces every occurrence of the string `bob` or the string `mary` with `fred` in a line of text:

```
$string =~ s/bob|mary/fred/gi;
```

Without going into too many of the details, Table 44.7 explains what the preceding line says.

TABLE 44.7 Explanation of `$string =~ s/bob|mary/fred/gi;`

Element	Explanation
`$string =~`	Performs this pattern match on the text found in the variable called `$string`.
`s`	Substitute.
`/`	Begins the text to be matched.
`bob|mary`	Matches the text `bob` or `mary`. You should remember that it is looking for the text `mary`, not the word `mary`; that is, it will also match the text `mary` in the word `maryland`.
`fred`	Replaces anything that was matched with the text `fred`.
`/`	Ends replace text.

Element	Explanation
`g`	Does this substitution globally; that is, replaces the match text wherever in the string you match it (and any number of times).
`i`	The search text is not case sensitive. It matches `bob`, `Bob`, or `bOB`.
`;`	Indicates the end of the line of code.

If you are interested in the details, you can get more information using the regex (7) section of the man page by entering `man 7 regex` from the command line.

Although replacing one string with another might seem a rather trivial task, the code required to do the same thing in another language (for example, C) is rather daunting unless supported by additional subroutines from external libraries.

Access to the Shell

Perl can perform for you any process you might ordinarily perform by typing commands to the shell through the \\ syntax. For example, the code in Listing 44.4 prints a directory listing.

LISTING 44.4 Using Backticks to Access the Shell

```
$curr_dir = 'pwd';
@listing = 'ls -al';
print "Listing for $curr_dir\n";
foreach $file (@listing) {
     print "$file";
}
```

NOTE

The \\ notation uses the backtick found above the Tab key (on most keyboards), not the single quotation mark.

You can also use the `Shell` module to access the shell. `Shell` is one of the standard modules that comes with Perl; it allows creation and use of a shell-like command line. Look at the following code for an example:

```
use Shell qw(cp);
cp ("/home/httpd/logs/access.log", :/tmp/httpd.log");
```

This code almost looks like it is importing the command-line functions directly into Perl. Although that is not really happening, you can pretend that the code is similar to a command line and use this approach in your Perl programs.

A third method of accessing the shell is via the `system` function call:

```
$rc = 0xffff & system('cp /home/httpd/logs/access.log /tmp/httpd.log');
if ($rc == 0) {
  print "system cp succeeded \n";
} else {
  print "system cp failed $rc\n";
}
```

The call can also be used with the `or die` clause:

```
system('cp /home/httpd/logs/access.log /tmp/httpd.log') == 0
  or die "system cp failed: $?"
```

However, you cannot capture the output of a command executed through the `system` function.

Modules and CPAN

A great strength of the Perl community (and the Linux community) is the fact that it is an open-source community. This community support is expressed for Perl via the *Comprehensive Perl Archive Network (CPAN)*, which is a network of mirrors of a repository of Perl code.

Most of CPAN is made up of *modules*, which are reusable chunks of code that do useful things, similar to software libraries containing functions for C programmers. These modules help speed development when building Perl programs and free Perl hackers from repeatedly reinventing the wheel when building a bicycle.

Perl comes with a set of standard modules installed. Those modules should contain much of the functionality that you will initially need with Perl. If you need to use a module not installed with Ubuntu, use the CPAN module (which is one of the standard modules) to download and install other modules onto your system. At www.perl.com/CPAN, you will find the CPAN Multiplex Dispatcher, which will attempt to direct you to the CPAN site closest to you.

Typing the following command puts you into an interactive shell that gives you access to CPAN. You can type `help` at the prompt to get more information on how to use the CPAN program:

```
matthew@seymour:~$ perl -MCPAN -e shell
```

After installing a module from CPAN (or writing one of your own), you can load that module into memory where you can use it with the use function:

```
use Time::CTime;
```

`use` looks in the directories listed in the variable `@INC` for the module. In this example, `use` looks for a directory called `Time`, which contains a file called `CTime.pm`, which in turn is

assumed to contain a package called `Time::CTime`. The distribution of each module should contain documentation on using that module.

For a list of all the standard Perl modules (those that come with Perl when you install it), see `perlmodlib` in the Perl documentation. You can read this document by typing `perldoc perlmodlib` at the command prompt.

Code Examples

The following sections contain a few examples of things you might want to do with Perl.

Sending Mail

You can get Perl to send email in several ways. One method that you see frequently is opening a pipe to the `sendmail` command and sending data to it (shown in Listing 44.5). Another method is using the `Mail::Sendmail` module (available through CPAN), which uses socket connections directly to send mail (as shown in Listing 44.6). The latter method is faster because it does not have to launch an external process. Note that `sendmail` must be running on your system for the Perl program in Listing 44.5 to work.

LISTING 44.5 Sending Mail Using `Sendmail`

```
#!/usr/bin/perl
open (MAIL, "| /usr/sbin/sendmail -t"); # Use -t to protect from users
print MAIL <<EndMail;
To: you\
From: me\
Subject: A Sample Email\nSending email from Perl is easy!\n
.
EndMail
close MAIL;
```

NOTE

The `@` sign in the email addresses must be escaped so that Perl does not try to evaluate an array of that name. That is, `dpitts@mk.net` will cause a problem, so you need to use `dpitts\<indexterm startref="iddle2799" class="endofrange" significance="normal"/>:}].`

The syntax used to print the mail message is called a *here document*. The syntax is as follows:

```
print <<EndText;
.....
EndText
```

The `EndText` value must be identical at the beginning and at the end of the block, including any whitespace.

LISTING 44.6 Sending Mail Using the `Mail::Sendmail` Module

```
#!/usr/bin/perl
use Mail::Sendmail;

%mail = ( To => "you@there.com",
  From => "me@here.com",
  Subject => "A Sample Email",
  Message => "This is a very short message"
  );

sendmail(%mail) or die $Mail::Sendmail::error;

print "OK. Log says:\n", $Mail::Sendmail::log;

use Mail::Sendmail;
```


Perl ignores the comma after the last element in the hash. It is convenient to leave it there; if you want to add items to the hash, you do not need to add the comma. This is purely a style decision.

USING PERL TO INSTALL A CPAN MODULE

You can use Perl to interactively download and install a Perl module from the CPAN archives by using the `-M` and `-e` commands. Start the process by using Perl like this:

```
# perl -MCPAN -e shell
```

When you press Enter, you see some introductory information, and you are asked to choose an initial automatic or manual configuration, which is required before any download or install takes place. Type `no` and press Enter to have Perl automatically configure for the download and install process; or if you want, just press Enter to manually configure for downloading and installation. If you use manual configuration, you must answer a series of questions regarding paths, caching, terminal settings, program locations, and so on. Settings are saved in a directory named `.cpan` in the current directory.

When finished, you see the CPAN prompt:

```
cpan>
```

To have Perl examine your system and then download and install a large number of modules, use the `install` keyword, specify Bundle at the prompt and then press Enter, like this:

```
cpan> install Bundle::CPAN
```

To download a desired module (using the example in Listing 44.6), use the `get` keyword like this:

```
cpan> get Mail::Sendmail
```

The source for the module is downloaded into the `.cpan` directory. You can then build and install the module using the `install` keyword, like this:

```
cpan> install Mail::Sendmail
```

The entire process of retrieving, building, and installing a module can also be accomplished at the command line by using Perl's `-e` option, like this:

```
# perl -MCPAN   -e "install Mail::Sendmail"
```

Note also that the `@` sign did not need to be escaped within single quotation marks (' '). Perl does not *interpolate* (evaluate variables) within single quotation marks but does within double quotation marks and here strings (similar to `<<` shell operations).

Purging Logs

Many programs maintain some variety of logs. Often, much of the information in the logs is redundant or just useless. The program shown in Listing 44.7 removes all lines from a file that contain a particular word or phrase, so lines that you know are not important can be purged. For example, you might want to remove all the lines in the Apache error log that originate with your test client machine because you know those error messages were produced during testing.

LISTING 44.7 Purging Log Files

```
#!/usr/bin/perl
# Be careful using this program!
# This will remove all lines that contain a given word
# Usage:  remove <word> <file>
$word=@ARGV[0];
$file=@ARGV[1];
if ($file)  {
  # Open file for reading
  open (FILE, "$file") or die "Could not open file: $!";      @lines=<FILE>;
  close FILE;
  # Open file for writing
  open (FILE, ">$file") or die "Could not open file for writing: $!";
  for (@lines)  {
    print FILE unless /$word/;
  } # End for
  close FILE;
} else {
  print "Usage:  remove <word> <file>\n";
}  #  End if...else
```

The code uses a few idiomatic Perl expressions to keep it brief. It reads the file into an array using the `<FILE>` notation; it then writes the lines back out to the file unless they match the pattern given on the command line.

The `die` function kills program operation and displays an error message if the `open` statements fail. `$!` in the error message, as mentioned in the section on special variables, is the error message returned by the operating system. It will likely be something like `'file not found'` or `'permission denied'`.

Posting to Usenet

If some portion of your job requires periodic postings to Usenet—an FAQ listing, for example—the following Perl program can automate the process for you. In the code example, the posted text is read in from a text file, but your input can come from anywhere.

The program shown in Listing 44.8 uses the `Net::NNTP` module, which is a standard part of the Perl distribution. You can find more documentation on the `Net::NNTP` module by entering `'perldoc Net::NNTP'` at the command line.

LISTING 44.8 Posting an Article to Usenet

```
#!/usr/bin/perl
# load the post data into @post
open (POST, "post.file");
@post = <POST>;
close POST;
# import the NNTP module
use Net::NNTP;
$NNTPhost = 'news';
# attempt to connect to the remote host;
# print an error message on failure
$nntp = Net::NNTP->new($NNTPhost)
  or die "Cannot contact $NNTPhost: $!";
# $nntp->debug(1);
$nntp->post()
  or die "Could not post article: $!";
# send the header of the post
$nntp->datasend("Newsgroups: news.announce\n");
$nntp->datasend("Subject: FAQ - Frequently Asked Questions\n");
$nntp->datasend("From: ADMIN <root>\n"
$nntp->datasend("\n\n");
# for each line in the @post array, send it
for (@post)     {
  $nntp->datasend($_);
} #  End for
$nntp->quit;
```

One-Liners

One medium in which Perl excels is the one-liner. Folks go to great lengths to reduce tasks to one line of Perl code. Perl has the rather undeserved reputation of being unreadable. The fact is that you can write unreadable code in any language. Perl allows for more than one way to do something, and this leads rather naturally to people trying to find the most arcane way to do things.

Named for Randal Schwartz, a *Schwartzian* transform is a way of sorting an array by something that is not obvious. The sort function sorts arrays alphabetically; that is pretty obvious. What if you want to sort an array of strings alphabetically by the third word? Perhaps you want something more useful, such as sorting a list of files by file size? A Schwartzian transform creates a new list that contains the information that you want to sort by, referencing the first list. You then sort the new list and use it to figure out the order that the first list should be in. Here's a simple example that sorts a list of strings by length:

```
@sorted_by_length =
  map { $_ => [0] }            # Extract original list
  sort { $a=>[1] <=> $b=>[1] } # Sort by the transformed value
  map { [$_, length($_)] }     # Map to a list of element lengths
  @list;
```

Because each operator acts on the thing immediately to the right of it, it helps to read this from right to left (or bottom to top, the way it is written here).

The first thing that acts on the list is the `map` operator. It transforms the list into a hash in which the keys are the list elements and the values are the lengths of each element. This is where you put in your code that does the transformation by which you want to sort.

The next operator is the `sort` function, which sorts the list by the values.

Finally, the hash is transformed back into an array by extracting its keys. The array is now in the desired order.

Command-Line Processing

Perl is great at parsing the output of various programs. This is a task for which many people use tools such as `awk` and `sed`. Perl gives you a larger vocabulary for performing these tasks. The following example is very simple but illustrates how you might use Perl to chop up some output and do something with it. In the example, Perl is used to list only those files that are larger than 10KB:

```
matthew@seymour:~$ ls -la | perl -nae 'print "$F[8] is $F[4]\n" if $F[4] > 10000;'
```

The `-n` switch indicates that I want the Perl code run for each line of the output. The `-a` switch automatically splits the output into the `@F` array. The `-e` switch indicates that the Perl code is going to follow on the command line.

RELATED UBUNTU AND LINUX COMMANDS

You will use these commands and tools when using Perl with Linux:

- **a2p**—A filter used to translate `awk` scripts into Perl
- **find2perl**—A utility used to create Perl code from command lines using the `find` command
- **perldoc**—A Perl utility used to read Perl documentation
- **s2p**—A filter used to translate sed scripts into Perl
- **vi**—The `vi` (actually `vim`) text editor

References

- *Sams Teach Yourself Perl in 21 Days, Second Edition*, by Laura Lemay, Sams Publishing, ISBN: 0-672-32035-5.
- *Sams Teach Yourself Perl in 24 Hours, Second Edition*, by Clinton Pierce, Sams Publishing, ISBN: 0-672-32793-7.
- *Learning Perl, Third Edition*, by Randal L. Schwartz, Tom Phoenix, O'Reilly & Associates, ISBN: 1-449-30358-7.
- *Programming Perl, Third Edition*, by Larry Wall, Tom Christiansen, and Jon Orwant, O'Reilly & Associates, ISBN: 0-596-00492-3.
- *Effective Perl Programming: Writing Better Programs with Perl*, by Joseph Hall, Addison-Wesley Publishing Company, ISBN: 0-321-49694-9.
- *Mastering Regular Expressions*, by Jeffrey Friedl, O'Reilly & Associates, ISBN: 0-596-52812-4.
- **www.perl.com**—This is the place to find all sorts of information about Perl, from its history and culture to helpful tips. This is also the place to download the Perl interpreter for your system.
- **http://cpan.perl.org**—CPAN is the place for you to find modules and programs in Perl. If you write something in Perl that you think is particularly useful, you can make it available to the Perl community here.
- **http://perldoc.perl.org/index-faq.html**—FAQ index of common Perl queries; this site offers a handy way to quickly search for answers about Perl.
- **http://learn.perl.org**—One of the best places to start learning Perl online. If you master Perl, go to http://jobs.perl.org.
- **www.pm.org**—The Perl Mongers are local Perl user groups. There might be one in your area. The Perl advocacy site is www.perl.org.