

Bonus Chapter C

The Perl Programming Language

Sitting between shell and C, and influenced by many of the standard Linux tools, Larry Wall's Perl programming language is ideally suited to text manipulation, CGI scripting, and system administration tasks. However, as we will see, Perl is very extensible (you can even write graphical user interfaces in it using the PerlTk extensions). It would be fair to say that anything you can do in C, you can do in Perl, and probably much more easily, too. Perl is particularly easy to learn, since it borrows elements from a whole host of programming languages and utilities which will probably already be familiar to you; C and shell programmers will feel particularly at home with it. sed, awk, Basic, and Tcl programmers will not feel left out either.

One of the nicest features of Perl is that it provides a platform-independent Linuxlike abstraction layer on top of your operating system. What does this mean? Perl runs on a huge variety of platforms, including Windows, the Apple Macintosh, and, of course, anything that looks like Linux. You can bring over all of the ideas about Linux programming we've seen so far, and Perl will implement them as well as it can on whatever we're running on. In effect, you can pretend any target operating system looks and behaves exactly like Linux does, which, as I'm sure you'll agree, is such a pleasant way of looking at things.

In this chapter, we'll be looking at how to write basic Perl scripts, carrying over what you've learned so far into Perl. We can't deal with the whole of the Perl language in this chapter, but we'll examine the most useful and most commonly used areas, as well as how we'd implement our familiar CD database application in Perl.

An Introduction to Perl

Let's start off by looking at the basic features of the Perl language: variables, operators and functions, regular expressions, and file input and output. There's a lot of information coming over the next few pages, and we won't really put it all together until the end. However, you should be familiar with all of the concepts coming up, so hopefully, it shouldn't be too heavy reading.

First, I suppose, we ought to check to make sure you've got a copy of Perl on your system. Most Linux distributions come with Perl these days, so you should be able to just type `perl -v` and get something like the following:

```
This is perl, version 5.005_03 built for i386-linux
```

```
Copyright 1987-1999, Larry Wall
```

```
Perl may be copied only under the terms of either the Artistic License or the GNU  
General Public License, which may be found in the Perl 5.0 source kit.
```

Complete documentation for Perl, including FAQ lists, should be found on this system using 'man perl' or 'perldoc perl'. If you have access to the Internet, point your browser at <http://www.perl.com/>, the Perl Home Page.

If that doesn't happen, check your path and your Linux package manager, if you have one, to see if you've actually got Perl installed; if not, you can get a recent copy from your Linux CD, the contrib or main sections of your Linux distributor's FTP site, or in source form from CPAN (which we'll describe later). As it says, <http://www.perl.com/> is the Perl Home Page, and contains a wealth of documentation, news, and generally useful information about Perl.

Hello, Perl

First of all, let's see what a simple Perl program looks like, and how we run it. Here's our script, which we'll call `hello.pl`:

```
print "Hello, World\n";
```

Now, we could run this through the perl interpreter from the command line:

```
$ perl hello.pl
Hello, World
$
```

As expected, Perl says "hello" and returns us to our shell prompt. Just like when we were writing shell scripts, we can use the `#!` convention to tell Linux where the interpreter is installed. Let's change `hello.pl` so that it looks like this:

```
#!/usr/bin/perl -w
# hello.pl, version 2
print "Hello, World\n";
```

(You might need to change the `#!/usr/bin/perl` to the actual location of your perl binary. Use `which perl`, or your shell's equivalent, to find out where it resides.)

Next, we need to make the file executable before we can run it:

```
$ chmod 755 hello.pl
```

Now, we can run the file as we did with shell scripts:

```
$ ./hello.pl
```

Some things to note from this example:

1. Perl statements, like C statements, end with a semicolon. Also as in C, newlines in strings are represented by `\n`. All of the other string "metacharacters" are borrowed wholesale from C.
2. Just like shell and Tcl, we're allowed to have comments starting with hashes. (Although, just like shell, `#!` has a special meaning.)
3. The `-w` in the first line is an option to the perl interpreter, which turns on extra warnings. This is highly recommended; although Perl allows us to get away with a lot of sloppy coding, our scripts will usually run more reliably if they pass the `-w` test.

Perl Variables

Perl has three variable types: scalars, arrays, and hashes.

Scalars

Scalar variables are ordinary strings and numbers; these look and operate a lot like shell variables. We could rewrite our script, should we want to, like this:

```
$message = "Hello, World\n";  
print $message;
```

C programmers might feel happier saying `print($message)`, and you can do that too. Perl is very relaxed about the need for braces around functions like `print`. Use this to increase readability, not decrease it. Similarly, unlike shell but like C, we can have spaces around the equals sign of an assignment statement if we want them. Note that unlike C, however, we don't need to allocate and deallocate space for our variables, nor do we need to declare them in advance; Perl takes care of all that, leaving you free to concentrate on getting the program written efficiently.

C programmers may also be happier using `printf` instead of `print`; Perl provides you with a `printf` function, which works just like the C and shell functions of the same name. However, `print` is more efficient, so unless you need the formatting capabilities of `printf`, use `print` where you can.

Arrays

Arrays are the variable incarnation of **lists**, just like in Tcl. Instead of starting with a `$`, array variables are denoted by an `@`. Here's a simple list:

```
(1, 2, 3, 4)
```

and here's how we put it into a variable:

```
@mylist = (1, 2, 3, 4);
```

Now, what if we want to get things out of it again? Let's have a look at a simple list in operation, then we'll explain what's going on.

```
@message = ("\n", " ", "World", "Hello,");  
print $message[3], $message[1], $message[2], $message[0];
```

OK, you know exactly what it's going to say, but how does it do it? The first thing to note is that we can have multiple parameters to `print` separated by commas. These actually form a **list**, but that's not the important part. (It'll get important later.)

Next, you'll have noted that our array is called `@message`, but we referenced it with `$message[element]`. This makes sense if you think about it from the point of view of "what you want," not "what you've got" — when we're pulling elements out one-by-one, we actually "want" scalar values from it, not arrays. (Don't be tricked into thinking that `$message` and `@message` refer to the same variable, though — they're independent; `$message` on its own doesn't refer to any elements in the array.)

Finally, the first array element is 0; fine if you're used to C, but possibly disturbing if you program BASIC, Pascal, or `sed/awk`. You can change the number of the first array element to 1 if you want, but the effects can be quite horrific, so it's best to leave it alone.

You may not want to grab individual elements, but rather a range, or *array slice*. You can do this by giving a range of elements, and remembering that this time we do want a list, not a scalar:

```
@a = ("zero", "one", "two", "three", "four");  
@b = @a[0,2..3]; # @b is ("zero", "two", "three");
```

```
# Reducing it to one statement:
@b = ("zero", "one", "two", "three", "four")[0,2..3];
```

Perl automatically **flattens** lists; that is to say, lists cannot contain other lists, nor can arrays be multi-dimensional. (You can do that with references, but that's beyond the scope of this chapter; see the `perlref` and `perl101` manpages for more on how to do that.) This means that the following statements are equivalent:

```
@a = ("zero", "one", "two", "three", "four");
@a = ("zero", "one", ("two", "three"), "four");
@half = ("zero", "one", "two"); @half2 = ("three", "four");
@a = (@half, @half2);
```

You're also allowed to have lists on the left-hand side of an assignment, and, in fact, this is a very useful thing to do:

```
($first, $last) = ("alpha", "omega");
@a = ("alpha", "omega") ; ($first, $last) = @a; # Same thing.
```

Things that are put on the left-hand side of an assignment are called "lvalues"; we say lists are "lvaluable." One really flash use of this is to swap the values of two variables without using a temporary variable. Here's the C programmer's way to do it:

```
$temp = $last;
$last = $first;
$first = $temp;
```

Whereas a native would say:

```
($first, $last) = ($last, $first);
```

Hashes

The final variable type we can have are **hashes**, also known as associative arrays. These allow you to store and retrieve data by keys, and the symbol for a hash is `%`. Hashes are very handy for representing relationships between data. Here's a simple hash in use to store and retrieve phone numbers:

```
%phonebook = ( "Bob" => "247305",
               "Phil" => "205832",
               "Sara" => "226010" );
```

We created our hash just like a list, but used the `=>` operator (which is equivalent to a comma in most cases) to separate our key-value pairs. Now, let's get at an entry:

```
print "Sara's phone number is ", $phonebook{"Sara"}, "\n";
print "Bob's phone number is ", $phonebook{Bob}, "\n";
```

Again, we want a scalar, so we use the dollar sign. To signify which key we want to reference, we use braces rather than square brackets, and the second line illustrates the fact that we don't need to put quotes around the key. Be careful, though: hash keys, just like variable names, are case sensitive:

`$phonebook{Bob}` is the same as `$phonebook{"Bob"}` but not the same as `$phonebook{bob}`.

Once we've got this far, changing an entry in the hash is easy enough:

```
$phonebook{Bob} = "293028";
print "Bob's new number is ", $phonebook{Bob}, "\n";
```

Quoting and Substitution

Just like in shell scripting, variables inside double-quoted strings are substituted unless escaped with backslashes. We could rewrite our phonebook example like this:

```
%phonebook = ( "Bob" => "247305",
               "Phil" => "205832",
               "Sara" => "226010" );
print "Sara's phone number is $phonebook{Sara}\n";
print "Bob's phone number is $phonebook{Bob}\n";
```

(Don't put double-quoted strings inside double-quoted strings, obviously.)

As you might expect, as in the shell, single quotes don't cause substitution to happen or special characters like `\n` to work:

```
$myvar = "quoting";
print 'Take care when $myvar strings\n';
print "Take care when $myvar strings\n";
```

gives the following output:

```
Take care when $myvar strings\nTake care when quoting strings
```

As in Tcl, Perl converts between numbers and strings on demand:

```
print "4 bananas"+1; # Gives "5"
print "123" + "456"; # Gives "579"
```

Special Variables

Perl has a huge number of special variables, and we'll come across a few of them as we continue, but the most important three are these: `$_` is the default scalar for a lot of operations and functions. For instance, `print` with no arguments prints the value of `$_`. This can be really useful, but can also lead to code like this:

```
# Strip comment lines.
while (<>) { print unless /^#/ }
```

The variable `$_` was used three times there—once to get a line from standard input (the `<>` or `readline` operator, which we'll cover later; but for the moment, remember what it does), once as the argument to `print`, and once to test against whether it started with a hash sign. (Again, we'll explain the tricky bit `/^#/` when we come to regular expressions later.) To a seasoned Perl programmer, this sort of code is bread-and-butter, but it can scare off newcomers. Don't omit `$_` in your own code unless your intention is clear.

There's also the `@ARGV` array, which contains the arguments to your program; unlike in C, you don't get `argv[0]` as the name of your script. (That comes from `$0`.) Instead, `$ARGV[0]` is the first argument.

Finally, the `%ENV` hash allows you to inspect and change environment variables, as you would in shell programming, like this:

```
print $ENV{PATH}; # /usr/local/bin:/usr/bin...
print $ENV{EDITOR}; # vi
$ENV{EDITOR} = "emacs"; # change to emacs for the rest of the program.
```

Operators and Functions

In Perl, the ideas of “function” and “operator” aren’t very clearly separated; some things you might think of as functions turn out to be operators, and vice versa. Because of this, we’re not really going to make the distinction here. Let’s use “function” and “operator” interchangeably.

Numeric Operators

What things can we do to numbers? Well, the first four are pretty guessable: +, -, *, and / all do what you would expect, and can be combined with brackets in the usual way. Rules of precedence are pretty much the same as in C.

```
$a = (4*5)+3; # 23
$b = 1/(4+4); # 0.125
$c = 1/4+4; # 4.25
```

We can also use the remainder operator (also known as the *modulus* operator) %. However, be sure you’re dealing with positive numbers when you use this; if you have `$a % -$b`, the result is `($a % $b) - $a`, which might not be what you expected. There’s also the *exponentiation* operator, `**`.

```
$a = 17 % 5; # 17 into 5 goes 3 times remainder 2, so $a = 2
$a = 22 % -7; # 22 into 7 goes 3 times remainder 1, so $a = 1-7 = -6
$a = 2 ** 8; # 2 raised to the 8 is 256
```

For variables, we can use pre- and post-fix decrement and increment, (although these can operate differently on strings) as in C.

```
$a = 5; $b = 7;
$c = ++$a + $b; # $c is 6+7 = 13
```

Then there are the normal scientific functions: trigonometric `sin` and `cos`, `sqrt`, and `log` for square roots and natural logarithms, and so on. The “perlop” and “perlfunc” manpages will tell you all about these.

String Operators

One of the most common things we’ll do to strings is to concatenate them, and we do this with the dot operator. Auto-conversion between numbers and strings takes place as normal:

```
$a = "foo" . "bar" . "baz" ; # Gives us "foobarbaz"
$a = "number " . 1 ; # Gives "number 1"
$a = "1" . "2" ; # Gives "12"
```

We might also want to repeat a string a given number of times. This is done with the cross, or `x` operator:

```
$a = "ba".("na"x4) ; # "banananana"
$a = 1 x 3 ; # 111
```

The second example gives us a good reason not to confuse `x` and `*`.

Another common thing to do to a string is to remove the last character. The `chop` function removes the last character, no matter what it might be; `chomp` is more subtle, and removes the input record separator (usually a newline) from the end of the string. This is superb when you’re reading input in from text files, where you don’t want to blindly assume that the last character you read will be a newline. `chop` and `chomp` both return the new string and change the variable you gave them.

```
$a="bite me\n";
chomp($a); # $a is now "bite me"
chomp($a); # $a is still "bite me" since there's no newline
chop($a) # $a is now "bite m"
```

If our strings are explicitly alphabetic, incrementing them works as you might expect—the ASCII value of the last character is increased, wrapping from “z” or “Z” to “a.” If the strings start with a number, however, Perl will convert them to a numeral and the rest will be lost.

```
$a = "abc"; print ++$a; # Returns "abd"
$a = "azz"; print ++$a; # Returns "baa"
$a = "0 Goodbye Cruel World"; print ++$a; # Returns 1
```

As well as joining strings together, it’s also very useful to split them up, which is done with the `split` function. You can split on a string or on a regular expression; if you give no arguments, it splits `$_` on whitespace; you can give as an optional first argument, a pattern to split on, and as an optional second argument, the text to split. Note that if you give a string to split on, you must also supply a pattern as the first argument; a common mistake is to omit the pattern but leave the text.

```
# Split $_ on whitespace
$_ = "one two three";
@a = split; # ("one", "two", "three")

$passwd="simon:x:500:500:Simon Cozens,,,:/home/simon:/bin/zsh";

@passwd= split ":", $passwd; # Split on a string.
# Can also write it like this:
@passwd= split /:/ $passwd; # Split on a regular expression.(Same thing)
($uid, $gid) = @passwd[2,3];

# More idiomatically:
($uid, $gid) = (split ":", $passwd)[2,3];
```

The exact opposite—converting a list to a string—can be done with `join`. Again, you can join things together with delimiters:

```
@mylist = ("one", "two", "three", "four");
$mystring = join "?", @mylist; # one?two?three?four
```

Anyone familiar with BASIC will know what `substr` does; it returns a substring from inside the string. You must supply two parameters: the string and the offset. These two alone will get you everything from the offset to the end of the string. (A negative offset means “count back from the end of the string”: -1 is the last character, -2 the last but one, and so on.) Supplying another parameter, the length, will get you a maximum of that many characters.

```
$string="the glistening trophies";
print substr($string, 15); # trophies
print substr($string, -3); # ies
print substr($string, -18, 9); # listening
print substr($string, 4, 4); # glis
```

You can also use `substr` to modify a string, either by giving it another parameter:

```
substr($string, 7, 4, "tter"); # Returns "sten"
print $string; # the glittering trophies
```

(Note that this changes the string and returns what was in its place) or, the more idiomatic way, by the assignment:

```
substr($string, 7, 4) = "tter"; # Functions as lvalues.
```

Let's quickly run through the final few functions since they're simple enough. `length` does what you expect: returns the length of a string. (You can't truncate a string by assigning to its length, though.) `reverse` returns the string in reverse. However, `reverse` is usually a list operation; it treats its parameters as a list, and returns the list in reverse order. If we feed it a string, though, it treats it as a one-element list – which is the same in reverse order. To get what we expect, we need to force it into *scalar* operation, with the `scalar` keyword.

```
$a="Just Another Perl Hacker";
print length $a; # 24
print reverse $a; # list context - "Just Another Perl Hacker"
print scalar reverse $a; # "rekcaH lreP rehtonA tsuJ"
```

Finally, `uc` and `lc` convert to upper and lowercase, respectively, and `ucfirst` and `lcfirst` convert the first letter of the string to upper and lowercase.

```
$zippy="YOW!! I am having FUN!!";
print uc($zippy); # YOW!! I AM HAVING FUN!!
print lc($zippy); # yow!! i am having fun!!
print ucfirst(lc($zippy)); # Yow!! i am having fun!!
print lcfirst(uc($zippy)); # yow!! I AM HAVING FUN!!
```

Logical and Bitwise Operators

The ordinary bitwise operators `&`, (and) `|`, (or) `^` (xor), and `~` (not), as well as the bit shift operators `>>` and `<<` all work on integers as you'd expect from C; you can even use the `0x` and `0` prefixes for hexadecimal and octal numbers.

```
0xF0 | 0x0F = 255 (0xFF)
0xAA ^ 0x10 = 186 (0xBA)
```

The logical operators `&&`, `||`, and `!` also work as in C, but you can also use the English “and”, “or,” and “not.” Because of Perl's short-circuit evaluation, these are often used as control structures:

```
risky_function()
  and print "Worked fine\n"
or print "Function didn't succeed\n";

# Also written as:
risky_function() && print "Worked fine\n"
|| print "Function didn't succeed\n";
```

You can also use `if` and `unless` in the same way:

```
print "Worked fine\n" if risky_function();
$a="Default value" unless $a;
```

You can obtain your true or false values either from scalars (“0” and undefined are false values – everything else is true) or from comparisons, of which there are many. You can compare numbers with the standard `<`, `>`, `==` (`=` is assignment; don't confuse the two.), and `!=`, but for strings you have to use a special set of comparisons: `lt` for lexicographically less, `gt` for greater, `eq` for equality, and `ne` for inequality.

Array Operations

We've seen most of the things we can do with scalars; that is, strings and numbers. What about arrays and lists?

One important thing you'll want to do with an array is to find the number of elements in it; you might think about using `length`, but this won't work. Instead, we have to evaluate the array in scalar context,

like we did when reversing strings. Notice, however, that Perl doesn't support "sparse arrays"; the array is assumed to be filled with elements that are undefined.

```
@array = ("zero","one","two","three");
print scalar @array; # 4 elements in the array

$array[200] = "two hundred";
print scalar @array; # 201 elements; some of them are empty, though.
```

Perl also allows us to get at the index of the highest-numbered element. This is usually one less than the number of elements in the array, as we start counting from zero:

```
@array = ("zero","one","two","three");
print $#array; # 3
```

Next, we can take elements off the array. We can either think of it as an array in shell programming and `shift` the elements off the front of it, or as a stack, and `pop` elements off the end.

```
@array = ("zero","one","two","three");

print shift @array; # zero - array is now ("one","two","three")
print pop @array; # three - array is now ("one","two")
print shift @array; # one - array is now ("two")
print pop @array; # two - array is now ()
```

Similarly, we can put things back; either with `unshift` or `push`. We can put things on several elements at a time.

```
@array = ();
push @array, "two"; # array is now ("two")
unshift @array, "one"; # array is now ("one","two")
push @array, "three", "four"; # array is now ("one","two","three","four")
unshift @array, "minus one","zero";
print join " ", @array;
# minus one,zero,one,two,three,four
```

Now, we can demonstrate the list use of `reverse`:

```
print join " ", reverse @array;
# four,three,two,one,zero,minus one
```

We can sort lists into ASCII order with `sort`;

```
@a = ("delta","alpha","charlie","bravo");
@b = sort @a; # ("alpha","bravo","charlie","delta");
```

You can also pick your own sort order by using the special BLOCK form; Perl sets `$a` and `$b` to be the two values under comparison. In this case, for numerical values, we use the special comparison operator `<=>`, which returns -1 if the left is bigger than the right, 0 if they are equal, and 1 if the right is bigger than the left.

```
@a = (5, 8, 3, 0, 1);
@b = sort {$a <=> $b} @a; #(0, 1, 3, 5, 8)
```

Hash Operations

Finally, we can do various things to hashes. The most magical is the `reverse` operation, which reverses the lookup tables:

```
%phonebook = ( "Bob" => "247305",
```

```

        "Phil" => "205832",
        "Sara" => "226010" );
%index = reverse %phonebook;
print $index{"226010"}; # Sara

```

You must be careful that you do not end up with two keys with the same value—one will get lost. You can also extract a list of the keys and the values with the `keys` and `values` functions:

```

@names = keys %phonebook; # ("Bob", "Phil", "Sara")
@numbers = values %phonebook; # ("247305", "205832", "226010")

```

You can also get key-value pairs with `each`; every call to `each` returns a two-element list consisting of a new key and its value, until there are no more left.

```

while ( ($key, $value) = each %phonebook ) {
    print "$key's phone number is $value\n";
}

```

Don't be fooled into thinking that the order you put items into a hash is the same as the order you'll get them out. Perl returns items in a seemingly random order—in fact, you can't guarantee that you'll get the same order twice.

Regular Expressions

Regular expressions (insiders call them “regexps”) are one of Perl's greatest strengths. They allow for extremely powerful pattern matching and substitution, and are probably the biggest weapon in the Perl programmer's armory. Those familiar with `sed` and `awk` or `egrep` will know the general principles of regular expressions; Perl's regular expression engine encompasses and extends the `sed` model.

First, we'll look at how we use regular expressions to match strings; then we'll see how to substitute with them.

Matching

The most basic regular expression is just a piece of text we want to find inside a string. We traditionally enclose regexps in forward slashes, like this: `/regexp/`, and we “apply” a regexp to a variable or scalar using the syntax `$scalar =~ /regexp/`. This functions as an operator which returns true if the match was successful. (There's also `!~`, which is like `=~`, only negated—returns true if the match failed.) So, let's see if our string contains “jaws”:

```

$sea = "water sand Jaws swimmers";
print "Shark alert!" if $sea =~ /jaws/;

```

Well, that didn't print anything, because regular expressions are case sensitive by default; we can turn this off with the *modifier* `i` like this:

```

print "Shark alert!" if $sea =~ /jaws/i;

```

Next, we can have special characters in our regexp: `^` stands for “beginning of string”—that is, the expression we're trying to find must be the first thing in the string. Similarly, `$` represents the end of the string; it should, for obvious reasons, come at the end of the regexp. If we tell you that `/regexp/` on its own tests the `$_` variable, we're now in a position to explain the confusing code we used to warn against implicit use of `$_` a few pages ago:

```

# Strip lines starting with a hash
while (<>) { print unless /^#/ }

```

To write it explicitly:

```
# Strip lines starting with a hash.
while ($_ = <>) { print $_ unless $_ =~ /^#/ }
```

In other words, while `$_` is true (i.e., contains something) after being set to the next line of standard input, we print the line unless the line starts with a hash sign.

Now, we come up against the quantifiers: `?` matches a character 0 or 1 times, that is, it states that the character it follows is optional in the match. `*` matches 0 or more times, just like in the shell. (However, be careful with that one: `/q*/` will **always** match, even if there are no “q”s in the string; it matched 0 times.) `+` will match at least once. Here are some examples of quantifiers:

<code>/shoo?t/</code>	matches “shot” or “shoot.”
<code>/sho+t/</code>	matches “shot” or “shoot” or “shoooot,” and so on, but not “sht.”
<code>/sho*t/</code>	matches “sht,” “shot,” “shoot” and so on.

The next set of special characters match different types of character: As well as all our usual metacharacters, `\t` for tab, `\n` for newline and so on, `\s` matches anything that looks like whitespace, `\w` matches a “word” (alphanumeric or “_”) character, and `\d` matches a digit. These can be negated by capitalizing them: `\S` is a nonspace, `\W` is a nonword, and `\D` is a nondigit. Furthermore, `.` matches anything at all. So:

<code>/push\s*chair/</code>	matches “push,” then zero or more spaces or tabs, then “chair.”
<code>/number\s*\d+\s/</code>	matches “number,” then some optional whitespace, then one or more digits followed by a space.
<code>/^e.*d\$/</code>	matches an “e” at the beginning of the line, then anything at all, and a “d” at the end of the line.
<code>/\S/</code>	matches a line that contains something other than just spaces.
<code>/\sPerl\s/</code>	matches the string “Perl” surrounded by space.

The last example is almost always better written as `/\bPerl\b/` – the “word boundary” metacharacter matches nonword characters (allowing spaces, punctuation, and so on) and the beginning or end of the string. Hence, `/\b\w+\b/` matches a “word.”

What if we want to know what was matched? Well, if we place brackets around the part of the regular expression we’re interested in, Perl will save it away for us. Just like `sed`, Perl will put the first bracketed expression into a variable called `$1`, the next into `$2`, and so on. From this, we can get an insight into how the matching process works:

```
$test = "he said she said";
$test =~ /\b(\w+)\b/; # $1 is set to "he"
$test =~ /(sa.*d)/; # $1 is set to "said she said"
```

From the first example, Perl returns the first match it finds from the left, always. (This is called “eagerness.”) From the second, Perl keeps matching from the first match it finds for as long as it can; it returns the first match it finds and seeks to make that as big a match as possible. (This is called “greed.”) You can turn off greed by adding a `?` onto the quantifier – `/(sa.*?d)/` would have just matched “said.” Helpfully, Perl also returns all the bracketed matches as a list, if we’re looking for a list:

```
$test = "he said she said";
```

```
@matches = $test =~ /\s*(\w+)\s*(\w+)\s*(\w+)\s*(\w+)/;
# @matches is ("he", "said", "she", "said");
# (Now you can see that split() actually splits on /\b/)
```

(Of course, this gets tedious for large operations; we'll see another way of doing this later.)

We can also use brackets to give options: `/(boy|girl)/` matches if the string contains either "boy" or "girl." If you don't want this match to populate a special variable, write it like this: `(?:boy|girl)`.

Finally, in this brief tour of regular expressions, we can define character classes—these are delimited by square brackets. For instance, `[a-z]` matches all lowercase letters, and `[aeiou]` matches lowercase vowels. You can put metacharacters inside character classes, and you can also negate character classes by placing a `^` at the beginning of the class: `[^a-zA-Z]` matches anything that is not a letter.

As you might expect, if you want to match any of these characters themselves, like `?`, `.`, `(`, `/`, and so on, they must be escaped with a backslash. `/\.\s*\((\.\?)*\)/` matches a full stop, then zero or more spaces, then an open bracket, then copies text into a variable (we call these "backreferences" for reasons which will very soon become apparent) until it comes to the next close bracket.

Substitution

After matching a string, we might want to substitute the match with some other text. This is done with the syntax: `s/regexp/replacement/`.

```
$test = "he said she said";
$test =~ s/said/did/; # Gives "he did she said"
```

As you can see, this finds the first match, replaces it, and then stops. Well, there's another modifier, `g`, which applies the search-and-replace globally. (This goes for `sed`, and even `vi` and `ed`, too.)

```
$test = "he said she said";
$test =~ s/said/did/g; # Gives "he did she did"
```

You can use the `g` modifier in matches, too, like this:

```
$test= "123 456 7 890";
@array = $test =~ /\b(\d+)\b/g; # (123, 456, 7, 890)
```

Of course, in the replacement text part of the substitution, match metacharacters don't have special meaning. You can't replace a match with "any digit" or "0 or more t's," for instance. Replacement texts work like double-quoted strings, with variable substitution, and so on. (Actually, so do regexps—you can put "sub-regexps" in variables to make the expression tidier.) The really nice part about this is that the backreference variables, `$1`, `$2`, and so on are available here:

```
$test = "Swap this and that.";
$test =~ s/Swap (.*) and (.*)\./Swap $2 and $1./;
print $test; # "Swap that and this."
```

As a final example of regular expressions, let's extend our comment-stripper to remove comments in the middle of lines:

```
# Strip comments, version two.
while (<>) { s/#.*// ; print if /\S/ }
```

What does this do? Well, again we're bringing in a line and storing it in `$_`. Next, we replace a hash mark and everything that follows it with nothing at all. Then, if we've got anything left—that is, if there is a nonblank character in there—we print the line. (Don't actually use this to remove comments from your Perl programs; first, it will remove half of any line which uses the `$#array` syntax or hash signs in strings or regexps, and, second, it's almost always better to leave the comments in.)

There are many, many more things you can do with regular expression matching, but this brief overview provides most of the common uses. Type `perldoc perlre` at the shell prompt to read the full documentation about matching.

Control Structures and Subroutines

We've seen, in passing, a few of the Perl control structures; `while`, the inline use of `if` and `unless`, and so the rest should be pretty easy:

Tests

Perl has, of course, a C-like `if-elsif-else` statement; the only difference is that you **must** wrap blocks in braces. That is, you can't say this:

```
if (/y(?:es)?/i)
    $answer = 1;
else
    $answer = 0;
```

Rather, you have to say

```
if (/y(?:es)?/i) {
    $answer = 1;
} else {
    $answer = 0
}
```

(Of course, in reality, you wouldn't write that anyway; you'd write something like `$answer = /y(?:es)/i`; instead.) Perl also allows you to use `unless` as the negative of `if` if it makes your code clearer.

Loops

We've seen a `while` loop, and you might have been able to guess that an `until` loop is available for the negative of `while`. There are also two types of `for` loop—the standard C one:

```
for ($i=0;$i<10;$i++) { print "Counter: $i\n"; }
```

There's also the `foreach` loop, which iterates over a list, setting either `$_` or a variable, if one is given, to each element in turn. You can say either `foreach` or `for` to get `foreach` loops.

```
for $i (0..9) { print "Counter: $i\n"; } # As above.
foreach (@array) { print $_."\n"; } # Print each element of an array.
```

The interesting part is the way you can control loop flow: a `next` statement, like a `continue` in C, will immediately jump to the next iteration of a `while`-like or `for`-like loop. Here's another comment (and blank line) stripper:

```
while (<>) { next if /^#/; next unless /\S/; print }
```

`last`, on the other hand, finishes a loop altogether like C's `break`:

```
# Get the Subject of an email, then give up:
while (<>) {
    last if ($subj)=/^Subject: (.*)/;
}
print "Subject was $subj\n";
```

The little-used `redo` statement goes to the top of the loop again, without testing the conditional. This is useful if the input changes under your feet:

```
while (<>) {
    if (/\\s*$/) { # Line ends with a backslash - continuation
        $_.=<>;
        redo;
    }
    ...
}
```

One thing you may miss, though, is the C switch (in other languages, case) statement. This is a common concern, and there are many ways to address it; the `perlsyn` and `perlfaq7` documentation discuss some solutions.

Statement Modifiers

As well as the ordinary C-style syntax for control structures, we can also apply a control structure to a single statement by means of a “statement modifier”, or what I call an “inline” control structure. We’ve seen `if` and `unless` used in this way:

```
print "Operation successful" if all_ok();
$logfile = "output.log" unless $nologging;
```

You may also use `while`, `until`, and `for` as statement modifiers:

```
$input .= $_ while <>; # One (bad) way of concatenating a file

$cowscomehome=0;
main_loop() until $cowscomehome;

    print chr for (74, 117, 115, 116, 32, 65, 110, 111, 116, 104, 101,
    114, 32, 80, 101, 114, 108, 32, 72, 97, 99, 107, 101, 114,);
```

Not many people use `for` like that, though.

Subroutines

We don’t make a distinction between *functions* and *subroutines* in Perl; you can return a value from them if you want. Simple subroutines are declared like this:

```
sub greeting {
    print "Hello, world\n";
}
```

You can then use `greeting()`; (or just `greeting`; if you know that Perl isn’t going to try to interpret this as a string) to call the subroutine. Parameter passing is done through the `@_` array:

```
sub action {
    ($one, $two, $three) = @_;
    warn "Not enough parameters!"
        unless $one and $two and $three;
    print "The $one $two on the $three\n";
}

action("cat", "sat", "mat");
```

The `warn` function prints out an error message, but allows your program to continue; there is also a `die` function which prints out an error message and stops the program immediately.

Since @_ is the default variable for array operations, as \$_ is the default for scalars, it's traditional to use shift to get parameters:

```
sub greeting2 {
    $name = shift;
    print "Hello, $name\n";
}

greeting2("Robert");
greeting2("world");
```

You can return values using, predictably, the return function; you may return a scalar or a list—the wantarray function tells you what context the caller is expecting.

```
Sub mysub {
    # Do some stuff
    return unless defined wantarray; # void context, go home early
    if (wantarray) { # Want an array
        return @results;
    } else { # Just want a scalar
        return $summary
    }
    # Can't get here
}
```

Because of the way Perl scoping takes place, variables are usually global—changes you make in subroutines affect the caller. If you want private variables, use the my function. Lexically scoped variables (often abbreviated to “lexical variables” in the Perl documentation) are declared with my, and cannot even be seen in subroutines called from the current one; they are truly private. If you want to scope something for the current subroutine and below, use dynamic global variables.

```
$myvar = "outside";
sub a { $myvar = "changed by a"; }
a();
print $myvar; # "changed by a";
$myvar = "outside";
sub b { my $myvar = "changed by b";
    print "In b: $myvar";
}
b(); # "In b: changed by b"
print $myvar; # "outside"
```

File Input and Output

So far the only file input and output we've seen has been grabbing a line at a time from standard input. While this is fine for writing filters, perhaps, real applications need the ability to read from and write to other files as well.

File access in Perl is usually done through filehandles. When we said there were three variable types, we weren't exactly telling the truth; a filehandle is a very special type of variable. When the program starts, as you'd expect, there are three filehandles open: standard input, STDIN; standard output, STDOUT; and standard error, STDERR. We can write to these filehandles using a special form of print; indeed, 'print list' is just shorthand for 'print STDOUT list'

```
if ($statusok) {
    print STDOUT "Processed successfully.\n";
} else {
    print STDERR "An error occurred...\n";
}
```

Look carefully at that; there is no comma between the filehandle and the text to be output; the filehandle is not part of the list. There are two separate syntaxes to print, one with a filehandle and one without. It's very, very important not to confuse the two.

Now, we've also seen that we can read in a line from standard input, using `<>` – this is, in turn, a shorthand for `<STDIN>`. What about files from the file system, then? We can create a filehandle with the `open` function; just as in C, we give a filehandle and the filename, and state whether we're opening it for input or output. However, instead of assigning a mode number, we use shell-style syntax: "`filename`" (or "`<filename`") to read from the file, "`>filename`" to truncate or create the file and write to it, "`>>filename`" to append to the file, and so on. (You can even open to and from pipes!) Here, we read from a file, noting errors in a log file:

```
open LOG, ">error.log"
    or die "Can't write on error.log: $!";
# $! tells you why not.
print LOG "Error logging started.\n";
open INPUT, $inputfile;
while (<INPUT>) {
    next unless /\S/; # Skip blank lines.
    next if /^#/; # You know what this does.
    chomp;
    if (fictitious_error_generator($_)) {
        print LOG "Error processing $_\n";
    }
    do_something_with($_);
}
close LOG;
close INPUT;
```

Perl will close filehandles for you at the end of the program, but we close them here with the `close` function, just to be polite.

`system()/``

Before we finish our tour of Perl, let's have a look at interacting with the system – that is, running external commands. There are two ways to do this: first `system()`, which works in exactly the same way as the C construct, and backquotes (```), which work in the same way as the shell construct, apart from the way it handles newlines, keeping both sets of programmers happy.

There are, of course, significant differences between the two methods; `system()` suspends the running program and allows user interaction, but does not return the program's `STDOUT`; backquotes return the program's `STDOUT` and don't show it on the screen, so in interactive programs, the user may not be able to see any prompts. `system()` allows you to get at the return value of the executed program (divide the return value of `system()` by 256 or examine `$?`), whereas backquotes do not.

Basically, if you just want to run a program, use `system()` – if you want to know what it returned, use backquotes. Both of them use the shell to process the command line arguments, pipes, redirection, and so on – with `system()`, you can turn this facility off by splitting the call up into a list, rather than giving one parameter.

There is a temptation among some programmers – particularly those coming from shell scripting – to shell out for nearly everything. On the other hand, there are those who only shell out when absolutely necessary. Use whichever method keeps your code both tidy and efficient.

```
system("clear"); # Easiest way to clear screen.

$status = system("sendmail -q"); # Flush mail queue
# To avoid shell processing, use
# $status = system("sendmail", "-q");
print STDERR "Something funny happened to sendmail: $?"
    unless $status==0;
```

```

$mail= `frm`;
if ($mail !~ /^You have no mail\.\/) {
    print "You have new mail:\n";
    print $mail;
    # $mail will contain newlines, so we don't need them.
}

```

A Full Example

Now that we've got everything we need to write real, working Perl programs, let's see how we'd implement the shell version of the CD database. This is pretty much a straightforward translation of the shell script, with only one or two Perl-like features. This also means we don't tidy up some of the less desirable features of that script – there's still the limitation that you can't have commas in track names, for instance. We'll see a full reimplementations of the database utility at the end of the chapter. In the meantime, you might want to compare this program with the one in Chapter 2 [ED: Please confirm chapter reference.] line-by-line.

One thing we do change, however, is, instead of manipulating the data inside files, we'll read the files into arrays at the beginning of the program, and write them out again at the end. This avoids all that troublesome messing around with temporary files (and gets over the Perl problem that we can't easily write to and read from a file at the same time).

1. First, as in the shell script, we tell the kernel this is a Perl script, and we follow with our copyright notice:

```

#!/usr/bin/perl -w

# Perl translation of chapter 2's shell CD database
# Copyright (C) 1999 Wrox Press.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

```

2. Now we set the global variables as before. Notice the last line: we're setting the signal handler for an interrupt (*Ctrl+C* on the keyboard) to be a reference to a subroutine; we create what's called an "anonymous subroutine reference" by putting what we want to achieve inside "sub { }" and taking the return value. This subroutine reference calls the subroutine `tidy_up` to dump out the track and title arrays to files, and then exits. The `tidy_up` subroutine writes out the files in a similar way to the way we read them in. The only troublesome bit is the newline processing; the files should have newlines to separate records, but the array elements need not have newlines in them. So, we chomp newlines on the way in, and add them again on the way out.

```

$menu_choice="";
$title_file="title.cdb";

```

```

$tracks_file="tracks.cdb";
$temp_file="/tmp/cdb.$$";
SIG{INT} = sub { tidy_up(); exit; } ;

sub read_in {
open TITLES, $title_file or die "Couldn't open $title_file : $!\n";
while (<TITLES>) { chomp; push @titles, $_ };
close TITLES;

open TRACKS, $tracks_file or die "Couldn't open $tracks_file : $!\n";
while (<TRACKS>) { chomp; push @tracks, $_ };
close TRACKS;
}

sub tidy_up {
# Die aborts with an error, and $! is the error message from open()
open TITLES, ">".$title_file or die "Couldn't write to $title_file : $!\n";
foreach (@titles) { print TITLES "$_\n"; }
close TITLES;

open TRACKS, ">".$tracks_file or die "Couldn't open $tracks_file : $!\n";
foreach (@tracks) { print TRACKS "$_\n"; }
close TRACKS;
}

```

3. Now, our two little functions for getting keyboard input:

```

sub get_return {
    print "Press return ";
    <> # Get a line from STDIN, and ignore it.
}

sub get_confirm {
    print "Are you sure? ";
    while (1) {
        $_ = <>; # Get a reply into $_
        return 1 if (/^y(?:es)?$/i); # 1 is true, not 0
        if (/^no?$/i) {
            print "Cancelled!\n";
            return 0;
        }
        print "Please enter yes or no.\n";
    }
}

```

4. Now, we display the main menu and get a choice from the user. The <<EOF syntax is called a *here-document* and prints until it finds the word EOF or whatever delimiter string you choose.

```

sub set_menu_choice {
    print `clear`; # Shelling out to clear screen. Yuck.

    print <<EOF;
a) Add new CD
f) Find CD
c) Count the CDs and tracks in the catalog
EOF

    if ($cdcatnum) {
        print "    l) List tracks on $cdtitle\n";
        print "    r) Remove $cdtitle\n";
        print "    u) Update track information for $cdtitle\n";
    }

    print "    q) Quit\n\n";
    print "Please enter choice then press return\n";
    chomp($menu_choice=<>);
    return
}

```

5. Then come the one-liners as before to add new records to the arrays, and the subroutine to add track information.

```
sub insert_title {
    push @titles, (join ", " , @_);
}

sub insert_track {
    push @tracks, (join ", " , @_);
}

sub add_record_tracks {
    print "Enter track information for this CD\n";
    print "When no more tracks enter q\n";
    $cdtrack=1;
    $cdtttitle="";
    while ($cdtttitle ne "q") {
        print "Track $cdtrack, track title? ";
        chomp($cdtttitle=<>);
        if ($cdtttitle =~ /,/ ) {
            print "Sorry, no commas allowed.\n";
            redo;
        }
        if ($cdtttitle and $cdtttitle ne "q") {
            insert_track($cdcatnum,$cdtrack,$cdtttitle);
            $cdtrack++;
        }
    }
}
```

6. Now, we implement the `add_records` to add the record of a new CD to the database.

```
sub add_records {
    print "Enter catalog name ";
    chomp($cdcatnum=<>);
    $cdcatnum =~ s/,.*//; # Drop everything after a comma.

    print "Enter title ";
    chomp($cdtttitle=<>);
    $cdtttitle =~ s/,.*//;

    print "Enter type ";
    chomp($cdtype=<>);
    $cdtype =~ s/,.*//;

    print "Enter artist/composer ";
    chomp($cdac=<>);
    $cdac =~ s/,.*//;

    print "About to add a new entry\n";
    print "$cdcatnum $cdtttitle $cdtype $cdac\n";

    if (get_confirm()) {
        insert_title($cdcatnum,$cdtttitle,$cdtype,$cdac);
        add_record_tracks();
    } else {
        remove_records();
    }
}
```

7. Since we've got an array of lines, finding the CD is very simple; we could iterate through the array and pick out the matches. However, it's easier to use Perl's `grep` function, which was intended for this very purpose.

```

sub find_cd {
    # $asklist is true if the first member of @_
    # (That is, the first parameter) is not "n"
    $asklist = ($_[0] ne "n");

    $cdcatnum="";
    print "Enter a string to search for in the CD titles ";
    chomp($searchstr=<>);
    return 0 unless $searchstr;

    # The \Q and \E metacharacters stop other metacharacters
    # from working, so question marks, asterisks and so on
    # in titles aren't dangerous.

    @matches = grep /\Q$searchstr\E/, @titles;
    if (scalar @matches == 0) {
        print "Sorry, nothing found.\n";
        get_return();
        return 0;
    } elsif (scalar @matches != 1 ) {
        print "Sorry, not unique.\n";
        print "Found the following:\n";
        foreach (@matches)
            print "$_\n";
    }
    get_return();
    return 0;
}
($cdcatnum,$cdtitle,$cdtype,$cdac) =
    split " , " , $matches[0];
unless ($cdcatnum) {
    print "Sorry, could not extract catalog field\n";
    get_return();
    return 0;
}
print "\nCatalog number: $cdcatnum\n";
print "Title: $cdtitle\n";
print "Type: $cdtype\n";
print "Artist/Composer: $cdac\n\n";
get_return();
if ($asklist) {
    print "View tracks for this CD? ";
    $_ = <>;
    if (/^y(?:es)?$/i) {
        print "\n";
        list_tracks();
        print "\n";
    }
}
return 1;
}

```

8. `update_cd` is nice and easy to implement, apart from the bit where we delete the old tracks from the array. We'll do this using another `grep`, but this time, we can negate the regular expression using `!/regex/`.

```

sub update_cd {
    unless ($cdcatnum) {
        print "You must select a CD first\n";
        find_cd("n");
    }
    if ($cdcatnum) {
        print "Current tracks are :-\n";
        list_tracks();
        print "\nThis will re-enter the tracks for $cdtitle\n";
        if (get_confirm()) {
            @tracks = grep !/^$cdcatnum/, @tracks;
            add_record_tracks();
        }
    }
}

```

```

    }
}

```

9. Since it's all stored in arrays, counting the contents of the database is trivial.

```

sub count_cds {
    print "Found " .(scalar @titles). " CDs, ";
    print "with a total of " .(scalar @tracks). " tracks.\n";
    get_return();
}

```

10. We've seen how to use `grep` with a negated regexp to remove entries from an array; let's do this again:

```

sub remove_records {
    unless ($cdcatnum) {
        print "You must select a CD first\n";
        find_cd("n");
    }

    if ($cdcatnum) {
        print "You are about to delete $cdtitle\n";
        if (get_confirm()) {
            @titles = grep !/^$cdcatnum/, @titles;
            @tracks = grep !/^$cdcatnum/, @tracks;
            @cdcatnum="";
            print "Entry removed";
        }
        get_return();
    }
}

```

11. `list_tracks` requires a pager, so we need to write out a temporary file and shell out.

```

sub list_tracks {
    unless ($cdcatnum) {
        print "No CD selected yet.\n";
        return
    }
    open(TEMP, ">$temp_file")
        or die "Can't write to $temp_file: $!\n";
    @temp = grep /^$cdcatnum/, @tracks;
    if (scalar @temp == 0) {
        print "No tracks found for $cdtitle\n";
    } else {
        print TEMP "\n$cdtitle :-\n\n";
        foreach (@temp) {
            s/^.*/; # Remove the first field
            print TEMP $_.\n";
        }
        close TEMP;
        system("more $temp_file");
        unlink($temp_file); # Delete it.
    }
    get_return();
}

```

12. Now, the main routine; we must remember to write out the arrays before exiting. We also make sure the files exist before reading from them, by creating them. Of course, we needn't

have done it this way – the alternative is not to complain if the files do not exist; the arrays would be empty, and the files would be created when we leave.

```
# File tests work like shell
system("touch $title_file") unless ( -f $title_file );
system("touch $tracks_file") unless ( -f $tracks_file );

read_in();

system("clear");
print "\n\nMini CD manager\n";
sleep(3);
while (1) {
    set_menu_choice();
    if ($menu_choice =~ /a/i) { add_records(); }
    elsif ($menu_choice =~ /r/i) { remove_records(); }
    elsif ($menu_choice =~ /f/i) { find_cd("y"); }
    elsif ($menu_choice =~ /u/i) { update_cd(); }
    elsif ($menu_choice =~ /c/i) { count_cds(); }
    elsif ($menu_choice =~ /l/i) { list_tracks(); }
    elsif ($menu_choice =~ /b/i) {
        print "\n";
        foreach (@titles) {
            print "$_\n";
        }
        print "\n";
        get_return();
    }
    elsif ($menu_choice =~ /q/i) { last; }
    else { print "Sorry, choice not recognized.\n"; }
}

tidy_up();
exit;
```

Perl on the Command Line

Now that we've seen a full-blown Perl program, what about the little everyday uses of Perl? Well, like `sed` and `awk`, it's perfectly possible to use Perl as a filter for housekeeping tasks. Indeed, Perl provides quite a useful set of command line options to help us do this. Like we did with `-w` right at the beginning of the chapter, we can also put these on the `#!` line in our scripts.

The first option is `-e`; like `sed` and `awk`, this allows us to run a one-line Perl script:

```
$ perl -e 'print "Hello, world\n";'
Hello, world
$
```

Similarly, we can provide Perl with a filename that it should take as standard input; we could rewrite our familiar comment-killer like this:

```
$ perl -e 'while(<>) { print unless /^#/ }' myfile
```

which would print out `myfile` without all the comment lines. However, looping through each line in a file is such a common operation that Perl provides a special syntax for it: `-n`

```
$ perl -n -e 'print unless /^#/' myfile
```

We could even do without the `print` – the `-p` flag acts as the following code:

```
while (<>) {
```

```
        ...
        print or die "-p destination: ${!\n}";
    }
```

This is very useful for search-and-replace regular expressions: a line like `perl -p -e 's/foo/bar/g' file` will print out the file with every occurrence of “foo” changed to “bar.” Now, let’s go one further; let’s say we do want to change, for instance, all references to “August” to “September” in a file. We’d normally have to collect the output into a temporary file, and then use `mv` to replace the old file with the temporary one. Not so in Perl. Perl supports the modification of files “in place,” using the `-i` option: `perl -p -i -e 's/August/September/g' myfile` is equivalent to this, in shell:

```
$ sed 's/August/September/g' myfile > tmpfile;
$ mv tmpfile myfile;
```

Want to take a backup of the original file? No problem! Just add a suffix to `-i`, like this: `perl -p -i.bak -e 's/August/September/g' myfile` changes the file, and saves a backup to `myfile.bak`. This is the way to build up very powerful filters and file editors with ease.

So, what other command-line magic can we achieve? We’ve seen `-w` to turn on additional warnings for your script. There’s also `-c` to check the syntax of your program without running it, and `-d`, the debugger—a very powerful tool for tracking down problems with your scripts. Finally, how about this: tired of adding “\n” to all your `print` statements, and chomping your input? Turn on automatic line processing, with `-l`. This will automatically chomp anything coming into the script via the readline operator, and also adds \n to anything going out via `print` statements. Combine with `-p -e` for hours of fun!

Modules

If you’re writing serious Perl programs, you will come to realize that a lot of the code you write has probably been written before; things like network programming, manipulating text or HTML, processing command line options, storing data in files, and so on. There may also be things you don’t feel you can do within Perl, and need a C extension for.

Perl modules are the answer in both these situations: they provide for reusability of code, like libraries in C, and also allow interfacing with other languages. We won’t go into a great deal of detail here about how to use modules or build your own modules, but try to give you a flavor of some of the things you can do with them.

CPAN

The main repository for Perl modules is called CPAN, the Comprehensive Perl Archive Network. As the name suggests, it’s a set of mirrored archives for pretty much every Perl module you could think of. The entry point for CPAN is at <http://www.cpan.org/> or at <http://www.perl.com/CPAN/>; these should direct you to your nearest mirror site. You can also download documentation, tutorials, and the latest sources of Perl from CPAN, but what you’ll want from it most often are the modules. Look in the `/modules/by-category` subdirectory of your local mirror for a sorted list of modules, or the file `CPAN.html` in the main directory for descriptions of all the CPAN modules.

Installing a Module

Having downloaded a module from CPAN, you can install it into your system as follows. We'll take the `Net::Telnet` module as an example. Assuming we've downloaded `Net-Telnet-3.01.tar.gz` from CPAN:

```
$ tar xzf Net-Telnet-3.01.tar.gz
$ cd Net-Telnet-3.01
$ perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Net::Telnet
$ make install
```

The last step may need certain privileges to write to the Perl directories.

There is another way to install modules. The CPAN module by Andreas König, which comes with your Perl distribution, will guide you through installing them. Simply type:

```
$ perl -MCPAN -e shell
```

and follow the instructions. This can also be used for installing “bundles” of modules, like `libwww`, which have many dependencies.

Documentation (*perldoc*)

All modules (including modules bundled with Perl, and the Perl language itself) should come with full documentation. This documentation is written in the POD (Plain Old Documentation) syntax, and can be read with the command `perldoc`. To read the documentation for the `Net::Telnet` module above, just type:

```
$ perldoc Net::Telnet
```

To find out more about the Perl language, start from `perldoc perl` and, if you're patient enough, work through all the pages that it points you to. `perldoc` also provides two very useful options: `-q keyword` will search through the extensive Perl FAQ for the specified keyword. For instance, `perldoc -q Y2K` will tell you about “Year 2000 bug” compliance, and so on. `-f function` brings up the section of the `perlfunc` page relating to the function in question. Try, for instance, `perldoc -f unshift`.

Networking

Now let's see how these modules can actually help us. You'll need to download the relevant set of modules from CPAN.

LWP

The LWP (`libwww-perl`) is a suite of modules that cover web server and client operations. Let's assume we've got the whole bundle installed. We can now use the module `LWP::Simple` for simple operations. Let's get the HTML version of the day's news:

```
use LWP::Simple;
$news = get "http://news.bbc.co.uk/text_only.htm";
```

Now, we can extract all the links from it, using the `HTML::LinkExtor` module found in the `HTML-Parser` library:

```
use HTML::LinkExtor;
$p = HTML::LinkExtor->new();
$p->parse($news);
@links = $p->links; # Array of all the links in the file
```

OK, maybe this isn't that clear to you, because we haven't introduced object-orientated programming – however, just like C libraries, Perl modules can seriously simplify programming.

IO::Socket

Now, how about socket networking? Remember the C program that connects to a time server to get the time? Here's how we'd do it in Perl. The socket library, `IO::Socket`, is a standard module that should have come with your Perl distribution.

```
use IO::Socket;
$host = "localhost" unless ($host = shift);
$socket = IO::Socket::INET->new(
    PeerAddr => $host,
    PeerPort => "daytime")
or die "Couldn't connect to $host: $@";
$time = <$socket>; # Sockets act like filehandles.
print $time;
```

Of course, we could do it all with Perl's built-in socket functions (`socket`, `connect`, `gethostbyname`, and so on), but this way is a lot neater. Let the modules do the work.

Net Modules

If you're using Perl to automate system tasks, you might find the `Net::` series of modules useful. The `Net::Telnet` module we installed earlier provides access to a telnet session, including automating connecting, logging in, and executing commands; similarly, `Net::FTP` (from the libnet bundle) allows for FTP sessions to be done automatically. Here's how we'd get the MD5 module from CPAN:

```
use Net::FTP;
$ftp = Net::FTP->new("ftp.cpan.org") or die "Couldn't connect: $@\n";
$ftp->login("anonymous");
$ftp->cwd("/pub/modules/by-name/MD5/");
$ftp->get("MD5-1.7.tar.gz");
$ftp->quit();
```

There's also `Net::NNTP` (libnet) for news reading and posting, `Net::DNS` (Net-DNS) for DNS querying, `Net::POP3` (libnet) for fetching mail, `Net::Ping`, `Net::Whois` (Net-Whois), `Net::IRC` (Net-IRC), and so on.

Databases

There are a number of ways we can store and retrieve data in Perl; we've seen how to deal with flat-file databases. A more common method is the DBM system we saw in Chapter 7. [ED: Please confirm chapter reference.] We can access DBMs by *tying* them to hashes. This means that the data in the hash will be linked to the data on the disk. We use the standard `AnyDBM_File` module as our interface to the GDBM libraries. (Note that `AnyDBM_File` can, in fact, access a number of DBM packages.)

```
use AnyDBM_File;
tie %database, "AnyDBM_File", "data.db";
# We can now use %database as a normal hash; any adding, deleting or
# modifying of keys will be reflected in the data file.

untie %database;
```

There are a number of similar modules to tie data structures to files. For instance, `DB_File` ties an array to a text file using the Berkeley DB library – in our CD database example, we could have used this instead of

reading in and writing out the data files at the beginning and end of the program. MLDBM (Multi-level DBM) is a module used for storing complex data structures in a DBM, and we'll use this to implement our final version of the CD database.

Finally, using the DBI interface, one can store data in relational databases like MySQL, PostgreSQL, Oracle, and Informix, and execute SQL queries and statements. The relevant modules are DBI, for the interface, and DBD::[Oracle,mysql...] for the drivers for the individual databases.

The CD Database Revisited

Now that we've seen what Perl can do, let's have a look at the CD database program as it would be written by a Perl native. There will be a few concepts in here (references and nested data structures, for instance) that we haven't covered, but that's OK; don't worry too much if you don't understand everything here. You're not meant to – the idea is just to give you an impression of a “real” Perl program.

1. First, we start with the comments as before:

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

# Perl translation of chapter 2's shell CD database
# Copyright (C) 1999 Wrox Press.

# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-#1307 USA
```

2. We'll place the whole of the program here, and flesh out the functions later:

```
use MLDBM qw(AnyDBM_File);
my $record;
tie(%tmp, "MLDBM", "cddb.db")
or die "Couldn't tie DB.\n"; # Scary complex hash contains the whole DB.
%database = %tmp; # Overcome a limitation in MLDBM. *sigh*

# Tidy up nicely
$SIG{INT} = sub { %tmp = %database; untie %tmp } ;
system("clear");
print "\n\nCD Database Manager\n\n";
while (1) {
    my $menu_choice = main_menu($record);
    if ($menu_choice eq "a") { $record = add_cd(); }
    elsif ($menu_choice eq "r") { remove_cd($record); undef $record; }
    elsif ($menu_choice eq "f") { $record = find_cd("y"); }
    elsif ($menu_choice eq "u") { update_cd($record); }
    elsif ($menu_choice eq "c") { count_cds(); }
    elsif ($menu_choice eq "l") { list_tracks($record); }
    elsif ($menu_choice =~ /q/i) { last; }
    else {
        print "Can't get here.\n";
    }
}
```

```
%tmp=%database;
untie %tmp;
```

3. Now, we display the main menu and validate the choice:

```
sub main_menu {
    my $record = shift;
    my $choice;
    my $title = $database{$record}->{title} if $record;
    print <<EOF;

    Options :

        a) Add new CD
        f) Find CD
        c) Count CDs and tracks in the catalogue
    EOF
    if ($record) {
        print "    l) List tracks on $title\n";
        print "    r) Remove $title\n";
        print "    u) Update entry for $title\n";
    }
    print "    q) Quit\n";
    print "Your choice: ";
    while (1) {
        $choice=lc(<>);
        substr($choice,1)="";
        # Now, we see if the choice is contained in the string of
        # acceptable options, (Which includes l, r and u if we've
        # selected a record.) by using it as a regexp. Looks weird?
        return $choice if ("afcqr"{$record?"lru":""} =~ /$choice/);

        # If not, that's invalid
        print "Invalid choice.\nTry again: ";
    }
}
}
```

4. Let's tackle adding records to the database next. The database is actually quite a complicated hash; the keys are the catalog numbers, and the values are hashes themselves. These hashes have keys "title", "type", "artist", and "tracks". That's why we used the funky-looking `$database{$record}->{title}` above - `$database{$record}` is a hash. (It's actually a *reference* to a hash; C programmers can think of them as pointers. For more about reference, look at the [perlref](#) documentation.) The `->{title}` syntax looks inside the hash reference and gets the value of the "title" key. The value of "tracks" is, of course, a reference to an array of tracks. Arrays inside hashes inside hashes; it takes a little getting used to.

```
sub add_cd {
    while(1) {
        print "Enter catalog number: ";
        chomp($record=<>);
        if (exists $database{$record}) {
            print "Already exists. ";
            print "Please enter a different number.\n";
        } else {
            last;
        }
    }

    print "Enter title: ";
    chomp($title=<>);

    print "Enter type: ";
    chomp($type=<>);
    print "Enter artist/composer: ";
}
```

```

    chomp($artist=<>);

    $database{$record}= {
        "title" => $title,
        "type" => $type,
        "artist" => $artist
    };

    add_tracks($record);
    return $record; # Tell the main menu the new record number.
}

```

5. Now, the subroutine to add the tracks; this is where we bring out the array reference.

```

sub add_tracks {
    my $record = shift;
    print "Enter track information for this CD\n";
    print "Enter a blank line to finish.\n\n";
    my $counter=0; my @tracks;
    while (1) {
        print ++$counter.".": ";
        chomp($track=<>);
        if ($track) {
            # @{...} means "interpret as an array"
            push @{$database{$record}->{tracks}}, $track;
        } else {
            last;
        }
    }
}

```

6. The code to find a CD is a bit complicated, since we need to look through all the values of `$database{$record}->{title}` for each value of `$record` in the hash. `grep` to the rescue again.

```

sub find_cd {
    $view = ($_[0] eq "y");

    print "Enter a string to search for: ";
    chomp($search=<>);

    # For each key, (record) add the key to the @found array if the
    # title field of that record contains the search string.
    @matches = grep {$database{$_}->{title} =~ /\Q$search\E/ }
                keys %database;
    if (scalar @matches == 0) {
        print "Sorry, nothing found.\n";
        return;
    } elsif (scalar @matches != 1) {
        print "Sorry, not unique.\n";
        print "Found the following:\n";
        foreach (@matches) {
            print $database{$_}->{title}."\n";
        }
        return;
    }
    $record=$matches[0];
    print "\n\nCatalog number: ".$record."\n";
    print "Title: ".$database{$record}->{title}."\n";
    print "Type: ".$database{$record}->{type}."\n";
    print "Artist/Composer: ".$database{$record}->{artist}."\n\n";

    if ($view) {
        print "Do you want to view tracks? ";
        $_ = <>;
        if (/^y(?:es)?$/i) {
            print "\n";
            list_tracks($record);
            print "\n";
        }
    }
}

```

```

    }
    return $record;
}

```

7. Once we've got this far, listing the tracks isn't difficult!

```

sub list_tracks {
    my $record = shift;
    foreach (@{$database{$record}->{tracks}}) {
        print $_."\n";
    }
}

```

8. Updating a CD just means removing the old tracks and adding a new set:

```

sub update_cd {
    my $record = shift;
    print "Current tracks are: \n";
    list_tracks($record);
    print "\nDo you want to reenter them?\n";
    if (($_ = <>) =~ /^y(?:es)?$/i) {
        # Remove the old entry from the hash
        delete $database{$record}->{tracks};
        add_tracks($record);
    } else {
        print "OK, canceling.\n"
    }
}

```

9. Similarly, removing a CD just means deleting its hash entry:

```

sub remove_cd {
    my $record = shift;
    print "\nDo you want to delete this CD?\n";
    if (($_ = <>) =~ /^y(?:es)?$/i) {
        delete $database{$record};
    } else {
        print "OK, cancelling.\n"
    }
}

```

10. Finally, counting the CDs is easy—it's just the number of keys in the hash. Counting the tracks, however, is a little more tricky; we evaluate the tracks array in scalar context for each of the keys in the database, and add the values together. (You could do this with `map()`, but that would be less clear.)

```

sub count_cds {
    my $totaltracks=0;
    print "Found ".(scalar keys %database)." CDs and ";
    foreach (keys %database) {
        $totaltracks+= scalar @{$database{$_}->{tracks}};
    }
    print $totaltracks." tracks.\n";
}

```

Summary

In this chapter, we've seen how to use some features of the Perl programming language, examined some of the modules available for it, and seen how our CD database application could be implemented in Perl.