6

Files and Data

We're starting to write real programs now, and real programs need to be able to read and write files to and from your hard drive. At the moment, all we can do is ask the user for input using `<STDIN>` and print data on the screen using `print`. Pretty simple stuff, yes, but these two ideas actually form the basis of a great deal of the file handling you'll be doing in Perl.

What we want to do in this chapter is extend these techniques into reading from and writing to files, and we'll also look at the other techniques we have for handling files, directories, and data.

Filehandles

First though, let's do some groundwork. When we're dealing with files, we need something that tells Perl which file we're talking about, which allows us to refer to and access a certain file on the disk. We need a label, something that will give us a 'handle' on the file we want to work on. For this reason, the 'something' we want is known as a **filehandle**.

We've actually already seen a filehandle: the **`STDIN`** of `<STDIN>`. This is a filehandle for the special file 'standard input', and whenever we've used the idiom `<STDIN>` to read a line, we've been reading from the standard input file. Standard input is the input provided by a user either directly as we've seen, by typing on the keyboard, or indirectly, through the use of a 'pipe' that (as we'll see) pumps input into the program.

As a counterpart to standard input, there's also standard output: **`STDOUT`**. Conversely, it's the output we provide to a user, which at the moment we're doing by writing to the screen. Every time we've used the function print so far, we've been implicitly using `STDOUT`:

```
print STDOUT "Hello, world.\n";
```

is just the same as our original example in Chapter 1. There's one more 'standard' filehandle: standard error, or **`STDERR`**, which is where we write the error messages when we `die`.

Every program has these three filehandles available, at least at the beginning of the program. To read and write from other files, though, you'll want to open a filehandle of your own. Filehandles are usually one-way: You can't write to the user's keyboard, for instance, or read from his or her screen. Instead, filehandles are open either for reading or for writing, for input or for output. So, here's how you'd open a filehandle for reading:

```
open FH, $filename or die $!;
```

The operator for opening a filehandle is `open`, and it takes two arguments, the first being the name of the filehandle we want to open. Filehandles are slightly different from ordinary variables, and they do not need to be declared with `my`, even if you're using `strict` as you should. It's traditional to use all-capitals for a filehandle to distinguish them from keywords.

The second argument is the file's name – either as a variable, as shown above, or as a string literal, like this:

```
open FH, 'output.log' or die $!;
```

You may specify a full path to a file, but don't forget that if you're on Windows, a backslash in a double-quoted string introduces an escape character. So, for instance, you should say this:

```
open FH, 'c:/test/news.txt' or die $!;
```

rather than:

```
open FH, "c:\test\news.txt" or die $!;
```

as `\t` in a double-quoted string is a tab, and `\n` is a new line. You could also say `"c:\\test\\news.txt"` but that's a little unwieldy. My advice is to make use of the fact that Windows allows forward slashes internally, and forward slashes do not need to be escaped: `"c:/test/news.txt"` should work perfectly fine.

So now we have our filehandle open – or have we? As I mentioned in Chapter 4, the `X or Y` style of conditional is often used for ensuring that operations were successful. Here is the first real example of this.

When you're dealing with something like the file system, it's dangerous to blindly assume that everything you are going to do will succeed. A file may not be present when you expect it to be, a file name you are given may turn out to be a directory, something else may be using the file at the time, and so on. For this reason, you need to check that the `open` did actually succeed. If it didn't, we `die`, and our message is whatever is held in `$!`.

What's `$!`? This is one of Perl's **special variables**, designed to give you a way of getting at various things that Perl wants to tell you. In this case, Perl is passing on an error message from the system, and this error message should tell you why the `open` failed: It's usually something like 'No such file or directory' or 'permission denied'.

There are special variables to tell you what version of Perl you are running, what user you are logged in as on a multi-user system, and so on. Appendix B contains a complete description of Perl's special variables.

So, for instance, if we try and open a file that is actually a directory, this happens:

```
#!/usr/bin/perl
# badopen.plx
use warnings;
use strict;
open BAD, "/temp" or die "We have a problem: $!";
```

```
>perl badopen.plx
Name "main::BAD" used only once: possible typo at badopen.plx line 5
We have a problem: Permission denied at badopen.plx line 5.
>
```

The first line we see is a warning. If we were to finish the program, adding further operations on `BAD` *(or get rid of* `use warnings`*), it wouldn't show up.*

You should also note that if the argument you give to `die` does not end with a new line, Perl automatically adds the name of the program and the location that had the problem. If you want to avoid this, always remember to put new lines on the end of everything you `die` with.

Reading Lines

Now that we can open a file, we can then move on to reading the file one line at a time. We do this by replacing the `STDIN` filehandle in `<STDIN>` with our new filehandle, to get `<FH>`. Just as `<STDIN>` reads a single line from the keyboard, `<FH>` reads one line from a filehandle. This <…> construction is called the **diamond operator**, or **readline operator**:

Try It Out : Numbering Lines

We'll use the `<FH>` construct in conjunction with a `while` loop to go through each line in a file. So then, to print a file with line numbers added, you can say something like this:

```
#!/usr/bin/perl
# nl.plx
use warnings;
use strict;

open FILE, "nlexample.txt" or die $!;
my $lineno = 1;

while (<FILE>) {
   print $lineno++;
   print ": $_";
}
```

Now, create the file `nlexample.txt` with the following contents:

```
One day you're going to have to face
   A deep dark truthful mirror,
And it's gonna tell you things that I still
   Love you too much to say.
####### Elvis Costello, Spike, 1988 #######
```

This is what you should see when you run the program:

```
> perl nl.plx
1: One day you're going to have to face
2:      A deep dark truthful mirror,
3: And it's gonna tell you things that I still
4:      Love you too much to say.
5. ####### Elvis Costello, Spike, 1988 #######
>
```

How It Works

We begin by opening our file and making sure it was opened correctly:

```
open FILE, "nlexample.txt" or die $!;
```

Since we're expecting our line numbers to start at one, we'll initialize our counter:

```
my $lineno = 1;
```

Now we read each line from the file in turn, which we do with a little magic:

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

This syntax is actually equivalent to:

```
while (defined ($_ = <FILE>)) {
```

That is, we read a line from a file and assign it to `$_`, and we see whether it is defined. If it is, we do whatever's in the loop. If not, we are probably at the end of the file so we need to come out of the loop. This gives us a nice, easy way of setting `$_` to each line in turn.

As we have a new line, we print out its line number and advance the counter:

```
    print $lineno++;
```

Finally, we print out the line in question:

```
    print ": $_";
```

There's no need to add a newline since we didn't bother `chomp`ing the incoming line. Of course, using a statement modifier, we can make this even more concise:

```
open FILE, "nlexample.txt" or die $!;
my $lineno = 1;

print $lineno++, ": $_" while <FILE>
```

But since we're going to want to expand the capabilities of our program -adding more operations to the body of the loop – we're probably better off with the original format.

Creating Filters

As well as the three standard filehandles, Perl provides a special filehandle called **ARGV**. This reads the names of files from the command line and opens them all, or if there is nothing specified on the command line, it reads from standard input. Actually, the @ARGV array holds any text after the program's name on the command line, and <ARGV> takes each file in turn. This is often used to create filters, which read in data from one or more files, process it, and produce output on STDOUT.

Because it is used so commonly, Perl provides an abbreviation for <ARGV>: an empty diamond, or <>. We can make our line counter a little more flexible by using this filehandle:

```
#!/usr/bin/perl
# nl2.plx
use warnings;
use strict;

my $lineno = 1;

while (<>) {
   print $lineno++;
   print ": $_";
}
```

Now Perl expects us to give the name of the file on the command line:

```
> perl nl2.plx nlexample.txt
1: One day you're going to have to face
2:     A deep dark truthful mirror,
3: And it's gonna tell you things that I still
4:     Love you too much to say.
5. ####### Elvis Costello, Spike, 1988 #######
>
```

We can actually place a fair number of files on the command line, and they'll all be processed together. For example:

```
> perl nl2.plx nlexample.txt nl2.plx
1: One day you're going to have to face
2:     A deep dark truthful mirror,
3: And it's gonna tell you things that I still
4:     Love you too much to say.
5. ####### Elvis Costello, Spike, 1988 #######
6: #!/usr/bin/perl
7: # nl2.plx
8: use warnings;
9: use strict;
10:
11: my $lineno = 1;
12:
13: while (<>) {
14:     print $lineno++;
15:     print ": $_";
16: }
```

If we need to find out the name of the file we're currently reading, it's stored in the special variable $ARGV. We can use this to reset the counter when the file changes.

Try it out : Numbering Lines in Multiple Files

By detecting when `$ARGV` changes, we can reset the counter and display the name of the new file:

```
#!/usr/bin/perl
# nl3.plx
use warnings;
use strict;

my $lineno;
my $current = "";

while (<>) {
   if ($current ne $ARGV) {
      $current = $ARGV;
      print "\n\t\tFile: $ARGV\n\n";
      $lineno=1;
   }

   print $lineno++;
   print ": $_";
}
```

And now we can run this on our example file and itself:

```
> perl nl3.plx nlexample.txt nl3.plx

                File: nlexample.txt

1: One day you're going to have to face
2:      A deep dark truthful mirror,
3: And it's gonna tell you things that I still
4:      Love you too much to say.
5. ######## Elvis Costello, Spike, 1988 ########

                File: nl3.plx

1: #!/usr/bin/perl
2: # nl3.plx
3: use warnings;
4: use strict;
5:
6: my $lineno;
7: my $current = "";
8:
9: while (<>) {
10:      if ($current ne $ARGV) {
11:           $current = $ARGV;
12:           print "\n\t\tFile: $ARGV\n\n";
13:           $lineno=1;
14:      }
15:
16:      print $lineno++;
17:      print ": $_";
18: }
>
```

How It Works

This is a technique you'll often see in programming to detect when a variable has changed. `$current` is meant to contain the current value of `$ARGV`. But if it doesn't, `$ARGV` has changed:

```
if ($current ne $ARGV) {
```

so we set `$current` to what it should be – the new value – so we can catch it again next time:

```
   $current = $ARGV;
```

We then print out the name of the new file, offset by new lines and tabs:

```
   print "\n\t\tFile: $ARGV\n\n";
```

and reset the counter so we start counting the new file from line one again.

```
   $lineno=1;
}
```

As with most tricks like these, it's actually really simple to code it once you've seen how it's coded. The catch is having to solve problems like these for the first time by yourself.

Reading More than One Line

Sometimes we'll want to read more than one line at once. When you use the diamond operator in a scalar context, as we've been doing so far, it'll provide you with the next line. However, in a list context, it will return all of the remaining lines. For instance, you can read in an entire file like this:

```
open INPUT, "somefile.dat" or die $!;
my @data;
@data = <INPUT>;
```

This is, however, quite memory-intensive. Perl has to store every single line of the file into the array, whereas you may only want to be dealing with one or two of them. Usually, you'll want to step over a file with a `while` loop as before. However, for some things, an array is the easiest way of doing things. For example, how do you print the last five lines in a file?

The problem with reading a line at a time is that you don't know how much text left you've got to read. You can only tell when you run out of data, so you'd have to keep an array of the last five lines read and drop an old line when a new one comes in. You'd do it something like this:

```
#!/usr/bin/perl
# tail.plx
use warnings;
use strict;

open FILE, "gettysburg.txt" or die $!;
my @last5;

while (<FILE>) {
   push @last5, $_; # Add to the end
   shift @last5 if @last5 > 5; # Take from the beginning
}

print "Last five lines:\n", @last5;
```

And that's exactly how you'd do it if you were concerned about memory use on big files. Given a suitably primed `gettysburg.txt`, this is what you'd get:

```
>perl tail.plx
Last five lines:
- that from these honored dead we take increased devotion to that cause for
which they gave the last full measure of devotion - that we here highly resolve
that these dead shall not have died in vain, that this nation under God shall
have a new birth of freedom, and that government of the people, by the people,
for the people shall not perish from the earth.
>
```

However, if memory wasn't a problem, or you knew you were going to be primarily dealing with small files, this would be perfectly sufficient:

```
#!/usr/bin/perl
# tail2.plx
use warnings;
use strict;

open FILE, "gettysburg.txt" or die $!;
my @speech = <FILE>;

print "Last five lines:\n", @speech[-5 ... -1];
```

What's My Line (Separator)?

So far we've been reading in single lines – a series of characters ending in a new line. One of the other things we can do is to alter Perl's definition of what separates a line.

The special variable `$/` is called the 'input record separator'. Usually, it's set to be the newline character, `\n`, and each 'record' is a line. We might say more correctly that `<FILE>` reads a single **record** from the file. Furthermore, `chomp` doesn't just remove a trailing new line – it removes a trailing record separator. However, we can set this separator to whatever we want, and this will change the way Perl sees lines. So if, for instance, our data was defined in terms of paragraphs, rather than lines, we could read one paragraph at a time by changing `$/`.

Try It Out : Fortune Cookie Dispenser

The fortune cookies file for the UNIX `fortune` program – as well as some 'tagline' generators for e-mail and news articles – consist of paragraphs separated by a percent sign on a line of its own, like this:

```
We all agree on the necessity of compromise.  We just can't agree on
when it's necessary to compromise.
    -- Larry Wall
%
All language designers are arrogant.  Goes with the territory...
    -- Larry Wall
%
Oh, get a hold of yourself. Nobody's proposing that we parse English.
    -- Larry Wall
%
Now I'm being shot at from both sides. That means I *must* be right.
     -- Larry Wall
%
```

Save this as `quotes.dat` and then write a program to pick a random quote from the file:

```
#!/usr/bin/perl
# fortune.plx
use warnings;
use strict;

$/ = "\n%\n";

open QUOTES, "quotes.dat" or die $!;
my @file = <QUOTES>;

my $random = rand(@file);
my $fortune = $file[$random];
chomp $fortune;

print $fortune, "\n";
```

This is what you get (or might get – it is random, after all):

```
> perl fortune.plx
Now I'm being shot at from both sides. That means I *must* be right.
    -- Larry Wall
>
```

How It Works

Once we've set our record separator appropriately, most of the work is already done for us. This is how we change it:

```
$/ = "\n%\n";
```

Now a 'line' is everything up to a newline character and then a percent sign on its own and then another new line, and when we read the file into an array, it ends up looking something like this:

```
my @file = (
   "We all agree on the necessity of compromise.  We just can't agree on
when it's necessary to compromise.\n    -- Larry Wall\n%\n",
   "All language designers are arrogant.  Goes with the territory...\n    -- Larry
Wall\n%\n",
   ...
);
```

We want a random line from the file. The operator for this is `rand`:

```
my $random = rand(@file);
my $fortune = $file[$random];
```

`rand` produces a random number between zero and the number given as an argument. What's the argument we give it? As you know, an array in a scalar context gives you the number of elements in the array. `rand` actually generates a fractional number, but when we look it up in the array, as we've seen in Chapter 3, Perl ignores the fractional part. Actually, it's more likely that in existing code you'll see those two statements combined into one, like this:

```
my $fortune = $file[rand @file];
```

Now we have our fortune, but it still has the record separator on the end, so we need to `chomp` to remove it:

```
chomp $fortune ;
```

Finally, we can print it back out, remembering that we need to put a new line on the end:

```
print $fortune, "\n";
```

Reading Paragraphs at a Time

If you set the input record separator, `$/`, to the empty string, "", Perl reads in a paragraph at a time. Paragraphs must be separated by a completely blank line, with no spaces on it at all. Of course, you can use `split` or similar to extract individual lines from each paragraph. This program creates a 'paragraph summary' by printing out the first line of each paragraph in a file:

Try It Out : Paragraph Summariser

We'll use `split` to get at the first line in each paragraph, and we'll number the paragraphs:

```
#!/usr/bin/perl
# summary.plx
use warnings;
use strict;

$/ = "";
my $counter = 1;

while (<>) {
   print $counter++, ":";
   print ((split /\n/, $_)[0]);
   print "\n";
}
```

When run on the beginning of this chapter, it gives the following output:

```
> perl summary.plx chapter6
1:We're starting to write real programs now, and real programs
2:What we want to do in this chapter is extend these techniques
3:First though, let's do some groundwork. When we're dealing
4:We've actually already seen a filehandle: the STDIN of <STDIN>.
5:As a counterpart to standard input, there's also standard
6:Every program has these three filehandles available, at least
>
```

We're assuming here that each line in the paragraph ends with a newline character rather than wrapping around to the next line. In the latter case, our program would return each of the paragraphs in their entirety, because split is being based on `\n`.

How It Works

This time we're reading from files specified on the command line, so we use the diamond operator. We start by putting the input record separator into paragraph mode:

```
$/ = "";
```

For every paragraph we read in, we print a new number, then get the first line of the paragraph:

```
print ((split /\n/, $_)[0]);
```

First we split the paragraph into lines, by `split`ting around a newline character. Since `split` just produces a list, we can take the first element of this list in the same way as any other.

Reading Entire Files

Finally, you may want to read a whole file into a single string. You could do this easily enough using `join`, but Perl provides another special value of `$/` for this. If we want to say that there is no record separator, we set `$/` to the undefined value. So, for instance, to read the whole of the above quotes file into a variable, we do this:

```
$/ = undef;
open QUOTES, "quotes.dat" or die $!;
my $file = <QUOTES>;
```

You may also see the form `undef $/` doing the same job: the `undef` operator gives a variable the undefined value.

Writing to Files

We've been using the `print` operator to print a list to standard output. We'll also use a different form of the `print` operator to print to a file. However, as we mentioned above, files are usually open either for reading *or* for writing – not both. We've been opening files and reading from them, but how do we open them for writing?

Opening a File for Writing

We actually use a syntax that's used by the shell for writing to files. In Windows and UNIX, if we want to put standard output into a file, we add the operator **>** and the file name to the end of the command. For example, saying something like this:

> **perl summary.plx chapter6 > summary6**

will create a file called `summary6`, which contains the following text:

```
1:We're starting to write real programs now, and real programs
2:What we want to do in this chapter is extend these techniques
3:First though, let's do some groundwork. When we're dealing
4:We've actually already seen a filehandle: the STDIN of <STDIN>.
5:As a counterpart to standard input, there's also standard
6:Every program has these three filehandles available, at least
```

Now, to open a file for writing, we do this:

```
open FH, "> $filename" or die $!;
```

This will either create a new file or completely wipe out the contents of an already existing file and let us start writing from the beginning. Don't use this on files you want to keep! If we want to add things to the end of an existing file, use two arrows, like this:

```
open FH, ">> $filename" or die $!;
```

There's no easy way of adding or changing text at the beginning or middle of a file. The typical way to do this is to read in the original and write the changed data to another file. We'll see shortly how this is done.

Similarly, you can redirect data to a program's standard *input* by using the left arrow, like this:

```
>perl summary.plx < chapter6.txt
```

As you've probably guessed, this means you can open files for input by saying:

```
open FH, "< $filename";
```

This is exactly the same as `open FH, $filename`; as we've used previously; it's just a little more explicit.

Writing on a Filehandle

We're now ready to write the file, which we'll do by using a special form of the `print` operator. Normally, to print things out from the screen, we say this:

```
print list;
```

When we want to write to a file, we'll use this instead:

```
print FH list;
```

So, for instance, here's one way of copying a file:

Try It Out : File Copying

We'll read in a file one line at a time, writing each line onto the new file:

```
#!/usr/bin/perl
# copy.plx
use warnings;
use strict;

my $source = shift @ARGV;
my $destination = shift @ARGV;
```

```
open IN, $source or die "Can't read source file $source: $!\n";
open OUT, ">$destination" or die "Can't write on file $destination: $!\n";

print "Copying $source to $destination\n";

while (<IN>) {
   print OUT $_;
}
```

Now there isn't much to see in this program, but let's run it anyway:

```
> perl copy.plx gettysburg.txt speech.txt
Copying gettysburg.txt to speech.txt
>
```

How It Works

We get the name of the file to copy from the command line:

```
my $source = shift @ARGV;
my $destination = shift @ARGV;
```

The command line arguments to our program are in the `@ARGV` array, as we saw in Chapter 4, and we use `shift` (which pops the top element of an array into a variable) to get an element out. We could quite easily have said this:

```
my $source = $ARGV[0];
my $destination = $ARGV[1];
```

However, `shift` is slightly more common. Next, open our two files:

```
open IN, $source or die "Can't read source file $source: $!\n";
open OUT, "> $destination" or die "Can't write on file $destination: $!\n";
```

The first of those lines should be familiar. The second, meanwhile, adds the arrow to show we want to write on that file. It's a double-quoted string so, as always, the destination file name is interpolated. Notice that we're taking care to check if the files can be opened for reading and writing; it is essential to let the user know if, for example, they do not have permission to access a certain file, or the file does not exist. There's never really good reason not to do this.

The copying procedure is simple enough: read a line from the source file, and then write it to the destination:

```
while (<IN>) {
   print OUT $_;
}
```

`<IN>` returns a list of as many lines as it can in list context. So the `while` loop steps through this list, copies each line to memory and `print`ing to the destination file `OUT`, one at a time for each cycle. So why don't we just say:

```
print OUT <IN>;
```

The trouble is, that's not very memory conscious. Perl would have to read in the *whole* file at once in order to construct the list and only then pass it out to `print`. For small files, this is fine. On the other hand, if we thought we could get away with reading the whole file into memory at one go, we also could do it this way:

```
$/ = undef;
print OUT <IN>;
```

This will read the whole file as a single entry, which is faster for sure, since Perl won't have to think about separating each line and building up a list, but still only suited to small files. Since we want to allow for large files, too, we'll stick with our original technique.

Let's see another example. This time, instead of writing the file straight out, we'll sort the lines in it first. In this case, we can't avoid reading in every line into memory. We need to have all the lines in an array or something similar. Let's see how we'd go about doing this.

Try It Out : File Sorter

If you've ever needed to sort the lines in a file, this is for you. The program works in three stages:

- First, open the files that the user specifies.
- Next, read in the file and sort it.
- Finally, write the sorted lines out.

Here's the full listing:

```
#!/usr/bin/perl
# sort.plx
use warnings;
use strict;

my $input = shift;
my $output = shift;
open INPUT, $input or die "Couldn't open file $input: $!\n";
open OUTPUT, ">$output" or die "Couldn't open file $input: $!\n";

my @file = <INPUT>;
@file = sort @file;

print OUTPUT @file;
```

Now if we have the following file, `sortme.txt`:

```
Well, I finally found someone to turn me upside-down
And nail my feet up where my head should be
If they had a king of fools then I could wear that crown
And you can all die laughing, because I'd wear it proudly
```

We can run our program like this:

```
>perl sort.plx sortme.txt sorted.txt
>
```

And we'll end up with a file, `sorted.txt`:

```
And nail my feet up where my head should be
And you can all die laughing, because I'd wear it proudly
If they had a king of fools then I could wear that crown
Well, I finally found someone to turn me upside-down
```

How It Works

The first stage, that of opening the files, is very similar to what we did before, with one small change:

```
my $input = shift;
my $output = shift;
open INPUT, $input or die "Couldn't open file $input: $!\n";
open OUTPUT, ">$output" or die "Couldn't open file $input: $!\n";
```

We don't tell Perl which array to `shift`, so it assumes we want `@ARGV`, which is just as well, because in this case, we do!

Getting the file sorted is a simple matter of reading it into an array and calling `sort` on the array:

```
my @file = <INPUT>;
@file = sort @file;
```

In fact, we could just say `my @file = sort <INPUT>;` and that would be slightly more efficient. Perl would only have to throw the list around once.

Finally, we write the sorted array out:

```
print OUTPUT @file;
```

We could even do all this in one line, without using an array:

```
print OUTPUT sort <INPUT>;
```

This is arguably the most efficient solution, and you might think it's relatively easy to understand. What are we doing after all? We're printing the sorted input file on the output file. But it's the least extensible way of writing it. We can't change any of the stages when it's written like that.

What could we change? Well, remember that there are at least two ways to sort things: `sort` usually does an ASCII-order sort, but this doesn't help us when we're sorting columns of numbers. To do that, we need to use the numeric comparison operator, `<=>`, when we're sorting. As we saw in Chapter 3, the syntax would be something like this:

```
@sorted = sort { $a <=> $b } @unsorted;
```

Let's now extend our sort program to optionally sort numerically. Add the following lines:

```
#!/usr/bin/perl
# sort2.plx
use warnings;
use strict;

my $numeric = 0;
my $input = shift;
if ($input eq "-n") {
   $numeric = 1;
   $input = shift;
}
my $output = shift;

open INPUT, $input or die "Couldn't open file $input: $!\n";
open OUTPUT, ">$output" or die "Couldn't open file $input: $!\n";

my @file = <INPUT>;
if ($numeric) {
   @file = sort { $a <=> $b } @file;
} else {
   @file = sort @file;
}

print OUTPUT @file;
```

What have we done? We've declared a flag, `$numeric`, which will tell us whether or not we're to do a numeric sort. If the first thing we see on the command line after our program's name is the string `-n`, then we're doing a numeric sort, and so we set our flag. Now that we've dealt with the `-n`, the input and output are the next two things on the command line. So we have to `shift` again.

Now that we've read the file in, we can choose which way we want to sort it: either normally, if `-n` was not given, or numerically if it was. If we have a file containing a list of numbers, called `sortnum.txt`, we can see the difference between the two methods:

>perl sort2.plx sortnum.txt sorted.txt

will write

```
121
1324515
13461
7446
576124
```

to the file `sorted.txt`, while:

>perl sort2.plx –n sortnum.txt sorted.txt

gives us:

```
121
7446
13461
576124
1324515
```

Try expanding the one-line version of `sort.plx` to match that.

Accessing Filehandles

Before we leave this program, there's one more thing we should do. One piece of programming design UNIX encourages is that it's better to string together lots of little things than deal with a huge program. Houses are built from individual bricks, not single lumps of rock. This is a design principle that's useful everywhere, not just on UNIX, and so let's try and make use of it here.

UNIX invented the use of **pipes** to connect programs. Perl supports these, and we'll see how they work later on. To make use of them, though, our program must be able to read lines from the standard input and put out sorted lines to the standard output in the event that no parameters were specified. Let's modify our program to do this:

Try It Out : Sort As A Filter

To see how many parameters have been passed, we'll test to see if `$input` and `$output` are defined after we shift them:

```
#!/usr/bin/perl
# sort3.plx
use warnings;
use strict;

my $numeric = 0;
my $input = shift;
if (defined $input and $input eq "-n") {
   $numeric = 1;
   $input = shift;
}
my $output = shift;

if (defined $input) {
   open INPUT, $input or die "Couldn't open file $input: $!\n";
} else {
   *INPUT = *STDIN;
}

if (defined $output) {
   open OUTPUT, ">$output" or die "Couldn't open file $input: $!\n";
} else {
   *OUTPUT = *STDOUT;
}

my @file = <INPUT>;
if ($numeric) {
   @file = sort { $a <=> $b } @file;
} else {
   @file = sort @file;
}

print OUTPUT @file;
```

This time, we'll give no parameters but instead pass data on the command-line using the left arrow:

```
> perl sort.plx < sortme.txt
And nail my feet up where my head should be
And you can all die laughing, because I'd wear it proudly
If they had a king of fools then I could wear that crown
Well, I finally found someone to turn me upside-down
>
```

As you can see, the data ends up on standard output. But how?

How It Works

The key magic occurs in the following lines:

```
if (defined $input) {
   open INPUT, $input or die "Couldn't open file $input: $!\n";
} else {
   *INPUT = *STDIN;
}
```

If there's an input file name defined, we use that. Otherwise, we do this strange thing with the stars. What we're doing is telling Perl that `INPUT` should be the same filehandle as standard input. If we wanted to set array `@a` to be the same as array `@b`, we'd say `@a = @b;` With filehandles, we can't just say `INPUT = STDIN;` we have to put a star before their names. From now on, everything that is read from `INPUT` will actually be taken from `STDIN`. Similarly, everything that is written to `OUTPUT` goes to `STDOUT`:

> **What the star – or, to give it its proper name, the glob – does is actually very subtle: `*a = *b` makes everything called `a` – that is `$a`, `@a`, `%a`, and the filehandle called `a` – into an alias for everything called `b`. This is more than just setting them to the same value – it makes them the same thing. Now everything that alters `$a` also alters `$b` and vice versa. That's why it's a good convention to keep filehandles purely upper-case. That keeps them distinct from other variables, meaning you won't inadvertently alias two variables.**

The reason we have to do this to manipulate filehandles is because there isn't a 'type symbol' for them as there is for scalars, arrays, and hashes. This is now seen as a mistake, but there's little we can do about it at this stage.

Writing Binary Data

So far, we've been dealing primarily with text files: speeches, Perl programs, and so on. When we get to data that's designed for computers to read – binary files – things change somewhat. The first problem is the newline character, `\n`. `This` is actually nothing more than a convenient fiction, allowing you to denote a new line using ASCII symbols on whatever operating system you're working with. In truth, different operating systems have differing ideas about what a newline really is when written to a file.

On UNIX, it really is `\n` – character number 10 in the ASCII sequence. When a Macintosh reads a file, the lines are separated by character 13 in the ASCII sequence, which you can generate by saying `\r`. A Macintosh version of Perl, then, will convert `\r` on the disk to `\n` in the program when reading in from a file, then write `\r` to the disk in place of `\n` when writing out to a file.

Windows, on the other hand, is different again. The DOS family of operating systems use `\r\n` on the disk to represent a new line. Therefore Perl has to silently drop or insert `\r` in the relevant places to make it look as if you're dealing with `\n` all the time.

When you're dealing with text, this is exactly what you want to happen. Perl's idea of a newline needs to correspond with the native operating system's idea of a newline – whatever that may be. However, with binary files, where every byte is important, you don't want Perl fiddling with the data just because it looks like the end of a line of text. You want those `\r`s to stay where they are!

Worse still, on DOS, Windows, and friends, character 26 is seen as the end of a file. Perl will stop reading once it sees this character, regardless of whether there's any data to follow.

To get a round both these problems, you need to tell Perl when you're reading from and writing to binary files, so that it can compensate. You can do this by using the `binmode` operator:

```
binmode FILEHANDLE;
```

To ensure your files are read and written correctly, **always** use `binmode` on binary files, **never** on text files.

Selecting a Filehandle

Normally, when you `print`, the data goes to the `STDOUT` filehandle. To send it somewhere else, you say `print FILEHANDLE ...;` However, if you're sending a lot of data to a file, you might not want to have to give the filehandle every time, it would be useful if it were possible to change the default filehandle. You can do this very simply by `select`ing the filehandle:

```
select FILEHANDLE;
```

This will change the default location for `print` to `FILEHANDLE`. Remember to set `STDOUT` back when you're done. A good use of this is to optionally send your program's output to a log file instead of the screen:

Try It Out : Selecting A Log File

This program does very little of interest; however, it does it using a log file. We'll use `select` to control where its output should go.

```
#!/usr/bin/perl
#logfile.plx
use warnings;
use strict;

my $logging = "screen";    # Change this to "file" to send the log to a file!

if ($logging eq "file") {
   open LOG, "> output.log" or die $!;
   select LOG;
}

print "Program started: ", scalar localtime, "\n";
sleep 30;
print "Program finished: ", scalar localtime, "\n";

select STDOUT;
```

As it is, the program will print something like this:

```
> perl logfile.plx
Program started: Sun Apr 22 14:17:07 2000
Program finished: Sun Apr 22 14:17:37 2000
>
```

However, if we change line 6 to this:

```
    my $logging = "file";
```

we apparently get no output at all:

```
> perl logfile.plx
>
```

However, we'll find the same style output in the file `output.log`. How?

How It Works

Since the value of `$logging` has changed, it's reasonable to assume that the difference is due to something acting on `$logging` – Perl is nice and deterministic like that. So, sure enough, on line 8, `$logging` gets examined:

```
    if ($logging eq "file") {
```

If `$logging` has the value `file`, which it does now:

```
    open LOG, "> output.log" or die $!;
```

We open a filehandle for writing, on the file `output.log`:

```
    select LOG;
```

Then we `select` that filehandle. Now any `print` statements that don't specify which filehandle to print to go out on `LOG`. If we wanted to write on standard output from now on, we'd have to write:

```
    print STDOUT "This goes to the screen.\n";
```

Or, alternatively, we could `select` standard output again:

```
    select STDOUT;
```

How did we get Perl to print out the time? The key is in this line:

```
    print "Program started: ", scalar localtime, "\n";
```

`localtime` is a function that returns the current time in the local time zone. Ordinarily, it returns a list like this:

```
($sec, $min, $hour,
$day_of_month,
$month_minus_one,
$year_minus_nineteen_hundred,
$day_of_week,
$day_of_year,
$is_this_daylight_savings_time)
```

Right now it would return:

53, 47, 14, (It's 14:47:53.)
22, (It's the 22nd)
3, (April is the third month of the year, counting from the zeroth)
100, (It's the year 2000.)
6, (It's a Saturday. Sunday is the first day of the week, day zero.)
112, (It's the 112th day of the year, counting from the zeroth. January the first is day zero.)
0 (It's not daylight savings time right now.)

> **Always be careful when dealing with `localtime`. Hopefully by now you see the merit in counting from zero when you're dealing with computers, but it can sometimes catch you out – the month of the year, day of the week and day of the year start from zero, but the day of the month starts from one.**

Thankfully, it's now a lot harder to imagine that the fifth element returned is the year. Last year `(localtime)[5]` returned `99`, which some foolhardy programmers assumed was the last two digits of the year. Fortunately, Perl turned out to be perfectly Y2K compliant, unfortunately, those programmers weren't. `(localtime)[5]` is (and has always been) the year minus 1900. If you find this weird and inconsistent, you can blame it all on the fact that Perl bases its idea on how to represent time from C, which first perpetrated this insanity.

In scalar context however, `localtime` provides a much easier value to deal with: It's a string representing the current time in a form designed for human consumption. This allows us to easily produce timestamps to mark when operations happened. However, we must remember that since `print` takes a list, we need to explicitly tell `localtime` to be in scalar context in order to force it to return this string.

Buffering

Try this little program:

```
#!/usr/bin/perl
#time.plx
use warnings;
use strict;

for (1...20) {
   print ".";
   sleep 1;
}
print "\n";
```

You'd probably expect it to print twenty dots, leaving a second's gap between each one – on Windows with ActiveState Perl, that's exactly what it does. However, this is something of an exception. On most other operating systems, you'll have to wait for twenty seconds first, before it prints all twenty at once.

So what's going on? Operating systems often won't actually write something to (or read something from) a filehandle until the end of the line. This is to save doing a lot of short, repetitious read/write operations. Instead, they keep everything you've written queued up in a buffer and access the filehandle once only.

However, you can tell Perl to stop the OS doing this, by modifying the special variable `$|`. If this is set to zero, which it usually is, Perl will tell the operating system to use output buffering if possible. If it's set to one, Perl turns off buffering for the currently selected filehandle.

So, to make our program steadily print out dots – as you might do to show progress on a long operation – we just need to set `$|` to 1 before we do our printing:

```
#!/usr/bin/perl
#time2.plx
use warnings;
use strict;

$| = 1;
for (1...20) {
   print ".";
   sleep 1;
}
print "\n";
```

If you need to turn off buffering when writing to a file, be sure to `select` the appropriate filehandle before changing `$|`, possibly selecting `STDOUT` again when you've done so.

Permissions

Before going on, let's look briefly at the issue of file permissions. If you use UNIX or other multi-user systems, you'll almost certainly be familiar with the very specific access controls that can be imposed, determining who's allowed to do what with any given file or directory. It's most likely that a file or directory on such a system will have at least three sets of permissions for each of three sets of users:

- The file owner,
- The group with which the owner is associated, and
- Everyone else on the system.

Each of these can have 'read', 'write' and 'execute' permissions assigned. You may have seen these displayed along with other file information as a sequence like:

```
drwxrwxrwx
```

which denotes full access on a directory (denoted by the prefix '`d`') for all users or:

```
-rwx--x---
```

which denotes a file (prefix '-') to which the owner has full access, members of their group can execute (but not read or modify), and everyone else has no access at all.

> *In fact, the subtleties of permission hierarchies mean that it's not always quite this clear cut. For example, a UNIX file without public 'write' permissions can actually be deleted by any user at all if the file's parent directory has granted them the relevant permission. Take care.*

Perl gives us the function `umask(`*`expr`*`)`, which we can use to set the permission bits to be used when we create a file. The expression it will expect is a three digit octal number, representing the state of the nine flags we've seen. If we consider these as bits in a binary number, we can interpret our second example above as:

```
111001000
```

which breaks down groupwise as:

```
(111) (001) (000)
```

and in octal as:

```
710
```

We can therefore specify `umask(0710);` and subsequent files will be created with any permissions it has been specifically given `AND`ed with the `umask` value. In a nutshell, by setting the umask value, we have set the default permissions for all files or directories on top of which other permissions can be set.

In general, it's a good idea to set the `umask` to `0666` for creating regular files. If you work backwards from the file, you realize that this equates to giving everyone read and write access to the file but no-one execute permissions. Likewise, it's a fairly safe bet to set `umask` to `0777` – full control for everyone – for the creation of directories and, of course, executable files.

Opening Pipes

`open` can be used for more than just plain old files. You can read data from and send data to programs as well. Anything that can read from or write to standard output can talk directly to Perl via a **pipe**.

Pipes were invented by a man called Doug MacIlroy for the UNIX operating system and were soon carried over to other operating systems. They're one of those things that sound amazingly obvious once someone else has thought of it:

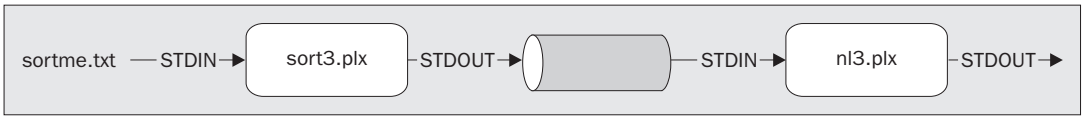
> **A pipe is something that connects two filehandles together.**

That's it. Usually, you'll be connecting the standard output of one program to the standard input of another.

For instance, we've written two filters in this chapter: one to number lines in a file and one to sort files. Let's see what happens when we connect them together:

```
> perl sort3.plx < sortme.txt | perl nl3.plx

        File: -

1: And nail my feet up where my head should be
2: And you can all die laughing, because I'd wear it proudly
3: If they had a king of fools then I could wear that crown
4: Well, I finally found someone to turn me upside-down
>
```

That bar in the middle is the pipe. Here's a diagram of what's going on:



The pipe turns the standard output of `sort3.plx` into input for `nl3.plx`.

While pipes are usually used for gluing programs together on the shell command line, exactly as we've just done above, we can use them in Perl to read from and write to programs.

Piping In

To read the output of a program, simply use `open` and the name of the program (with any command line you want to give it), and put a pipe at the end. For instance, the program `lynx` is a command-line web browser, available via http://lynx.browser.org/. If I say `lynx -source` http://www.perl.com/, `lynx` gets the HTML source to the page and sends it to standard output. I can pick this up from Perl using a pipe.

If you're using Windows, you may need to modify your global path settings – the list of directory paths in which Windows will look for perl, or lynx, or any other executable that you want to call without specifying it's location. It's only because `PATH` contains `C:\Perl\bin\` that we can say:

```
>perl <filename>
```

without saying anything about where `perl.exe` lives. On Windows 9x you can edit the default value of PATH inside the file `autoexec.bat`, which you'll find in the root directory. On Windows 2000, you'll find this under Start Menu|Program Files|Administrative Tools|Computer Management – call up Properties for the local machine, and it's on the Advanced tab, in Environment Variables.

Simply add the full path of the directory into which you've installed `lynx.exe`, separated from previous entries (you should see `C:\Perl\bin\` there already) by a semicolon. Mine now looks like this:

```
C:\PERL\BIN\;C:\PERL\BIN;C:\WINDOWS;C:\WINDOWS\COMMAND;C:\LYNX\DIST\
```

One simpler alternative is to enter this at the DOS command line:

```
set PATH=%PATH%;<add directory path to lynx.exe here>
```

This has the benefit of being quicker. It's also safer, as any modification you make like this is local to the current DOS shell, but that means you'll have to do it again next shell around…

You may still find that lynx still won't run from outside it's own directory and gives you a message like:

```
Configuration file ./lynx.cfg is not available.
```

You can get round this problem by copying the relevant file from the lynx directory into your current one. It's a bit of a fudge, but it does the trick:

Try it out : Perl headline

```
#!/usr/bin/perl
# headline.plx
# Display the www.perl.com top story.
use warnings;
use strict;

open LYNX, "lynx -source http://www.perl.com/ |" or die "Can't open lynx: $!";

# Define $_ and skip through LYNX until a line containing "standard.def"
$_ = "";
$_ = <LYNX> until /standard\.def/;

# The headline is in the following line:
my $head = <LYNX>;

# Extract "Headline" from "<A HREF=something>Headline</a>..."
$head =~ m|^<A HREF=[^>]+>(.*?)</a>|i;

print "Today's www.perl.com headline: $1\n";
```

Run today, this tells me:

```
>perl headline.plx
Today's www.perl.com headline: What's New in 5.6.0.
>
```

Note that this program will work with the layout of www.perl.com *at the time of writing. If the site's layout changes, it might not work in the future.*

How It Works

The important thing, for our purposes, is the pipe:

```
open LYNX, "lynx -source http://www.perl.com/ |" or die "Can't open lynx: $!";
```

What it's saying is that, instead of a file on the disk, the filehandle `LYNX` should read from the standard output of the command `lynx -source http://www.perl.com`. The pipe symbol `|` at the end of the string tells Perl to run the command and collect the output. The effect is just the same as if we'd had `lynx` write the output to a file and then had Perl read in that file. Each line we read from `LYNX` is the next line of source in the output.

Let's now have a look at how we extracted the headline from the source.

The site is laid out in a standard format, and the headline is on the line following the text "standard.def". So we can happily keep getting new lines until we come across one matching that text:

```
$_ = <LYNX> until /standard\.def/;
```

Note that we have to assign the new line to `$_` ourselves. The assignment to `$_` is only done automatically when you say something like `while(<FILEHANDLE>)`.

The headline is in the next line, so we get that:

```
my $head = <LYNX>;
```

The line containing the headline will look something like this:

```
<A HREF="http://www.perl.com/pub/2000/05/...">Perl used in wombat sexing</A>
```

To retrieve the headline from the middle, we use a regular expression. Generally speaking, reading HTML with a regular expression is a really bad idea, as `perlfaq9` explains. HTML tags are far more complex than just "start at an open bracket and end with a close bracket". That definition would fail spectacularly with tags in comments, tags split over multiple lines, or tags containing a close bracket symbol as part of a quoted string. It's a much harder problem than it first appears, due to the scope of the HTML language.

To read HTML to any degree of accuracy, you need to use an extension module like `HTML::Parser`. However, when the scope of the problem is as limited as the one we're faced with, we can get away with taking a few liberties.

We know that the piece of HTML in question is a single line. We know that the tag we're looking for starts at the beginning of the line and that there are no close brackets within it. So, our regular expression finds `"<A HREF="` at the beginning of the line. After that, we read anything that's not a closing bracket, followed by a closing bracket.

Next, we want our headline: This is the smallest amount of text that will be directly followed by `</A>`. Since there's a forward slash in what we're trying to match, we use alternate delimiters to make the expression more understandable. As we're using alternate delimiters, we need to put an `m` on the front to make it clear that this is a match:

```
$head =~ m|^<A HREF=[^>]+>(.*?)</A>|;
```

We could have said: `$head =~ /^<A HREF=[^>]+>(.*?)<\/A>/;` backslashing the forward slash to avoid it being treated as the end of the regular expression, but that would have been unnecessarily confusing. This is exactly the sort of situation that alternate delimiters were provided for, so we're right to make the most of them.

Why do we use `[^>]+` instead of `.*` or similar? Consider what would happen if there were two stories on the line:

```
<A HREF="http://www.perl.com/...">Perl is really cool</A><A HREF="...">Story 2</A>
```

`<A HREF=.*>` matches **as much as possible** before a close bracket. In this case, the most it can get before a close bracket would be to match everything up until just before Story 2, and we'd miss the main headline altogether. This is because `.` means everything, and everything includes a closing bracket. By saying [^>]+ we're making it clear that there can be no closing brackets in the text we're matching.

Piping Out

As well as reading data in from external programs, we can write out to the standard input of another program. For instance, we could send mail out by writing to a program like `sendmail`, or we could be generating output that we'd like to have sorted before it gets to the user. We'll deal with the second example because, while it's easy enough to collect the data into an array and sort it ourselves before writing it out, we know we have a sorting program handy. After all, we wrote one a few pages ago!

Try It Out : Taking Inventory

Things hide in the kitchen cabinet. Tins of tomatoes can lurk unseen for weeks and months, springing to vision only after I've bought another can. Every so often, then, I need to investigate the cabinets and take inventory to enumerate my baked beans and root out reticent ravioli. The following program can help me do that:

```
#!/usr/bin/perl
# inventory.plx
use warnings;
use strict;

my %inventory;
print "Enter individual items, followed by a new line.\n";
print "Enter a blank line to finish.\n";
while (1) {
   my $item = <STDIN>;
   chomp $item;
   last unless $item;
   $inventory{lc $item}++;
}

open(SORT, "| perl sort.plx") or *SORT = *STDOUT;
select *SORT;
while (my ($item, $quantity) = each %inventory) {
   if ($quantity > 1) {
      $item =~ s/^(\w+)\b/$1s/ unless $item =~ /^\w+s\b/;
   }
   print "$item: $quantity\n";
}
```

Now let's take stock:

```
>perl inventory.plx
Enter individual items, followed by a new line.
Enter a blank line to finish.
jar of jam
loaf of bread
tin of tuna
packet of pancake mix
tin of tomatos
tin of tuna
packet of pasta
clove of garlic
packet of pasta

clove of garlic: 1
jar of jam: 1
loaf of bread: 1
packet of pancake mix: 1
packets of pasta: 2
tin of tomatos: 1
tins of tuna: 2
```

As you can see, we get back a sorted list of totals.

How It Works

Whenever you're counting how many of each items you have in a list, you should immediately think about hashes. Here we use a hash to key each item to the quantity; each time we see another one of those items, we add to the quantity in the hash:

```
while (1) {
   my $item = <STDIN>;
   chomp $item;
   last unless $item;
   $inventory{lc $item}++;
}
```

The only way this infinite loop will end is if `$item` contains nothing after being chomped – it was nothing more than a new line.

To ensure that the capitalization of our item isn't significant, we use the operator `lc` to return a lower-case version of the item. Otherwise, "Tin of beans", "TIN OF BEANS" and "tin of beans" would be treated as three totally separate items, instead of three examples of the same thing. By forcing them into lower case, we remove the difference.

> *The `lc` operator returns the string it was given, but with upper-case characters turned into lower case. So `print lc("FuNnY StRiNg");` should give you the output 'funny string'. There's also a `uc` operator that returns an upper-cased version of the string, so `print uc("FuNnY StRiNg");` will output 'FUNNY STRING'.*

Next, we open our pipe. We're going to pass data from our program to another, external program. If you look up at the pipe diagrams above, you'll see that the data flows from left to right. Therefore, we want to put the command to run that external program on the right-hand side of the pipe:

```
open(SORT, "| perl sort.plx") or *SORT = *STDOUT;
```

If we can't successfully open the pipe – the program wasn't found or we couldn't execute Perl – we alias `SORT` to `STDOUT` to get an unsorted version.

Now we can print the data out:

```
while (my ($item, $quantity) = each %inventory) {
```

We use `each` to get each key/value pair from the hash, as explained in chapter 3.

```
    if ($quantity > 1) {
       $item =~ s/(\w+)/$1s/ unless $item =~ /\w+s\b/;
    }
```

This will make the output a little more presentable. If there is more than one of the current item, the name should be pluralized, unless it already ends in an 's'. `\w+` will get the first word in the string, and we add an 's' after it.

This is a relatively crude method for pluralizing English words, If you want to do it properly, there's a module on CPAN called `Lingua::EN::Inflect` that will do the trick.

```
    print "$item: $quantity\n";
```

Last of all, we print this out. It's actually going to the SORT filehandle, because that's the one that's currently selected – that filehandle is, in turn, connected to the sort program.

File Tests

So far, we've just been reading and writing files, and `die`ing if anything bad happens. For small programs, this is usually adequate, but if we want to use files in the context of a larger application, we should really check their status before we try and open them and, if necessary, take preventative measures. For instance, we may want to warn the user if a file we wish to overwrite already exists, giving them a chance to specify a different file. We'll also want to ensure that, for instance, we're not trying to read a directory as if it was a file.

> *This sort of programming – anticipating the consequences of future actions – is called defensive programming. Just like defensive driving, you assume that everything is out to get you. Files will not exist or not be writeable when you need them, users will specify things inaccurately, and so on. Properly anticipating, diagnosing, and working around these areas is the mark of a top class programmer.*

Perl provides us with **file tests**, which allow us to check various characteristics of files. These act as logical operators, and return a true or false value. For instance, to check if a file exists, we write this:

```
if (-e "somefile.dat") {...}
```

The test is `-e`, and it takes a file name as its argument. Just like `open`, this file name can also be specified from a variable. You can just as validly say:

```
if (-e $filename) {...}
```

where `$filename` contains the name of the file you want to check.

For a complete list of file tests, see Appendix C. The table below shows the most common ones:

Test	Meaning
`-e`	True if the file exists.
`-f`	True if the file is a plain file – not a directory.
`-d`	True if the file is a directory.
`-z`	True if the file has zero size.
`-s`	True if the file has nonzero size – returns size of file in bytes.
`-r`	True if the file is readable by you.
`-w`	True if the file is writable by you.
`-x`	True if the file is executable by you.
`-o`	True if the file is owned by you.

The last four tests will only make complete sense on operating systems for which files have meaningful permissions, such as UNIX and Windows NT. If this isn't the case, they'll frequently *all* return true (assuming the file or directory exists). So, for instance, if we're going to write to a file, we should check to see whether the file already exists, and if so, what we should do about it.

Note that on systems that don't use permissions comprehensively, `-w` is the most likely of the last four tests to have any significance, testing for Read-only *status. On Windows 9x, this can be found (and modified) on the* General *tab of the file's* Properties *window:*

Try It Out : Paranoid File Writing

This program does all it can to find a safe place to write a file:

```
#!/usr/bin/perl
# filetest1.plx
use warnings;
use strict;
```

```
my $target;
while (1) {
   print "What file should I write on? ";
   $target = <STDIN>;
   chomp $target;
   if (-d $target) {
      print "No, $target is a directory.\n";
      next;
   }
   if (-e $target) {
      print "File already exists. What should I do?\n";
      print "(Enter 'r' to write to a different name, ";
      print "'o' to overwrite or\n";
      print "'b' to back up to $target.old)\n";
      my $choice = <STDIN>;
      chomp $choice;
      if ($choice eq "r") {
         next;
      } elsif ($choice eq "o") {
         unless (-o $target) {
            print "Can't overwrite $target, it's not yours.\n";
            next;
         }
         unless (-w $target) {
            print "Can't overwrite $target: $!\n";
            next;
         }
      } elsif ($choice eq "b") {
         if ( rename($target,$target.".old") ) {
            print "OK, moved $target to $target.old\n";
         } else {
            print "Couldn't rename file: $!\n";
            next;
         }
      } else {
         print "I didn't understand that answer.\n";
         next;
      }
   }
   last if open OUTPUT, "> $target";
   print "I couldn't write on $target: $!\n";
   # and round we go again.
}
print OUTPUT "Congratulations.\n";
print "Wrote to file $target\n";
```

So, after all that, let's see how it copes, first of all with a text file that doesn't exist:

```
> perl filetest1.plx
What file should I write on? test.txt
Wrote to file test.txt
>
```

Seems OK. What about if I 'accidentally' give it the name of a directory? Or give it a file that already exists? Or give it a response it's not prepared for?

```
> perl filetest1.plx
What file should I write on? work
No, work is a directory.
What file should I write on? filetest1.plx
File already exists. What should I do?
(Enter 'r' to write to a different name, 'o' to overwrite or
'b' to back up to filetest1.plx.old)
r
What file should I write on? test.txt
File already exists. What should I do?
(Enter 'r' to write to a different name, 'o' to overwrite or
'b' to back up to test.txt.old)
g
I didn't understand that answer.
What file should I write on? test.txt
File already exists. What should I do?
(Enter 'r' to write to a different name, 'o' to overwrite or
'b' to back up to test.txt.old)
b
OK, moved test.txt to test.txt.old
Wrote to file test.txt
>
```

How It Works

The main program takes place inside an infinite loop. The only way we can exit the loop is via the `last` statement at the bottom:

```
last if open OUTPUT, "> $target";
```

That `last` will only happen if we're happy with the file name and the computer can successfully open the file. In order to be happy with the file name, though, we have a gauntlet of tests to run:

```
if (-d $target) {
```

We need to first see whether or not what has been specified is actually a directory. If it is, we don't want to go any further, so we go back and get another file name from the user:

```
print "No, $target is a directory.\n";
next;
```

We print a message and then use `next` to take us back to the top of the loop.

Next, we check to see whether or not the file already exists. If so, we ask the user what we should do about this.

```
if (-e $target) {
   print "File already exists. What should I do?\n";
   print "(Enter 'r' to write to a different name, ";
   print "'o' to overwrite or\n";
   print "'b' to back up to $target.old\n";
   my $choice = <STDIN>;
   chomp $choice;
```

If they want a different file, we merely go back to the top of the loop:

```
   if ($choice eq "r") {
      next;
```

If they want us to overwrite the file, we see if this is likely to be possible:

```
   } elsif ($choice eq "o") {
```

First, we see if they actually own the file; it's unlikely they'll be allowed to overwrite a file that they do not own.

```
      unless (-o $target) {
         print "Can't overwrite $target, it's not yours.\n";
         next;
      }
```

Next we check to see if there are any other reasons why we can't write on the file, and if there are, we report them and go around for another file name:

```
      unless (-w $target) {
         print "Can't overwrite $target: $!\n";
         next;
      }
```

If they want to back up the file, that is, rename the existing file to a new name, we see if this is possible:

```
   } elsif ($choice eq "b") {
```

The `rename` operator renames a file; it takes two arguments: the current file name and the new name.

```
      if ( rename($target,$target.".old") ) {
         print "OK, moved $target to $target.old\n";
      } else {
```

If we couldn't rename the file, we explain why and start from the beginning again:

```
         print "Couldn't rename file: $!\n";
         next;
      }
```

Otherwise, they said something we weren't prepared for:

```
} else {
   print "I didn't understand that answer.\n";
   next;
}
```

You may think this program is excessively paranoid, after all, it's 50 lines just to print a message on a file. In fact, it isn't paranoid enough: it doesn't check to see whether the backup file already exists before renaming the currently existing file. This just goes to show you can never be too careful when dealing with the operating system. Later, we'll see how to turn big blocks of code like this into reusable elements so we don't have to copy that lot out every time we want to safely write to a file.

Directories

As well as files, we can use Perl to examine directories on the disk. There are two major ways to look at the contents of a directory:

Globbing

If you're used to using the command shell, you may well be used to the concept of a **glob**. It's a little like a regular expression, in that it's a way of matching file names. However, the rules for globs are much simpler. In a glob, `*` matches any amount of text.

So, if I were in a directory containing files: `00INDEX 3com.c 3com.txt perl mail Mail`

- `*` would match everything.
- `3*` would match `3com.c` and `3com.txt`.
- `?ail` would match `mail` and `Mail`.
- `*l` would match `perl`, `mail` and `Mail`.

We can do this kind of globbing in Perl: the glob operator takes a string and returns the matching files:

```
#!/usr/bin/perl
# glob.plx
use warnings;
use strict;

my @files = glob("*l");
print "Matched *l : @files\n";
```

```
>perl glob.plx
perl mail Mail
>
```

To get all the files in a directory, you would say `my @files = glob("*");`

Reading Directories

That's the simple way. For more flexibility, you can read files in a directory just like lines in a file. Instead of using `open`, you use `opendir`. Instead of getting a filehandle, you get a **directory handle**:

```
opendir DH, "." or die "Couldn't open the current directory: $!";
```

Now to read each file in the directory, we use `readdir` on the directory handle:

Try It Out : Examining A Directory

This program lists the contents of the current directory and uses filetests to examine each file.

```
#!/usr/bin/perl
# directory.plx
use warnings;
use strict;

print "Contents of the current directory:\n";
opendir DH, "." or die "Couldn't open the current directory: $!";
while ($_ = readdir(DH)) {
        next if $_ eq "." or $_ eq "..";
        print $_, " " x (30-length($_));
        print "d" if -d $_;
        print "r" if -r _;
        print "w" if -w _;
        print "x" if -x _;
        print "o" if -o _;
        print "\t";
        print -s _ if -r _ and -f _;
        print "\n";
}
```

Part of its output looks like this:

```
>perl directory.plx
Contents of the current directory:
...
directory.plx           rwxo    449
filetest1.plx           rwxo    1199
inventory.plx           rwxo    515
mail                    drwxo
nl.plx                  rwxo    240
todo.log                rwo     3583
...
>
```

The number at the end is the size of the file in bytes; as for the letters, 'd' shows that this is a directory, 'r' stands for readable, 'w' for writable, 'x' for executable, and 'o' shows that I am the owner.

How It Works

As we've seen on the previous page, once we've opened our directory handle, we can read from it. We read one file name at a time into $_, and while there's still some information there, we examine it more closely:

```
while ($_ = readdir(DH)) {
```

The files . and .. are special directories on DOS and UNIX, referring to the current and parent directories, respectively. We skip these in our program:

```
next if $_ eq "." or $_ eq "..";
```

We then print out the name of each file, followed by some spaces. The length of the file name plus the number of spaces will always add up to thirty, so we have nicely arranged columns:

```
print $_, " " x (30-length($_));
```

First we test to see if the file is a directory, using the ordinary filetests we saw above:

```
print "d" if -d $_;
```

No, this isn't a typo: I do mean _ and not $_ here. Just as $_ is the default value for some operations, such as `print`, _ is the default filehandle for filetests. It actually refers to the last file explicitly tested. Since we tested $_ above, we can use _ for as long as we're referring to the same file:

```
print "r" if -r _;
print "w" if -w _;
```

> *When Perl does a filetest, it actually looks up all the data at once – ownership, readability, writeability and so on; this is called a* stat *of the file. _ tells Perl not to do another stat, but to use the data from the previous one. As such, it's more efficient that stat-ing the file each time.*

Finally, we print out the file's size. This is only possible if we can read the file and only useful if the file is not a directory:

```
print -s _ if -r _ and -f _;
```

Summary

Files give our data permanence by allowing us to store it on the disk. It's no good having the best accounting program in the world, if it loses all your accounts every time the computer is switched off. What we've seen here are the fundamentals of getting data in and out of Perl. In our chapter on Databases, we'll see more practical examples of how to read structured files into Perl data structures and write them out again.

Files are accessed through filehandles. To begin with, we have standard input, standard output, and standard error. We can open other filehandles, either for reading or for writing, with the `open` operator, and we must always remember to check what happened to the `open` call.

The diamond operator `<FILEHANDLE>` reads a line in from the specified filehandle. We can control the definition of a line by altering the value of the record separator, held in special variable `$/`.

Writing to a file is done with the `print` operator. Normally, this writes to standard output, so the filehandle must be specified. Alternatively, you may `select` another filehandle as the recipient of `print`'s output.

Pipes can be used to talk to programs outside of Perl. We can read in and write out data to them as if we were looking at the screen or typing on the keyboard. We can also use them as filters to modify our data on the way in or out of a program.

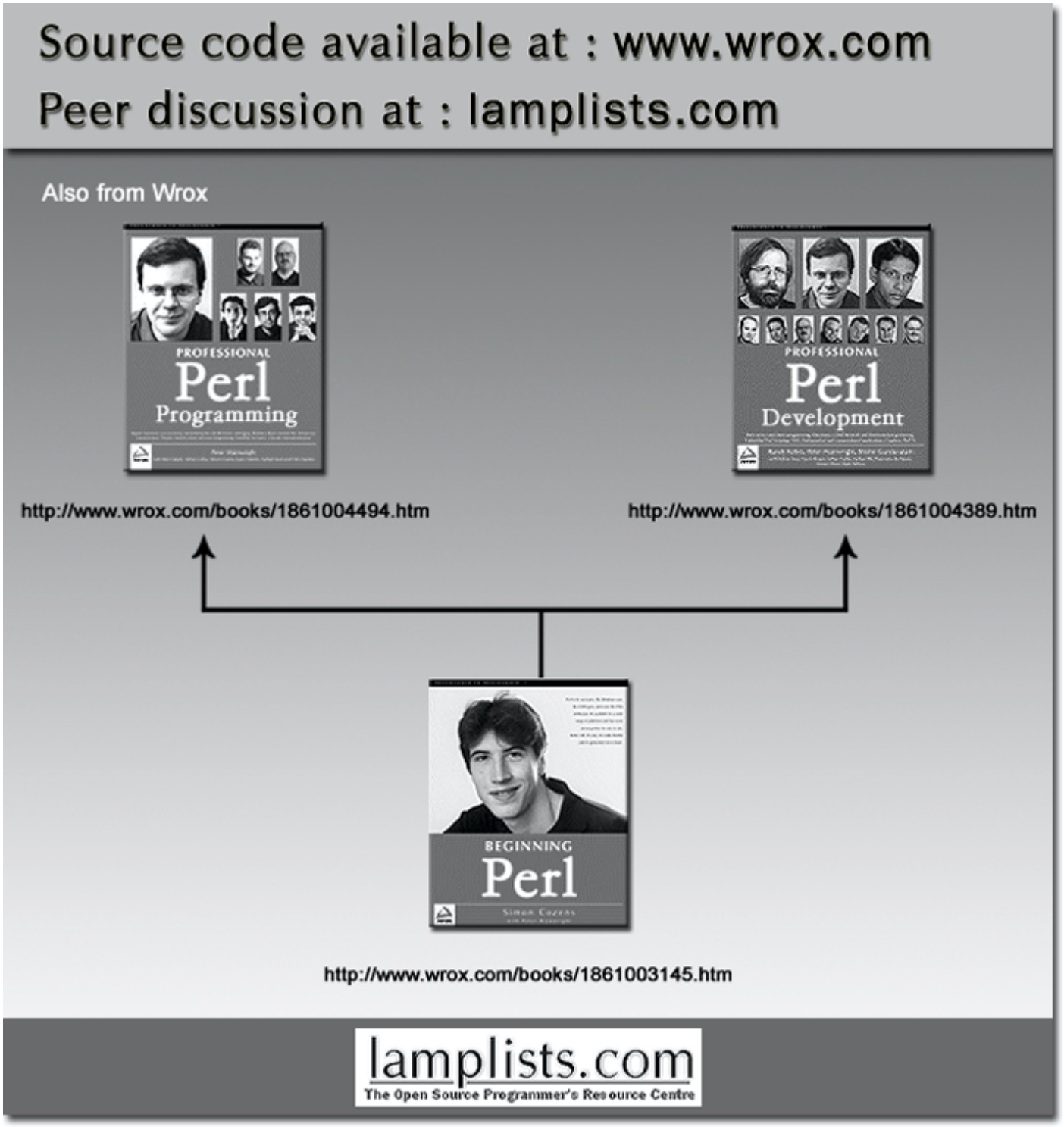
Filetests can be used to check the status of a file in various ways, and we've seen an example of using filetests to ensure that there are no surprises when we're reading or writing a file.

Finally, we've seen how to read files from directories using the `opendir` and `readdir` operators.

Exercises

1. Write a program that can search for a specified string within all the files in a given directory.

2. Modify the file backup facility in filetest1.plx so that it checks to see if a backup already exists before renaming the currently existing file. When a backup does exist, the user should be asked to confirm that they want to overwrite it. If not, they should be returned to the original query.



1

First Steps In Perl

Vitually all programming languages have certain things in common. The fundamental concepts of programming are the same, no matter what language you do them in. In this chapter, we'll investigate what you need to know before you start writing any programs at all. For instance:

- What is programming anyway? What does it mean to program?
- What happens to the program that we write?
- How do we structure programs and make them easy to understand?
- How do computers see numbers and letters?
- How do we find and eliminate errors in our programs?

Of course, we'll be looking at these from a Perl perspective, and we'll look at a couple of basic Perl programs, and see how they're constructed and what they do. At the end of this chapter, I'm going to ask you to write a couple of simple Perl programs of your own.

Programming Languages

The first question I suppose we really should ask ourselves when we're learning programming is, 'What is programming?' That may sound particularly philosophical, but the answer is easy. Programming is telling a computer what you want it to do. The only trick, then, is to make sure that the program is written in a way the computer can understand. and to do this, we need to write it in a language that it can comprehend – a programming language, such as Perl.

Writing a program does not require special skills, but it does call for a particular way of thinking. When giving instructions to humans, you take certain things for granted.

- Humans can ask questions if we don't understand instructions.
- We can break up complex tasks into manageable ones.
- We can draw parallels between the current task and ones we have completed in the past.
- Perhaps most importantly, we can learn from demonstrations and from our own mistakes.

Computers can't yet do any of these things very well – it's still much easier to explain to someone how to tie their shoelaces than it is to set the clock on the video machine.

The most important thing you need to keep in mind, though, is that you're never going to be able to express a task to a computer if you can't express it to yourself. Computer programming leaves little room for vague specifications and hand waving. If you want to write a program to, say, remove useless files from your computer, you need to be able to explain how to determine whether a file is useless or not. You need to examine and break down your own mental processes when carrying out the task for yourself: Should you delete a file that hasn't been accessed for a long time? How long, precisely? Do you delete it immediately, or do you examine it? If you examine it, how much of it? And what are you examining it for?

The first step in programming is to stop thinking in terms of 'I want a program that removes useless files,' but instead thinking 'I want a program that looks at each file on the computer in turn and deletes the file if it is over six months old and if the first five lines do not contain any of the words 'Simon', 'Perl' or 'important'. In other words, you have to specify your task precisely.

When you're able to restructure your question, you need to translate that into the programming language you're using. Unfortunately, the programming language may not have a direct equivalent for what you're trying to say. So, you have to get your meaning across using what parts of the language are available to you, and this may well mean breaking down your task yet further. For instance, there's no way of saying 'if the first five lines do not contain any of the following words' in Perl. However, there is a way of saying 'if a line contains this word', a way of saying 'get another line', and 'do this five times'. Programming is the art of putting those elements together to get them to do what you want.

So much for what you have to do – what does the computer have to do? Once we have specified the task in our programming language, the computer takes our instructions and performs them. We call this **running** or **executing** the program. Usually, we'll specify the instructions in a file, which we edit with an ordinary text editor; sometimes, if we have a small program, we can get away with typing the whole thing in at the command line. Either way, the instructions that we give to the computer – in our case, written in Perl – are collectively called the **source code** (or sometimes just **code**) to our program.

Interpreted vs. Compiled Source Code

What exactly does the computer do with our source code, then? Traditionally, there were two ways to describe what computer languages did with their code: You could say they were **compiled**, or that they were **interpreted**.

An interpreted language, such as Basic, needs another program called an **interpreter** to process the source code every time you want to run the program. This translates the source code down to a lower level for the computer's consumption as it goes along. We call the lower-level language **machine code**, because it's for machines to read, whereas **source code** is for humans. While the latter can look relatively like English, for example, (`"do_this() if $that"`), machine code looks a lot more like what you'd expect computers to be happier with, for example, `"4576616E67656C6961"`, and that's the easy-to-read version! The exact machine code produced depends on the processor of the computer and the operating system it runs, the translation would be very different for an x86 computer running Windows NT compared to a Sun or Digital computer running Unix.

A compiled language, on the other hand, such as C, uses a compiler to do all this processing one time only before the code is ever run. After that, you can run the machine code directly, without needing the

compiler any more. Because you don't need to process the source code every time you run it, compiled code will usually run faster than an interpreted equivalent. You can also give the compiled code to people who don't have a compiler themselves. This will prevent other people from reading your source code – handy if you're using a proprietary algorithm or if your code is particularly embarrassing. However, because you're distributing machine code that not all types of computers can understand, this isn't necessarily portable.

Recent languages have blurred the compiled/interpreted distinction. Java and Perl both class as 'byte-compiled' languages so they have been particularly blurry. In the case of Perl, where the interpreter (which we'll always call **perl** with a small 'p') reads your source code, it actually compiles the whole program at once. However, instead of compiling into the machine code spoken by the computer you happen to be on, it compiles into a special **virtual machine** code for a fictitious computer. Java's 'virtual machine' is quite like a normal computer's processor in terms of what it can do, and people have tried building processors that can speak the Java virtual machine code 'natively'. In comparison, Perl's virtual machine doesn't much resemble any existing computer processor and is far less likely to be built.

Once you've got this machine code, which we call **bytecode**, you can do a number of things with it. You can:

- Save it away to be run later.
- Translate it to the native machine code of your computer, and run that instead.
- Run it through a program that pretends to be the virtual machine and steps through the bytecode, and performs the appropriate actions.

We don't really do the first of these in Perl, although Java does. The 'Perl compiler' tries to do the second, but it's a very tricky job, and hasn't quite accomplished it. Normally, however, we do the third, and so after perl has finished compiling the source into bytecode, it then takes on the role of interpreter, translating the virtual machine code into real code. Hence Perl isn't strictly a compiled language or an interpreted one.

What some people will say is that Perl is a 'scripting' language, by which they probably mean an interpreted language. As we've seen, that's not actually true. However, be aware that you might hear the word 'script' where you might expect 'program'.

Libraries, Modules, and Packages

A lot of people use Perl. One consequence of this is that, unsurprisingly, a lot of Perl code has been written. In fact, a lot of the Perl code that you will ever need to write has probably already been written before. To avoid wasting time reinventing the wheel, Perl programmers package up the reusable elements of their code and distribute it, notably on CPAN – the Comprehensive Perl Archive Network – which you can find online at `http://www.perl.com/CPAN/`.

The biggest section of CPAN deals with Perl **modules**. A module is a file or a bundle of files that helps accomplish a task. There is a module for laying out text in paragraphs, one for drawing graphs, and even one for downloading and installing other modules. Your programs can use these modules and acquire their functionality. Later on, we'll devote the whole of Chapter 10 to using, downloading, and writing modules.

Closely linked to the idea of a module is the concept of a **package**, which is another way to divide up a program. By using packages, you can be sure that what you do in one section of your program does not affect another section. Whereas a module works with a file or bunch of files on your disk, a package is purely part of the source code. A single file, for instance, can contain several packages. Conversely, a package can be spread over several files. A module typically lives in its own package, to keep it distinct from the code that you write and to keep it from interfering. Again, we'll come to this later on in Chapter 10.

Every Perl installation comes with a collection of 'core modules'. The **core**, unsurprisingly, is the collective term for the files that are installed with your Perl distribution. At times, they're also referred to as the 'module library', although this could cause confusion if you intend to look back at older Perl code: 'library files' were used in Perl in versions 4 and earlier until replaced by modules in Perl 5. They are the same thing – pieces of code that you can use in your program to do a job that's been done before. However, they didn't have a package of their own, and so they put themselves in the same package as the rest of your program. It's also fairly simple to spot which file is a library and which is a module – the extension for a library file is usually `.pl`, whereas the extension for a module is `.pm`.

The result of this is that the module library contains library files as well as modules, and so it's hopelessly unclear what 'library' refers to any more. From now on, if we talk about a 'library', we're referring to the collection of files distributed with Perl, rather than Perl 4 library files; we won't be doing any work with library files (while library files have more or less been replaced by modules, they can still be useful) but will use the new-style modules instead.

Why Is Perl Such A Great Language?

Perl is in use on millions of computers, and it's one of the fastest-growing programming languages available. Why is this? We've already seen a number of reasons for this in the introduction, but I think it's worth restating them briefly here.

It's Really Easy

Perl is not a difficult language to learn. It's a language that tries to shape itself around the way humans think about problems and provides nothing contrary to their expectations. Perls' designers believe that Perl is a populist language – and not just for the mathematicians and computer scientists of this world. I know plenty of people with scientific and non-scientific backgrounds alike who successfully use Perl.

Flexibility Is Our Watchword

Perl doesn't want you to see things the way the computer does – that's not what it's for. Instead, Perl allows you to develop your personal approach to programming. It doesn't say that there's one right or wrong way to get a job done. In fact, it's quite the opposite – the Perl motto is "There's more than one way to do it", and Perl allows you to program whichever way makes most sense to you.

Perl on the Web

Perl's influence is not felt among the shell scripters of the world alone. Not only can it be used for rooting around in directories or renaming files, it also has massive importance in the world of **CGI**

scripting out on the World Wide Web. You'll find lots of Perl automating communication between servers and browsers world-wide and in more than one form. **Perlscript** is a (relatively new) derivation of Perl into a proper scripting language that can run both client- and server-side web routines, just as Javascript can. As we've said however, Perl's main function on the web is as a way to script CGI routines.

For a while, CGI was the standard way for a web server to communicate with other programs on the server, allowing the programs to do the hard work of generating content in a web page while the server dedicated itself to pass that content onto browsers as fast as it could. Of course, web pages are completely text-based and, thanks to its excellent text-handling abilities, PerlCGI set the standard for web server automation in the past. It's CGI (and Perl) that we have to thank for the wonderfully dynamic web pages we have become accustomed to on the Internet.

Later on in Chapter 12, we will explore the world of CGI in some detail, and among other things, we'll also see how to write CGI scripts using Perl. For the moment, however, let's get back to learning about Perl itself. If you would like to take a look, more information on PerlCGI and PerlScript is available at `www.fastnetltd.ndirect.co.uk/Perl/index.html`.

The Open Source Effort

Perl is free. It belongs to the world. It's Larry Wall's creation language, of course, but anyone in the world can download it, use it, copy it, and help make improvements. Roughly six hundred people are named in the changes files for assisting in the evolution from Perl 4.0 to Perl 5.0 to Perl 5.6, and that doesn't include the people who took the time to fill in helpful bug reports and help us fix the problems they had.

When I say 'anyone can help', I don't mean anyone who can understand the whole of the Perl source code. Of course, people who can knuckle down and attack the source files are useful, but equally useful work is done by the army of volunteers who offer their services as testers, documenters, proofreaders and so on. Anyone who can take the time to check the spelling or grammar of some of the core documentation can help, as can anyone who can think of a new way of explaining a concept, or anyone who can come up with a more helpful example for a function.

Perl development is done in the open, on the **perl5-porters** mailing list. The `perlbug` program, shipped with Perl, can be used to report problems to the list, but it's a good idea to check to make sure that it really is a problem and that it isn't fixed in a later or development release of Perl.

Developers Releases and Topaz

Perl is a living language, and it continues to evolve. The development happens on two fronts:

Stable releases of Perl, intended for the general public, have a version number x.y.z where z is less than 50. Currently, we're at 5.6.0; the next major stable release is going to be 5.8.0. Cases where z is more than 0 are maintenance releases issued to fix any overwhelming bugs. This happens extremely infrequently. For example, the 5.5 series had three maintenance releases in approximately one year of service.

Meanwhile between stable releases, the porters work on the development track, (where y is odd). When 5.6.0 was released, work began on 5.7.0 (the development track) to eventually become 5.8.0. Naturally, releases on the development track happen much more frequently than those on the stable track, but don't think that you should be using a development Perl to get the latest and greatest features or just because your stable version of last year seems old in comparison to the bright and shiny Perl released last week. No guarantees whatsoever are made about a development release of Perl.

Releases are coordinated by a 'patch pumpkin holder', or 'pumpking' – a quality controller who, with help from Larry, decides which contributions make the grade and when and bears the heavy responsibility of releasing a new Perl. He or she maintains the most current and official source to Perl, which they sometimes make available to the public: Gurusamy Sarathy is the current pumpkin, and keeps the very latest Perl at `ftp://ftp.linux.activestate.com/pub/staff/gsar/APC/perl-current/`

> *Why a pumpkin? To allow people to work on various areas of Perl at the same time and to avoid two people changing the same area in different ways, one person has to take responsibility for bits of development, and all changes are to go through them. Hence, the person who has the patch pumpkin is the only person who is allowed to make the change. Chip Salzenburg explains:*
>
> *'David Croy once told me once that at a previous job, there was one tape drive and multiple systems that used it for backups. But instead of some high-tech exclusion software, they used a low-tech method to prevent multiple simultaneous backups: a stuffed pumpkin. No one was allowed to make backups unless they had the "backup pumpkin".'*

So what development happens? As well as bug fixes, the main thrust of development is to allow Perl to build more easily on a wider range of computers and to make better use of what the operating system and the hardware provides for example support for 64-bit processors. (The Perl compiler, mentioned above, is steadily getting more useful but still has a way to go.) There's also a range of optimizations to be done, to make Perl faster and more efficient, and work progresses to provide more helpful and more accurate documentation. Finally, there are a few enhancements to Perl syntax that are being debated – the 'Todo' file in the Perl source kit explains what's currently on the table.

The other line of development that's going on is the **Topaz** project, led by Chip Salzenburg, an attempt to rewrite the entirety of Perl in C++. Compared to the main development track, this is going slowly but steadily. Topaz is by no means ready for use; currently, it can merely emulate some of the Perl internals; there is no interpreter or compiler yet and probably will not be for some time. However, it's expected that Topaz development will speed up in the near future. The homepage of the Topaz project is `http://topaz.sourceforge.net/`.

Our First Perl Program

I'm assuming that by now you've got a copy of Perl installed on your machine after following the instructions in the introduction. If so, you're ready to go. If not, go back and follow the instructions. What we'll do now is set up a directory for all the examples we'll use in the rest of the book and write our first Perl program.

Here's what it'll look like:

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

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

The 'Hello World' example is the traditional incantation to the programming gods and will ensure your quick mastery of the language, so please make sure you actually do this exercise, instead of just reading about it.

Before we go any further however, a quick note on editors. Perl source code is just plain text and should be written with a plain text editor, rather than a word processor. If you're using Windows, you really will want to investigate getting hold of a good programmer's editor. Notepad may be fine for this example, despite its annoying tendency to want to rename file extensions to `.plx.txt` for you, but I wouldn't recommend its use beyond that. WordPad also renames file extensions for you, and additionally, you must remember to save as plain text, not Word format. Edit was bearable, but no longer ships with Windows versions after 95.

A decent editor will help you with bracket matching indentation and may even use different colors to point out different parts of your code. You will almost certainly want to view and edit your code in a fixed-width font. The usual Unix editors, vi, emacs, and so on are perfectly suitable, and versions ('ports') of these are available for Windows – I personally use a port of vim a vi-like editor – available at `http://shareware.cnet.com` –, when programming on Windows.

Right then, back to the code.

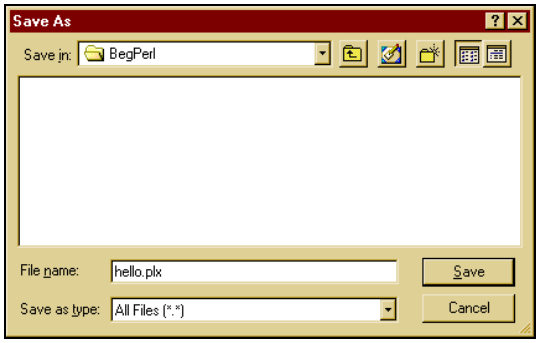
If You are a Windows User

1. Open Windows Explorer. Left click on the icon for your C: drive and choose New | Folder from the File menu.

2. Give the folder the name 'BegPerl' and press `Return`.

3. Open Notepad, which you'll find in the Programs | Accessories menu under the Start button, and type in the two lines of code as shown above.

4. Choose Save As from the File menu and change the menu option in Save as type to All Files (*.*). Find your `BegPerl` folder, and save the file as `hello1.plx`. The caption box should look this.



5. Click `Save` and then exit Notepad.

6. It's possible that Notepad will have renamed your file `hello1.plx.txt`, so in Windows Explorer, go to the `BegPerl` folder. If it has been renamed, right-click on the file and select Rename. Rename the file back to `hello1.plx`

7. The icon should change to a picture of a pearl – double click on it, and you'll see a window appear briefly and disappear before you have time to read it. This is your first lesson about clicking on Perl programs – a window will open to run them in, run them, and then close as soon as they are finished. In order to actually keep the results of our program on screen, we need to open an MS-DOS Prompt window first. So let's do that.

8. Click Start and select MS-DOS Prompt from the Programs menu. Type `cd c:\BegPerl` and press *Return*.

   Type `perl hello1.plx` – If Perl is in your path and all is well, this is what you should see on screen:

```
>perl hello1.plx
Hello, World.

>
```

Congratulations. You've successfully run your first piece of code!

If You're A Unix User

1. Open up a terminal window if you haven't already got one open, and `cd` to your home directory.

2. Type `mkdir begperl; cd begperl`

3. Open your favorite editor and edit `hello1.plx` – for example, `vi hello1.plx`

4. Confirm that your Perl distribution has been installed in `/usr/bin/perl` as the first line suggests, by typing `which perl` – if this doesn't give you anything, try `whence perl`. If the result is not `/usr/bin/perl`, be prepared to make appropriate changes.

5. Type in the two lines of code as shown above, save, and exit.

6. Run the file by typing `perl hello1.plx` – you should get similar output to the Windows users:

```
>perl hello1.plx
Hello, World.
>
```

Note that from this point on, we'll not run through these steps again. Instead, the name we've given the file will be shown as a comment on the second line of the program.

You may also have noticed that the output for hello1.plx on Windows and Unix differs in that Windows adds a silent print \n to all its perl programs. From now on, we'll only print the Unix output that is more strict. Windows users please be aware of this.

How It Works

So, all being well, your Perl program has greeted the light of day. Let's see how it was done, by going through it a line at a time. The first line is:

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

Now normally, Perl treats a line starting with # as a comment and ignores it. However, the # and ! characters together at the start of the first line tell Unix how the file should be run. In this case the file should be passed to the perl interpreter, which lives in `/usr/bin/perl`.

Perl also reads this line, regardless of whether you are on Unix, Windows, or any other system. This is done to see if there are any special options it should turn on. In this case, `-w` is present, and it instructs perl to turn on additional warning reporting. Using this flag, or its alternative, is a very good habit to get into, and we shall see why in just a moment. But first, let's have a look at the second line of our program:

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

The `print` function tells perl to display the given text without the quotation marks. The text inside the quotes is not interpreted as code (except for some 'special cases') and is called a **string**. As we'll see later, strings start and end with some sort of quotation mark. The `\n` at the end of the quote is one of these 'special cases' – it's a type of escape sequence, which stands for 'new line'. This instructs perl to finish the current line and take the prompt to the start of a new one.

You may be wondering why `-w` is so helpful. Well, suppose we altered our program to demonstrate this and made two mistakes by leaving out `-w` and by typing `printx` instead of `print`. Then `hello1.plx` would look like this:

```
#!/usr/bin/perl

printx "Hello, world.\n";
```

Remember to save these changes in `Hello2.plx` before exiting your file. Now let's get back to the command prompt, and type:

```
>perl Hello2.plx
```

Instead of getting the expected

```
Hello, world.
>
```

the output we get has a plethora of rather-nasty looking statements like this:

```
String found where operator expected at hello.plx line 2, near "printx "hello, world. \n""
(Do you need to predeclare printx?)
syntax error at hello.plx line 2, near "printx "Hello, world. \n""
Execution of Hello.plx aborted due to compilation errors.
>
```

If we now correct one of our mistakes by including `-w` in our program, then `Hello2.plx` looks like this:

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

printx "Hello, world. \n";
```

Once we have saved this new change into the program, we can run it again. The output that we get now contains a warning as well as the error message, so the screen looks like this:

```
>perl hello2.plx

Unquoted string "printx" may clash with future reserved word at hello2.plx line 3.
String found where operator expected at hello.plx line 2, near "printx "hello, world. \n""
(Do you need to predeclare printx?)
Syntax error at hello2.plx line 2, near "printx "Hello, world. \n""
Execution of Hello2.plx aborted due to compilation errors.
```

On the surface of things, it may seem that we have just given ourselves more nasty-looking lines to deal with. But bear in mind that the first line is now a **warning** message and is informing us that perl has picked something up that may (or may not) cause problems later on in our program. Don't worry if you don't understand everything in the error message at the moment, just so long as you are beginning to see the usefulness of having an early-warning system in place.

For versions of Perl 5.6.x and higher, the `-w` switch *should be replaced with a* `use warnings` directive, which follows **after** the shebang line. Although `-w` will still be recognized by perl, it has been deprecated, and for arguments sake we will assume from now on that you have Perl 5.6.x or higher. The resulting "en vogue" (and correct) version of `hello.plx` then, will look like this:

```
#!/usr/bin/perl
use warnings;

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

Program Structure

One of the things we want to develop throughout this book is a sense of good programming practice. Obviously this will not only benefit you while using Perl, but in almost every other programming language, too. The most fundamental notion is how to structure and lay out the code in your source files. By keeping this tidy and easy to understand, you'll make your own life as a programmer easier.

Documenting Your Programs

As we saw earlier, a line starting with a sharp (#) is treated as a comment and ignored. This allows you to provide comments on what your program is doing, something that'll become extremely useful to you when working on long programs or when someone else is looking over your code. For instance, you could make it quite clear what the program above was doing by saying something like this:

```
#!/usr/bin/perl
use warnings;

# Print a short message
print "Hello, world.\n";
```

Actually, this isn't the whole story. A line may contain some Perl code, and be followed by a comment. This means that we can document our program 'inline' like this:

```
#!/usr/bin/perl
use warnings;

print "Hello, world.\n"; # Print a short message
```

When we come to write more advanced programs, we'll take a look at some good and bad commenting practice.

Keywords

There are certain instructions that perl recognizes and understands. The word `print` above was one such example. On seeing `print`, perl knew it had to print out to the screen whatever followed in quotes. Words that perl is already aware of are called **keywords**, and they come in several classes. `print` is one example of the class called **functions.** These are the verbs of a programming language, and they tell perl what to do. There are also control keywords, such as `if` and `else`. These are used in context like this:

```
if  Condition;
do this;

else
do this;
```

It's a good idea to respect keywords and not reuse them as names. For example, a little later on we'll learn that you can create and name a variable, and that calling your variable `$print` is perfectly allowable. The problem with this is that it leads to confusing and uninformative statements like `print $print`. It is always a good idea to give a variable a meaningful name, one that relates to its content in a logical manner. For example `$my_name`, `$telephone_number`, `@shopping_list`, and so on, rather than `$a`, `$b` and `%c`.

Statements and Statement Blocks

If functions are the verbs of Perl, then **statements** are the sentences. Instead of a full stop, a statement in Perl usually ends with a semicolon, as we saw above:

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

To print something again, we can add another statement:

```
print "Hello, world.\n";
print "Goodbye, world.\n";
```

There are times when you can get away without adding the semicolon, such as when it's absolutely clear to perl that the statement has finished. However, it is good practice to put a semicolon at the end of each statement. For example, you can miss out the final semicolon in the example above, without causing a problem. Missing out the first would be incorrect.

We can also group together a bunch of statements into a **block** – which is a bit like a paragraph – by surrounding them with braces: {…}. We'll see later how blocks are used to specify a set of statements that must happen at a given time and also how they are used to limit the effects of a statement. Here's an example of a block:

```
{
     print "This is";
     print "a block";
     print "of statements.\n";
}
```

Do you notice how I've used indentation to separate the block from its surroundings? This is because, unlike paragraphs, you can put blocks inside of blocks, which makes it easier to see on what level things are happening. This:

```
print "Top level\n";
{
     print "Second level\n";
     {
          print "Third level\n";
     }
     print "Where are we?";
}
```

is easier to follow than this:

```
print "Top level\n";
{
print "Second level\n";
{
print "Third level\n";
}
print "Where are we?";
}
```

As well as braces to mark out the territory of a block of statements, you can use parentheses to mark out what you're giving a function. We call the set of things you give to a function the **arguments**, and we say that we **pass** the arguments to the function. For instance, you can pass a number of arguments to the `print` function by separating them with commas:

```
print "here ", "we ", "print ", "several ", "strings.\n";
```

The `print` function happily takes as many arguments as it can, and it gives us the expected answer:

```
here we print several strings.
```

Surrounding the arguments with brackets clears things up a bit:

```
print ("here ", "we ", "print ", "several ", "strings.\n");
```

We can also limit the amount of arguments we pass by moving the brackets:

```
print ("here ", "we ", "print "), "several ", "strings.\n";
```

We only pass three arguments, so they're the ones that get printed:

```
here we print
```

What happens to the others? Well, we didn't give perl instructions, so nothing happens.

In the cases where semicolons or brackets are optional, the important thing to do is to use your judgment. Sometimes code will look perfectly clear without the brackets, but when you've got a complicated statement and you need to be sure of which arguments belong to which function, putting in the brackets can clarify your work. Always aim to help the readers of your code, and remember that these reader will more than likely include you.

ASCII and Unicode

Computers are, effectively, lumps of sand and metal. They don't know much about the world. They don't understand words or symbols or letters. They do, however, know how to count. As far as a computer is concerned, everything is a number, and every character, albeit a letter or a symbol, is represented by a number in a sequence. This is called a 'character set', and the character set that computers predominantly use these days is called the 'ASCII' sequence. If you're interested, you can find the complete ASCII character set in Appendix F for reference.

The ASCII sequence consists of 256 characters, running from character number 0 (all computers, and plenty of computer users, start counting from zero) to character number 255. The letter 'E', for instance, is number 69 in the sequence, and a plus sign (+) is number 43. 255 is a key number for computers and computer programmers alike, because it's the largest number you can store in one 'byte'.

The big problem with ASCII is that it's American. Well, that's not entirely the problem; the real reason is that it's not particularly useful for people who don't use the Roman alphabet. What used to happen was that particular languages would stick their own alphabet in the upper range of the sequence, between 128 and 255. Of course, we then ended up with plenty of variants that weren't quite ASCII, and the whole point of standardization was lost.

Worse still, if you've got a language like Chinese or Japanese that has hundreds or thousands of characters, then you really can't fit them into a mere 256. This meant that programmers had to forget about ASCII altogether and build their own systems using pairs of numbers to refer to one character.

To fix this, **Unicode** was developed by a number of computer companies, standards organizations, and bibliographic interests. It is currently maintained and developed by the Unicode Consortium, an organization in California. They have also produced a couple of new character sets, UTF8 and UTF16. UTF8 uses two bytes instead of one, so it can contain 65536 characters, which is enough for most people. You can learn more about Unicode at `http://www.unicode.org/`

Perl 5.6 introduces Unicode support. Previously, you could print any data that you were capable of producing in your editor or from external sources. However, the functions to translate between lower and upper case wouldn't necessarily work with Greek letters without a lot of support from your operating system. Now, if you have Unicode data, you can consider a single Japanese *kana* to be one character instead of two. So, if you use a Unicode editor for your programming:

- You can write your variable names in your native alphabet.
- You can match certain classes of symbol or character regardless of language, while processing data.

Escape Sequences

So, UTF8 gives us 65536 characters, and ASCII gives us 256 characters, but on the average keyboard, there only a hundred or so keys. Even using the shift keys, there will still be some characters that you aren't going to be able to type. There'll also be some things that you don't want to stick in the middle of your program, because they would make it messy or confusing. However, you'll want to refer to some of these characters in strings that you output. Perl provides us with mechanisms called 'escape sequences' as an alternative way of getting to them. We've already seen the use of `\n` to start a new line. Here are the more common escape sequences:

Escape Sequence	Meaning
`\t`	Tab
`\n`	Start a new line (Usually called 'newline')
`\b`	Back up one character ('backspace')
`\a`	Alarm (Rings the system bell)
`\x{1F18}`	Unicode character

In the last example, 1F18 is a hexadecimal number (see 'Number Systems' just below) referring to a character in the Unicode character set, which runs from 0000-FFFF. As another example, `\x{2620}` is the Unicode character for a skull-and-crossbones!

White Space

White space is the name we give to tabs, spaces, and new lines. Perl is very flexible about where you put white space in your program. We have already seen how we're free to use indentation to help show the structure of blocks. You don't need to use any white space at all, if you don't want to. If you prefer, your programs can all look like this:

```
print"Top level\n";{print"Second level\n";{print"Third level\n";}print"Where are
we?";}
```

Personally, though, I'd call that a bad idea. White space is another tool we have to make our programs more understandable. Let's use it as such.

Number Systems

If you thought the way computers see characters is complicated, we have a surprise for you.

The way most humans count is using the decimal system, or what we call base 10; we write 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and then when we get to 10, we carry 1 in the 10s column and start from 0 again. Then when the 10s column gets to 9 and the 1s column gets to 9, we carry 1 in the 100s column and start again. Why 10? We used to think it's because we have 10 fingers, but then we discovered that the Babylonians counted up to 60, which stopped that theory.

On the other hand, computers count by registering whether or not electricity flows in a certain part of the circuit. For simplicity's sake, we'll call a flow of electricity a 1, and no flow a 0. So, we start off with 0, no flow. Then we get a flow, which represents 1. That's as much as we can do with that part of the circuit: 0 or 1, on or off. Instead of base 10, the decimal system, this is **base 2**, the **binary system**. In the binary system, one digit represents one unit of information: one **bi**nary digi**t**, or **bit**.

When we join two parts of the circuit together, things get more interesting. Look at them both in a row, when they are both off, the counter reads 00. Then one comes on, so we get 01. Then what? Well, humans get to 9 and have to carry one to the next column, but computers only get to 1. The next number, number two, is represented as 10. Then 11. And we need some more of our circuit. Number four is 100, 5 is 101, and so ad infinitum. If we got used to it, and we used the binary system naturally, we could count up to 1023 on our fingers.

This may sound like an abnormal way to count, but even stranger, counting mechanisms are all around us. As I write this, it's 7:59pm. In one minute, it'll be 8:00pm, which seems unremarkable. But that's a base 60 system. In fact, it's worse than that – time doesn't stay in base 60, because hours carry at 24 instead of 60. Anyone who's used the Imperial measurement system, a Chinese abacus, or pounds, shillings, and pence knows the full horror of mixed base systems, which are far more complicated than what we're dealing with here.

As well as binary, there are two more important sequences we need to know about when talking to computers. We don't often get to deal with binary directly, but the following two sequences have a logical relationship to base 2 counting. The first is **octal, base 8**.

Eight is an important number in computing. Bits are organized in groups of eight to form **byte**s, giving you the range of 0 to 255 that we saw earlier with ASCII. Each ASCII character can be represented by one byte. As we said in the paragraph before, octal is one way of counting bits – it has, however, fallen out of fashion these days. Octal numbers all start with 0, (that's a zero, not an oh) so we know they're octal and proceed as you'd expect: 00, 01, 02, 03, 04, 05, 06, 07, carry one, 010, 011, 012…017, carry one, 020 and so on. Perl recognizes octal numbers if you're certain to put that zero in front, like this:

```
print 01101;
```

prints out the decimal number:

```
577
```

The second is called the **hexadecimal** system, as mentioned above. Of course, programmers are lazy, so they just call it **hex**. (They like the wizard image.)

Decimal is base 10, and hexagons have six sides, so this system is base 16. As you might have guessed from the number 1F18 above, digits above 9 are represented by letters, so A is 10, B is 11, and so on, all the way through to F which is 15. We then carry one and start with 10 (which, in decimal, is 16) all the way up through 19, 1A, 1B, 1C, 1D, 1E, 1F, and carry one again to get 20 (which in decimal is 32). The magic number 255, the maximum number we can store in one byte, is FF. Two bytes next to each other can get you up to FFFF, better known as 65535. We met 65535 as the highest number in the Unicode character set, and you guessed it, a Unicode character can be stored as a pair of bytes.

To get perl to recognize hex, place `0x` in front of the digits so that:

```
print 0xBEEF;
```

gives the answer:

```
48879
```

The Perl Debugger

One thing you'll notice about programming is that you'll make mistakes; mistakes in programs are called **bugs**. Bugs are almost entirely unavoidable, and creating bugs does not mean you're a bad programmer. Windows 2000 allegedly shipped with 65,000 bugs (but then that's a special case) and even the greatest programmers in the world have problems with bugs. Donald Knuth's typesetting software TeX has been in use for 18 years, and bugs were still found until a couple of years ago.

While we will be showing you ways to avoid getting bugs in your program, Perl provides you with a tool to help find and trace the causes of bugs. Naturally, any tool for getting rid of bugs in your program is called a 'debugger'. Mundanely enough, the corresponding tool for putting bugs into your program is called a 'programmer'.

Summary

We've started on the road to programming in Perl, and programming in general. We've seen our first piece of Perl code, and hopefully, you've had it running. If you haven't, please do get through it and all the examples to come; trying everything yourself is the best way to learn.

Programming is basically telling a computer what to do in a language it comprehends. It's about breaking down problems or ideas into byte-sized chunks (as it were) and examines the task at hand in order to communicate them clearly to the machine.

Thankfully, Perl is a language that allows us a certain degree of freedom in our expression, and so long as we work within the bounds of the language, it won't enforce any particular method of expression on us. Of course, it may judge what we're saying to be wrong, because we're not speaking the language correctly, and that's how the majority of bugs are born. Generally though, if a program does what we want, that's enough - There's More Than One Way To Do It.

We've also seen a few ways of making it easy for ourselves to spot potential problems, and we know there are tools that can help us if we need it. We have examined a little bit of what goes on inside a computer, how it sees numbers, and how it sees characters, as well as what it does to our programs when and as it executes them.

I'm now going to ask you to write a simple program for yourself, nothing strenuous, and nothing harder than we've already seen. But it's important that you take that psychological step into programming right now.

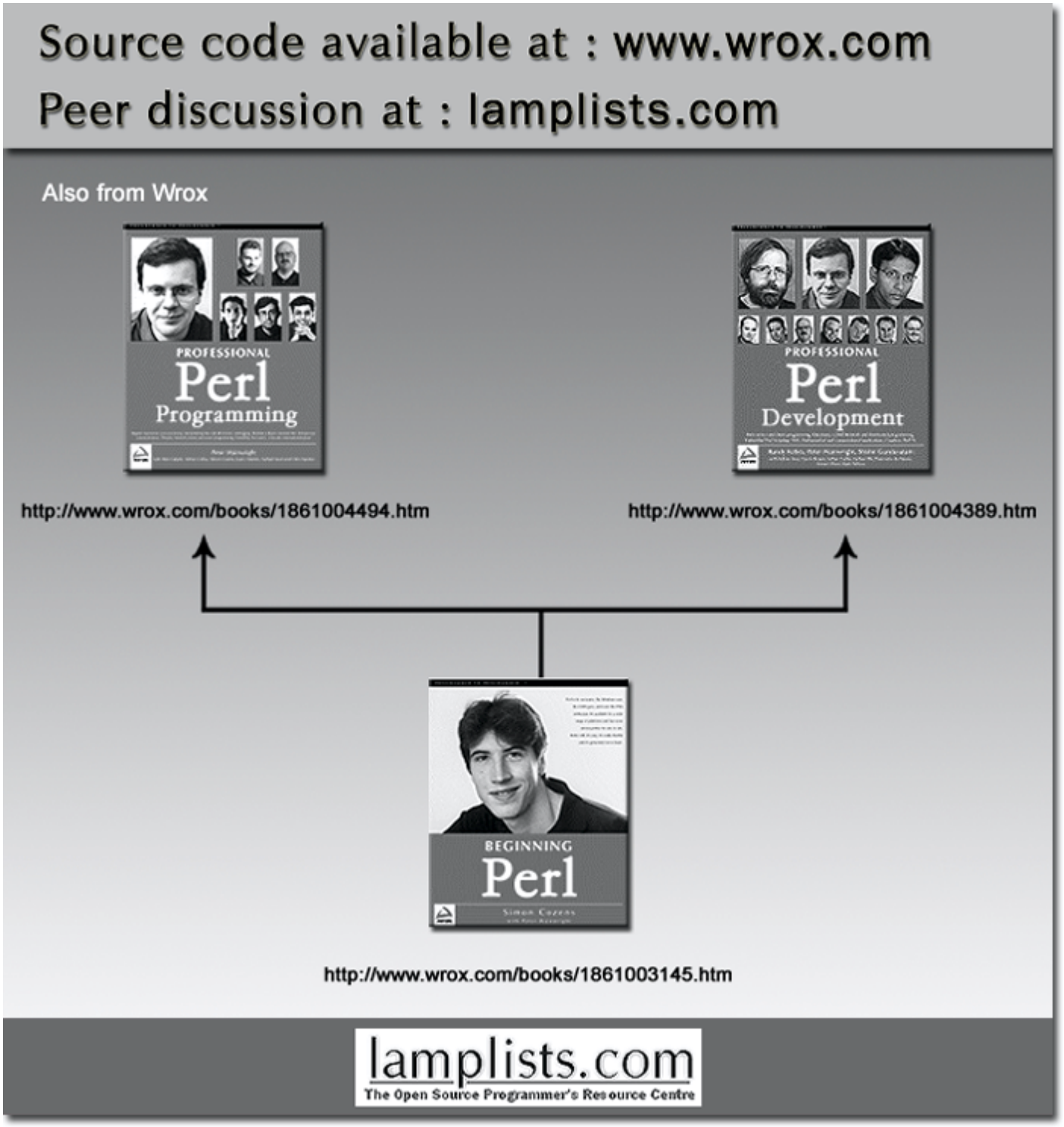
Exercises

1. Look through the documentation installed with your Perl distribution.

2. Create a program newline.plx containing print "Hi Mum.\nThis is my second program. \n". Run this and then to replace \n with a space or an Enter and compare the results.

3. Download the code for this book from the wrox website at http://www.wrox.com.

4. Have a look around the Perl homepage at www.perl.com and at our Beginning_Perl mailing list at http://p2p.wrox.com.



2

Working with Simple Values

The essence of programming is computation – we want the computer to do some work with the input (the data we give it). Very rarely do we write programs that tell us something we already know. Even more rarely do we write programs that do nothing interesting with our data at all. So, if we're going to write programs that do more than say "hello" to us, we're going to need to know how to perform computations, or operations, on our data. The things that perform these operations are called **operators**, and the second part of this chapter will be dedicated to looking at some common operators in Perl.

Variables are another key topic we'll introduce in this chapter. Variables give us somewhere to store a value while we're doing calculations on it, allowing us to do long computations with intermediary stages. As their name suggests, they also allow us to change their contents at will. Variables are the basis for all serious programming, so we need to meet them sooner rather than later.

Finally in the chapter, we'll see one way of getting data from the user, and we'll use that to build our first 'useful' program.

Types of Data

A lot of programming jargon is about familiar words in an unfamiliar context. We've already seen a string, which was a series of characters. We could also describe that string as a **scalar literal constant**. What does that mean?

It's a **literal**, because it's something that means what it says, as opposed to a variable. A variable is more like a pigeonhole for data; the important thing is to look inside it and see what it contains. A variable, such as `$fish`, is probably not going to stand for the word 'fish' preceded by a dollar sign, it's more likely to contain 'trout', 42, or –10. A literal, on the other hand, such as the string `"Hello, world"` is the piece of paper that goes into a pigeonhole – it doesn't stand for something else. It represents literally those twelve characters.

It's also a **constant**, because it can't change. Variables, as their name implies, may change their contents, but constants are written into the text of your program once and for all, and the program can't change that. Another way of expressing this is that the data is **hard-coded** into the program. We will see later how it's almost always better to avoid hard-coding information.

By calling a variable a **scalar**, we're describing the type of data it contains. If you remember your math (and even if you don't) a **scalar** is a plain, simple, one-dimensional value. In math, the word is used to distinguish it from a vector, which is expressed as several numbers. Velocity, for example, has a pair of co-ordinates (speed and direction), and so must be a vector. In Perl, a scalar is the fundamental, basic unit of data of which there are two kinds – numbers and strings.

> *We use the term 'scalar' to distinguish it from aggregates, like* ***lists*** *or* ***hashes****, which are single entities made up of several scalars. We'll look at what we can do with these two data types and how to manipulate them in the next chapter.*

Numbers

Numbers are…well, they're numbers. Now there are two types of number that we're interested in as Perl programmers: integers and floating-point numbers. The latter we'll come to in a minute, but let's work with integers right now. **Integers** are whole numbers with no numbers after the decimal point like 42, -1, or 10. The following program prints a couple of integer literals in Perl.

```
#!/usr/bin/perl
#number1.plx
use warnings;
print 25, -4;
```

```
> perl number1.plx
25-4>
```

Well, that's what you see, but it's not exactly what we want. Our program has a bug. Fortunately, this is a pretty easy bug to understand and fix. First, we didn't tell perl to separate the numbers with a space, and second, we didn't tell it to insert a new line at the end. Let's change the program so it does that:

```
#!/usr/bin/perl
#number2.plx
use warnings;
print 25, " ", -4, "\n";
```

This will do what we were thinking of:

```
> perl number2.plx
25 -4
>
```

For very large integers, we might find it easier to split the number up. So when we write out ten million, we're likely to split up the thousands with commas, like this: 10,000,000. We can also do this in Perl, but with an underscore (_) instead of a comma. Note that this is only to help us make our code clearer – perl ignores it. Change your program to look like the following, and then save it.

```
#!/usr/bin/perl
#number3.plx
use warnings;
print 25_000_000, " ", -4, "\n";
```

Notice, that those underscores don't appear in the output:

```
> perl number3.plx
25000000 –4
>
```

As well as integers, there's another class of number – **floating-point numbers**. These contain everything else, like 0.5, -0.01333, and 1.1. Now, floating-point numbers have a big problem. Take what happens when you divide 1 by 7, you get a number that starts off 0.14285714285714… and keeps going. It's an infinite sequence, and you can't possibly write out all of it. You have to stop somewhere, and this means you lose accuracy.

We've seen that computers represent numbers internally in binary form, and this is true for fractional numbers too. 0.1 is equivalent to a 1/2, or what we would call 0.5 in decimal; 0.01 is 1/4, or 0.25 in decimal; 0.001 is 1/8, and so on. The upshot of all this is that numbers we can express perfectly accurately in decimal, such as one-fifth (0.2), cannot be accurately expressed by a computer, as its binary representation is 0.001100110011…. Because of this, you need to be careful when working with floating-point numbers. While perl does try to provide sensible looking answers whenever possible, you may get the odd occasion where you end up with a number like 24.999999999999, instead of 25, which is what you should see. There's an old programming adage that goes 'don't compare floating-point numbers solely for equality' – allow for a bit of 'fudge factor'. We'll see how this is done when we get to comparisons.

The other potential inaccuracy is that Perl, by default, only uses a set number of bits to store each of your numbers in. To see how much storage your computer allows, change your program again to this:

```
#!/usr/bin/perl
#number4.plx
use warnings;
print 25_000_000, " ", 3.14159265358979323846264338327 9, "\n";
```

Here's what happens on my computer:

```
> perl number4.plx
25000000 3.14159265358979
>
```

As you can see, what we put in is only good to 14 decimal places. Some computers may have more than that, but those that don't may emulate arbitrarily long storage with the core `Math::BigFloat` module. Integers are also limited by the computer's storage, the maximum available size for storing a single integer is typically 32 bits, or 4294967295, and everything above that gets stored as a floating-point number. There's also a `Math::BigInt` module, included in the standard Perl distribution, for allowing larger integers than this. We will see more of modules in Chapter 10.

Binary, Hexadecimal, and Octal Numbers

As we saw in the previous chapter, we can express numbers as binary, hexadecimal, or octal numbers in our programs. We can mix the various representations in our program at will.

Try it out – Number systems

Here we'll create a simple program to demonstrate how we use the various number systems. Type in the following code, and save it as `goodnums.plx`:

```
#!/usr/bin/perl
#goodnums.plx
use warnings;
print 255,         "\n";
print 0377,        "\n";
print 0b11111111, "\n";
print 0xFF,        "\n";
```

All of these are representations of the number 255, and accordingly, we get the following output:

```
> perl goodnums.plx
255
255
255
255
>
```

How It Works

When perl reads your program, it reads and understands numbers in any of the allowed number systems: `0` for octal, `0b` for binary, and `0x` for hex.

What happens, you might ask, if you specify a number in the wrong system? Well, let's try it out. Edit `goodnums.plx` to give you a new program `badnums.plx` that looks like this:

```
#!/usr/bin/perl
#badnums.plx
use warnings;
print 255,          "\n";
print 0378,        "\n";
print 0b11111112, "\n";
print 0xFG,        "\n";
```

Since octal digits only run from 0 to 7, binary digits from 0 to 1, and hex digits from 0 to F, none of the last three lines make any sense. Let's see what perl makes of it:

```
> perl badnums.plx
Illegal octal digit '8' at badnums.plx line 5, at end of line
Illegal binary digit '2' at badnums.plx line 6, at end of line
Bareword found where operator expected at badnums.plx line 7, near "0xFG"
        (Missing operator before G?)
syntax error at badnums.plx line 7, near "0xFG"
Execution of badnums.plx aborted due to compilation errors.
>
```

Now, let's match those errors up with the relevant lines:

```
Illegal octal digit '8' at badnums.plx line 5, at end of line
```

And line 5 is:

```
print 0378,       "\n";
```

As you can see, perl thought it was dealing with an octal number, but then along came an 8, which stopped it from making sense, so perl quite rightly complained. The same thing happened on the next line:

```
Illegal binary digit '2' at badnums.plx line 6, at end of line
```

And line 4 is:

```
print 0b11111112, "\n";
```

The problem with the next line is even bigger:

```
Bareword found where operator expected at badnums.plx line 7, near "0xFG"
      (Missing operator before G?)
syntax error at badnums.plx line 7, near "0xFG"
```

'What's a bareword?' I hear you asking. A **bareword** is a series of characters outside of a string that perl doesn't recognize. The word could mean a number of things, and Perl can usually understand what you mean. In this case, the bareword was 'G': perl had understood `0xF`, but couldn't see how the 'G' fitted in. We might have wanted an operator do something with it, but there was no operator there. In the end, perl gave us a 'syntax error', which is the equivalent of it giving up in disgust saying, 'How do you expect me to understand this?'

Strings

The other type of scalar available to us is the string, and we've already seen a few examples of them. In the last chapter, we met the string `"Hello, world\n"` and I mentioned that a string was a series of characters surrounded by some sort of quotation marks. Strings can contain ASCII (or Unicode) data and escape sequences such as the `\n` of our example, and there is no maximum length restriction on a string imposed by Perl. Practically speaking, there is a limit imposed by the amount of memory in your computer, but it's quite hard to hit.

Single- vs Double-Quoted Strings

The quotation marks you choose for your string are significant. So far we've only seen **double-quoted** strings, like this: `"Hello, world\n"`. There is another type of string – one which has been **single-quoted**. Predictably, they are surrounded by single quotes: `''`. The important difference is that no processing is done within single quoted strings, except on `\\` and `\'` . We'll also see later that variable names inside double-quoted strings are replaced by their contents, whereas single-quoted strings treat them as ordinary text. We call both these types of processing **interpolation**, and say that single-quoted strings are not interpolated.

Consider the following program, bearing in mind that `\t` is the escape sequence that represents a tab.

```
#!/usr/bin/perl
#quotes.plx
use warnings;
print'\tThis is a single quoted string.\n';
print "\tThis is a double quoted string.\n";
```

The double-quoted string will have its escape sequences processed, and the single-quoted string will not. The output we get is:

```
> perl quotes.plx
\tThis is a single quoted string.\n          This is a double quoted string.
>
```

What do we do if we want to have a backslash in a string? This is a common concern for Windows users, as a Windows path looks something like this: `C:\WINNT\Profiles\...`. In a double-quoted string, a backslash will start an escape sequence, which is not what we want it to do.

Well, there is, of course, more than one way to do it. We can either use a single-quoted string, as above, or we can **escape** the backslash. One principle that we'll see often in Perl, and especially when we get to regular expressions, is that we can use a backslash to turn off any special effect a character may have. For example, a full stop in a regular expression denotes 'any character'. If you escape the full stop by placing a backslash in front of it, like so `\.` you get the ordinary meaning of 'a full stop'. This operation is called escaping, or more commonly, **backwhacking**.

In this case, we want to turn off the special effect a backslash has, and so we escape it:

```
#!/usr/bin/perl
#quotes2.plx
use warnings;
print"C:\\WINNT\\Profiles\\\n";
print 'C:\WINNT\Profiles\ ', "\n";
```

This prints:

```
> perl quotes2.plx
C:\WINNT\Profiles\
C:\WINNT\Profiles\
>
```

Aha! Some of you may have got this message instead:

```
Can't find string terminator " ' " anywhere before EOF at quotes2.plx line 5.
```

The reason for this is that you have probably left out the space character in line 5 before the second single quote. Remember that `\'` tells perl to escape the single quote, and so it merrily heads off to look for the next quote, which of course is not there. Try this program to see how perl treats these special cases:

```
#!/usr/bin/perl
#aside1.plx
use warnings;
print 'ex\\ er\\' , ' ci\' se\'' , "\n";
```

The output you get this time is:

```
> perl aside1.plx
ex\ er\ ci' se'
>
```

Can you see how perl did this? Well, we simply escaped the backslashes and single quotes. It will help you to sort out what is happening if you look at each element individually. Remember, there are three arguments in this example. Don't let all the quotes confuse you.

Actually, there's an altogether sneakier way of doing it. Internally, Windows allows you to separate paths in the Unix style with a forward slash, instead of a backslash. If you're referring to directories in Perl on Windows, you may find it easier to say `C:/WINNT/Profiles/` instead. This allows you to get the variable interpolation of double-quoted strings without the 'Leaning Toothpick Syndrome' of multiple backslashes.

So much for backslashes, what about quotation marks? The trick is making sure perl knows where the end of the string is. Naturally, there's no problem with putting single quotes inside a double-quoted string, or vice versa:

```
#!/usr/bin/perl
#quotes3.plx
use warnings;
print"It's as easy as that.\n";
print '"Stop," he cried.', "\n";
```

This will produce the quotation marks in the right places:

```
> perl quotes3.plx
It's as easy as that.
"Stop," he cried.
>
```

The trick comes when we want to have double quotes inside a double-quoted string or single quotes inside a single-quoted string. As you might have guessed, though, the solution is to escape the quotes on the inside. Suppose we want to print out the following quote, including both sets of quotation marks:

```
'"Hi," said Jack. "Have you read Slashdot today?"'
```

Here's a way of doing it with a double-quoted string:

```
#!/usr/bin/perl
#quotes4.plx
use warnings;
print"'\"Hi,\" said Jack. \"Have you read Slashdot today?\"'\n";
```

Now see if you can modify this to make it a single-quoted string – don't forget that `\n` needs to go in separate double quotes to make it interpolate.

Alternative Delimiters

Of course, it would be nicer if you could select a completely different set of quotes so that there would be no ambiguity and no need to escape any quotes inside the text. The first operators we're going to meet are the **quote-like operators** that do this for us. They're written as `q//` and `qq//`, the first acting like a single-quoted string and the second, like a double-quoted string. Now instead of the above, we can write:

```
#!/usr/bin/perl
#quotes5.plx
use warnings;
print qq/'"Hi," said Jack. "Have you read Slashdot today?"'\n/;
```

That's all very well, of course, until we want a `/` in the string. Suppose we want to replace 'Slashdot' with '/.' – now we're back where we started, having to escape things again. Thankfully, Perl allows us to choose our own delimiters so we don't have to stick with `//`. Any non-alphanumeric (that is, non-alphabetic and non-numeric) character can be used as a delimiter, provided it's the same on both sides of the text. Furthermore, you can use `{}`, `[]`, `()` and `<>` as left and right delimiters. Here are a few ways of doing the above, all of which have the same effect:

```
#!/usr/bin/perl
#quotes6.plx
use warnings;
print qq|'"Hi," said Jack. "Have you read /. today?"'\n|;
print qq#'"Hi," said Jack. "Have you read /. today?"'\n#;
print qq('"Hi," said Jack. "Have you read /. today?"'\n);
print qq<'"Hi," said Jack. "Have you read /. today?"'\n>;
```

We'll see more of these alternative delimiters when we start working with regular expressions.

Here-Documents

There's one final way of specifying a string – by using a **here-document**. This idea was taken from the Unix shell, and works on any platform. Effectively, it means that you can write a large amount of text within your program, and it will be treated as a string provided it is identified correctly. Here's an example.

```
#!/usr/bin/perl
#heredoc.plx
use warnings;
print<<EOF;

This is a here-document. It starts on the line after the two arrows,
and it ends when the text following the arrows is found at the beginning
of a line, like this:

EOF
```

A here-document must start with << and then a label. The label can be anything you choose, but is traditionally `EOF` (End Of File). The label must follow directly after the arrows with no spaces between, unless the same number of spaces precedes the end marker. It ends when the label is found at the beginning of
a line. In our case, the semicolon does not form part of the label, because it marks the end of the `print` statement.

By default, a here-document works like a double-quoted string. In order for it to work like a single-quoted string, surround the label in single quotes. This will become important when variable interpolation comes into play, as we'll see later on.

Converting between Numbers and Strings

The perl interpreter treats numbers and strings on an equal footing, and where necessary, perl converts between strings, integers, and floating-point numbers behind the scenes. This means that you don't have to worry about making the conversions yourself, like you do in other languages. If you have a string literal `"0.25"`, and multiply it by four, perl treats it as a number and gives you the expected answer, `1`.

There is, however, one area where this doesn't take place. Octal, hex, and binary numbers in string literals or strings stored in variables don't get converted automatically:

```
#!/usr/bin/perl
#octhex1.plx
use warnings;
print"0x30\n";
print "030\n";
```

gives you

```
> perl octhex1.plx
0x30
030
>
```

If you ever find yourself with a string containing a hex or octal value that you need to convert into a number, you can use the `hex()` or `oct()` functions accordingly:

```
#!/usr/bin/perl
#octhex2.plx
use warnings;
print hex("0x30"), "\n";
print oct("030"), "\n";
```

This will now produce the expected answers, 48 and 24. Note that for `hex()` or `oct()`, the prefix `0x` or `0`, respectively, is not required. If you know that what you have is definitely supposed to be a hex or oct number, then `hex(30)` and `oct(30)` will produce the results above. As you can see from that, the string `"30"` and the number `30` are treated as the same.

Furthermore, these functions will stop reading when they get to a digit that doesn't make sense in that number system:

```
#!/usr/bin/perl
#octhex3.plx
use warnings;
print hex("FFG"), "\n";
print oct("178"), "\n";
```

These will stop at `FF` and `17`, respectively, and convert to 255 and 15.

What about binary numbers? Well, there's no corresponding `bin()` function, but there is actually a little trick here. If you have the correct prefix in place for any of the number systems, (`0`, `0b`, or `0x`) you can use `oct()` to convert it to decimal. For example `print oct("0b11010")` prints 26.

Operators

Now we know how to specify our strings and numbers, let's see what we can do with them. The majority of the things we'll be looking at here are numeric operators (operators that act on and produce numbers) like plus and minus, which take two numbers as 'arguments' and add or subtract them. There aren't as many string operators, but there are plenty of string functions. Perl doesn't draw a very strong distinction between functions and operators, but the main difference between the two is that operators tend to go in the middle of their arguments – for example: 2 + 2. Functions go before their arguments and have them separated by commas. Both of them take arguments, do something with them, and produce a new value. We generally say they **return** a value. Let's take a look:

Numeric Operators

The numeric operators take at least one number as an argument and return another number. Of course, because perl automatically converts between strings and numbers, the arguments may appear as string literals or come from strings in variables. We'll group these operators into three types: ordinary arithmetic operators, bitwise operators, and logic operators.

Arithmetic Operators

The arithmetic operators are those that deal with basic mathematics like adding, subtracting, multiplying, dividing, and so on. To add two numbers together, we would write something like this:

```
#!/usr/bin/perl
#arithop1.plx
use warnings;
print 69 + 118;
```

And, of course, we would see the answer 187. Subtracting numbers is easy, too, and we can subtract at the same time:

```
#!/usr/bin/perl
#arithop2.plx
use warnings;
print "21 from 25 is: ", 25 - 21, "\n";
print "4 + 13 - 7 is: ", 4 + 13 - 7, "\n";
```

```
>perl arithop2.plx
21 from 25 is: 4
4 + 13 - 7 is: 10
>
```

Our next set of operators (multiplying and dividing) is where it gets interesting. We use the `*` and `/` operators to multiply and divide, respectively.

```
#!/usr/bin/perl
#arithop3.plx
use warnings;
print"7 times 15 is ", 7 * 15, "\n";
print "249 over 3 is ", 249 / 3, "\n";
```

The fun comes when you want to multiply something and then add something, or add then divide. Here's an example of the problem:

```
#!/usr/bin/perl
#arithop4.plx
use warnings;
print 3 + 7 * 15, "\n";
```

Now this could mean one of two things: either perl must add the three and the seven and then multiply by fifteen, or it must multiply seven and fifteen first, then add. Which does Perl do? Try it and see.

So, perl should have given you 108, as it did the multiplication first. The order in which perl performs operations is called **precedence**. Multiply and divide have a higher precedence than add and subtract, and so they get performed first. We can start to draw up a table of precedence as follows:

* /
+ -

To force perl to perform an operation of lower precedence first, we need to use brackets, like so:

```
#!/usr/bin/perl
#arithop5.plx
use warnings;
print(3 + 7) * 15;
```

Unfortunately, if you run that, you'll get a warning and 10 is returned. What happened? The problem is that `print` is itself an operator as well, and the precedence of operators like `print` is highest of all.

`print` as an operator takes a list of arguments and performs an operation (printing them to the screen). It returns a 1 if it succeeds or no value if it does not. Perl calculated 3 plus 7, printed the result, and then multiplied the result of the returned value (1) by 15, throwing away the final result of 15.

To get what we actually want then, we need another set of brackets:

```
#!/usr/bin/perl
#arithop6.plx
use warnings;
print((3 + 7) * 15);
```

This now gives us the correct answer, 150, and we can put another entry in our table of precedence:

List operators
* /
+ -

Next we have the exponentiation operator, `**`, which simply raises one number to the power of another – squaring, cubing, and so on. Here's an example of some exponentiation:

```
#!/usr/bin/perl
#arithop7.plx
use warnings;
print 2**4, " ", 3**5, " ", -2**4, "\n";
```

That's 2*2*2*2, 3*3*3*3*3, and -2*-2*-2*-2. Or is it?

The output we get is:

```
>perl arithop7.plx
16 243 -16
>
```

Hmm, the first two look OK, but the last one's a bit wrong. –2 to the 4th power should be positive. Again, it's a precedence issue. Turning a number into a negative number requires an operator, the 'unary minus' operator. It's called 'unary' because unlike the ordinary minus operator, it only takes one argument. Although unary minus has a higher precedence than times and divide, it has a lower precedence than exponentiation. What's actually happening, then, is `-(2**4)` instead of `(-2)**4`. Let's put these two operators in the table as well:

List operators
**
Unary minus
* /
+ -

The last arithmetic operator is `%`, the remainder, or 'modulo' operator. This calculates the remainder when one number divides another. For example, six divides into fifteen twice, with a remainder of three, as our next program will confirm:

```
#!/usr/bin/perl
#arithop8.plx
use warnings;
print"15 divided by 6 is exactly ", 15 / 6, "\n";
print "That's a remainder of ", 15 % 6, "\n";
```

```
>perl arithop8.plx
15 divided by 6 is exactly 2.5
That's a remainder of 3
>
```

The modulo operator has the same precedence as multiply and divide.

Bitwise Operators

Those operators worked on numbers in the way we think of them. However, as we already know, computers don't see numbers the same as we do; they see them as a string of bits. These next few operators perform operations on numbers one bit at a time – that's why we call them bitwise. These aren't used quite so much in Perl as in other languages, but we'll see them when dealing with things like low-level file access.

First, let's have a look at the kind of numbers we're going to use in this section, just so we get used to them:

0 in binary is 0, but let's write it as 8 bits:	00000000
51 in binary is	00110011
85 in binary is	01010101
170 in binary is	10101010
204 in binary is	11001100
255 in binary is	11111111

Does it surprise you that 10101010 (170) is twice as much as 01010101 (85)? It shouldn't, when we multiply a number by 10 in base 10, all we do is slap a zero on the end, so 21 becomes 210. Similarly, to multiply a number by 2 in base 2, we do exactly the same.

Bitwise operators work from right to left. The rightmost bit is called the 'least significant bit', and the leftmost is called the 'most significant bit'.

The 'and' Operator

The easiest operator to fathom is called the '`and`' operator and is written `&`. This compares pairs of bits as follows:

1 and 1 gives 1

1 and 0 gives 0

0 and 1 gives 0

0 and 0 gives 0

For example, `51 & 85` looks like this:

```
51   00110011
85   01010101
17   00010001
```

Sure enough, if we ask Perl:

```
#!/usr/bin/perl
#bitop1.plx
use warnings;
print"51 ANDed with 85 gives us", 51 & 85, "\n";
```

It'll tell us the answer is 17. Notice that since we're comparing one pair of bits at a time, it doesn't really matter which way around the arguments go, `51 & 85` is exactly the same as `85 & 51`. Operators with this property are called **associative** operators.

Here's another example, look at the bits, and see what you get:

```
51    00110011
170   10101010
34    00100010
```

The 'or' Operator

As well as checking whether the first **and** the second bits are 1, we can check whether one **or** another is 1. The '**or**' operator in Perl is |, and this is how we would calculate `204 | 85`

```
204   11001100
85    01010101
221   11011101
```

Now we produce zeros only if both the bits are zero, if either or both are one, we produce a one. As a quick rule of thumb, `X & Y` will always be smaller or equal to the smallest value of `X` and `Y`, and `X | Y` will be bigger than or equal to the largest value of `X` or `Y`.

The 'exclusive or' Operator

What if you really want to know if one or the other, but not both, are set to one? For this, you need the '**exclusive or**' operator, written as the ^ operator:

```
204   11001100
170   10101010
102   01100110
```

The 'not' Operator

Finally, you can flip the number completely, and replace all the ones by zeros and vice versa. This is done with the '**not**', or ~ operator:

```
85    01010101
170   10101010
```

Let's see, however, what happens when we try this in Perl:

```
#!/usr/bin/perl
#bitop2.plx
use warnings;
print"NOT 85 is", ~85, "\n";
```

On my computer, I get:

```
NOT 85 is 4294967210
>
```

Your answer might be different, and I'll explain why in a second.

Why is it so big? Well, let's look at that number in binary to see if we can find a clue as to what's going on:

4294697210 11111111111111111111111110101010

Aha! The last part is right, but it's a lot wider than we're used to. That's because in the examples, I've only used 8 bits across, whereas my computer stores integers as 32 bits across, what's actually happened is this:

85 00000000000000000000000001010101
4294697210 11111111111111111111111110101010

If you get a much bigger number, it's because your computer represents numbers internally with 64 bits instead of 32, and Perl has been configured to take advantage of this.

Truth and Falsehood

"What is truth?" If we had asked that of a Perl programmer, we could be sure that he would have replied something like this: "Truth is anything that is not zero, an empty string, an undefined value, or an empty list."

Later, we will want to perform actions based on whether something is true or false, for example if one number is bigger than another, or, unless a problem has occurred, or, while there is data left to examine. We will use **comparison operators** to evaluate whether these things are true or false so that we can make decisions based on them.

It's customary to represent false as 0 and true as 1. This allows us to use operators very similar to those bitwise operators we've just met to combine our comparisons, to say 'if this *or* this is true', 'if this is *not* true', and so on. The idea of combining values that represent truth and falsehood is called **Boolean logic**, after George Boole, who invented the concept in 1847, and we call the operators that do the combining '**Boolean operators**'.

Comparing Numbers for Equality

The first simple comparison operator is ==. Two equals signs tells perl to 'return true if the two numeric arguments are equal.' If they're not equal, return false. Boolean values of truth and falsehood aren't very exciting to look at, but let's see them anyway:

```
#!/usr/bin/perl
#bool1.plx
use warnings;
print"Is two equal to four? ",          2 == 4, "\n";
print "OK, then, is six equal to six? ", 6 == 6, "\n";
```

This will produce:

```
>perl bool1.plx
Is two equal to four?
OK, then, is six equal to six? 1
>
```

The second line is definitely true, and as we'd expect, we get a one back from the operator. But what happened in the first line? Well, there's a special value in Perl that is conspicuous by its absence. Can you guess what it is? You might have noticed before that I mentioned "… an undefined value or an empty list." This next paragraph will help you work it out.

The undefined value isn't simply a string with nothing in it – it's nothing at all. In a very Zen-like way, a string with no characters **is** still a string. The undefined value isn't zero either, although it gets converted to zero if you use it as a number in the same way that an empty string does. The undefined value represents nothing, empty, void.

The obvious counterpart to test whether things are equal is to test whether they're not equal. The way we do this is with the `!=` operator. Note that there's only one `=` this time. We'll find out later why there had to be two before.

```
#!/usr/bin/perl
#bool2.plx
use warnings;
print"So, two isn't equal to four? ", 2 != 4, "\n";
```

```
>perl bool2.plx
So, two isn't equal to four? 1
>
```

There you have it – irrefutable proof that two actually isn't four. Good.

Comparing Numbers for Inequality

So much for equality, let's check if one thing is bigger than another. Just like in mathematics, we use the greater-than and less-than signs to do this: < and >.

```
#!/usr/bin/perl
#bool3.plx
use warnings;
print"Five is more than six? ",      5 >  6, "\n";
print "Seven is less than sixteen? ", 7 < 16, "\n";
print "Two is equal to two? ",        2 == 2, "\n";
print "One is more than one? ",       1 >  1, "\n";
print "Six is not equal to seven? ",  6 != 7, "\n";
```

The results should hopefully not be very new to you:

```
>perl bool3.plx
Five is more than six?
Seven is less than sixteen? 1
Two is equal to two? 1
One is more than one?
Six is not equal to seven? 1
>
```

Let's have a look at one last pair of comparisons. We can check greater-than-or-equal-to and less-than-or-equal-to with the >= and <= operators, respectively.

```
#!/usr/bin/perl
#bool4.plx
use warnings;
print"Seven is less than or equal to sixteen? ", 7 <= 16, "\n";
print "Two is more than or equal to two? ", 2 >= 2,  "\n";
```

As expected, perl faithfully prints out:

```
>perl bool4.plx
Seven is less than or equal to sixteen? 1
Two is more than or equal to two? 1
>
```

There's also a special operator that isn't really a Boolean comparison because it doesn't give us a true-or-false value. Instead it returns 0 if the two are equal, -1 if the right hand side is bigger, and 1 if the left-hand side is bigger. It is denoted by `<=>`. Think of it as a balance, pointing towards the lower number:

```
#!/usr/bin/perl
#bool5.plx
use warnings;
print"Compare six and nine? ",   6 <=> 9, "\n";
print "Compare seven and seven? ",7 <=> 7, "\n";
print "Compare eight and four? ", 8 <=> 4, "\n";
```

Gives us:

```
>perl bool5.plx
Compare six and nine? -1
Compare seven and seven? 0
Compare eight and four? 1
>
```

We'll see this in more detail when we look at sorting things, where we have to know whether something goes before, after, or in the same place as something else.

Boolean Operators

As well as being able to evaluate the truth and falsehood of some statements, we can also combine such statements. For example, we may want to do something if one number is bigger than another and another two numbers are the same. The combining is done in a very similar manner to the bitwise operators we saw earlier. We can ask if one value **and** another value are both true, or if one value **or** another value are true, and so on.

The operators even resemble the bitwise operators. To ask if both truth-values are true, we would use `&&` instead of `&`.

> **In many cases, `&` and the other bitwise operators will work just fine, if you are sure that the values are either one or zero. But as we know, truth is anything that is not zero, an empty string, an undefined value, or an empty list, rather than just one or zero. For example, `-2` is a true value. However, `~-2` is also a true value. When testing truths, always use the Boolean rather than the bitwise operators.**

So, to test whether six is more than three **and** twelve is more than four, we can write:

```
6 > 3 && 12 > 4
```

To test if nine is more than seven **or** eight is less than six, we use the doubled form of the | operator, ||:

```
9 > 7 || 6 > 8
```

To negate the sense of a test, however, use the slightly different operator !. This has a higher precedence than the comparison operators, so use brackets. For example, this tests whether two is not more than three,

```
!(2>3)
```

while this one tests whether !2 is more than three:

```
!2>3
```

2 is a true value. !2 is therefore a false value, the undefined value, which gets converted to zero when we do a numeric comparison. We're actually testing if zero is more than three, which has the opposite effect to what we wanted.

Instead of those forms, &&, ||, and !, we can also use the slightly easier-to-read versions, and, or, and not. There's also xor, for exclusive or (one or the other but not both are true) which doesn't have a symbolic form. However, you need to be careful about precedence again:

```
#!/usr/bin/perl
#bool6.plx
use warnings;
print"Test one: ", 6 > 3 && 3 > 4, "\n";
print "Test two: ", 6 > 3 and 3 > 4, "\n";
```

This prints, somewhat surprisingly:

```
> perl bool6.plx
Test one:
Test two: 1>
```

Well, we can tell from the position of the prompt (or least Unix users can – Windows users need to be a bit more alert) that something is amiss because the second newline did not get printed. The trouble is that and has a lower precedence than &&. What has actually happened is this:

```
print ("Test two: ", 6 > 3) and 3 > 4, "\n";
```

Now, six is more than three, so that returned 1, `print` then returned one, and the rest was irrelevant. However, we can use this fact to our advantage.

Perl uses a technique called **lazy evaluation**. As soon as it knows the answer to the question, it stops working. If you ask if *x* and *y* are both true, and it finds that *x* isn't, it doesn't need to look at *y*. No matter whether *y* is true or not, it can't make them both true, so there's no point testing. Similarly, if you ask whether *x* or *y* is true, you can stop if you find that *x* is true. Whether *y* is true or not will not affect matters at all. So, we can write something like this:

```
4 >= 2 and print "Four is more than or equal to two\n";
```

If the first test is true, perl has to check if the other side is true as well, and that means printing our message. If the first test is false, there's no need to check, so the message doesn't get printed. It's a crude way of saving time if a condition is met. We won't use that for the moment, until we've seen a less crude way to do it.

String Operators

After that lot, there are surprisingly few string operators. Actually, for the moment, we're only going to look at two.

The first one is the **concatenation operator**, which glues two strings together into one. Instead of saying:

```
print "Print ", "several ", "strings ", "here", "\n";
```

we could say:

```
print "Print " . "one ". "string " . "here" . "\n";
```

As it happens, printing several strings is slightly more efficient, but there will be times you really do need to combine strings together, especially if you're putting them into variables.

What happens if we try and join a number to a string? The number is evaluated and then converted:

```
#!/usr/bin/perl
#string1.plx
use warnings;
print"Four sevens are ". 4*7 ."\n";
```

which tells us, reassuringly, that:

```
> perl string1.plx
Four sevens are 28
>
```

The other string operator is the **repetition operator**, marked with an `x`. This repeats a string a given number of times:

```
#!/usr/bin/perl
#string2.plx
use warnings;
print "GO! "x3, "\n";
```

will print:

```
> perl string2.plx
GO! GO! GO!
>
```

We can, of course, use it in conjunction with concatenation. Its precedence is higher than the concatenation operator's, as we can easily see for ourselves:

```
#!/usr/bin/perl
#string3.plx
use warnings;
print "Ba". "na"x4 ,"\n";
```

On running this, we'll get:

```
> perl string3.plx
Bananananana
>
```

In this case, the repetition is done first ("nananana") and is then concatenated with the "Ba". The precedence of the repetition operator is the same as the arithmetic operators, so if you're working out how many times to repeat something, you're going to need brackets:

```
#!/usr/bin/perl
#string4.plx
use warnings;
print"Ba". "na"x4*3 ,"\n";
print "Ba". "na"x(4*3) ,"\n";
```

Compare:

```
>perl string4.plx
Ba0
Bananananananananananananananana
>
```

Why was the first one `Ba0`? Well, think what happened. The first thing was the repetition, giving us "nananana". Then the multiplication – What's nananana times three? When perl converts a string to a number, it takes any spaces, an optional minus sign, and then as many digits as it can from the beginning of the string, and ignores everything else. Since there were no digits here, the number value of nananana was zero.

That zero was then multiplied by three, to give zero. Finally, the zero was turned back into a string to be concatenated onto the Ba.

Try it out – Converting Strings to Numbers

You can see how other strings convert to numbers by adding zero to them:

```
#!/usr/bin/perl
#str2num.plx
use warnings;
print"12 monkeys"     + 0,  "\n";
print "Eleven to fly" + 0,  "\n";
print "UB40"          + 0,  "\n";
print "-20 10"        + 0,  "\n";
print "0x30"          + 0,  "\n";
```

You get a warning for each line saying that the strings aren't 'numeric in addition (+)', but what can be converted is. Ignoring the warnings then, here's what they come out as:

```
>perl str2num.plx
12
0
0
-20
0
>
```

How It Works

Our first string, `"12 monkeys"`, did pretty well. Perl understood the 12 and stopped after that. The next one was not handled so well – English words don't get converted to numbers. Our third string was also a non-starter, as perl only looks for a number at the beginning of the string. If something other than a number is there, it's evaluated as a zero. Similarly, perl only looks for the first number in the string. Any numbers after that are discarded. Finally, perl doesn't convert binary, hex. or octal to decimal when it's stringifying a number, so you have to use the `hex()` or `oct()` functions to do that. On our last effort, perl stopped at the `x`, returning 0. If we had an octal number, such as `030`, that would be treated as the decimal number 30.

String Comparison

As well as comparing the value of numbers, we can compare the value of strings. By this, I don't mean we convert a string to a number, although if you say something like `"12" > "30"`, perl will convert to numbers for you. What I mean is, we can compare the strings alphabetically: "Bravo" comes after "Alpha" but before "Charlie", for instance.

In fact, it's more than alphabetical order: The computer is using either ASCII or Unicode internally to represent the string and has converted it to a series of numbers in the relevant sequence. This means, for example, "Fowl" comes before "fish", because a capital F has a smaller ASCII value (70) than a lower case f (102). See Appendix F for the full ASCII table.

We can find the character's value by using the `ord()` function, which tells us where in the (ASCII) **ord**er it comes. Let's see which comes first, a `#` or a `*`?

```
#!/usr/bin/perl
#ascii.plx
use warnings;
print"A # has ASCII value ", ord("#"),"\n";
print "A * has ASCII value ", ord("*"),"\n";
```

This should say:

```
>perl ascii.plx
A # has ASCII value 35
A * has ASCII value 42
>
```

I suppose if we're only concerned with one character at a time we can compare the return values of `ord()` using the `<` and `>` operators. However, when comparing entire strings, it may get a bit tedious. If the first character of each string is the same, you would move onto the next character in each string, and then the next, and so on.

Instead, there are string comparison operators that do this all for us. Whereas the comparison operators for numbers were mathematical symbols, the operators for strings are abbreviations. To test whether one string is less than another, use `lt`. 'Greater than' becomes `gt`, 'equal to' becomes `eq`, and 'not equal' becomes `ne`. There's also `ge` and `le` for 'greater than or equal to' and 'less than and equal to'. The three-way-comparison becomes `cmp`.

Here are a few examples of these:

```
#!/usr/bin/perl
#strcomp1.plx
use warnings;
print"Which came first, the chicken or the egg? ";
print "chicken" cmp "egg", "\n";
print "Are dogs greater than cats? ";
print "dog" gt "cat", "\n";
print "Is ^ less than + ? ";
print "^" lt "+", "\n";
```

And the results:

```
>perl strcomp1.plx
Which came first, the chicken or the egg? -1
Are dogs greater than cats? 1
Is ^ less than + ?
>
```

But watch this carefully:

```
#!/usr/bin/perl
#strcomp2.plx
use warnings;
print "Test one: ", "four" eq "six", "\n";
print "Test two: ", "four" == "six", "\n";
```

```
>perl strcomp2.plx
Test one:
Test two: 1
>
```

Is the second line really claiming that four is equal to six? No, but if you compare them as numbers, they get converted to numbers. `"four"` converts to 0, and `"six"` converts to 0. The 0s are equal, so our test returns true and we get a couple of warnings telling us that they were not numbers to begin with. The moral of this story is, compare strings with string comparison operators, and compare numbers with numeric comparison operators. Otherwise, your results may not be what you anticipate.

Operators To Be Seen Later

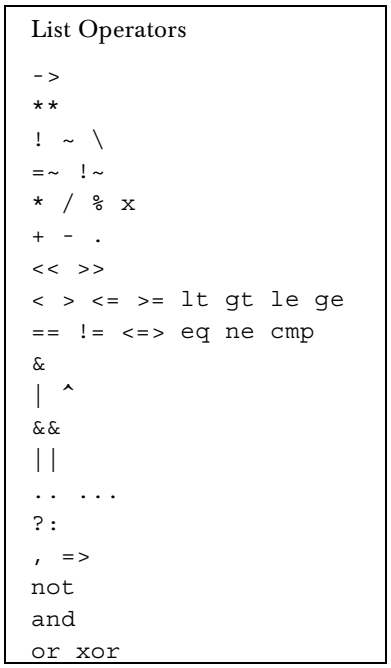
There are a few operators left that we are not going to go into in detail right now. Don't worry, we'll come across the more important ones again in time.

- The ternary hook operator looks like this: `a?b:c`. It returns `b` if `a` is true, and `c` if it is false.
- The range operators, `...` and `...`, make a range of values.
- We've seen the comma for separating arguments to functions like `print`. In fact, the comma is an operator that builds a list, and `print` works on a list of arguments. The operator `=>` works like a comma with certain additional properties.
- The `=~` and `!~` operators are used to 'apply' a regular expression to a string.
- As well as providing an escape sequence and backwhacking special characters, `\` is used to take a reference to a variable, to examine the variable itself rather than its contents.
- The `>>` and `<<` operators 'shift' a binary number right and left a given number of bits.
- `->` is hairy voodoo. We will get to it later on.

Operator Precedence

Here, finally, is a full table of precedence for all the operators we've seen so far, listed in descending order of precedence.

Remember that if you need to get things done in a different order, you will need to use brackets. Also remember that you can use brackets even when they're not strictly necessary, and you should certainly do so to help keep things readable. While perl knows full well what order to do `7+3*2/6-3+5/2&3` in, you may find it easier on yourself to spell it out, because next week you may not remember everything you have just written.



Variables

Variables! We've talked about them all the time, but what are they? As I've explained, a variable is storage for your scalars. Once you've calculated `42*7`, it's gone. If you want to know what it was, you must do the calculation again. Instead of being able to use the result as a halfway point in more complicated calculations, you've got to spell it all out in full. That's no fun.

What we need to be able to do, and what variables allow us to do, is store a scalar away and refer to it again later. As previously mentioned, there are three types of data: **scalars**, **lists**, and **hashes**. There are also three types of variable to put them in: scalar variables, arrays, and hashes. We'll look at the latter two in chapters to come and just concentrate on scalar variables for now.

A scalar variable starts with a dollar sign. Here's a simple scalar variable: `$name`. We can put certain types of data into it. Scalar variables can hold either numbers or strings and are only limited by the size of your computer's memory. To put data into our scalar, we assign the data to it, with the assignment operator =. (Incidentally, this is why numeric comparison is ==, because = was taken to mean the assignation operator.)

What we're going to do here is tell Perl that our scalar contains the string `"fred"`. Now we can get at that data by simply using the variable's name:

```
#!/usr/bin/perl
#vars1.plx
use warnings;
$name = "fred";
print "My name is ", $name, "\n";
```

Lo and behold, our computer announces to us that:

```
>perl vars1.plx
My name is fred
>
```

Now we're cut free at last from the problem of once-off data. We've got somewhere to store our data, and some way to get it back again. The next logical step is to be able to change it.

Modifying a Variable

Modifying the contents of a variable is easy, just assign something different to it. We can say:

```
#!/usr/bin/perl
#vars2.plx
use warnings;
$name = "fred";
print "My name is ",            $name, "\n";
print "It's still ",            $name, "\n";
$name = "bill";
print "Well, actually, it's ", $name, "\n";
$name = "fred";
print "No, really, it's ",      $name, "\n";
```

And watch our computer have an identity crisis:

```
>perl vars2.plx
My name is fred
It's still fred
Well, actually, it's bill
No, really, it's fred
>
```

We can also do a calculation in several stages:

```
#!/usr/bin/perl
#vars3.plx
use warnings;
$a = 6*9;
print "Six nines are ", $a, "\n";
$b = $a + 3;
print "Plus three is ", $b, "\n";
$c = $b / 3;
print "All over three is ", $c, "\n";
$d = $c + 1;
print "Add one is ", $d, "\n";
print "\nThose stages again: ", $a, " ", $b, " ", $c, " ", $d, "\n";
```

```
>perl vars3.plx
Six nines are 54
Plus three is 57
All over three is 19
Add one is 20
```

```
Those stages again: 54 57 19 20
>
```

While this works perfectly fine, it's often easier to stick with one variable and modify its value, if you don't need to know the stages you went through at the end:

```
#!/usr/bin/perl
#vars4.plx
use warnings;
$a = 6 * 9;
print "Six nines are ", $a, "\n";
$a = $a + 3;
print "Plus three is ", $a, "\n";
$a = $a / 3;
print "All over three is ", $a, "\n";
$a = $a + 1;
print "Add one is ", $a, "\n";
```

The assignment operator =, has very low precedence. This means that perl will do the calculations on the right hand side of it, including fetching the current value, before assigning the new value. To illustrate this, take a look at the sixth line of our example. perl takes the current value of `$a`, adds three to it, and then stores it back in `$a`.

Operating and Assigning at Once

Operations, like fetching a value, modifying it, or storing it, are very common, so there's a special syntax for them. Generally:

```
$a = $a <some operator> $b;
```

can be written as

```
$a <some operator>= $b;
```

For instance, we could rewrite the example above as follows:

```
#!/usr/bin/perl
#vars5.plx
use warnings;
$a = 6 * 9;
print "Six nines are ", $a, "\n";
$a += 3;
print "Plus three is ", $a, "\n";
$a /= 3;
print "All over three is ", $a, "\n";
$a += 1;
print "Add one is ", $a, "\n";
```

This works for `**=`, `*=`, `+=`, `-=`, `/=`, `.=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, `&&=` and `||=`. These all have the same precedence as the assignment operator =.

Autoincrement and Autodecrement

There are two more operators, ++ and --. They add and subtract one from the variable, but their precedence is a little strange. When they precede a variable, they act before everything else. If they come afterwards, they act after everything else. Let's examine these:

Try it out – The autoincrement and autodecrement operators

Type in and run the following code:

```
#!/usr/bin/perl
#auto1.plx
use warnings;
$a=4;
$b=10;
print "Our variables are ", $a, " and ", $b, "\n";
$b=$a++;
print "After incrementing, we have ", $a, " and ", $b, "\n";
$b=++$a*2;
print "Now, we have ", $a, " and ", $b, "\n";
$a=--$b+4;
print "Finally, we have ", $a, " and ", $b, "\n";
```

You should see the following output:

```
>perl auto1.plx
Our variables are 4 and 10
After incrementing, we have 5 and 4
Now, we have 6 and 12
Finally, we have 15 and 11
>
```

How It Works

Let's work this through a piece at a time. First we set up our variables, giving the values `4` and `10` to `$a` and `$b`, respectively. :

```
$a=4;
$b=10;
print "Our variables are ", $a, " and ", $b, "\n";
```

Now in the following line, the assignment happens before the increment. So `$b` is set to `$a`'s current value, `4` and then `$a` is autoincremented, becoming 5.

```
$b=$a++;
print "After incrementing, we have ", $a, " and ", $b, "\n";
```

This time, however, the incrementing takes place first. `$a` is now 6, and `$b` is set to twice that, 12.

```
$b=++$a*2;
print "Now, we have ", $a, " and ", $b, "\n";
```

Finally, `$b` is decremented first and becomes 11. `$a` is set to `$b` plus 4, which is 15.

```
$a=--$b+4;
print "Finally, we have ", $a, " and ", $b, "\n";
```

The autoincrement operator actually does something interesting if the variable contains a string of only alphabetic characters, followed optionally by numeric characters. Instead of converting to a number, perl 'advances' the variable along the ranges a-z, A-Z, and 0-9. This is more easily understood from a few examples:

```
#!/usr/bin/perl
#auto2.plx
use warnings;
$a = "A9"; print ++$a, "\n";
$a = "bz"; print ++$a, "\n";
$a = "Zz"; print ++$a, "\n";
$a = "z9"; print ++$a, "\n";
$a = "9z"; print ++$a, "\n";
```

Should produce:

```
>perl auto2.plx
B0
ca
AAa
aa0
10
>
```

This shows that a 9 turns into a 0 and increments the next digit left. A 'z' turns into an 'a' and increments the next digit left. If there are no more digits to the left, either an 'a' or an 'A' is created, depending on the case of the current leftmost digit.

Multiple Assignments

We've said that = is an operator, but does that mean it returns a value? Well, actually it does. It returns whatever was assigned. This allows us to set up several variables at once. Here's a simple example of this (read it from right to left):

```
$d = $c = $b = $a = 1;
```

First we set `$a` to 1, and the result of this is 1. `$b` is set with that, the result of which is 1. And so it goes.

A slightly more complicated version occurs when you operate on the return value of the assignment. As usual, we need to pay attention to precedence. This won't work:

```
$b = 4 + $a = 1;
```

which is just as well, because it's horribly confusing. Perl complains that it 'Can't modify addition (+) in scalar assignment'. That is to say, it's trying to assign `1` to `4+$a`, and you can only assign to a variable, not to an addition. We say that addition is not a legal **lvalue**. It is not allowed on the left-hand side of an assignment.

If you wanted to do this, you'd have to say:

```
$b = 4 + ($a = 1);
```

This sets `$a` to 1 and `$b` to 5 as expected, but it's considered a bit messy. The reason for this is that setting various different variables with different values in one go is complicated to read and just the sort of thing that gives Perl a bad name.

Scoping

All the variables we've seen so far in our programs have been **global** variables, that is, they can be seen and changed from anywhere in the program. For the moment, that's not too much of a problem, since our programs are very small, and we can easily understand where things get assigned and used. However, when we start writing larger programs, this becomes a problem.

Why is this? Well, suppose one part of your program uses a variable, `$counter`. If another part of your program wants a counter, it can't call it `$counter` as well for fear of clobbering the old value. This becomes more of an issue when we get into **subroutines**, which are little sections of code we can temporarily call upon to accomplish something for us before returning to what we were previously doing. Currently, we'd have to make sure all the variables in our program had different names, and with a large program, that's not desirable. It would be easier to restrict the life of a variable to a certain area of the program. Let's see how this is done.

Try it out – Lexical variables

To achieve this, Perl provides another type of variable, called **lexical** variables. These are constrained to the enclosing block and all blocks inside it. If they're not currently inside a block, they are constrained to the current file. To tell perl that a variable is lexical, we say '`my $variable;`'. This creates a brand-new lexical variable for the current block and sets it to the undefined value. Here's an example:

```
#!/usr/bin/perl
#scope1.plx
use warnings;
$record = 4;
print "We're at record ", $record, "\n";

{
     my $record;
     $record = 7;
     print "Inside the block, we're at record ", $record, "\n";
}

print "We're still at record ", $record, "\n";
```

And this should tell you:

```
>perl scope1.plx
We're at record 4
Inside the block, we're at record 7
We're still at record 4
>
```

How It Works

Firstly, we set our global variable `$record` to 4.

```
$record = 4;
print "We're at record ", $record, "\n";
```

Now we enter a new block and create a new lexical variable. Important! This is completely and utterly unrelated to the global variable `$record` as `my` creates a **new** lexical variable. This exists for the duration of the block only, and has the undefined value.

```
{
     my $record;
```

Next, the lexical varable is set to `7` and printed out. The global is unchanged.

```
     $record = 7;
     print "Inside the block, we're at record ", $record, "\n";
```

Finally, the block ends, and the lexical ends with it. We say that it has gone 'out of scope'. The global remains, however, and so `$record` has the value 4.

```
}

print "We're still at record ", $record, "\n";
```

In order to make us think clearly about our programming, we will ask Perl to be strict about our variable use. The statement '`use strict;`' checks that, among other things, we've declared all our variables. We declare lexicals with `my`, and we can also declare globals in the same way with `our`. Here's what happens if we change our program to '`use strict`' format:

```
#!/usr/bin/perl
#scope2.plx
use strict;
use warnings;
$record = 4;
print "We're at record ", $record, "\n";

{
     my $record;
     $record = 7;
     print "Inside the block, we're at record ", $record, "\n";
}

print "We're still at record ", $record, "\n";
```

Now, the global `$record` is not declared. So sure enough, perl complains about it, telling us that:

```
Global symbol "$record" requires explicit package name.
```

We'll see exactly what this means in later chapters, but for now it suffices to declare `$record` as either a global or a lexical. Normally, we'd try and avoid globals as much as possible, but let's make it a global this once:

```
#!/usr/bin/perl
#scope3.plx
use strict;
use warnings;
our $record;
$record = 4;
print "We're at record ", $record, "\n";

{
     my $record;
     $record = 7;
     print "Inside the block, we're at record ", $record, "\n";
}

print "We're still at record ", $record, "\n";
```

Now perl is happy, and we get the same output as before. You should almost always start your programs with those two lines – turn on warnings, and then turn on strictness. Of course nobody's going to force you to use them, but they will help you avoid a lot of mistakes and will certainly give other people who have to look at your code more confidence in it.

Variable Names

We've not really examined yet what the rules are regarding what we can call our variables. We know that scalar variables have to start with a dollar sign, but what next? The next character must be a letter (uppercase or lowercase) or an underscore. After that, any combination of numbers, letters, and underscores is permissible, up to a total of 251 characters.

Be aware that Perl's variables are case-sensitive so `$user` is different from `$User`, and both are different from `$USER`.

The following are legal variable names: `$I_am_a_long_variable_name`, `$simple`, `$box56`, `$__hidden`, `$B1`

The following are not legal variable names: `$10c` (doesn't start with letter or underscore), `$mail-alias` (- is not allowed), `$your name` (spaces not allowed).

The Special Variable $_

There are certain variables, which Perl provides internally, that you either are not allowed to, or do not want to, overwrite. One which is allowed by the naming conventions above is `$_`, a very special variable indeed. `$_` is the 'default variable' that a lot of functions read from and write to if no other variable is given. We'll see plenty of examples of it throughout the book. For your reference, Appendix B lists all the special variables that perl uses and what they do.

Apart from the prefix, the same restrictions apply to arrays and hashes. Scalar variables are prefixed by a dollar sign ($), arrays begin with an at sign (@), and hashes begin with a percent sign (%).

Variable Interpolation

We said earlier that double-quoted strings interpolate variables. What does this mean? Well, if you mention a variable, sa,y `$name` in the middle of a double-quoted string, you get the value of the variable, rather than the actual characters. Interpolation happens for scalar variables and arrays but not for hashes. As an example, see what perl does to this:

```
#!/usr/bin/perl
#varint1.plx
use warnings;
use strict;
my $name = "fred";
print "My name is $name\n";
```

This is what comes out:

```
>perl varint1.plx
My name is fred
>
```

Perl interpolates the value of `$name` into the string. Note that this doesn't happen with single-quoted strings, just like escape sequence interpolation:

```
#!/usr/bin/perl
#varint2.plx
use warnings;
use strict;
my $name = "fred";
print 'My name is $name\n';
```

Here we get:

```
>perl varint2.plx
My name is $name\n
>
```

This doesn't just happen in things we print, it happens every time we construct a string:

```
#!/usr/bin/perl
#varint3.plx
use warnings;
use strict;
my $name = "fred";
my $salutation = "Dear $name,";
print $salutation, "\n";
```

This gives us:

```
>perl varint3.plx
Dear fred,
>
```

This has exactly the same effect as:

```
my $salutation = "Dear ". $name. ",";
```

but is more concise and easier to understand.

If you need to place text immediately after the variable, you can use braces to delimit the name of the variable. Take this example:

```
#!/usr/bin/perl
#varint4.plx
use warnings;
use strict;
my $times = 8;
print "This is the $timesth time.\n";
```

This won't work, because perl looks for a variable `$timesth` that hasn't been declared. In this case, we have to change the last line to this:

```
print "This is the ${times}th time.\n";
```

Now we get the right thing:

```
>perl varint4.plx
This is the 8th time.
>
```

Currency Converter

Let's begin to wind up this chapter with a real example – a program to convert between currencies. This is our very first version, so we won't make it do anything too complex. As we get more and more advanced, we'll be able to hone and refine it.

Try it out – Currency Converter

Open your editor, and type in the following program:

```
#!/usr/bin/perl
#currency1.plx
use warnings;
use strict;
my $yen = 180;
print "49518 Yen is ", (49_518/$yen), " pounds\n";
print "360 Yen is ",   (   360/$yen), " pounds\n";
print "30510 Yen is ", (30_510/$yen), " pounds\n";
```

Save this, and run it through perl. This is what you should see:

```
> perl currency1.plx
49518 Yen is 275.1 pounds
360 Yen is 2 pounds
30510 Yen is 169.5 pounds
>
```

How It Works

First, we start our program in the usual way:

```
#!/usr/bin/perl
use warnings;
use strict;
```

Next, we declare the exchange rate to be a lexical variable and set it to `180`. (At the time I wrote this, there were roughly 180 Yen to the Pound.)

```
my $yen = 180;
```

Notice that we can declare and assign a variable at the same time. Now we do some calculations based on that exchange rate:

```
print "49518 Yen is ", (49_518/$yen), " pounds\n";
print "360 Yen is ",   (   360/$yen), " pounds\n";
print "30510 Yen is ", (30_510/$yen), " pounds\n";
```

And amazingly, the calculations come out to roughly round numbers!

Of course, this is currently of limited use, because the exchange rate fluctuate, and we might want to change some different amounts at times. To do either of these things, we need to be able to ask the user for additional data when we run the program.

Introducing <STDIN>

The way we do this is with the construct `<STDIN>`. We'll explain it in detail when we look at file handling in Chapter 6, but it reads a line from the file called **standard input**. Usually, the standard input is not really a file, but the user's keyboard. Similarly, the print function by default writes to the file called **standard output**, which is usually the user's screen.

So, in order to ask the user for a line of text, we say something like:

```
print "Please enter something interesting\n";
$comment = <STDIN>;
```

This will read one line from the user and assign the string that was read to the variable `$comment`. Let's use this to get the exchange rate from the user when the program is run.

Try it out - Currency Converter, Mark 2

Using your editor, change the file `currency1.plx` to `currency2.plx` as follows:

```
#!/usr/bin/perl
#currency2.plx
use warnings;
use strict;
print "Currency converter\n\nPlease enter the exchange rate: ";
my $yen = <STDIN>;
print "49518 Yen is ", (49_518/$yen), " pounds\n";
print "360 Yen is ",   (   360/$yen), " pounds\n";
print "30510 Yen is ", (30_510/$yen), " pounds\n";
```

Now when you run the program, you'll be asked for the exchange rate. The currency values will be calculated using the rate you entered:

```
> perl currency2.plx
Currency converter

Please enter the exchange rate: 100
49518 Yen is 495.18 pounds
360 Yen is 3.6 pounds
30510 Yen is 305.1 pounds
>
```

How It Works

This time we read the exchange rate from the user's keyboard, and perl converts the string to a number in order to perform the calculation.

So far, we haven't done any checking to make sure that the exchange rate given makes sense; This is something we'll need to think about in future.

Summary

Perl has three main data types – scalars, lists, and hashes. Lists and hashes are made up of scalars, which are in turn made up of integers, floating-point numbers, and strings. Perl converts between these three automatically, so we don't need to distinguish between them.

Double- and single-quoted strings differ in the way they process the text inside them. Single-quoted strings do little to no processing at all, whereas double-quoted strings interpolate escape sequences and variables. We can use alternative delimiters and here-documents as alternative ways of entering strings.

We can operate on these scalars in a number of ways – ordinary arithmetic, bitwise arithmetic, string manipulation, and logical comparison. We can also combine logical comparisons with Boolean operators. These operators vary in precedence, which is to say that some take effect before others, and as a result we must use brackets to enforce the precedence we want.

Scalar variables are a way of storing scalars so that we can get at them and change them. Scalar variables begin with a dollar sign (`$`) and are followed by one or more alphanumeric characters or underscores. There are two types of variables – lexical and global. Globals exist all the way through the program and so can be troublesome if we don't keep very good track of where they are being used. Lexicals have a life span of the current block, so we can use them safely without worrying about clobbering similarly named variables somewhere else in the program.

Finally, we've seen a way of getting input from the user, storing it into a variable, and acting upon it. Try the exercises that follow. They are a good indication of how much you have learned.

Exercises

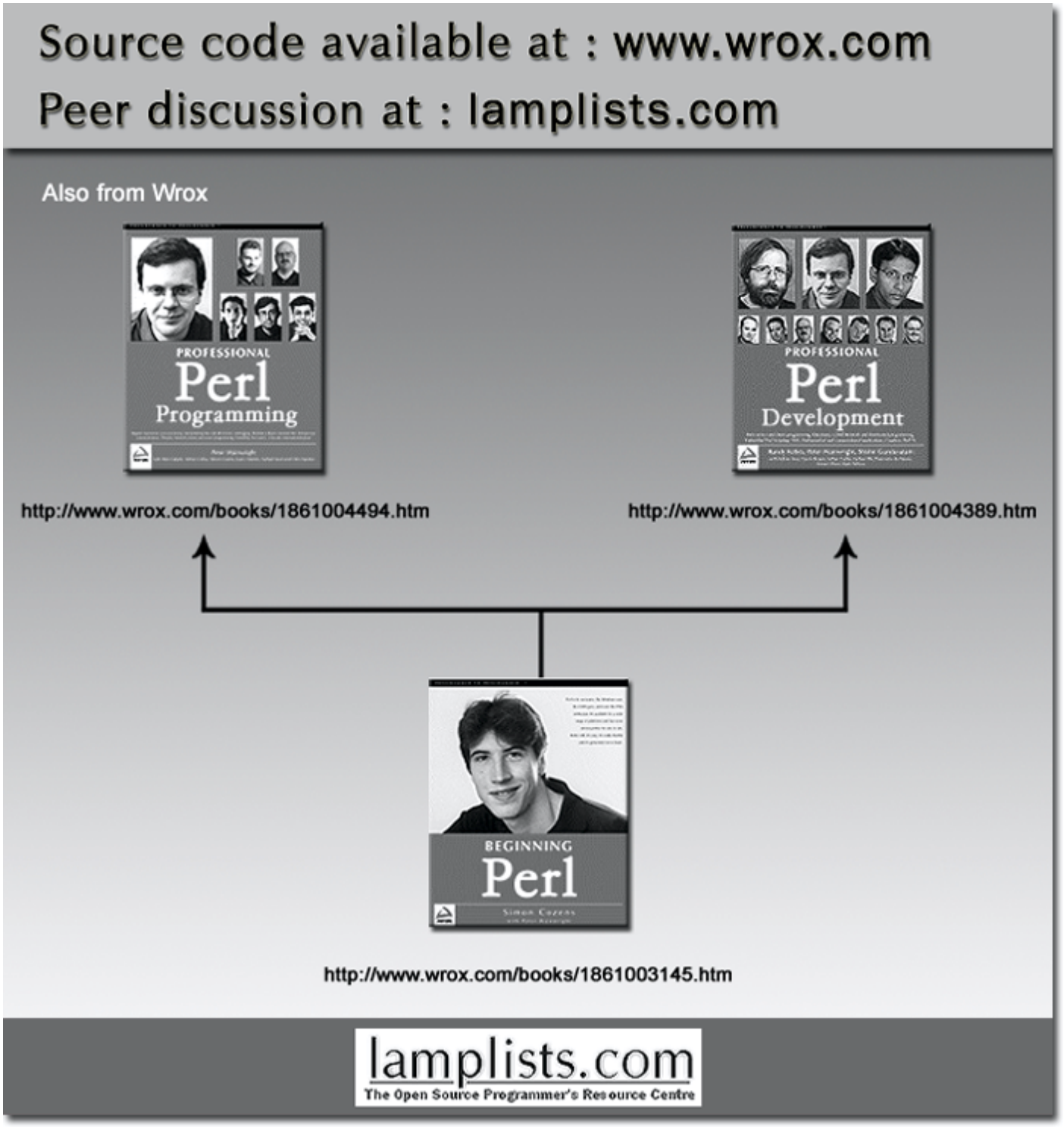
1. Change the currency conversion program so that it asks for an exchange rate and three prices to convert.

2. Write a program that asks for a hexadecimal number and converts it to decimal. Then change it to convert an octal number to decimal.

3. Write a program that asks for a decimal number less than 256 and converts it to binary. (Hint: You may want to use the bitwise and operator, 8 times.)

4. Without the aid of the computer, work out the order in which each of the following expressions would be computed and their value. Put the appropriate parentheses in to reflect the normal precedence:

   - 2+6/4-3*5+1
   - 17+-3**3/2
   - 26+3^4*2
   - 4+3>=7||2&4*2<4



3

Lists and Hashes

As we saw from the previous chapter, there are three types of data: scalars, lists, and hashes. So far we've only been working with scalars – single numbers or strings. We've joined two single strings together to make one, converted one currency only, and held one number in a variable.

There are times, when we'll want to group together information or express correspondences between information. Just like the ingredients in a recipe or the pieces in a jigsaw, some things belong together in a natural sequence, for example, individual lines in a file, or the names of players in a squash ladder. In Perl, we represent these relationships in lists – series of scalars. They can be stored in another type of variable called an **array**, and we call each piece of data in the list an **element**.

Alternatively, some things are better expressed as a set of one-to-one correspondences. A phone book, for example, is a set of correspondences between addresses and phone numbers. In Perl, structures like the phone book are represented as a **hash**. Some people call them 'associative arrays' because they look a bit like arrays where each element is associated with another value. Most Perl programmers find that a bit too long-winded and just call them hashes.

In this chapter, we'll see how we build up lists and hashes and what we can do with them when we've got them. We'll also begin to look at some control structures, which will enable us to step through lists and arrays. As well as all this, we'll learn how to process data more than once without having to write out the relevant sections of our program again and again.

Lists

We're all familiar with lists from everyday life. Think about a shopping list, what properties does it have? First of all, it's a single thing, one piece of paper. Secondly, it's made up of a number of values. In the case of a shopping list, you might want to say that these values are actually strings – `"ketchup"`, `"peanut butter"`, `"ice cream"`, and so on. Finally, it's also ordered, which means that there's a first item and a last item.

Lists in Perl aren't actually that much different: They're counted as a single thing, but they're made up of a number of values. In Perl, these values are scalars, rather than purely strings. They're also stored in the order they were created.

We'll specify lists in our program code as literals, just like we did with strings and numbers. We'll also be able to perform certain operations on them. Let's begin by looking at a few simple lists and how we create them.

Simple Lists

The simplest shopping list is one where you have nothing to buy. Similarly, the simplest list in Perl has no elements in it. Here's what it looks like:

```
()
```

A simple pair of parentheses – that's how we denote a list. However, it's not very interesting. Let's try putting in some values:

```
(42)
("cheese")
```

As you can see, we have created two lists, one containing a number, and one containing a string – so far so good. Now, remember that I said `print` was a list operator? The magic about operators like `print` is that you can omit the brackets. Saying `print "cheese"` is just the same as saying `print("cheese")`. So what we give to `print` is really a list. We're allowed to leave out the parentheses if we wish.

From this, we should be able to work out how to put multiple values into a list. When we said:

```
print("Hello, ", "world", "\n");
```

we were actually passing the following list to the print operator:

```
("Hello ", "world", "\n")
```

As you can see, this is a three-element list, and the elements are separated with commas. Computers and computer people start counting from zero, so here's your chance to practise. The zeroth element is `"Hello "`, the first is `"world"`, and the second is `"\n"`. Now, let's do that again with numbers instead of strings:

```
(123, 456, 789)
```

This is exactly the same as before, and if we were to print this new list, this is what would happen:

```
#!/usr/bin/perl
# numberlist.plx
use warnings;
use strict;

print (123, 456, 789);
```

```
>perl numberlist.plx
123456789>
```

As before, perl doesn't automatically put spaces between list elements for us when it prints them out, it just prints them as it sees them. Similarly, it doesn't put a new line on the end for us. If we want to add spaces and new lines, then we need to put them into the list ourselves.

More Complex Lists

We can also mix strings, numbers, and variables in our lists. Let's see an example of a list with several different types of data in it:

Try It Out – Mixed Lists

Although this isn't very different from what we were doing with print in the last chapter, this example reinforces the point that lists can contain any scalar literals and scalar variables. So, type this in, and save it as `mixedlist.plx`.

```
#!/usr/bin/perl
# mixedlist.plx
use warnings;
use strict;

my $test = 30;
print
      "Here is a list containing strings, (this one) ",
      "numbers (",
      3.6,
      ") and variables: ",
      $test,
      "\n"
;
```

When you run that, here's what you should see:

```
> perl mixedlist.plx
Here is a list containing strings, (this one) numbers (3.6) and variables: 30
>
```

How It Works

This is how we're going to start programs from now on, in order to make sure that we have both warnings and extra checks turned on. Remember that if you're using a version of Perl less than 5.6 you'll need to say `#!/usr/bin/perl -w` for the first line to turn on warnings, and also leave out the `use warnings;` line:

```
#!/usr/bin/perl
# mixedlist.plx
use warnings;
use strict;
```

Next, we initialize our variable. Note that we can declare the variable and give it a value on the same statement. It's exactly the same as doing this:

```
my $test = 30;
```

but is just as clear and saves a line, so it's a common thing to do – it's one of Perl's many idioms:

```
my $test;
$test = 30;
```

> **Perl is more like a human language than most programming languages. Perl was designed to be easy for humans to write, not for computers to read. Just like human languages, Perl has shortcuts and idioms. Perl programmers do tend to use a lot of these idioms in their code, and you may come across them if you're reading other people's programs. As a result of this, we're not going to shy away from those idioms, even if they can be slightly confusing at times. Instead, we'll try taking them apart to see how they work.**

Finally, we have our list. It's a list of six elements, including literal strings, literal numbers, and a scalar variable for good measure:

```
print
      "Here is a list containing strings, (this one) ",
      "numbers (",
      3.6,
      ") and variables: ",
      $test,
      "\n"
;
```

Since variables interpolate in double-quoted strings inside lists just as well as at any other time, we could have done that all as one long single-element list:

```
print ("Here is a list containing strings, (this one) numbers (3.6) and variables:
$test\n");
```

There is a disadvantage of writing your code this way. New lines in your string literals will turn into new lines in your output. So, if you keep the maximum length of the lines in your source code to about 80 columns (it's a good idea to keep your programs readable), one long string will wrap over, and you'll see this sort of thing:

```
> perl mixedlist.plx
Here is a list containing strings, (this one) numbers (3.6) and
variables: 30
>
```

So if you're ever printing long strings, consider splitting it up into a list of smaller strings on separate lines as we've done above.

In the same way, single-quoted strings act no differently when they're list elements: `('A number:', '$test')` will actually give you two strings, and if you print out that list, you will see this:

```
A number:$test
```

Similarly, `q//` and `qq//` can be used to delimit strings when you're using them as list elements. There's absolutely no difference between the previous example and `(q/A number:/, q/$test/)`

However, there's another trick. When your lists are made up purely from single words, you can specify them with the `qw//` operator. Just like the `q//` and `qq//` operators, you can choose any paired brackets or non-word characters as your delimiters. The following lists are all identical:

```
('one', 'two', 'three', 'four')
qw/one two three four/
qw(one two three four)
qw<one two three four>
qw{one two three four}
qw[one two three four]
qw|one two three four|
```

You shouldn't separate your words with commas inside `qw//`. In fact, if you do, perl will complain, especially since we always have warnings turned on! For example, if we ran this:

```
#!/usr/bin/perl
# badlist.plx
use warnings;
use strict;
print qw(one,two,three,four);
```

we would quickly see

```
> perl badlist.plx
Possible attempt to separate words with commas at badlist.plx line 5.
Possible attempt to separate words with commas at badlist.plx line 5.
Possible attempt to separate words with commas at badlist.plx line 5.
one,two,three,four>
```

You can use any white space, tabs, or new lines to separate your elements. The same list as above `('one', 'two', 'three', 'four')` can also be written like this:

```
qw(
      one
      two
      three
      four
)
```

One last thing to note is that perl automatically **flattens** lists. That is, if you try putting a list inside another list, the internal list loses its identity. In effect, perl removes all the brackets apart from the outermost pair. There's no difference at all between any of these three lists:

```
(3, 8, 5, 15)
((3, 8), (5, 15))
(3, (8, 5), 15)
```

Similarly, perl sees each of these lists as exactly the same as the others:

```
('one', 'two', 'three', 'four')
(('one', 'two', 'three', 'four'))
(qw(one two three), 'four')
(qw(one two), q(three), 'four')
(qw(one two three four))
```

This doesn't mean that you can't store a list inside another list and keep the structure of the first list intact. For the moment we can't do it, but we'll see how it's done when we look at references in Chapter 7.

Accessing List Values

We've now seen most of the ways of building up lists in Perl, and we can throw lists at list operators like `print`. But another thing we need to be able to do with lists is access a specific element or set of elements within it. The way to do this is to place the number of the elements we want in square brackets after the list, like this:

```
#!/usr/bin/perl
# access.plx
use warnings;
use strict;

print (('salt', 'vinegar', 'mustard', 'pepper')[2]);
print "\n";
```

Before you run this, though, see if you can work out which word will be printed.

```
>perl access.plx
mustard
>
```

Did you think it was going to be 'vinegar'? Don't forget that computers start counting things from zero!

You should also notice that we had to put brackets around the whole thing. This is because the precedence of `print` is extremely high. Without the brackets, perl groups the statement in two parts like this:

```
print('salt', 'vinegar', 'mustard', 'pepper')    [2];
```

This means the whole of the list is passed to `print`, after which perl attempts to retrieve the second element of `print`. The problem is, you can only take an element from a list, and as we already know, `print` isn't a list.

So, since print needs to be passed a list, we make a list out of the element we want:

```
print (
      ('salt', 'vinegar', 'mustard', 'pepper')[2]
);
```

The element you want doesn't have to be given as a literal – variables work just as well. Here's an example we'll draw on later:

Try It Out – Months Of The Year

We'll create a list of the months of the year, and then use a variable to access them. Save this file as `months.plx`:

```
#!/usr/bin/perl
# months.plx
use warnings;
use strict;

my $month = 3;
print qw(
     January     February    March
     April       May         June
     July        August      September
     October     November    December
)[$month];
```

When this is run, you should now be expecting it to give you 'April', and it does:

```
>perl months.plx
April>
```

How It Works

The key piece of code for this example is the last statement:

```
print qw(
     January     February    March
     April       May         June
     July        August      September
     October     November    December
)[$month];
```

We have `$month` as `3`, and so we are telling perl to print out the third element of the list, starting from zero. Because we're using `qw//` we can use arbitrary whitespace, tabs, and new lines to separate each list element, which allows us to present the months in a neat table.

This is exactly the sort of situation that `qw//` was created for. We have a list comprised completely of single words, and we want a way to represent that to perl in a tidy way in our source code. It's far easier to read than spelling the list out longhand, even though these statements are equivalent:

```
print (('January','February', 'March', 'April', 'May', 'June', 'July', 'August',
'September', 'October', 'November', 'December')[$month]);
```

What do you think would happen if we chose a non-integer value for our element? Let's use a value with a fractional part. Change the above file so that line 5 reads:

```
my $month = 2.2;
```

Perl will round the number in this case, and you should get the answer March. In fact, perl always rounds towards zero, so anything between 2 and 3 will get you March.

What about negative numbers? Actually, something interesting happens here – perl starts counting backwards from the end of the list. So element -1 is the last one, -2 the second before last, and so on.

```
#!/usr/bin/perl
# backwards.plx
use warnings;
use strict;

print qw(
      January     February    March
      April       May         June
      July        August      September
      October     November    December
)[-1];
```

And, true to form, we'll get the last element of the array when we run the program.

```
>perl backwards.plx
December>
```

List Slices

So much for getting a single element out of a list. What if we want to get several? Well, instead of putting a number or a scalar variable inside those square brackets, you can actually put a list there instead. For example, this:

```
(19, 68, 47, 60, 53, 51, 58, 55, 47)[(4, 5, 6)]
```

returns another list consisting of elements four, five, and six: `(53, 51, 58)`. Actually, inside the square brackets, we don't need the additional set of parentheses, so you might as well say:

```
(19, 68, 47, 60, 53, 51, 58, 55, 47)[4, 5, 6]
```

We call this getting a **list slice,** and the same methods work with lists of strings:

Try It Out – Multiple Elements Of A List

This program is called `multilist.plx`. Just like the above examples, we're taking several elements from a list:

```
#!/usr/bin/perl
# multilist.plx
use warnings;
use strict;

my $mone; my $mtwo;
($mone, $mtwo) = (1, 3);

print (("heads ", "shoulders ", "knees ", "toes ")[$mone, $mtwo]);
print "\n";
```

Try and think what it's going to produce before you run it. Here's what happens:

```
> perl multilist.plx
shoulders toes
>
```

As you may have realized, we simply printed out the first and the third elements from the list, if you start counting from zero.

How It Works

There are two key tricks in this example. The first is on line seven:

```
($mone, $mtwo) = (1, 3);
```

You might be able to see what this line does, from how the rest of the program runs. The value of `$mone` is set to 1, and `$mtwo` to 3. But how does this work?

Perl allows lists on the left-hand side of an assignment operator – we say that lists are legal **lvalues**. When we assign one list to another, the right-hand list is built up first. Then perl assigns each element in turn, from the right hand side of the statement to the left. So 1 is assigned to `$mone`, and then 3 is assigned to `$mtwo`.

If you're okay with that, then now's a good time for a quick quiz. Suppose we've done the above: `$mone` is 1 and `$mtwo` is 3. What do you think would happen if we said this:

```
($mone, $mtwo) = ($mtwo, $mone);
```

Well, the right-hand list is built up first, so perl looks at the values of the variables and constructs the list `(3, 1)`. Then the 3 is assigned to `$mone`, and the 1 assigned to `$mtwo`. In effect, we've swapped the values of the variables around – a handy trick to learn and remember. Chances are that it's something you'll need to do again and again over time.

Back to our example! Once we've set `$mone` to 1 and `$mtwo` to 3, we can pick out these elements from a list. There's nothing that says that we have to use literals to pick out the elements we want. This:

```
print (("heads ", "shoulders ", "knees ", "toes ")[$mone, $mtwo]);
```

is interpreted by perl just the same as this:

```
print (("heads ", "shoulders ", "knees ", "toes ")[1, 3]);
```

Indeed, both statements equate to the same thing – picking out a list consisting of the first and third elements of our original list and printing them. In effect, we call:

```
print ("shoulders ", "toes ");
```

which is indeed what happens.

Ranges

Often our lists will be a lot simpler than a group of different values. We'll want to talk about "the numbers 1 to 10" or "the letters a to z." Rather than write them out longhand, Perl gives us the ability to specify a range of numbers or letters. Suppose we say:

```
(1 .. 6)
```

This will give us a list of 6 elements from 1 to 6, exactly the same as if we had said `(1, 2, 3, 4, 5, 6)`. This can really save time when you're dealing with a few hundred elements, but note that this only works for integers. If you'll recall our efforts to use lists to get at elements of another list, the fractional values in the list were rounded towards zero. Exactly the same thing happens here:

```
(1.4 .. 6.9)
```

would produce `(1, 2, 3, 4, 5, 6)` again. There's no problems with using negative numbers in you ranges, though. For example:

```
(-6 .. 3)
```

produces the list `(-6, -5, -4, -3, -2, -1, 0, 1, 2, 3)`

The right-hand number must, however, be higher than the left-hand one, so we can't use this technique to count down. Instead, you can reverse any list using the `reverse` operator, as we'll see very shortly.

We can do the same for letters as well:

```
('a'..'k')
```

This will give us an 11-element list, consisting of each letter from 'a' to 'k' inclusive. Note that we can't mix letters and numbers within a range. If we try, perl will interpret the string as a number, and treat it as zero:

Try It Out – Counting Up And Down

Here's a demonstration of all the things we can do with ranges:

```
#!/usr/bin/perl
# ranges.plx
use warnings;
use strict;

print "Counting up: ", (1 .. 6), "\n";
print "Counting down: ", (6 .. 1), "\n";
print "Counting down  (properly this time) : ", reverse(1 .. 6), "\n";

print "Half the alphabet: ", ('a' .. 'm'), "\n";
print "The other half (backwards): ", reverse('n' .. 'z'), "\n";

print "Going from 3 to z: ", (3 .. 'z'), "\n";
print "Going from z to 3: ", ('z' .. 3), "\n";
```

Which of those will work and which won't? Let's find out…:

```
> perl ranges.plx
Argument "z" isn't numeric in range (or flop) at ranges.plx line 13.
Argument "z" isn't numeric in range (or flop) at ranges.plx line 14.
Counting up: 123456
Counting down:
Counting down  (properly this time): 654321
Half the alphabet: abcdefghijklm
The other half (backwards): zyxwvutsrqpon
Going from 3 to z
Going from z to 30123
>
```

How It Works

After the usual opening, we first count upwards with a range:

```
print "Counting up: ", (1 .. 6), "\n";
```

We've seen the range in action before, and we know this produces `(1, 2, 3, 4, 5, 6)`. We pass `print` a list containing the string `"Counting up: "`, the six elements, and a new line. Because a list inside a list gets flattened, we're actually just passing an eight-element list. It's the same as if we'd done:

```
print "Counting up: ", 1, 2, 3, 4, 5, 6, "\n";
```

And we get the expected result:

```
Counting up: 123456
```

Next, we try and count down:

```
print "Counting down: ", (6 .. 1), "\n";
```

This doesn't work because the right hand side needs to be bigger than the left, and all that's produced is the empty list, `()`. To count down properly, we need to make a list using `(1 .. 6)` as before and turn it around. The `reverse` operator turns any list on its head. For example:

```
reverse (qw(The cat sat on the mat))
```

produces the same as:

```
qw(mat the on sat cat The)
```

In this case, `reverse(1..6)` produces `(1, 2, 3, 4, 5, 6)` and then turns it around to become `(6, 5, 4, 3, 2, 1)`, and we see the list appear in that order:

```
Counting down  (properly this time) : 654321
```

Next we demonstrate a simple alphabetic range:

```
print "Half the alphabet: ", ('a' .. 'm'), "\n";
```

This range expands to the values 'a', 'b', 'c', and so, on all the way to 'm'. Doing that backwards is easy:

```
print "The other half (backwards): ", reverse('n' .. 'z'), "\n";
```

Now we come to the ones that don't work, and it's no surprise that perl warns us against them:

```
Argument "z" isn't numeric in range (or flop) at ranges.plx line 13.
Argument "z" isn't numeric in range (or flop) at ranges.plx line 14.
```

The lines in question are:

```
print "Going from 3 to z: ", (3 .. 'z'), "\n";
print "Going from z to 3: ", ('z' .. 3), "\n";
```

What does the error message mean? Well, pretty much what it says: we gave an argument of 'z' to a range, when it was expecting a number instead. The interpreter converted the 'z' to a number, as per the rules in the last chapter, and got a 0. It's equivalent to this:

```
print "Going from 3 to z: ", (3 .. 0), "\n";
print "Going from z to 3: ", (0 .. 3), "\n";
```

The first one produces an empty list, and the second one counts up from 0 to 3.

Combining Ranges and Slices

We can, of course, use ranges in our list slices. The following gets March through September:

```
(qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec)[2..8])
```

And this gets November through February via December and January (remember that –2 is the second to last, and -1 the last):

```
(qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec)[-2..1])
```

We can also use a mixture of ranges and literals in our slice. This gives you January, April, and August to December:

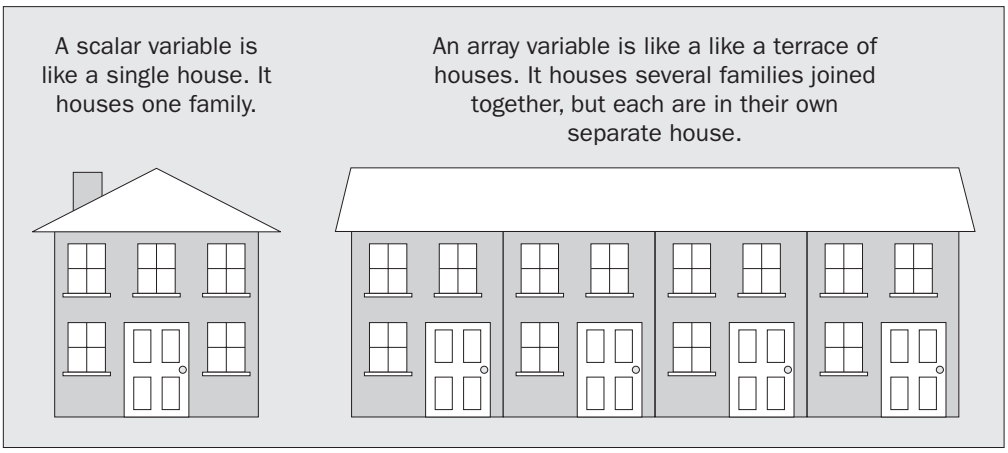
```
(qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec)[0,3,7..11])
```

It may be a bit confusing, but have a go at slicing your own arrays, and you'll get the hang of it in no time at all.

Arrays

Just as with scalars, there's only so much you can do with literals. Literal lists get cumbersome to repeat and don't allow us to manipulate them at all. If we wanted to say 'the same list, but without the last element', we couldn't do it. As before, we need to find a way to store them in a variable.

The variable storage we use for lists is called an **array**. Whereas the name of a scalar variable started with a dollar sign, arrays start with an at sign (@). The same rules for naming your arrays apply as for any other variables: start with an alphabetic character or underscore, followed by one or more alphabetic characters, underscores, or numbers:



Assigning Arrays

We store a list in an array just like we store a scalar literal into a scalar variable, by assigning it with =:

```
@array = (1,2,3);
```

Once we've assigned our array, we can use our array where we would use a list:

```
#!/usr/bin/perl
# dayarray.plx
use warnings;
use strict;

my @days;
@days = qw(Monday Tuesday Wednesday Thursday Friday Saturday Sunday);
print @days, "\n";
```

This prints:

```
> perl dayarray.plx
MondayTuesdayWednesdayThursdayFridaySaturdaySunday
>
```

Note that `$days` is a completely different variable from `@days` – setting one does nothing to the other. In fact, if you were to do this:

```
#!/usr/bin/perl
# baddayarray1.plx
use warnings;
use strict;

my @days;
@days = qw(Monday Tuesday Wednesday Thursday Friday Saturday Sunday);
$days = 31;
```

you would get the following error:

```
Global symbol "$days" requires explicit package name at dayarray.plx line 8.
```

This is because you have declared `@days` to be a lexical variable, but not `$days`. Even when you declare them both, setting one has no effect on the other:

```
#!/usr/bin/perl
# baddayarray2.plx
use warnings;
use strict;

my @days;
my $days;
@days = qw(Monday Tuesday Wednesday Thursday Friday Saturday Sunday);
$days = 31;
print @days, "\n";
print $days, "\n";
```

prints:

```
MondayTuesdayWednesdayThursdayFridaySaturdaySunday
31
```

What would happen if you assigned an array to a scalar variable? Well, let's see:

Try It Out - Assigning An Array To A Scalar

Here's an example of two arrays that we will assign to two different scalar variables:

```
#!/usr/bin/perl
# arraylen.plx
use warnings;
use strict;

my @array1;
my $scalar1;
@array1 = qw(Monday Tuesday Wednesday Thursday Friday Saturday Sunday);
$scalar1 = @array1;

print "Array 1 is @array1\nScalar 1 is $scalar1\n";

my @array2;
my $scalar2;
@array2 = qw(Winter Spring Summer Autumn);
$scalar2 = @array2;

print "Array 2 is @array2\nScalar 2 is $scalar2\n";
```

Save this as `arraylen.plx`, and run it through perl:

```
> perl arraylen.plx
Array 1 is Monday Tuesday Wednesday Thursday Friday Saturday Sunday
Scalar 1 is 7
Array 2 is Winter Spring Summer Autumn
Scalar 2 is 4
>
```

Hmm… The first array has seven elements, and the scalar value is 7. The second has four elements, and the scalar value is 4.

How It Works

Note how array variables interpolate in a double-quoted string. We've seen that if you put a scalar-variable name inside a string, perl will fill in the value of the variable. Now we've put an array variable in a string, and perl has filled it in, but it has placed spaces between the elements. Look at the following two `print` statements:

```
@array = (4, 6, 3, 9, 12, 10);
print @array, "\n";
print "@array\n";
```

The first one does exactly what we've seen with lists, printing all the elements next to each other. The second statement, however, inserts a space between each element:

```
46391210
4 6 3 9 12 10
```

This adding of spaces between elements is what happens when an array is interpolated in a double-quoted string. As with scalars, interpolation is not confined to `print`. For example:

```
$scalar =  "@array\n";
```

is the same as:

```
$scalar = "4 6 3 9 12 10\n";
```

Forcing variables to make sense in a string is called **stringifying** them.

Scalar vs List Context

What happens when we assign an array to a scalar variable? Well, one key point to remember is that perl knows exactly what type of value you want, whether a scalar or an array, at any stage in an operation, and will do its best to make sure you get it.

For example, if we're looking to assign to a scalar variable, we need to have a scalar value – the assignment is taking place in **scalar context**. On the other hand, for example, `print` expects to see a list of arguments. Those arguments are in **list context**. However, some operations may return different values depending on which context they are called. That's what's happening in this case:

```
print @array1;
$scalar1 = @array1;
```

The first line is in list context. In list context, an array returns the list of its elements. In the second line, however, the assignment wants to see a single result, or scalar value, and therefore it is in scalar context. In scalar context, an array returns the number of its elements, in our case, 7 for the days and 4 for the seasons.

If we were to do this:

```
@array2 = @array1;
```

we would be assigning to an array. So we're looking for a **list** of values to fill `@array2`. Here, we're back in list context, and so `@array2` gets filled with all of the values of `@array1`.

We can force something to be in scalar context when it expects to be in list context by using the `scalar` operator. Compare these two statements:

```
print @array1;
print scalar @array1;
```

As we've explained before, `print` usually wants a list, so perl evaluates `print`'s arguments in list context. In the example above, `print` is looking to get a list from each of arguments. That's why the first statement prints the contents of `@array1`. If we force `@array1` into scalar context, then the number of elements in the array is passed to `print` and not the contents of the array:

> **Perl distinguishes between operations that want a list and operations that want a scalar. Those that want a list, such as `print` or assigning to an array, are said to be in list context. Those that want a scalar are said to be in scalar context. The value of an array in list context is the list of its elements – the value in scalar context is the number of its elements.**

Adding to an Array

How do we add elements to an array? Well, one way to do it is by using the 'list flattening' principle and treating our arrays as lists. This isn't a particularly good way to do it, but it works:

```
#!/usr/bin/perl
# addelem.plx
use warnings;
use strict;

my @array1 = (1, 2, 3);
my @array2;
@array2 = (@array1, 4, 5, 6);
print "@array2\n";

@array2 = (3, 5, 7, 9);
@array2 = (1, @array2, 11);
print "@array2\n";
```

```
>perl addelem.plx
1 2 3 4 5 6
1 3 5 7 9 11
>
```

It's far better, however, to use the functions we're going to see later on – `push`, `pop`, `shift`, and `unshift`.

Accessing an Array

Once we've got our list of scalars into an array, it would make sense to be able to get them back out again. We do this slightly differently to the way we get values out of lists.

Accessing Single Elements

So, we can now put elements into an array:

```
my @array = (10, 20, 30);
```

If we look at the array in scalar context, we get the number of elements in it. So:

```
print scalar @array;
```

will print the value 3. But how do we get at one of those elements? We could use the list assignment we were looking at earlier:

```
my $scalar1; my $scalar2; my $scalar3;
($scalar1, $scalar2, $scalar3) = @array;
print "Scalar one is $scalar1\n";
print "Scalar two is $scalar2\n";
print "Scalar three is $scalar3\n";
```

This will print out each of the elements:

```
Scalar one is 10
Scalar two is 20
Scalar three is 30
```

To get at a single element, we do something quite similar to what we did with a list. To get a single element from a list, if you remember, we put the number we want in square brackets after it:

```
$a = (10, 20, 30)[0];
```

This will set `$a` to the zeroth element, 10. We could do this:

```
$a = (@array)[0];
```

in exactly the same way. However, it's more usual to write that as follows:

```
$a = $array[0];
```

Look carefully at that. Even though `@array` and `$array` are different variables, we use the `$array[]` form. Why?

> **The prime rule is this: the prefix represents what you want to get, not what you've got. So @ represents a list of values, and $ represents a single scalar. Hence, when we're getting a single scalar from an array, we never prefix the variable with @ – that would mean a list. A single scalar is always prefixed with a $.**

`$array[0]` and `@array` aren't related – `$array[0]` can only refer to an element of the `@array` array. If you try and use the wrong prefix, perl will complain with a warning:

```
#!/usr/bin/perl
# badprefix.plx
use warnings;
use strict;

my @array = (1, 3, 5, 7, 9);
print @array[1];
```

will print:

```
>perl badprefix.plx
Scalar value @array[1] better written as $array[1] at badprefix.plx line 8.
3>
```

We call the number in the square brackets the **array index** or **array subscript**. The array index is the number of the element that we want to get hold of. Back in our little street, we could explain arrays like so:

Just like extracting elements from lists, we can use a scalar variable as our subscript:

```
#!/usr/bin/perl
# scalarsub.plx
use warnings;
use strict;

my @array = (1, 3, 5, 7, 9);
my $subscript = 3;
print $array[$subscript], "\n";
$array[$subscript] = 12;
```

This prints the third element from zero, which has the value `7`. It then changes that 7 to a 12. Negative subscripts work from the end; as before, `$array[-1]` will give you the last element in the array.

Now let's write something to extract a given element from an array:

Try It Out – The Joke Machine

We'll use arrays to write a program to tell us some (really bad) jokes. We actually set up two arrays – one containing the question, and one containing the answer:

```
#!/usr/bin/perl
# joke1.plx
use warnings;
use strict;

my @questions = qw(Java Python Perl C);
my @punchlines = (
      "None. Change it once, and it's the same everywhere.",
      "One. He just stands below the socket and the world revolves around him.",
      "A million. One to change it, the rest to try and do it in fewer lines.",
      '"CHANGE?!!"'
);

print "Please enter a number between 1 and 4: ";
my $selection = <STDIN>;
$selection -= 1;
print "How many $questions[$selection] ";
print "programmers does it take to change a lightbulb?\n\n";
sleep 2;
print $punchlines[$selection], "\n";
```

```
> perl joke1.plx
Please enter a number between 1 and 4: 3
How many Perl programmers does it take to change a lightbulb?

A million. One to change it, the rest to try and do it in fewer lines.
```

Hmm. I don't think I'm ready for the move into stand-up comedy quite yet.

How It Works

We first set up our arrays. One is a list of words and so we can use `qw//` to specify it. The other is a list of strings, so we use the ordinary list style:

```
my @questions = qw(Java Python Perl C);
my @punchlines = (
      "None. Change it once, and it's the same everywhere.",
      "One. He just stands below the socket and the world revolves around him.",
      "A million. One to change it, the rest to try and do it in fewer lines.",
      '"CHANGE?!!"'
);
```

We now ask the user to choose their joke:

```
print "Please enter a number between 1 and 4: ";
my $selection = <STDIN>;
$selection -= 1;
```

Why take one from it? Well, we've asked for a number between one and four, and our array subscripts go from zero to three.

Next we display the set-up line:

```
print "How many $questions[$selection] ";
print "programmers does it take to change a lightbulb?\n\n";
```

From the first line, we see that array elements stringify just like scalar variables. Next, this new function `sleep`:

```
sleep 2;
```

What `sleep` does, as you'll know if you've run the program, is pause the program's operation for a number of seconds. In this case, we're telling it to sleep for two seconds.

After the user has had time to think about it, we display the punchline:

```
print $punchlines[$selection], "\n";
```

Hopefully, you're starting to see alternative ways we can use arrays by now. Of course, we've only been pulling single values from arrays so far. The next logical step is to start working with multiple array elements.

Accessing Multiple Elements

If you'll recall, we created and used a list slice by putting ranges or several numbers in brackets to get multiple elements from a list. If we want to get multiple elements from an array, we can use the analogous concept, an **array slice**.

List slices, if you remember, looked like this:

```
(qw(Jan Feb Mar May Apr Jun Jul Aug Sep Oct Nov Dec))[3,5,7..9]
```

Can you work out which elements the slice above consists of? If not, write a short Perl program to print them out, and see if you can get it to separate them with spaces. (Hint: Only arrays stringify with spaces, so you'll need to use one.)

Array slices look very similar. However, now that we are accessing multiple elements and expecting a list, we no longer want to use `$` as the prefix – now we should be using `@`.

We can get the same list as the above like this:

```
my @array = qw(Jan Feb Mar May Apr Jun Jul Aug Sep Oct Nov Dec);
print @array[3,5,7..9];
```

Array slices act like any normal list, and so can be used as an lvalue. Here's a load of slices to mess around with:

Try It Out - Array Slices

Here are a year's sales results for a fictitious bathroom tile shop:

```
#!/usr/bin/perl
# aslice.plx
use warnings;
use strict;

my @sales = (69, 118, 97, 110, 103, 101, 108, 105, 76, 111, 118, 101);
my @months = qw(Jan Feb Mar May Apr Jun Jul Aug Sep Oct Nov Dec);

print "Second quarter sales:\n";
print "@months[3..5]\n@sales[3..5]\n";
my @q2=@sales[3..5];

# Incorrect results in May, August, Oct, Nov and Dec!
@sales[4, 7, 9..11] = (68, 101, 114, 111, 117);

# Swap April and May
@months[3,4] = @months[4,3];
```

Most of the work is behind the scenes, but this is what you'd see if you run it:

```
Second quarter sales:
May Apr Jun
110 103 101
```

Let's take a look at what's actually going on.

How It Works

We set up our two arrays – one holding our sales figures, and the other holding the names of the months:

```
my @sales = (69, 118, 97, 110, 103, 101, 108, 105, 76, 111, 118, 101);
my @months = qw(Jan Feb Mar May Apr Jun Jul Aug Sep Oct Nov Dec);
```

To extract the information about the second quarter, we use an array slice for the months in question:

```
print "Second quarter sales:\n";
print "@months[3..5]\n@sales[3..5]\n";
my @q2=@sales[3..5];
```

In addition to saving the relevant elements to another array, we can also print out the slice and it will be stringified. We can also assign values to an array slice, as well as getting data from it:

```
@sales[4, 7, 9..11] = (68, 101, 114, 111, 117);
```

This sets new values for `$sales[4]`, `$sales[7]`, `$sales[9]`, `$sales[10]` and `$sales[11]`.

Finally, we can use something similar to the `($a, $b) = ($b, $a)` list trick to swap two array elements:

```
@months[3,4] = @months[4,3];
```

This is exactly the same as the following statement:

```
($months[3], $months[4]) = ($months[4], $months[3]);
```

As you can see, this isn't all that far from the list assignment to swap two variables:

```
($mone, $mtwo) = ($mtwo, $mone);
```

Watch your parentheses and square brackets, though.

Running through Arrays

One thing we'll want to do quite often is run over each of the elements in an array or list in turn. If we want to double every value in an array, then **for** each element we come across, we multiply by two. The keyword to use here is `for`. Here's a **for loop**, which prints each element of an array, followed by a new line:

```
#!/usr/bin/perl
# forloop1.plx
use warnings;
use strict;

my @array = qw(America Asia Europe Africa);
my $element;
for $element (@array) {
     print $element, "\n";
}
```

We set up an array, and we declare another scalar variable, `$element`. What we then say is 'set each element of `@array` to `$element` in turn, and then do all the statements in the following block'. So, on our first iteration, `$element` is set to `America`, and then the `print` statement is run. Then `$element` is set to `Asia`, and the `print` statement runs again. This continues until the end of the array is reached.

This should print:

```
>perl forloop1.plx
America
Asia
Europe
Africa
>
```

`$element` is called an **iterator variable** or **loop variable**. It's what we 'see' when we look at each element in turn. This is the syntax of the for loop:

```
for <ITERATOR> (<LIST OR ARRAY>) <BLOCK>
```

The block must start with an opening brace and end with a closing brace, and the list or array that we're running over must be surrounded by parentheses. If we don't supply an iterator variable of our own, perl uses the special $_ variable, which is often used in Perl functions as a 'default value'. Note that the for loop doesn't require a semicolon after the block.

So, when processing a for loop, perl makes the iterator a copy of each element of the list or array in turn, and then runs the block. If the block happens to change the value of the iterator, the corresponding array element changes as well. We can double each element of an array like this:

```
#!/usr/bin/perl
# forloop2.plx
use warnings;
use strict;

my @array=(10, 20, 30, 40);
print "Before: @array\n";
for (@array) { $_ *= 2 }
print "After: @array\n";
```

This prints:

```
>perl forloop2.plx
Before: 10 20 30 40
After: 20 40 60 80
>
```

If you need to know the number of the element you're currently processing, it's usually best to have the iterator as the range of numbers you're processing – from 0 up to the highest element number in the array. Let's rewrite the joke machine so that it tells *all* the bad jokes, without prompting:

Try It Out – Joke Machine II – The Revenge

Here we use the same jokes tell each of them in turn:

```
#!/usr/bin/perl
# joke2.plx
use warnings;
use strict;

my @questions = qw(Java Python Perl C);
my @punchlines = (
      "None. Change it once, and it's the same everywhere.",
      "One. He just stands below the socket and the world revolves around him.",
      "A million. One to change it, the rest to try and do it in fewer lines.",
      '"CHANGE?!!"'
);

for (0..$#questions) {
      print "How many $questions[$_] ";
      print "programmers does it take to change a lightbulb?\n";
      sleep 2;
      print $punchlines[$_], "\n\n";
      sleep 1;
}
```

The changes to our old `joke1.plx` program produce this result:

```
> perl joke2.plx
How many Java programmers does it take to change a lightbulb?
None. Change it once, and it's the same everywhere.

How many Python programmers does it take to change a lightbulb?
One. He just stands below the socket and the world revolves around him.

How many Perl programmers does it take to change a lightbulb?
A million. One to change it, the rest to try and do it in fewer lines.

How many C programmers does it take to change a lightbulb?
"CHANGE?!!"

>
```

I promise I'll keep my day-job....

How It Works

The for loop is now the main part of our program. Let's have a look at it again:

```
for (0..$#questions) {
     print "How many $questions[$_] ";
     print "programmers does it take to change a lightbulb?\n";
     sleep 2;
     print $punchlines[$_], "\n\n";
     sleep 1;
}
```

The key thing about this example is that we need to match the questions to the punchlines. This means we can't just go through one or the other of the arrays, but we have to go through them both together. We do this by using a list, which counts up from 0 to the highest element of one of the arrays. Since the arrays are both the same size, it doesn't matter which one. The line that does this is:

```
for (0..$#questions) {
```

`$#questions` is the index of the highest element in the `@questions` array. That's different from the value we get when we look at `@questions` in a scalar context. Look:

```
#!/usr/bin/perl
# elems.plx
use warnings;
use strict;

my @array = qw(alpha bravo charlie delta);

print "Scalar value   : ", scalar @array, "\n";
print "Highest element: ", $#array, "\n";
```

```
>perl elems.plx
Scalar value     : 4
Highest element: 3
>
```

Why? There are four elements in the array – so that's the scalar value. Their indices are 0, 1, 2, and 3. Since we're starting at zero, the highest element (`$#array`) will always be one less than the number of elements in the array.

So, we count up from 0 to the index of the highest element in `@questions`, which happens to be 3. We set the iterator to each number in turn. Where's the iterator? Since we didn't give one, perl will use `$_`. Now we do the block four times, once when `$_` is 0, once when it is 1, and so on:

```
        print "How many $questions[$_] ";
```

This line prints the zeroth element of `@questions` the first time around, then the first, then the second, third, and fourth:

```
        print $punchlines[$_], "\n\n";
```

And so it is with the punchlines. If we'd just said:

```
for (@questions) {
```

`$_` would have been set to each question in turn, but we would not have advanced our way through the answers.

Array Functions

It's time we met some more of the things we can do with arrays. These are variously called **array functions** and **array operators**. As mentioned previously, perl doesn't draw much distinction between functions and operators. The important part is that they all do some kind of work on an array. We've already met one of them: `reverse`, which we used to count down ranges instead of counting up. We can use `reverse` on arrays as well as lists:

```
#!/usr/bin/perl
# countdown.plx
use warnings;
use strict;

my @count = (1..5);
for (reverse(@count)) {
     print "$_...\n";
     sleep 1;
}

print "BLAST OFF!\n";
```

Hopefully, you should have a good idea of what this will print out before you run it.

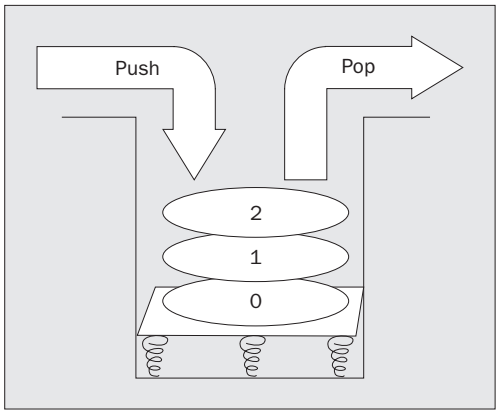
```
>perl countdown.plx
5...
4...
3...
2...
1...
BLAST OFF!
>
```

Now a while back I mentioned some useful functions for adding elements to arrays. Here they are now, along with a couple of other useful tips and tricks.

Pop and Push

Now we've already seen a simple way to add elements to an array: `@array = (@array, $scalar)`.

One of the original metaphors that computer programmers like to use to analyze arrays is a **stack** of spring-loaded plates in a canteen. You push down when you put another plate on the top, and the stack pops up when a plate is taken away:



Following this metaphor, `push` is the operator that adds an element, or list of elements, to the end of an array. Similarly, to remove the top element – the element with the highest index, we use the `pop` operator:

Try It Out – Paper Stacks

Stacks are all around us. In my case, they're all stacks of paper. We can manipulate arrays just as we can manipulate these stacks of paper:

```
#!/usr/bin/perl
# stacks.plx
use warnings;
use strict;

my $hand;
my @pileofpaper = ("letter", "newspaper", "gas bill", "notepad");
```

```
print "Here's what's on the desk: @pileofpaper\n";

print "You pick up something off the top of the pile.\n";
$hand = pop @pileofpaper;
print "You have now a $hand in your hand.\n";

print "You put the $hand away, and pick up something else.\n";
$hand = pop @pileofpaper;
print "You picked up a $hand.\n";

print "Left on the desk is: @pileofpaper\n";

print "You pick up the next thing, and throw it away.\n";
pop @pileofpaper;

print "You put the $hand back on the pile.\n";
push @pileofpaper, $hand;

print "You also put a leaflet and a bank statement on the pile.\n";
push @pileofpaper, "leaflet", "bank statement";

print "Left on the desk is: @pileofpaper\n";
```

Watch what happens:

```
>perl stacks.plx
Here's what's on the desk: letter newspaper gas bill notepad
You pick up something off the top of the pile.
You have now a notepad in your hand.
You put the notepad away, and pick up something else.
You picked up a gas bill.
Left on the desk is: letter newspaper
You pick up the next thing, and throw it away.
You put the gas bill back on the pile.
You also put a leaflet and a bank statement on the pile.
Left of the desk is: letter gas bill leaflet bank statement
>
```

How It Works

Let's take this play-by-play. First off, we initialize our `$hand` and our `@pileofpaper`. Since the pile of paper is a stack, the zeroth element (the letter), is at the bottom, and the notepad is at the top:

```
my $hand;
my @pileofpaper = ("letter", "newspaper", "gas bill", "notepad");
```

We use `pop @array` to remove the top element from the array and it returns that element, which we store in `$hand`. So, we take the notepad from the stack and put it into our hand. What's left? The letter at the bottom of the stack, then the newspaper and gas bill:

```
print "You pick up something off the top of the pile.\n";
$hand = pop @pileofpaper;
print "You have now a $hand in your hand.\n";
```

As we pop again, we take the next element (the gas bill) off the top of the stack and store it again in $hand. Since we didn't save the notepad from last time, it's lost forever now:

```
print "You put the $hand away, and pick up something else.\n";
$hand = pop @pileofpaper;
print "You picked up a $hand.\n";
```

The next item is the newspaper. We pop this as before, but we never store it anywhere:

```
print "You pick up the next thing, and throw it away.\n";
pop @pileofpaper;
```

We've still got the gas bill in $hand from previously. push @array, $scalar will add the scalar onto the top of the stack. In our case, we're putting the gas bill on top of the letter:

```
print "You put the $hand back on the pile.\n";
push @pileofpaper, $hand;
```

push can also be used to add a list of scalars onto the stack – in this case, we've added two more strings. We could add the contents of an array to the top of the stack with push @array1, @array2. So we now know that we can replace a list with an array:

```
print "You also put a leaflet and a bank statement on the pile.\n";
push @pileofpaper, "leaflet", "bank statement";
```

As you might suspect, you can also push lists of lists onto an array: They simply get flattened first into a single list and then added.

Shift and Unshift

While the functions push and pop deal with the 'top end' of the stack, adding and taking away elements from the highest index of the array – the functions unshift and shift do the corresponding jobs for a similar job for the bottom end:

```
#!/usr/bin/perl
#shift.plx
use warnings;
use strict;

my @array = ();
unshift(@array, "first");
print "Array is now: @array\n";
unshift @array, "second", "third";
print "Array is now: @array\n";
shift @array ;
print "Array is now: @array\n";
```

```
>perl shift.plx
Array is now: first
Array is now: second third first
Array is now: third first
>
```

First we `unshift()` an element onto the array, and the element appears at the beginning of the list. It's not easy to see this since there are no other elements, but it does. We then `unshift` two more elements. Notice that the entire list is added to the beginning of the array all at once, rather than one element at a time. We then use `shift` to take off the first element, ignoring what it was.

Sort

One last thing you may want to do while processing data is put it in alphabetical or numeric order. The `sort` operator takes a list and returns a sorted version:

```
#!/usr/bin/perl
#sort1.plx
use warnings;
use strict;

my @unsorted = qw(Cohen Clapton Costello Cream Cocteau);
print "Unsorted: @unsorted\n";
my @sorted = sort @unsorted;
print "Sorted:   @sorted\n";
```

```
>perl sort1.plx
Unsorted: Cohen Clapton Costello Cream Cocteau
Sorted:     Clapton Cocteau Cohen Costello Cream
>
```

This is only good for strings and alphabetic sorting. If you're sorting numbers, there is a problem. Can you guess what it is? This may help:

```
#!/usr/bin/perl
#sort2.plx
use warnings;
use strict;

my @unsorted = (1, 2, 11, 24, 3, 36, 40, 4);
my @sorted = sort @unsorted;
print "Sorted:   @sorted\n";
```

```
>perl sort2.plx
Sorted:   1 11 2 24 3 36 4 40
>
```

What? 11 doesn't come between 1 and 2. What we need to do is compare the numeric values instead of the string ones. Cast your mind back to last chapter and recall how to compare two numeric variables, `$a` and `$b`. Here, we're going to use the `<=>` operator. `sort` allows us to give it a block to describe how two values should be ordered, and we do this by comparing `$a` and `$b`.These two variables are given to us by the `sort` function:

```
#!/usr/bin/perl
#sort3.plx
use warnings;
use strict;
my @unsorted = (1, 2, 11, 24, 3, 36, 40, 4);
```

```
my @string = sort { $a cmp $b } @unsorted;
print "String sort:  @string\n";

my @number = sort { $a <=> $b } @unsorted;
print "Numeric sort:  @number\n";
```

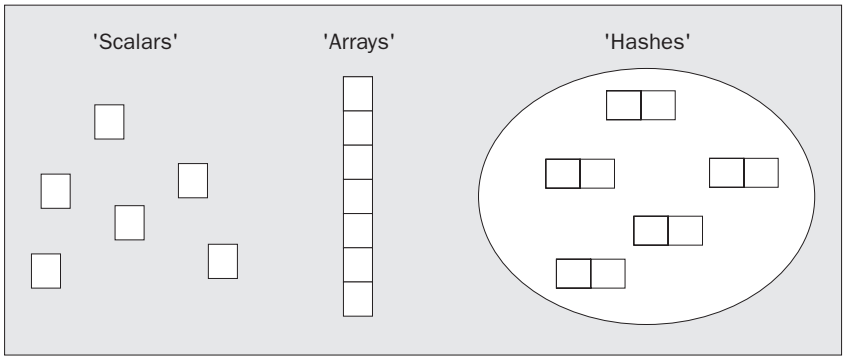
```
>perl sort3.plx
String sort:  1 11 2 24 3 36 4 40
Numeric sort:  1 2 3 4 11 24 36 40
>
```

Another good reason for using string comparison operators for strings and numeric comparison operators for numbers!

Hashes

The final variable type we have is the hash. In the introduction, I said that the hash was like a dictionary or a phone book, but that's not quite right. There is a slight difference in that a phone book is normally ordered – the names are sorted alphabetically. In a hash the data is totally unsorted and has no intrinsic order. In fact, it's more like directory enquiries than a phone book in that you can easily find out what the number is if you have the name. Someone else keeps the order for you, and you needn't ask what the first entry is.

Here's where a diagram helps:



A scalar is one piece of data; it's like a single block. An array or a list is like a tower of blocks: it's kept in order, and it's kept together as a single unit. A hash, on the other hand, is more like the diagram on above. It contains several pairs of data. The pairs are in no particular order (no pair is 'first' or 'top'), and they're all scattered around the hash.

Creating a Hash

A hash looks very similar to a list, and it also behaves very much like a list. It's only actually effective as a hash when you store it in a hash variable. Just like scalar variables have a `$` prefix, and arrays have a `@` prefix, hashes have their own prefix – a percent sign `%`. Again, the same naming rules apply, and the variables `%hash`, `$hash`, and `@hash` are all different.

There are two ways of writing a hash. First, just like an ordinary list of pairs:

```
%where=(
        "Gary"     , "Dallas",
        "Lucy"     , "Exeter",
        "Ian"      , "Reading",
        "Samantha" , "Oregon"
);
```

In this case, the hash could be saying that "Gary's whereabouts is Dallas", "Lucy lives in Exeter" and so on. All it really does is pair Gary and Dallas, Lucy and Exeter, and so on. How the pairing is interpreted is up to you.

If we want to make the relationship a little clearer, as well as highlighting the fact that we're dealing with a hash, we can use the `=>` operator. That's not `>=`, which is greater-than-or-equal-to; the `=>` operator acts like a 'quoting comma'. Essentially, it's a comma, but whatever appears on the left hand side of it – and only the left – is treated as a double-quoted string:

```
%where=(
        Gary     => "Dallas",
        Lucy     => "Exeter",
        Ian      => "Reading",
        Samantha => "Oregon"
);
```

The scalars on the left of the arrow are called the **hash keys**, the scalars on the right are the **values**. We use the keys to look up the values:

> **Hash keys must be unique. You cannot have more than one entry for the same name, and if you try to add a new entry with the same key as an existing entry, the old one will be over-written. Hash values meanwhile need not be unique.**

Key uniqueness is more of an advantage than a limitation. Every time the word 'unique' comes into a problem, like counting the unique elements of an array, your mind should immediately echo 'use a hash!'

Because hashes and arrays are both built from structures that look like lists, you can convert between them, from array to hash like this:

```
@array = qw(Gary Dallas Lucy Exeter Ian Reading Samantha Oregon);
%where = @array;
```

And then back to an array, like so:

```
@array = %where;
```

However, you need to be careful when converting back from a hash to an array. Hashes do not have a guaranteed order. Although values will always follow keys, you cannot tell what order the keys will come in. Since hash keys are unique, however, we can be sure that `%hash1 = %hash2` will copy a hash accurately.

If you need to turn your hash around, to look up people by location, you can use this list-like structure to your advantage... just `reverse` the list. Be careful though – if you have two values that are the same, then converting them to keys means that one will be lost. Remember that keys must be unique:

```
@array = qw(Gary Dallas Lucy Exeter Ian Reading Samantha Oregon);
%where = @array;
```

`%where` now holds the same value as if the following call had been made:

```
%where=(
        Gary     => "Dallas",
        Lucy     => "Exeter",
        Ian      => "Reading",
        Samantha => "Oregon"
);
```

Likewise, `%who` will hold the same values no matter which of the two calls below were made:

```
%who = reverse @array;
%who = (
     Oregon  => "Samantha",
     Reading => "Ian",
     Exeter  => "Lucy",
     Dallas  => "Gary"
);
```

Working with Hash Values

To look up a value in a hash, we use something similar to the index notation for arrays. However, instead of locating elements by number, we're now locating them by name; instead of using square brackets, we use braces (curly brackets):

Try It Out – Using Hashes

Here's a simple example of looking up details in a hash:

```
#!/usr/bin/perl
#hash1.plx
use warnings;
use strict;

my $place = "Oregon";
my %where=(
        Gary     => "Dallas",
        Lucy     => "Exeter",
        Ian      => "Reading",
        Samantha => "Oregon"
);
my %who = reverse %where;

print "Gary lives in ", $where{Gary}, "\n";
print "Ian lives in $where{Ian}\n";
print "$who{Exeter} lives in Exeter\n";
print "$who{$place} lives in $place\n";
```

```
> perl hash1.plx
Gary lives in Dallas
Ian lives in Reading
Lucy lives in Exeter
Samantha lives in Oregon
>
```

How It Works

First, we set up our main hash, which tells us where people live:

```
my %where=(
        Gary     => "Dallas",
        Lucy     => "Exeter",
        Ian      => "Reading",
        Samantha => "Oregon"
);
```

By reversing the order of the list, we produce a hash that tells us who lives where:

```
my %who = reverse %where;
```

When doing this you need to be careful, as I have already mentioned. You must not have two values the same, since they will need to become keys, and keys must be unique – one or other of them will get lost.

Now we can look up an entry in our hashes – we'll ask 'Where does Gary live?':

```
print "Gary lives in ", $where{Gary}, "\n";
```

This is almost identical to looking up an array element, except for the brackets and the fact that we are now allowed to use strings as well as numbers to index our elements.

```
print "Ian lives in $where{Ian}\n";
print "$who{Exeter} lives in Exeter\n";
```

The braces of a hash look-up can also quote what is inside them in double quotes if we do not provide the quotes ourselves:

```
print "$who{$place} lives in $place\n";
```

Just as with array elements, we need not use a literal to index the element – we can look-up using a variable as well.

Adding, Changing, and Taking Values Away from a Hash

Hash entries are very much like ordinary scalar variables, except that you need not declare an individual hash key before assigning to it or using it. We can add a new person to our hash just by assigning to their hash entry:

```
$where{Eva} = "Uxbridge";
print "Eva lives in $where{Eva}\n";
```

A new entry springs into existence, without any problems. We can also change the entries in a hash just by reassigning to them. Let's move people around a little:

```
$where{Eva}      = "Denver";
$where{Samantha} = "California";
$where{Lucy}     = "Tokyo";
$where{Gary}     = "Las Vegas";
$where{Ian}      = "Southampton";

print "Gary lives in $where{Gary}\n";
```

To remove an entry from a hash, you need to use the `delete()` function, as we do in this little variant on `hash1.plx`:

```
#!/usr/bin/perl
#badhash1.plx
use warnings;
use strict;

my %where=(
        Gary     => "Dallas",
        Lucy     => "Exeter",
        Ian      => "Reading",
        Samantha => "Oregon"
);

delete $where{Lucy};
print "Lucy lives in $where{Lucy}\n";
```

Now here we delete Lucy's entry in `%where` before we access it. So after running it, we should get an error. Sure enough, we get:

```
> perl badhash1.plx
Use of uninitialized value in concatenation (.) at badhash1.plx line 11
Lucy lives in Exeter
>
```

It's not that we haven't initialized poor Lucy, but rather that we've decided to get rid of her.

Accessing Multiple Values

The problem with hashes looking like lists is that we can't really use for loops on them directly. If we did, we would get both keys and values with no indication as to which was which. To help us, Perl provides three functions for iterating over hashes.

First, there is `keys (%hash)`. This gives us a list of the keys (all of the scalars on the left-hand side). This is usually what we want when we wish to visit each hash entry in turn:

Try It Out – Looping Over A Hash

Let's leave the computer to run over hash and tell us where each person lives:

```
#!/usr/bin/perl
#hash2.plx
use warnings;
use strict;

my %where=(
        Gary     => "Dallas",
        Lucy     => "Exeter",
        Ian      => "Reading",
        Samantha => "Oregon"
);

for (keys %where) {
     print "$_ lives in $where{$_}\n";
}
```

Currently, this tells me:

```
>perl hash2.plx
Samantha lives in Oregon
Gary lives in Dallas
Lucy lives in Exeter
Ian lives in Reading
>
```

You may find that the output appears in a different order on your machine. Don't worry, as I said, hashes are unordered, and there's no guarantee that the keys will come out in the same order each time. It really depends on the particular version of Perl that you are using.

How It Works

Here is the part that does all the work:

```
for (keys %where) {
     print "$_ lives in $where{$_}\n";
}
```

`keys` is a function which, like `sort` and `reverse`, returns a list. The list in my case was `qw(Samantha Gary Lucy Ian)`, and `for` visited each of those values in turn. As `$_` was set to each one, we could print the name and look up that entry in the hash.

The counterpart to `keys` is `values`, which returns a list of all of the values in the hash. This is somewhat less useful, since you can always find the value if you have the key, but you cannot easily find the key if you have the value. It's almost always advantageous to use `keys` instead.

The final function is `each`, which we will look at later. It returns each hash entry as a key-value pair.

Summary

Lists are a series of scalars in order. Arrays are variable incarnations of lists. Both lists and arrays are flattened, so we cannot yet have a distinct list inside another list. We get at both lists and arrays with square-bracket subscripts. These can be single numbers, or a list of elements. If we're looking up a single scalar in an array, we need to remember to use the form `$array[$element]`, because the variable prefix always refers to what we want, not what we have got. We can also use ranges to save time and to specify list and array slices.

Perl differentiates between scalar and list context and returns different values depending on what the statement is expecting to see. For instance, the scalar context value of an array is the number of elements in it; the list context value is of course the list of the elements themselves.

Hashes are unordered structures made up of pairs, each pair consisting of a key and a value. Given the key, we can look up the entry. Generally, `$hash{$key} = $value`. We can loop over all the elements of a list or array using a for loop. We need to modify this when looping over two lists at once or when looking for the keys or values of a hash.

Exercises

1. When you assign to a list, the elements are copied over from the right to the left:

```
($a, $b) = ( 10, 20 );
```

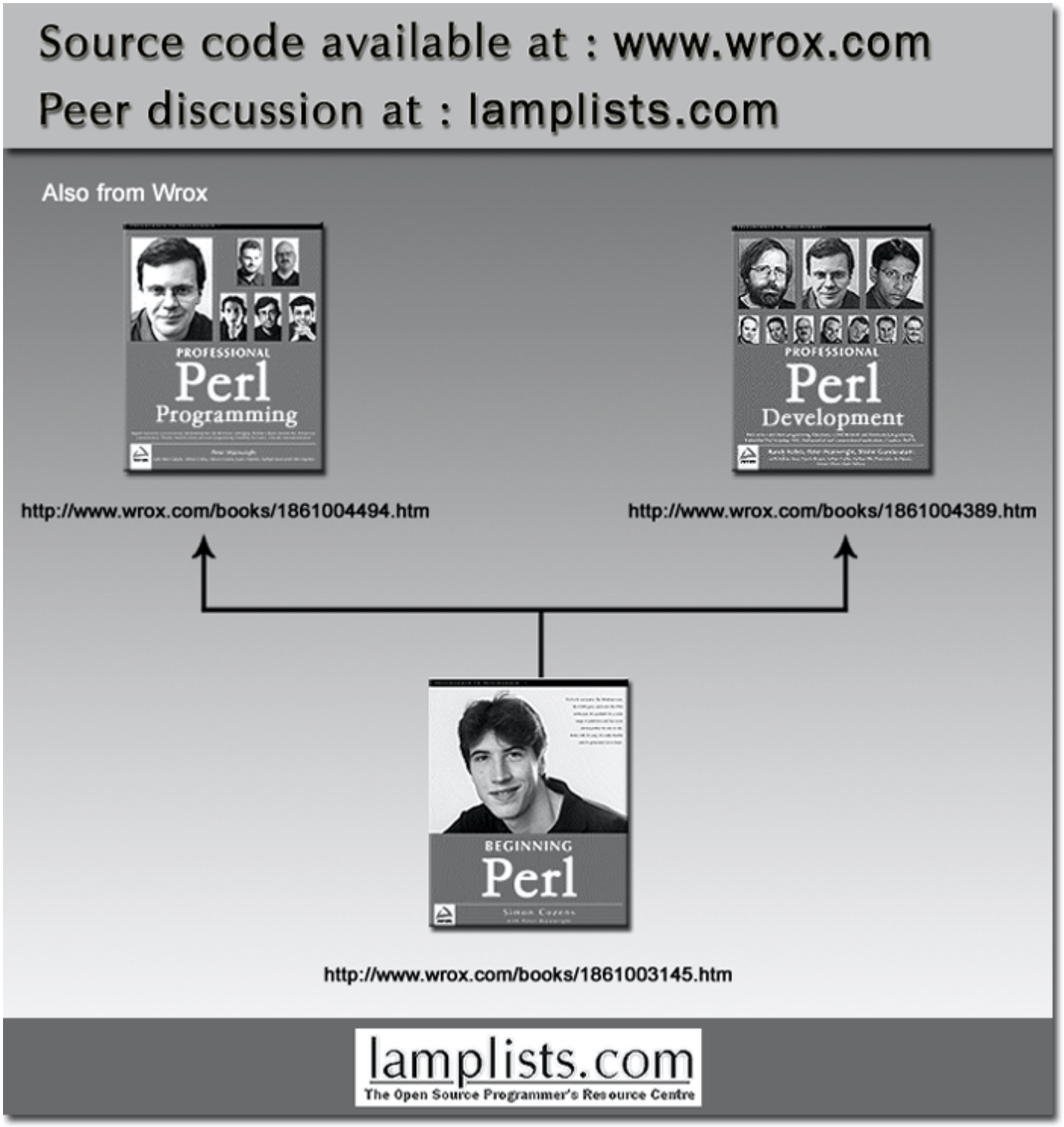
   will make `$a` become 10 and `$b` become 20. Investigate what happens when:

   - There are more elements on the right than on the left.
   - There are more elements on the left than on the right.
   - There is a list on the left but a single scalar on the right.
   - There is a single scalar on the left but a list on the right.

2. What elements make up the range `('aa' .. 'bb')`? What about `('a0' .. 'b9')`?

3. Store your important phone numbers in a hash. Write a program to look up numbers by the person's name.

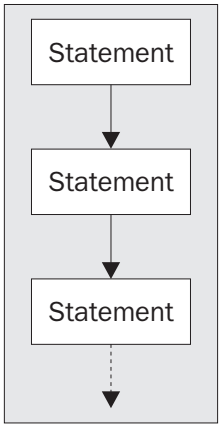
4. Turn the joke machine program from two arrays into one hash. While doing so, write some better lightbulb jokes.



4

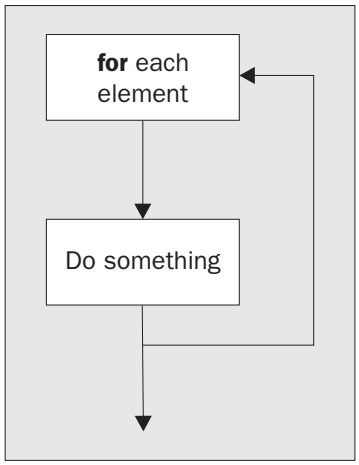
Loops and Decisions

Most of the programs so far have been very simply structured – they've done one statement after another in turn. If we were to represent statements as boxes, our programs would look like this:

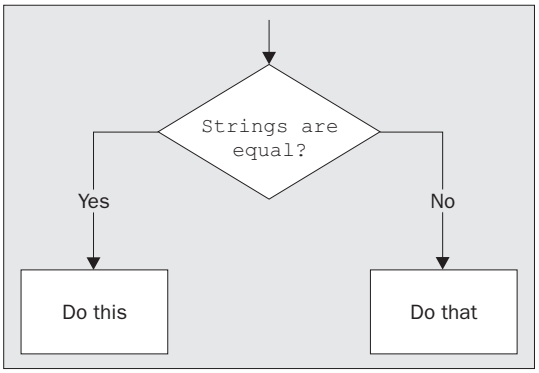


This sort of diagram is called a **flow chart**, and programmers have used them for a long time to help design their programs. They're considered a bit passé these days, but they're still useful. The path Perl (or any other language) takes by following the arrows is called the **flow of execution** of the program. Boxes denote statements (or a single group of statements), and diamonds denote tests. There are also a whole host of other symbols for magnetic tapes, drum storage, and all sorts of wonderful devices, now happily lost in the mists of time.

In the last chapter, we introduced the **for loop** to our growing repertoire of programming tools. Instead of taking a straight line through our program, perl did a block of statements again and again for each element of the list. The for loop is a **control structure** – it controls the flow of execution. Instead of going in a straight line, we can now go around in a loop:



Not only that, but we can choose our path through the program depending on certain things, like the comparisons we looked at in Chapter 2. For instance, we'll do something **if** two strings are equal:



We'll take a look at the other sorts of control structures we have in Perl, for example, structures that do things **if** or **unless** something is true. We'll see structures that do things **while** something is true or **until** it is true. Structures that loop **for** a certain number of times, or **for each** element in a list. Each of the words in bold is a Perl keyword, and we'll examine each of them in this chapter.

Deciding If...

Let's first extend our previous currency conversion program a little, using what we learned about hashes from the previous chapter.

Try It Out : Multiple Currency Converter

We'll use a hash to store the exchange rates for several countries. Our program will ask for a currency to convert from, then a currency to convert to, and the amount of currency we would like to convert:

```
#!/usr/bin/perl
# convert1.plx
use warnings;
use strict;

my ($value, $from, $to, $rate, %rates);
%rates = (
    pounds          => 1,
    dollars         => 1.6,
    marks           => 3.0,
    "french francs" => 10.0,
    yen             => 174.8,
    "swiss francs"  => 2.43,
    drachma         => 492.3,
    euro            => 1.5
);

print "Enter your starting currency: ";
$from = <STDIN>;
print "Enter your target currency: ";
$to = <STDIN>;
print "Enter your amount: ";
$value = <STDIN>;

chomp($from,$to,$value);
$rate = $rates{$to} / $rates{$from};

print "$value $from is ",$value*$rate," $to.\n";
```

Here's a sample run of this program:

```
> perl convert1.plx
Enter your starting currency: dollars
Enter your target currency: euro
Enter your amount: 200
200 dollars is 187.5 euro.
>
```

Let's first see how this all works, and then we'll see what's wrong with it.

How It Works

The first thing we do is to declare all our variables. You don't have to do this at the start of the program, but it helps us organize:

```
my ($value, $from, $to, $rate, %rates);
```

Note that you **do** need to put brackets when you're declaring more than one variable at once. Once again it's a question of precedence – `my` has a lower precedence than a comma. Now we can set out our rates table:

```
%rates = (
    pounds          => 1,
    dollars         => 1.6,
    marks           => 3.0,
    "french francs" => 10.0,
    yen             => 174.8,
    "swiss francs"  => 2.43,
    drachma         => 492.3,
    euro            => 1.5
);
```

Using `<STDIN>` as we did in the last chapter to read a line from the console, we'll ask for two currencies and the amount:

```
print "Enter your starting currency: ";
$from = <STDIN>;
print "Enter your target currency: ";
$to = <STDIN>;
print "Enter your amount: ";
$value = <STDIN>;
```

Now we have a problem. We read in entire lines, and lines end with a new line character. In order for us to look up the currencies in the hash and to display the currencies back properly, we have to get rid of this. The way we do this is to use the `chomp` operator on the values we've just read in – `chomp` gets rid of a final new line if one is present but does nothing if there is no new line. For instance, it turns `"euro\n"`, which isn't in the hash, into `"euro"`, which is:

```
chomp($from,$to,$value);
```

Note that it actually changes the value of the variables passed to it. Instead of returning the new string, it modifies the variable, and actually returns the number of new lines removed. Now we don't actually *need* to remove the new line from `$value`. All we do is use it in calculation and perl will convert it to a number. When we do that the new line will automatically be lost. However, since we might want to print out, '200 marks', we need to make sure there is no new line after '200'.

Next we calculate the exchange rate, which is just the rate of the target currency divided by the rate of the initial currency:

```
$rate = $rates{$to} / $rates{$from};
```

Finally, we multiply the value by the exchange rate and print out the results:

```
print "$value $from is ",$value*$rate," $to.\n";
```

Now, this is all well and good, but watch what happens if one of the currencies we ask for isn't in the hash:

```
> perl convert1.plx
Enter your starting currency: dollars
Enter your target currency: lira
Enter your amount: 300
Use of uninitialized value in division (/) at convert1.plx line 26, <STDIN> line 3.
300 dollars is 0 lira.
```

What was that warning? Well, the message tells us that something we used during the division at line 26:

```
$rate = $rates{$to} / $rates{$from};
```

was not defined. We know in this case it's the target currency. `$rates{lira}` is not in the hash. When the other currency is undefined, then we get more serious problems:

```
> perl convert1.plx
Enter your starting currency: lira
Enter your target currency: dollars
Enter your amount: 132000
Use of uninitialized value in division (/) at convert1.plx line 26, <STDIN> line 3.
Illegal division by zero at convert.plx line 26, <STDIN> line 3.
```

This time the other side of the division is undefined, and Perl converts the undefined value to zero. Unfortunately, you can't divide by zero. To solve both these problems we really want to be able to stop the program when an unknown currency is entered – that is, if a certain string does not exist in the hash.

We now need to find out if something happened, and perform a certain action if it did. This expression is called an **if statement**. Here's what an if statement looks like in Perl:

```
if ( <some test> ) {
   <do something>
}
```

In our case, we want to ensure a hash key exists. Now Perl isn't a difficult language; to sort a list, you use the `sort` keyword, to find the length of a string, you use the `length` keyword. To see if a hash key exists, we use the aptly named `exists` keyword for our test – `exists $rates{$to}` and `exists $rates{$from}`:

Try It Out : Testing Invalid Keys

Let's now put the `if` statement to use and test to make sure we are given valid, existing keys:

```
#!/usr/bin/perl
# convert2.plx
use warnings;
use strict;
```

```
my ($value, $from, $to, $rate, %rates);
%rates = (
    pounds          => 1,
    dollars         => 1.6,
    marks           => 3.0,
    "french francs" => 10.0,
    yen             => 174.8,
    "swiss francs"  => 2.43,
    drachma         => 492.3,
    euro            => 1.5
);

print "Enter your starting currency: ";
$from = <STDIN>;
print "Enter your target currency: ";
$to = <STDIN>;
print "Enter your amount: ";
$value = <STDIN>;

chomp($from,$to,$value);

if (not exists $rates{$to}) {
   die "I don't know anything about $to as a currency\n";
}
if (not exists $rates{$from}) {
   die "I don't know anything about $from as a currency\n";
}

$rate = $rates{$to} / $rates{$from};

print "$value $from is ",$value*$rate," $to.\n";
```

Now if we enter a currency that is unknown, we get our own error message and the program ends:

```
> perl convert2.plx
Enter your starting currency: dollars
Enter your target currency: lira
Enter your amount: 300
I don't know anything about lira as a currency
>
```

How It Works

After we've got the currency names, and before we try to divide, we use the following code to see if the currencies are valid entries in the hash. We do two very similar comparisons, one for the start currency and one for the target, so let's just examine one of them:

```
if (not exists $rates{$to}) {
   die "I don't know anything about $to currency\n";
}
```

This is our `if` statement: if the entry `$to` does not exist in the `%rates` hash, then we give an error message. `die` is a way of making Perl print out an error message and finish the program. It also reports to the operating system – Windows, Unix, or whatever it may be, that the program finished with an error. The part in brackets, `not exists $rates{$to}` is known as the **condition**. If that condition is true, we do the action in braces and terminate the program.

How do we construct conditions, then?

Logical Operators Revisited

The `if` statement, and all the other control structures we're going to visit in this chapter, test to see if a condition is true or false. They do this using the Boolean logic mentioned in Chapter 2, together with Perl's ideas of true and false. To remind you of these:

- An empty string, `""`, is false.
- The number zero and the string `"0"` are both false.
- An empty list, `()`, is false.
- The undefined value is false.
- Everything else is true.

However, you need to be careful for a few traps here. A string containing invisible characters, like spaces or new lines, is true. A string that isn't `"0"` is true, even if its numerical value is zero, so `"0.0"` for instance, is true.

Larry Wall has said that programming Perl is an empirical science – you learn things about it by trying it out. Is `(())` a true value? You can look it up in books and the online documentation, or you can spend a few seconds writing a program like this:

```
#!/usr/bin/perl
use strict;
use warnings;

if ( (()) ) {
   print "Yes, it is.\n";
}
```

This way you get the answer right away, with a minimum of fuss. (If you're interested, it isn't a true value.) We'll see in later chapters how to make this sort of test program easier and faster to write, but what we know now is sufficient to test the hypothesis. I'm continually writing these little programs to check out facets of Perl I'm not sure about. Try getting into the habit of doing it, too.

We've also seen that conditional operators can test things out, returning `1` if the test was successful and the undefined value if it was not. Let's see more of the things we can test.

Comparing Numbers

We can test whether one number is bigger, smaller, or the same as another. Assuming we have two numbers, stored in the variables `$a` and `$b`, here are the operators we can use for this:

`$a > $b`	`$a` is greater than `$b`
`$a < $b`	`$a` is less than `$b`
`$a == $b`	`$a` has the same numeric value as `$b`
`$a != $b`	`$a` does not have the same numeric value as `$b`

Don't forget that the numeric comparison needs a doubled equals sign (==), so that Perl doesn't think you're trying to set `$a` to the value of `$b`:

Also remember that Perl converts `$a` and `$b` to numbers in the usual way. It reads numbers or decimal points from the left for as long as possible, ignoring initial spaces, and then drops the rest of the string. If no numbers were found, the value is set to zero.

Try It Out : Guess My Number

This is a very simple guessing game. The computer has a number, and the user has to guess what it is. If the user doesn't guess correctly, the computer gives a hint. As we learn more about Perl, we'll add the opportunity to give more than one try and to pick a different number each game:

```
#!/usr/bin/perl
# guessnum.plx
use warnings;
use strict;

my $target = 12;
print "Guess my number!\n";
print "Enter your guess: ";
my $guess = <STDIN>;

if ($target == $guess) {
   print "That's it! You guessed correctly!\n";
   exit;
}
if ($guess > $target) {
   print "Your number is bigger than my number\n";
   exit;
}
if ($guess < $target){
   print "Your number is less than my number\n";
   exit;
}
```

Let's have a few go's at it:

```
> perl guessnum.plx
Guess my number!
Enter your guess: 3
Your number is less than my number
> perl guessnum.plx
Guess my number!
Enter your guess: 15
Your number is bigger than my number
> perl guessnum.plx
Guess my number!
Enter your guess: 12
That's it! You guessed correctly!
>
```

How It Works

First off, we set up our secret number. OK, at the moment it's not very secret, since it's right there in the source code, but we can improve on this later. After this, we get a number from the user:

```
my $guess = <STDIN>;
```

Then we do three sorts of comparisons with the numeric operators we've just seen. We use the basic pattern of the if statement again, `if ( <condition> ) { <action> }`:

```
if ($target == $guess) {
   print "That's it! You guessed correctly!\n";
   exit;
}
```

Since only one of the tests can be true – the user's number can't be both smaller than our number and the same as it – we may as well stop work after a test was successful. The `exit` operator tells perl to stop the program completely. You can optionally give `exit` a number from 0 to 255 to report back to the operating system. Traditionally, 0 denotes success and anything else is failure. By default, `exit` reports success.

Comparing Strings

When we're comparing strings, we use a different set of operators to do the comparisons:

`$a gt $b`	`$a` sorts alphabetically after `$b`
`$a le $b`	`$a` sorts alphabetically before `$b`
`$a eq $b`	`$a` is the same as `$b`
`$a ne $b`	`$a` is not the same as `$b`

Here's a very simple way of testing if a user knows a password. Don't use this for anything you value, since the user can just read the source code to find it!

```
#!/usr/bin/perl
# password.plx
use warnings;
use strict;

my $password = "foxtrot";
print "Enter the password: ";
my $guess = <STDIN>;
chomp $guess;
if ($password eq $guess) {
   print "Pass, friend.\n";
}
if ($password ne $guess) {
   die "Go away, imposter!\n";
}
```

Here's our security system in action:

```
> perl password.plx
Enter the password: abracadabra
Go away, imposter!
> perl password.plx
Enter the password: foxtrot
Pass, friend.
>
```

How It Works

As before, we ask the user for a line:

```
my $guess = <STDIN>;
```

Just a warning: this is a horrendously bad way of asking for a password, since it's echoed to the screen, and everyone looking at the user's computer would be able to read it. Even though you won't be using a program like this, if you ever do need to get a password from the user, the Perl FAQ provides a better method. In `perlfaq8`, type `perldoc -q password` to find it.

```
chomp $guess;
```

We must never forget to remove the new line from the end of the user's data. We didn't need to do this for numeric comparison, because Perl would remove that for us anyway during conversion to a number. Otherwise, even if the user had put the right password in, Perl would have tried to compare `"foxtrot"` with `"foxtrot\n"` and it could never be the same:

```
if ($password ne $guess) {
   die "Go away, imposter!\n";
}
```

Then if the password we have isn't the same as the user's input, we send out a rude message and terminate the program.

Other Tests

What other tests can we perform? We've seen `exists` at the beginning of this chapter, for determining whether a key exists in a hash. We can test if a variable is defined (It must contain something other than the undefined value), by using `defined`:

```
#!/usr/bin/perl
# defined.plx
use warnings;
use strict;

my ($a, $b);
$b = 10;
if (defined $a) {
   print "\$a has a value.\n";
}
if (defined $b) {
   print "\$b has a value.\n";
}
```

Not surprisingly, the result we get is this:

```
>perl defined.plx
$b has a value.
>
```

You can use this to avoid the warnings that occur when you try and use a variable that doesn't have a value. If we'd tried to say `if ($a == $b)`, Perl would have said:

```
Use of uninitialized value in numeric eq (==)
```

So we have our basic comparisons. Don't forget that some functions will return a true value if they were successful and the undefined value if they were not. You will often want to check whether the return value of an operation (particularly one that relates to the operating system) is true or not.

How do you actually test whether something is a true value or not? You may want to see if a user's input isn't empty after being chomped, for example. Well, don't do it like this:

```
my $true = (1 == 1);
if ($a == $true) { … }
```

The whole point of `if` is that it does the action if something is true. You should just say `if ($a) { ... }`

Logical Conjunctions

We also saw in Chapter 2 that we can join together several tests into one, by the use of the logical operators. Here's a summary of those:

`$a and $b`	True if both `$a` and `$b` are true.
`$a or $b`	True if either of `$a` or `$b`, or both are true.
`not $a`	True if `$a` is not true.

In fact, we saw `not` earlier:

```
if (not exists $rates{$to})
```

There is also another set of logical operators: `&&` for `and`, `||` for `or`, and `!` for `not`. However, I find the first set easier to read and understand. Don't forget there is a difference in precedence between the two – `and`, `or`, and `not` all have lower precedence than their symbolic representations.

Running Unless...

There's another way of saying `if (not exists $rates{$to})`. As always in Perl, there's more than one way to do it. Some people prefer to think 'if this doesn't exist, then { ... }', but other people think 'unless this does exist, then { ... }'. Perl caters for both sets of thought patterns, and we could just as easily have written this:

```
unless (exists $rates{$to}) {
   die "I don't know anything about {$to} as a currency\n";
}
```

The psychology is different, but the effect is the same: `unless ($a)` is effectively `if (not ($a))`. We'll see later how Perl provides a few alternatives for these control structures to help them more effectively fit the way you think.

Statement Modifiers

When we're talking in English, it's quite normal for us to say

- If this doesn't exist, then this happens, or
- Unless this exists, this happens.

Similarly, it's also quite natural to reverse the two phrases, saying

- This happens, if this doesn't exist, or
- This happens unless this exists.

Going back to our currency converter example, convert2.plx, we could turn around the if statements within to read:

```
die "I don't know anything about $rates{$to} as a currency\n"
   if not exists $rates{$to};
```

Notice how the syntax here is slightly different, it's *<action> if <condition>*. There is no need for brackets around the condition, and there are no braces around the action. Indeed, the indentation isn't part of the syntax, so we could even put the whole statement on one line. Only a single statement will be covered by the condition. The condition modifies the statement, and so is called a **statement modifier**.

We can turn `unless` into a statement modifier, too. So instead of this:

```
if (not exists $rates{$to}) {
   die "I don't know anything about {$to} as a currency\n";
}
if (not exists $rates{$from}) {
   die "I don't know anything about{$from} as a currency\n";
}
```

you may find it more natural to write this:

```
die "I don't know anything about $to as a currency\n"
   unless exists $rates{$to};
die "I don't know anything about $from as a currency\n"
   unless exists $rates{$from};
```

Sure enough, if you swap those lines into `convert2.plx`, you'll get the same results.

Using Logic

There is yet another way to do something if a condition is true, and we saw it briefly in Chapter 2. By using the fact that perl stops processing a logical conjunction when it knows the answer for definite, we can create a sort of `unless` conditional:

```
exists $rates{$to}
   or die "I don't know anything about {$to} as a currency\n";
```

How does this work? Well, it's reliant on the fact that perl uses lazy evaluation to give a logical conjunction its value. If we have the statement `X or Y`, then if `X` is true, it doesn't matter what `Y` is, so perl doesn't look at it. If `X` isn't true, perl has to look at `Y` to see whether or not that's true. So if `$rates{$to}` exists in the hash, then our currency converter won't die with an error message. Instead, it will do nothing and continue executing the next statement.

This form of conditional is most often used when checking that something we did succeeded or returned a true value. We will see it often when we're handling files.

To create a positive `if` conditional this way, use `and` instead of `or`. For example, to add one to a counter if a test is successful, you may say:

```
$success and $counter++;
```

If you recall, `and` statements are reliant on both sub-statements being true. So, if `$success` is not true, perl won't bother evaluating `$counter++` and upping its value by one. If `$success` was true, then it would.

Multiple Choice

If you look back to when we did our password tester, you'll see the following lines:

```
if ($password eq $guess) {
   print "Pass, friend.\n";
}
if ($password ne $guess) {
   die "Go away, imposter!\n";
}
```

While this does what we want, we know that if the first one is true, then the second one will not be – we're asking exactly opposite questions: Are these the same? Are they not the same?

In which case, it seems wasteful to do two tests. It'd be much nicer to be able to say 'if the strings are the same, do this. Otherwise, do that.' And in fact we can do exactly that, although the keyword is not 'otherwise' but '**else**':

```
if ($password eq $guess) {
   print "Pass, friend.\n";
} else {
   die "Go away, imposter!\n";
}
```

That's:

```
if ( <condition> ) { <action> } else { <alternative action> }
```

if elsif else

In some cases, we'll want to test more than one condition. When looking at several related possibilities, we'll want to ask questions like "Is this true? If this isn't, then is that true? If that's not true, how about the other?" Note that this is distinct from asking three independent questions; whether we ask the second depends on whether or not the first was true. In Perl, we could very easily write something like this:

```
if ( <condition 1> ) {
   <action>
} else {
   if ( <condition 2> ) {
      <second action>
   } else {
      if ( <condition 3> ) {
         <third action>
      } else {
         <if all else fails>
      }
   }
}
```

I hope you'll agree though that this looks pretty messy. To make it nicer, we can combine the `else` and the next `if` into a single word: `elsif`. Here's what the above would look like when rephrased in this way:

```
if ( <condition 1> ) {
   <action>
} elsif ( <condition 2> ) {
   <second action>
} elsif ( <condition 3> ) {
   ...
} else {
   <if all else fails>
}
```

Much neater! We don't have an awful cascade of closing brackets at the end, and it's easier to see what we're testing and when we're testing it.

Try It Out : Want To Go For A Walk?

I'll certainly not go outside if it's raining, but I'll always go out for a walk in the snow. I'll not go outside if it's less than 18 degrees Celsius. Otherwise, I'll probably go out unless I've got too much work to do. Do I want to go for a walk?

```
#!/usr/bin/perl
# walkies.plx
use warnings;
use strict;

print "What's the weather like outside? ";
my $weather = <STDIN>;
print "How hot is it, in degrees? ";
my $temperature = <STDIN>;
print "And how many emails left to reply to? ";
my $work = <STDIN>;
chomp($weather, $temperature);

if ($weather eq "snowing") {
   print "OK, let's go!\n";
} elsif ($weather eq "raining") {
   print "No way, sorry, I'm staying in.\n";
} elsif ($temperature < 18) {
   print "Too cold for me!\n";
} elsif ($work > 30) {
   print "Sorry - just too busy.\n";
} else {
   print "Well, why not?\n";
}
```

It's 20 degrees, I've got 27 emails to reply to, and it's cloudy out there. Let's see what the Simulated Simon would do:

```
> perl walkies.plx
What's the weather like outside? cloudy
How hot is it, in degrees? 20
And how many emails left to reply to? 27
Well, why not?
>
```

Looks like I can fit a walk in after all. Maybe after I show you how this program works.

How It Works

The point of this rather silly little program is that once it has gathered the information it needs, it runs through a series of tests, each of which could cause it to finish. First, we check to see if it's snowing:

```
if ($weather eq "snowing") {
   print "OK, let's go!\n";
```

If so, then we print our message and, this is the important part, do no more tests. If not, then we move onto the next test:

```
} elsif ($weather eq "raining") {
   print "No way, sorry, I'm staying in.\n";
```

Again, if this is true, we stop testing; otherwise, we move on. Finally, if none of the tests are true, we get to the `else`:

```
} else {
   print "Well, why not?\n";
}
```

Please remember that this is very different to what would happen if we used four separate `if` statements. The tests overlap, so it is possible for more than one condition to be true at once. For example, if it were snowing and I have over 30 emails to reply to, we'd get two conflicting answers. `elsif` tests should be read as 'Well, how about if…?'

Just in case you were curious, there is no `elsunless`. This is a Good Thing.

More Elegant Solutions

For three or four tests, it's reasonable to use `if-elsif-elsif-…-else`. But for any more than that, it starts to look ugly. What happens if we get input from the user and there are ten options? There are two general solutions to this, the first of which is to use a hash. We'll see in a few chapters time how you can store code to be executed inside a hash. If you can't use a hash, you're pretty much stuck with a chain of `elsifs`. You may, however, find it easier to do it like this:

```
print "Please enter your selection (1 to 10): ";
my $choice = <STDIN>;
for ($choice) {
   $_ == 1 && print "You chose number one\n";
   $_ == 2 && print "You chose number two\n";
   $_ == 3 && print "You chose number three\n";
   ...
}
```

We're using a `for` loop just like in the last chapter, but with a list of one thing. Why? Two reasons really:

- To give our program a bit of structure – brackets and indenting should make you realize there's a control structure going on.
- To alias `$choice` to `$_` for more convenient access.

Let's have a look in more detail about how the `for` loop works.

1, 2, Skip A Few, 99, 100

Now we know how to do everything once. But what if we need to repeat an operation or series of operations? Of course, there are methods available to specify this in perl too. We saw the `for` loop in Chapter 3, and this is one example of a class of control structures called loops.

In programming, there are various types of loop. Some loop forever and are called **infinite loops**, while most, in contrast, are finite loops. We say that a program 'gets into' or 'enters' a loop and then 'exits' or 'gets out' when finished. Infinite loops may not sound very useful, but they certainly can be – particularly because most languages, Perl included, provide you with a 'site door' by which you can exit

the loop. They will also be useful for when you just want the program to continue running until the user stops it manually, the computer powers down, or the heat death of the universe occurs, whichever is sooner.

There's also a difference between 'definite' loops and 'indefinite' loops. In a definite loop, you know how many times the block will be repeated in advance – a `for` loop is definite, because it will always iterate for each item in the array. An indefinite loop will check a condition in each iteration to whether it should do another or not.

There's also a difference between an indefinite loop that checks before the iteration and one that checks afterward. The latter will always go through at least one iteration in order to get to the check, whereas the former checks first and so may not go through any iterations at all.

Perl supports ways of expressing all of these types of loop. First, let's examine again the `for` loops we saw in the previous chapter.

for Loops

The `for` loop executes the statements in a block for each element in a list. Because of this, it's also known as the `foreach` loop, and you can use `foreach` anywhere you'd use `for`. For example:

```
#!/usr/bin/perl
#forloop1.plx
use warnings;
use strict;

my @array = (1, 3, 5, 7, 9);
my $i;
for $i (@array) {
   print "This element: $i\n";
}
```

This does exactly the same thing, and gives exactly the same output as this:

```
#!/usr/bin/perl
#forloop2.plx
use warnings;
use strict;

my @array = (1, 3, 5, 7, 9);
my $i;
foreach $i (@array) {
   print "This element: $i\n";
}
```

It's mainly a question of personal style – you won't go wrong if you use `foreach` all the time when talking about arrays. There's another form of `for` that does something completely different, and we'll see that a bit later on. We borrowed the syntax from a language called C, and so people who are used to programming in C can sometimes be confused by seeing `for` used with an array. If you use `foreach`, you'll keep them happy.

However, `foreach` is longer to type, and Perl programmers are notoriously lazy. And what's more, this is Perl, not C. Personally, I try and use `for` for constant lists and ranges like `(1...10)`, and `foreach` for arrays, but I'm not really consistent in that. Use whatever suits you.

As we mentioned above, the `for` loop is definite. You can work out, before you enter the loop, how many times you are going to repeat. It's also finite, since it's not possible to construct an infinitely long list.

Choosing an Iterator

We can specify the iterator variable ourselves as we did in the examples above, or we can use the default one, `$_` . Furthermore, if we're being good and using `strict`, we can make our iterator variable a lexical, `my` variable as we go along. That is, we could write the above like this:

```
#!/usr/bin/perl
#forloop3.plx
use warnings;
use strict;

my @array = (1, 3, 5, 7, 9);
foreach my $i (@array) {
   print "This element: $i\n";
}
```

There's actually a very subtle difference between declaring your iterator inside and outside of the loop. If you declare your iterator outside the loop, any value it had then will be restored afterwards. We can check this out by setting the variable and testing it afterwards:

```
#!/usr/bin/perl
#forloop4.plx
use warnings;
use strict;

my @array = (1, 3, 5, 7, 9);
my $i="Hello there";
foreach $i (@array) {
   print "This element: $i\n";
}
print "All done: $i\n";
```

This will produce the following output:

```
> perl forloop4.plx
This element: 1
This element: 3
This element: 5
This element: 7
This element: 9
All done: Hello there
>
```

Meanwhile declaring the iterator within the loop, as in `forloop3.plx`, will create a new variable `$i` each time, which will only exist for the duration of the loop.

As a matter of style, it's usual to keep the names of iterator variables very short. The traditional iterator is `$i`, as I've used here. The length of a variable name should be related to the importance of the variable; iterators are throwaway variables that only exist for one block, so they shouldn't be prominently named.

What We Can Loop Over

We can use `foreach` and `for` loops on any type of list whatsoever: A constant list:

```
my @array = qw(the quick brown fox ran over the lazy dog);
for (6, 3, 8, 2, 5, 4, 0, 7) {
   print "$array[$_] ";
}
```

an array:

```
my @array = qw(the quick brown fox ran over the lazy dog);
my $word;
for $word (@array) {
   print "$word ";
}
```

even a list generated by a function, like `sort` or `keys`:

```
my %hash = ( car => 'voiture', coach => 'car', bus => 'autobus' );
for (keys %hash) {
   print "English: $_\n";
   print "French: $hash{$_}\n\n";
}
```

It's a very powerful tool for those of you who need to list or 'enumerate' the contents of a hash or array, but there is a proviso before you go and use your `for` loops unwisely.

Aliases and Values

Be aware that the `for` loop creates an alias, rather than a value. Any changes you make to the iterator variable, whether it be `$_` or one you supply, will be reflected in the original array. For instance:

```
#!/usr/bin/perl
# forloop5.plx
use warnings;
use strict;

my @array = (1..10);
foreach (@array) {
   $_++;
}

print "Array is now: @array\n";
```

will change the actual contents of the array, as follows:

```
> perl forloop5.plx
Array is now: 2 3 4 5 6 7 8 9 10 11
>
```

Naturally, you can't change things that are constant, so doing the following will give an error:

```
#!/usr/bin/perl
# forloop6.plx
use warnings;
use strict;

foreach (1, 2, 3) {
   $_++;
}
```

```
> perl forloop6.plx
Modification of a read-only value attempted at forloop6.plx line 7
>
```

This means exactly what it says – we tried to modify (by adding one) to something that we could only read from, and not write to – in this case, the literal value of one. If you need to change the iterator for any reason, make a local copy, like this:

```
#!/usr/bin/perl
# forloop7.plx
use warnings;
use strict;

foreach (1, 2, 3) {
   my $i = $_;
   $i++;
}
```

Because `$i` is unrelated to the original list, we don't run into this problem.

Statement Modifiers

Just as there was a statement modifier form of `if`, like this:

```
die "Something wicked happened" if $error;
```

there's also a statement modifier form of `for`. This means you can iterate an array, executing a single statement every time. Here, however, you don't get to choose your own iterator variable: it's always `$_`. Let's create a simple totalling program using this idiom:

Try It Out : Quick Sum

The aim of this program is to take a list of numbers and output the total. For ease of use, we'll take the numbers from the command line.

```
#!/usr/bin/perl
# quicksum.plx
use warnings;
use strict;

my $total=0;
$total += $_ for @ARGV;
print "The total is $total\n";
```

Now when we give the program a few numbers to sum, it does just that:

```
>perl quicksum.plx 15 62 3 8 4
The total is 92
>
```

How It Works - @ARGV Explained

The whole trickery of this program is in that one line:

```
$total += $_ for @ARGV;
```

The first key point is the `@ARGV` array. This contains everything that's on the command line after the name of the program we're running. Perl receives, from the system, an array containing everything on the command line. This is split up a little like a Perl array without the commas. A single word is one element, as is a number, or a string in quotes. Depending on the shell, which is the thing that talks to the system in the first place, you may be able to backslash a space to stop it separating. Let's quickly write something that prints out each element of `@ARGV` separately, so we can see how they're split up:

```
#!/usr/bin/perl
# whatsargv.plx
use warnings;
use strict;

foreach (@ARGV) {
   print "Element: |$_|\n";
}
```

> **Why the strange parallel bars? You'll be pleased to hear it's not some arcane Perl syntax for doing anything special. All we're doing is placing a symbol on either side of our element. This is an oft-used debugging trick: the idea is that it allows you to see if there are any spaces at the start of end of the data, and allows you to tell the difference between an empty string and a string consisting entirely of spaces.**

Now let's try a few examples:

```
> perl whatsargv.plx one two three
Element: |one|
Element: |two|
Element: |three|
> perl whatsargv.plx "a string"    12
Element: |a string|
Element: |12|
> perl whatsargv.plx
>
```

In the first case, the three words were split up into separate elements. In the second, we kept two words together by giving them as a string in quotes. We also had another element afterwards, and the amount of white space between them made no difference to the number of elements. Finally, if there is nothing after the name of the program, there's nothing in `@ARGV`.

Let's get back to our program. We've now got an array constructed from of the command line, and we're going over it with a for loop:

```
$total += $_ for @ARGV;
```

With these statement modifiers, if they're not obviously clear, think how they'd be written normally. In this case:

```
for (@ARGV) {
   $total += $_;
}
```

that is, for each element, add that element to the total. This is more or less exactly how you take a total.

Looping While...

Now we come to the indefinite loops. As mentioned above, these check a condition, then do an action. The first one is `while`. As you might be able to work out from the name, this means anaction continues while a condition is true. The syntax of `while` is much like the syntax of `if`:

```
while ( <condition> ) { <action> }
```

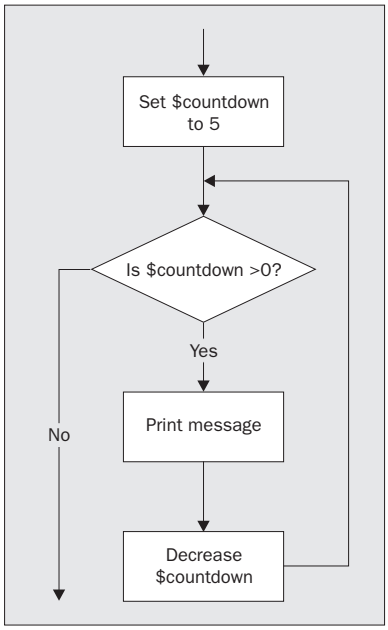
Here's a very simple while loop:

```
#!/usr/bin/perl
# while1.plx
use warnings;
use strict;

my $countdown = 5;

while ($countdown > 0) {
   print "Counting down: $countdown\n";
   $countdown--;
}
```

And here's what it produces:

```
>perl while1.plx
Counting down: 5
Counting down: 4
Counting down: 3
Counting down: 2
Counting down: 1
>
```

Let's see a flow chart for this program:



While there's still a value greater than zero in the `$counter` variable, we do the two statements:

```
print "Counting down: $countdown\n";
$countdown--;
```

Perl goes through the loop a first time when `$countdown` is 5 – the condition is met, so a message gets printed, and `$countdown` gets decreased to 4. Then, as the flow chart implies, back we go to the top of the loop. We test again: `$countdown` is still more than zero, so off we go again. Eventually, `$countdown` is 1, we print our message, `$countdown` is decreased, and it's now zero. This time around, the test fails, and we exit the loop.

while (<STDIN>)

Of course, another way to ensure that your loop is going to terminate is to make the condition do the change. This is sometimes thought of as bad style, but there's one example that is extremely widespread:

```
while (<STDIN>) {
   ...
}
```

Actually, this is a bit of shorthand; another example of a common Perl idiom. To write it out fully, it would look like this:

```
while (defined($_ = <STDIN>)) {
   ...
}
```

Since `<STDIN>` reads a new line from the user, the condition itself will depend on changing information. What we're doing is setting `$_` to each new line of input until we run out. We'll see in Chapter 6 how this is used to process files.

Infinite Loops

The important but obvious point is that what we're testing gets changed inside the loop. If our condition is always going to give a true result, we have ourselves an **infinite loop**. Let's just remove the second of those two statements:

```
#!/usr/bin/perl
# while2.plx
use warnings;
use strict;

my $countdown = 5;

while ($countdown > 0) {
   print "Counting down: $countdown\n";
}
```

`$countdown` never changes. It's always going to be 5, and 5 is, we hope, always going to be more than zero. So this program will keep printing its message until you interrupt it by holding down Ctrl and C. Hopefully, you can see why you need to ensure that what you do in your loop affects your condition.

Should we actually want an infinite loop, there's a fairly standard way to do it. Just put a true value – typically `1` – as the condition:

```
while (1) {
   print "Bored yet?\n";
}
```

The converse of course is to say `while(0)` in the loop's declaration, but nothing will ever happen because this condition is tested before any of the commands in the loop are executed. A bit silly, really.

Try It Out : English – Sdrawkcab Translator

In this example, we'll use our newly-introduced `while(<STDIN>)` construction to take a line of text from the user and produce the equivalent sentence translated into a language called Sdrawkcab. Sdrawkcab is a word in Sdrawkcab meaning 'backwards' – I hope you can see why.

```
#!/usr/bin/perl
# sdrawkcab1.plx
use warnings;
use strict;

while (<STDIN>) {
   chomp;
   die "!enod llA\n" unless $_;
   my $sdrawkcab = reverse $_;
   print "$sdrawkcab\n";
}
```

And here's a sample run.

```
> perl sdrawkcab1.plx
Hello
olleH
How are you?
?uoy era woH

!enod llA
>
```

(!oot ,rotalsnart hsilgnE-backwardS a sa ti esu yllautca nac uoy taht si rotalsnart siht tuoba gniht taerg ehT)

How It Works

The main part of this program is a loop that takes in a line from the user and places it in `$_`:

```
while (<STDIN>) {
   ...
}
```

Inside that loop, what do we do? First, we remove the new line. If we're going to turn it into Sdrawkcab, we want the new line at the end, not the beginning:

```
chomp;
```

If, after removing that new line, there's nothing left, `$_` is an empty string, a false value, we then finish the program:

```
die "!enod llA\n" unless $_;
```

Next, we do our actual translation – we can't just do `print reverse $_` however, because `reverse` in a list context, such as supplied by `print`, treats its arguments as a list and reverses the order of the items. Since we've only got one item here, that wouldn't be very interesting. You'll just get what you typed in:

```
my $sdrawkcab = reverse $_;
```

Then, finally, we print it out, and go back to get another line.

```
print "$sdrawkcab\n";
```

It's not very elegant, granted, but it gets the job done. Of course, there's more than one way to do it as I'll show you in the next section. But before you read on, you might like to make the translator a little 'prettier', prompting the user for a phrase to translate and prefacing the translation with a suitable phrase. From streams of such small improvements to an established core, most programs came.

Running at Least Once

When we were categorizing our lists, we divided indefinite loops into two catagories: those that those that execute at least once and those that may execute zero times. The `while` loop we've seen so far tests the condition first; if the condition isn't true the first time around, the 'body' of the loop never gets executed. There's another way to write our loop to ensure that the body is always executed at least once:

```
do { <actions> } while (<condition>)
```

Now we do the test after the block. This is equivalent to moving the diamond in our flow chart from the top to the bottom.

You may find it more natural to write the previous program like this:

```
#!/usr/bin/perl
# sdrawkcab2.plx
use warnings;
use strict;

do {
   $_ = <STDIN>;
   chomp;
   my $sdrawkcab = reverse $_;
   print "$sdrawkcab\n";
} while ($_);
print "!enod llA\n";
```

This does more or less the same thing, but in a slightly different way. First a line is read, then the translation produced, then we see if we need to get another line. There's one slight problem with this, when we want to end, we input a blank line that Perl 'translates' and prints out. See if you can fix this, and then see if you prefer the end result with the first program.

Statement Modifying

As before, you can use `while` as a statement modifier. Following the pattern of `for` and `if`, here's what you'd do with `while`:

```
while ( <condition> ) { <statement> }
```

becomes:

```
<statement> while <condition>
```

So, here's a way of writing our countdown program in three lines (if you exclude '`use strict`' and '`use warnings`', of course):

```
#!/usr/bin/perl
my $countdown = 5;
print "Counting down: $countdown\n" while $countdown-- > 0;
```

Don't be confused by the fact that the `while` is at the end – the condition is tested first, just as an ordinary `while` loop.

Looping Until

The opposite of `if` is `unless`, and the opposite of `while` is `until`. It's exactly the same as `while (not <condition>) { ... }`:

```
#!/usr/bin/perl
# until.plx
use warnings;
use strict;

my $countdown = 5;

until ($countdown-- == 0) {
   print "Counting down: $countdown\n";
}
```

Controlling Loop Flow

When we wrote our Sdrawkcab translator, the only way we could stop the loop was to end the program with a `die` command. Of course, there is another way to do it – by keeping a variable set to tell us whether or not we want to go through another loop. We can test this in our `while` condition. This kind of Boolean variable is called a **flag**, because it indicates something about the status of our program. We **set a flag** when we change its value.

Here's a version of the Sdrawkcab program that sets a flag when it's time to finish:

```
#!/usr/bin/perl
# sdrawkcab3.plx
use warnings;
use strict;

my $stopnow = 0;
until ($stopnow) {
   $_ = <STDIN>;
   chomp;
if ($_) {
      my $sdrawkcab = reverse $_;
      print "$sdrawkcab\n";
   } else {
      $stopnow = 1;
   }
}
print "!enod llA\n";
```

When `$_` becomes the empty string, and hence a false value, the `if ($_)` test fails. This sets `$stopnow` to `1` and will end the `until` loop.

There's a school of thought, called 'structured programming' that urges strict adherence to these loops and conditionals. Unfortunately, you end up with code like on the previous page. Most programmers, though, take a less strict approach. When it's time to leave the loop, they don't wait for the test to come around again, they just leave.

Breaking Out

The keyword `last`, in the body of a loop, will make perl immediately exit, or 'break out of' that loop. The remaining statements are not processed, and you up right at the end. This is exactly what we want to do to make the above program easier to deal with:

```
#!/usr/bin/perl
# sdrawkcab4.plx
use warnings;
use strict;

while (<STDIN>) {
   chomp;
   last unless $_;
   my $sdrawkcab = reverse $_;
   print "$sdrawkcab\n";
}
# and now we can carry on with something else...
```

You can use this in a `for` loop as well:

```
#!/usr/bin/perl
# forlast.plx
use warnings;
use strict;

my @array = ( "red", "blue", "STOP THIS NOW", "green");
for (@array) {
   last if $_ eq "STOP THIS NOW";
   print "Today's colour is $_\n";
}
```

```
>perl forlast.plx
Today's colour is red
Today's colour is blue
>
```

If you try to do a `last` when you're not in a loop, perl will complain, even if you have forgotten to use `use warnings`:

```
#!/usr/bin/perl
# badlast.plx

last;
```

```
Can't "last" outside a block at badlast.plx line 4.
```

Going onto the Next

If you want to skip the rest of the processing of the body, but don't want to exit the loop, you can use `next` to immediately go back to the start of the loop, passing the next value to the iterator. This is an oft-used technique to process only selected elements:

```
#!/usr/bin/perl
# next.plx
use strict;
use warnings;

my @array = (8, 3, 0, 2, 12, 0);
for (@array) {
   if ($_ == 0) {
      print "Skipping zero element.\n";
      next;
   }
   print "48 over $_ is ", 48/$_, "\n";
}
```

In `next.plx` then, we have set a trap for all those dastardly zeroes that want to cause our divisions to fail:

```
>perl next.plx
48 over 8 is 6
48 over 3 is 16
Skipping zero element.
48 over 2 is 24
48 over 12 is 4
Skipping zero element.
>
```

> **Be careful: while `next` takes you to the next iteration of the loop, `last` doesn't take you to the last iteration, it takes you past it.**

On rare occasions, you'll want to go back to the top of the loop, but without testing the condition (in the case of a `for` loop) or getting the next element in the list (as in a `while` loop). If you feel you need to do this, the keyword to use is `redo`:

Try It Out - Debugging Loops 101

It's perfectly possible to have a loop inside a loop. The interesting part comes when you need to go to the end or the beginning of a external loop from an internal loop. For example, if you're reading some input from the user, and the input is any one of a series of pre-determined 'safe words', you end the loop. Here's what you might want to do:

```
#!/usr/bin/perl
# looploop1.plx
use warnings;
use strict;
my @getout = qw(quit leave stop finish);
```

```
while (<STDIN>) {
   chomp;
   for my $check (@getout) {
      last if $check eq $_;
   }
   print "Hey, you said $_\n";
}
```

The problem with this is that it doesn't work. Now, 'it doesn't work' is possibly the worst way to approach finding a bug. What do we mean by 'it doesn't work'? Does it sit on the couch all day watching TV? We need to be specific! What doesn't work about it?

How It Doesn't Work and Why

Well, even if we put in one of the words that's supposed to let us quit, Perl carries on, like this:

```
>perl looploop1.plx
Hello
Hey, you said Hello
quit
Hey, you said quit
stop
Hey, you said stop
leave
Hey, you said leave
finish
Hey, you said finish
```

We've specifically isolated the problem. Now, let's see if we can find any clues as to what's causing it. The fact that it's printing out means it's finished the `for` loop. Let's add in a couple of `print` statements to help us investigate what the `for` loop is actually doing:

```
for my $check (@getout) {
   print "Testing $check against $_\n";
   last if $check eq $_;
   print "Well, it wasn't $check\n";
}
```

Now run it again:

```
Hello
Testing quit against Hello
Well, it wasn't quit
Testing leave against hello
Well, it wasn't leave
Testing stop against Hello
Well, it wasn't stop
Testing finish against Hello
Well, it wasn't finish
Hey, you said Hello
quit
Testing quit against quit
Hey, you said quit
```

Aha, more clues. So it's testing properly, and it's finishing when it sees 'quit', which is one of the stop words. That's a relief to know, but it's only finishing the `for` loop, rather than finishing the `while` loop. This is the root of the problem:

> **'It doesn't work' is not a bug report. First you need to be specific about what doesn't work. Then you need to detail what doesn't work about it. Then you can start to examine why it doesn't work. When you've got over the 'doesn't work' feeling and fully investigated what it's really doing and how that differs from your expectations, only then can you begin to fix it.**

So, how do we fix this one? What we need to do is to distinguish between the two loops, the inner `for` loop and the outer `while` loop. The way we distinguish between them is by giving them names, or **labels**.

A **label** goes before the `for`, `while`, or `until` of a loop, and ends with a colon. The rules for naming labels are the same as for naming variables, but it's usual to construct labels from uppercase letters.

Here's our program with labels attached:

```
#!/usr/bin/perl
# looploop2.plx
use warnings;
use strict;

my @getout = qw(quit leave stop finish);

OUTER: while (<STDIN>) {
   chomp;
   INNER: for my $check (@getout) {
      last if $check eq $_;
   }
   print "Hey, you said $_\n";
}
```

Now for the finale, we can direct last, next, and redo to a particular loop by giving them the label. Here's the fixed version:

```
#!/usr/bin/perl
# looploop3.plx
use warnings;
use strict;

my @getout = qw(quit leave stop finish);

OUTER: while (<STDIN>) {
   chomp;
   INNER: for my $check (@getout) {
      last OUTER if $check eq $_;
   }
   print "Hey, you said $_\n";
}
```

Now when we find a matching word, we don't just jump out of the `for` loop – we go all the way to the end of the outer `while` loop as well, which is exactly what we wanted to do.

Goto

As a matter of fact, you can put a label before any statement whatsoever. If you want to really mess around with the structure of your programs, you can use `goto LABEL` to jump anywhere in your program. Whatever you do, don't do this. This is not to be used. Don't go that way.

I'm telling you about it for the simple reason that if you see it in anyone else's Perl, you can laugh heartily at them. There are other, more acceptable forms of `goto`, which we'll see when we come to subroutines. But `goto` with a label is to be avoided like the plague.

Why? Because not only does it turn the clock back thirty years (the structured programming movement started with the publication of a paper called 'Use of goto considered harmful'), but it tends to make your programs amazingly hard to follow. The flow of control can shoot off in any direction at any time, into any part of the file, perhaps into a different file. You can even find yourself jumping into the middle of loops, which really doesn't bear thinking about. Don't use it unless you really, really, really understand why you shouldn't. And even then, don't use it. Larry Wall has never used `goto` with a label in Perl, and he wrote it.

Don't. (He's watching - *Ed*)

Summary

Before this chapter, our programs plodded along in a straight line, following one statement with another.

We've now seen how we can react to different circumstances in our programs, which is the start of flexible and powerful programming. We can test whether something is true or false using `if` and `unless` and take appropriate action. We've also examined how to test multiple related conditions, using `elsif`.

We can repeat areas of a program, in several different ways: once per element of a list, using `for`, or continually while a condition is true or false, using `while` and `until`.

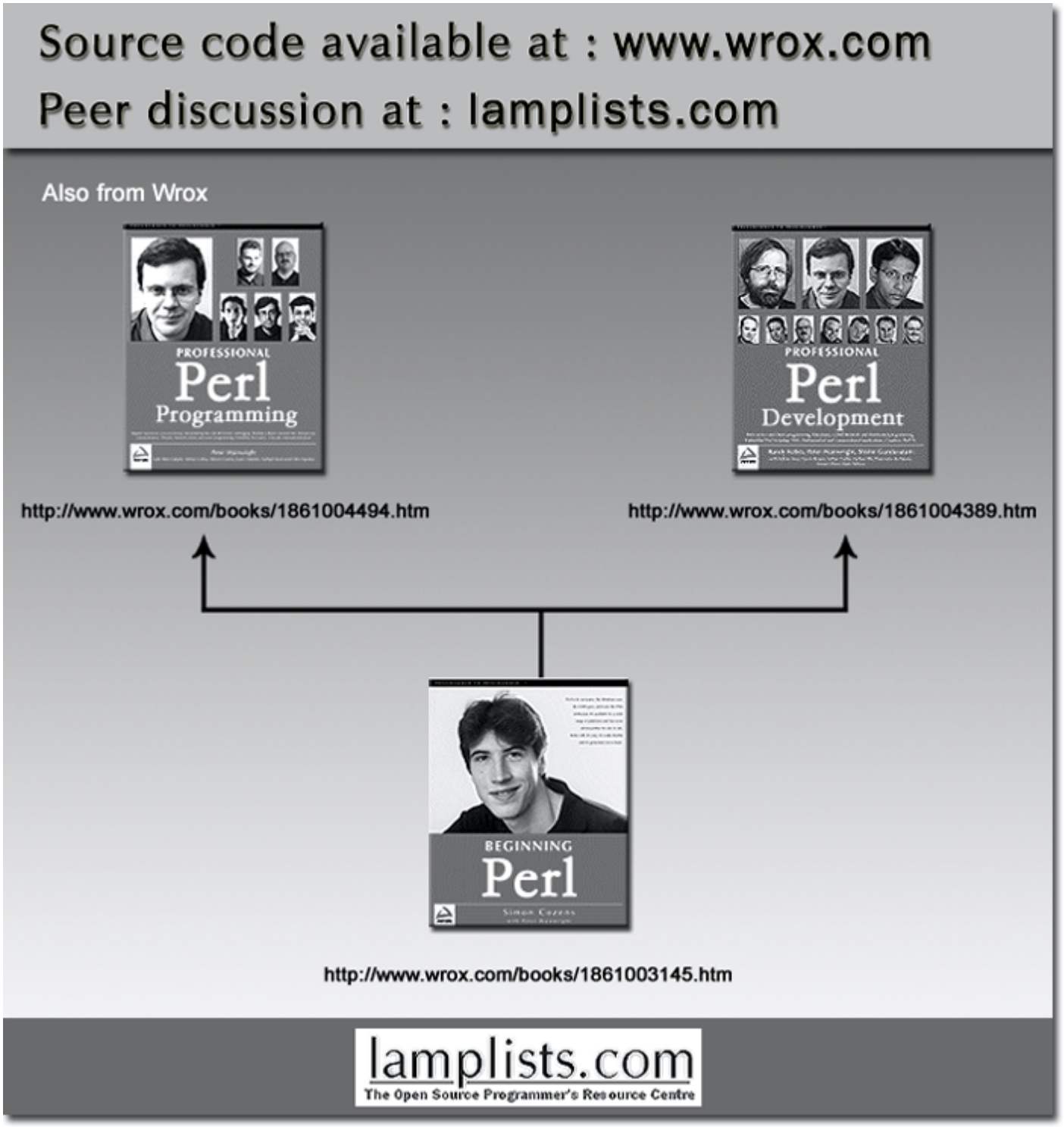
Finally, we've examined some ways to alter the flow of perl's execution through these loops. We can break out of a loop with `last`, skip to the next element with `next`, and start processing the current element again with `redo`.

Exercises

1. Modify the currency program convert2.plx to keep asking for currency names until a valid currency name is entered.

2. Modify the number-guessing program guessnum.plx so that it loops until the correct answer is entered.

3. Write your own program to capture all the prime numbers between 2 and a number the user gives you.



5

Regular Expressions

"11:15. Restate my assumptions:

1. *Mathematics is the language of nature.*
2. *Everything around us can be represented and understood through numbers.*
3. *If you graph these numbers, patterns emerge. Therefore: There are patterns everywhere in nature."*

- Max Cohen in Pi*, 1998*

Whether or not you agree that Max's assumptions give rise to his conclusion is your own opinion, but his case is much easier to follow in the field of computers – there are certainly patterns everywhere in programming.

Regular expressions allow us look for patterns in our data. So far we've been limited to checking a single value against that of a scalar variable or the contents of an array or hash. By using the rules outlined in this chapter, we can use that one single value (or pattern) to describe what we're looking for in more general terms: we can check that every sentence in a file begins with a capital letter and ends with a full stop, find out how many times James Bond's name is mentioned in 'Goldfinger', or learn if there are any repeated sequences of numbers in the decimal representation of greater than five in length.

However, regular expressions are a very big area – they're one of the most powerful features of Perl. We're going to break our treatment of them up into six sections:

- Basic patterns
- Special characters to use
- Quantifiers, anchors and memorizing patterns
- Matching, substituting, and transforming text using patterns
- Backtracking
- A quick look at some simple pitfalls

Generally speaking, if you want to ask perl something about a piece of text, regular expressions are going to be your first port of call – however, there's probably one simple question burning in your head…

What Are They?

The term "Regular Expression" (now commonly abbreviated to "RegExp" or even "RE") simply refers to a pattern that follows the rules of syntax outlined in the rest of this chapter. Regular expressions are not limited to perl – Unix utilities such as `sed` and `egrep` use the same notation for finding patterns in text. So why aren't they just called 'search patterns' or something less obscure?

Well, the actual phrase itself originates from the mid-fifties when a mathematician called Stephen Kleene developed a notation for manipulating 'regular sets'. Perl's regular expressions have grown and grown beyond the original notation and have significantly extended the original system, but some of Kleene's notation remains, and the name has stuck.

Patterns

History lessons aside, it's all about identifying patterns in text. So what constitutes a pattern? And how do you compare it against something?

The simplest pattern is a word – a simple sequence of characters – and we may, for example, want to ask perl whether a certain string contains that word. Now, we can do this with the techniques we have already seen: We want to split the string into separate words, and then test to see if each word is the one we're looking for. Here's how we might do that:

```
#!/usr/bin/perl
# match1.plx
use warnings;
use strict;

my $found = 0;
$_ = "Nobody wants to hurt you... 'cept, I do hurt people sometimes, Case.";

my $sought = "people";

foreach my $word (split) {
   if ($word eq $sought) {
      $found = 1;
      last;
   }
}

if ($found) {
   print "Hooray! Found the word 'people'\n";
}
```

Sure enough the program returns success:

```
>perl match1.plx
Hooray! Found the word 'people'
>
```

But that's messy! It's complicated, and it's slow to boot! Worse still, the `split` function (which breaks each of our lines up into a list of 'words' – we'll see more of this, later on in the chapter) actually **keeps** all the punctuation – the string '`you`' wouldn't be found in the above, whereas '`you...`' would. This looks like a hard problem, but it should be easy. Perl was designed to make easy tasks easy and hard things possible, so there should be a better way to do this. This is how it looks using a regular expression:

```
#!/usr/bin/perl
# match1.plx
use warnings;
use strict;

$_ =    "Nobody wants to hurt you... 'cept, I do hurt people sometimes, Case.";

if ($_ =~ /people/) {
   print "Hooray! Found the word 'people'\n";
}
```

This is much, much easier and yeilds the same result. We place the text we want to find between forward slashes – that's the regular expression part – that's our pattern, what we're trying to match. We also need to tell perl which particular string we're looking for in that pattern. We do this with the =~ operator. This returns 1 if the pattern match was successful (in our case, whether the character sequence 'people' was found in the string) and the undefined value if it wasn't.

Before we go on to more complicated patterns, let's just have a quick look at that syntax. As we noted previously, a lot of Perl's operations take $_ as a default argument, and regular expressions are one such operation. Since we have the text we want to test in $_, we don't need to use the =~ operator to 'bind' the pattern to another string. We could write the above even more simply:

```
$_ = "Nobody wants to hurt you... 'cept, I do hurt people sometimes, Case.";
if (/people/) {
   print "Hooray! Found the word 'people'\n";
}
```

Alternatively, we might want to test for the pattern not matching – the word not being found. Obviously, we could say `unless (/people/)`, but if the text we're looking at isn't in $_, we may also use the negative form of that =~ operator, which is `!~`. For example:

```
#!/usr/bin/perl
# nomatch.plx
use warnings;
use strict;

my $gibson =
   "Nobody wants to hurt you... 'cept, I do hurt people sometimes, Case.";

if ($gibson !~ /fish/) {
   print "There are no fish in William Gibson.\n";
}
```

True to form, for cyberpunk books that don't regularly involve fish, we get the result.

```
>perl nomatch.plx
There are no fish in William Gibson.
>
```

Literal text is the simplest regular expression of all to look for, but we needn't look for just the one word – we could look for any particular phrase. However, we need to make sure that we exactly match *all* the characters: words (with correct capitalization), numbers, punctuation, and even whitespace:

```
#!/usr/bin/perl
# match2.plx
use warnings;
use strict;

$_ = "Nobody wants to hurt you... 'cept, I do hurt people sometimes, Case.";

if (/I do/) {
   print "'I do' is in that string.\n";
}

if (/sometimes Case/) {
   print "'sometimes Case' matched.\n";
}
```

Let's run this program and see what happens:

```
>perl match2.plx
'I do' is in that string.
>
```

The other string didn't match, even though those two words are there. This is because everything in a regular expression has to match the string, from start to finish: first "sometimes", then a space, then "Case". In `$_`, there was a comma before the space, so it didn't match exactly. Similarly, spaces inside the pattern are significant:

```
#!/usr/bin/perl
# match3.plx
use warnings;
use strict;

my $test1 = "The dog is in the kennel";
my $test2 = "The sheepdog is in the field";

if ($test1 =~ / dog/) {
   print "This dog's at home.\n";
}

if ($test2 =~ / dog/) {
   print "This dog's at work.\n";
}
```

This will only find the first dog, as perl was looking for a space followed by the three letters, 'dog':

```
>perl match3.plx
This dog's at home.
>
```

So, for the moment, it looks like we shall have to specify our patterns with absolute precision. As another example, look at this:

```
#!/usr/bin/perl
# match4.plx
use warnings;
use strict;
```

```
$_ = "Nobody wants to hurt you... 'cept, I do hurt people sometimes, Case.";
if (/case/) {
   print "I guess it's just the way I'm made.\n";
} else {
   print "Case? Where are you, Case?\n";
}
```

```
> perl match4.plx
Case? Where are you, Case?
>
```

Hmm, no match. Why not? Because we asked for a small 'c' when we had a big 'C' – regexps are (if you'll pardon the pun) case-sensitive. We can get around this by asking perl to compare insensitively, and we do this by putting an 'i' (for 'insensitive') after the closing slash. If we alter the code above as follows:

```
if (/case/i) {
   print "I guess it's just the way I'm made.\n";
} else {
   print "Case? Where are you, Case?\n";
}
```

then we find him:

```
>perl match4.plx
I guess it's just the way I'm made.
>
```

This 'i' is one of several **modifiers** that we can add to the end of the regular expression to change its behavior slightly. We'll see more of them later on.

Interpolation

Regular expressions work a little like double-quoted strings; variables and metacharacters are interpolated. This allows us to store patterns in variables and determine what we are matching when we run the program – we don't need to have them hard-coded in:

Try it out – Pattern Tester

This program will ask the user for a pattern and then test to see if it matches our string. We can use this throughout the chapter to help us test the various different styles of pattern we'll be looking at:

```
#!/usr/bin/perl
# matchtest.plx
use warnings;
use strict;

$_ = q("I wonder what the Entish is for 'yes' and 'no'," he thought.);
# Tolkien, Lord of the Rings

print "Enter some text to find: ";
my $pattern = <STDIN>;
chomp($pattern);
```

```
if (/$pattern/) {
   print "The text matches the pattern '$pattern'.\n";
} else {
   print "'$pattern' was not found.\n";
}
```

Now we can test out a few things:

> **perl matchtest.plx**
Enter some text to find: **wonder**
The text matches the pattern 'wonder'.

> **perl matchtest.plx**
Enter some text to find: **entish**
'entish' was not found.

> **perl matchtest.plx**
Enter some text to find: **hough**
The text matches the pattern 'hough'.

> **perl matchtest.plx**
Enter some text to find: **and 'no',**
The text matches the pattern 'and 'no''.

Pretty straightforward, and I'm sure you could all spot those not in $_ as well.

How It Works

`matchtest.plx` has its basis in the three lines:

```
my $pattern = <STDIN>;
chomp($pattern);

if (/$pattern/) {
```

We're taking a line of text from the user. Then, since it will end in a new line, and we don't necessarily want to find a new line in our pattern, we `chomp` it away. Now we do our test.

Since we're not using the `=~` operator, the test will be looking at the variable `$_`. The regular expression is `/$pattern/`, and just like the double-quoted string `"$pattern"`, the variable `$pattern` is interpolated. Hence, the regular expression is purely and simply whatever the user typed in, once we've got rid of the new line.

Escaping Special Characters

Of course, regular expressions can be more than just words and spaces. The rest of this chapter is going to be about the various ways we can specify more advanced matches – where portions of the match are allowed to be one of a number of characters, or where the match must occur at a certain position in the string. To do this, we'll be describing the special meanings given to certain characters – called **metacharacters** – and look at what these meanings are and what sort of things we can express with them.

At this stage, we might not want to use their special meanings – we may want to literally match the characters themselves. As you've already seen with double-quoted strings, we can use a backslash to escape these characters' special meanings. Hence, if you want to match '. . .' in the above text, you need your pattern to say '\ . \ . \ .'. For example:

```
> perl matchtest.plx
Enter some text to find: Ent+
The text matches the pattern 'Ent+'.

> perl matchtest.plx
Enter some text to find: Ent\+
'Ent\+' was not found.
```

We'll see later why the first one matched – due to the special meaning of +.

> **These are the characters that are given special meaning within a regular expression, which you will need to backslash if you want to use literally:**
>
> **. * ? + [ ] ( ) { } ^ $ | **
>
> **Any other characters automatically assume their literal meanings.**

You can also turn off the special meanings using the escape sequence `\Q`. After perl sees `\Q`, the 14 special characters above will automatically assume their ordinary, literal meanings. This remains the case until perl sees either `\E` or the end of the pattern.

For instance, if we wanted to adapt our `matchtest` program just to look for literal strings, instead of regular expressions, we could change it to look like this:

```
    if (/\Q$pattern\E/) {
```

Now the meaning of + is turned off:

```
> perl matchtest.plx
Enter some text to find: Ent+
'Ent+' was not found.
>
```

Note that all `\Q` does is turn off the regular expression magic of those 14 characters above – it doesn't stop, for example, variable interpolation.

> *Don't forget to change this back again: We'll be using `matchtest.plx` throughout the chapter, to demonstrate the regular expressions we look at. We'll need that magic fully functional!*

Anchors

So far, our patterns have all tried to find a match anywhere in the string. The first way we'll extend our regular expressions is by dictating to perl where the match must occur. We can say 'these characters must match the beginning of the string' or 'this text must be at the end of the string'. We do this by **anchor**ing the match to either end.

The two anchors we have are ^, which appears at the beginning of the pattern anchor a match to the beginning of the string, and $ which appears at the end of the pattern and anchors it to the end of the string. So, to see if our quotation ends in a full stop – and remember that the full stop is a special character – we say something like this:

```
>perl matchtest.plx
Enter some text to find: \.$
The text matches the pattern '\.$'.
```

That's a full stop (which we've escaped to prevent it being treated as a special character) and a dollar sign at the end of our pattern – to show that this must be the end of the string.

Try, if you can, to get into the habit of reading out regular expressions in English. Break them into pieces and say what each piece does. Also remember to say that each piece must immediately follow the other in the string in order to match. For instance, the above could be read 'match a full stop immediately followed by the end of the string'.

If you can get into this habit, you'll find that reading and understanding regular expressions becomes a lot easier, and you'll be able to 'translate' back into Perl more naturally as well.

Here's another example: do we have a capital I at the beginning of the string?

```
> perl matchtest.plx
Enter some text to find: ^I
'^I' was not found.
>
```

We use ^ to mean 'beginning of the string', followed by an I. In our case, though, the character at the beginning of the string is a ", so our pattern does not match. If you know that what you're looking for can only occur at the beginning or the end of the match, it's extremely efficient to use anchors. Instead of searching through the whole string to see whether the match succeeded, perl only needs to look at a small portion and can give up immediately if even the first character does not match.

Let's see one more example of this, where we'll combine looking for matches with looking through the lines in a file:

Try it out : Rhyming Dictionary

Imagine yourself as a poor poet. In fact, not just poor, but downright bad – so bad, you can't even think of a rhyme for 'pink'. So, what do you do? You do what every sensible poet does in this situation, and you write the following Perl program:

```
#!/usr/bin/perl
# rhyming.plx
use warnings;
use strict;

my $syllable = "ink";
while (<>) {
   print if /$syllable$/;
}
```

We can now feed it a file of words, and find those that end in 'ink':

```
>perl rhyming.plx wordlist.txt
blink
bobolink
brink
chink
clink
>
```

For a really thorough result, you'll need to use a file containing every word in the dictionary – be prepared to wait though if you do! For the sake of the example however, any text-based file will do (though it'll help if it's in English). A bobolink, in case you're wondering, is a migratory American songbird, otherwise known as a ricebird or reedbird.

How It Works

With the loops and tests we learned in the last chapter, this program is really very easy:

```
while (<>) {
   print if /$syllable$/;
}
```

We've not looked at file access yet, so you may not be familiar with the `while(<>){...}` construction used here. In this example it opens a file that's been specified on the command line, and loops through it, one line at a time, feeding each one into the special variable `$_` – this is what we'll be matching.

Once each line of the file has been fed into `$_`, we test to see if it matches the pattern, which is our syllable, 'ink', anchored to the end of the line (with `$`). If so, we print it out.

The important thing to note here is that perl treats the 'ink' as the last thing on the line, even though there is a new line at the end of `$_`. Regular expressions typically ignore the last new line in a string – we'll look at this behavior in more detail later.

Shortcuts and Options

All this is all very well if we know exactly what it is we're trying to find, but finding patterns means more than just locating exact pieces of text. We may want to find a three-digit number, the first word on the line, four or more letters all in capitals, and so on.

We can begin to do this using **character classes** – these aren't just single characters, but something that signifies that any one of a *set* of characters is acceptable. To specify this, we put the characters we consider acceptable inside square brackets. Let's go back to our `matchtest` program, using the same test string:

```
$_ = q("I wonder what the Entish is for 'yes' and 'no'," he thought.);
```

```
> perl matchtest.plx
Enter some text to find: w[aoi]nder
The text matches the pattern 'w[aoi]nder'.
>
```

What have we done? We've tested whether the string contains a 'w', followed by either an 'a', an 'o', or an 'i', followed by 'nder'; in effect, we're looking for either of 'wander', 'wonder', or 'winder'. Since the string contains 'wonder', the pattern is matched.

Conversely, we can say that everything is acceptable **except** a given sequence of characters – we can 'negate the character class'. To do this, the character class should start with a `^`, like so:

```
> perl matchtest.plx
Enter some text to find: th[^eo]
'th[^eo]' was not found.
>
```

So, we're looking for 'th' followed by something that is neither an 'e' or an 'o'. But all we have is 'the' and 'thought', so this pattern does not match.

If the characters you wish to match form a sequence in the character set you're using – ASCII or Unicode, depending on your perl version – you can use a hyphen to specify a range of characters, rather than spelling out the entire range. For instance, the numerals can be represented by the character class `[0-9]`. A lower case letter can be matched with `[a-z]`. Are there any numbers in our quote?

```
> perl matchtest.plx
Enter some text to find: [0-9]
'[0-9]' was not found.
>
```

You can use one or more of these ranges alongside other characters in a character class, so long as they stay inside the brackets. If you wanted to match a digit and then a letter from 'A' to 'F', you would say `[0-9][A-F]`. However, to match a single hexadecimal digit, you would write `[0-9A-F]` or `[0-9A-Fa-f]` if you wished to include lower-case letters.

Some character classes are going to come up again and again: the digits, the letters, and the various types of whitespace. Perl provides us with some neat shortcuts for these. Here are the most common ones, and what they represent:

Shortcut	Expansion	Description
`\d`	`[0-9]`	Digits 0 to 9.
`\w`	`[0-9A-Za-z_]`	A 'word' character allowable in a Perl variable name.
`\s`	`[ \t\n\r]`	A whitespace character that is, a space, a tab, a newline or a return.

also, the negative forms of the above:

Shortcut	Expansion	Description
`\D`	`[^0-9]`	Any non-digit.
`\W`	`[^0-9A-Za-z_]`	A non-'word' character.
`\S`	`[^ \t\n\r]`	A non-blank character.

So, if we wanted to see if there was a five-letter word in the sentence, you might think we could do this:

```
> perl matchtest.plx
Enter some text to find: \w\w\w\w\w
The text matches the pattern '\w\w\w\w\w'.
>
```

But that's not right – there are no five-letter words in the sentence! The problem is, we've only asked for five letters in a row, and any word with **at least** five letters contains five in a row will match that pattern. We actually matched 'wonde', which was the first possible series of five letters in a row. To actually get a five-letter word, we might consider deciding that the word must appear in the middle of the sentence, that is, between two spaces:

```
> perl matchtest.plx
Enter some text to find: \s\w\w\w\w\w\s
'\s\w\w\w\w\w\s' was not found.
>
```

Word Boundaries

The problem with that is, when we're looking at text, words aren't always between two spaces. They can be followed by or preceded by punctuation, or appear at the beginning or end of a string, or otherwise next to non-word characters. To help us properly search for words in these cases, Perl provides the special `\b` metacharacter. The interesting thing about `\b` is that it doesn't actually match any character in particular. Rather, it matches the point between something that isn't a word character (either `\W` or one of the ends of the string) and something that is (a word character), hence `\b` for **b**oundary. So, for example, to look for one-letter words:

```
> perl matchtest.plx
Enter some text to find: \s\w\s
'\s\w\s' was not found.

> perl matchtest.plx
Enter some text to find: \b\w\b
The text matches the pattern '\b\w\b'.
```

As the I was preceded by a quotation mark, a space wouldn't match it – but a word boundary does the job. Later, we'll learn how to tell perl how many repetitions of a character or group of characters we want to match without spelling it out directly.

What, then, if we wanted to match anything at all? You might consider something like `[\w\W]` or `[\s\S]`, for instance. Actually, this is quite a common operation, so Perl provides an easy way of specifying it – a full stop. What about an 'r' followed by two characters – any two characters – and then a 'h'?

```
> perl matchtest.plx
Enter some text to find: r..h
The text matches the pattern 'r..h'.
>
```

Is there anything after the full stop?

```
> perl matchtest.plx
Enter some text to find: \..
'\..' was not found.
>
```

What's that? One backslashed full stop to mean a full stop, then a plain one to mean 'anything at all'.

Posix and Unicode Classes

Perl 5.6.0 introduced a few more character classes into the mix – first, those defined by the POSIX (Portable Operating Systems Interface) standard, which are therefore present in a number of other applications. The more common character classes here are:

Shortcut	Expansion	Description
`[[:alpha:]]`	`[a-zA-Z]`	An alphabetic character.
`[[:alnum:]]`	`[0-9A-Za-z]`	An alphabetic or numeric character.
`[[:digit:]]`	`\d`	A digit, 0-9.
`[[:lower:]]`	`[a-z]`	A lower case letter.
`[[:upper:]]`	`[A-Z]`	An upper case letter.
`[[:punct:]]`	`[!"#$%&'()*+,-./:;<=>?@\[\\\]^_`{|}~]`	A punctuation character – note the escaped characters `[`, `\`, and `]`.

The Unicode standard also defines 'properties', which apply to some characters. For instance, the '`IsUpper`' property can be used to match any upper-case character, in whichever language or alphabet. If you know the property you are trying to match, you can use the syntax `\p{}` to match it, for instance, the upper-case character is `\p{IsUpper}`.

Alternatives

Instead of giving a series of acceptable characters, you may want to say 'match either this or that'. The 'either-or' operator in a regular expression is the same as the bitwise 'or' operator, `|`. So, to match either 'yes' or 'maybe' in our example, we could say this:

```
> perl matchtest.plx
Enter some text to find: yes|maybe
The text matches the pattern 'yes|maybe'.
>
```

That's either 'yes' or 'maybe'. But what if we wanted either 'yes' or 'yet'? To get alternatives on part of an expression, we need to group the options. In a regular expression, grouping is always done with parentheses:

```
> perl matchtest.plx
Enter some text to find: ye(s|t)
The text matches the pattern 'ye(s|t)'.
>
```

If we have forgotten the parentheses, we would have tried to match either 'yes' or 't'. In this case, we'd still get a positive match, but it wouldn't be doing what we want – we'd get a match for any string with a 't' in it, whether the words 'yes' or 'yet' were there or not.

You can match either 'this' or 'that' or 'the other' by adding more alternatives:

```
> perl matchtest.plx
Enter some text to find: (this)|(that)|(the other)
'(this)|(that)|(the other)' was not found.
>
```

However, in this case, it's more efficient to separate out the common elements:

```
> perl matchtest.plx
Enter some text to find: th(is|at|e other)
'th(is|at|e other)' was not found.
```

You can also nest alternatives. Say you want to match one of these patterns:

- 'the' followed by whitespace or a letter,
- 'or'

You might put something like this:

```
> perl matchtest.plx
Enter some text to find: (the(\s|[a-z]))|or
The text matches the pattern '(the(\s|[a-z]))|or'.
>
```

It looks fearsome, but break it down into its components. Our two alternatives are:

- `the(\s|[a-z])`
- `or`

The second part is easy, while the first contains 'the' followed by two alternatives: `\s` and `[a-z]`. Hence 'either "the" followed by either a whitespace or a lower case letter, or "or"'. We can, in fact, tidy this up a little, by replacing `(\s|[a-z])` with the less cluttered `[\sa-z]`.

```
> perl matchtest.plx
Enter some text to find: (the[\sa-z])|or
The text matches the pattern '(the[\sa-z])|or'.
>
```

Repetition

We've now moved from matching a specific character to a more general type of character – when we don't know (or don't care) exactly what the character will be. Now we're going to see what happens when we want to talk about a more general quantity of characters: more than three digits in a row; two to four capital letters, and so on. The metacharacters that we use to deal with a number of characters in a row are called **quantifiers**.

Indefinite Repetition

The easiest of these is the question mark. It should suggest uncertainty – something may be there, or it may not. That's exactly what it does: stating that the immediately preceding character(s) – or metacharacter(s) – may appear once, or not at all. It's a good way of saying that a particular character or group is optional. To match the word 'he or she', you can put:

```
> perl matchtest.plx
Enter some text to find: \bs?he\b
The text matches the pattern '\bs?he\b'.
>
```

To make a series of characters (or metacharacters) optional, group them in parentheses as before. Did he say 'what the Entish is' or 'what the Entish word is'? Either will do:

```
> perl matchtest.plx
Enter some text to find: what the Entish (word )?is
The text matches the pattern 'what the Entish (word )?is'.
>
```

Notice that we had to put the space inside the group: otherwise we end up with two spaces between 'Entish' and 'is', whereas our text only has one:

```
> perl matchtest.plx
Enter some text to find: what the Entish (word)? is
'what the Entish (word)? is' was not found.
>
```

As well as matching something one or zero times, you can match something one or more times. We do this with the plus sign – to match an entire word without specifying how long it should be, you can say:

```
> perl matchtest.plx
Enter some text to find: \b\w+\b
The text matches the pattern '\b\w+\b'.
>
```

In this case, we match the first available word – I.

If, on the other hand, you have something which may be there any number of times but might not be there at all – zero or one or many – you need what's called 'Kleene's star': the * quantifier. So, to find a capital letter after any – but possibly no – spaces at the start of the string, what would you do? The start of the string, then any number of whitespace characters, then a capital:

```
> perl matchtest.plx
Enter some text to find: ^\s*[A-Z]
'^\s*[A-Z]' was not found.
>
```

Of course, our test string begins with a quote, so the above pattern won't match, but, sure enough, if you take away that first quote, the pattern will match fine.

Let's review the three qualifiers:

`/bea?t/`	Matches either 'beat' or 'bet'
`/bea+t/`	Matches 'beat', 'beaat', 'beaaat'…
`/bea*t/`	Matches 'bet', 'beat', 'beaat'…

Novice Perl programmers tend to go to town on combinations of dot and star, and the results often surprise them, particularly when it comes to searching-and-replacing. We'll explain the rules of the regular expression matcher shortly, but bear the following in mind:

A regular expression should hardly ever start or finish with a starred character.

You should also consider the fact that `.*` and `.+` in the middle of a regular expression will match as much of your string as they possibly can. We'll look more at this 'greedy' behavior later on.

Well-Defined Repetition

If you want to be more precise about how many times a character or roups of characters might be repeated, you can specify the maximum and minimum number of repeats in curly brackets. '2 or 3 spaces' can be written as follows:

```
> perl matchtest.plx
Enter some text to find: \s{2,3}
'\s{2,3}' was not found.
>
```

So we have no doubled or trebled spaces in our string. Notice how we construct that – the minimum, a comma, and the maximum, all inside braces. Omitting either the maximum or the minimum signifies 'or more' and 'or fewer' respectively. For example, `{2,}` denotes '2 or more', while `{,3}` is '3 or fewer'. In these cases, the same warnings apply as for the star operator.

Finally, you can specify exactly how many things are to be in a row by simply putting that number inside the curly brackets. Here's the five-letter-word example tidied up a little:

```
> perl matchtest.plx
Enter some text to find: \b\w{5}\b
'\b\w{5}\b' was not found.
>
```

Summary Table

To refresh your memory, here are the various metacharacters we've seen so far:

Metacharacter	Meaning
`[abc]`	any one of the characters `a`, `b`, or `c`.
`[^abc]`	any one character other than `a`, `b`, or `c`.

Table continued on following page

Metacharacter	Meaning
`[a-z]`	any one ASCII character between `a` and `z`.
`\d \D`	a digit; a non-digit.
`\w \W`	a 'word' character; a non-'word' character.
`\s \S`	a whitespace character; a non-whitespace character.
`\b`	the boundary between a `\w` character and a `\W` character.
`.`	any character (apart from a new line).
`(abc)`	the phrase '`abc`' as a group.
`?`	preceding character or group may be present 0 or 1 times.
`+`	preceding character or group is present 1 or more times.
`*`	preceding character or group may be present 0 or more times.
`{x,y}`	preceding character or group is present between *x* and *y* times.
`{,y}`	preceding character or group is present at most *y* times.
`{x,}`	preceding character or group is present at least *x* times.
`{x}`	preceding character or group is present *x* times.

Backreferences

What if we want to know what a certain regular expression matched? It was easy when we were matching literal strings: we knew that 'Case' was going to match those four letters and nothing else. But now, what matches? If we have `/\w{3}/`, which three word characters are getting matched?

Perl has a series of special variables in which it stores anything that's matched with a group in parentheses. Each time it sees a set of parentheses, it copies the matched text inside into a numbered variable – the first matched group goes in `$1`, the second group in `$2`, and so on. By looking at these variables, which we call the **backreference** variables, we can see what triggered various parts of our match, and we can also extract portions of the data for later use.

First, though, let's rewrite our test program so that we can see what's in those variables:

Try it out : A Second Pattern Tester

```
#!/usr/bin/perl
# matchtest2.plx
use warnings;
use strict;

$_ = '1: A silly sentence (495,a) *BUT* one which will be useful. (3)';

print "Enter a regular expression: ";
my $pattern = <STDIN>;
chomp($pattern);
```

```
if (/$pattern/) {
    print "The text matches the pattern '$pattern'.\n";
    print "\$1 is '$1'\n" if defined $1;
    print "\$2 is '$2'\n" if defined $2;
    print "\$3 is '$3'\n" if defined $3;
    print "\$4 is '$4'\n" if defined $4;
    print "\$5 is '$5'\n" if defined $5;
} else {
    print "'$pattern' was not found.\n";
}
```

Note that we use a backslash to escape the first 'dollar' symbol in each `print` *statement, thus displaying the actual symbol, while leaving the second in each to display the contents of the appropriate variable.*

We've got our special variables in place, and we've got a new sentence to do our matching on. Let's see what's been happening:

```
> perl matchtest2.plx
Enter a regular expression: ([a-z]+)
The text matches the pattern '([a-z]+)'.
$1 is 'silly'

> perl matchtest2.plx
Enter a regular expression: (\w+)
The text matches the pattern '(\w+)'.
$1 is '1'

> perl matchtest2.plx
Enter a regular expression: ([a-z]+)(.*)([a-z]+)
The text matches the pattern '([a-z]+)(.*)([a-z]+)'.
$1 is 'silly'
$2 is ' sentence (495,a) *BUT* one which will be usefu'
$3 is 'l'

> perl matchtest2.plx
Enter a regular expression: e(\w|n\w+)
The text matches the pattern 'e(\w|n\w+)'.
$1 is 'n'
```

How It Works

By printing out what's in each of the groups, we can see exactly what caused perl to start and stop matching, and when. If we look carefully at these results, we'll find that they can tell us a great deal about how perl handles regular expressions.

How the Engine Works

We've now seen most of the syntax behind regular expression matching and plenty of examples of it in action. The code that does all the matching is called perl's 'regular expression engine'. You might now be wondering about the exact rules applied by this engine when determining whether or not a piece of text matches. And how much of it matches what. From what our examples have shown us, let us make some deductions about the engine's operation.

Our first expression, `([a-z]+)` plucked out a set of one-or-more lower-case letters. The first such set that perl came across was '`silly`'. The next character after '`y`' was a space, and so no longer matched the expression.

- **Rule one**: Once the engine starts matching, it will keep matching a character at a time for as long as it can. Once it sees something that doesn't match, however, it has to stop. In this example, it can never get beyond a character that is not a lower case letter. It has to stop as soon as it encounters one.

Next, we looked for a series of word characters, using (`\w+`). The engine started looking at the beginning of the string and found one, '1'. The next character was not a word character (it was a colon), and so the engine had to stop.

- **Rule two**: Unlike me, the engine is **eager**. It's eager to start work and eager to finish, and it starts matching as soon as possible in the string; if the first character doesn't match, try and start matching from the second. Then take every opportunity to finish as quickly as possible.

Then we tried this: `([a-z]+)(.*)([a-z]+)`. The result we got with this was a little strange. Let's look at it again:

```
> perl matchtest2.plx
Enter a regular expression: ([a-z]+)(.*)([a-z]+)
The text matches the pattern '([a-z]+)(.*)([a-z]+)'.
$1 is 'silly'
$2 is ' sentence (495,a) *BUT* one which will be usefu'
$3 is 'l'
>
```

Our first group was the same as what matched before – nothing new there. When we could no longer match lower case letters, we switched to matching anything we could. Now, this *could* take up the rest of the string, but that wouldn't allow a match for the third group. We have to leave at least one lower-case letter.

So, the engine started to reverse back along the string, giving characters up one by one. It gave up the closing bracket, the 3, then the opening bracket, and so on, until we got to the first thing that would satisfy all the groups and let the match go ahead – namely a lower-case letter: the 'l' at the end of 'useful'.

From this, we can draw up the third rule:

- **Rule three**: Like me, in this case, the engine is **greedy**. If you use the `+` or `*` operators, they will try and steal as much of the string as possible. If the rest of the expression does not match, it grudgingly gives up a character at a time and tries to match again, in order to find the fullest possible match.

We can turn a greedy match into a non-greedy match by putting the `?` operator after either the plus or star. For instance, let's turn this example into a non-greedy version: `([a-z]+)(.*?)([a-z]+)`. This gives us an entirely different result:

```
> perl matchtest2.plx
Enter a regular expression: ([a-z]+)(.*?)([a-z]+)
The text matches the pattern '([a-z]+)(.*?)([a-z]+)'.
$1 is 'silly'
$2 is ' '
$3 is 'sentence'
>
```

Now we've shut off rule three, rule two takes over. The smallest possible match for the second group was a single space. First, it tried to get nothing at all, but then the third group would be faced with a space. This wouldn't match. So, we grudgingly accept the space and try and finish again. This time the third group has some lower case letters, and that can match as well.

What if we turn off greediness in all three groups, and say this: `([a-z]+?)(.*?)([a-z]+?)`

```
> perl matchtest2.plx
Enter a regular expression: ([a-z]+?)(.*?)([a-z]+?)
The text matches the pattern '([a-z]+?)(.*?)([a-z]+?)'.
$1 is 's'
$2 is ''
$3 is 'i'
>
```

What about this? Well, the smallest possible match for the first group is the 's' of silly. We asked it to find one character or more, and so the smallest it could find was one. The second group actually matched no characters at all. This left the third group facing an 'i', which it took to complete the match.

Our last example included an alternation:

```
> perl matchtest2.plx
Enter a regular expression: e(\w|n\w+)
The text matches the pattern 'e(\w|n\w+)'.
$1 is 'n'
>
```

The engine took the first branch of the alternation and matched a single character, even though the second branch would actually satisfy greed. This leads us onto the fourth rule:

- **Rule four**: Again like me, the regular expression engine **hates decisions**. If there are two branches, it will always choose the first one, even though the second one might allow it to gain a longer match.

To summarize:

> **The regular expression engine starts as soon as it can, grabs as much as it can, then tries to finish as soon as it can, while taking the first decision available to it.**

Working with RegExps

Now that we've matched a string, what do we do with it? Well, sometimes it's just useful to know whether a string contains a given pattern or not. However, a lot of the time we're going to be doing search-and-replace operations on text. We'll explain how to do that here. We'll also cover some of the more advanced areas of dealing with regular expressions.

Substitution

Now we know all about matching text, substitution is very easy. Why? Because all of the clever things are in the 'search' part, rather than the 'replace': all the character classes, quantifiers and so on only make sense when matching. You can't substitute, say, a word with any number of digits. So, all we need to do is take the 'old' text, Our match, and tell perl what we want to replace it with. This we do with the `s///` operator.

The `s` is for 'substitute' – between the first two slashes, we put our regular expression as before. Before the final slash, we put our text replacement. Just as with matching, we can use the `=~` operator to apply it to a certain string. If this is not given, it applies to the default variable `$_`:

```
#!/usr/bin/perl
# subst1.plx
use warnings;
use strict;

$_ = "Awake! Awake! Fear, Fire, Foes! Awake! Fire, Foes! Awake!";
# Tolkien, Lord of the Rings

s/Foes/Flee/;
print $_,"\n";
```

```
>perl subst1.plx
Awake! Awake! Fear, Fire, Flee! Awake! Fire, Foes! Awake!
>
```

Here we have substituted the first occurrence of 'Foes' with the word 'Flee'. Had we wanted to change every occurrence, we would have needed to use another modifier. Just as the `/i` modifier for matching case-insensitively, the `/g` modifier on a substitution acts **g**lobally:

```
#!/usr/bin/perl
# subst1.plx
use warnings;
use strict;

$_ = "Awake! Awake! Fear, Fire, Foes! Awake! Fire, Foes! Awake!";
# Tolkien, Lord of the Rings

s/Foes/Flee/g;
print $_,"\n";
```

```
> perl subst1.plx
Awake! Awake! Fear, Fire, Flee! Awake! Fire, Flee! Awake!
>
```

Like the left-hand side of the substitution, the right-hand side also works like a double-quoted string and is thus subject to variable interpolation. One useful thing, though, is that we can use the backreference variables we collected during the match on the right hand side. So, for instance, to swap the first two words in a string, we would say something like this:

```
#!/usr/bin/perl
# subst2.plx
use warnings;
use strict;

$_ = "there are two major products that come out of Berkeley: LSD and UNIX";
# Jeremy Anderson

s/(\w+)\s+(\w+)/$2 $1/;
print $_, "?\n";
```

```
>perl subst2.plx
are there two major products that come out of Berkeley: LSD and UNIX?
>
```

What would happen if we tried doing that globally? Well, let's do it and see:

```
#!/usr/bin/perl
# subst2.plx
use warnings;
use strict;

$_ = "there are two major products that come out of Berkeley: LSD and UNIX";
# Jeremy Anderson

s/(\w+)\s+(\w+)/$2 $1/g;
print $_, "?\n";
```

```
>perl subst2.plx
are there major two that products out come Berkeley of: and LSD UNIX?
>
```

Here, every word in a pair is swapped with its neighbor. When processing a global match, perl always starts where the previous match left off.

Changing Delimiters

You may have noticed that `//` and `s///` looks like `q//` and `qq//`. Well, just like `q//` and `qq//`, we can change the delimiters when matching and substituting to increase the readability of our regular expressions. The same rules apply: Any non-word character can be the delimiter, and paired delimiters such as `<>`, `()`, `{}`, and `[]` may be used – with two provisos.

First, if you change the delimiters on `//`, you must put an `m` in front of it. (m for 'match'). This is so that perl can still recognize it as a regular expression, rather than a block or comment or anything else.

Second, if you use paired delimiters with substitution, you must use two pairs:

```
s/old text/new text/g;
```

becomes:

```
s{old text}{new text}g;
```

You may, however, leave spaces or new lines between the pairs for the sake of clarity:

```
s{old text}
 {new text}g;
```

The prime example of when you would want to do this is when you are dealing with file paths, which contain a lot of slashes. If you are, for instance, moving files on your Unix system from `/usr/local/share/` to `/usr/share/`, you may want to munge the file names like this:

```
s/\/usr\/local\/share\//\/usr\/share\//g;
```

However, it's far easier and far less ugly to change the delimiters in this case:

```
s#/usr/local/share/#/usr/share/#g;
```

Modifiers

We've already seen the `/i` modifier used to indicate that a match should be case insensitive. We've also seen the `/g` modifier to apply a substitution. What other modifiers are there?

- `/m` – treat the string as multiple lines. Normally, `^` and `$` match the very start and very end of the string. If the `/m` modifier is in play, then they will match the starts and ends of individual lines (separated by `\n`). For example, given the string: `"one\ntwo"`, the pattern `/^two$/` will not match, but `/^two$/m` will.
- `/s` – treat the string as a single line. Normally, `.` does not match a new line character; when `/s` is given, then it will.
- `/g` – as well as globally replacing in a substitution, allows us to match multiple times. When using this modifier, placing the `\G` anchor at the beginning of the regexp will anchor it to the end point of the last match.
- `/x` – allow the use of whitespace and comments inside a match.

Regular expressions can get quite fiendish to read at times. The `/x` modifier is one way to stop them becoming so. For instance, if you're matching a string in a log file that contains a time, followed by a computer name in square brackets, then a message, the expression you'll create to extract the information may easily end up looking like this:

```
# Time in $1, machine name in $2, text in $3
/^([0-2]\d:[0-5]\d:[0-5]\d)\s+\[([^\]]+)\]\s+(.*)$/
```

However, if you use the /x modifier, you can stretch it out as follows:

```
/^
(         # First group: time
   [0-2]\d
   :
   [0-5]\d
   :
   [0-5]\d
   )
\s+
\[      # Square bracket
   (         # Second group: machine name
   [^\]]+   # Anything that isn't a square bracket
   )
\]      # End square bracket


\s+
   (         # Third group: everything else
   .*
   )
$/x
```

Another way to tidy this up is to put each of the groups into variables and interpolate them:

```
my $time_re = '([0-2]\d:[0-5]\d:[0-5]\d)';
my $host_re = '\[[^\]]+)\]';
my $mess_re = '(.*)';

/^$time_re\s+$host_re\s+$mess_re$/;
```

Split

We briefly saw `split` earlier on in the chapter, where we used it to break up a string into a list of words. In fact, we only saw it in a very simple form. Strictly speaking, it was a bit of a cheat to use it at all. We didn't see it then, but `split` was actually using a regular expression to do its stuff!

Using `split` on its own is equivalent to saying:

```
split /\s+/, $_;
```

which breaks the default string `$_` into a **list** of substrings, using whitespace as a delimiter. However, we can also specify our own regular expression: perl goes through the string, breaking it whenever the regexp matches. The delimiter itself is thrown away.

For instance, on the UNIX operating system, configuration files are sometimes a list of fields separated by colons. A sample line from the password file looks like this:

```
kake:x:10018:10020::/home/kake:/bin/bash
```

To get at each field, we can split when we see a colon:

```
#!/usr/bin/perl
# split.plx
use warnings;
use strict;

my $passwd = "kake:x:10018:10020::/home/kake:/bin/bash";
my @fields = split /:/, $passwd;
print "Login name : $fields[0]\n";
print "User ID : $fields[2]\n";
print "Home directory : $fields[5]\n";
```

```
>perl split.plx
Login name : kake
User ID : 10018
Home directory : /home/kake
>
```

Note that the fifth field has been left empty. Perl will recognize this as an empty field, and the numbering used for the following entries takes account of this. So `$fields[5]` returns `/home/kake`, as we'd otherwise expect. Be careful though – if the line you are splitting contains empty fields at the end, they will get dropped.

Join

To do the exact opposite, we can use the `join` operator. This takes a specified delimiter and interposes it between the elements of a specified array. For example:

```
#!/usr/bin/perl
# join.plx
use warnings;
use strict;

my $passwd = "kake:x:10018:10020::/home/kake:/bin/bash";
my @fields = split /:/, $passwd;
print "Login name : $fields[0]\n";
print "User ID : $fields[2]\n";
print "Home directory : $fields[5]\n";

my $passwd2 = join "#", @fields;
print "Original password : $passwd\n";
print "New password :      $passwd2\n";
```

```
>perl join.plx
Login name : kake
User ID : 10018
Home directory : /home/kake
Original password : kake:x:10018:10020::/home/kake:/bin/bash
New password :      kake#x#10018#10020##/home/kake#/bin/bash
>
```

Transliteration

While we're looking at regular expressions, we should briefly consider another operator. While it's not directly associated with regexps, the transliteration operator has a lot in common with them and adds a very useful facility to the matching and substitution techniques we've already seen.

What this does is to correlate the characters in its two arguments, one by one, and use these pairings to substitute individual characters in the referenced string. It uses the syntax `tr/one/two/` and (as with the matching and substitution operators) references the special variable `$_` unless otherwise specified with `=~` or `!~`. In this case, it replaces all the 'o's in the referenced string with 't's, all the 'n's with 'w's, and all the 'e's with 'o's.

Let's say you wanted to replace, for some reason, all the numbers in a string with letters. You might say something like this:

```
$string =~ tr/0123456789/abcdefghij/;
```

This would turn, say, "2011064" into "cabbage". You can use ranges in transliteration but not in any of the character classes. We could write the above as:

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

The return value of this operator is, by default, the number of characters matched with those in the first argument. You can therefore use the transliteration operator to count the number of occurrences of certain characters. For example, to count the number of vowels in a string, you can use:

```
my $vowels = $string =~ tr/aeiou//;
```

Note that this will not actually substitute any of the vowels in the variable `$string`. As the second argument is blank, there is no correlation, so no substitution occurs. However, the transliteration operator can take the `/d` modifier, which *will* delete occurrences on the left that do not have a correlating character on the right. So, to get rid of all spaces in a string quickly, you could use this line:

```
$string =~ tr/ //d;
```

Common Blunders

There are a few common mistakes people tend to make when writing regexps. We've already seen that `/a*b*c*/` will happily match any string at all, since it matches each letter zero times. What else can go wrong?

- **Forgetting To Group**
  `/Bam{2}/` will match 'Bamm', while `/(Bam){2}/` will match 'BamBam', so be careful when choosing which one to use. The same goes for alternation: `/Simple|on/` will match 'Simple' and 'on', while `/Sim(ple|on)/` will match both 'Simple' and 'Simon' Group each option separately.
- **Getting The Anchors Wrong**
  `^` goes at the beginning, `$` goes at the end. A dollar anywhere else in the string makes perl try and interpolate a variable.

- **Forgetting To Escape Special Characters**.
  Do you want them to have a special meaning? These are the characters to be careful of: `. * ? + [ ] ( ) { } ^ $ | and of course \ itself.`
- **Not Counting from Zero**
  The first entry in an array is given the index zero.
- **Counting from Zero**
  I know, I know! All along I've been telling you that computers start counting from zero. Nevertheless, there's always the odd exception – the first backreference is `$1`. Don't blame Perl though – it took this behavior from a language called `awk` which used `$1` as the first reference variable.

More Advanced Topics

We've not actually plumbed the depths of the regular expression language syntax – Perl has a habit of adding wilder and more bizarre features to it on a regular basis. All of the more off-the-wall extensions begin with a question mark in a group – this is supposed to make you stop and ask yourself: 'Do I really want to do this?'

Some of these are experimental and may change from perl version to version (and may soon disappear altogether), but there are others that aren't so tricky. Some of these are extremely useful, so let's dive in!

Inline Comments

We've already seen how we can use the `/x` modifier to add comments and whitespace to our regular expressions. We can also do this with the `(?#)` pattern:

```
/^Today's (?# This is ignored, by the way)date:/
```

Unfortunately, there's no way to have parentheses inside these comments, since perl closes the comment as soon as it sees a closing bracket. If you want to have longer or more detailed comments, you should consider using the `/x` modifier instead.

Inline Modifiers

If you are reading patterns from a file or constructing them from inside your code, you have no way of adding a modifier to the end of the regular expression operator. For example:

```
#!/usr/bin/perl
# inline.plx
use warnings;
use strict;

my $string = "There's more than One Way to do it!";

print "Enter a test expression: ";
my $pat = <STDIN>;
chomp($pat);

if ($string =~ /$pat/) {
print "Congratulations! '$pat' matches the sample string.\n";
} else {
print "Sorry. No match found for '$pat'";
}
```

If we run this and momentarily forgot how our sample string had been capitalized, we might get this:

```
>perl inline.plx
Enter a test expression: one way to do it!
Sorry. No match found for 'one way to do it!'
>
```

So how can we make this case-insensitive? The solution is to use an inline modifier, the syntax for which is `(?i)`. This will make the enclosing group match case-insensitively. Therefore we have:

```
>perl inline.plx
Enter a test expression: (?i)one way to do it!
Congratulations! '(?i)one way to do it!' matches the sample string.
>
```

If, conversely, you have a modifier in place that you temporarily want to get rid of, you can say, for example, `(?-i)` to turn it off. If we have this:

```
/There's More Than ((?-i)One Way) To Do It!/i;
```

the words 'One Way' alone are matched case-sensitively.

Note that you can also inline the `/m`, `/s`, and `/x` modifiers in the same way.

Grouping without Backreferences

Parentheses perform the function of grouping and populating the backreference variables. If you have a portion of your match in parentheses, it will, if successful, be placed in one of the numbered variables. However, there may be times when you only want to use brackets for grouping. For example, you're expecting the first backreference to contain something important, but there may be some preceding text in the way. You could have something like this:

```
/(X-)?Topic: (\w+)/;
```

You can't be certain whether your first defined backreference is going to end up in `$1` or `$2` – it depends on whether the '`X-`' part is present or not. For example, if we tried to match the string "Topic: the weather", we'd find that `$1` was left undefined. If we'd tried to do something with its contents, we'd get the warning:

```
Use of uninitialized value in concatenation
```

Now that's not necessarily a problem here. After all, we'll find our word in `$2` whether or not there's anything preceding `"Topic: "`. Surely we can just be careful not to use `$1`?

But what if there's more than one optional field? Say we had an expression that left all but the 2nd and 6th groups optional. We then have to look in $2 for our first word and $6 for our second, while $1, $3, $4, and $5 are left undefined. This really isn't good programming style and *is* asking for trouble! We really shouldn't backreference fields if we don't need to.

We can resolve this problem very simply, by adding the characters `?:` like this:

```
/(?:X-)?Topic: (\w+)/;
```

This ensures that the first set of brackets will now group only and not fill a backreference variable. Our word will always be put into `$1`.

Lookaheads and Lookbehinds

Sometimes you may want to say something along the lines of 'substitute the word "fish" with "cream", but only if the next word is "cake".' You can do this very simply by saying:

```
s/fish cake/cream cake/
```

What does this do? The regular expression engine scans a referenced string, looking for a match on "fish cake" On finding one, it substitutes the text "cream cake". Not too bad – it does the job. In this case it's not too big a deal that it has to substitute five characters from each match with five *identical* characters from the substitution string. It's not hard to see how this sort of inefficiency could really start to bog a program down if we used substitutions excessively.

What we want is a way of putting an assertion into the match – a 'match the text *only if* the next word is "cake"' clause – without actually matching the assertion itself. Having matched "fish", we really just want to *look ahead*, to see if it says " cake" (and give the match a thumbs-up if it does), then forget about "cake" altogether.

In life, that's not so easy. Fortunately in Perl we have an operator for just this sort of thing:

```
/fish(?= cake)/
```

will match exactly what we want – it looks for "fish", does a positive lookahead on " cake", and matches "fish" only if that succeeds. For example:

```
#!/usr/bin/perl
# look1.plx
use warnings;
use strict;

$_ = "fish cake and fish pie";
print "Our original order was ", $_, "\n";

s/fish(?= cake)/cream/;
print "Actually, make that ", $_, " instead.\n";
```

will return

```
>perl look1.plx
Our original order was fish cake and fish pie
Actually, make that cream cake and fish pie instead.
>
```

We can also look ahead negatively, by using an exclamation mark instead of the equals sign:

```
/fish(?! cake)/
```

which will match "fish" only if the following word is *not* " cake". If we adapt `look1.plx` like so:

```
#!/usr/bin/perl
# look2.plx
use warnings;
use strict;

$_ = "fish cake and fish pie";
print "Our original order was ", $_, "\n";

s/fish(?! cake)/cream/;
print "Actually, make that ", $_, " instead.\n";
```

then sure enough, it's "fish pie" that gets matched this time and not "fish cake".

```
>perl look2.plx
Our original order was fish cake and fish pie
Actually, make that fish cake and cream pie instead.
>
```

Lookaheads are very powerful as you'll soon discover if you experiment a little, particularly when you start to use less specific expressions (using metacharacters) with them.

However, we may also wish to look at the text preceding a matched pattern. We therefore have a similar pair of **lookbehind** operators. We now use the < sign to point 'behind' the match, matching "cake" only if "fish" *precedes* it. So to find all those boring old fish cakes, we use:

```
/(?<=fish )cake/
```

but to find all the cream cakes and chocolate cakes, do this:

```
/(?<!fish )cake/
```

Let's have fish and chips instead of our fish cakes and cream doughnuts instead of cream cakes:

```
#!/usr/bin/perl
# look3.plx
use warnings;
use strict;

$_ = "fish cake and cream cake";
print "Our original order was ", $_, "\n";

s/(?<=fish )cake/and chips/;
print "No, wait. I'll have ", $_, " instead\n";

s/(?<!fish )cake/slices/;
print "Actually, make that ", $_, ", will you?\n";
```

```
>perl look3.plx
Our original order was fish cake and cream cake
No, wait. I'll have fish and chips and cream cake instead
Actually, make that fish and chips and cream slices, will you?
>
```

One very important thing to note about lookbehind assertions is that they can only handle fixed-width expressions. So while you *can* use most of the metacharacters, indeterminate quantifiers like `.`, `?`, and `*` aren't allowed.

Backreferences (again)

Finally, in our tour of regular expressions, let's look again at backreferences. Suppose you want to find any repeated words in a string. How would you do it? You might think about doing this:

```
if (/\b(\w+) $1\b/) {
   print "Repeated word: $1\n";
}
```

Except, this doesn't work, because `$1` is only set when the match is complete. In fact, if you have warnings turned on, you'll be alerted to the fact that `$1` is undefined every time. In order to match while still inside the regular expression, you need to use the following syntax:

```
if (/\b(\w+) \1\b/) {
   print "Repeated word: $1\n";
}
```

However, when you're replacing, you'll get a warning if you try and use the `\<number>` syntax on the wrong side. It'll work, but you'll be told "`\1 better written as $1`".

Summary

Regular expressions are quite possibly the most powerful means at your disposal of looking for patterns in text, extracting sub-patterns and replacing portions of text. They're the basis of any text shuffling you do in Perl, and they should be your first port of call when you need to do some string manipulation.

In this chapter, we've seen how to match simple text, different classes of text, and then different amounts of text. We've also seen how to provide alternative matches, how to refer back to portions of the match, and how to substitute and transliterate text.

The key to learning and understanding regular expressions is to be able to break them down into their component parts and unravel the language, translating it piecewise into English. Once you can fluently read out the intention of a complex regular expression, you're well on your way to creating powerful matches of your own.

You can find a summary of regular expression syntax in Appendix A. Section 6 of the Perl FAQ (at www.perl.com) contains a good selection of regexp hints and tricks.

Exercises

1. Write out English descriptions of the following regular expressions, and describe what the operations actually do:

```
$var =~ /(\w+)$/

$code !~ /^#/

s/#{2,}/#/g
```

2. Using the contents of the gettysburg.txt file (provided in the download for Chapter 6), use regular expressions to do the following, and print out the result. (Tip: use a here-document to store the text in your file):

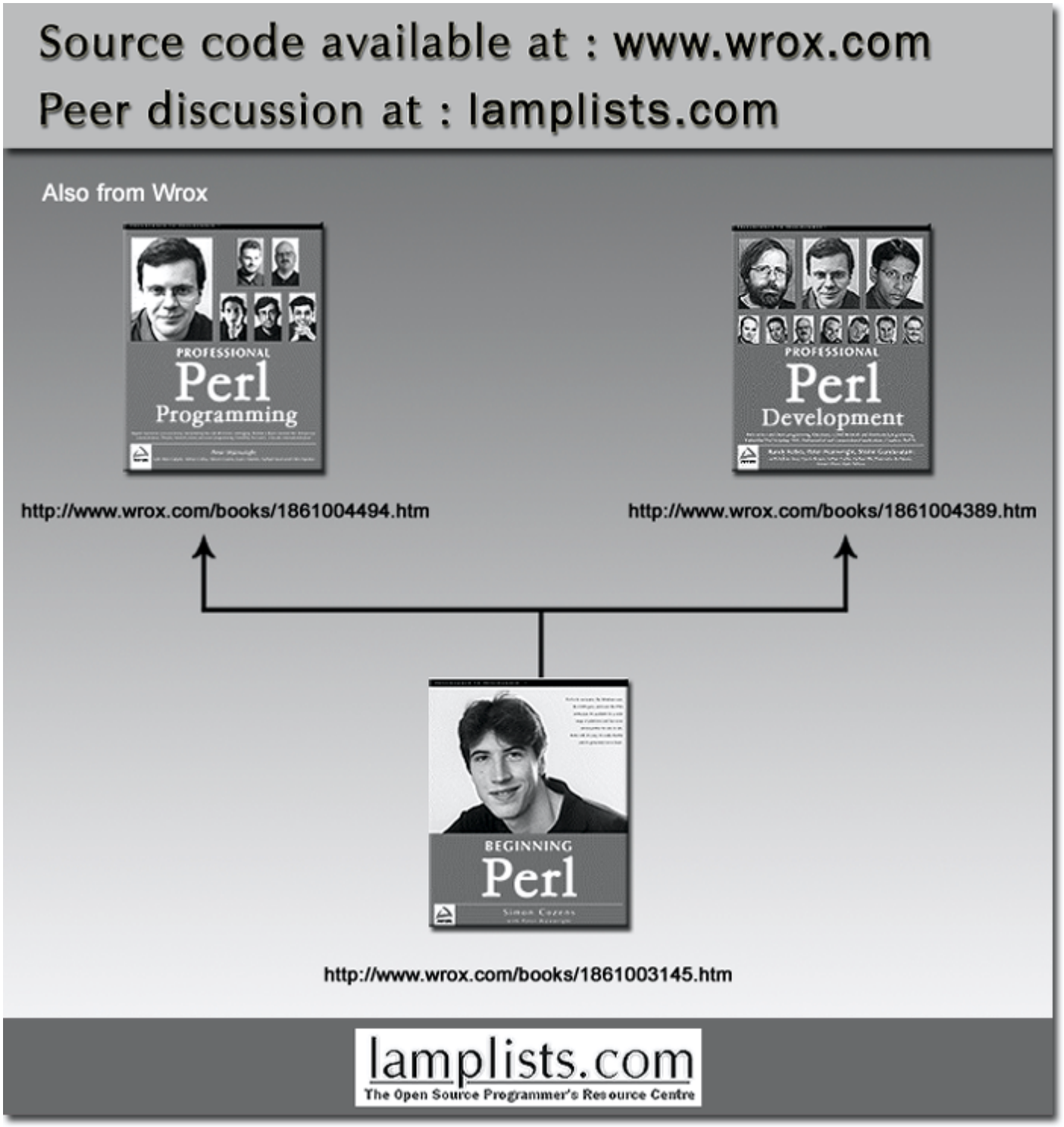
   a. Count the number of occurences of the word 'we'.

   b. Reformat the text, so that each sentences is displayed as a separate paragraph.

   c. Check that there are no multiple spaces in the text, replacing any with single spaces.

3. When we use groups, the // operator returns a list of all the text strings that have been matched. Modify our example program matchtest2.plx, so that it produces its output from this list, rather than using special variables.

4. If we want to sort a list of words into alphabetical order, one simple and quite effective way is to write a program that performs a 'bubble sort': working through the whole list, it compares each pair of consecutive words; if it finds them in the wrong order, it swaps them over. On reaching the end of the list it repeats the process – unless the previous scan didn't yield any swaps, in which case the list is already properly ordered. Use regular expressions along with the other techniques you've seen so far, and write this program so that it will work with a list of words separated by newline characters. One small hint – the pos() function may come in useful here. You can use this to adjust the position of the \G boundary, for example: pos($var) = 10 will set it just after the tenth character in $var. A subsequent global search will therefore start from this point.



7

References

Way back in Chapter 2 we learned that we couldn't get away with putting one list inside another. Perl flattens lists, and an inner list would get subsumed into whatever we try to put it inside. Similarly, hashes have a single scalar key attached to a single scalar value; there's apparently no way we can put several pieces of data in one hash key.

However, these are both things we'll want to do from time to time. For instance, we might want to represent a chessboard as eight lists of eight squares so that we can address each square by row and column. We might also want to store information about someone – their address, phone number, and occupation – and key it to their name.

Of course, we've seen ways we could do this already: We could store our chessboard as an array of 64 squares, and write some code to convert between row-and-column co-ordinates and a number from 0 to 63. For the address book, we could just use three hashes, each using the same set of names as keys – not a terribly elegant solution but one that does the job with the techniques we've seen so far.

However, in this chapter, we're going to be looking at a very powerful facility in Perl that lets us do this sort of thing and a whole lot more besides – **references**.

What Is a Reference?

Put at its very simplest, a reference is a piece of data that tells us the location of another piece of data. If I told you to "see the first paragraph on page 130", I'd effectively be giving you a reference to the text in that paragraph. It wouldn't be the text itself, but it would tell us where to find it. This would also let us talk about (refer to) the text right away, despite the fact that it's somewhere else in the book. That's why references are so useful – we can specify data once, and they let us access it from wherever else we are.

In Perl, a reference is always a scalar, although the data it refers to may not be: our cross-reference above wasn't even a sentence, but referred to an entire paragraph. Likewise, a reference, even though it's only a scalar, can talk about the data stored in an array or hash.

Languages like C and C++ have a feature that's similar to references, called **pointers**. Now if you're familiar with pointers, please try and put the knowledge aside while you're going through this chapter. They are similar to references in that both point us to locations in the computer's memory. However, pointers tend to leave interpretation of what's there for the programmer to disentangle. References, on the other hand, only store memory locations for specific, clearly defined data structures – maybe not *pre*defined, but defined nevertheless. They allow you to leave the arrangement of computer memory to the computer itself. For me, this is a huge relief, as the machine's far better at that sort of thing than I am.

The main use we have for references is the one we discussed above – as flat-pack storage for arrays and hashes. We can now refer *unambiguously* to the contents of an array or a hash, using a single scalar, so we're now in a position to do things like putting hashes inside hashes, lists inside lists, even hashes in lists, and vice-versa. But that's not all…

Anonymity

We can also use references to create **anonymous data**. Anonymous data, as you might have guessed, is data that doesn't have a variable name attached to it. Instead, it's placed at a certain memory location, and we're given a simple reference to that location. Our list (or hash or whatever) has no name to speak of, but we know exactly where to find it, should we need to use it.

> *This is a bit like literal data, where we had literal scalars and lists in our program, but not quite – literal data was constant: we couldn't change it.*

For example, instead of creating an array `(1,2,3)` called `@array` and then creating a reference to `@array`, we can cut out the middle man, by referencing `(1,2,3)` directly.

This lets us create real scalars, arrays, and hashes, containing data that we can refer to and modify, just as if it were a normal variable. This doesn't mean that we leave arrays and hashes floating about randomly in our program to be plucked out of the air whenever we need them. We know where to find this anonymous data (we have a reference that's telling us just this), and it only exists for as long as part of our program is using it.

The Lifecycle of a Reference

To understand how we deal with references, let's look at the three areas of a reference's life cycle – creation, use, and destruction. After that, we'll see how we can practically use references to create more complicated data structures than simple arrays and hashes.

Reference Creation

There are two ways to create a reference, one for each of the following situations:

- You've already got the data in a variable.
- You want to use anonymous data to go straight to a reference.

The simple rule for the first situation where the variable is already defined is:

> **You create a reference by putting a backslash in front of the variable.**

That's it. Let's see some examples:

```
my @array   = (1, 2, 3, 4, 5);
my $array_r = \@array;
```

We create a perfectly normal array variable and then take a reference to it by putting a backslash before the variable's name. That's literally all there is to it. In the same way, we can create a reference to a hash:

```
my %hash   = ( apple => "pomme", pear => "poire" );
my $hash_r = \%hash;
```

or a scalar:

```
my $scalar   = 42;
my $scalar_r = \$scalar;
```

We can treat our references just like ordinary scalars, so we can put them in an array:

```
my $a = 3;
my $b = 4;
my $c = 5;
my @refs = (\$a, \$b, \$c);
```

Or, if you don't like putting so many backslashes in your array definitions, you can declare this kind of array in a second way:

```
my @refs=\($a, $b, $c);
```

So, if you try referencing a list, you won't actually get a reference to the list, but rather a list of references to each element *in* the list. If this isn't what you want, you can always put the data into an array. We can also put references in a hash, but only as values. Perl doesn't yet support references as hash keys. You can certainly do this, though:

```
my @english = qw(January February March April May June);
my @french  = qw(Janvier Fevrier  Mars  Avril Mai Juin);
my %months  = ( english => \@english, french => \@french );
```

So what does this give us? We have a hash with two keys, `english` and `french`. The `english` key contains a reference to an array of English month names, while the `french` key contains a reference to an array of French month names. With these references, we can access and modify the original data, which means that, in effect, we've stored two arrays inside a single hash.

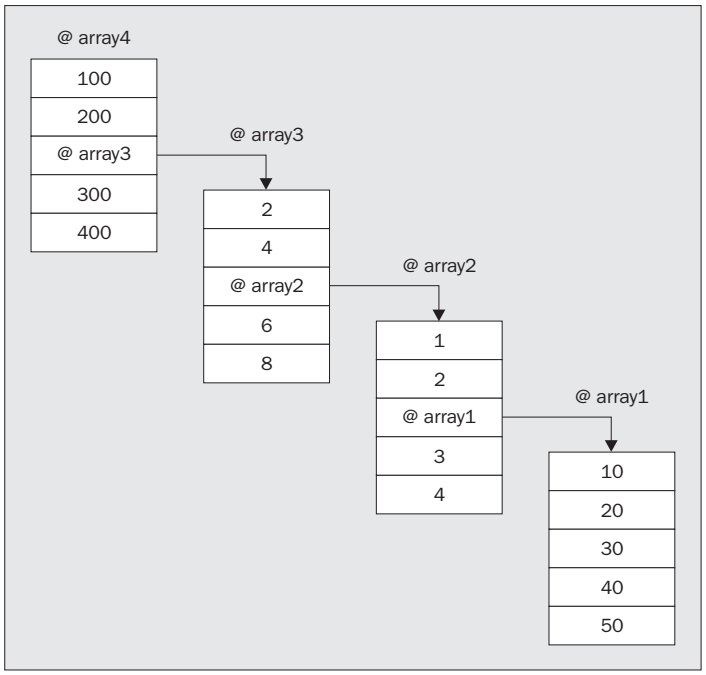
We can use the same trick to store arrays inside arrays:

```
my @array1 = ( 10, 20, 30, 40 );
my @array2 = ( 1, 2, \@array1, 3, 4);
```

Now `@array2` is made up of five scalars, and the middle one is a reference to another array. We can do this over and over again, if we want to:

```
my @array3 = (2, 4, \@array2, 6, 8);
my @array4 = (100, 200, \@array3, 300, 400);
```

This gives us a very versatile way to store complex data structures. What we've just done is to store a structure that looks like this:



Anonymous References

Our next step is to do all this without having to go through the interim stages of creating the variables. Anonymous references will let us go straight from our raw data to a reference, and the rules here are just as simple:

> **To get an array reference instead of an array, use square brackets `[]` instead of parentheses.**
>
> **To get a hash reference instead of a hash, use curly braces `{}` instead of parentheses.**

So, referring to our examples above, instead of doing this:

```
my @array   = (1, 2, 3, 4, 5);
my $array_r = \@array;
```

we can go straight to an array reference like this:

```
my $array_r = [1, 2, 3, 4, 5];
```

Likewise, to get a hash reference, instead of doing this:

```
my %hash   = ( apple => "pomme", pear => "poire" );
my $hash_r = \%hash;
```

we say:

```
my $hash_r = { apple => "pomme", pear => "poire" };
```

We can put anonymous references inside hashes and arrays, just like references created from variables:

```
my %months  = (
   english => ["January", "February", "March", "April", ",May", ",June"],
   french  => ["Janvier", "Fevrier", "Mars", "Avril", "Mai", "Juin"]
);
```

And we can put references inside references:

```
my @array = ( 100,200,[ 2,4,[ 1,2,[ 10,20,30,40,50 ],3,4 ],6,8 ],300,400 );
```

That's exactly the same structure as we created above. Here it is again, with a lot more spacing added:

```
my @array = ( 100, 200,
                [ 2, 4,
                   [ 1, 2,
                      [ 10, 20, 30, 40, 50 ],
                   3, 4 ],
                6, 8 ],
             300, 400 );
```

What about creating an anonymous scalar – what happens if we try this? Well, as we saw above, trying to create a reference to a list gives us a list of references to the list's elements. So if we did this:

```
my @refs = \(1, 2, 3, 4);
```

we'd expect it to give us four references, to `1`, `2`, `3`, and `4`. Perl does in fact do this, but while it will let us retrieve the numbers, it won't allow us to change them – it's almost like trying to modify a literal in your variable. If we ever want to get a scalar reference, it's best to use a temporary variable.

Using References

Once we've created our references (whether to real variables or anonymous data), we're going to want to use them. So how do we access the data? The operation we use to get data back from a reference is called **dereferencing**, and once again, the rule's very simple:

> **To dereference data, put the reference in curly braces wherever you would normally use a variable's name.**

First, we'll see how to do this with arrays. Say we've got an array and a reference:

```
my @array   = (1, 2, 3, 4, 5);
my $array_r = \@array;
```

We can get at the array like this:

```
my @array2  = @{$array_r};
```

We put the reference, `$array_r`, inside curly braces, and use that instead of our original array variable `@array`. We can use this dereferenced array, `@{$array_r}`, anywhere we might otherwise use an array:

Try It Out : Constructing and Dereferencing

For our first attempt, we'll do something simple. We'll just create a reference to an array, then use it as we'd normally use an array:

```
#!/usr/bin/perl
# deref1.plx
use warnings;
use strict;

my @array   = (1, 2, 3, 4, 5);
my $array_r = \@array;

print "This is our dereferenced array: @{$array_r}\n";
for (@{$array_r}) {
   print "An element: $_\n";
}
print "The highest element is number $#{$array_r}\n";
print "This is what our reference looks like: $array_r\n";
```

Let's run this:

```
>perl deref1.plx
This is our dereferenced array: 1 2 3 4 5
An element: 1
An element: 2
An element: 3
An element: 4
An element: 5
The highest element is number 4
This is what our reference looks like: ARRAY(0xa063fbc)
>
```

How It Works

We've seen a few examples of creating references now, so you should be familiar with this syntax. First, we define an array variable and its contents and then backslash it to create a reference to it.

```
my @array   = (1, 2, 3, 4, 5);
my $array_r = \@array;
```

Now we can use `@{$array_r}` instead of `@array`. Both refer to exactly the same data, and both do exactly the same things. For instance, `@{$array_r}` will interpolate inside double quotes:

```
print "This is our dereferenced array: @{$array_r}\n";
```

Just as if we'd used the original `@array`, our dereferenced array prints out the contents of the array, separated by spaces:

```
This is our dereferenced array: 1 2 3 4 5
```

In the same way, we can use the array in a `for` loop, with no surprises:

```
for (@{$array_r}) {
   print "An element: $_\n";
}
```

Finally, we can also get the highest element number in the array, just as if we'd said `$#array`, like this:

```
print "The highest element is number $#{$array_r}\n";
```

Now, we take a look at what our reference actually looks like itself. After all, it's a scalar, so it must have a value that we can print out and look at. It does, and this is what we get if we print out the reference:

```
This is what our reference looks like: ARRAY(0xa063fbc)
```

Well, the ARRAY part obviously tells us that we have an array reference, but what about the part in brackets? Well, we know that a reference is a memory location, telling us where the data is stored in the computer's memory. We generally don't need to worry about this actual value, as we can't do that much with it. Note also that it's unlikely that you'll get exactly the same value as I have here. It will simply depend on what hardware your system has, what other software you're running, and what perl is doing.

> *There is one way you might want to make use of this value directly: to see if two references refer to the same piece of data, you can compare them as numbers using ==.*

If we try and manipulate it, it ceases to be a reference and becomes an ordinary number – the value of the hexadecimal above. We can see that if we run the following program:

```
#!/usr/bin/perl
# noref.plx
use warnings;
use strict;

my $ref = [1, 2, 3];
print "Before: $ref\n";
print "@{$ref}\n";
$ref++;
print "After: $ref\n";
print "@{$ref}\n";
```

will give us something like this:

```
>perl noref.plx
Before: ARRAY(0xa041160)
1 2 3
After: 168038753
Can't use string ("168038753") as an ARRAY ref while "strict refs" in use at noref.plx line 11.
>
```

When we tried to modify our reference, it degenerated to the ordinary number `168038752`, which is the `0xa041160` mentioned above. Adding one to that gave us the number above, which is an ordinary string, rather than a reference. Perl then complains if we try and use it as a reference.

This is why we can't use references as hash keys – these can only be strings, so our references will get 'stringified' to something like the form above. Once that happens, we're not able to use them as references again.

Array Elements

What about the individual elements in an array? How do we access these? Well, the rule is pretty much the same as for the array as a whole; just use the reference in curly braces in the same way you would the array name:

```
#!/usr/bin/perl
# deref2.plx
use warnings;
use strict;

my @band = qw(Crosby Stills Nash Young);
my $ref  = \@band;
for (0..3) {
   print "Array    : ", $band[$_]  , "\n";
   print "Reference: ", ${$ref}[$_], "\n";
}
```

As you can see, these refer to the same thing:

```
>perl deref2.plx
Array     : Crosby
Reference : Crosby
Array     : Stills
Reference : Stills
Array     : Nash
Reference : Nash
Array     : Young
Reference : Young
>
```

The important thing to note here is that these are not two different arrays – they are two ways of referring to the *same* piece of data. This is very important to remember when we start modifying references.

Reference Modification

If we want to modify the data referred to by a reference, the same rule applies as before. Replace the name of the array with the reference in curly brackets. However, when we do this, the data in the original array will change, too:

```
#!/usr/bin/perl
# modify1.plx
use warnings;
use strict;

my @band = qw(Crosby Stills Nash Young);
my $ref  = \@band;
print "Band members before: @band\n";
pop @{$ref};
print "Band members after: @band\n";
```

```
>perl modify1.plx
Band members before: Crosby Stills Nash Young
Band members after: Crosby Stills Nash
>
```

We can still use `push`, `pop`, `shift`, `unshift` (and so on) to manipulate the array. However, in doing so, we'll also be changing what's stored in `@band`.

It's quite possible to have multiple references to the same data. Just as before, if you use one to change the data, you change it for the others, too. This will give the same results as before:

```
my @band = qw(Crosby Stills Nash Young);
my $ref1 = \@band;
my $ref2 = \@band;
print "Band members before: @band\n";
pop @{$ref1};
print "Band members after: @{$ref2}\n";
```

The same goes for anonymous references:

```
my $ref1 = [qw(Crosby Stills Nash Young)];
my $ref2 = $ref1;
print "Band members before: @{$ref2}\n";
pop @{$ref1};
print "Band members after: @{$ref2}\n";
```

Notice here that we're using `[qw(…)]`, which is the same as saying

```
[('Crosby', 'Stills', 'Nash', 'Young')]
```

and the brackets inside get removed, just like when we said `((1,2,3))` back in Chapter 3.

Because anonymous references give us a reference straight away, it's possible to say things like:

```
@{[ 1, 2, 3 ]}
```

This little bit of trickery (thanks to Randal Schwartz) will, of course, give us the list 1, 2, 3. However, it's less useless than it seems. An array dereference will interpolate just like an ordinary array, so you can use this to make functions interpolate inside strings. For example:

```
print "The time is @{[scalar localtime]} according to my clock";
```

will display just the same as:

```
print "The time is ", scalar localtime, " according to my clock";
```

You can also modify individual elements, using the syntax `${$reference}[$element]`:

```
#!/usr/bin/perl
# modelem.plx
use warnings;
use strict;

my @array = (68, 101, 114, 111, 117);
my $ref = \@array;
${$ref}[0] = 100;
print "Array is now : @array\n";
```

```
>perl modelem.plx
Array is now 100 101 114 111 117
>
```

And again, you can do the same with anonymous data:

```
my $ref = [68, 101, 114, 111, 117];
${$ref}[0] = 100;
print "Array is now : @{$ref}\n";
```

Hash References

For references to hashes, the rule is exactly the same. So, to access the hash that a reference points to, you use `%{$hash_r}`. If you want to get at a hash entry `$hash{green}`, you say `${hash_r}{green}`:

```
#!/usr/bin/perl
# hash.plx
use warnings;
use strict;

my %hash = (
   1 => "January",    2 => "February", 3 => "March",     4 => "April",
   5 => "May",        6 => "June",     7 => "July",      8 => "August",
   9 => "September", 10 => "October", 11 => "November", 12 => "December"
);

my $href = \%hash;
for (keys %{$href}) {
   print "Key: ", $_, "\t";
   print "Hash: ",$hash{$_}, "\t";
   print "Ref: ",${$href}{$_}, "\n";
}
```

As expected, we get the same data when using the hash as when using the reference:

```
>perl hash.plx
Key: 1    Hash: January     Ref: January
Key: 2    Hash: February    Ref: February
Key: 3    Hash: March       Ref: March
Key: 10   Hash: October     Ref: October
Key: 4    Hash: April       Ref: April
Key: 11   Hash: November    Ref: November
Key: 5    Hash: May         Ref: May
Key: 12   Hash: December    Ref: December
Key: 6    Hash: June        Ref: June
Key: 7    Hash: July        Ref: July
Key: 8    Hash: August      Ref: August
Key: 9    Hash: September   Ref: September
>
```

This should also help to remind you that Perl's hashes aren't ordered as you might expect!

Notation Shorthands

There are two more rules, but they're not essential for understanding and using references. They just make it easier for us to write programs manipulating references:

> **You don't have to write the curly brackets.**

You may find that it makes your code a little clearer if you omit the curly brackets around the reference. For example, we could rewrite our original dereferencing example `deref1.plx` like this:

```
#!/usr/bin/perl
# dref1alt.plx
use warnings;
use strict;

my @array   = (1, 2, 3, 4, 5);
my $array_r = \@array;

print "This is our dereferenced array: @$array_r\n";
for (@$array_r) {
   print "An element: $_\n";
}
print "The highest element is number $#$array_r\n";
print "This is what our reference looks like: $array_r\n";
```

Our hash example `hash.plx` would then look like this:

```
#!/usr/bin/perl
# hashalt.plx
use warnings;
use strict;
```

```
my %hash = (
   1 => "January",    2 => "February", 3 => "March",     4 => "April",
   5 => "May",        6 => "June",     7 => "July",      8 => "August",
   9 => "September", 10 => "October", 11 => "November", 12 => "December"
);

my $href = \%hash;
for (keys %$href) {
   print "Key: ", $_, " ";
   print "Hash: ",$hash{$_}, " ";
   print "Ref: ",$$href{$_}, " ";
   print "\n";
}
```

However, it may sometimes be clearer to leave the curly brackets in. Consider these three assignments:

```
$$hashref{KEY}   = "VALUE"; # 1
${$hashref}{KEY} = "VALUE"; # 2
${$hashref{KEY}} = "VALUE"; # 3
```

Case 1 is the same as case 2, whereas case 3 dereferences the scalar reference stored in `$hashref{KEY}`.

You can also run into problems when you have one reference stored inside another. If we have the following array reference:

```
$ref = [ 1, 2, [ 10, 20 ] ];
```

we can get at the internal array reference by saying `${$ref[2]}`. But say we want to get at the second element of that array – the one containing the value 20. Well, we could store the reference inside another scalar and then dereference it, like this:

```
$inside  = ${$ref}[2];
$element = ${$inside}[1];
```

Or we could get the element directly, by repeatedly substituting references for array names:

```
$element = ${${ref}[2]}[1];
```

This gets very ugly, very quickly, especially if you're dealing with hash references, where it becomes hard to tell if the curly braces surround a reference or a hash key.

So, to help us clear it up again, we introduce another rule:

> **Instead of `${$ref}`, we can say `$ref->`**

Let's demonstrate this, by taking one of our previous examples, `modelem.plx`, and incorporating this into the code. Here's the relevant piece of the original:

```
my @array = (68, 101, 114, 111, 117);
my $ref = \@array;
${$ref}[0] = 100;
print "Array is now : @array\n";
```

and here it is rewritten:

```
my @array = (68, 101, 114, 111, 117);
my $ref = \@array;
$ref->[0] = 100;
print "Array is now : @array\n";
```

Likewise for hashes, we can use this arrow notation to make things a bit clearer for ourselves. Recall `hash.plx` from a little while ago:

```
for (keys %{$href}) {
   print "Key: ", $_, " ";
   print "Hash: ",$hash{$_}, " ";
   print "Ref: ",${$href}{$_}, " ";
   print "\n";
}
```

Instead of that, we can write:

```
for (keys %{$href}) {
   print "Key: ", $_, " ";
   print "Hash: ",$hash{$_}, " ";
   print "Ref: ",$href->{$_}, " ";
   print "\n";
}
```

Now we can get at our array-in-an-array like this:

```
$ref = [ 1, 2, [ 10, 20 ] ];
$element = {$ref->[2]}->[1];
```

or more simply:

```
$element = $ref->[2]->[1];
```

However, we've got one more sub-rule that can simplify this even further:

> **Between sets of brackets, the arrow is optional.**

We can therefore rewrite the above as:

```
$element = $ref->[2][1];
```

Personally, I never omit the arrow in this way – it's far too easy to confuse `$ref->[0][1]` with `$ref[0][1]`, which perl will interpret as a dereference of the first element in the ordinary array `@ref`.

Reference Counting and Destruction

We've now seen all the ways you can create and use references. So when and how are references destroyed? Well, every piece of data in Perl has something called a **reference count** attached to it. This keeps track of the number of instances of the executing code accessing that exact chunk of data.

When we create a reference to some data, the data's reference count goes up by one. When we stop referring to it – we reassign the reference variable or 'break' it (as we saw above, when we tried to modify its value) – the reference count goes down. When nobody's using the data, and the reference count gets down to zero, the data is removed. Consider the following example:

```
#!/usr/bin/perl
# refcount.plx
use warnings;
use strict;

my $ref;
{
   my @array = (1, 2, 3);
   $ref = \@array;
   my $ref2 = \@array;
   $ref2 = "Hello!";
}
undef $ref;
```

Now, let's look at the references to the array `(1, 2, 3)` as we go through the program. To start with, the array is created, and the data `(1, 2, 3)` has one reference, which is in use by the array `@array`:

```
my $ref;
{
   my @array = (1, 2, 3);
```

Now we've created another reference to it, and the reference count increases to two:

```
   $ref = \@array;
```

Once again we create a reference, and the count goes up to three:

```
   my $ref2 = \@array;
```

However, we've now changed that reference to be an ordinary string – it's not pointing at our array any more, so the reference count on `(1, 2, 3)` goes back down to two. Note that changing `$ref2` doesn't affect the original array. That only happens when we dereference:

```
   $ref2 = "Hello!";
```

Now a block ends, and all the lexical variables – the `my` variables – inside that block go out of scope. That means that `$ref2` and `@array` are destroyed. The reference count of the data `(1, 2, 3)` goes down again because `@array` is no longer using it. However, `$ref` still has a reference to it, so the reference count is still one, and the data itself is not removed from the system. `$ref` still refers to `(1, 2, 3)` and can access and change this data as before, that is, of course, until we get rid of it:

```
}
```

Now the final reference to the data `(1, 2, 3)` is removed, that array is finally freed:

```
undef $ref;
```

Counting Anonymous References

Anonymous data works in the same way. However, it doesn't get its initial reference count from being attached to a variable, but rather from when its first explicit reference is created:

```
my $ref = [1, 2, 3];
```

This data therefore has a reference count of one, rather than:

```
my @array = (1, 2, 3);
my $ref = \@array;
```

which has a count of two.

Using References for Complex Data Structures

Now that we've looked at what references are, you might be asking: why on earth would we want to use them? Well, as we mentioned in the introduction, we often want to create data structures that are more complex than simple arrays or hashes. We may need to store arrays inside arrays, or hashes inside hashes, and References help us do this.

So let's now take a look at a few of the complex data structures we can create with references.It won't be exhaustive by any means, but it should serve to give you ideas as to how complex data structures look and work in Perl, and it should also help you to understand the most common data structures.

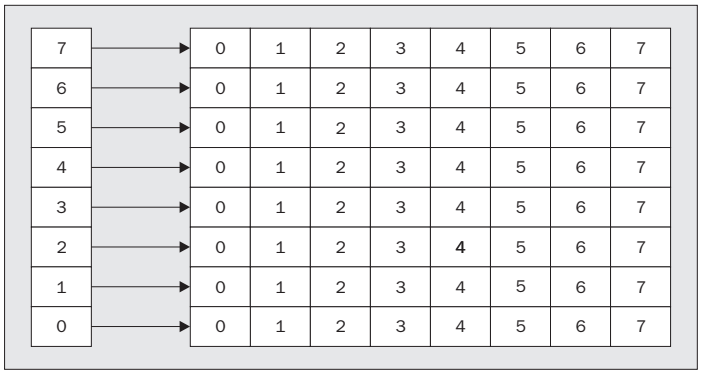
Matrices

What is a matrix? No, not the thing that Keanu Reeves wants out of. A matrix is simply an array of arrays. You can refer to any single element with a combination of two subscripts, which you might want to think of as a row number and a column number. It's harking back to the chessboard example we mentioned in the introduction to this chapter.

If you use the arrow syntax, matrices are very easy to use. You get at an element by saying:

```
$array[$row]->[$column]
```

`$array[$row]` is an array reference, and we're derefencing the `$column`'th element in it. With a chessboard example, it would look like this:



So, `$array[0]->[0]` would be the bottom left hand corner of our chessboard, and `$array[7]->[7]` would be the top right.

Autovivification

Now, there's one last thing we need to know about references before we go on. If we assign values to a reference, perl will automatically create all appropriate references necessary to make it work. So, if we say this:

```
my $ref;
$ref->{UK}->{England}->{Oxford}->[1999]->{Population} = 500000;
```

perl will automatically know that we need `$ref` to be a hash reference. So, it'll make us a nice new anonymous hash:

```
$ref = {};
```

Then we need `$ref->{UK}` to be a hash reference, because we're looking for the hash key `England`; that hash entry needs to be an array reference, and so on. Perl effectively does this:

```
$ref = {};
$ref->{UK} = {};
$ref->{UK}->{England} = {};
$ref->{UK}->{England}->{Oxford} = [];
$ref->{UK}->{England}->{Oxford}->[1999] = {};
$ref->{UK}->{England}->{Oxford}->[1999]->{Population} = 500000;
```

What this means is that we don't have to worry about creating all the entries ourselves. So we can just write:

```
my @chessboard;
$chessboard[0]->[0] = "WR";
```

This is called **autovivification** – things springing into existence. We can use it to greatly simplify the way we use references:

Try It Out : A Chess Game

Now that we can represent our chessboard, let's set up a chess game. This will consist of two stages: setting up the board, and making moves. The computer will have no idea of the rules, but will simply function as a board, allowing us to move pieces around. Here's our program:

```
#!/usr/bin/perl
# chess.plx
use warnings;
use strict;

my @chessboard;
my @back = qw(R N B Q K N B R);
for (0..7) {
   $chessboard[0]->[$_] = "W" . $back[$_]; # White Back Row
   $chessboard[1]->[$_] = "WP";            # White Pawns
   $chessboard[6]->[$_] = "BP";            # Black Pawns
   $chessboard[7]->[$_] = "B" . $back[$_]; # Black Back Row
}

while (1) {
   # Print board
   for my $i (reverse (0..7)) { # Row
       for my $j (0..7) {       # Column
          if (defined $chessboard[$i]->[$j]) {
            print $chessboard[$i]->[$j];
          } elsif ( ($i %2) == ($j %2) ) {
             print "..";
          } else {
             print "  ";
          }
         print " ";  # End of cell
      }
      print "\n";     # End of row
   }

   print "\nStarting square [x,y]: ";
   my $move = <>;
   last unless ($move =~ /^\s*([1-8]),([1-8])/);
   my $startx = $1-1; my $starty = $2-1;

   unless (defined $chessboard[$starty]->[$startx]) {
      print "There's nothing on that square!\n";
      next;
   }
   print "\nEnding square [x,y]: ";
   $move = <>;
   last unless ($move =~ /([1-8]),([1-8])/);
   my $endx = $1-1; my $endy = $2-1;

   # Put starting square on ending square.
   $chessboard[$endy]->[$endx] = $chessboard[$starty]->[$startx];
   # Remove from old square
   undef $chessboard[$starty]->[$startx];
}
```

Now let's see the first part of a game in progress:

```
> perl chess.plx
 BR  BN  BB  BQ  BK  BN  BB  BR
 BP  BP  BP  BP  BP  BP  BP  BP
     ..      ..      ..      ..
 ..      ..      ..      ..
     ..      ..      ..      ..
 ..      ..      ..      ..
 WP  WP  WP  WP  WP  WP  WP  WP
 WR  WN  WB  WQ  WK  WN  WB  WR

Starting square [x,y]: 4,2

Ending square [x,y]: 4,4
 BR  BN  BB  BQ  BK  BN  BB  BR
 BP  BP  BP  BP  BP  BP  BP  BP
     ..      ..      ..      ..
 ..      ..      ..      ..
     ..      WP      ..      ..
 ..      ..      ..      ..
 WP  WP  WP  ..  WP  WP  WP  WP
 WR  WN  WB  WQ  WK  WN  WB  WR

Starting square [x,y]: 4,7

Ending square [x,y]: 4,5
 BR  BN  BB  BQ  BK  BN  BB  BR
 BP  BP  BP  .   BP  BP  BP  BP
     ..      ..      ..      ..
 ..      ..  BP  ..      ..
     ..      WP      ..      ..
 ..      ..      ..      ..
 WP  WP  WP  ..  WP  WP  WP  WP
 WR  WN  WB  WQ  WK  WN  WB  WR
```

How It Works

Our first task is to set up the chessboard, with the pieces in their initial positions. Remember that we're assigning `$chessboard[$row]->[$column] = $thing`. First, we set up an array of pieces on the 'back row'. We'll use this to make it easier to put each piece in its appropriate column:

```
my @back = qw(R N B Q K N B R);
```

Now we'll go over each column:

```
for (0..7) {
```

In row zero, the back row for white, we want to place the appropriate piece from the array in each square:

```
    $chessboard[0]->[$_] = "W" . $back[$_]; # White Back Row
```

In row one of each column, we want a white pawn, WP:

```
    $chessboard[1]->[$_] = "WP";            # White Pawns
```

Now we do the same again for black's pieces on rows 6 and 7:

```
    $chessboard[6]->[$_] = "BP";            # Black Pawns
    $chessboard[7]->[$_] = "B" . $back[$_]; # Black Back Row
}
```

What about the rest of the squares on board? Well, they don't exist right now, but will spring into existence when we try and read from them.

Next we go into our main loop, printing out the board and moving the pieces. To print the board, we obviously want to look at each piece. So we loop through each row and each column:

```
for my $i (reverse (0..7)) { # Row
    for my $j (0..7) {       # Column
```

If the element is defined, it's because we've put a piece there, so we print it out:

```
        if (defined $chessboard[$i]->[$j]) {
          print $chessboard[$i]->[$j];
```

Note that at this point, we're accessing all 64 squares. this means any square that didn't exist before will do from now on. This next piece of prettiness prints out the "checkered" effect. On a checkerboard, dark squares come on odd rows in odd columns and even rows in even columns. `$x % 2` tests whether `$x` divides equally by two – whether it is odd or even. If the 'oddness' (or 'evenness') of the row and column is the same, we print a dark square:

```
        } elsif ( ($i %2) == ($j %2) ) {
           print "..";
```

Otherwise, we print a blank square consisting of two spaces:

```
        } else {
           print "  ";
        }
```

To separate the cells, we use a single space:

```
        print " ";  # End of cell
    }
```

And at the end of each row, we print a new line:

```
    print "\n";     # End of row
}
```

Now we ask for a square to move from:

```
print "\nStarting square [x,y]: ";
my $move = <>;
```

We're looking for two digits with a comma in the middle:

```
last unless ($move =~ /([1-8]),([1-8])/);
```

Now we convert human-style coordinates (1 to 8) into computer-style coordinates (0 to 7):

```
my $startx = $1-1; my $starty = $2-1;
```

Next, check if there's actually a piece there. Note that a y coordinate is a row, so it goes first – look back at the diagram if you're not sure how this works:

```
unless (defined $chessboard[$starty]->[$startx]) {
   print "There's nothing on that square!\n";
   next;
}
```

We do the same for the ending square, and then move the piece. We copy the piece to the new square:

```
$chessboard[$endy]->[$endx] = $chessboard[$starty]->[$startx];
```

And then we delete the old square:

```
undef $chessboard[$starty]->[$startx];
```

We've now used a matrix, a two-dimensional array. The nice thing about perl's auto vivification is that we didn't need to say explicitly that we were dealing with references. Perl takes care of all that behind the scenes, and we just assigned the relevant values to the right places. However, if we were to look at the contents of the `@chessboard` array, we'd see eight array references.

Trees

We're now going to build on the principle of matrices, by introducing **tree**-like data structures, in which we use hashes as well as arrays. The classic example of one of these structures is an address book. Suppose we want to keep someone's address and phone number in a hash. We could say this:

```
%paddy = (
   address => "23, Blue Jay Way",
   phone   => "404-6599"
);
```

That's all very well, and it makes sense. The only problem is, you have to create a separate hash for each person in your address book and put each one in a separate variable. This isn't easy at all at run time, and is very messy to write. So instead, you use references.

What we do is create a main 'address book' hash, referenced as `$addressbook`, with everyone else's hashes as values off that:

```
$addressbook{"Paddy Malone"} = {
   address => "23, Blue Jay Way",
   phone   => "404-6599"
};
```

> **Note that if you've included the `use strict;` pragma, you'll have to declare this hash explicitly as `my %addressbook;` before using it.**

It's now very easy to take new entries from the user and add them to our address book:

```
print "Give me a name:"; chomp $name   =<>;
print "Address:";        chomp $address=<>;
print "Phone number:";   chomp $phone  =<>;
$addressbook{$name} = {
   address => $address,
   phone   => $phone
};
```

To print out a single person, we'd use this:

```
if (exists $addressbook{$who}) {
   print "$who\n";
   print "Address:  ", $addressbook{$who}->{address}, "\n";
   print "Phone no: ", $addressbook{$who}->{phone},   "\n";
}
```

To print every address, we'd use this:

```
for $who (keys %addressbook) {
   print "$who\n";
   print "Address:  ", $addressbook{$who}->{address}, "\n";
   print "Phone no: ", $addressbook{$who}->{phone},   "\n";
}
```

Deleting an address is very simple:

```
delete $addressbook{$who};
```

How about adding another level to our tree. Can we have an array of 'friends' for each person? No problem. We just use an anonymous array:

```
$addressbook{"Paddy Malone"} = {
   address => "23, Blue Jay Way",
   phone   => "404-6599",
   friends => [ "Baba O'Reilly", "Mick Flaherty" ]
};
```

We can get at each person's friends by saying `$addressbook{$who}->{friends}`. That will give us an anonymous array. We can then dereference that to a real array and print it out:

```
for $who (keys %addressbook) {
   print "$who\n";
   print "Address:  ", $addressbook{$who}->{address}, "\n";
   print "Phone no: ", $addressbook{$who}->{phone},   "\n";
   my @friends = @{$addressbook{$who}->{friends}};
   print "Friends:\n";
   for (@friends) {
      print "\t$_\n";
   }
}
```

This would now give us something like:

```
Paddy Malone
Address:  23, Blue Jay Way
Phone no: 404-6599
Friends:
      Baba O'Reilly
      Mick Flaherty
```

What we now have is one hash (address book), containing another hash (peoples' details), in turn containing an array (each person's friends).

We can quite easily **traverse** the tree structure, that is, move from person to person by following links. We do this by visiting a link, then adding all of that person's friends onto a 'to do' array. We must be very careful here not to get stuck in a loop. If one person links to another, and the other links back again, we need to avoid bouncing about between them indefinitely. One simple way to keep track of the links we've already processed is to use a hash. Here's how we can do it:

```
$, = "\t"            # Set output field separator for tabulated display
my @todo = ("Paddy Malone"); # Start point
my %seen;
while (@todo) {
   my $who = shift @todo; # Get person from the end
   $seen{$who}++;         # Mark them as seen.
   my @friends = @{$addressbook{$who}->{friends}};
   print "$who has friends: ", @friends, "\n";
   for (@friends) {
      # Visit unless they're already visited
      push @todo, $_ unless exists $seen{$_};
   }
}
```

The reference `$seen` is used to build up a hash table of everyone whose name has been held in the variable `$who`. The `for` loop at the bottom only adds names to the `@todo` list if they're not defined in that hash, That is, if they've not been displayed already. Given a fairly closed community, we could see something like this:

```
Paddy Malone has friends Baba O'Reilly Mick Flaherty
Baba O'Reilly has friends Bob McDowell Mick Flaherty Andy Donahue
Mick Flaherty has friends Paddy Malone Timothy O'Leary
Bob McDowell has friends Andy Donahue Baba O'Reilly
Andy Donahue has friends Jimmy Callahan Mick Flaherty
Timothy O'Leary has friends Bob McDowell Mick Flaherty Paddy Malone
Jimmy Callahan has friends Andy Donahue Baba O'Reilly Mick Flaherty
```

Linked Lists

The last thing we're going to look at is creating **linked lists**. These actually cover quite a broad range of data structures, but all have one common feature:

> **One part of each record in the list refers to at least one other record in the list.**

Just as any good page on the web will link to at least one other page, each record in a linked list will include a reference to another record in the list, and possibly several. That's all well and good, but what improvement does this give us on the structures we've seen already? We know how to use a value held in one record to reference another – rather handy, but not exactly earth-shattering.

The fact is, while this is how linked lists hang together, it's not quite the full story. The examples we've seen so far have been passing references to and from records in a single root data structure: the addressbook hash reference. We take the name of a friend and use that as a key in the hash to access that friend's details.

Now, what if I have a bunch of friends at work, where there's already a data structure in place containing just this sort of information. Now, I want to include colleagues in my list of friends, but it's not practical to copy all the data from one to the other. What's more, while the work system uses a similar structure to the addressbook one, `$work` (the root reference – equivalent to `$addressbook`) uses ID numbers as **indices in an array**. For example, my friend Dan is registered as employee 4109, so his details are referenced by `$work[4108]` – yes, array indices start at 0. Anyway, it seems I can't have "Dan Maharry" as one of my friends.

Maybe I could just put '4109' in as his name. What the heck, I'll know who it is. No, of course we'd still be trying to access the addressbook hash reference, and "4109" isn't in there.

What if we get the program to check *both* root references for a suitable match? That works fine, until I'm sending out Christmas mail (automatically, of course. It's what perl does best!), and he gets one starting:

```
Dear 4109,
Let me tell you all about this new book I've written….
```

Hmm. Not really ideal. What we really need is to have our 'friends' key reference a hash table (instead of a list), with the key "Dan Maharry" assigned the value of the appropriate reference. So, instead of:

```
friends => [ "Baba O'Reilly", "Mick Flaherty" ]
```

we put:

```
friends => {   "Baba O'Reilly" => $addressbook("Baba O'Reilly"),
               "Mick Flaherty" => $addressbook("Mick Flaherty"),
               "Dan Maharry"   => $work[4109]
           }
```

The power and versatility (and some would say beauty) of a linked list derives from a very simple fact:

> **The internal structure of any record in a linked list can be independent of all others.**

In the simplest case, all our references were *from* addressbook entries *to* addressbook entries. This belies the fact that each of them could actually refer to **any** data structure at all. As we saw though, the flexibility of Perl references allows us to link up all sorts of different structures.

Summary

We've looked at references, a way to put one type of data structure inside another. References work because they allow us to refer to another piece of data. They tell us where Perl stores it and give us a way to get at it. Because references are always scalars, you can think of them as flat-pack storage for arrays and hashes.

We can create a reference explicitly by putting a backslash in front of a variable's name: `\%hash` or `\@array`, for example. Alternatively, we can create an anonymous reference by using { } instead of `()` for a hash and `[]` instead of `()` for an array. Finally, we can create a reference by creating a need for one. If a reference needs to exist for what we're doing, Perl will spring one into existence by autovivification.

We can use a reference by placing it in curly brackets where a variable name should go. `@{$array_r}` can replace `@array` everywhere and we don't even need the brackets if it's clear what we mean. We can then access elements of array or hash references using the arrow notation: `$array_ref->[$element]` for an array and `$hash_ref->{$key}` for a hash.

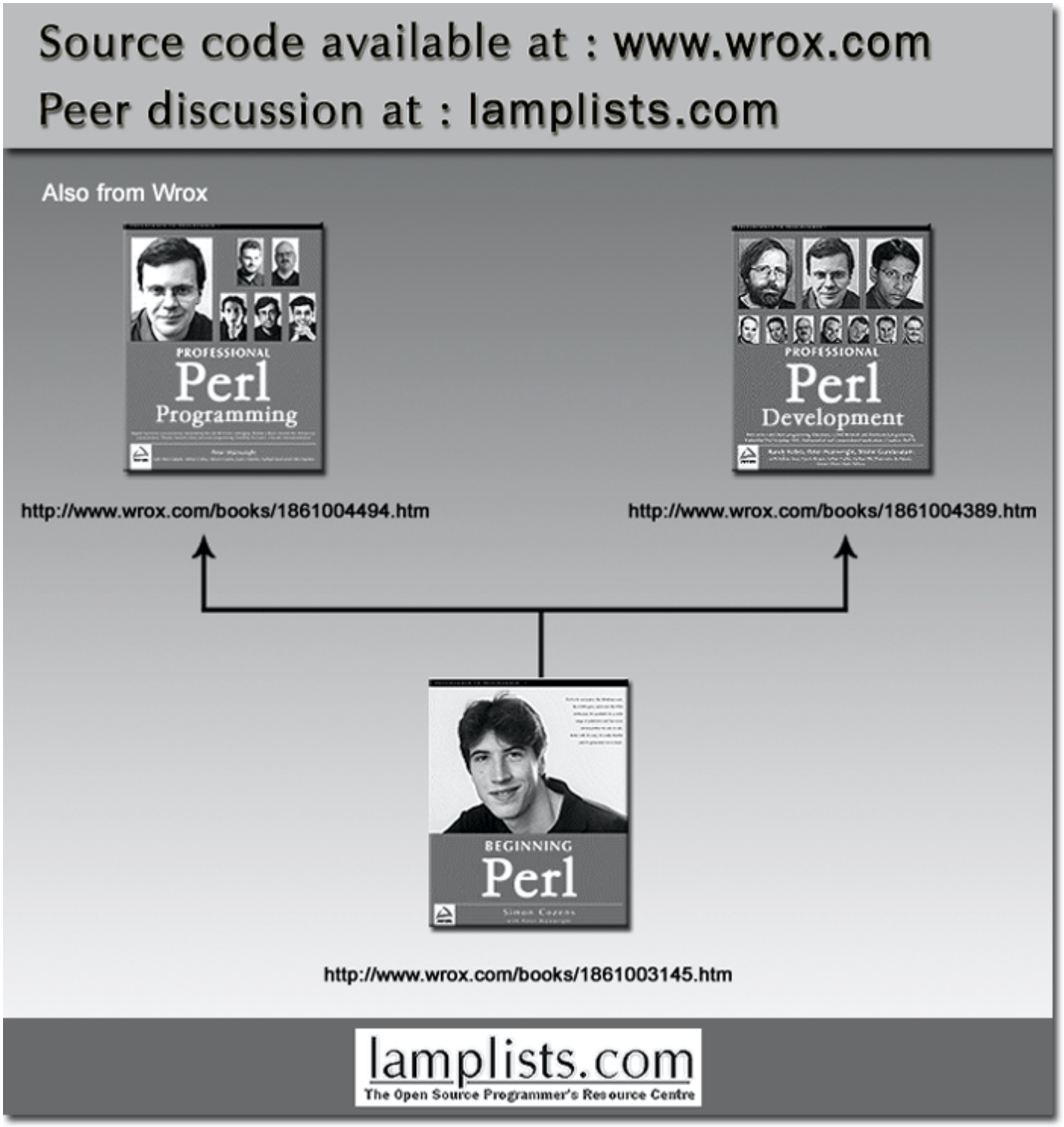
We've also seen a few complex data structures: matrices, which are arrays of arrays; trees, which may contain hashes or arrays; and linked lists, which contain references to other parts of the data structure, or even other data structures. For more information on these kinds of data structure, consult the Perl 'Data Structures Cookbook' documentation (`perldsc`) or the Perl 'List of Lists' documentation. (`perllol`)

If you're really interested in data structures from a computer science point of view, *Mastering Algorithms in Perl* by Orwant et al. (*O'Reilly – ISBN 1-56592-398-7*) has some chapters on these kinds of structure, primarily, trees and tree traversal. The ultimate guide to data structures is still '*The Art Of Computer Programming, Volume 1*', by Donald Knuth (*Addison Wesley – ISBN 0201896834*) – affectionately known as 'The Bible'.

Exercises

1. Construct an array of arrays to form a multiplication table covering from one times one to six times six but as words. Then ask the user to query it and return the result in words only.

2. Take the chess program and revise it so it checks for the validity of the knight's moves. Remember that the knight cannot move off the board or take one of its own pieces. The knight moves in an L-shape – two squares horizontally or vertically and then one square at ninety degrees to that.



8

Subroutines

When programming, there'll naturally be processes you want to do again and again: adding up the values in an array, stripping extraneous blank spaces from a string, getting information into a hash in a particular format, and so on. It would be tedious to write out the code for each of these little processes every time we need to use one, and it would be horrific to maintain too: if there are bugs in the way we've specified it, we'll have to go through and find each one of them and fix it. It would be better if we could define a particular process just once, and then be able to call on that just like we've been calling on Perl's built-in operators.

This is exactly what **subroutines** allow us to do. Subroutines (or just **subs**) give us the ability to give a name to a section of code. Then when we need to use that code in our program, we just call it by name.

Subroutines help our programming for two main reasons. First, they let us reuse code, as we've described above. This makes it easier to find and fix bugs and makes it faster for us to write programs. The second reason is that they allow us to chunk our code into organizational sections. Each subroutine can, for example, be responsible for a particular task.

So, when is it appropriate to use subroutines in Perl? I would say there would be two cases when a piece of code should be put into a subroutine: first, when you know it will be used to perform a calculation or action that's going to happen more than once. For instance, putting a string into a specific format, printing the header or footer of a report, turning an incoming data record into a hash, and so on.

One thing we'll see later on is that we can use subroutines in a similar way to the way we've been using Perl's built-in operators. We can give them arguments and get scalars and lists returned to us.

Second, if there are logical units of your program that you want to break up to make your program easier to understand. I can imagine few things worse than debugging several thousand lines of Perl that are not broken up in any way (well, maybe one or two things). As an extreme example, sometimes – and only sometimes – I like to have a 'main program', which consists entirely of calls to subroutines, like this:

```
#!/usr/bin/perl
use warnings;
use strict;

setup();
get_input();
process_input();
output();
```

This immediately shows me the structure of my program. Each of those four subroutines would, of course, have to be defined, and they'd probably call on other subroutines themselves. This allows us to partition up our programs, to change our single, monolithic piece of code into manageable chunks for ease of understanding, ease of debugging, and ease of maintaining the program.

The 'Difference' Between Functions and Subroutines

Instead of the term 'subroutine', you're sure to come across the word 'function' many times as you deal with Perl and Perl resources. So let's have a look at the difference between 'function', 'subroutine', and 'operator'. The problem is that other programming languages use the terms ever so slightly differently.

Usually

In most programming languages, and in computer science in general, the following definitions apply:

- A **function** is something that takes a number of arguments (possibly zero), does something with them, and returns a value. A function can either be built into the programming language or it can be supplied by the user.
- An **operator** is a function that is usually represented by a symbol rather than a name and is almost always built into the programming language.
- A **subroutine** is some code provided by the user that performs an action and doesn't return a value. Unfortunately, languages like C have functions that can return nothing. These 'void functions' could be called subroutines – but they're not. That's life.

In Perl

Because some people who know other languages use the usual terms, Perl's definitions are a little confusing:

- If someone mentions a **function** in Perl, they almost certainly mean something built into Perl. However, they might be coming from C and mean a subroutine. The main reference documentation for Perl built-ins is called `perlfunc`. You can also find the complete list in Appendix C.
- An **operator** in Perl can have a name instead of a symbol, so it can look very much like a function. Hence, some people tend to use the terms interchangeably. Those built-ins that have symbols instead of names are documented in `perlop`, which also refers to 'named operators'. `perl` itself speaks about the 'print operator', so we've used that terminology in this book. However, you're equally likely to hear Perl people talk about 'the print function'.
- **Subroutines** in Perl are akin to C's functions – they are sections of code that can take arguments, perform some operations with them, and may return a meaningful value, but don't have to. However, they're always user-defined rather than built-ins:

> **Simply put: Subroutines are chunks of code you give Perl; Functions and Operators are things that Perl provides.**

Understanding Subroutines

Now we know what subroutines are, it's time to look at how to define them and how to use them. First, we'll learnhow to create subroutines.

Defining a Subroutine

So, we can give Perl some code, and we can give it a name, and that's our subroutine. Here's how we do it:

```
sub marine {
   ...
}
```

There are three sections to this declaration:

- The keyword `sub`. This is case-sensitive and needs to be in lower case.
- The name we're going to give it. The rules for naming a subroutine are exactly those for naming variables; names must begin with an alphabetic character or an underscore, to be followed by one or more alphanumerics or underscores. Upper case letters are allowed, but we tend to reserve all-uppercase names for special subroutines. And again, as for variables, you can have a scalar `$fred`, an array `@fred`, a hash `%fred`, a filehandle `fred`, and a subroutine `fred`, and they'll all be distinct.
- A block of code delimited by curly brackets, just as we saw when we were using `while` and `if`. Notice that we don't need a semicolon after the closing brace.

After we've done that, we can use our subroutine.

Before we go any further, it's worth taking a quick time out to ponder how we name our subroutines. You can convey a lot about a subroutine's purpose with its name, much like that of a variable. Here are some guidelines – not hard-and-fast rules – about how you should name subroutines.

- If they're primarily about performing an activity, name them with a verb, for example, `summarize` or `download`.
- If they're primarily about returning information, name them after what they return, for example, `greeting` or `header`.
- If they're about testing whether a statment is true or not, give them a name that makes sense in an `if` statement; starting with `is_`… or `can_`… helps, or if that isn't appropriate, name them with an adjective: for example, `is_available`, `valid`, or `readable`.
- Finally, if you're converting between one thing and another, try and convey both things. Traditionally this is done with an `2` or `_to_` in the middle: `text2html`, `metres_to_feet`. That way you can tell easily what's being expected and what's being produced.

Try It Out : Version Information

It's traditional for programs to tell you their version and name either when they start up or when you ask them with a special option. It's also convenient to put the code that prints this information into a subroutine to get it out of the way. Let's take our very first program and update it for this traditional practice.

Here's what we started with, version 1:

```
#!/usr/bin/perl
use warnings;
print "Hello, world.\n";
```

And here it is with warnings and strict modes turned on and version information:

```
#!/usr/bin/perl
# hello2.plx
use warnings;
use strict;

sub version {
   print "Beginning Perl's \"Hello, world.\" version 2.0\n";
}

my $option = shift;
version if $option eq "-v" or $option eq "--version";
print "Hello, world.\n";
```

Now, we're starting to look like a real utility:

```
>perl hello2.plx -v
Beginning Perl's "Hello, world." version 2.0
Hello, world.
```

How It Works

As before, we have the `sub` keyword, a name, `version`, and then the block of code. We've defined the `version` subroutine as follows:

```
sub version {
   print "Beginning Perl's \"Hello, world.\" version 2.0\n";
}
```

It's a simple block of code that calls the `print` statement. It didn't have to – it could have done anything. Any code that's valid in the main program is valid inside a subroutine, including:

- Calling other subroutines
- Calling the current subroutine again – see the section 'Recursion' at the end of the chapter on this very subject.

We call this block the **body** of the subroutine, just like we had the body of a loop; similarly, it stretches from the open curly bracket after the subroutine name to the matching closing bracket.

Now we've defined it, we can use it. We just give the name, and Perl runs that block of code, albeit with the proviso that we've added the right flag on the command line:

```
version if $option eq "-v" or $option eq "--version";
```

When it's finished doing `version`, it comes back and carries on with the next statement:

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

No doubt version th3ree will address the warnings that Perl gives if you call this program without appending -v or --version to its name.

Order of Declaration

If we just call our subroutines by name, as we did above, we're forced to declare them before we use them. This may not sound much of a limitation, but there are times when we'll want to declare our subroutines after the main part of the program. In fact, that's the usual way to structure a program. This is because when you open up the file in your editor, you can see what's going on right there at the top of the file, without having to scroll through a bunch of definitions first. Take the extreme example at the beginning of this chapter:

```
#!/usr/bin/perl
use warnings;
use strict;

setup();
get_input();
process_input();
output();
```

That would then be followed, presumably, by something like this:

```
sub setup {
   print "This is some program, version 0.1\n";
   print "Opening files...\n";
   open_files();
   print "Opening network connections...\n";
   open_network();
   print "Ready!\n";
}

sub open_files {
   ...
}
```

That's far easier to understand than trawling through a pile of subroutines before getting to the four lines that constitute our main program. It also encourages the 'top-down' school of programming.

> *Traditional programming methodology, which I've been using here, states that we should start at the highest level of our program and break it down into smaller and smaller problems – starting at the top and working down. There's also a bottom-up school of thought that dictates you should write your basic operations first, then glue them together. There's even been the suggestion of a 'middle-out' style that starts at a middle layer and adds smaller operations and higher-level structure at the same time. I encourage you to start with top-down programming until something else becomes natural.*

However, in order to get this to work, we need to provide hints to Perl as to what we're doing. That's why the calls to subroutines above have a pair of brackets around them: `setup()`, `open_files()`, and so on. This helps to tell Perl that it should be looking for a subroutine somewhere instead of referring to a filehandle or anything else it could have been. What happens if we don't do this?

```
#!/usr/bin/perl
# subdecl.plx
use warnings;
use strict;

setup;
sub setup {
   print "This is some program, version 0.1\n";
}
```

```
>perl subdec1.plx
Bareword "setup" not allowed while "strict subs" in use at subdecl.plx line 6.
Execution of subdec1.plx aborted due to compilation errors.
>
```

Perl didn't know what we meant at the time, so it complained. To tell it we're talking about a subroutine, we use brackets, just like when we want the parameters to an operator like `print` to be unambiguous.

There's another way we can tell Perl that we're going to refer to a subroutine and that's to provide a **forward definition** – also known as **pre-declaring** the subroutine. This means 'we're not going to define this right now, but look out for it later.'

We do this by just saying `sub NAME;`. Note that this does require a semicolon at the end. Here's another way of writing the above:

```
#!/usr/bin/perl
use warnings;
use strict;
sub setup; sub get_input; sub process_input; sub output;
sub open_files; sub open_network;
...
```

From now on, we can happily use the subroutines without the brackets:

```
setup;
get_input;
process_input;
output;

sub setup {
   print "This is some program, version 0.1\n";
   print "Opening files...\n";
   open_files;
   print "Opening network connections...\n";
   open_network;
   print "Ready!\n";
}

sub open_files {
   ...
}
```

Alternatively, you can ask Perl to provide the forwards for you. If we say `use subs (…)`, we can provide a list of subroutine names to be pre-declared:

```
#!/usr/bin/perl
use warnings;
use strict;
use subs qw(setup get_input process_input output pen_files open_network);
...
```

Personally, however, I tend to leave in the brackets to remind me I'm dealing with subroutines. You may also see yet another way of calling subroutines:

```
&setup;
&get_input;
&process_input;
&output;
```

This was popular in the days of Perl 4, and we'll see later why the ampersand is important. For the time being, think of the ampersand as being the 'type symbol' for subroutines.

Subroutines for Calculation

As we mentioned at the beginning of the chapter, as well as being set pieces of code to be executed whenever we need them, we can also use subroutines just like Perl's built-in functions and operators. We can pass parameters to the subroutine and expect an answer back.

Parameters and Arguments

Just like with Perl's built-ins, we pass parameters by placing them between the brackets:

```
my_sub(10,15);
```

What happens to them there? Well, they end up in one of Perl's special variables, the array `@_` and from there we can get at them:

Try It Out : Totalling a List

We'll write a subroutine that takes a list of values, adds them up, and prints the total:

```
#!/usr/bin/perl
# total1.plx
use warnings;
use strict;

total(111, 107, 105, 114, 69);
total(1...100);

sub total {
   my $total = 0;
   $total += $_ for @_;
   print "The total is $total\n";
}
```

And to see it in action:

```
> perl total1.plx
The total is 506
The total is 5050
>
```

How It Works

We can pass any list to a subroutine, just like we can to `print`. When we do so, the list ends up in `@_` where it's up to us to do something with it. Here, we go through each element of it and add them up:

```
$total += $_ for @_;
```

This is a little cryptic, but it's how you're likely to see it done in real Perl code. You could write this a little less tersely as follows:

```
my @args = @_;
foreach my $element (@args) {
   $total = $total+$element;
}
```

In the first example, `@_` would contain `(111, 107, 105, 114, 69)`, and we'd add each value to `$total` in turn.

Return Values

However, sometimes we don't want to perform an action like printing out the total, but instead we want to return a result. We may also want to return a result to indicate whether what we were doing succeeded. This will allow us to say things like:

```
$sum_of_100 = total(1...100);
```

There are two ways to do this: implicitly or explicitly. The implicit way is nice and easy. We just make the value we want to return the last thing in our subroutine:

```
#!/usr/bin/perl
# total2.plx
use warnings;
use strict;

my $total      = total(111, 107, 105, 114, 69);
my $sum_of_100 = total(1...100);

sub total {
   my $total = 0;
   $total += $_ for @_;
   $total;
}
```

It doesn't need to be a variable: we could use any expression there. We can also return a list instead of a single scalar.

Try It Out : Splitting Time

Let's split a time in seconds up to hours, minutes, and seconds. We give a subroutine a time in seconds, and it returns a three-element list with the hours, minutes, and remaining seconds:

```
#!/usr/bin/perl
# seconds1.plx
use warnings;
use strict;

my ($hours, $minutes, $seconds) = secs2hms(3723);
print "3723 seconds is $hours hours, $minutes minutes and $seconds seconds";
print "\n";

sub secs2hms {
   my ($h,$m);
   my $seconds = shift;
   $h = int($seconds/(60*60)); $seconds %= 60*60;
   $m = int($seconds/60);      $seconds %= 60;
   ($h,$m,$seconds);
}
```

This tells us that:

```
>perl seconds1.plx
3723 seconds is 1 hours, 2 minutes and 3 seconds
>
```

How It Works

Just like a built-in function, when we're expecting a subroutine to return a list, we can use an array or list of variables to collect the return values:

```
my ($hours, $minutes, $seconds) = secs2hms(3723);
```

When `secs2hms` returns, this'll be equivalent to:

```
my ($hours, $minutes, $seconds) = (1,2,3);
```

Now let's look at how the subroutine works. We start in the usual way: `sub`, the name, and a block:

```
sub secs2hms {
```

We have two variables to represent hours and minutes, and we read the parameters in from `@_`. If you don't tell `shift` which array to take data from, it'll read from `@_` if you're in a subroutine or `@ARGV` if you're not:

```
   my ($h,$m);
   my $seconds = shift;
```

Then the actual conversion: There are 3600 (60*60) seconds in an hour, and so the number of hours is the number of seconds divided by 3600. However, that'll give us a floating-point number – if we divided 3660 by 3600, we'd get 1.0341666… we'd rather have 'one and a bit', so we use `int()` to get the integer value, the '1' part of the division, and use the modulus operator to get the remainder. After dealing with the first 3600 seconds, we want to carry on looking at the next 123:

```
$h = int($seconds/(60*60)); $seconds %= 60*60;
```

The second statement on this line sets `$seconds` to `$seconds % (60*60)`. If it was 3723 before, it'll be 123 now.

The same goes for minutes: we divide to get 'two and a bit', and the remainder tells us that there are three seconds outstanding. Hence, our values are 1 hour, 2 minutes, and 3 seconds:

```
$m = int($seconds/60);      $seconds %= 60;
```

We return this just by leaving a list of the values as the last thing in the subroutine:

```
($h,$m,$seconds);
```

The return Statement

The explicit method of returning something from a subroutine is to say `return(...)`. The first `return` statement we come across will immediately return that list to the caller. So, for instance:

```
sub secs2hms {
   my ($h,$m);
   my $seconds = shift;
   $h = int($seconds/(60*60)); $seconds %= 60*60;
   $m = int($seconds/60);      $seconds %= 60;
   return ($h,$m,$seconds);
   print "This statement is never reached.";
}
```

This also means we can have more than one `return` statement, and it's often useful to do so.

Caching

One particularly effective use of this is called **caching**, and it's a technique we can use to make subroutines that do calculations work faster. To use caching, we store each answer we generate from a set of parameters into a cache, usually a hash. If we see those parameters again, we can fetch the answer from the cache rather than work it all out from scratch. For example, here's a subroutine that gets the first line in a file:

```
sub first_line {
   my $filename = shift;
   open FILE, $filename or return "";
   my $line = <FILE>;
   return $line;
}
```

And here's that subroutine with caching:

```
my %cache;
sub first_line {
   my $filename = shift;
   return $cache{$filename} if exists $cache{$filename}
   open FILE, $filename or return "";
   my $line = <FILE>;
   $cache{filename} = $line;
   return $line;
}
```

Although it's possible that the first lines of those files change while we're running the program, it's not likely. So, we check to see if we've seen a file before; if we have, we give the answer we got last time and return. If we haven't seen it before, we open the file, check it out, and then store the answer in the cache for next time.

If you've got subroutines where the answer is likely to be the same every time you call with a given parameter, and where you're doing significantly more work than a simple lookup, consider using a cache like this.

Context

Some of Perl's built-ins do different things in different contexts: `localtime`, for instance, returns a string in scalar context and a breakdown of the time in list context. As `perlfunc` puts it, '*There is no rule that relates the behavior of an expression in list context to its behavior in scalar context, or vice versa. It might do two totally different things.*'

We can make our subs sensitive to context as well. Perl provides two functions to allow us to examine how we were called. The more complex one is `caller`, and the one we'll look at is `wantarray`. Strictly speaking, it tells us whether our caller wants a list. If so, it will be true. If a single scalar is required, then it will be false. If the caller isn't planning to do anything with what we give it, it will be the undefined value. So, for instance, we can emulate `localtime` like this:

```
#!/usr/bin/perl
# seconds2.plx
use warnings;
use strict;
my ($hours, $minutes, $seconds) = secs2hms(3723);
print "3723 seconds is $hours hours, $minutes minutes and $seconds seconds\n";
my $time = secs2hms(6868);
print "6868 seconds is $time\n";

sub secs2hms {
   my ($h,$m);
   my $seconds = shift;
   $h = int($seconds/(60*60)); $seconds %= 60*60;
   $m = int($seconds/60);      $seconds %= 60;
   if (wantarray) {
      return ($h,$m,$seconds);
   }
   return "$h hours, $m minutes and $seconds seconds";
}
```

```
>perl seconds2.plx
3723 seconds is 1 hours, 2 minutes and 3 seconds
6868 seconds is 1 hours, 54 minutes and 28 seconds
>
```

To be honest, however, it's pretty unlikely that you'll ever do this: It's best to have a subroutine that returns the same thing all the time, unless it's being used by someone other than yourself.

Subroutine Prototypes

If your subroutines are likely to be used by someone else, you might want to consider using subroutine prototypes. You'll also need to think about these if you're planning on passing more than one array to a subroutine. We'll look later at how that is done.

A subroutine prototype tells Perl what sort of arguments it's expecting. This can be used to check to ensure that the user is passing the right number of parameters, and it can also change the way Perl reads your program. For instance, you can make it possible to leave off the brackets from around your parameters, in the same way that `print "one", "two";` is the same as `print("one","two");` and you can chose whether:

```
print mysub "one", "two";
```

means:

```
print( mysub("one", "two") );
```

or:

```
print( mysub("one"), "two" );
```

That is, how many arguments your subroutine should swallow up.

Prototypes talk about the number of scalars we allow, and we use a dollar sign for each one. So, the prototype for a subroutine that takes two arguments would be `$$`. Prototypes come between the name and the block of the subroutine definition, in brackets, like this:

```
sub sum_of_two_squares ($$) {
   my ($a,$b) = (shift, shift);
   return $a**2+$b**2;
}
```

The problem is, just like when we wanted to use subroutines without the brackets, Perl hadn't read as far as their definition when it came across the call and so didn't know what to expect. When using prototypes we need to ensure that Perl gets to read the prototype before we use the subroutine, and to do this, we can use a forward definition at the top of the program, like so:

```
#!/usr/bin/perl
# sumsquare.plx
use warnings;
use strict;
sub sum_of_two_squares ($$);
```

Try It Out : Using Prototypes

Now if we try to give any more or less than two parameters, Perl complains even before the program starts:

```
#!/usr/bin/perl
# sumsquare.plx
use warnings;
use strict;
sub sum_of_two_squares ($$);

my ($first, $second) = @ARGV;
print "The sum of the squares of $first and $second is ";
print sum_of_two_squares($first, $second),"\n";

print sum_of_two_squares($first, $second, 0),"\n";

sub sum_of_two_squares ($$) {
   my ($a,$b) = (shift, shift);
   return $a**2+$b**2;
}
```

We try to use three parameters, but Perl won't allow it because we've told it only to accept two:

```
>perl sumsquare.plx 10 20
Too many arguments for main::sum_of_two_squares at sumsquare.plx line 11, near "0)"
Execution of sumsquare.plx aborted due to compilation errors.
>
```

If we comment out that line, it works as expected:

```
> perl sumsquare.plx 10 20
The sum of the squares of 10 and 20 is 500
>
```

You can specify that the number may vary by the use of a semicolon in the prototype. Everything after the semicolon is tentative; you can also use an `@_` sign to denote 'any number of parameters'.

Understanding Scope

It's now time to have a serious look at what we're doing when we declare a variable with `my`. The truth, as we've briefly glimpsed it, is that Perl has two types of variable. One type is the **global variable** (or **package variable**), which can be accessed anywhere in the program. The second type is the **lexical variable,** which we declare with `my`.

Global Variables

Global variables are what you get if you don't use `my`. If we were to say:

```
#!/usr/bin/perl
$x = 10;
```

then $x would be a global variable. They're also called package variables because they live inside a package (a package is just a convenient place to put subroutines and variables).

When we start programming, we're in a package called main. If we assign $x, as above, then we create a package variable $x in package main. Perl knows it by its full name, $main::x – the variable $x in the main package. But because we're in the main package when we make the assignment, we can just call it by its short name, $x. It's like the phone system – you don't have to dial the area code when you call someone in the same region as you.

We can create a variable in another package by using a fully-qualified name. Instead of the main package, we can have a package called Fred. Here we'll store all of Fred's variables and subroutines. So, to get at the $name variable in package Fred, we say $Fred::name, like this:

```
$x = 10;
$Fred::name = "Fred Flintstone";
```

The fact that it's in a different package doesn't mean we can't get at it. Remember that these are global variables, available from anywhere in our program. All packages do is give us a way of subdividing the namespace.

What do we mean by 'subdividing the namespace'? Well, the namespace is the set of names we can give our variables. Without packages, we could only have one $name. What packages do is help us make $name in package Fred different to $name in package Barney and $name in package main.

```
#!/usr/bin/perl
# globals.plx
use warnings;
$main::name   = "Your Name Here";
$Fred::name   = "Fred Flintstone";
$Barney::name = "Barney Rubble";

print "\$name in package main   is $name\n";
print "\$name in package Fred   is $Fred::name\n";
print "\$name in package Barney is $Barney::name\n";
```

```
> perl globals.plx
$name in package main   is Your Name Here
$name in package Fred   is Fred Flintstone
$name in package Barney is Barney Rubble
```

You can change what package you're currently working in with the aptly named package operator. We could write the above like this:

```
#!/usr/bin/perl
# globals2.plx
use warnings;
$main::name   = "Your Name Here";
$Fred::name   = "Fred Flintstone";
$Barney::name = "Barney Rubble";

print "\$name in package main   is $name\n";
package Fred;
print "\$name in package Fred   is $name\n";
package Barney;
print "\$name in package Barney is $name\n";
package main;
```

When `use strict` is in force, it makes us use the full names for our package variables. If we try and say this:

```
#!/usr/bin/perl
#strict1.plx
use warnings;
use strict;
$x = 10;
print $x;
```

Perl will give us an error – Global symbol "$x" requires explicit package name. The package name it's looking for is `main`, and it wants us to say `$main::x`

```
#!/usr/bin/perl
#strict2.plx
use warnings;
use strict;
$main::x = 10;
print $main::x;
```

As we've seen before, we can also use the `our` operator to tell Perl that a given variable should be treated as a package variable in the current package. This works just as well:

```
#!/usr/bin/perl
#strict3.plx
use warnings;
use strict;
our $x;
$x = 10;
print $x;
```

Global variables can be accessed and altered at any time by any subroutine or assignment that you care to apply to it. Of course, this is handy if you want to store a value – for instance, the user's name – and be able to get it anywhere.

It's also an absolute pain in the neck when it comes to subroutines. Here's why:

```
$a = 25;
$b = some_sub(10);
print $a;
```

Looks innocent, doesn't it? Looks like we should see the answer 25. But what happens if `some_sub` uses and changes the global `$a`? Any variable anywhere in your program can be wiped out by another part of your program. We call this 'action at a distance', and it gets real spooky to debug. Packages alleviate the problem, but to make sure that we never get into this mess, you have to ensure that every variable in your program has a different name. In small programs, that's feasible, but in huge team efforts, it's a nightmare. It's far clearer to be able to restrict the possible effect of a variable to a certain area of code, and that's exactly what lexical variables do.

Lexical Variables

The range of effect that a variable has is called its **scope**, and lexical variables declared with `my` are said to have **lexical scope,** that is, they exist from the point where they're declared until the end of the enclosing block, brackets, subroutine, or file. The name 'lexical' comes from the fact that they're confined to a well-defined chunk of text.

Each block has got a 'pad' in which it keeps its current lexical variables, if any. If Perl doesn't find the variable you're referring to in the current pad, it'll look to the surrounding blocks until it finds it – or doesn't. Every time you say `my`, you're creating a new variable attached to the current pad. It's completely independent of any variables in other pads, and you use can use it to 'hide' similarly-named lexicals that exist outside of the current block:

```
my $x;
$x = 30;
{
   my $x; # New $x
   $x = 50;
   # We can't see the old $x, even if we want to.
}
# This $x is, and always has been, 30.
```

Great. We can now use variables in our subroutines in the knowledge that we're not going to upset any behavior outside them. We know that if we say:

```
sub strip {
   my $input = shift;
   $input =~ s/^\s+//;
   $input =~ s/\s+$//;
   return $input;
}
```

that we're not going to clobber any other `$input` in the program. The highlighted part shows you the lifespan of the variable: It comes into existence at the `my` statement and goes away at the end of the nearest set of braces. We say that it 'goes out of scope' at the end of the subroutine. Once it's out of scope, we shouldn't expect to be able to get to it again. In a sense, we've created a temporary variable.

Runtime Scope

However, we can't use this trick for global variables, and Perl's special variables such as `$_` and `$/` are globals. What can we do to temporarily set their value? One way to do it is like this:

```
sub slurp {
   my $save = $/;
   undef $/;
   my $file = <>;
   $/ = $save;
   return $file;
}
```

That is, we can save away the current contents to a separate variable, and replace `$/` with its old contents when we're finished. Alternatively, we can get Perl to do the saving and restoring for us automatically: to give a global variable a specific local value, use the `local` operator:

```
sub slurp {
   local $/ = undef;
   my $file = <>;
   return $file;
}
```

`local` gives a variable **runtime scope**. This means that any statement executed between `local` and the end of the block will see the new value of the variable. How does this differ from lexical scope? The key is that, as we've seen in this chapter, program flow doesn't just go straight through blocks of code We can temporarily bounce off into subroutines, too. So, the difference is:

> **Runtime scope means a variable has a temporary value for the duration of the current block, inclusive of any side trips into other subroutine blocks, that is seen everywhere in the program – because it's a global. Lexical scope, on the other hand, creates a variable that is only visible to the statements inside the block.**

Try It Out : Runtime Scope

This program uses `local` to give `$_` a runtime scope. You should be able to see how `local` differs from `my`:

```
#!/usr/bin/perl
# runtime.plx
use strict;
use warnings;
my $x = 10;              # Line 5
$_ = "alpha";
{
   my $x = 20;
   local $_ = "beta";
   somesub();            # Line 10
}
somesub();

sub somesub {
   print "\$x is $x\n";
   print "\$_ is $_\n";
}
```

```
>perl runtime.plx
$x is 10
$_ is beta
$x is 10
$_ is alpha
>
```

How It Works

Can you see what's happening? Although we say `my $x = 20;` on line 8, that only affects statements between line 8 and the end of the block, which is line 11. It's a lexical variable that is constrained by the actual text, not by the order of execution. It doesn't have any effect when we call `somesub` on line 10. `local`, on the other hand, affects everything we do between lines 9 and 11, and that includes calling `somesub`. Its scope is determined by the statements that get executed.

When to Use my() and When to Use local

Mark-Jason Dominus gives simple but effective advice:

> **Don't use `local`. Always use `my`.**

This is somewhat of an overstatement, but it's a justified one. Unless you're dealing with special variables like `$/`, you usually want to use `my`. If you need to lie to Perl for some period of time about a global's value, try rethinking your design.

Passing More Complex Parameters

Sometimes we want to pass things other than an ordinary list of scalars, so it's important to understand how passing parameters works.

@_ Provides Aliases!

Remember when we did something like this:

```
@array = (1, 2, 3, 4);
for (@array) {
   $_++;
}
print "@array\n";
```

We found that this would print "2, 3, 4, 5". The elements of the array had been affected. We said then that the iterator variable is an alias to the elements of the list. Well, the same goes for the elements of `@_`. They're actually aliases for the things we pass. That's why we've got to be careful when we're dealing with `@_` directly. It's dangerous to say, for example:

```
sub add_one_and_double {
   $_[0]++;
   return $_[0]*2;
}
```

because if we tried:

```
add_one_and_double(1);
```

Perl would try to modify a constant, which is by definition impossible. Hence, we tend to avoid using `@_` directly and instead make local copies of the arguments, either wholesale into an array:

```
my @args = @_;
```

into named variables as a group:

```
my ($filename, $title, $description) = @_;
```

or individually by calling `shift` (especially if the number of parameters can vary):

```
my $filename    = shift;
my $title       = shift;
my $description = shift;
```

`@_` has, effectively, runtime scope. Each subroutine has its own copy of `@_`, meaning that if one subroutine calls another, we have not lost the argument values to one of them:

```
#!/usr/bin/perl
# subscope.plx
use warnings;
use strict;

first(1,2,3);

sub first {
   print "In first, arguments are @_\n";
   second(4,5,6);
   print "Back in first, arguments are @_\n";

}
sub second {
   print "In second, arguments are @_\n";
}
```

```
In first, arguments are 1 2 3
In second, arguments are 4 5 6
Back in first, arguments are 1 2 3
```

The question of which variable has scope to where can often be quite tricky to answer, but remember that a lot of trouble may be avoided by naming your variables wisely in the first place.

Lists Always Collapse

We've seen this before, but it's worth saying it again: when you put an array inside a list, the list collapses. The original structure of the array is lost, even before we start putting anything in the parameter array `@_`. That's why you can't say something like:

```
check_same(@a, @b)
```

and expect to work out where `@a` ends and `@b` starts. As far as Perl's concerned there's just one list there. To get around this, you can use references.

Passing References to a Subroutine

There's actually nothing special about passing references into a subroutine, so long as we remember that we can modify the original value when we dereference:

```
#!/usr/bin/perl
# subrefs1.plx
use warnings;
use strict;

my $a = 5;
increment(\$a);
print $a;

sub increment {
   my $reference = shift;
   $$reference++;
}
```

However, what we can do is use prototypes to take a reference behind the scenes. If in a prototype, instead of a dollar sign, we give a type symbol followed by a backslash, Perl will automatically take a reference to that type of variable. So, `sub something (\$)` will look for a single scalar variable and take a reference to it. `sub something ($\%$)` looks for a scalar, a hash, and a scalar and will take a reference to the hash.

For instance, if we change the above to:

```
#!/usr/bin/perl
# subrefs2.plx
use warnings;
use strict;
sub increment (\$);


my $a = 5;
increment($a);
print $a;


sub increment (\$) {
   my $reference = shift;
   $$reference++;
}
```

Notice how we no longer need to take the reference ourselves. We can just say `increment($a)` instead of `(\$a)`. Other languages call this **pass by reference**, as opposed to **pass by value**. Actually, all we're doing is passing a reference and Perl constructs that for us.

This is exactly how we get arrays and hashes to keep their structure when we're passing them to a subroutine.

Passing Arrays and Hashes to a Subroutine

Because the prototype can make a reference for us, we can actually take arrays, hashes and more complicated data structures and let them keep their structure.

Try It Out : Passing Arrays

So, to see if two arrays have the same contents, you could do this:

```
sub check_same (\@\@) {
   my ($ref_one, $ref_two) = @_;
   # Same size?
   return 0 unless @$ref_one == @$ref_two;
   for my $elem (0..$#$ref_one) {
      return 0 unless $ref_one->[$elem] eq $ref_two->[$elem];
   }
   # Same if we got this far
   return 1;
}
```

Putting that into a program looks like this:

```
#!/usr/bin/perl
# passarray.plx
use warnings;
use strict;

sub check_same (\@\@);

my @a = (1, 2, 3, 4, 5);
my @b = (1, 2, 4, 5, 6);
my @c = (1, 2, 3, 4, 5);
print "\@a is the same as \@b" if check_same(@a,@b);
print "\@a is the same as \@c" if check_same(@a,@c);


sub check_same (\@\@) {
   my ($ref_one, $ref_two) = @_;
   # Same size?
   return 0 unless @$ref_one == @$ref_two;
   for my $elem (0..$#$ref_one) {
      return 0 unless $ref_one->[$elem] eq $ref_two->[$elem];
   }
   # Same if we got this far
   return 1;
}
```

As expected:

```
>perl passarray.plx
@a is the same as @c
>
```

How It Works

Using the prototype here and at the top of the program means that Perl will take references to two arrays. Hence, what we'll see in `@_` are two array references:

```
sub check_same (\@\@) {
   my ($ref_one, $ref_two) = @_;
```

> **If you use a prototype at the start of your program as a forward definition, you must explicitly use the same prototype again at the definition proper, or Perl will complain of a prototype mismatch.**

We can special-case check the size: if our arrays aren't the same size, there's no way they can be the same.

```
   return 0 unless @$ref_one == @$ref_two;
```

Now we come to the comparison. We're going to stop as soon as we find something that differs, since that proves that they're not the same:

```
   for my $elem (0..$#$ref_one) {
      return 0 unless $ref_one->[$elem] eq $ref_two->[$elem];
   }
```

If we got to the end of the array and we didn't return, then they didn't differ:

```
   return 1;
```

> **This only works when we're passing something to a subroutine. We can't do a similar trick for returning arrays, and hence**
>
> **`(@a, @b) = somesub();`**
>
> **will never work. The list will be flattened, there'll be no way to tell where `@a` ends and `@b` begins, and everything will end up in `@a`. If you need to do this, pass references to the arrays and have the subroutine fill them.**

Passing Filehandles to a Subroutine

Passing filehandles to a subroutine is somewhat special. You can actually either pass a glob or a reference to a glob. It doesn't make any difference. You can then collect the filehandle into a glob, like this:

```
sub say_hello {
   *WHERE = shift;
   print WHERE "Hi there!\n"
}
say_hello(*STDOUT);
```

Alternatively, you can also collect the filehandle into an ordinary scalar and use that in place of a filehandle, as we do below:

```
sub say_hello {
   my $fh = shift;
   print $fh "Hi there!\n"
}
sub get_line {
   my $fh = shift;
   my $response = <$fh>;
   chomp $response;
   $response =~ s/^\s+//;
   return $response;
}

say_hello(*STDOUT);
get_line (*STDIN );
```

Default Parameter Values

One thing that's occasionally useful is the ability to give the parameters for your subroutine a default value, that is, give the parameter a value to run through the subroutine with if one is not specified when the subroutine is called. This is very easily done with the || operator.

The logical or operator, ||, has a very special feature: it returns the last thing it saw. So, for instance, if we say `$a = 3 || 5`, then `$a` will be set to 3. Because 3 is a true value, it has no need to examine anything else, and so 3 is the last thing it sees. If, however, we say `$a = 0 || 5`, then `$a` will be set to 5; 0 is not a true value, so it looks at the next one, 5, which is the last thing it sees.

Hence, anything we get from `@_` that doesn't have a true value can be given a default with the || operator. We can create subroutines with a flexible number of parameters and have Perl fill in the blanks for us:

```
#!/usr/bin/perl
# defaults.plx
use warnings;
use strict;

sub log_warning {
   my $message = shift || "Something's wrong";
   my $time    = shift || localtime; # Default to now.
   print "[$time] $message\n";
}

log_warning("Klingons on the starboard bow", "Stardate 60030.2");
log_warning("/earth is 99% full, please delete more people");
log_warning();
```

```
>perl defaults.plx
[Stardate 60030.2] Klingons on the starboard bow
[Wed May 3 04:07:50 2000] /earth is 99% full, please delete more people
[Wed May 3 04:07:51 2000] Something's wrong
>
```

One by-product of specifying defaults for parameters is the opportunity to use those parameters as flags. Your subroutine can then alter its functionality based on the number of arguments passed to it.

Named Parameters

One of the more horrid things about calling subroutines is that you have to remember which order the parameters are set. Was it username first and then password, or host first and then username, or…?

Named parameters are a neat way of solving this. What we'd rather say is something like this:

```
logon( username => $name, password => $pass, host => $hostname);
```

and then give the parameters in any order. Now, Perl makes this really, really easy because that set of parameters can be thought of as a hash:

```
sub logon {
   die "Parameters to logon should be even" if @_ % 2;
   my %args = @_;
   print "Logging on to host $args{hostname}\n";
   ...
}
```

Whether and how often you use named parameters is a matter of style. For subroutines that take lots of parameters, some of which may be optional, it's an excellent idea; For those that take two or three parameters, it's probably not worth the hassle.

References to Subroutines

Just like variables, you can take references to subroutines. That's where the ampersand (&) type symbol comes in.

Declaring References to Subroutines

The same rules apply here as for taking references to variables. Put a backslash before the name, but include the ampersand:

```
sub something { print "Wibble!\n" }

my $ref = \&something;
```

Alternatively, we can create an anonymous subroutine by saying `sub {BLOCK}`:

```
my $ref = sub { print "Wibble!\n" }
```

Calling a Subroutine Reference

Just like before, there are two ways to call subroutine references. Directly:

```
&{$ref};
&{$ref}(@parameters);
&$ref(@parameters);
```

Or through an arrow notation:

```
$ref->();
$ref->(@parameters);
```

Callbacks

OK, now we can create and use subroutine references. Why would we want to? The usual thing we do with them is pass them to another subroutine. This is called a **callback**, because it allows the subroutine to 'call back' our code at certain times. This means we can turn a very general subroutine into something that does exactly what we want.

Try It Out : Using a Callback

For instance, the core module `File::Find` will give us a subroutine called `find`. This takes two (or more) parameters: a callback and a list of directories. All it does – and this is a harder task than it sounds – is go through every file underneath each directory in the list, walk into any directories it finds, and call the callback with certain variables set. We can use this to create a directory browser:

```
#!/usr/bin/perl
# biglist.plx
use warnings;
use strict;
use File::Find;
find ( \&callback, "/") ; # Warning: Lists EVERY FILE ON THE DISK!

sub callback {
   print $File::Find::name, "\n";
}
```

Or we could delete every file whose name ends in `.bak`: (a typical extension for temporary backup files):

```
#!/usr/bin/perl
# backupkill.plx
use warnings;
use strict;
use File::Find;
find ( \&callback, "/") ;

sub callback {
   unlink $_ if /\.bak$/;
}
```

or indeed, anything we want. We'll see more of `File::Find` in Chapter 10, where we'll explain how these examples work. We'll also see at the end of the book that callbacks are particularly important for graphical applications.

Arrays and Hashes of References to Subroutines

Another use for subroutine references is to allow us to call one of a selection of subroutines. For instance, if we're writing a menu system that calls a subroutine related to each menu option. We could naturally write it like this:

```
print "Type c for customer menu, s for sales menu and o for orders menu.\n";
chomp (my $choice = <>);
if      ($choice eq "c") {
   customer_menu();
} elsif ($choice eq "s") {
   sales_menu();
} elsif ($choice eq "o") {
   orders_menu();
} else {
   print "Unknown option.\n";
}
```

However, that's messy. What we're doing is relating a string to a subroutine, and relating one thing to another in Perl should always make you think of a hash. Here's how we could use a hash of subroutine references:

```
my %menu = (
   c => \&customer_menu,
   s => \&sales_menu,
   o => \&orders_menu
}
print "Type c for customer menu, s for sales menu and o for orders menu.\n";
chomp (my $choice = <>);
if (exists $menu{$choice}) {
   # Call it!
   $menu{$choice}->();
} else {
   print "Unknown option.\n";
}
```

Much neater.

Recursion

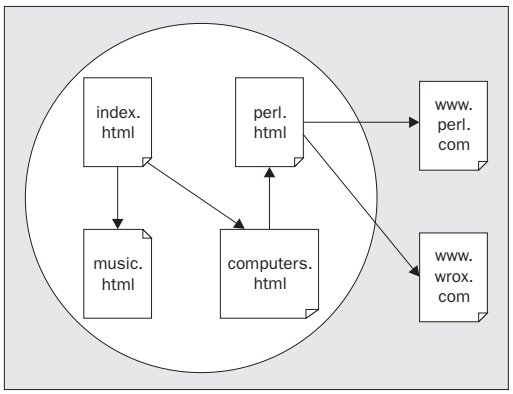
recursion, *n.*: *See* recursion

The above joke, so old it has hair on it, gives you an idea as to what recursion is – it's something that refers to itself in its definition. Specifically, recursion in computer programming is a subroutine that calls itself as part of its operation.

Of course, we have to be careful when we're doing this: we've got to make sure we stop somewhere and that our programs don't loop away into oblivion. The thing that tells us when to stop is called the **terminating condition**.

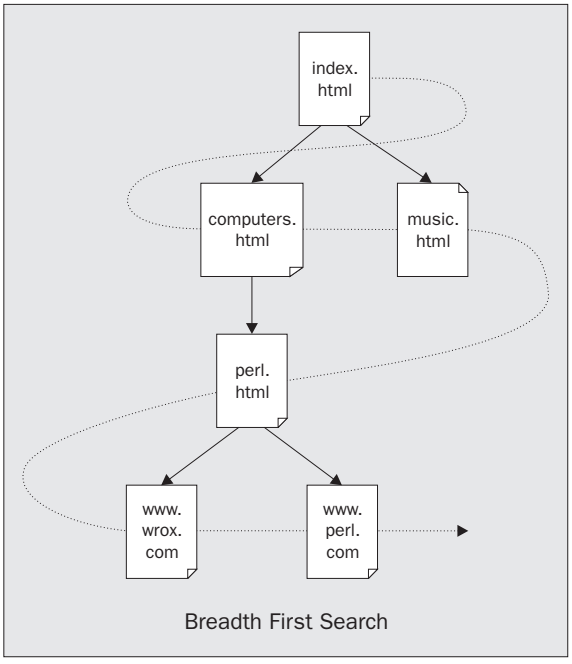
Try It Out : Spidering a Web Site

A web site is a collection of pages linked together in some way. If you're running a web site, you might want to ensure that all the links work properly: that the pages inside your site can be read and that links to other sites on the Internet are still valid. The general procedure we need to follow is something like this: to check a page, get the web page, extract all links, get those pages to ensure that they are valid and reachable, and then check those pages still on our site. So, let's say we had the following set of pages:



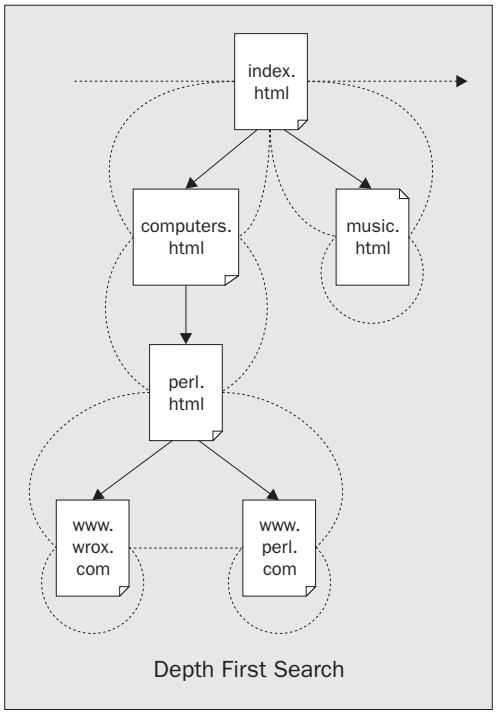
We'd start at http://www.mysite.org/index.html, and from there we'd find links to `computers.html` and `music.html` – we'd want to check each of these. Examining each of those for links would give us a link to `perl.html`, where we'd find links to http://www.perl.com/ and http://www.wrox.com/. We'd want to make sure that these pages were reachable, but since these were off our site, we wouldn't examine them for further links. Any broken pages beyond that are not something we can do anything about.

Now, you should notice that there are two routes we can take to do this, starting from `index.html`: We could see extract links on the first level, `computing.html` and `music.html`. Then we could visit the links we got from that level, `perl.html` Then we could go to the external sites, where we'd have to stop. That's called a **breadth-first search**, and it looks like this:



The important thing about a breadth-first search is that for each 'level' we need to keep track of which links to visit on the next level. It's what we did when traversing the tree of references at the end of Chapter 6. This isn't recursive, because we're not doing exactly the same thing with each site we get to.

However, there's another way we could do this which avoids the need to explicitly keep track of where we're going next time: we could go first to `computers.html`, then follow the link to `perl.html`, then follow the external links, and then back up and visit `music.html`. If there's a link, we visit it. If not, we go back to where we were. This is a **depth-first search**, and we can implement it recursively:



Depth First Search

Notice how each visit to a page is a similar shape to the ones above? That's the recursion. And notice how on external sites and pages with no links, we close the loop and head off to the next page? Those are our terminating conditions. We'll also add another terminating condition, of course – don't investigate a page if the link is dead.

Here's what it looks like in Perl:

```
sub traverse {
   my $url = shift;
   return if $seen{$url}++;        # Break circular links
   my $page = get($url);
   if ($page) {
      print "Link OK : $url\n";
   } else {
      print "Link dead : $url\n";
      return;                    # Terminating condition : if dead.
   }
   return unless in_our_site($url); # Terminating condition : if external.
   my @links = extract_links($page);
   return unless @links;            # Terminating condition : no links
   for my $link (@links) {
      traverse($link) # Recurse;
   }
}
```

Now let's turn that into a full program:

```
#!/usr/bin/perl
# webchecker.plx
use warnings;
use strict;
my %seen;

print "Web Checker, version 1.\n";
die "Usage: $0 <starting point> <site base>\n"
   unless @ARGV == 2;

my ($start, $base) = @ARGV;
$base .= "/" unless $base=~m|/$|;

die "$start appears not to be in $base\n"
   unless in_our_site($start);
traverse($start);

sub traverse {
   my $url = shift;
   $url =~ s|/$|/index.html|;
   return if $seen{$url}++;        # Break circular links
   my $page = get($url);
   if ($page) {
      print "Link OK : $url\n";
   } else {
      print "Link dead : $url\n";
      return;                    # Terminating condition : if dead.
   }
   return unless in_our_site($url); # Terminating condition : if external.
   my @links = extract_links($page, $url);
   return unless @links;            # Terminating condition : no links
   for my $link (@links) {
      traverse($link) # Recurse
   }
}

sub in_our_site {
   my $url = shift;
   return index($url, $base) == 0;
}

sub get {
   my $what = shift;
   sleep 5; # Be friendly
   return `lynx -source $what`;
}

sub extract_links{
   my ($page, $url) = @_;
   my $dir = $url;
   my @links;
   $dir =~ s|(.*)/.*?$|$1|;
   for (@links = ($page=~/<A HREF=["']?([^\s"'>]+)["']?/gi)) {
      $_ = $base.$_ if s|^/||;
      $_ = $dir."/".$_ if !/^(ht|f)tp:/;
   }
   return @links;
}
```

While it isn't very polished – it's quite primitive – it works:

```
> http://www.wrox.com/Default.asp http://www.wrox.com/
Web Checker, version 1.
Link OK : http://www.wrox.com/Default.asp
Link OK : http://www.wrox.com/Consumer/DJ.asp
Link OK : http://www.wrox.com/Consumer/Store/ListTitles.asp?By=105&Category=Consumer
Link OK : http://www.wroxconferences.com
Link OK : http://www.wrox.com/Consumer/Store/ListTitles.asp?By=104&Category=Consumer
Link OK : http://www.wrox.com/Consumer/Forums/Default.asp?Category=Consumer
Link OK : http://www.wrox.com/Consumer/Store/Download.asp?Category=Consumer
Link OK : http://www.wrox.com/Consumer/EditDetails.asp?Category=Consumer
Link OK : http://www.wrox.com/Consumer/Contacts.asp
…
>
```

Now, we'll see how it works, and then we'll see what's wrong with it.

How It Works

First, we need to know two things: the first URL we're going to start with and the base for the site, so we know when we're about to visit an external site:

```
die "Usage: $0 <starting point> <site base>\n"
   unless @ARGV == 2;

my ($start, $base) = @ARGV;
```

If the base URL doesn't end with a slash, we give it one, since we depend on this fact later:

```
$base .= "/" unless $base=~m|/$|;
```

Next, we'll check that the page we're starting from is actually part of the site. You never know…:

```
die "$start appears not to be in $base\n"
   unless in_our_site($start);
```

And then we kick off the action:

```
traverse($start);
```

Now here's the subroutine we saw above, slightly modified:

```
sub traverse {
   my $url = shift;
```

If the URL ends in a slash, we treat it as an index page:

```
   $url =~ s|/$|/index.html|;
```

This is our first problem. It's a bad assumption. Some sites have the index page as `index.html`, some as `index.htm`, some as `Default.asp` – in fact, it could be anything. The only way we can tell is to look at the exact response from the server when we ask for a URL ending in a slash.

Next we need to make sure we haven't seen the page before, because web sites can have circular links and we don't want to go whizzing around forever:

```
    return if $seen{$url}++;        # Break circular links
```

And we get our page. If we successfully retrieve it, we say so. If the link is dead, there's no point trying to find other links from it:

```
    my $page = get($url);
    if ($page) {
       print "Link OK : $url\n";
    } else {
       print "Link dead : $url\n";
       return;                    # Terminating condition : if dead.
    }
```

We don't look for links in external sites:

```
    return unless in_our_site($url); # Terminating condition : if external.
```

Now we extract the links and give up if we can't find any:

```
    my @links = extract_links($page, $url);
    return unless @links;            # Terminating condition : no links
```

Now we call ourselves on each of the links:

```
    for my $link (@links) {
       traverse($link) # Recurse
    }
```

Imagine how this would work for our example site above: the first call to `index.html` would put `computers.html` and `music.html` in `@links`. Then we'd call ourselves first on `computers.html`, which would in turn call ourselves on `perl.html`, which would then check the external links and return. There's nothing else on `computers.html`, so that'd return. Then we'd move onto `music.html` and return, and we'd be done – this is a depth-first search, just like in the diagram. We look at the first link we see, every time:

```
}
```

Now we come to the helper subroutines:

```
sub in_our_site {
   my $url = shift;
   return index($url, $base) == 0;
}
```

We just check that the URL we're about to look at starts with the same characters as the base. This isn't foolproof, but it's close enough. We know that `$base` has to end in a slash, so the only things we allow if we're looking at http://www.mysite.org/ are things that start http://www.mysite.org/... It counts out FTP, HTTPS, or any other protocol, but it'll do.

```
sub get {
   my $what = shift;
   sleep 5; # Be friendly
   return `lynx -source $what`;
}
```

We use `lynx` again to get our web pages. While we make the effort to be friendly to the web servers by not bombarding them with requests as fast as we can, we don't check that the page we get back is valid. Sometimes if we're behind a cache and we request a dud site, we'll get back a perfectly fine page – with an error message on it! We don't do any error checking here at all! Again, the only way to be really sure is to connect to the server directly and examine the exact response.

> **Thankfully, we don't have to do all that work. There's a module called `LWP::Simple` which provides a subroutine, also called `get`, which does the job properly. If you've got that installed, just add `use LWP::Simple;` after the `use strict;` line, and remove this subroutine. If not, we'll be looking at it and how it works in Chapter 10.**

Now we try and extract the links from the HTML file. There are two problems here: first, finding and extracting the links, and second turning them into real URLs. In order to prepare us for the second problem, we take the URL we've just looked up, and extract the directory name from it:

```
my $dir = $url;
$dir =~ s|(.*)/.*$|$1|;
```

This'll turn http://mysite.org/pictures/index.html into http://mysite.org/pictures, or so we hope.

Now we try and extract the links:

```
for (@links = ($page=~/<A HREF=["']?([^\s"'>]+)/gi)) {
```

We look for all examples of `<A HREF=` followed by an optional double or single quote mark, and then some text that isn't a space, quote mark, or closing tag that we extract. This should extract all the links, right?

I've said before that parsing HTML using regular expressions is a potentially risky operation, and I stand by it. This makes a couple of assumptions that may not always hold true:

- `HREF` always follows `A` with a single space and no elements in between.
- There are no spaces, greater-than signs or quotation marks in the URL. According to the standards, there won't be, but the standards aren't always adhered to.
- This piece of text won't be found inside a `<PRE>` tag, a comment tag, or anything else that changes it from the usual meaning.
- This URL doesn't contain a #-sign to point to a spot in the middle of the page.

And so on. A lot of these assumptions are usually going to be true, but we can't rely on them. As before, the only way to be sure is to go through and check the data piece by piece, and as before there's a module that does this for us. `HTML::LinkExtor` is designed to extract links from HTML files, but it's pretty tricky to use. Further, it's about a hundred times slower. When you must have the right answer, use that; when 'close enough is good enough', use the above.

Now, there are two types of filename that we'll find in there. Absolute URLs include the Internet host they're coming from: for example, http://www.mysite.org/perl.html. Relative URLs, on the other hand, speak about a file on the same server as the current on: for instance, from http://www.mysite.org/perl.html, we could say `/music.html` to get to http://www.mysite.org/music.html. Because relative URLs only give you directions from the current page, not from anywhere on the

Internet, we have to convert them to absolute URLs before looking them up. This is why we need to know the directory name of the file we're currently looking at. The rules for turning relative URLs into absolute ones are tricky, but we simplify them here:

- If the URL starts with a forward slash, it should be taken as from the base of the site. (This is another dangerous assumption – the base of the site may not be the base of the server, which is where relative URLs are really measured from.) Since we know `$base` ends with a forward slash, we string the initial slash from our relative URL and glue them together.
- Otherwise, if it doesn't start with `http://` or `ftp://`, it's a relative URL and we need to add the URL for the current directory to the beginning of it:

```
$_ = $base.$_ if s|^/||;
$_ = $dir."/".$_ if !/^(ht|f)tp:/;
```

Again, this may well work for some – or maybe even most – cases, but it's not a complete solution. It doesn't take into account things like the fact that saying "./" refers to another file in the current directory. As usual, there's a module – URI – which can help us convert between relative and absolute URLs.

Hopefully, I've made you think about some of the assumptions that you can make in your programming and why you need to either cover every case you can think of – or get someone else to cover it for you. In Chapter 10, we'll be looking at modules like the ones mentioned here that can get these things right for us.

Style Point: Writing Big Programs

Subroutines give us the perfect opportunity to think about that it means to program, and how to approach the programming problem. Learning the bricks and mortar of a programming language is one thing, but learning how to put them all together into a complex program is quite another.

One approach, then, is to separate out the various components; if a program's going to be performing a variety of tasks, you're obviously going to need to write code for each one. So, stage one in building a large program:

- Identify what the program will do.

This turns the question from 'How do I write a program which handles my business?' into 'How do I write a program which does X, Y, and Z?' We've now identified individual goals and turned a very general problem into a little more specific one. Now we've got to work out how to achieve those goals. It might be useful at this stage to break the goals into manageable chunks; this is where subroutines can be a useful mirror of the development process.

- Break down goals into a series of ideas

Imagine you're directing a robot. You've got a chair in front of you, and a wardrobe over on one wall. On top of the wardrobe is a box, and you want the robot to bring you the box. Unfortunately, you can't just say 'bring me the box'; that's not a primitive enough operation for the robot. So you have to consider the stages involved and write them out explicitly. So, our draft program would go:

```
Put chair in front of wardrobe.
Stand on chair.
Pick up box.
Get down off chair.
Move to human.
Put down box.
```

That'd certainly be enough for most humans, but it probably wouldn't be enough for most robots. If they don't know about "put something somewhere", you're going to have to break it down some more. This is where subroutines come in, to break down the big tasks into simpler goals:

```
sub "Put chair in front of wardrobe" {
    Move to chair
    Pick up chair
    Move to wardrobe
    Put down chair
}
```

Incidentally, this way of interspersing English descriptions with programming terminology to describe the outline of a program is called "pseudocode" – it's one popular way to plan out a program.

Of course, you may find you have to define things like 'move' and 'pick up' in terms of individual movements; this depends on the tools already provided for you. With Perl, you've got a reasonably high-level set of tools to play with; you don't have to break up strings yourself, for instance, as you do in some languages. Getting to the computer's level is our final stage:

- Specify each idea in a way the computer can understand.

Easier said than done, of course, because it means you need to know exactly what the computer can and can't understand; thankfully, though, computers are more than able to tell you when they can't understand something. Effectively, though, programming is just explaining how you want a task to be performed, in simple enough stages. Subroutines give you the ability to group those stages around individual tasks.

Summary

Subroutines are a bit of code with a name, and they allow us to do two things: chunk our program into organizational units and perform calculations and operations on pieces of data, possibly returning some more data. The basic format of a subroutine definition is:

```
sub name BLOCK
```

We can call a subroutine by just saying `name` if we've had the definition beforehand. If the definition is lower down in the program, we can say `name()`, and you may see `&name` used in older programs. Otherwise, we can use a forward definition to tell Perl that `name` should be interpreted as the name of a subroutine.

When we pass parameters to a subroutine, they end up in the special array `@_` – this contains aliases of the data that was passed. Prototypes allow us to specify how many parameters to accept, and they also allow us to pass references instead of aliases; this in turn allows us to pass arrays and hashes without them being flattened.

We can take references to subroutines by saying `\&name`, and use them by saying `$subref->()` or `&$subref`. We can get anonymous subroutines by saying `sub { BLOCK }` with no name. Subroutine references give us callbacks and the ability to fire off a subroutine from a set of several.

Ordinary subroutines are allowed to call other subroutines; they're also allowed to call themselves, which is called recursion. Recursion needs a terminating condition, or else the subroutinr will never end. Perl takes care of where it's going, where it came from, and the parameters that have been passed at each level.

Finally, we looked at how to divide up programs into subroutines, as well as the top-down level of programming: start with the goal, then subdivide into tasks and put these tasks into subroutines. Then subdivide again if necessary, until we've got to a level that the computer can understand.

Exercises

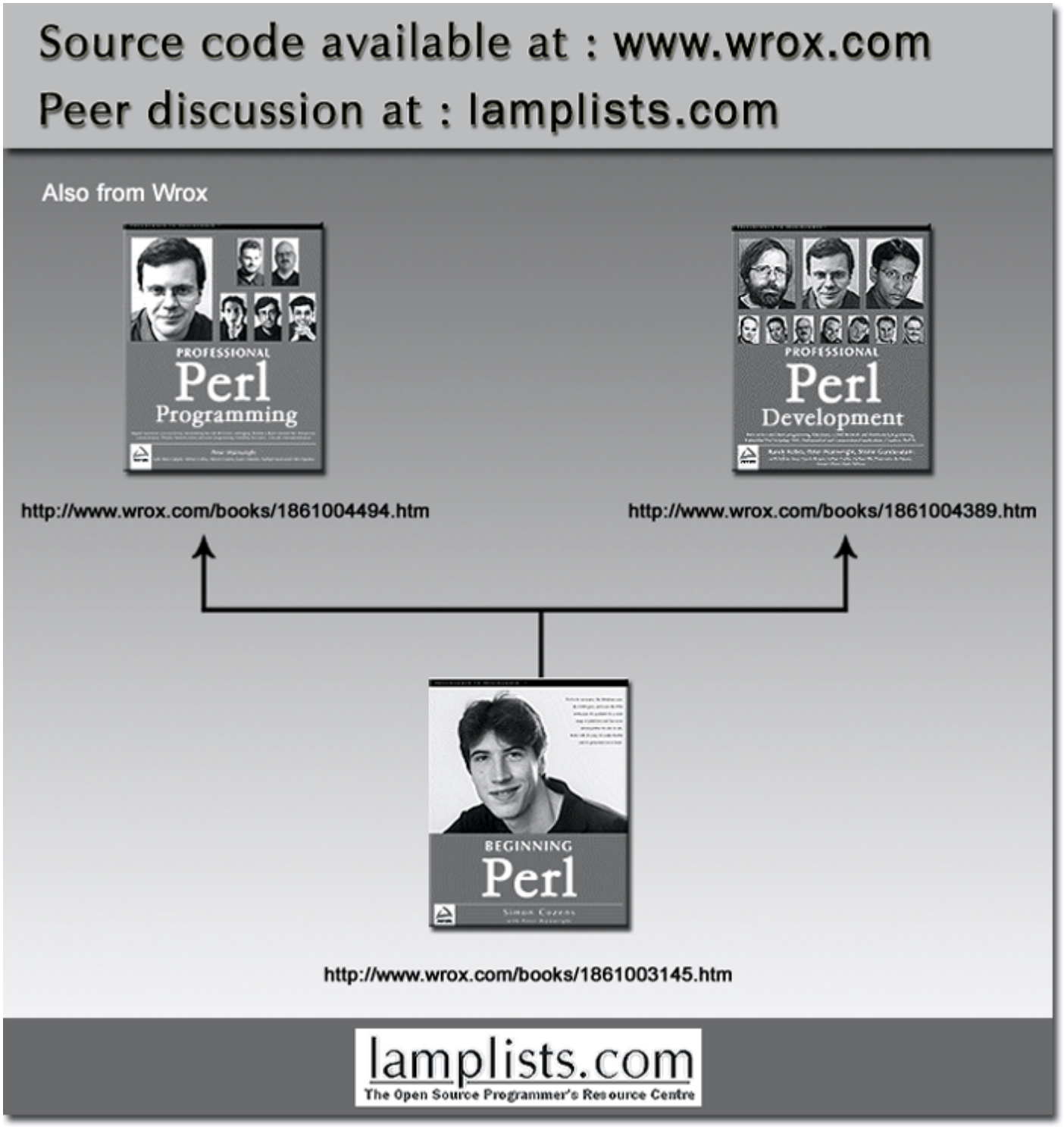
1. Go back to the seconds1.plx program seen earlier in the chapter. Rewrite it so that it contains a second subroutine that asks the user for a number, puts it into a global variable and converts that into hours, minutes, and seconds.

2. Create three subroutines such that each identify themselves on screen and then calls the next in the list – that is, sub1 calls sub2 which calls sub3 – until 300 subroutine calls have been made in total. When that does occur, break out of the loop and identify which was the last subroutine called. First do this using a global variable....

3. Repeat this exercise passing the current call number and call limit around as parameters

4. Write a subroutine that receives by reference an array containing a series of numbers, initially (1, 1). The subroutine then calculates the sum of the two most recent references and adds another element to the array that is the sum of both. Do this ten times and then print out your array. You should get the first twelve numbers of the Fibonacci sequence

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144
```



9

Running and Debugging Perl

By now, we've essentially covered the whole of the core Perl language. We've yet to see how we can use pre-packaged modules to accomplish a great many common tasks, including applying Perl to networking, CGI and database manipulation. But right now, we've finished as much of the language as you'll need to know for pretty much everything you'll want to do with Perl. Congratulations for getting this far!

You should also be getting used to analyzing the problem you want to solve, breaking it down into component parts, and thinking about how to explain those parts to the computer in a language it can understand. That's not all, however.

Everyone makes mistakes. It's a simple fact of life, and programming is just the same. When you write programs, you will make mistakes. As we mentioned in the first chapter, the name for a mistake in programming is a **bug**, and the process of removing bugs is called **debugging**. After breaking down your ideas and writing the code, you'll come to the next two phases of software development: testing and debugging.

In this chapter, we'll see how Perl helps us with these stages. In particular, we'll cover the following areas:

- **Error Messages**
  How the perl interpreter tells you you've used the language incorrectly.
- **Diagnostic Modules**
  What modules can help us isolate and understand problems with our code.
- **Perl Command Line Switches**
  Creating test programs using the perl command line.
- **Debugging Techniques and the Perl Debugger**
  How to remove the problems that we've found.

By the end of this chapter, you should be able to recognize, diagnose, and hopefully fix any programming errors you make. We'll also look at how to construct test cases and quick one-line programs on the perl command line.

Error Messages

There are two types of mistake you can make when programming: a syntax error and a logic error. A **syntax error** is something like a typo or a result of misunderstanding how to use the language, meaning that your code doesn't actually make sense any more. Since your code isn't properly written in Perl, perl can't understand it and complains about it.

A **logic error**, on the other hand, is where the instructions you give make perfect sense, but don't actually do what you think they ought to. This type of error is far more dastardly to track down but there are ways and means to do so. For the time being, though, we'll start by looking at the way Perl detects and reports syntax errors:

Try It Out : Examining Syntax Errors

Let's create a few syntax errors, and see how Perl reports them to us. Take the following program, for example:

```
#!/usr/bin/perl
# errors.plx
use warnings;
use strict;

my $a;
print "Hello, world."
$a=1;
if ($a == 1 {
   print "\n";
}
```

As you should be able to see if you look carefully, this contains a number of mistakes. This is what Perl makes of it:

```
>perl errors.plx
Scalar found where operator expected at errors.plx line 8, near "$a"
      (Missing semicolon on previous line?)
syntax error at errors.plx line 8, near "$a"
syntax error at errors.plx line 9, near "1 {"
Execution of errors.plx aborted due to compilation errors.
>
```

How it Works

What's Perl complaining about? Firstly, it sees something up on line 8:

```
$a=1;
```

Well, there's nothing wrong with that. That's perfectly valid code. When we're trying to track down and understand syntax errors, the key thing to remember is that the line number Perl gave us is *as far as it got* before realizing there was a problem – that doesn't necessarily mean that the line itself has a problem. If, for instance, we miss out a closing bracket, Perl may go all the way to the end of the file before complaining. In this case, though, Perl gives us an additional clue:

(Missing semicolon on previous line?)

In fact, this is exactly the problem:

```
print "Hello, world."
```

Line 7 doesn't end with a semicolon. But what of the error message, 'Scalar found where operator expected'? What does this mean? Like all of Perl's error messages, it means exactly what it says. Perl found a scalar where it thought there should be an operator. But why? Well, Perl had just finished processing a string, which was fed to `print`. But since there wasn't a semicolon, it was trying to find a way to continue the statement. The only way to continue would be to have an operator to link the string with something else: the concatenation operator, for instance, to connect it to another scalar. However, instead of such an operator, Perl found the scalar `$a`. Since you can't put a string right next to a variable, Perl complains, and as there's no way for this to make sense, it also gives us a 'syntax error'.

The next problem is in line 9:

```
if ($a == 1 {
```

Here we have no clue to help us track down the bug. It's a syntax error pure and simple, and we can fix it easily by providing the missing bracket. It should, of course, look like this:

```
if ($a == 1) {
```

Syntax Error Checklist

Tracking down syntax errors can be troublesome, but it's a skill that comes with practice. Most of the errors you're likely to experience are going to fall into one of the six categories below:

Missing Semicolons

We've seen this already, and it's probably the most common syntax error there is. Every statement in Perl, unless it's at the end of a block, should finish with a semicolon. Sometimes you'll get the helpful hint we got above:

(Missing semicolon on previous line?)

but otherwise you've just got to find it yourself. Remember that the line number you get in any error message may well not be the line number the problem occurs on – just when the problem is detected.

Missing Open/Close Brackets

The next most common error comes when you forget to open or close a bracket or brace. Missed closing braces are the most troublesome, because Perl sometimes goes right the way to the end of the file before reporting the problem. For example:

```
#!/usr/bin/perl
# braces.plx
use warnings;
use strict;
```

```
if (1) {
   print "Hello";

my $file = shift;
if (-e $file) {
   print "File exists.\n";
}
```

This will give us:

```
>perl braces.plx
Missing right curly or square bracket at braces.plx line 12, at end of line
syntax error at braces.plx line 12, at EOF
Execution of braces.plx aborted due to compilation errors.
>
```

The problem is, our missing brace is only at line 7, but Perl can't tell that. To find where the problem is in a large file, there are a variety of things you can do:

- Indent your code as we have done to make the block structure as clear as possible. This won't affect what perl sees, but it helps *you* to see how the program hangs together, making it more readily obvious when this sort of thing happens.
- Deliberately leave out semicolons where you think a block should end, and you'll cause a syntax error more quickly. However, you'll need to remember to add the semicolon if you add extra statements to the block.
- Use an editor which helps you out: Editors like `vi` and `emacs` automatically flash up matching braces and brackets (called **balancing**) and are freely available for both UNIX and Windows.

We'll also be looking at some more general techniques for tracking down bugs later on in this chapter.

Runaway String

In a similar vein, don't forget to terminate strings and regular expressions. A runaway string will cause a cascade of errors as code looks like strings and strings look like code all the way through your program. If you're lucky though, Perl will catch it quickly and tell you where it starts – miss off the closing `"` in line 7 of the above example, and Perl will produce this message amongst the rest of the mess:

```
(Might be a runaway multi-line "" string starting on line 7)
```

This is also particularly pertinent when you're dealing with here-documents. Let's look again at the example we saw in Chapter 2:

```
#!/usr/bin/perl
#heredoc.plx
use warnings;
print<<EOF;

This is a here-document. It starts on the line after the two arrows,
and it ends when the text following the arrows is found at the beginning
of a line, like this:

EOF
```

Since perl treats everything between `print<<EOF;` and the terminator `EOF` as plain text, it only takes a broken terminator for perl to interpret the rest of your program as nothing more than a long string of characters.

Missing Comma

If you forget a comma where there should be one, you'll almost always get the 'Scalar found where operator expected' message. This is because Perl is trying to connect two parts of a statement together and can't work out how to do it.

Brackets Around Conditions

You need brackets around the conditions of `if`, `for`, `while`, and their English negatives `unless`, `until`. However, you don't need brackets around the conditions when using them as statement modifiers.

Barewords

If an error message contains the word 'bareword', it means that Perl couldn't work out what a word was supposed to be. Was it a scalar variable and you forgot the type symbol? Was it a filehandle used in a funny context? Was it an operator or subroutine name you spelled wrong? For example, if we run:

```
#!/usr/bin/perl
#bareword.plx
use warnings;
use strict;

Hello;
```

perl will tell us:

```
>perl bareword.plx
Bareword "Hello" not allowed while "strict subs" in use at bareword.plx line 5.
Execution of braces.plx aborted due to compilation errors.
>
```

We'll see more in the section on barewords in `use strict` below.

Diagnostic Modules

Hopefully, I've already drummed into you the importance of writing `use strict` and `use warnings` in your code. Now it's time to explain what those, and other modules like them, actually do.

As we'll see in the next chapter, `use` introduces an external module, while `warnings` and `strict` are both standard Perl modules that come with the Perl distribution. They're just ordinary Perl code. The special thing about them is that they fiddle with internal Perl variables, which will alter the behavior of the perl interpreter.

Strictly speaking these are **pragmas** (or, for the linguistically inclined, *pragmata*) rather than modules. These have all lower-case names and are particularly concerned with altering the operation of perl itself, rather than providing you with ready-made code.

use warnings

The `warnings` pragma changes the way perl produces warnings. Ordinarily, there are a number of warnings that you can turn on and off, categorized into a series of areas: syntactic warnings, obsolete ways of programming, problems with regular expressions, input and output, and so on.

Redeclaring Variables

By default, all warnings are turned off. If you merely say `use warnings`, everything is turned on. So, for example, without specifying `use warnings`, the following code will execute without issue:

```
#!/usr/bin/perl
# warntest.plx
# add 'use warnings;' command here

my $a = 0;
my $a = 4;
```

However, with `use warnings` specified after the filename comment, here is what perl tells you:

```
>perl warntest.plx
"my" variable $a masks earlier declaration in same scope at warntest.plx line 6.
>
```

What does this mean? It means that in line 6, we declared a new variable `$a`. If you remember, `my` creates a completely new variable. However, we already have a variable `$a`, which we declared in line 5. By re-declaring it in line 6, we lose the old value of `0`. This is a warning in the 'misc' category.

Misspelling Variable Names

Let's see another common cause of error - misspelling variable names:

```
#!/usr/bin/perl
# warntest2.plx
# add 'use warnings;' command here

my $total = 30;
print "Total is now $total\n";
$total += 10;
print "Total is now $tutal\n";
```

Without warnings, we see this:

```
> perl warntest2.plx
Total is now 30
Total is now
```

Why has our variable lost its value? Let's turn on warnings and run this again. Now we get:

```
> perl warntest2.plx
Name "main::tutal" used only once: possible typo at warntest2.plx line 8.
Total is now 30
Use of uninitialized value in concatenation (.) at warntest2.plx line 8.
Total is now
```

Aha! A warning in the 'once' category has been fired, telling us that we've only used the variable `tutal` once. Obviously, we've misspelled `total` here.

That's enough to help us track down and fix the problem, but what about the other error: `$tutal` certainly had an uninitialized value, but where is the concatenation? We didn't use the `.` operator – however, perl did. Internally, perl understands `"something $a"` to be `"something ".$a`. Since the `$a` in this case was undefined, perl complained.

The Scope of use warnings

The `warnings` pragma is **lexically scoped**, so its effects will last throughout the same block of code as a `my` variable would – that is, within the nearest enclosing braces or the current file. For instance, the following program has warnings throughout:

```
#!/usr/bin/perl
# warntest3.plx
use warnings;

{
   my @a = qw(one , two , three , four);
}
my @b = qw(one , two , three , four);
```

Therefore perl responds with the following warnings, in the `qw` category:

```
>perl warntest3.plx
Possible attempt to separate words with commas at warntest3.plx line 6.
Possible attempt to separate words with commas at warntest3.plx line 8.
>
```

reminding us that since `qw()` automatically changes separate words into separate elements, we don't need to separate them with commas.

If you really **do** want commas as some elements of your array, you may turn warnings off by saying `no warnings`. In the following program, warnings are only turned on for the code outside the brackets:

```
#!/usr/bin/perl
# warntest4.plx
use warnings;

{
   no warnings;
   my @a = qw(one , two , three , four);
}
my @b = qw(one , two , three , four);
```

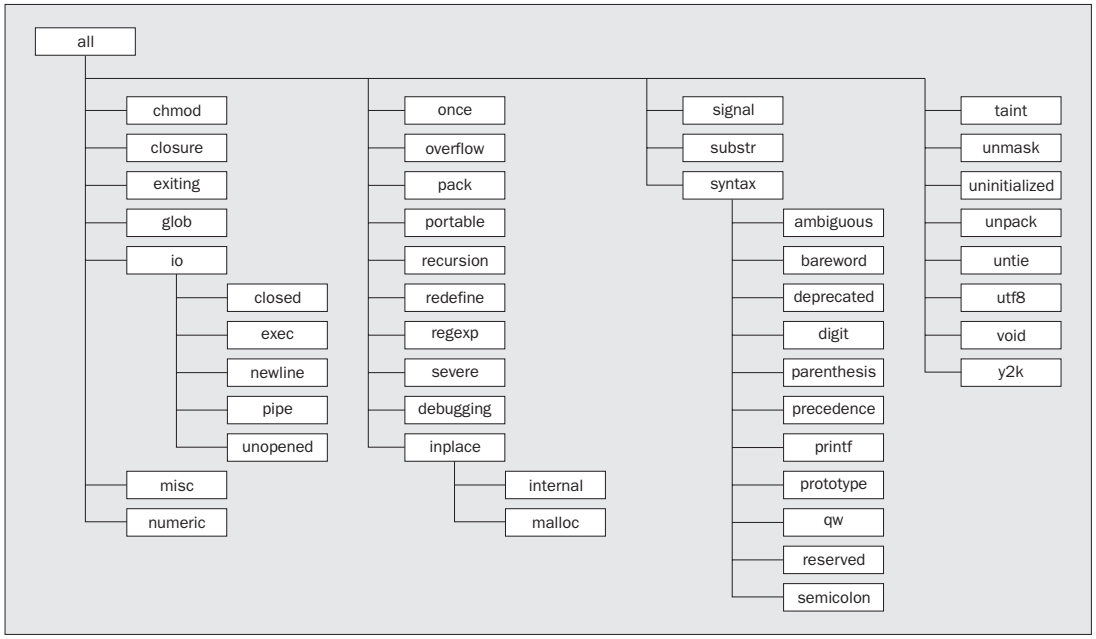
Now perl will only give the one warning, for the second array:

```
> perl warntest4.plx
Possible attempt to separate words with commas at warntest3.plx line 9.
>
```

To turn off or on certain classes of warnings, give them as a list after the `use` or `no warnings`. So, in this case, to just turn off the warning about `qw` but leave the others untouched, you would write:

```
#!/usr/bin/perl
# warntest4.plx
use warnings;

{
   no warnings "qw";
   my @a = qw(one , two , three , four);
}
my @b = qw(one , two , three , four);
```

The categories of warnings you can turn on and off are organized hierarchically as follows, and the warnings they cover are detailed in the `perldiag` documentation:



use strict

You should also know by now that `use strict` forces you to declare your variables before using them. In fact, it controls three areas of your programming: variables, references, and subroutines.

Strict on Variables

First, we'll look at the variables. When `use strict` is applied, a variable must either be declared lexically (using `my $var`) and belong to a block or file, or be declared globally, to be available in every part of the program. You can do this either by using `our $var` (in the same way as `my`) or by specifying its full name, `$main::var`. We'll see where the `main` comes from in the next chapter.

You may see another way of declaring a global, `use vars '$var'`, which does exactly the same as `our`. `our` was introduced in Perl 5.6.0 and is recommended for use, wherever backward compatibility isn't an issue.

If `use strict` applies and you have not used one of these forms of declaration, Perl will not allow you to run the program:

```
#!/usr/bin/perl
# strictvar.plx
use warnings;
use strict;

my $a = 5;
$main::b = "OK";
our $c = 10;
$d = "BAD";
```

The first three variables are fine, but:

```
>perl strictvar.plx
Global symbol "$d" requires explicit package name at strictvar.plx line 9.
Execution of strictvar.plx aborted due to compilation errors.
>
```

To fix this, you just need to use one of the above ways of declaring the variable. Here's an important lesson in debugging: don't turn off the warnings – **fix the bug**. This is especially important when we come to our next cause of problems, references.

Strict on References

One thing novice programmers often want to do is to construct a variable whose name is generated from the contents of another variable. For instance, you're totalling numbers in a file with several sections. Each time you come to a new section, you want to keep the total in another variable. So, you might think you want `$total1`, `$total2`, `$total3`, and so on, with `$section` pointing to the current section. The problem then is to create a variable out of `"total"` plus the current value of `$section`. How do you do it?

- **Honest answer**:
  You can say `${"total".$section}`.
- **Better answer**:
  Don't do it. In such cases, it's almost always better to use a hash or an array. Here, since the sections are numeric, you'd use an array of totals. It's far easier to say `$total[$section]`. More generally, if your sections are named, you'd use a hash, `$total{$section}`.

Why? Well, the most obvious reason is because you know how to use hashes and arrays, and when the question was asked, you didn't know how to construct a variable by name. Use what you know! Don't try and be too clever if there's a simple solution. Constructing these **symbolic references**, as they are known, can play havoc with any of your variables.

Suppose you're making a variable not out of `${"total".$section}` but `${$section}` where `$section` is read from the file. If reading the section name goes horribly wrong, you may have `$section` become one of Perl's special variables, either causing an error or creating weird behavior later in your program – arrays may suddenly stop working, regular expression behavior may become unpredictable, and so on. This kind of thing is a nightmare to debug.

Even if it goes right, there's no guarantee that `$section` won't contain a name you're using somewhere else in the program. A variable name you're using may be blown away at any moment by something outside your program. This isn't a pretty situation to get into, and `use strict` stops you from getting into it by disallowing the use of symbolic references.

Strict on Subroutines

Last, but not least, `use strict` disallows 'poetry optimization', which lets you use barewords as strings. This means if you want to use the name of a subroutine without brackets, you must declare the subroutine first. For example:

```
#!/usr/bin/perl
# strictsubs1.plx
use warnings;
use strict;

$a = twelve;
sub twelve { return 12 }
```

blows up with an error:

```
>perl strictsubs1.plx
Bareword "twelve" not allowed while "strict subs" in use at strictsubs1.plx line 6
Execution of strictsubs1.plx aborted due to compilation errors.
>
```

However, this is okay. You'll get the query 'Name "main::a" used only once: possible typo' but that's simply because we've declared `$a` and then not used it. We'll come back to this error in a minute:

```
#!/usr/bin/perl
# strictsubs2.plx
use warnings;
use strict;

sub twelve { return 12 }
$a = twelve;
```

Of course, you can always get round the limitation on barewords, simply by not using them. A subroutine name with parentheses is always OK:

```
#!/usr/bin/perl
# strictsubs3.plx
use warnings;
use strict;

sub twelve { return 12 }
$a = twelve();
```

These three areas – variables, symbolic references and subroutines – are split into categories just like the warnings. These are `vars`, `refs`, and `subs` respectively.

As before, `use strict` turns on all checks. You can turn on and off all or individual checks on a lexical basis just as you could with `use warnings`:

```
#!/usr/bin/perl
# nostrict.plx
use warnings;
use strict;

our $first = "this";
our $second = "first";
our $third;

{
   no strict ('refs');
    $third = ${$second};
}

print "$third\n";
```

>**perl nostrict.plx**
Name "main::first" used only once: possible typo at nostrict.plx line 6.
this
>

The warnings have been turned off for our symbolic link, but again we get that warning about only explicitly using `$first` once, even though we have indirectly used it again. This is a useful reminder of how warnings work: perl will check to see that the code *looks* structurally sound, but won't actually calculate runtime values or resolve variables. If it did, it would have picked up above on `${$second}` being resolved as `$first`.

> **Don't turn off these checks simply because they stop your program from running. You should always find a way to fix the program so as to satisfy them.**

use diagnostics

There's another pragma that may help you while debugging. `use diagnostics` will show you not only an error message or warning but also the explanatory text from the `perldiag` documentation page. For instance:

```
#!/usr/bin/perl
# diagtest.plx
use warnings;
use strict;
use diagnostics;

my $a, $b = 6;
$a = $b;
```

should give something like:

>perl **diagtest.plx**
Parentheses missing around "my" list at diagtest.plx line 7 (#1)

(W parenthesis) You said something like

my $foo, $bar = @_;

```
    when you meant

        my ($foo, $bar) = @_;

    Remember that "my", "our", and "local" bind tighter than comma.
>
```

This is helpful when you're debugging but remember that when `use diagnostics` is seen, the entirety of the `perldiag` page has to be read into memory, which takes up some time. It's a good idea to use it when writing a program and then remove it when you're done.

Alternatively, there's a standalone program called `splain`, which explains Perl's warnings and error messages in the same way. Simply collect up the output of your program and paste it to `splain`. If you're going to use a pipe, remember that warnings end up on standard error, so you'll have to say `perl myprogram.plx | 2>&1 | splain` to feed standard error there, too. Note that `splain` won't work on Windows.

Perl Command Line Switches

All of our programs so far have started with this line, the 'shebang' line:

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

and we've called our program by saying:

>perl program.plx

or possibly on UNIX with:

>./program.plx

The primary purpose of that first line is to tell UNIX what to do with the file. If we say `./program.plx`, this just says 'run the file `program.plx`', it's the 'shebang' line that says how it should be run. It should be passed to the file `/usr/bin/perl`, which is where the Perl interpreter will usually live.

However, that's not all it does, and it isn't just for UNIX: Perl reads this line itself and looks for any additional text, in the form of **switches**, which notify Perl of any special behavior it should turn on when processing the file. If we call the Perl interpreter directly on the command line, by saying `perl program.plx`, we can also specify some switches before Perl even starts looking at the file in question.

Switches all start with a minus sign and an alphanumeric character, and must be placed after `perl` but before the name of the program to be run. For instance, the switch `-w`, which is roughly equivalent to `use warnings;` can be specified in the file, like this:

```
#!/usr/bin/perl -w
# program.plx
...
```

or on the command line like this:

```
>perl -w program.plx
```

This allows us to change Perl's behavior either when writing the program or when running it. Some switches can only be used on the command line. By the time perl has opened and read the file, it may be too late to apply the behavior. This is most clearly illustrated in the case of the `-e` switch, which we'll be taking a look at next.

There are two major types of switch: those that take an argument and those that do not. `-w` does not take an argument, and neither does `-c`. (We'll see what `-c` does very soon.) If you want to specify both switches, you can either put them one after the other, `-w -c`, or combine them in a **cluster**, by saying `-wc`.

For switches that take an argument, such as `-i`, the argument must directly follow the switch. So, while you can combine `-w`, `-c`, and `-i00` as `-wci00`, you may not say `-i00wc`, as the `wc` will be interpreted as part of the argument to `-i`. You must either put switches that take an argument at the end of a cluster or separate them entirely.

-e

The most commonly used switch is `-e`. This may only be used on the command line, because it tells perl not to load and run a program file but to run the text following `-e` as a program. This allows you to write quick Perl programs on the command line. For example, the very first program we wrote can be run from the command line like this:

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

Notice that we surround the entire program in single quotes. This is because, as we saw when looking at `@ARGV`, the shell itself splits up the arguments on the command line into separate words. Without the quotes, our program would just be `print`, with `"Hello world\n"` as the first element of `@ARGV`.

There are two problems with this. First, we can't put single quotes inside our single quotes, and second, some operating systems' shells prefer you to use double rather than single quotes around your program. They then have differing degrees of difficulty coping with quotes in the program.

For instance, DOS, Windows and so on, will want to see this:

```
>perl -e "print \"Hello world\n\";"
```

You can get around most of this by judicious use of the `q//` and `qq//` operators. For instance, you could say `perl -e 'print qq/Hello world\n/;'`, which easily translates to a DOS-acceptable form as `perl -e "print qq/Hello world\n/;"`. Note that on UNIX systems, single quotes are usually preferable, as they prevent the shell interpolating your variables.

In the following examples, we'll be showing examples in single-quoted format. If you're using Windows, just convert them to double-quoted format as described above.

This technique is most commonly used for two purposes:

- To construct quick programs in conjunction with some of the other switches we'll see below
- To test out little code snippets and check how Perl works.

For example, if I wasn't sure whether an underscore would be matched by `\w` in a regular expression, I'd write something like this to check:

```
> perl -e 'print qq/Yes, it's included\n/ if q/_/ =~ /\w/;'
Yes, it's included
>
```

It's often quicker to do this than to go hunting through books and online documentation trying to look it up. As Larry Wall says, 'Perl programming is an empirical science'. You learn by doing it. If you're not sure about some element of Perl, get to a command line and try it out!

-n and -p

As mentioned above, you can combine `-e` with other switches to make useful programs on the command line. The most common switches used in this way are `-n` and `-p`. These are both concerned with reading `<ARGV>`. In fact, `-n` is equivalent to this:

```
while (<>) { "..your code here.." }
```

We can use this to produce programs for scanning through files, searching for matching lines, changing text, and so on. For example, here's a one-liner to print out the subject of any new items of mail I have, along with whom the mail is from:

Try It Out : New Mail Check

All the incoming mail arrives in a file called `Mailbox` on my computer. Each piece of mail contains a header, which contains information about it. For instance, here's part of the header from an email I sent to `perl5-porters`:

```
Date: Mon, 3 Apr 2000 14:22:03 +0900
From: Simon Cozens <simon@cozens.net>
To: perl5-porters@perl.org
Subject: [PATCH] t/lib/b.t
Message-ID: <20000403142203.A1437@SCOZENS>
Mime-Version: 1.0
Content-Type: text/plain; charset=us-ascii
X-Mailer: Mutt 1.0.1i
```

As you can see, each header line consists of some text, then a colon and a space, then some more text. If we extract the lines that start `Subject:` and `From:`, we can summarize the contents of the mailbox.

Here's how to do it on the command line:

```
>perl -ne 'print if /^(Subject|From): /' Mailbox
From: Simon Cozens <simon@brecon.co.uk>
Subject: [PATCH] t/lib/b.t
>
```

How It Works

To extract the relevant lines, we could write a program like this:

```
#!/usr/bin/perl
use warnings;
use strict;

open INPUT, "Mailbox" or die $!;
while (<INPUT>) {
   print if /^(Subject|From): /;
}
```

However, that's a lot of work for a little job, and Perl was invented to make this sort of thing easy. Instead we use the `-n` flag to give us a `while(<>)` loop and `-e` to provide the remaining line. Perl internally translates our one-line incantation to this:

```
LINE: while (defined($_ = <ARGV>)) {
    print $_ if /^(Subject|From): /;
}
```

As you may suspect, we're not confined to just printing text with these one-liners. Indeed, we can use this to modify parts of a file. Let's say we had an old letter file `newyear.txt` containing this text:

```
Thank you for your custom throughout the previous year. We
look forward to facing the challenges that 1999 will bring us,
and hope that we will continue to serve you this year as well.

All our best wishes for a happy and prosperous 1999!
```

We could use perl to print an updated version of it as follows:

```
>perl -ne 's/1999/2000/g; print' newyear.txt
Thank you for your custom throughout the previous year. We
look forward to facing the challenges that 2000 will bring us,
and hope that we will continue to serve you this year as well.

All our best wishes for a happy and prosperous 2000!
>
```

Of course, we're only printing the changed version to `STDOUT`. We could go the next logical step and use redirection to save this output to a file instead, as we saw in Chapter 6.

```
>perl -ne 's/1999/2000/g; print' newyear.txt >changedfile.txt
>
```

Since this is a pretty common operation – 'do something to the incoming data and print it out again' – perl lets use the `-p` flag instead of `-n` to automatically print out the line once we're finished. We can therefore save ourselves a valuable few keystrokes by saying this:

```
>perl -pe 's/1999/2000/g' newyear.txt
```

As you saw from the translation, these are ordinary loops, and we can use `next` and `last` on them as usual. To print out only those lines that don't start with a hash sign (#) we can say this:

```
>perl -ne 'next if /^#/; print' strictvar.plx
use warnings;
use strict;

my $a = 5;
$main::b = "OK";
our $c = 10;
$d = "BAD";
>
```

Note that we don't, and actually **can't** say:

```
>perl -pe 'next if /^#/'
```

This is because `-p` uses a special control structure, `continue`, translating internally to this:

```
LINE: while (defined($_ = <ARGV>)) {
    "Your code here";
} continue { print $_;}
```

Anything in a `continue { }` block will always get executed at the end of an iteration – even if `next` is used (although is still by-passed by `last`).

-c

`-c` stops perl from running your program – instead, all it does is check that the code can be compiled as valid Perl. It's a good way to quickly check that a program has no glaring syntax errors. It also loads up and checks any modules that the program uses, so you can use it to check that the program has everything it needs:

```
>perl –ce 'print "Hello, world\n";'
-e syntax OK
>perl -ce 'print ("Hello, world\n"));'
syntax error at -e line 1, near "))"
-e had compilation errors.
>
```

Be careful though, because this won't necessarily prove that your program will run properly – it checks that your program is grammatically correct, but not whether it makes sense. For instance, this looks fine:

```
>perl –ce 'if (1) { next }'
-e syntax OK
>
```

but if you try to run it normally, you'll get an error:

```
>perl -e 'if (1) { next }'
Can't "next" outside a loop block at -e line 1.
>
```

This is the difference between a compile-time and a runtime error. A compile-time error can be detected in advance and means that perl couldn't understand what you said. A runtime error means that what you said was comprehensible but (for whatever reason) can't be done.
`-c` only checks for compile-time errors.

-i

When we're searching and replacing the contents of a file, we usually don't want to produce a new, revised copy on standard output, but rather change the file as it stands. You might think of doing something like this:

```
>perl -pe 's/one/two/g' textfile.txt > textfile.txt
```

There's a problem with that though, as you'll know if you've tried it, it's quite possible that you'll completely lose the file. This is because (unless you're running in a shell that's smart enough to watch your back) the shell opens the file it's writing to first and **then** passes the filehandle to perl as standard output. Perl opens the file after this has taken place, but by this time, the original contents of the file have been wiped out.

To get around this yourself, you'd have to go through contortions like this:

```
>perl -pe 's/one/two/g' textfile.txt > textfile.new
>mv textfile.new textfile.txt
```

The UNIX command `mv` is the same as the `ren` command in Windows: Both commands are used to rename files.

Perl provides you with a way to avoid this. The `-i` switch opens a temporary file and automatically replaces the file to be edited with the temporary file after processing. You can do what we want just like this:

```
>perl -pi -e 's/one/two/g' textfile.txt
```

Well, you *might* be able to – as it stands, you may find that this just returns the message:

```
Can't do inplace edit without backup
```

This happens because perl doesn't know how you want to name the temporary file. Notice though, that I separated `-i` from the `-e` switch: this is because `-i` takes an optional argument. Anything immediately following the `-i` will be treated as an extension to be added to the original filename as a name for the backup file. So, for instance:

```
>perl -pi.old -e 's/one/two/g' textfile.txt
```

will take in a file, `textfile.txt`, save it away as `textfile.txt.old`, and then replace every instance of 'one' with 'two' in `textfile.txt`.

-M

If you need to load any modules from the command line, you can use the `-M` switch. For instance, to produce politically correct one-liners, we should really say something like this:

```
>perl -Mstrict -Mwarnings –e ...
```

However, the kind of code we're likely to put on the command line doesn't really need this sort of strictness. It's still useful to have the -M switch to load modules – the CPAN modules `LWP::Simple`, `Tk`, and `HTML::Parser` have been used in the past to create a one-line graphical web browser!

-s

As well as passing switches to perl, you may want your program to have switches of its own. The `-s` switch (usually specified on the shebang line) tells perl to interpret all command line switches following the filename as variables (for example: `$v`, `$h`) and removed from `@ARGV`. This means that you can process these switches in any way you want.

For instance, a lot of programs will display a help message explaining their usage, when called with the `-h` switch at the command line. Similarly, they'll give their version number if `-v` is used. Let's make some of our own programs do this:

Try It Out : Reading Command Line Options

We're going to add 'help' and 'version number' messages to `nl.plx`, the line numbering program we wrote in the last chapter. Note that this example uses `our` and will therefore only work for Perl versions 5.6 and above:

```
#!/usr/bin/perl -s
# nl3.plx
use warnings;
use strict;

my $lineno;
my $current = "";
our ($v,$h);

if (defined $v) {
   print "$0 - line numberer, version 3\n";
   exit;
}
if (defined $h) {
   print <<EOF;
$0 - Number lines in a file

Usage : $0 [-h|-v] [filename filename...]

This utility prints out each line in a file to standard output,
with line numbers added.
EOF
   exit;
}

while (<>) {
    if ($current ne $ARGV) {
        $current = $ARGV;
        print "\n\t\tFile: $ARGV\n\n";
        $lineno=1;
    }
    print $lineno++;
    print ": $_";
}
```

If we now pass the `-h` option, it's not treated as a filename, but rather as a request for help:

```
>perl –s nl3.plx -h
nl3.plx  Number lines in a file

Usage : nl3.plx [-h|-v] [filename filename...]

This utility prints out each line in a file to standard output,
with line numbers added.
>
```

If you're fortunate enough to be using an operating system that allows you to use Perl programs as executables, the shebang line will take care of specifying the `–s` switch, so you won't need to repeat it on the command line. So while this will work fine on UNIX:

```
>nl3.plx -v
nl3.plx - line numberer, version 3
```

you'll probably need to say this on Windows:

```
>perl –s nl3.plx –v
nl3.plx - line numberer, version 3
```

How It Works

The `-s` on the shebang or command line tells perl that any switches following the name of the Perl program will cause a Perl variable of the same name to be defined. For instance, this command line:

```
>perl -s something -v -abc
```

will set the variables `$v` and `$abc`. We therefore need to be ready to receive these variables, otherwise we will fall foul of `use strict`, so we put:

```
our ($v,$h);
```

If the variable is defined, we do something with it:

```
if (defined $v) {
   print "$0 - line numberer, version 3\n";
   exit;
}
```

The special variable `$0` contains the name of the program currently being run. It's good form to put this in any informational messages you produce about the program.

While `-s` is handy for quick tasks, there are two things that make it unsuitable for use in big programs:

- You have no control over what switches should be recognized. Perl will set any variable, regardless of whether you want it to or not. If you're not actually using that switch, this will generate warnings. For instance:

  ```
  >perl -s nl3.plx -v -foobar
  Name "main::foobar" used only once: possible typo.
  nl3.plx - line numberer, version 3
  >
  ```

- `-abc` is treated as one switch and sets `$abc`, rather than the three switches `-a`, `-b`, and `-c`.

For this reason, it's recommended that you use the standard modules `Getopt::Std` or `Getopt::Long` instead. Appendix D gives a brief rundown of all perl's standard modules or for more detailed information, refer to the `perlmod` manpage.

-I and @INC

Perl knows where to look for modules and Perl files included via `do` or `require` by looking for them in the locations given by a special variable, the array `@INC`. You can add directories to this search path on the command line, by using the -I option:

```
perl –I/private/perl program
```

will cause Perl to look in the directory `/private/perl` for any files it needs to find besides those in `@INC`. For more details on working with `@INC` up close, just have a look in the next chapter.

-a and -F

One of perl's ancestors is the UNIX utility `awk`. The great thing about `awk` was that when reading data in a tabular format, it could automatically split each column into a separate variable for you. The perl equivalent would use an array and would look something like this:

```
while (<>) {
   my @array = split;
   ...
}
```

The `-a` switch, used with `-n` and `-p`, does this kind of `split` for you. It splits to the array `@F`, so

>perl -an '...'

is equivalent to:

```
LINE: while (defined($_ = <ARGV>)) {
    @F = split;
    'Your code here';
}
```

So, to get the first word of every line in a file, you could say this:

>perl -ane 'print $F[0],"\n"' chapter9.txt
Running

By

You

Everyone
…
>

By default, `-a` splits on spaces – although you can change it by specifying another switch, `-F`, which will take your chosen delimiter as an argument. For instance, the fields in a UNIX password file are (as we saw in Chapter 5), delimited by colons. We can extract the home directory from the file by looking at the fifth element of the array. If our `passwd` file contains the line:

```
simon:x:10018:10020::/home/simon:/bin/bash
```

We'll get the following result:

```
>perl -F: -ane 'print $F[5],"\n" if /^simon/' passwd
/home/simon
>
```

-l and –0

It's rather annoying to have to specify `"\n"` on the end of everything we `print`, just to get a new line, especially if we're doing things on the command line. The `-l` switch sets the *output* record separator `$\` equal to the current value of the *input* record separator `$/`. The former is added on automatically at the end of every `print` statement. Since the latter is usually `\n`, the newline character, `-l` adds a newline to everything we print. Additionally, if used with the `-n` or `-p` switches, it will automatically `chomp` any input.

We can cut the above program down by a few more keystrokes like this:

```
>perl -F: -lane 'print $F[5] if /^simon/' passwd
/home/simon
>
```

If `-l` is followed by a valid octal number, then the character with that ASCII value (see Appendix F) is used as the output record separator instead of new line. However, this is relatively rare.

Alternatively, you can set the input record separator using the `-0` switch. Likewise, if this is followed by an octal number, `$/` will be set to that character. For instance, `-0100` will effectively execute `$/="A";` at the beginning of the program. `-0` on its own or followed by something that isn't an octal number will cause `$/` to be set to the `undefined` value, causing the entire file to be read in at once:

> **While you can conceivably use `-l` on the shebang line to save printing newlines in your program, it's actually a bad idea – many people will probably miss it and wonder where all the new lines are coming from. It will also get you into trouble if you want a `print` statement that doesn't cause a newline.**

-T

When you're dealing with data that's being downloaded from an unreliable source from the outside world, you'll probably want to be careful what you do with it. If you're asking the user for a filename, which you pass directly to `open`, you're potentially allowing the user to do all kinds of horrible things. Say, for instance, you were given the filename `rm -rf /|` and used it as it was (**DON'T!**). You may well find afterwards that several of the files on your disk had disappeared...

To force you to clean up this insecure data, Perl has a switch, `-T`, that turns on 'taint mode'. When this switch is in operation, any data coming into your program is tainted, and may not be used for any operations Perl deems 'unsafe' for example, passing to `open`. Furthermore, any data derived from tainted data becomes tainted itself. The **only** way to untaint data is to take a regular expression backreference:

```
$tainted =~ /([\w.]+)/;
$untainted = $1;
```

We'll look at this in a lot more detail in the section on taint checking in Chapter 13.

Debugging Techniques

Earlier in the chapter, we looked at bugs that perl can trap easily – bugs that turn up when what you write doesn't make sense. However, a lot of the time you'll write something that makes sense but doesn't do what you want it to do. While there's no magic formula to find the problem for you here, there are several techniques you can use to track down the problem. Perl itself includes a debugging environment to help you in your investigations.

Before the Debugger...

Before I explain how the debugger works, though, I have to admit that I'm an old-fashioned soul, and don't really believe in debuggers. People seem to see the debugger as a substitute for understanding the problem – just run the program through the debugger, and it'll magically uncover the error. While that would be lovely, it's not actually the case. The debugger can only help you along the way, and there are other ways of debugging a program that may well be far more effective than firing it up.

Debugging Prints

It's an old programming proverb: When in doubt, print it out. Are you sure that the data coming into your program is what you think it is? Print it out! Do you know that a regular expression has done what you think it should have to a variable? Print it out, before and after. Do you know how many times Perl has gone through a certain loop or section of code? Is Perl taking far longer than it should with something trivial? Print out a little message saying where you are in the code. `print` is by far the most powerful and useful debugging tool at your disposal!

Pare It Down

If you're not sure where an error is occurring, try and isolate it. Cut or comment out unrelated lines and see if the problem still occurs. Keep commenting lines out until the problem goes away, and then look at what you've changed.

The same technique can be used when you've got inexplicable behavior. It's a lot easier to spot a bug when the odd behavior is demonstrated in five lines than in fifty. Alternatively, if you can't reproduce the problem that way, start a new program that *just* has the troublesome logic in it and see if you can find anything odd about that. This will also test whether there's something wrong with the data you're feeding into your program.

In any case, the smaller you can make your demonstration code, the better – especially if you're planning on asking someone else about it. The smaller your haystack, the more chance you have of finding a needle in it. Furthermore, plenty of people may be willing to help you if you can produce two lines that demonstrate a problem – fewer will if they think you expect them to debug your entire program.

Here are a some other problems that can cause weirdness without actually causing an error:

Context

Is there a problem with context? Always make sure you know what you're expecting from a function – whether you want an array or a scalar – and ensure that you're collecting the result in the appropriate type of variable.

Scope

Has a variable you've declared with `my` gone out of scope and become undefined or returned to its original value (from before you `my`'d it)? Remember that declaring variables `my` inside a block or loop means you won't be able to get at their value outside of it.

Precedence

Are you saying something like `print (2+3)*5`? This would add two to three, print it, and multiply the result of the `print` by five. Have you forgotten brackets around a list? Having `warnings` turned on will help you pick up on most of these sorts of thing, but be careful. Whenever in doubt, bracket more than you need to.

Using the Debugger

The Perl debugger isn't a separate program, but a special mode under which `perl` runs – to enable it, you simply pass the `-d` switch. As it's a special mode for running your program, you won't get anywhere unless your program compiles correctly first. What the debugger will help you do is to trace the flow of control through your program and allow you to look at variables' values at various stages of operation.

When you start your program in the debugger, you should see something like this:

```
>perl –d nl3.plx
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or `h h' for help, or `perldoc perldebug' for more help.

main::(nl3.plx:6):      my $lineno;
  DB<1>
```

That line '`DB<1>`' is the debugger prompt. Here is a partial list of things you can do at that point:

Command	Description
`T`	Obtain a 'call trace' of all the subroutines perl is currently processing. This will tell you how you got to be where you are.
`s`	Step to the next line as you go one line at a time through your program.

Table continued on following page

Command	Description
`n`	Step over a subroutine. Call the subroutine reference on the current line, and stop again once control has returned from that.
`Return`	Repeat the last stepping command.
`r`	Keep going until the current subroutine returns.
`c`	Continue – keep going until something happens that causes the debugger to stop again.
`l`	List the next few lines to be processed.
`-`	List the previous lines processed.
`w`	List the lines around the current line.
`/pattern/`	Search forwards in the program code until the pattern matches.
`t`	Turn on (or off) trace mode. This prints every statement before executing it.
`b`	Set a breakpoint. Stop running the program and return to the debugger at the given line number or when the given condition is true.
`x`	Evaluate something in array context and give a tree view of the resulting data structure.
`!`	Do the previous command again.
`p`	Print something out.
`h`	Get more help.

We're not going to look any further at the debugger. While it can help you out – and once you start really developing in Perl it really will – for the time being it's a better learning experience to try and debug your code using the hints and techniques shown in the rest of this chapter. That way you can really get to know how Perl thinks and works.

Defensive Programming

Far and away the best way to debug your code is to try and make sure you never have to. While it's impossible to guarantee that there will never be any bugs in your program, there are a lot of things you can do to minimize their number, and to make sure that any potential bugs are easy to locate. In a sense, it's all about expecting the worst.

Strategy

Before writing another line, make sure you've got a plan. You need to use just as methodical an approach to debug code efficiently as you do to write it in the first place. Keep the following points in mind:

- Never try to write a large program without trying parts of that program first. Break the task down into small units, which can be tested as they're written.
- Track down the first bug, then try the program again – the second may just have been a consequence of the first one.

- Likewise, look out for additional errors after you've 'fixed' the first bug. There could have been a knock-on effect revealing subsequent errors.

Check Your Return Values

There's no excuse for not checking the return values on any operator that gives a meaningful return value. Any operator that interacts with the system will return something by which you can determine whether it succeeded or not, so make use of it. Furthermore, you can always attempt to pre-empt problems by looking to see what could go wrong. Chapter 6 contained an example of defensive programming, when we tested whether files were readable and writeable.

Be Prepared for the Impossible

Sometimes, things don't go the way you think they should. Data can get shuffled or wiped out by pieces of code in ways that you can't explain. In order to pick this up as soon as possible after it happens, test to see if the impossible has occurred. If you know a number's going to be 1, 2, or 3, do something like this:

```
if ($var == 1) {
    # Do first thing.
} elsif ($var == 2) {
    # Do second thing.
} elsif ($var == 3) {
    # Do third thing.
} else {
    die "Whoa! This can't happen!\n";
}
```

With luck, you'll never get there, but if you do, you'll be alerted to the fact that something higher up in the program has wiped out the variable. These are a type of trap called **assertions** or, less formally, '**can't happen' errors**. Eric Raymond, author of 'The Jargon File', says this about them:

> *"Although 'can't happen' events are genuinely infrequent in production code, programmers wise enough to check for them habitually are often surprised at how frequently they are triggered during development and how many headaches checking for them turns out to head off."*

Never Trust the User

Users are an extremely reliable source of bad data. Don't let bad data be the cause of bugs. Check to ensure you're getting the sort of data you want. Do you want to take the newline character off the end? Are you expecting to be upper case, lower case, mixed case, or don't you care? Try and be flexible wherever possible, since the user is more than likely to get something wrong. Above all, make sure you're completely happy with input before acting on it.

Definedness and Existence

If you're putting elements into arrays or hashes, should they already exist? Should they not exist? Check that you're not wiping data you want to keep, and if you are, ask yourself how you got into that situation. Are you sure you've got some data to put in? Check that you're putting the right sort of data into the right place. Are you sure there's something there when you take data out? Make sure the data exists when you've accessed a hash or array.

Have Truthful, Helpful Comments

Comments are a useful memory aid to help you keep track of what's going on in the program, so try and use them as intended. Comments that explain data flow – what the data means and where it comes from – are more helpful than comments that explain what you're doing. Contrast the usefulness of these two sections:

```
$a = 6.28318; # Assign 6.28318 to $a


$a = 6.28318; # pi*2
```

The problem with comments is that you have to keep them up to date when you change the code. Make sure your comments aren't a distraction (at best) or (at worst) downright misleading. There's an old saying: 'If code and comments disagree, both are probably wrong.'

Keep the Code Clean

Tidy code is much easier to understand and debug than messy code. It's easier to find problems if, among other things, you make sure to always:

- keep parallel items aligned together in columns
- keep indentation regular
- keep to one statement per line
- split long statements over multiple lines
- use white-space characters to increase readability

Again, contrast these two snippets. There's this:

```
while (<>) {
   if ( /^From:\s+(.*)/    ) { $from    = $1 }
   if ( /^Subject:\s+(.*)/ ) { $subject = $1 }
   if ( /^Date:\s+(.*)/    ) { $date    = $1 }


   print "Mail from $from on $date concerning $subject\n"
      unless /\S+/;

   next until /^From/;
}
```

versus this:

```
while(<>){if(/^From:\s+(.*)/){$from=$1}
if(/^Subject:\s+(.*)/){$subject=$1}
if(/^Date:\s+(.*)/){$date=$1}
print "Mail from $from on $date concerning $subject\n" unless /\S+/;
next until /^From/;}
```

Which one would *you* rather debug?

Summary

Whenever you program, you'll inevitably make mistakes and create bugs. There are two types of bug you'll come up against: the syntax error, which manifests itself with a violent bang, and the logic error, which hides away insidiously inside your program and drives you silently mad. This chapter has shown you how you can deal with both sorts of bug.

We've looked at Perl's error messages and the most common causes of syntax errors. We've seen how to decode the error messages perl gives, both by employing a little bit of logical thought (the best way) and by getting the `diagnostics` pragma to explain it to us (the easiest way).

We've also seen how to avoid creating bugs in the first place – `use warnings` and `use strict` act as checks to ensure that we're not doing anything too crazy. There are also plenty of ways to use defensive programming, imposing further checks to stop bugs before they happen.

Perl is a great tool for use on the command line. I'm forever using it to search files for patterns and change files with a search-and-replace, as well as using it to test out snippets of Perl code and examine Perl's behavior. We've looked at various command line switches, which make it easy for us to do complex things: loop over a file, change a file in place, check the syntax of a file or piece of code, and so on.

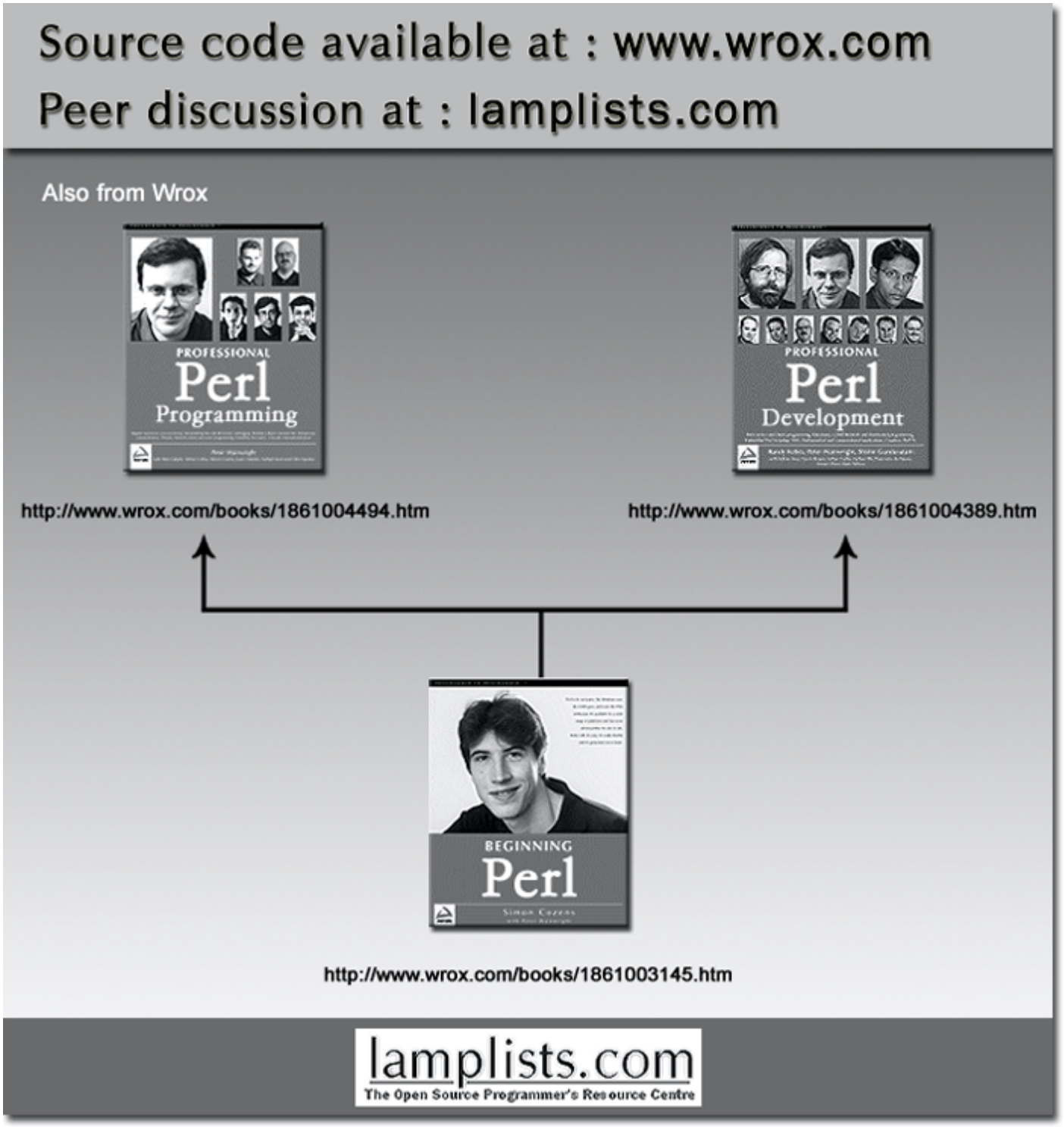
We've also doffed our cap to the Perl debugger, as well as some other ways to detect and remove bugs in our code. Now you're armed to do battle with any bugs that come your way – and come they will!

Exercises

Take a look at the following file, apply what you've read about, and see if you can knock it into shape:

```
#!/usr/bin/perl
#buggy.plx

my %hash;

until (/^q/i) {

print "What would you like to do? ('o' for options): "
$  = STDIN;

if ($  eq "o"){options}elsif($  eq "r"){read}elsif($  eq "l"){ list }elsif
($  eq "w"){ write }elsif ($  eq "d") { delete } elsif ($  eq "x") { clear }
else { print "Sorry, not a recognized option.\n"; }

sub options {
   print<<E0F
      Options available:
      o - view options
      r - read entry
      l - list all entries
      w - write entry
```

```
      d - delete entry
      x - delete all entries
EOF;
}

sub read {
my $keyname = getkey();

if (exists $hash{"$keyname"}) {
print "Element '$keyname' has value $hash{$keyname}";
} else {
print "Sorry, this element does not exist.\n"}}

sub list {foreach (sort keys(%hash)) {print "$  => $hash{$ }\n";}}

sub write {
my $keyname = getkey();
my $keyval = getval();
if (exists $hash{$keyname}) {print "Sorry, this element already exists.\n"
} else {$hash{$keyname}=$keyval;}}

sub delete {
my $keyname = getkey();
if (exists $hash{$keyname}) {
print "This will delete the entry $keyname.\n";
delete $hash{$keyname};}}

sub clear {undef %hash;}

sub getkey {print "Enter key name of element: "; chomp($  = <STDIN>);)

sub getval {print "Enter value of element: "; chomp($_ = <STDIN>);}
```



10

Modules

In Chapter 8, we divided our programs up into subroutines: functional units that help us organize our program. In this chapter, we'll look at modules, which are the next stage of division.

Very simply, a module is a package within a file. It's a collection of subroutines and variables, all belonging to the same package and stored away in its own file. While subroutines allow us to bundle up individual tasks, modules are more about bundling up an entire area of activity.

Perl comes with quite a large library of modules, which means there are quite a few tasks already coded. That's great for us, since it means we don't have to program them again – we'll be taking a look at some of the more useful ones here. Modules were invented to help code reuse, and recycling your code is an excellent principle, and a good habit to get into.

In fact, reusing ready-made code is such a good idea that there's a whole archive of modules out there, each providing us with a set of subroutines – a set of tools – to work in a different area. Most of the publicly available Perl modules can to be found on CPAN, the 'Comprehensive Perl Archive Network'. Later in the chapter we'll see how to find what we want on CPAN, and how to install modules from there. We'll also have a look at how to use some of the major modules that reside there.

Types of Module

Despite the fact that the modules you'll find out there can be on any subject under the sun, there are just a few standard ways in which they tend to be categorized. In fact, the CPAN Modules List classifies modules by subject area, development stage, where the support comes from, language used, and interface style. We'll use a slightly simpler classification here:

- Pragmatic modules – as we saw in Chapter 9, these alter the way Perl does certain things. Usually, they pass special information to the perl interpreter, which perl then uses internally.
- Standard modules – these make up the majority of modules out there and are purely Perl code. On the whole, they do things in a pretty standard way, which we'll examine later.
- Extension modules combine Perl with C (or other languages) to interface with either the operating system or third-party software.

There's a full list of both the pragmatic and standard modules installed with Perl in Appendix D.

Why Do I Need Them?

Why should you use modules? The simple answer is that it saves time. If you need that program written yesterday, it's exceptionally handy to be able to pull down a bunch of modules that you know will do the job and then simply glue them together. There's not much creativity involved, and you don't learn a great deal doing it that way – but in some cases, there's just not the time for creativity or learning.

The second answer is because programmers are lazy and don't like reinventing the wheel. Now, don't get me wrong – there's good laziness and there's bad laziness. Bad laziness says 'I should get someone else to do this for me', whereas good laziness says 'Maybe someone's already done this.' The good kind pays off. Most of the programming you'll be doing, at some level, has been done before.

There's also the fact that, as we've seen in several cases already, some of the things we want to do are far from straightforward. Unless we really know what we're doing, we run the risk of making incorrect assumptions or overlooking details.

Modules that have been kicking around on CPAN for a while will have been used by thousands of individuals, many of whom will have spent time fixing bugs and returning the results to the maintainer. Most of the borderline cases will have been worked out by now, and you can be pretty confident that the modules will do things correctly. When it comes to parsing HTML or reading CGI forms, I'm perfectly willing to admit that the people who wrote `HTML::Parser` and the `CGI` modules have done more work on the subject that I have – so I use their code, instead of trying to work out my own.

In short: don't reinvent the wheel – use modules.

Including Other Files

A module, as we've mentioned, is just a package stored in a file. We want to get perl to read that file and use it as part of our own program. We have three ways of doing this: `do`, `require` and `use`.

do

This is the most difficult of the three to understand; the others are just slightly varied forms of `do`.

`do` will look for a file by searching the `@INC` path (more on that later). If the file can't be found, it'll silently move on. If it is found, it will run the file just as if it was placed in a block within our main program – but with one slight difference: we won't be able to see lexical variables from the main program once we're inside the additional code. So if we have a file `dothis.plx`:

```
#!/usr/bin/perl
# dothis.plx
use warnings;
use strict;

my $a = "Been there, done that, got the T-shirt";
do "printit.plx";
```

and a file `printit.plx`:

```
print $a;
```

we'll get no output, not even a warning that `$a` is uninitialized within `printit.plx`, because we didn't turn on `warnings` in our included file. On the other hand, we can have subroutines in our included file and call them from the main file.

require

`require` is like `do`, but it'll only `do` once. It'll record the fact that a file has been loaded and will ignore further requests to `require` it again. It also fails with an error if it can't find the file you're loading:

```
#!/usr/bin/perl
# cantload.plx
use warnings;
use strict;

require "nothere.plx";
```

will die with an error like this:

```
>perl cantload.plx
Can't locate nothere.plx in @INC (@INC contains: /usr/local/lib/perl5/5.6.0/cygwin
/usr/local/lib/perl5/5.6.0 /usr/local/lib/perl5/site_perl/5.6.0/cygwin /usr/local/lib/perl5/site_perl/5.6.0
/usr/local/lib/perl5/site_perl .) at cantload.plx line 6.
>
```

This is the `@INC` array, which contains a list of paths where Perl looks for modules and other additional files. The first two paths are where Perl keeps the standard library. The first includes the word `cygwin`, which is the operating system I'm running on and contains the parts of the library specific to this operating system. The second is the part of the standard library, which does not depend on the operating system. In Windows, these two libraries are `C:/Perl/lib` and `C:/Perl/site/lib` by default.

The next two paths are the local 'site' modules, which are third-party modules that we'll install from CPAN or create ourselves. The version number (5.6.0) reminds us that these are modules specific to that version. The next path doesn't have a Perl version number in it, and that's for site modules that do not need a particular version of Perl. Finally, the `.` represents the current directory.

You can also use `require` like this:

```
require Wibble;
```

Using a bareword tells perl to look for a file called `Wibble.pm` in the `@INC` path. It also converts any instance of `::` into a directory separator. For instance, then:

```
require Monty::Python;
```

will send perl looking for `Python.pm` in a directory called `Monty` which is itself in one of the directories given in `@INC`.

use

The way we normally use modules is, logically enough, with the `use` statement. This is like `require`, except that perl applies it *before* anything else in the program starts. If Perl sees a `use` statement *anywhere* in your program, it'll include that module. So, for instance, you can't say this:

```
if ($graphical) {
   use MyProgram::Graphical;
} else {
   use MyProgram::Text;
}
```

because when perl's reading your program, it will include *both* modules – the `use` takes place way before the value of `$graphical` is decided. We say that `use` takes place at compile time and not at run time.

Changing @INC

The default contents of the search path `@INC` are decided when perl is compiled – if we move those directories elsewhere, we'll have to recompile perl to get it working again. However, we can tell it to search in directories other than these. `@INC` is an ordinary array, so you might expect us to be able to say:

```
push @INC, "my/module/directory";
use Wibble;
```

However, this isn't going to work. Why not? Well, remember that the statement above will execute at run time. Unfortunately the `use` statement takes place at compile time, well before that. No problem! There's a special subroutine called `BEGIN`, which is guaranteed execution at compile time, so we can put it there:

```
sub BEGIN {
   push @INC, "my/module/directory";
}
use Wibble;
```

Now that'll work just fine. However, it's a little messy, and what's more, there's an easier way to do it. We can use the `lib` pragma to add our directory to `@INC` before anything else gets a chance to look at it:

```
use lib "my/module/directory";
use Wibble;
```

Package Hierarchies

We've already seen how packages can help us break up a namespace: `$Fred::name` isn't the same variable as `$Barney::name`. When modules come into play, packages are used to identify the module. Now our variables have a nice namespace, but our modules have to identify themselves by a single word. With several thousand modules out there, it gets hard to find the one we want. So the librarians at CPAN have come up with a solution: we split up the module package names into hierarchies. Instead of having tens of modules about sorting, we now have `Sort::Fields`, `Sort::Versions`, and so on.

This hierarchy is only a naming scheme. It doesn't mean that `Sort::Fields` and `Sort::Versions` are somehow related to a bigger package called `Sort` – it's simply a way of making it easier to categorize modules.

So how do we store these in files? Some operating systems won't let us have colons inside file names, so `Sort::Versions.pm` won't be legal. However, since these names represent a consistent hierarchy, there's a natural way we can organize them on the disk: as mentioned above, `require` and `use` translate colons into directory separators, so `Sort::Versions` will actually be stored in a file called `Versions.pm` in a directory called `Sort` somewhere off one of the site paths.

Exporters

Since modules are usually packages stored in a file, a subroutine in the `Text::Wrap` module, for example, would normally be tucked away in the `Text::Wrap` package. However, let's say it would be more convenient for us to have this as a subroutine in the package we're currently in – usually the `main` package. To do this, perl uses a module called `Exporter`, which provides it with a way of **importing** subroutines from the module into the caller's package. Here's how it works:

When you `use` a module, as well as reading and executing the code, perl will try and run a subroutine called `import` inside the module's package. If that's not found, nothing happens, and there's no error. If it is found, though, it's called with all the parameters given on the `use` line. So, for instance:

```
use Wibble ("wobble", "bounce", "boing");
```

loads the `Wibble` module and then runs:

```
Wibble::import("wobble", "bounce", "boing");
```

Theoretically, this `import` subroutine could do anything. In fact, a few modules use it to let you pass parameters to setup the module. However, you'll usually want to use it to import subroutines and variables.

`Exporter` lets the modules that use it borrow a standard `import` subroutine. This subroutine checks a number of variables inside the module as well as the parameters that we give it. If we give an empty list, like this:

```
use Wibble ();
```

then nothing will be imported. If there's a particular subroutine we want to use – `wobble()` for example – then we could call it as `Wibble::wobble()`, and we'll get it imported into our current package. We can only import subroutines that the module is prepared to export, and it'll detail those in a package variable called `@EXPORT_OK`. So if, for instance, I wanted a to make a `Wibble` module from which we could import `wobble()`, `bounce()` and `boing()`, I'd say this:

```
package Wibble;
use warnings;
use strict;
```

```
use Exporter;
our @ISA = qw(Exporter);
our @EXPORT_OK = qw(wobble bounce boing);

sub wobble { print "wobble\n" }
sub bounce { warn  "bounce\n" }
sub boing  { die   "boing!\n" }
```

If we don't pass any parameters at all, we get the default subroutines, which are defined in `@EXPORT`. So if our module looked like this:

```
package Wibble;
use warnings;
use strict;

use Exporter;
our @ISA = qw(Exporter);
our @EXPORT_OK = qw(wobble bounce boing);
our @EXPORT    = qw(bounce);

sub wobble { print "wobble\n" }
sub bounce { warn  "bounce\n" }
sub boing  { die   "boing!\n" }
```

and we ran `use Wibble;` in our main program, we'd be able to call `bounce()` from the main program, but not `wobble()` or `boing()` – we would have to call these as `Wibble::wobble()` and `Wibble::boing()`.

We can also define `tags` with the `%EXPORT_TAGS` hash. This allows us to group together a bunch of subroutines or variables under a group name. For instance, the `CGI` module (which we'll be using in Chapter 12) allows us to say:

```
use CGI qw(:standard);
```

which will import all its most useful subroutines.

The Perl Standard Modules

As we've mentioned, Perl comes with a number of modules included. Some of these (such as `Socket`) are system specific and generally used by higher-level modules – some however, are useful on their own. You can find a list of all the standard modules in Appendix D. We'll take a quick look at some of the more useful and interesting ones here.

File::Find

We looked briefly at `File::Find` when we examined callbacks – we'll see more of these in the final chapter. This is a module for traversing directory trees, visiting each file in turn and running a subroutine (the callback) on them. We have two subroutines, `find` and `finddepth`. The former does a depth-first search (see Chapter 6), visiting directories only after their files have been processed. This is useful if, for example, you want to delete entire directory trees, since you're not usually permitted to delete a directory until you've deleted all the files in it.

Why shouldn't you do this yourself? One of the problems is symbolic links: some operating systems have the ability to point one directory into another, which can create loops in the file system, in which you'll get stuck. The main reason, though, is that it involves a lot of work – work that someone else has done already.

We call the subroutines with two parameters: the callback subroutine reference, and the directory (or a list of directories) to start from:

```
find(\&wanted, "/home/simon/");
```

The subroutine works under the following conditions:

- You are moved into the same directory as the file under consideration.
- The current directory, relative to the top of the tree is held in `$File::Find::dir`.
- `$_` contains the name of the current file.
- `$File::Find::name` is the name including the directory.

With that, we can do anything. Do you remember, way back in Chapter 1, we wanted a program that would delete useless files? Here it is:

```
#!/usr/bin/perl
# hoover.plx
use strict;
use warnings;

use File::Find;
find(\&cleanup, "/");

sub cleanup {
   # Not been accessed in six months?
   if (-A > 180) {
      print "Deleting old file $_\n";
      unlink $_ or print "oops, couldn't delete $_: $!\n";
      return;
   }
   open (FH, $_) or die "Couldn't open $_: $!\n";
   for (1..5) { # You've got five chances.
      my $line = <FH>;
      if ($line =~ /Perl|Simon|important/i) {
         # Spare it.
         return;
      }
   }
   print "Deleting unimportant file $_\n";
   unlink $_ or print "oops, couldn't delete $_: $!\n";
}
```

You can of course alter this so it doesn't look for the words 'Perl', 'Simon' or 'important' in their first five lines and indeed so it doesn't look through and delete files from your entire directory structure.

Getopt::Std

We saw in Chapter 9 how the `-s` flag gave us a rudimentary way to get perl to pass command line options to our program – it will take flags from the command line and let us access them as Perl variables with the same name (for example, `-h` becomes `$h`). However, it had a few limitations:

- We couldn't use `-abc` to mean the `a` flag, the `b` flag and the `c` flag.
- We couldn't give values to flags.
- We had to have all flags as global variables.

The `Getopt::Long` and `Getopt::Std` modules get us round all these problems and provide us with more flexibility besides. `Getopt::Std` is the simpler of the two, providing us with a way to get single-letter switches with values and support for clustered flags. We can also arrange to have the flags placed in a hash. For instance, to provide our wonderful 'Hello World' program (from Chapter 1) with help, a version identifier and (heavens above!) internationalization, we could do this:

```
#!/usr/bin/perl
# hello3.plx
# Hello World (Deluxe)
use warnings;
use strict;

use Getopt::Std;
my %options;
getopts("vhl:",\%options);

if ($options{v}) {
   print "Hello World, version 3.\n";
   exit;
} elsif ($options{h}) {
   print <<EOF;

$0: Typical Hello World program

Syntax: $0 [-h|-v|-l <language>]

   -h : This help message
   -v : Print version on standard output and exit
   -l : Turn on international language support.
EOF
   exit;
} elsif ($options{l}) {
   if ($options{l} eq "french") {
      print "Bonjour, tout le monde.\n";
   } else {
      die "$0: unsupported language\n";
   }
} else {
   print "Hello, world.\n";
}
```

`getopts` takes the following as its arguments: a specification, the letters we want to know about, and a hash reference. If we follow a letter with a colon, we expect that a value will be stored in the hash. If we don't use a colon, then the hash value stored is just true or false depending on whether or not the option was given. We can now get output like this:

```
>perl hello3.plx -l french
Bonjour, tout le monde.
>
```

`Getopt::Std` also produces a warning if it sees options it's not prepared for:

```
>perl hello3.plx -f
Unknown option: f
Hello, world.
>
```

Getopt::Long

The Free Software Foundation, when they were developing the GNU project, decided that single-letter flags weren't friendly enough, so they invented 'long' flags. These use a double minus sign followed by a word. To give a value, you'd say something like `--language=french`.

The module `Getopt::Long` handles this style of options. Its documentation is extremely informative, but it's still useful to see an example. Let's convert the above program to GNU options:

```
#!/usr/bin/perl
# hellolong.plx
# Hello World (Deluxe) - with long flags
use warnings;
use strict;

use Getopt::Long;
my %options;
GetOptions(\%options, "language:s", "help", "version");

if ($options{version}) {
   print "Hello World, version 3.\n";
   exit;
} elsif ($options{help}) {
   print <<EOF;

$0: Typical Hello World program

Syntax: $0 [--help|--version|--language=<language>]

   --help     : This help message
   --version  : Print version on standard output and exit
   --language : Turn on international language support.
EOF
   exit;
} elsif ($options{language}) {
   if ($options{language} eq "french") {
      print "Bonjour, tout le monde.\n";
   } else {
      die "$0: unsupported language\n";
   }
} else {
   print "Hello, world.\n";
}
```

We can still use the previous syntax, but now we can also say:

```
>perl hellolong.plx --language=french
Bonjour, tout le monde.
>
```

File::Spec

If we want to write really portable programs in Perl, we have to be careful when doing things like dealing with file names. `File::Spec` is a module for handling, constructing and breaking apart file names. It's actually installed as an alias to another module: `File::Spec::Unix`, `File::Spec::Win32`, `File::Spec::VMS`, or whatever's relevant to the local system.

Normally it has an object-oriented interface, but it's much easier to use the subroutine interface, `File::Spec::Functions`. Here are some of the subroutines it provides:

Function and Syntax	Description
`canonpath (`*$path*`)`	Cleans up *$path* to its simplest form.
`catdir(`*$directory1, $directory2*`)`	Concatenates the two directories together to form a new path to a directory, ensuring an appropriate separator in the middle and removing the separator from the end.
`catfile(`*$directory, $file*`)`	Like `catdir`, but the path will end with a file name.
`tmpdir()`	Finds a writeable directory for temporary files (see the `File::Temp` module before working with temporary files!).
`splitpath($path)`	Splits up a path into volume (drive on Windows, nothing on UNIX), directories and filename.
`splitdir($path)`	Splits a path into its constituent directories: the opposite of `catdir`.
`path()`	Returns the search path for executable files.

So to find out if there's a copy of the `dir` program on this computer, I might do this:

```
#!/usr/bin/perl
# whereisit.plx
use warnings;
use strict;

use File::Spec::Functions;
foreach (path()) {
   my $test = catfile($_,"dir");
   print "Yes, dir is in the $_ directory.\n";
   exit;
}
print "dir was not found here.\n";
```

Benchmark

There's More Than One Way To Do It – that's our motto. However, some ways are always going to be faster than others. How can you tell though? You could analyze each of the statements for efficiency, or you could simply roll your sleeves up and try it out.

Our next module is for testing and timing code. `Benchmark` exports two subroutines: `timethis` and `timethese`, the first of which, `timethis`, is quite easy to use:

```
#!/usr/bin/perl
# benchtest.plx
use warnings;
use strict;

use Benchmark;
my $howmany = 10000;
my $what    = q/my $j=1; for (1..100) {$j*=$_}/;

timethis($howmany, $what);
```

So, we give it some code and a set number of times to run it. Make sure the code is in single quotes so that Perl doesn't attempt to interpolate it. You should, after a little while, see some numbers. These will, of course, vary depending on the speed of your CPU and how busy your computer is, but mine says this:

```
>perl benchtest.plx
timethis 10000:  3 wallclock secs ( 2.58 usr +  0.00 sys =  2.58 CPU) @ 3871.47/s (n=10000)
>
```

This tells us that we ran something 10,000 times, and it took 3 seconds of real time. These seconds were 2.58 spent in calculating ('usr' time) and 0 seconds interacting with the disk (or other non-calculating time). It also tells us that we ran through 3871.47 iterations of the test code each second.

To test several things and weigh them up against each other, we can use `timethese`. Instead of taking a string to represent code to be run, it takes an anonymous hash. The hash keys are names given to sections of the code, and the values are corresponding subroutine references, which we usually create anonymously.

To check the fastest way to read a file from the disk, we could do this:

```
#!/usr/bin/perl
# benchtest2.plx
use warnings;
use strict;

use Benchmark;
my $howmany = 100;

timethese($howmany, {
   line => sub {
      my $file;
      open TEST, "words" or die $!;
      while (<TEST>) { $file .= $_ }
      close TEST;
   },
   slurp => sub {
      my $file;
      local undef $/;
      open TEST, "words" or die $!;
      $file = <TEST>;
      close TEST;
   },
```

```
    join => sub {
       my $file;
       open TEST, "words" or die $!;
       $file = join "", <TEST>;
       close TEST;
    }
});
```

One way reads the file in a line at a time, one slurps the whole file in at once, and one joins the lines together. As you might expect, the slurp method is quite considerably faster:

```
Benchmark: timing 100 iterations of join, line, slurp...
     join: 42 wallclock secs (35.64 usr +  3.78 sys = 39.43 CPU) @  2.54/s (n=100)
     line: 37 wallclock secs (29.77 usr +  3.17 sys = 32.94 CPU) @  3.04/s (n=100)
    slurp:  6 wallclock secs ( 2.87 usr +  2.65 sys =  5.53 CPU) @ 18.09/s (n=100)
```

Also bear in mind that each benchmark will not only time differently between each machine and the next, but often between times you run the benchtest – so *don't* base your life around benchmark tests. If a pretty way to do it is a thousandth of a second slower than an ugly way to do it, choose the pretty one. If speed is really *that* important to you, you should probably be programming in something other than Perl.

Win32

Those familiar with Windows' labyrinthine Win32 APIs will probably want to examine the `libwin32` modules. These all live in the `Win32::` hierarchy (older versions may have some in the `OLE::` hierarchy too, but this was moved to `Win32::OLE::`) and come as standard with ActiveState Perl. If you've compiled another Perl yourself on Windows, you can get a copy of the modules from CPAN – we'll see how in a second.

These modules, which give you access to such things as Semaphores, Services, OLE, the Clipboard, and a whole bunch of other things besides, will probably be of most interest to existing Windows programmers. For the rest of us though, there are two modules that will be of particular use:

Win32::Sound

The first, `Win32::Sound`, lets us play with the sound subsystem – we can play `.wav` files, set the speaker volume, and so on. We can also use it to play the standard system sounds.

Try It Out : Playing .wav Files

The following program will play all the .wav files in the current directory:

```
#!/usr/bin/perl
# wavplay.plx
use warnings;
use strict;
use Win32::Sound;

my $wav;
Win32::Sound::Volume(65535);
opendir (DIR, ".") or die "Couldn't open directory: $!";
while ($wav = readdir(DIR)) {
   Win32::Sound::Play($wav);
}
```

You won't see any output, but if you're in a directory containing `.wav` files, you should certainly be able to hear some!

How It Works

The `Win32::Sound` module provides us with a number of subroutines:

Function	Description
`Win32::Sound::Volume(`*$left, $right*`)`	Sets the left and right speaker volumes to the requested amount. If only `$left` is given, both speakers are set to that volume. If neither is given, the current volume is returned. You can give the volume either as a percentage or a number from 0 to 65535.
`Win32::Sound::Play(`*$name*`)`	Plays the named sound file, or the named system sound. (for example, `SystemStart`)
`Win32::Sound::Format(`*$filename*`)`	Returns information about the format of the given sound file.
`Win32::Sound::Devices()`	Lists all the available sound-related devices on the system.
`Win32::Sound::DeviceInfo(`*$device*`)`	Provides information on the given sound device.

You can get a full list of the subroutines from the `Win32::Sound` documentation page if you have the module installed.

Win32::TieRegistry

Windows uses a centralized system database to store information about applications, users and its own state. This is called the **registry**, and we can get at it by using Perl's `Win32::TieRegistry` module. This just provides a convenient layer around the `Win32::Registry` module that is rather more technical in nature. `Win32::TieRegistry` transforms the Windows registry into a Perl hash.

The registry is a complicated beast, and revolves around a hierarchical tree structure like a hash of hashes or a directory. For instance, information about users' software is stored under `HKEY_CURRENT_USER\Microsoft\Windows\CurrentVersion\`. Now we can get to this particular part of the hash by saying the following:

```
#!/usr/bin/perl
# registry.plx
use warnings;
use strict;
use Win32::TieRegistry (Delimiter => "/") ;
```

We load the module, and change the delimiter from a backslash to a forward slash so we don't end up drowning in a sea of backslashes:

```
my $users = $Registry->
  {HKEY_CURRENT_USER/Software/Microsoft/Windows/CurrentVersion/};
```

Now we've got that key, we can dig further into the depths of the registry. This is where the Windows Explorer tips are stored:

```
my $tips = $users->{Explorer/Tips};
```

and from there we can add our own tips:

```
$tips->{/186} = "It's easy to use Perl as a Registry editor with the
Win32::TieRegistry module.";
```

We can always delete them again, using ordinary hash techniques:

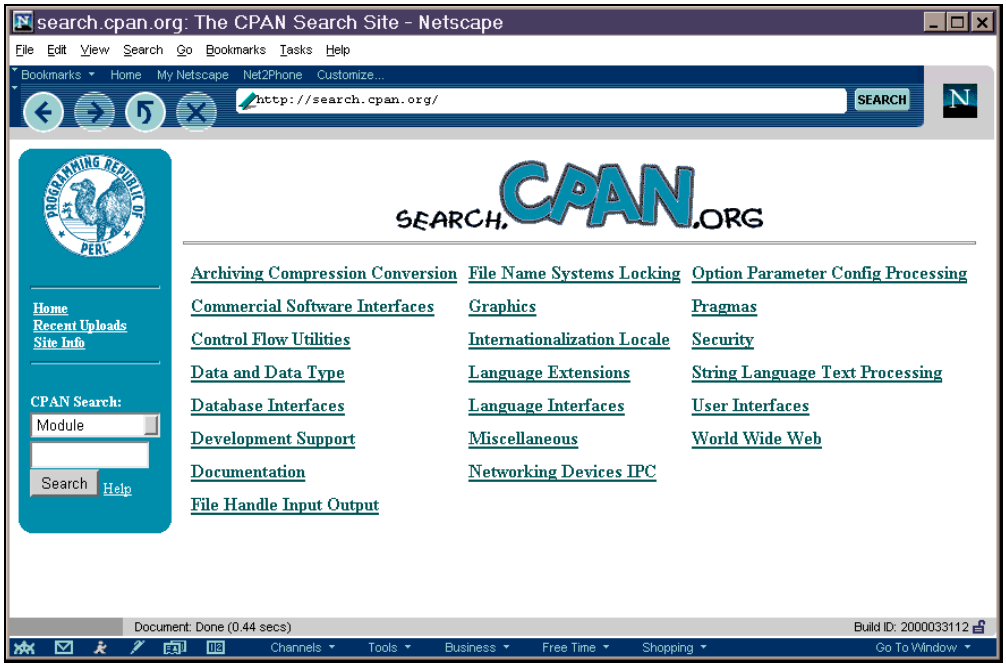
```
delete $tips->{/186};
```

Again, if you're after more information, it's available in the `Win32::TieRegistry` documentation, but I'd suggest you lay off reading that until you've digested the following chapter on object oriented Perl.

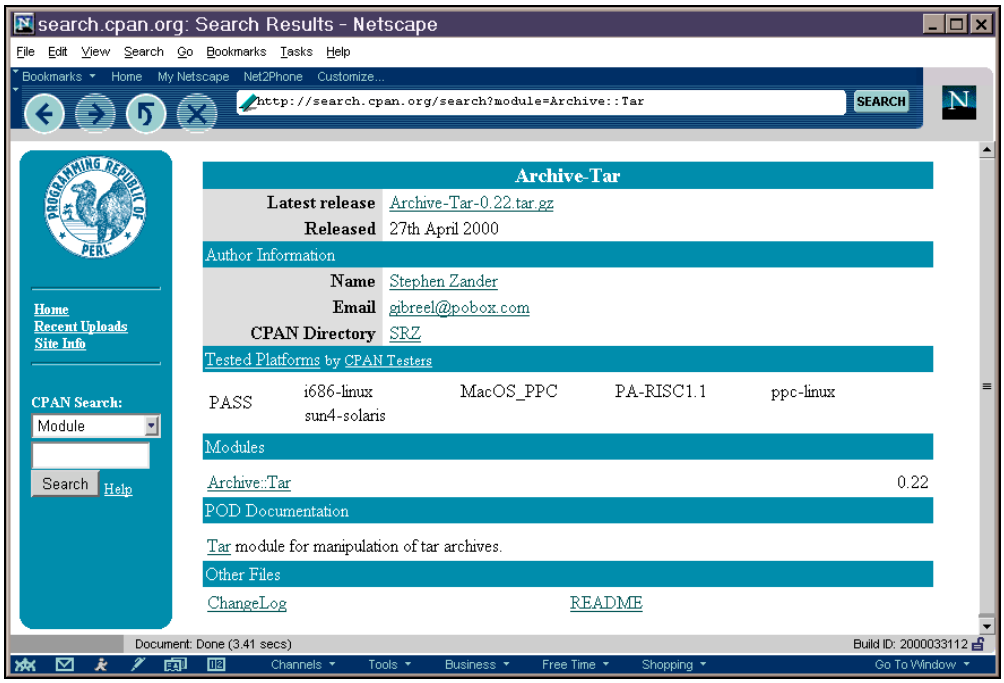
CPAN

So far we've been looking at standard modules provided with most perl distributions. However, as we mentioned in the introduction, there's also a central repository for Perl modules – collections of code that will do virtually any kind of job: the **Comprehensive Perl Archive Network**, or CPAN, which you can find on the web at http://www.cpan.org.

So before you ask 'how do I do...?' or start plugging away at any long task, it's always worth taking a quick look here to see if it's already been done. CPAN is searchable in plenty of different ways – the most common are by keyword, by topic, or by module name. There are also a few CPAN search engines, but the easiest for browsing is probably the web-based CPAN search engine at http://search.cpan.org/. Alternatively, if you know what you're looking for, http://theory.uwinnipeg.ca/search/cpan-search.html is rather good too.



This lets us look up modules by category, as well as searching for words in the modules' documentation. Once we've found a module that might do what we want, we follow a link to get further information on it and get ourselves a download. For example, this is what we get for the `Archive::Tar` module:



Now that we've seen how to find the modules we want, we're ready to look at the various ways in which we can install them.

Installing Modules with PPM

If you're using ActivePerl, module installation is made very simple by the **Perl Package Manager** (PPM). This is a useful little tool that's provided along with installations of ActivePerl, which allows us to install modules from the command line with the minimum of effort.

So without further ado, let's install the `Net::FTP` module. This is part of the 'libnet' bundle, a collection of modules which all relate to networking and which we'll be seeing again in Chapters 12 and 14. Installation is quite simple – it involves installing the libnet bundle of which `Net::FTP` is a part.

1. Type **ppm** at the command line, this will give you the PPM prompt: `PPM>`

2. Now type **install libnet** – you may be asked to confirm your request. If so type **y**.

3. If you have libnet already installed, you may be asked if you would like to modify/update your libnet configuration. You don't have to, so just type **no**.

4. Exit the PPM prompt by typing **quit**, and now you have `Net::FTP` installed, ready for the next couple of chapters.

Installing a Module Manually

We'll now take a look at what's involved in doing the same installation for ourselves. If you search CPAN for the module `Net::FTP`, you should find yourself looking at the file `libnet-1.0702.tar.gz` (unless there's a newer version out by the time you read this…) Download and unpack this file. On UNIX systems, `gzip -dc libnet-1.0702.tar.gz | tar -xvf` should do the trick, while you can use Winzip to extract these files on Windows.

> *Note that you will encounter serious problems with the following procedures if you're running on Windows9x. This being the case, I'd suggest sticking with PPM if you have the option. For Windows NT and Windows 2000 users however, read on.*

Every module should contain a `Makefile.PL`, which can be used to generate the instructions to install the module. Let's run that file first:

```
>perl Makefile.PL
Checking for Socket...ok
Checking for IO::Socket...ok
Checking if your kit is complete...
Looks good
Writing Makefile for Bundle::Libnet
>
```

If you can't install in Perl's site directories because you don't have the appropriate permissions, run:

```
>perl Makefile.PL PREFIX=/my/module/path
```

and it will arrange for the module to be installed there. You should then be sure to add that directory to `@INC` in all of your programs that will be making use of it. You can do this in three ways:

- use the `lib` pragma (as described above)
- use the `-I` flag at the command line
- modify the `PERL5LIB` environment variable

`Makefile.PL` first checks that we have all the modules it requires, and then that we've got everything we should have in the module archive itself – a file called `MANIFEST` contains a list of what should be in the archive. As we saw in step 3 of installing Libnet with PPM, we get asked for information concerning our networking configuration. The default option for each question is in brackets, and we'll get that if we press return. You can answer them as best you can or leave the defaults intact and once it's done with the questions, we'll finally see the message:

```
Writing Makefile for Net
```

Now we're ready to type `make` – assuming, of course, we have `make` on our system.

> *Windows users can download `nmake` from http://download.microsoft.com/download/vc15/Patch/1.52/W95/EN-US/Nmake15.exe. Just put it in one of your path directories and redirect the following calls from make to nmake.*

We run make, and it creates a directory called blib, holding all the files that it'll eventually copy to the real installation directory:

```
>make
mkdir blib
mkdir blib/lib
mkdir blib/arch
mkdir blib/arch/auto
mkdir blib/arch/auto/Net
mkdir blib/lib/auto
mkdir blib/lib/auto/Net
```

We run it again, and it now copies the files it needs to there:

```
>make
cp Net/Config.pm blib/lib/Net/Config.pm
cp Net/Domain.pm blib/lib/Net/Domain.pm
cp Net/SMTP.pm blib/lib/Net/SMTP.pm
cp Net/DummyInetd.pm blib/lib/Net/DummyInetd.pm
cp Net/Time.pm blib/lib/Net/Time.pm
cp Net/NNTP.pm blib/lib/Net/NNTP.pm
cp Net/FTP/dataconn.pm blib/lib/Net/FTP/dataconn.pm
cp Net/PH.pm blib/lib/Net/PH.pm
cp Net/FTP/A.pm blib/lib/Net/FTP/A.pm
cp Net/FTP/I.pm blib/lib/Net/FTP/I.pm
cp libnet.cfg blib/lib/Net/libnet.cfg
cp Net/POP3.pm blib/lib/Net/POP3.pm
…
```

Once that's done, we check to see if our module's working:

```
>make test
PERL_DL_NONLAZY=1 /usr/local/bin/perl -Iblib/arch -Iblib/lib -I/usr/local/lib/pe
rl5/5.6.0/cygwin -I/usr/local/lib/perl5/5.6.0 -e 'use Test::Harness qw(&runtests
 $verbose); $verbose=0; runtests @ARGV;' t/*.t
t/ftp.............ok
t/hostname.......ok
t/nntp............ok
t/ph..............ok
t/require.........ok
t/smtp............ok
All tests successful
Files=6, Tests=11, 10 wallclock secs ( 4.25 cusr +  4.19 csys =  8.44 CPU)
>
```

Finally, we actually install it, moving the files from blib to the correct location, as stored in @INC:

```
>make install
Installing /usr/local/lib/perl5/site_perl/5.6.0/Net/Cmd.pm
Installing /usr/local/lib/perl5/site_perl/5.6.0/Net/Config.pm
Installing /usr/local/lib/perl5/site_perl/5.6.0/Net/Domain.pm
Installing /usr/local/lib/perl5/site_perl/5.6.0/Net/DummyInetd.pm
Installing /usr/local/lib/perl5/site_perl/5.6.0/Net/FTP.pm
Installing /usr/local/lib/perl5/site_perl/5.6.0/Net/libnet.cfg
…
```

Hooray! The module's now installed.
However, there's a much, much easier way of doing it.

The CPAN Module

Another easy way to navigate and install modules from CPAN is to use the standard module called `CPAN`. The 'CPAN Shell' is an extremely powerful tool for finding, downloading, building and installing modules.

Again, note that you will encounter serious problems with the following procedures if you're running on Windows9x. Windows NT/2000 users note also that this routine really doesn't like spaces in directory paths, but there is a fix, as follows:

In Windows Explorer, go to `your_perl_install_directory\lib\CPAN`, and open the file `Config.pm` in your Perl editor. This is the CPAN module's systemwide configuration file and contains all the information the module needs to run. Scan down the list and if any of the paths to files contain spaces, you'll need to change them to their 8.3 format.

For example, let's say your copy of nmake.exe can be found at `C:\Program Files\Microsoft Visual Studio\vc98\bin\nmake.exe` because you previously installed it with Visual C++. Unfortunately, this path has been copied to `Config.pm` which means the module itself looks for `nmake` in `C:\Program` and of course doesn't find it. By changing the make entry in `Config.pm` from:

```
'make' => q[C:\Program Files\Microsoft Visual Studio\VC98\bin\nmake.EXE],
```

to

```
'make' => q[C:\Progra~1\Micros~3\VC98\bin\nmake.EXE],
```

the problem is solved.

Try It Out : Using the CPAN module

To get into the CPAN shell, put:

>perl –MCPAN –e shell

This is actually just the same as saying:

```
#!/usr/bin/perl
use CPAN;
shell();
```

The whole shell is actually a function in the (massively complex) `CPAN` module. The first time we run it, we'll see something like this:

/usr/local/lib/perl5/5.6.0/CPAN/Config.pm initialized.

```
CPAN is the world-wide archive of perl resources. It consists of about
100 sites that all replicate the same contents all around the globe.
Many countries have at least one CPAN site already. The resources
found on CPAN are easily accessible with the CPAN.pm module. If you
want to use CPAN.pm, you have to configure it properly.

If you do not want to enter a dialog now, you can answer 'no' to this
question and I'll try to autoconfigure. (Note: you can revisit this
dialog anytime later by typing 'o conf init' at the cpan prompt.)

Are you ready for manual configuration? [yes]
```

Press the *Enter* key, and you'll be asked a series of questions about your computer and the nearest CPAN server. If you don't know, just keep hitting Enter through the answers. Eventually, you'll end up at a prompt like this:

```
cpan shell -- CPAN exploration and modules installation (v1.52)
ReadLine support available (try "install Bundle::CPAN")

cpan>
```

Now we're ready to issue commands. The `install` command, as shown in the prompt, will download and install a module. For example, we could install the `DBD::mysql` module by simply saying

```
cpan>install DBD::mysql
```

Alternatively, we could get information on a module with the `i` command. Let's get some information on the MLDBM module, another module that we'll look into when we investigate databases:

```
cpan>i MLDBM
Module id = MLDBM
    DESCRIPTION  Transparently store multi-level data in DBM
    CPAN_USERID  GSAR (Gurusamy Sarathy <gsar@ActiveState.com>)
    CPAN_VERSION 2.00
    CPAN_FILE    GSAR/MLDBM-2.00.tar.gz
    DSLI_STATUS  RdpO (released,developer,perl,object-oriented)
    INST_FILE    (not installed)
```

So what does this tell us? Well, the module is called MLDBM, and there's a description of it. It was written by the CPAN author GSAR, who translates to Gurusamy Sarathy in the real world. It's at version 2.00, and it's stored on CPAN in the directory `GSAR/MLDBM-2.00.tar.gz`.

The funny little code thing is the CPAN classification, which we mentioned in the introduction. It tells us this module has been released (the implication being that it's been released for a while), that you should contact the developer if you need any support on it, that it's written purely in Perl without any extensions in C, and that it's object-oriented – and finally, that we don't have it installed. So let's install it:

```
cpan> install MLDBM
```

> *In fact, you don't even have to go into the shell to install a module. As well as exporting the `shell` subroutine, `CPAN` provides us with `install`, with which we can simply say* ***`perl -MCPAN -e 'install "MLDBM"'`*** *to produce the same results.*

You'll then see a few lines that will be specific to your computer. Different systems have different ways of downloading files and depend on whether or not you have the external programs `lynx`, `ftp`, or `ncftp` or the Perl `Net::FTP` module installed.

The `CPAN` module will download the file. Then, if you've got the `Digest::MD5` module installed, it will download a special file called a checksum – which provides a summary of that file so we make sure that what we've downloaded is what's on the server.

```
Checksum for /home/simon/.cpan/sources/authors/id/GSAR/MLDBM-2.00.tar.gz ok
```

Next, it'll unpack the file:

```
MLDBM-2.00/
MLDBM-2.00/Makefile.PL
MLDBM-2.00/t/
MLDBM-2.00/t/storable.t
MLDBM-2.00/t/freezethaw.t
MLDBM-2.00/t/dumper.t
MLDBM-2.00/lib/
MLDBM-2.00/lib/MLDBM/
MLDBM-2.00/lib/MLDBM/Serializer/
MLDBM-2.00/lib/MLDBM/Serializer/Storable.pm
MLDBM-2.00/lib/MLDBM/Serializer/FreezeThaw.pm
MLDBM-2.00/lib/MLDBM/Serializer/Data/
MLDBM-2.00/lib/MLDBM/Serializer/Data/Dumper.pm
MLDBM-2.00/lib/MLDBM.pm
MLDBM-2.00/Changes
MLDBM-2.00/README
MLDBM-2.00/MANIFEST
```

and then it will generate and run the Makefile:

```
CPAN.pm: Going to build GSAR/MLDBM-2.00.tar.gz

Checking if your kit is complete...
Looks good
Writing Makefile for MLDBM
mkdir blib
mkdir blib/lib
mkdir blib/arch
mkdir blib/arch/auto
mkdir blib/arch/auto/MLDBM
mkdir blib/lib/auto
mkdir blib/lib/auto/MLDBM
cp lib/MLDBM/Serializer/FreezeThaw.pm blib/lib/MLDBM/Serializer/FreezeThaw.pm
cp lib/MLDBM.pm blib/lib/MLDBM.pm
cp lib/MLDBM/Serializer/Storable.pm blib/lib/MLDBM/Serializer/Storable.pm
cp lib/MLDBM/Serializer/Data/Dumper.pm blib/lib/MLDBM/Serializer/Data/Dumper.pm
  /usr/local/bin/make  -- OK
```

Once that's successful, it'll test the module out:

```
Running make test
PERL_DL_NONLAZY=1 /usr/local/bin/perl -Iblib/arch -Iblib/lib -I/usr/local/lib/perl5/5.6.0/cygwin -
I/usr/local/lib/perl5/5.6.0 -e 'use Test::Harness qw(&runtests
 $verbose); $verbose=0; runtests @ARGV;' t/*.t
t/dumper.............ok
t/freezethaw........skipped test on this platform
t/storable..........skipped test on this platform
All tests successful, 2 tests skipped.
Files=3, Tests=4,  4 wallclock secs ( 1.29 cusr +  1.52 csys =  2.81 CPU)
  /usr/local/bin/make test -- OK
```

and finally install it:

```
Running make install
Installing /usr/local/lib/perl5/site_perl/5.6.0/MLDBM.pm
Installing /usr/local/lib/perl5/site_perl/5.6.0/MLDBM/Serializer/FreezeThaw.pm
Installing /usr/local/lib/perl5/site_perl/5.6.0/MLDBM/Serializer/Storable.pm
Installing /usr/local/lib/perl5/site_perl/5.6.0/MLDBM/Serializer/Data/Dumper.pm
Writing /usr/local/lib/perl5/site_perl/5.6.0/cygwin/auto/MLDBM/.packlist
Appending installation info to /usr/local/lib/perl5/5.6.0/cygwin/perllocal.pod
  /usr/local/bin/make install  -- OK

cpan>
```

Successfully installed, and with the minimum of effort!

How about if we don't actually know the name of the module we're looking for? Well, `CPAN` lets us use a regular expression match to locate modules. For instance, if we're about to do some work involving MIDI electronic music files, we could search for 'MIDI'. This is what we might see:

```
cpan>i /MIDI/
Distribution    F/FO/FOOCHRE/MIDI-Realtime-0.01.tar.gz
Distribution    S/SB/SBURKE/MIDI-Perl-0.75.tar.gz
Module          MIDI            (S/SB/SBURKE/MIDI-Perl-0.75.tar.gz)
Module          MIDI::Event     (S/SB/SBURKE/MIDI-Perl-0.75.tar.gz)
Module          MIDI::Opus      (S/SB/SBURKE/MIDI-Perl-0.75.tar.gz)
Module          MIDI::Realtime  (F/FO/FOOCHRE/MIDI-Realtime-0.01.tar.gz)
Module          MIDI::Score     (S/SB/SBURKE/MIDI-Perl-0.75.tar.gz)
Module          MIDI::Simple    (S/SB/SBURKE/MIDI-Perl-0.75.tar.gz)
Module          MIDI::Track     (S/SB/SBURKE/MIDI-Perl-0.75.tar.gz)
```

'Distributions' are archive files: zips or tar.gz files containing one or more Perl modules. We see that `MIDI-Realtime` contains just the `MIDI::Realtime` module, and Sean Burke's `MIDI-Perl` contains a few more modules, so perhaps we'd check that one out.

Bundles

Some modules depend on other modules being installed. For instance, the `Win32::TieRegistry` module needs `Win32::Registry` to do the hard work of getting at the registry. If you're downloading packages from CPAN manually, you'll have to try each package, find out what's missing and download another repeatedly until you've got everything you need. The `CPAN` module does a lot of this work for you. It can detect dependencies in packages and download and install everything that's missing.

This is fine for making sure that things work, but as well as *needing* other modules, some merely *suggest* other modules. For instance, the `CPAN` module itself works fine with nothing other than what's in the core, but if you have `Term::Readline` installed, it gives you a much more flexible prompt, with tab-completion, a command history meaning you can use the up and down arrows to scroll through previous commands, and other niceties.

Enter **bundles** – collections of packages that go well together. The CPAN bundle, `Bundle::CPAN`, for instance, contains various modules that make the CPAN shell easier to use: `Term::ReadLine` as we mentioned above, `Digest::MD5` for security checking the files downloaded, some `Net::` modules to make network communication with the CPAN servers nicer, and so on.

We'll now look here at two particularly useful bundles, which contain modules that I personally wouldn't go *anywhere* without.

Bundle::LWP

`Bundle::LWP` contains modules for *everything* to do with the Web. It has modules for dealing with HTML, HTTP, MIME types, handling URLs, downloading and mirroring remote web sites, creating web spiders and robots more advanced than our earlier webchecker program in Chapter 8, and so on.

The main chunk of the bundle is the `LWP` (`libwww-perl`) distribution, containing the modules for getting remote web sites. Back in Chapter 8, I mentioned that you could use `LWP::Simple` to get a `get` subroutine. Let's have a look at what else it gives us.

This module will export five subroutines to our current package.

- The **`get`** subroutine does exactly what we were previously trying to do with `lynx` – fetch a web site and return you the underlying HTML. However, this subroutine knows all about proxies, error codes, and the other things that we didn't properly check for:

```
$file = get("http://www.perl.com/");
```

- The **`head`** subroutine fetches the header of the site and returns a few headers: what type of document the page is (it's usually going to be `text/html`) how big it is in bytes, when it was last modified, when it should be regarded as old (these are both UNIX times suitable for feeding to `localtime`) and what the server has to say about itself. Some servers may not return all these headers:

```
($content_type, $document_length, $modified_time, $expires, $server) =
  head("http://www.perl.com/");
```

The next three routines are all quite similar in that they all involve retrieving an HTML page.

- The first, **`getprint`**, retrieves the HTML file and then prints it out to standard output – useful if you're redirecting to a file or using a filter as some sort of HTML formatter. You can copy a web page to a local file like this:

```
getprint("http://www.perl.com/");
```

- Alternatively, you can use the **`getstore`** subroutine to store it to a file:

```
perl -MLWP::Simple -e
   'getprint("http://www.perl.com/")' > perlpage.html
```

- Finally, **`mirror`** is like `getstore`, except it checks to see if the remote site's page is newer than the one we've already got:

```
perl -MLWP::Simple -e
   'getstore("http://www.perl.com/","perlpage.html")'
```

`Bundle::LWP` functions as a standard exporter, so, as we saw earlier in the chapter, we can prevent any of the five being pulled into our namespace by saying use `LWP::Simple()` – if you've already got a subroutine called `get` defined, this may be a good idea for example. Be sure to read the main `LWP` documentation and the `lwpcook` page which contains a few ideas for things to do with `LWP`.

Bundle::libnet

Similarly, `Bundle::libnet` contains a bunch of stuff for dealing with the network, although it's not nearly as big as LWP. The modules in `Bundle::libnet` and its dependencies allow you to use FTP, telnet, SMTP mail, and other network protocols. These are object-oriented modules, and we'll be looking at them some more in the next chapter.

Submitting Your Own Module to CPAN

CPAN contains almost everything you'll ever need. Almost. There'll surely come a day when you're faced with a problem where no known module can help you. If you think it's a sufficiently general problem that other people are going to come across, why not consider making your solution into a module and submitting it to CPAN? Think of it as a way of giving something back to the community that gave you all this…

Seriously, if you do have something you think would be useful to others, there are a few things you need to do to get it to CPAN. It will require a reasonable amount of work, and you may want to leave it until you've finished this book and had a look at the next book in our series, Professional Perl (*Due out October 2000 at time of publication -Ed*). Here's what you should do, though:

- Check it's not been written before. Search CPAN. Look at the main modules list at http://www.cpan.org/modules/00modlist.long.html – there's also a lot of good advice about how to lay out and prepare your module there.
- Read the `perlmod` and `perlmodlib` documentation pages until you really understand them.
- Learn about the `Carp` module, and use `carp` and `croak` instead of `warn` and `die`.
- Learn about the `Test` module and how to produce test suites for modules.
- Learn about documenting your modules in POD, Plain Old Documentation.
- Look at the source to a few simple modules like `Text::Wrap` and `Text::Tabs` to get a feel of how modules are written.
- Take a deep breath, and issue the following command:

```
>h2xs -AXn Your::Module::Name
```

- Edit the files produced, remembering to create a test suite and provide really good documentation.
- Run **`perl Makefile.PL`** and then **`make`**.

Your module's now ready to ship!

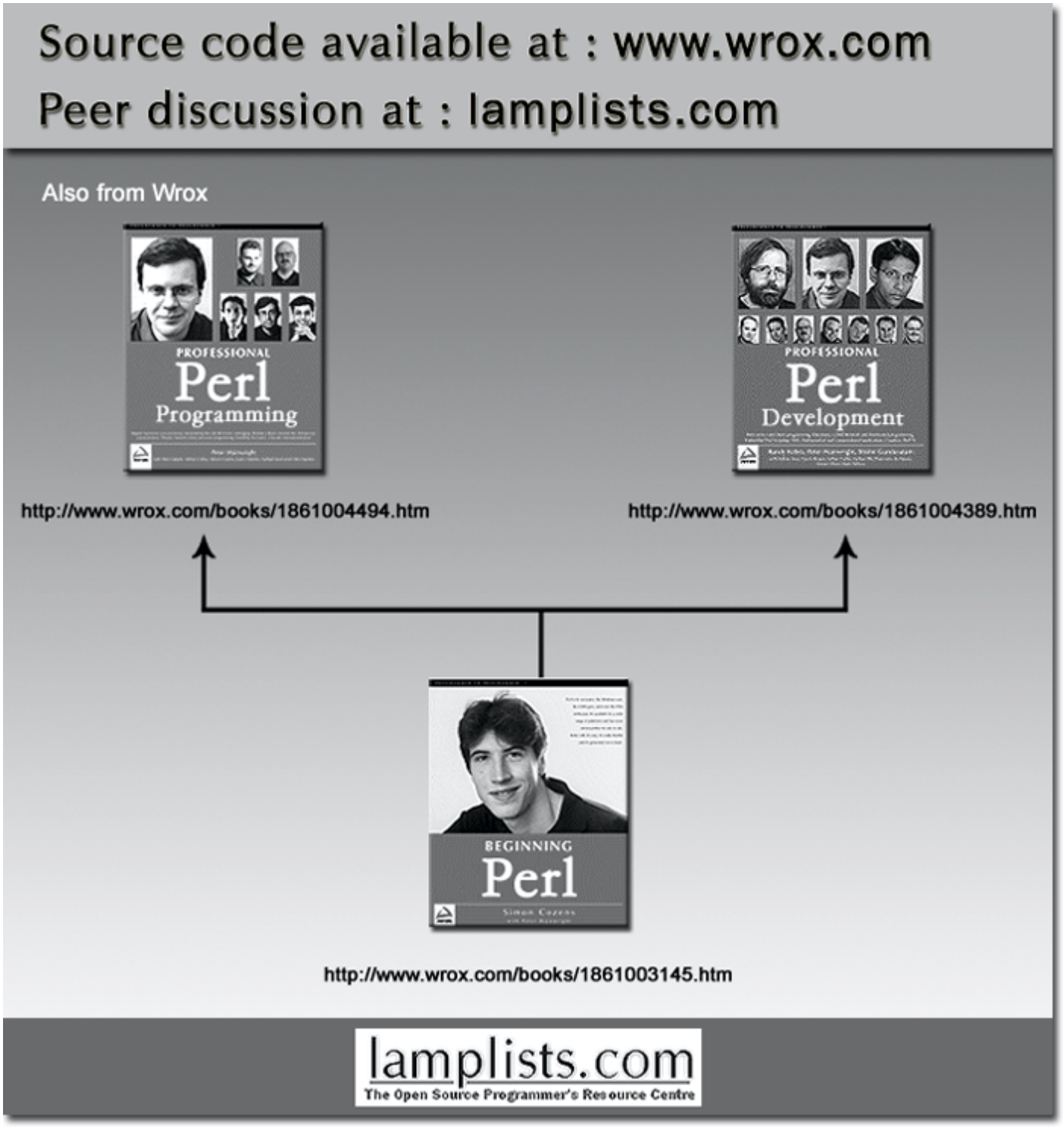
Summary

Modules save you time. Modules do things well. In essence, a module is just a package stored in a file, which we load with the `use` statement. We can load in other Perl code from files, using `require` or `do`, but `use` also calls the module's `import` subroutine, so that modules that want to can use `Exporter` to move subroutines into our package.

Perl provides a number of standard modules. You can check out the full list in Appendix D, and you can get documentation on each and every one by running `perldoc`. We looked briefly at `File::Find` (for examining files in directory trees), the `Getopt` modules (for reading options from the command line), the `File::Spec::Functions` module (for portable filename handling), the `Benchmark` module (for timing and testing code), and the `Win32` modules (for access to the Windows system and registry).

CPAN is the Comprehensive Perl Archive Network. It's a repository of good code. You can search it from http://search.cpan.org/ or use the Perl module `CPAN` for easy searching and installation. The `CPAN` module has the advantage of knowing about file dependencies and can therefore download and install files in the correct order.

Bundles provide sets of related modules. We looked at `LWP::Simple` (from the `libwww` bundle), and we'll look more at the `libnet` bundle in our chapter on networking. Finally, we looked at some of what's involved in abstracting your code and putting it into a module.



A

Regular Expressions

Pattern Matching Operators

Match – m//

Syntax: `m/`*`pattern`*`/`

If a match is found for the *`pattern`* within a referenced string (default `$_`), the expression returns true. (Note: If the delimiters `//` are used, the preceding `m` is not required.)

Modifiers: `g`, `i`, `m`, `o`, `s`, `x`

Substitution – s///

Syntax: `s/`*`pattern1`*`/`*`pattern2`*`/`

If a match is found for *`pattern1`* within a referenced string (default `$_`), the relevant substring is replaced by the contents of *`pattern2`*, and the expression returns true.

Modifiers: `e`, `g`, `i`, `m`, `o`, `s`, `x`

Transliteration – tr/// or y///

Syntax: `tr/`*`pattern1`*`/`*`pattern2`*`/`
`y/`*`pattern1`*`/`*`pattern2`*`/`

If any characters in *`pattern1`* match those within a referenced string (default `$_`), instances of each are replaced by the corresponding character in `pattern2`, and the expression returns the number of characters replaced. (Note: If one character occurs several times within `pattern1`, only the first will be used – for example, `tr/abbc/xyz/` is equivalent to `tr/abc/xyz/`.)

Modifiers: `c`, `d`, `s`

Delimiters

Patterns may be delimited by character pairs `<>`, `()`, `[]`, `{}`, or any other non-word character, e.g.:

```
s<pattern1><pattern2>
```

and

```
s#pattern1#pattern2#
```

are both equivalent to

```
s/pattern1/pattern2/
```

Binding Operators

Binding Operator =~

Syntax: *$refstring* `=~ m/`*pattern*`/`

Binds a match operator to a variable other than `$_`. Returns true if a match is found.

Negation Operator !~

Syntax: *$refstring* `!~ m/`*pattern*`/`

Binds a match operator to a variable other than `$_`. Returns true if a match is not found.

Modifiers

Match and Substitution

The following can be used to modify the behavior of match and substitution operators:

Cancel Position Reset - /c

Used only with global matches, that is, as `m//gc`, to prevent the search cursor returning to the start of the string if a match cannot be found. Instead, it remains at the end of the last match found.

Evaluate Replacement – /e

Evaluates the second argument of the substitution operator as an expression.

Global Match – /g

Finds all the instances in which the pattern matches the string rather than stopping at the first match. Multiple matches will be numbered in the operator's return value.

Case-Insensitive – /i

Matches pattern against string while ignoring the case of the characters in either pattern or string.

Multi-Line Mode – /m

The string to be matched against is to be regarded as a collection of separate lines, with the result that the metacharacters ^ and $, which would otherwise just match the beginning and end of the entire text, now also match the beginning and end of each line.

One-Time Pattern Compilation - /o

If a pattern to match against a string contains variables, these are interpolated to form part of the pattern. Later these variables may change, and the pattern will change with it when next matched against. By adding /o, the pattern will be formed once and will not be recompiled even if the variables within have changed value.

Single-Line Mode – /s

The string to be matched against will be regarded as a single line of text, with the result that the metacharacter . will match against the newline character, which it would not do otherwise.

Free-Form – /x

Allows the use of whitespace and comments inside a match to expand and explain the expression.

Transliteration

The following can be used to modify the behavior of the transliteration operator:

Complement - /c

Uses complement of `pattern1` – substitutes all characters *except* those specified in `pattern1`.

Delete - /d

Deletes all the characters found but not replaced.

Squash - /s

Multiple replaced characters squashed - only returned once to transliterated string.

Localized Modifiers

Syntax:

```
/CaseSensitiveTxt((?i)CaseInsensitiveTxt)CaseSensitiveText/

/CaseInsensitiveTxt((?-i)CaseSensitiveTxt)CaseInsensitiveText/i
```

The following inline modifiers can be placed within a regular expression to enforce or negate relevant matching behavior on limited portions of the expression:

Modifier	Description	inline enforce	inline negate
/i	case insensitive	(?i)	(?-i)
/s	single-line mode	(?s)	(?-s)
/m	multi-line mode	(?m)	(?-m)
/x	free-form	(?x)	(?-x)

Metacharacters

Metacharacter	Meaning
`[abc]`	Any one of a, b, or c.
`[^abc]`	Anything other than a, b, and c.
`\d \D`	A digit; a non-digit.
`\w \W`	A 'word' character; a non-'word' character.
`\s \S`	A whitespace character; a non-whitespace character.
`\b`	The boundary between a \w character and a \W character.
`.`	Any single character (apart from a new line).
`(abc)`	The phrase 'abc' as a group.
`?`	Preceding character or group may be present 0 or 1 times.
`+`	Preceding character or group is present 1 or more times.
`*`	Preceding character or group may be present 0 or more times.
`{x,y}`	Preceding character or group is present between *x* and *y* times.
`{,y}`	Preceding character or group is present at most *y* times.
`{x,}`	Preceding character or group is present at least *x* times.
`{x}`	Preceding character or group is present *x* times.

Non-greediness For Quantifiers

Syntax: `(`*`pattern`*`)+?`
`(`*`pattern`*`)*?`

The metacharacters + and * are greedy by default and will try to match as much as possible of the referenced string (while still achieving a full pattern match). This 'greedy' behavior can be turned off by placing a ? immediately after the respective metacharacter. A non-greedy match finds the minimum number of characters matching the pattern.

Grouping and Alternation

| *For Alternation*

Syntax: *`pattern1`*`|`*`pattern2`*

By separating two patterns with |, we can specify that a match on one *or* the other should be attempted.

() For Grouping And Backreferences ('Capturing')

Syntax: `(pattern)`

This will group elements in *pattern*. If those elements are matched, a backreference is made to one of the numeric special variables (`$1`, `$2`, `$3` etc.)

(?:) For Non-backreferenced Grouping ('Clustering')

Syntax: `(?:pattern)`

This will group elements in *pattern* without making backreferences.

Lookahead/behind Assertions

(?=) For Positive Lookahead

Syntax: *pattern1*`(?=`*pattern2*`)`

This lets us look for a match on '*pattern1* followed by *pattern2*', without backreferencing *pattern2*.

(?!) For Negative Lookahead

Syntax: *pattern1*`(?!`*pattern2*`)`

This lets us look for a match on '*pattern1* **not** followed by *pattern2*', without backreferencing *pattern2*.

(?<=) For Positive Lookbehind

Syntax: *pattern1*`(?<=`*pattern2*`)`

This lets us look for a match on '*pattern1* preceded by *pattern2*', without backreferencing *pattern2*. This only works if *pattern2* is of fixed width.

(?<!) For Negative Lookbehind

Syntax: *pattern1*`(?<!`*pattern2*`)`

This lets us look for a match on '*pattern1* **not** preceded by *pattern2*', without backreferencing *pattern2*. This only works if *pattern2* is of fixed width.

Backreference Variables

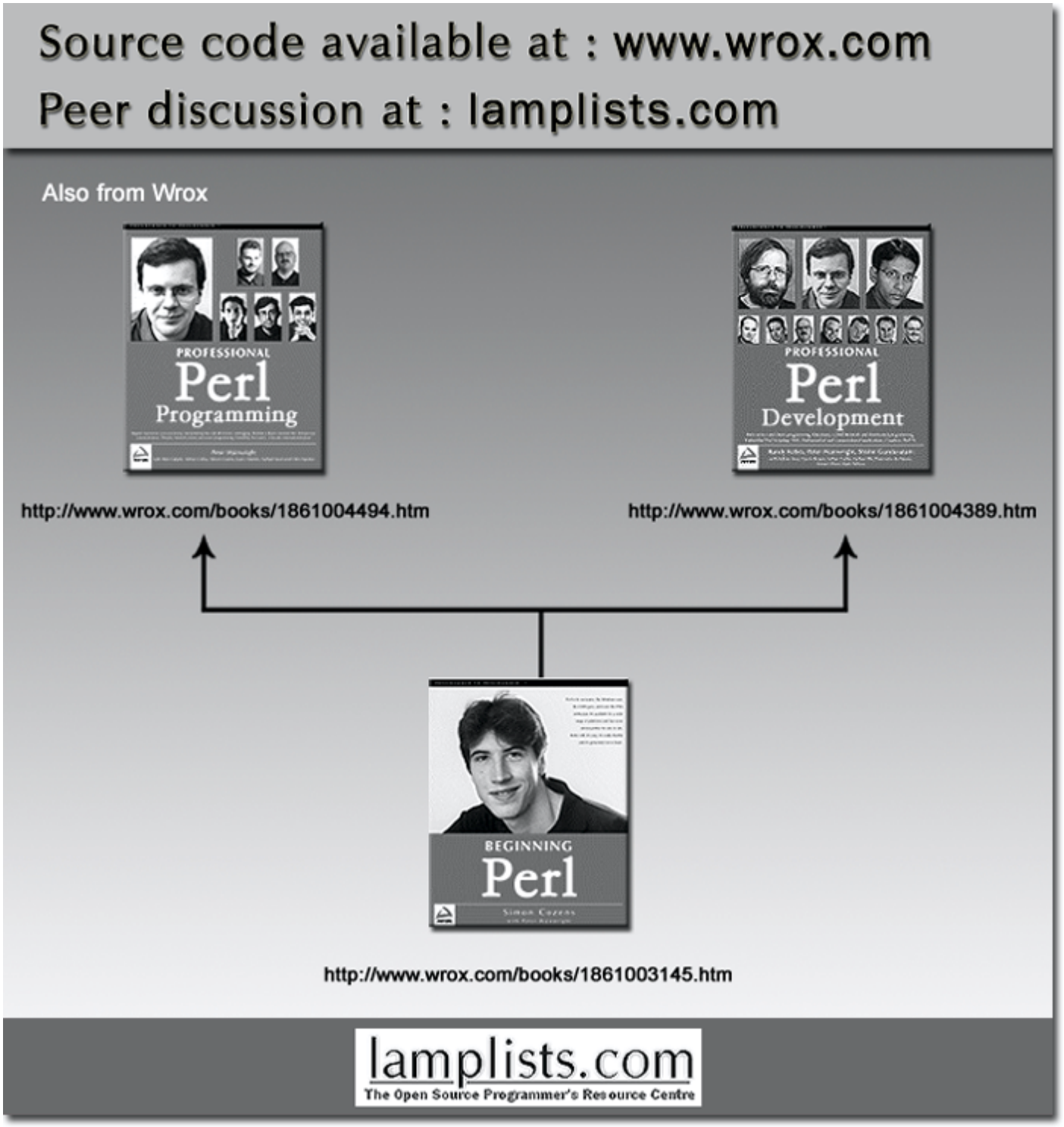
Variable	Description
num (num = 1, 2, 3…)	Within a regular expression, *num* returns the substring that was matched with the *num*th grouped pattern in that regexp.
$*num* (num = 1, 2, 3…)	Outside a regular expression, $*num* returns the substring that was matched with the *num*th grouped pattern in that regexp.
$+	This returns the substring matched with the last grouped pattern in a regexp.
$&	This returns the string that matched the whole regexp – this will include portions of the string that matched (?:) groups, which are otherwise not backreferenced.
$`	This returns everything preceding the matched string in $&.
$'	This returns everything following the matched string in $&.

Other

(?#) For Comments

Syntax: (?#*comment_text*)

This lets us place comments within the body of a regular expression – an alternative to the /x modifier.



B

Special Variables

Default Variables and Parameters

Variable	Description

`$_`	This global scalar acts as a default variable for function arguments and pattern-searching space – with many common functions, if an argument is left unspecified, `$_` will be automatically assigned, so, for example, the following statements are equivalent:
	`chop($_)` and `chop`
	`$_ =~ m/`*`expr`*`/` and `m/`*`expr`*`/`

`@_`	The elements of this array are used to store function arguments, which can be accessed (from within a function definition) as `$_[`*`num`*`]`. The array is automatically local to each function.
`@ARGV`	The elements of this array contain the command line arguments intended for use by the script.
`$ARGV`	This contains the name of the current file when reading from the null filehandle `<>`. (`<>` is a literal, and defaults to standard input, `<STDIN>`, if no arguments are supplied from elements in `@ARGV`).

Regular Expression Variables (all read-only)

Variable	Description
`$(num)`	The scalar `$n` contains the substring matched to the *n*'th grouped subpattern in the last pattern match, and remains in scope until the next pattern match with subexpressions. It ignores matched patterns occurring in nested blocks that are already exited. If there are no corresponding groups, then the undefined value is returned.

`$&`	This scalar contains the string matched by the last successful pattern match. Once again, this won't include any strings matched in nested blocks. For example:
	`'UnicornNovember' =~ /Nov/;`
	`print $&;`
	will print Nov. For versions of perl since 5.005, this is not an expensive variable to use.

`$'`	This scalar holds the substring following whatever was matched by the last successful pattern match. For example, if we say: `'UnicornNovember' =~ /Nov/;` `$'` will return ember.
`$``	This scalar holds the substring preceding whatever was matched by the last successful pattern match. For example, if we say: `'UnicornNovember' =~ /Nov/;` `$`` will return Unicorn.
`$+`	This scalar holds the last substring matched to a grouped subpattern in the last search. It comes in handy if you're not sure which of a set of alternative subpatterns matched. For example, if you successfully match on `/(ab)*| (bc*)/`, then `$+` stores either `$1` or `$2`, depending on whether it was the first or second grouped subpattern that matched. For example, following: `'UnicornNovember' =~ /(Nov)\|(Dec)/;` `$+` will return Nov.

`@+`	This array lists the back pointer positions (in the referenced string) of the last successful match. The first element `@+[0]` contains the pointer's starting position following that match – each subsequent value corresponds to its position just *after* having matched the corresponding grouped subpattern. For example, following:
	`'UnicornNovember' =~ /(U)\w?(N)/;`
	`@+` will return (8,1,8), while following:
	`'UnicornNovember' =~ /(Uni)\w?(Nov)/;`
	`@+` will return (10,3,10).

Variable	Description

`@-`	This array lists the front pointer positions (in the referenced string) of the last successful match. The first element `@-[0]` contains the pointer's starting position prior to that match – each subsequent value corresponds to its position just *before* having matched the corresponding grouped subpattern. For example following:
	`'UnicornNovember' =~ /(Uni)\w?(Nov)/;`
	`@-` will return (0,0,7), while following:
	`'UnicornNovember' =~ /(Uni)(\w?)(Nov)/;`
	`@-` will return (0,0,3,7).

Input/Output Variables

Variable	Description
`$.`	This scalar holds the **current line number** of the last filehandle on which you performed either a `read`, `seek`, or `tell`. It is reset when the filehandle is closed. NB: `<>` never does an explicit close, so line numbers increase across `ARGV` files – also, localizing `$.` has the effect of also localizing perl's notion of 'the last read filehandle'.
`$/`	This scalar stores the **input record separator**, which by default is the newline `\n`. If it's set to "", input will be read one paragraph at a time.

`$\`	This scalar stores the **output record separator** for `print` – normally this will just output consecutive records without any separation (unless explicitly included). This variable allows you to set it for yourself. For example:
	`$\ = "-";`
	`print "one";`
	`print "two";`
	will print one-two-.

`$|`	This corresponds to an internal flag used by perl to determine whether buffering should be used on a program's write/read operations to/from files. If the value is `TRUE` (`$|` is greater than 0), buffering is disabled.

`$,`	This is the **output field separator** for `print` – normally this will just output consecutive fields without any separation (unless explicitly included). This variable allows you to set it for yourself. For example:
	`$, = "-";`
	`print "one","two";`
	will print one-two.

Table continued on following page

Variable	Description

`$"`	This is the **output field separator** for array values interpolated into a double-quoted string (or similar interpreted string) – the default is a space. For example:
	`$" = "-";`
	`@ar = ("one", "two", "three");`
	`print "@ar";`
	will print one-two-three.

Filehandle/format Variables

Variable	Description
`$#`	This holds the **output format** for printed numbers. **NB: The use of this variable has been deprecated.**
`$|`	This corresponds to an internal flag used by perl to determine whether **buffering** should be used on a program's write/read operations to/from files – if its value is `TRUE` (`$|` is greater than 0), then buffering is disabled.
`$%`	The current page number of the selected output channel.
`$=`	The current **page length**, measured in printable lines – the default is 60. This only becomes important when a top-of-page format is invoked – if a `write` command doesn't fit into a given number of lines, then the top-of-page format is used, before any printing past the page length continues.
`$-`	The number of lines left on a page – when a page is finished, it's given the value of `$=`, and is then decremented for each line outputted.
`$~`	The currently selected **format name** – the default is the name of the filehandle.
`$^`	The name of the **top-of-page format**.
`$:`	The set of characters after which a string may be broken to fill continuation fields (starting with `^`) in a format – default is `" \n-"` to break on whitespace or hyphens.
`$^L`	This holds a character that is used by a format's output to request a form feed – default is `\f`.

Error Variables

Variable	Description
`$?`	This holds the status value returned by the last pipe close, backtick (``` `` ```) command, or `system()` operator.
`$@`	This holds the **syntax error message** from the last `eval()` command – it evaluates to null if the last `eval()` parsed and executed correctly (although the operations you invoked may have failed in the normal fashion).

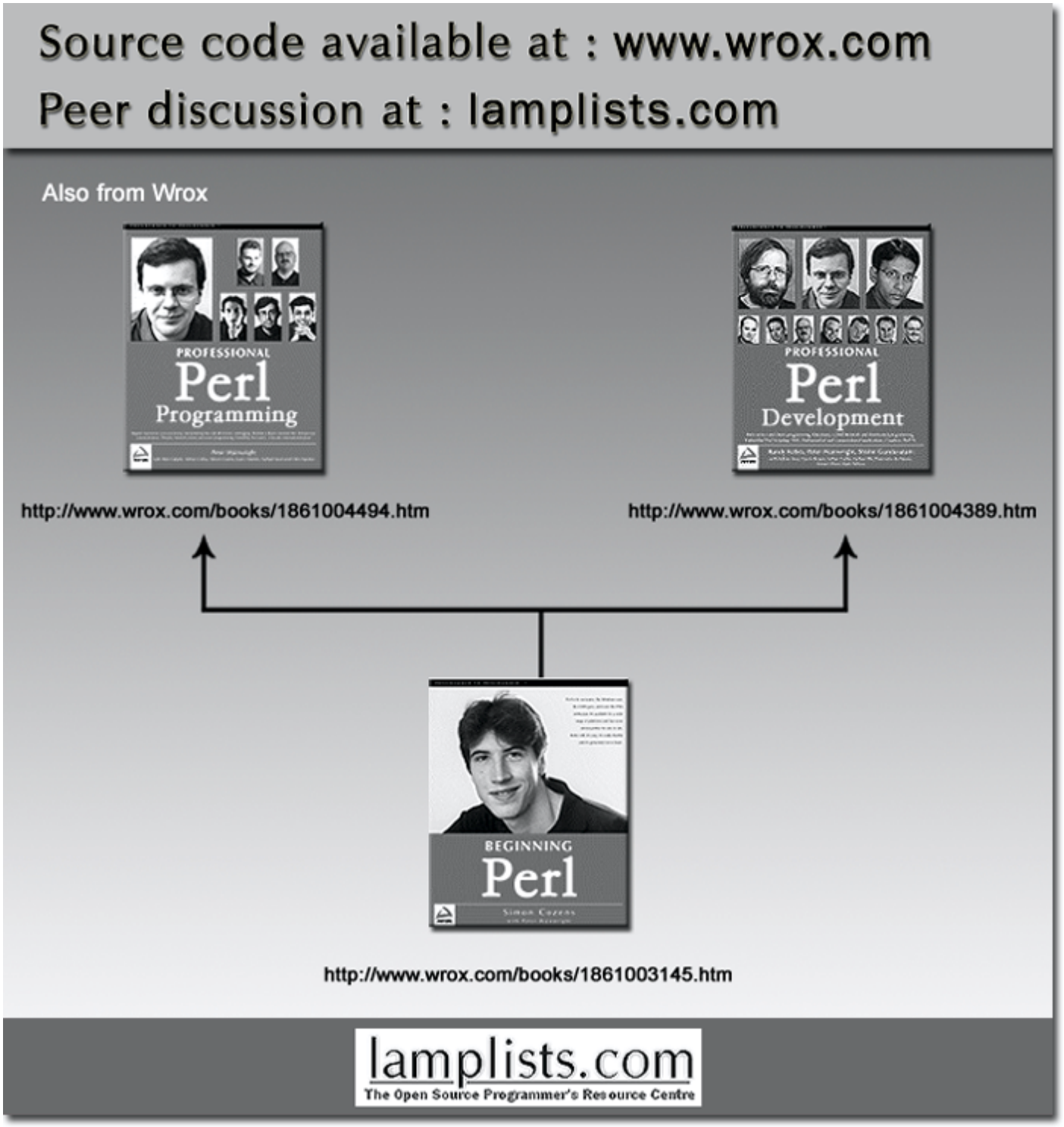
Variable	Description
`$!`	If used in a numeric context, this returns the current value of *errno*, with all the usual caveats. (so you shouldn't depend on `$!` to have any particular value unless you've got a specific error return indicating a system error.) If used in a string context, it returns the corresponding system error string. You can assign a set *errno* value to `$!` if, for instance, you want it to return the string for that error number, or you want to set the exit value for the `die()` operator.
`$^E`	This returns an **extended error message**, with information specific to the current operating system. At the moment, this only differs from `$!` under VMS, OS/2, and Win32 (and for MacPerl). On all other platforms, `$^E` is always the same as `$!`.

System Variables

Variable	Description
`$$`	The **process ID** (pid) of the Perl process running the current script.
`$<`	The **real user ID** (uid) of the current process.
`$>`	The **effective uid** of the current process. NB: `$<` and `$>` can only be swapped on machines supporting `setreuid()`.
`$(`	The **real group ID** (gid) of the current process.
`$)`	The **effective group ID** (gid) of the current process.
`$0` (zero)	The name of the file containing the Perl script being executed.
`$^X`	The name that the perl binary was executed as.
`$]`	The version number of the perl interpreter, including patchlevel / 1000 – can be used to determine whether the interpreter executing a script is within the right range of versions. See also `use VERSION` and `require VERSION` for a way to fail if the interpreter is too old.
`$^O`	The name of the operating system under which this copy of perl was built, as determined during the configuration process – identical to `$Config{'osname'}`.
`$^T`	The time at which the current script began running, in seconds since the beginning of 1970. Values returned by `-M`, `-A`, and `-C` filetests are based on this value.
`$^W`	The current value of the warning switch, either `TRUE` or `FALSE`.
`%ENV`	Your current environment – altering its value changes the environment for child processes.
`%SIG`	Used to set handlers for various signals.

Others

Variable	Description
`@INC`	A list of places to look for Perl scripts for evaluation by the `do EXPR`, `require`, or `use` constructs.
`%INC`	Contains entries for each filename that has been included via `do` or `require`. The key is the specified filename, and the value the location of the file actually found. The `require` command uses this array to determine whether a given file has already been included.



C

Function Reference

Below you'll find an alphabetical list of every function in Perl 5.6 starting with a runthrough of the file tests which are themselves functions. Marked against each function will be the syntax for the function, a brief description of what it does and any directly related functions.

File Tests

Function	Syntax	Description
`-X` (file tests)	`-X` *filehandle* `-X` *expression* `-X`	Runs a file test, as described in Chapter 6, determined by X, where X is one of the following letters: `ABCMORSTWX bcdefgkloprstuwxz` If the filehandle or expression arument is omitted, the file test checks against `$_`, with the exception of `-t`, which tests `STDIN`.

Here's a complete rundown of what each file test checks for.

Test	Meaning
`-A`	How long in days between the last access to the file and latest startup.
`-B`	True if the file is a binary file, (compare with `-T`).
`-C`	How long in days between the last inode change and latest startup.
`-M`	How long in days between the last modification to the file and latest startup.
`-O`	True if the file is owned by a real uid/gid.
`-R`	True if the file is readable by a real uid/gid.
`-S`	True if the file is a socket.

Table continued on following page

Test	Meaning
-T	True if the file is a text file, (Compare with -B).
-W	True if the file is writable by a real uid/gid.
-X	True if the file is executable by a real uid/gid.
-b	True if the file is a block special file.
-c	True if the file is a character special file.
-d	True if the file is a directory.
-e	True if the file exists.
-f	True if the file is a plain file - not a directory.
-g	True if the file has the setgid bit set.
-k	True if the file has the sticky bit set.
-l	True if the file is a symbolic link.
-o	True if the file is owned by an effective uid/gid.
-p	True if the file is a named pipe or if the filehandle is a named pipe.
-r	True if the file is readable by an effective uid/gid.
-s	True if the file has nonzero size - returns size of file in bytes.
-t	True if the filehandle is opened to a tty.
-u	True if the file has the setuid bit set.
-w	True if the file is writable by an effective uid/gid.
-x	True if the file is executable by an effective uid/gid.
-z	True if the file has zero size.

A

Function	Syntax	Description
abs	abs *value* abs	Returns the absolute (non-negative) value of an integer. E.g. abs(-1) and abs(1) both return 1 as a reuslt. If no *value* argument is given, abs returns the absolute value of $_.
accept	accept *newsocket, genericsocket*	Accepts an incoming socket connect with sessions enabled, if applicable.

Function	Syntax	Description
`alarm`	`alarm` *num_seconds* `alarm`	Starts a timer with *num_seconds* seconds on the clock before it trips a `SIGALRM` signal. Before the timer runs out, another call to `alarm` cancels it and starts a new timer with `num_seconds` on the clock. If `num_seconds` equals zero, the previous timer is cancelled without starting a new one.
`atan2`	`atan2` *x, y*	Returns the arctangent of *x/y* within the range -• to •.

B

Function	Syntax	Description
`bind`	`bind` *socket, name*	Binds a network address (TCP/IP, UDP, etc) to a *socket*, where *name* should be the packed address for the socket.
`binmode`	`binmode` *filehandle*	Sets the specified *filehandle* to be read in binary mode explicitly for those systems that cannot do this automatically. Unix and MacOS can, and thus `binmode` has no effect under these OS's.
`bless`	`bless` *ref, classname* `bless` *ref*	Takes the variable referenced by *ref* and makes it an object of class *classname*.

C

Function	Syntax	Description
`caller`	`caller` *expression* `caller`	Called within a subroutine, `caller` returns a list of information outlining what called it - the sub's context. Actually returns the caller's package name, its filename and line number of the call. Returns the undefined value if not in a subroutine. If *expression* is used, also returns some extra debugging information to make a stack trace.
`chdir`	`chdir` *new_directory* `chdir`	Changes your current working directory to *new_directory*. If *new_directory* is omitted, the working directory is changed to that one specified in `$ENV(HOME)`.
`chmod`	`chmod` *list*	Changes the permissions on a list of files. The first element of *list* must be the octal representation of the permissions to be given those files.
`chomp`	`chomp` *variable* `chomp` *list* `chomp`	Usually removes `\n` from a string. Actually removes the trailing record separator as set in `$/` from a string or from each string in a list, and then returns the number of characters deleted. If no argument is given, `chomp` acts on `$_`.

Table continued on following page

Function	Syntax	Description
chop	chop *variable* chop *list* chop	Removes the last character from a string or from each string in a list, and returns the (last) character chopped. If no argument is given, chop acts on $_.
chown	chown *list*	Changes the ownership on a list of files. Within *list*, the first two elements must be the user id and group id of that user and group to get ownership, followed by any number of filenames. Setting -1 for either id means, 'Leave this value unchanged.'
chr	chr *number* chr	Returns ASCII character number *number* as determined by Appendix F. If *number* is omitted, $_ is used.
chroot	chroot *directory* chroot	Changes the root directory for all further path lookups to *directory*. If *directory* is not given, $_ is used as the new root directory.
close	close *filehandle* close	Closes the file, pipe, or socket associated with the nominated *filehandle*, resetting the line counter $. as well. If *filehandle* is not given, closes the currently selected filehandle. Returns true on success.
closedir	closedir *dirhandle*	Closes the directory opened by opendir() given by *dirhandle*.
connect	connect *socket, address*	Tries to connect to a *socket* at the given *address*.
cos	cos *num_in_radians*	Calculates and returns the cosine of a number given in radians. If *num_in_radians* is not given, calculates the cosine of $_.
crypt	crypt *plaintext, key*	A one-way encryption function (there is no decrypt function) that takes some *plaintext* (a password usually) and encrypts it with a two character *key*.

D

Function	Syntax	Description
dbmclose	dbmclose *hash*	Deprecated in favor of untie(). Breaks the binding between a dbm file and the given *hash*.

Function	Syntax	Description
dbmopen	dbmopen *hash, dbname, mode*	Deprecated in favor of `tie()`. Binds the specified *hash* to the database *dbname*. If the database does not exist, it is created with the specified read\write *mode*, given as an octal number.
defined	defined *expression* defined	Checks whether the value, variable, or function in *expression* is defined. If *expression* is omitted, `$_` is checked.
delete	delete *$hash{key}* delete *@hash{keys %hash}*	Deletes one or more specified *key* and corresponding value from the *hash*. Returns the associated value(s).
die	die *message*	Writes *message* to the standard error output and then exits the currently running program with `$!` as its return value.
do	do *filename*	Executes the contents of *filename* as a perl script. Returns `undef` if it cannot read the file. Note: `do` *block* is not a function.
dump	dump *label* dump	Initiates a core dump to be undumped into a new binary executable file, which when run will start at *label*. If *label* is left out, the executable will start from the top of the file.

E

Function	Syntax	Description
each	each *hash*	Returns the next key/value pair from a *hash* as a two-element list. When *hash* is fully read, returns `null`.
endgrent	engrent	Frees the resources used to scan the `/etc/group` file or system equivalent.
endhostent	endhostent	Frees the resources used to scan the `/etc/hosts` file or system equivalent.
endnetent	endnetent	Frees the resources used to scan the `/etc/networks` file or system equivalent.
endprotoent	endprotoent	Frees the resources used to scan the `/etc/protocols` file or system equivalent.

Table continued on following page

Function	Syntax	Description
endpwent	endpwent	Frees the resources used to scan the /etc/passwd file or system equivalent.
endservent	endservent	Frees the resources used to scan the /etc/services file or system equivalent.
eof	eof *filehandle* eof() eof	Returns 1 if *filehandle* is either not open or will return end of file on next read. eof() checks for the end of the pseudo file containing the files listed on the command line as program was run. If eof does not have an argument, it will check the lst file to be read.
eval	eval *string* eval *block* eval	Parses and executes *string* as if it were a mini-program and returns its result. If no argument is given, it evaluates $_. If an error occurs or die() is called eval, returns undef. Works similarly with *block* except eval *block* is parsed only once. eval *string* is reparsed each time eval executes.
exec	exec *command*	Abandons the current program to run the specified system *command*.
exists	exists *$hash{$key}*	Returns true if the specified *key* exists within the specified *hash*.
exit	exit *status*	Terminates current program immediately with return value *status*. (N.B. The only universally recognized return values are 1 for failure and 0 for success.)
exp	exp *number*	Returns the value of e to the power of *number* (or $_ if number is omitted).

F

Function	Syntax	Description
fcntl	fcntl *filehandle, function, args*	Calls the fcntl function, to use on the file or device opened with *filehandle*.
fileno	fileno *filehandle*	Returns the file descriptor for *filehandle*.
flock	flock *filehandle, locktype*	Tries to lock or unlock a write-enabled file for use by the program. Note that this lock is only advisory and that other systems not supporting flock will be able to write to the file. *locktype* can take one of four values; LOCK_SH (new shared lock), LOCK_EX (new exclusive lock), LOCK_UN (unlock file), and LOCK_NB (do not block access to the file for a new lock if file not instantly available). Returns true for success, false for failure.

Function	Syntax	Description
`fork`	`fork`	System call that creates a new system process also running this program from the same point the fork was called. Returns the new process' id to the original program, `0` to the new process, or `undef` if the fork did not succeed.
`format`	`format`	Declares an output template for use with `write()`.
`formline`	`formline` *template, list*	An internal function used for formats. Applies *template* to the *list* of values and stores the result in `$^A`. Always returns `true`.

G

Function	Syntax	Description
`getc`	`getc` *filehandle* `getc`	Waits for the user to press Return and then retrieves the next character from *filehandle*'s file. Returns `undef` if at the end of a file. If *filehandle* is omitted, uses `STDIN` instead.
`getgrent`	`getgrent`	Gets the next group record from `/etc/group` or the system equivalent, returning an empty record when the end of the file is reached.
`getgrgid`	`getgrgid` *gid*	Gets the group record from `/etc/group` or the system equivalent whose id field matches the given group number *gid*. Returns an empty record if no match occurs.
`getgrnam`	`getgrnam` *name*	Gets the group record from `/etc/group` or the system equivalent whose name field matches the given group *name*. Returns an empty record if no match occurs.
`gethostbyaddr`	`gethostbyaddr` *address, addrtype* `gethostbyaddr` *address*	Returns the hostname for a packed binary network *address* of a certain address type. By default, *addrtype* is assumed to be IP.
`gethostbyname`	`gethostbyname` *hostname*	Returns the network address given its corresponding *hostname*.

Table continued on following page

Function	Syntax	Description
gethostent	gethostent	Gets the next network host record from /etc/hosts or the system equivalent, returning an empty record when the end of the file is reached.
getlogin	getlogin	Returns the user id for the currently logged in user.
getnetbyaddr	getnetbyaddr *address, addrtype* getnetbyaddr *address*	Returns the net name for a given network *address* of a certain address type. By default, *addrtype* is assumed to be IP.
getnetbyname	getnetbyname *name*	Returns the net address given its corresponding net *name*.
getnetent	getnetent	Gets the next entry from /etc/networks or the system equivalent, returning an empty record when the end of the file is reached.
getpeername	getpeername *socket*	Returns the address for the other end of the connection to this *socket*.
getpgrp	getpgrp *process_id* getpgrp	Returns the process group in which the specified process is running. Assumes current process if *process_id* is not given.
getppid	getppid	Returns the process id of the current process' parent process.
getpriority	getpriority *type, id*	Returns current priority for a process, process group, or user as determined by *type*.
getprotobyname	getprotobyname *name*	Returns the number for the protocol given in *name*.
getprotobynumber	getprotobynumber *number*	Returns the name of the protocol given its *number*.
getprotoent	getprotoent	Gets the next entry from /etc/protocols or the system equivalent, returning an empty record when the end of the file is reached.
getpwent	getpwent	Gets the next entry from /etc/passwd or the system equivalent, returning an empty record when the end of the file is reached.
getpwnam	getpwnam *name*	Gets the password record whose login name field matches the given *name*. Returns an empty record if no match occurs.

Function	Syntax	Description
`getpwuid`	`getpwuid` *uid*	Gets the password record whose user id field matches the given *uid*. Returns an empty record if no match occurs.
`getservbyname`	`getservbyname` *name, protocol*	Returns the port number for the *name*d service on the given *protocol*.
`getservbyport`	`getservbyport` *port, protocol*	Returns the port name for the service *port* on the given *protocol*.
`getservent`	`getservent`	Gets the next entry from `/etc/services` or the system equivalent, returning an empty record when the end of the file is reached.
`getsockname`	`getsockname` *socket*	Returns the address for this end of the connection to this *socket*.
`getsockopt`	`getsockopt` *socket, level,optname*	Returns the specified socket option or `undef` if an error occurs.
`glob`	`glob` *expression* `glob`	Returns a list of filenames matching the regular *expression* in the current directory. If *expression* is omitted, the comparison is made with `$_`.
`gmtime`	`gmtime`	Returns a nine-element integer array representing the given *time* (or `time()` if not given) converted to GMT. By index order, the nine elements (all zero-based) represent: 0 Number of seconds in the current minute. 1 Number of minutes in the current hour. 2 Current hour. 3 Current day of month. 4 Current month.

Table continued on following page

Function	Syntax	Description
`gmtime` (cont.)	`gmtime` *time* (cont.)	5 Number of years since 1900 6 Weekday (Sunday = 0) 7 Number of days since January 1. 8 Whether daylight savings time is in effect.
`goto`	`goto` *tag* `goto` *expression* `goto` *&subroutine*	Looks for *tag* either given literally or dynamically derived by resolving expression and resumes execution of the program there on the provision that it is not inside a construct that requires initializing. For example, a `for` loop. Alternatively, `goto` *&subroutine* switches a call to *subroutine* for the currently running subroutine.
`grep`	`grep` *expression, list* `grep` {*block*} *list*	Evaluates a given *expression* or *block* of code against each element in *list* and returns a list of those elements for which the evaluation returned `true`.

H

Function	Syntax	Description
`hex`	`hex` *string* `hex`	Reads in *string* as a hexadecimal number and returns the corresponding decimal equivalent. Uses `$_` if string is omitted.

I

Function	Syntax	Description
`import`	`import` *module list* `import` *module*	Patches a module's namespace into your own, incorporating the *list*ed subroutines and variables into your own package (or all of them if *list* isn't given).

Function	Syntax	Description
index	index *string, substring, position* index *string, substring*	Returns the zero-based position of *substring* in *string* first occuring after character number *position*. Assumes *position* equals zero if not given. Returns -1 if match not found.
int	int *number* int	Returns the integer section of *number* or $_ if *number* is omitted.
ioctl	ioctl *filehandle, function, argument*	Calls the ioctl function, to use on the file or device opened with *filehandle*.

J

Function	Syntax	Description
join	join *character*, *list*	Returns a single string comprising the elements of *list*, separated from each other by *character*.

K

Function	Syntax	Description
keys	keys *hash*	Returns a non-ordered list of the keys contained in *hash*.
kill	kill *signal*, *process_list*	Sends a *signal* to the processes and/or process groups in process_list. Returns number of signals successfully sent.

L

Function	Syntax	Description
last	last *label* last	Causes the program to break out of the *label*ed loop (or the innermost loop, if *label* is not given) surrounding the command and to continue with the statement immediately following the loop.
lc	lc *string*	Returns *string* in lower case or $_ in lower case if *string* is omitted.
lcfirst	lcfirst *string*	Returns *string* with the first character in lower case. Works on $_ if *string* is omitted.

Table continued on following page

Function	Syntax	Description
`length`	`length` *expression*	Evaluates *expression* and returns the number of characters in that value. Returns `length $_` if *expression* is omitted.
`link`	`link` *thisfile*, *thatfile*	Creates a hard link in the filesystem, from *thatfile* to *thisfile*. Returns `true` on sucess, `false` on failure.
`listen`	`listen` *socket*, *max_connectons*	Listens for connections to a particular *socket* on a server and reports when the number of connections exceeds *max_connections*.
`local`	`local` *var*	Declares a 'private' variable that is available to the subroutine in which it is declared and any other subroutines that may be called by this subroutine. Actually creates a temporary value for a global variable for the duration of the subroutine's execution.
`localtime`	`localtime` *time*	Returns a nine-element array representing the given *time* (or `time()` if not given) converted to system local time. See `gmtime()` for desription of elements.
`log`	`log` *number*	Returns the natural logarithm for a *number*. That is, returns *x* where e^x=*number*.
`lstat`	`lstat` *filehandle* `lstat` *expression* `lstat`	Returns a thirteen element status array for the symbolic link to a file and not the file itself. See `stat()` for further details.

M

Function	Syntax	Description
`m//`	`m//`	Tries to match a regular expression pattern against a string.
`map`	`map` *expression, list* `map` {*block*} *list*	Evaluates a given *expression* or *block* of code against each element in *list* and returns a list of the results of each evaluation.
`mkdir`	`mkdir` *dirname, mode*	Creates a directory called *dirname* and gives it the read\write permissions as specified in *mode* (an octal number).
`msgctl`	`msgctl` *id, cmd, arg*	Calls the System V IPC `msgctl` function.
`msgget`	`msgget` *key, flags*	Calls the System V IPC `msgget` function.
`msgrcv`	`msgrcv` *id, var, size, type, flags*	Calls the System V IPC `msgrcv` function.

Function	Syntax	Description
msgsnd	msgsnd *id, msg, flags*	Calls the System V IPC msgsnd function.
my	my *variable_list*	Declares the variables in *variable_list* to be lexically local to the block or file it has been declared in.

N

Function	Syntax	Description
next	next *label* next	Causes the program to start the next iteration of the *label*led loop (or the innermost loop, if *label* is not given) surrounding the command.
no	no *module_name*	Removes the functionality and semantics of the named module from the current package. Compare with use() which does the opposite.

O

Function	Syntax	Description
oct	oct *string* oct	Reads in *string* as an octal number and returns the corresponding decimal equivalent. Uses $_ if string is omitted.
open	open *filehandle, filename* open *filehandle*	Opens the file called *filename* and associates it with *filehandle*. If *filename* is omitted, open assumes that the file has the same name as *filehandle*.
opendir	opendir *dirhandle,dirname*	Opens the directory called *dirname* and associates it with *dirhandle*.
ord	ord *expression*	Returns the numerical ASCII value of the first character in *expression*.

P

Function	Syntax	Description
pack	pack *template, list*	Takes a *list* of values and puts them into a binary structure using *template* (a sequence of characers as shown below) to give the structure an ordered composition. The possible characters for *template* are:

Table continued on following page

Function	Syntax	Description
`pack` (cont.)	`pack` *template, list* (cont.)	a Null-padded ASCII string A Space-padded ASCII string. b A bit string (low-to-high). B A bit string (high-to-low). c A signed char value. C An unsigned char value. d A double-precision float in the native format. f A single-precision float in the native format. h A hexadecimal string, low to high. H A hexadecimal string, high to low. i A signed integer. I An unsigned integer. l A signed long value. L An unsigned long value. n A big-endian short (16-bit) value. N A big-endian long (32-bit) value. p A pointer to a null-terminated string. P A pointer to a fixed-length string. q A signed quad (64-bit) value. Q An unsigned quad (64-bit) value. s A signed short (16-bit) value. S An unsigned short (16-bit) value. v A little-endian short (16-bit) value. V A big-endian long (32-bit) value. u A uuencoded string. w A BER compressed integer - an unsigned integer in base 128, high-bit first. x A null byte. X Back up a byte.

Function	Syntax	Description
pack (cont.)	pack *template, list* (cont.)	Z A null-padded, null-terminated string. @ Null-fill to absolute position.
package	package *namespace*	Declares that the following block of code is to be defined within the specified *namespace*.
pipe	pipe *readhandle, writehandle*	Opens and connects two filehandles, such that the pipe reads content from *readhandle* and passes it to *writehandle*.
pop	pop *array* pop	Removes and returns the last element (at largest index position) from *array*. Pops @ARGV if *array* is not specified.
pos	pos *scalar*	Returns the position in *scalar* of the character following the last m//g match. Uses $_ for *scalar* if omitted.
print	print *filehandle list* print *list* print	Prints a *list* of comma-separated strings to the file associated with *filehandle* or STDOUT if not specified. If both arguments are omitted, prints $_ to the currently selected output channel.
printf	printf *filehandle format, list* printf *format, list*	As print() but prints to the output channel using a specified *format*.
prototype	prototype *function*	Returns the prototype of a *function* as a string or undef if the prototype does not exist.
push	push *array, list*	Adds the elements of *list* to the *array* at position max_index.

Q

Function	Syntax	Description
q//	q/*string*/	Alternative method of putting single quotes around a string.
qq//	qq/*string*/	Alternative method of putting double quotes around a string.
quotemeta	quotemeta *expression*	Scans through *expression* and returns it having prefixed all non-alphanumeric or -underscore characters with a backslash.

Table continued on following page

Function	Syntax	Description
qw//	qw/*strings*/	Returns a list of strings, the elements of which are created by splitting a *string* by whitespace or the *strings* sent to qw//.
qx//	qx/*string*/	Alternative method of backtick-quoting a *string* (which now acts as a command-line command.

R

Function	Syntax	Description
rand	rand *expression*	Evaluates expression and then returns a random value *x* where 0 <= *x* < the value of *expression*.
read	read *filehandle, scalar, length, offset* read *filehandle, scalar, length*	Reads *length* number of bytes in from *filehandle*, placing them in *scalar*. Starts by default from the start of the file, but you can specify *offset*, the position in the file you wish to start reading from. Returns the number of bytes read, zero if at the end of the file or undef if file doesn't exist.
readdir	readdir *dirhandle*	Returns the next entry in the directory specified by *dirhandle* or if being used in list context, the entire contents of the directory.
readline	readline *filehandle*	Returns a line from *filehandle*'s file if in scalar context or returns a list containing all the lines of the file as its elements.
readlink	readlink *linkname*	Returns the name of the file at the end of symbolic link *linkname*.
readpipe	readpipe *command*	Executes *command* on the command line and then returns the standard output generated by it as a string. Returns a list of lines from the standard output if in list context.
recv	recv *socket, scalar, length, flags*	Receives a *length* byte message over the named *socket* and reads it into a *scalar* string.
redo	redo *label* redo	Causes the program to restart the current iteration of the *label*ed loop (or the innermost loop, if *label* is not given) surrounding the command without checking the while condition.
ref	ref *reference* ref	Returns the type of object being referenced by *reference* or a package name if the object has been blessed into a package.

Function	Syntax	Description
rename	rename *oldname, newname*	Renames file *oldname* as *newname*. Returns 1 for success or 0 if otherwise.
require	require *file* require *package* require *num* require	Ensures that the named *package* or *file* are included at runtime. If *num* is argument, ensure that version of Perl currently running is greater than or equal to *num* (or $_ if omitted).
reset	reset *expression*	Resets all variables in current package beginning with one of the characters in *expression* and all ?? searches to their original state.
return	return *expression*	Returns the value of *expression* from a subroutine or eval().
reverse	reverse *list*	Returns either *list* with its elements in reverse order if in list context or a string consisting of the elements of *list* concatenated together and then written backwards.
rewinddir	rewinddir *dirhandle*	Resets the point of access for readdir() to the top of directory *dirhandle*.
rindex	rindex *string, substring, position* rindex *string, substring*	Returns the zero-based position of the last occurence of *substring* in *string* at or before character number *position*. Returns -1 if match not found.
rmdir	rmdir *dirname*	If directory *dirname* (or that given in $_ if omitted) is empty, it is removed. Returns true on success or false if otherwise.

S

Function	Syntax	Description
s///	s/*matchstring*/*replacestring*/	Searches for *matchstring* in $_ and replaces it with *replacestring* if found.
scalar	scalar *expression*	Evaluates *expression* in scalar context and returns the resultant value.
seek	seek *filehandle, position, flag*	Sets the *position* (character number) in a file denoted by *filehandle* from which the file will be read/written. *Flag* tells seek whether to goto character number *position* (*flag* = 0), number current position + *position* (*flag* = 1) or number EOF + *position* (*flag*=2). Returns 1 on success or 0 if otherwise.

Table continued on following page

Function	Syntax	Description
`seekdir`	`seekdir` *dirhandle, position*	Sets the *position* (entry number) in a directory denoted by *dirhandle* from which directory entries will be read.
`select`	`select` *filehandle* `select`	Changes the current default filehandle (starts as `STDOUT`) to *filehandle*. Returns the current default filehandle if *filehandle* is omitted.
`select`	`select` *rbits, wbits, ebits, timeout*	Calls the system select command to wait for *timeout* seconds until one (if any) of your filehandles become available for reading or writing and returns either success or failure.
`semctl`	`semctl` *id, sem_num, command, argument*	Calls the System V IPC `semctl` function.
`semget`	`semget` *id, semnum, command, argument*	Calls the System V IPC `semget` function.
`semop`	`semop` *key, opstring*	Calls the System V IPC `semop` function.
`send`	`send` *socket, message, flags, destination* `send` *socket, message, flags*	Sends a *message* from a *socket* to the connected socket, or, if *socket* is disconnected, to *destination*. Takes account of any system *flags* given it.
`setpgrp`	`setpgrp` *process_id, process_group*	Sets the *process_group* in which the process with the given *process_id* should run. The arguments default to 0 if not given.
`setpriority`	`setpriority` *which, id, priority*	Adds to or diminshes the priority of either a process, process group, or user, as determined by *which* and specifically identified by its *id*.
`setsockopt`	`setsockopt` *socket, level, option, optional_value*	Sets the *option* for the given *socket*. Returns `undef` if an error occurs.
`shift`	`shift` *array* `shift`	Returns the element at position 0 in array and then removes it from array. Returns `undef` if there are no elements in the array. `Shifts @_` within subroutines and formats or `@ARGV` otherwise if *array* is omitted.
`shmctl`	`shmctl` *id, command, argument*	Calls the System V IPC `shmctl` function.

Function	Syntax	Description
shmget	shmget *key, size, flags*	Calls the System V IPC shmget function.
shmread	shmread *id, variable, position, size*	Calls the System V IPC shmread function.
shmwrite	shmwrite *id, string, position, size*	Calls the System V IPC shmwrite function.
shutdown	shutdown *socket, manner*	Shuts down the *socket* specified in the following *manner*. 0 Stop reading data. 1 Stop writing data. 2 Stop using this socket altogether.
sin	sin *expression* sin	Evaluates *expression* as a value in radians and then returns the sine of that value. Returns the sine of $_ if exressino is omitted.
sleep	sleep *n* sleep	Causes the running script to 'sleep' for *n* seconds or forever if *n* is not given.
socket	socket *filehandle, domain, type, protocol*	Opens a socket and associates it to the given *filehandle*. This socket exists within the given *domain* of communcation, is of the given *type* and uses the given *protocol* to communicate.
socketpair	socketpair *sock1, sock2, domain, type, protocol*	Creates a pair of sockets named sock1 and sock2. These sockets exist within the given *domain* of communcation, are of the given *type* and use the given *protocol* to communicate.
sort	sort *subroutine list* sort *block list* sort *list*	Takes a *list* of values and returns it with the elements after being sorted into an order. If *subroutine* is given, this is used to sort *list*. If *block* is given, this is used as an anonymous subroutine to sort *list*. If neither are given, *list* is sorted by simple string comparisons.

Table continued on following page

Function	Syntax	Description
splice	splice *array, offset, length, list* splice *array, offset, length* splice *array, offset*	Takes *array* elements from index *offset* to (*offest+length*) and replaces them with the elements of *list*, if any. If *length* is removed, removes all the elements of array from index *offset* onwards. If negative, leaves that many elements at the end of the array. If offset is negative, splice starts from index number (maxindex-*offset*). Returns the last element removed if in scalar context or undef if nothing was removed.
split	split */pattern/, string, limit* split */pattern/, string* split */pattern/* split	Takes the given *string* and returns it as an array of smaller strings where any instances in the string matching *pattern* have been used as the delimiter for the array elements. If given, *limit* denotes the number of times pattern will be searched for in string. If *string* is omitted, $_ is split. If *pattern* is omitted, $_ is split by whitespace.
sprintf	sprintf *format, list*	As printf() but prints *list* to a string using a specified *format.*
sqrt	sqrt *expression* sqrt	Evaluates *expression* and then returns the square root of either it or $_ if it was left out of the call.
srand	srand *expression* srand	Seeds the random-number generator.
stat	stat *filehandle* stat *expression* stat	Returns a thirteen-element array comprising the following information about a file named by *expression*, represented by *filehandle* or contained in $_ (by index number). 0 $dev Device number of filesystem 1 $ino Inode number

Function	Syntax	Description
`stat` (cont.)	`stat` *filehandle*	2 `$mode` File mode.
	`stat` *expression*	
	`stat`	3 `$nlink` number of links to the file.
	(cont.)	
		4 `$uid` User id of file's owner.
		5 `$gid` Group id of file's owner.
		6 `$rdev` Device identifier.
		7 `$size` Total size of file.
		8 `$atime` Last time file was accessed.
		9 `$mtime` Last time file was modified.
		10 `$ctime` Last time inode was changed.
		11 `$blksize` Preferred block size for file I/O.
		12 `$blocks` Number of blocks allocated to file.
`study`	`study` *string*	Tells perl to optimize itself for repeated searches on *string* or on `$_` if *string* is omitted.
	`study`	
`sub`	`sub` *subname block*	Declares a *block* of code to be a subroutine with the name *subname*. If *block* is omitted, this is just a forward reference to a later declaration. If *subname* is omitted, this is an anonymous function declaration.
	`sub` *subname*	
	`sub` *block*	

Table continued on following page

Function	Syntax	Description
`substr`	`substr` *string, position, length, replacement* `substr` *string, position, length* `substr` *string, position*	Returns a substring of *string* that is *length* characters long, starting with the character at index number *position*. If given, that substring is then silently replaced with *replacement*. If *length* is not given, `substr` assumes the entire string from *position* onwards.
`symlink`	`symlink` *oldfile, newfile*	Creates *newfile* as a symbolic link to *oldfile*. Returns 1 on sucess, 0 on failure.
`syscall`	`syscall` *list*	Assumes the first element in the *list* is the name of a system call and calls it, taking the other elements of the *list* to be arguments to that call.
`sysopen`	`sysopen` *filehandle, filename, mode, permissions* `sysopen` *filehandle, filename, mode*	Opens file *filename* under the specified *mode* and associates it with the given *filehandle*. *Permissions* is the octal value representing the permissions that you want to assign to the file. If not given, the default is `0666`.
`sysread`	`sysread` *filehandle, scalar, length, offset* `sysread` *filehandle, scalar, length*	Reads *length* number of bytes in from *filehandle*, placing them in *scalar* using the system call `read`. Starts by default from the start of the file, but you can specify *offset*, the position in the file you wish to start reading from. Returns the number of bytes read, 0 if at the end of the file or `undef` if file doesn't exist.
`sysseek`	`sysseek` *filehandle, pos, flag*	Sets the system position for the file denoted by *filehandle*. *Flag* tells `sysseek` whether to goto position number *pos* (*flag* = 0), number current position + *pos* (*flag* = 1), or number EOF + *pos* (*flag* = 2). Returns 1 on success, 0 otherwise.
`system`	`system` *list*	Forks the process that the current program is running on, lets it complete, and then abandons the current program to run the specified system command in *list*. This will be the first element of *list*, and any arguments to it are stored in subsequent list elements.
`syswrite`	`syswrite` *filehandle, scalar, length, offset* `syswrite` *filehandle, scalar, length* `syswrite` *filehandle, scalar*	Writes *length* number of bytes from the *scalar* to the file denoted by *filehandle*, starting at character number *offset* if specified. If *length* is not given, writes the entire scalar to the file.

T

Function	Syntax	Description
tell	tell *filehandle* tell	Returns the current read/write position for the file marked by *filehandle*. If filehadle is not given, the info is given for the last accessed file.
telldir	telldir *dirhandle*	Returns the current readdir position for the directory marked by *dirhandle*.
tie	tie *variable, classname, list*	Binds the named *variable* to package class *classname*, which works on a variable of that type. Passes any arguments (in *list*) to the new function of the class (TIESCALAR, TIEHASH, or TIEARRAY).
tied	tied *variable*	Returns a reference to the object tied to the given *variable*.
time	time	Returns the number of non-leap seconds elpased since Jan 1, 1970. Can be translated into recognizable time values with gmtime() and localtime().
times	times	Returns a four-element list holding the user and system CPU times (in seconds) for both the current process and its child processes. The list is comprised as follows: $user Current process user time. $system Current process system CPU time. $cuser Child process user time. $csystem Child process system time.
tr///	tr/*string1*/*string2*/	Transliterates a string (also known as y///).
truncate	truncate *filehandle, length* truncate *expression, length*	Truncates the file given by *filehandle* or named literally by *expression* to *length* characters. Returns true on success, false if otherwise.

U

Function	Syntax	Description
uc	uc *string* uc	Returns *string* in upper case or $_ in upper case if *string* is omitted.
ucfirst	ucfirst *string* ucfirst	Returns *string* with the first character in upper case. Works on $_ if *string* is omitted.
umask	umask *expression* umask	Returns the current umask and then sets it to *expression* if this is given. The umask is a group of three octal numbers representing the access permissions for a file or directory of its owner, a group and other users, where execute = 1, write = 2, and read = 4. So an umask of 0777 would give all permissions to all three levels of user. 0744 would restrict all except the owner to read access only.
undef	undef *expression* undef	Removes the value of *expression*, leaving it undefined, or else it just returns the undefined value.
unlink	unlink *list*	Deletes the files specified in *list* (or $_ if not given), returning the number of files deleted. (N.B. For Unix users, unlink() removes a link to each file but not the fields themselves if other links to them still exist.)
unpack	unpack *template, string*	The reverse of pack(). Takes a packed *string* and then uses *template* to read through it and return an array of the values stored within it. See pack() for how *template* is constructed.
unshift	unshift *array, list*	Adds the elements of *list* in the same order to the front (index 0) of *array*, returning the number of elements now in *array*.
untie	untie *variable*	Unbinds the *variable* from the package class it had previously been tied to.

Function	Syntax	Description
use	use *module version list* use *module list* use *module* use *version*	Requires and imports the (*listed* elements of the) named *module* at compile time. Checks that module being used is the specified *version* if combined with *module* and *list*. use *version* meanwhile makes sure that the perl interpreter used is no older than the stated *version*.
utime	utime *atime, mtime, filelist*	Sets the access (*atime*) and modification (*mtime*) times on files listed in *filelist*, returning the number of successful changes that were made.

V

Function	Syntax	Description
values	values *hash*	Takes the named *hash* and returns a list containing copies of each of the values in it.
vec	vec *string, offset, bits*	Takes *string* and regards it as a vector of unsigned integers. Then returns the value of the element at position *offset*, given that each element has 2 to the power of *bits* in it.

W

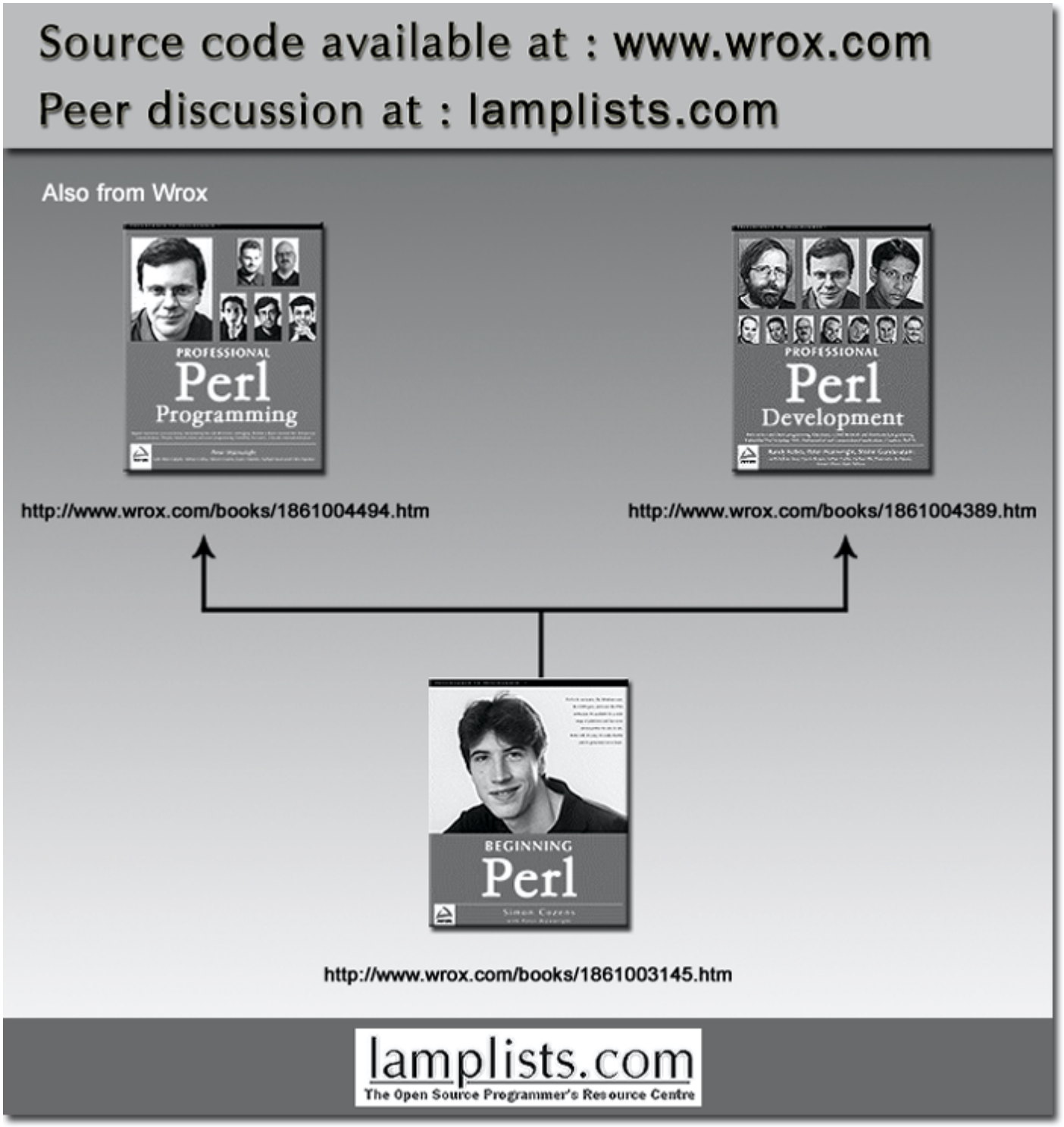
Function	Syntax	Description
wait	wait	Waits for a(ny) child process to die and then returns the process id of the child process that did or -1 if there are no child processes.
waitpid	waitpid *pid, flags*	Waits for the child process with process id *pid* to die

Table continued on following page

Function	Syntax	Description
`wantarray`	`wantarray`	Returns `true` if the subroutine currently running is running in list context. Returns `false` if not. Returns `undef` if the subroutine's return value is not going to be used.
`warn`	`warn` *message*	Prints *message* to `STDERR`, but doesn't throw an error or exception.
`write`	`write` *filehandle* `write` *expression* `write`	Writes a formatted record to *filehandle*, the file named after evaluating *expression*, or the current default output channel if neither are given.

Y

Function	Syntax	Description
`y///`	`y/`*string1*`/`*string2*`/`	Transliterates a string (also known as `tr///`).



D

The Perl Standard Modules

The following appendix lists and describes the standard and pragmatic modules that are installed with **Perl 5.6**. For this reference, they have been ordered alphabetically by group. Note that these modules' names are case sensitive and are given as they should be written in a `use` statement. For more detialed information, you should turn to the module documentation installed with your version of Perl.

Pragmatic Modules

Using pragmatic modules affects the compilation of your perl programs. These modules are lexically scoped and thus to `use` or to uninclude them with `no` like so

```
use attrs;
use warnings;
no integer;
no diagnostics;
```

is effective only for the duration of the block in which the declaration was made. Furthermore, these declarations may be reversed within any inner blocks in the program.

Name of Module	Function
`attributes`	Gets or sets the attribute values for a subroutine or variable.
`attrs`	Gets or sets the attribute values for a subroutine.
	Deprecated in Perl 5.6 in favour of `attributes`.
`autouse`	Moves the inclusion of modules into a program from compile time to runtime. Specifically, it postpones the module's loading until one of its functions is called.
`base`	Takes a list of modules, `requires` them and then pushes them onto `@ISA`. Essentially, it will establish an IS-A relationship with these classes at compile time.

Table continued on following page

Name of Module	Function
blib	Used on the command line as -Mblib switch to test your scripts against an uninstalled version of the package named after the switch.
caller	Causes program to inherit the pragmatic attributes of the program which has called it.
charnames	Allows you to specify a long name for a given string literal escape.
constant	Allows you to define constants as a name=>value pair.
diagnostics	Returns verbose output when errors occur at runtime. This verbose output consists of the message that perl would normally give plus any accompanying text that that error contained in the perldiag manpage. See Chapter 9 for more on diagnostics.
fields	Takes a list of valid class fields for the package and enables them at compile time.
filetest	Changes the operation of the -r -w -x -R -W and -X file test operators.
integer	Changes the mathematical operators in a program to work with integers only and not floating point numbers.
less	Currrently not implemented.
lib	Adds the listed directories to @INC.
locale	Enables\disables POSIX locales for built-in operations.
ops	Restricts potentially harmful operations occuring during compile time.
overload	Allows you to overload built-in perl operations with your own subroutines.
re	Allows you to alter the way regular expressions behave.
sigtrap	Enables some simple signal handlers.
strict	Enforces the declaration of variables before their use. See Chapter 9 for more on strict.
subs	Allows you to predeclare subroutine names.
utf8	Enables\ disables Unicode support. Note that at the time of writing, Unicode support in Perl was incomplete.
vars	Allows you to predeclare global variable names.
warnings	Switches on the extra syntactic error warning messages.

Standard Modules

The standard modules are the group of modules that are installed with your distribution of Perl.

A

Name of Module	Function
AnyDBM_File	Acts as a universal virtual base class for those wanting to access any of the fivetypes of DBM file.
AutoLoader	Works with Autosplit to delay the loading of subroutines into the program until they are called. These subroutines are defined following the __END__ token in a package file.
AutoSplit	Splits a program into files suitable for autoloading or selfloading.

B

Name of Module	Function
B	The Perl compiler module.
B::Asmdata	Contains autogenerated data about Perl ops used in the generation of bytecode.
B::Assembler	Assembles Perl bytecode for use elsewhere.
B::Bblock	Used by B::CC to walk through 'basic blocks' of code.
B::Bytecode	Compiler backend for generating Perl bytecode.
B::C	Compiler backend for generating C source code.
B::CC	Compiler backend for generating optimized C source code.
B::Debug	Walks the Perl syntax tree, printing debug info about ops
B::Deparse	Compiler backend for generating Perl source code from compiled
B::Disassembler	Disassembles Perl bytecode back to Perl source
B::Graph	Compiler backend for generating graph-description documents that show the program's structure.
B::Lint	Module to catch dubious constructs
B::Showlex	Shows the file-scope variables for a file or the lexical variables for a subroutine if one is specified.
B::Stackobj	Helper module for CC backend.
B::Stash	Shows what stashes are loaded
B::Terse	Walks the Perl syntax tree, printing terse info about ops
B::Xref	Compiler backend for generating cross-reference reports
Benchmark	Contains a suite of routines that let you benchmark your code
ByteLoader	Used to load byte-compiled Perl code

C

Name of Module	Function
CGI	The base class that provides the basic functionality for generating web content and CGI scripting. See Chapter 13 for more.
CGI::Apache	Backward compatibility module.
	Deprecated in Perl 5.6.
CGI::Carp	Holds the equivalent of the Carp module's error logging functions CGI routines for writing time-stamped entries to the HTTPD (or other) error log.
CGI::Cookie	Allows access and interaction with Netscape cookies.
CGI::Fast	Allows CGI access and interaction to a FastCGI web server.
CGI::Pretty	Generates 'pretty' HTML code on server in place of slightly less pretty HTML written in the CGI script file.
CGI::Push	Provides a CGI interface to server-side push functionality. For example, as used with channels.
CGI::Switch	Backward compatibility module.
	Deprecated in Perl 5.6.
CPAN	Provides you with the functionality to query, download, and build Perl modules from any of the CPAN mirrors.
CPAN::FirstTime	Utility for CPAN::Config to ask a few questions about the system and then write a config file.
CPAN::Nox	As CPAN module, but doesn't use any compiled extensions during its own execution.
Carp	Provides warn() and die() functionality with the added ability to say in which module something failed and what it was.
Carp::Heavy	Carp guts. For internal use only.
Class::Struct	Lets you declare C-style struct-like complex datatypes and manipulate them accordingly.
Config	Allows access to the options and settings used by Configure to build this installation of Perl.
Cwd	Gets the pathname of the current working directory

D

Name of Module	Function
DB	Programmatic interface to the Perl debugger's API (Application Programing Interface). N.B.: This may change.
DB_File	Provides an interface for access to Berkeley DB versions 1.x. Note that you can access versions 2.x and 3.x of Berkeley DB with this module but will have only the version 1.x functionality.

Name of Module	Function
Data::Dumper	Returns a stringified version of the contents of an object, given a reference to it.
Devel::DProf	A perl code profiler. Generates information on the frequency of calls to subroutines and on the speediness of the subroutines themselves.
Devel::Peek	A debugging tool for those trying to write C programs that interconnect with Perl programs.
Devel::SelfStubber	Stub generator for a SelfLoading module.
DirHandle	Provides an alternative set of functions to opendir(), closedir(), readdir() and rewinddir().
Dumpvalue	Dumps info about Perl data to the screen.
DynaLoader	Dynamically loads C libraries when required into your Perl code.

E

Name of Module	Function
English	Allows you to call Perl's special variables (see Appendix B) by their 'English' names.
Env	Allows you to access the key\value pairs in the environment hash %ENV as arrays or scalar values.
Errno	Exports (to your code) the contents of the errno.h include file. This contains all the defined error constants on your system.
Exporter	Implements the default import method for modules.
Exporter::Heavy	The internals of the Exporter module.
ExtUtils::Command	Contains equivalents of the common UNIX system commands for Windows users.
ExtUtils::Embed	Contains utilities for embedding a Perl interpreter into your C/C++ programs.
ExtUtils::Install	Contains three functions for installing, uninstalling and installing-as-autosplit/autoload, programs.
ExtUtils::Installed	Keeps track of what modules are and are not installed.
ExtUtils::Liblist	Determine which libraries should be used in an install and how to use them and sends its finding for inclusion in a Makefile.
ExtUtils::MM_Cygwin	Contains methods to override those in ExtUtils::MM_Unix when ExtUtils::MakeMaker is used on a Cygwin system.
ExtUtils::MM_OS2	Contains methods to override those in ExtUtils::MM_Unix when ExtUtils::MakeMaker is used on a OS\2 system.

Table continued on following page

Name of Module	Function
ExtUtils::MM_Unix	Contains the methods used by ExtUtils::MakeMaker to work.
ExtUtils::MM_VMS	Contains methods to override those in ExtUtils::MM_Unix when ExtUtils::MakeMaker is used on a VMS system.
ExtUtils::MM_Win32	Contains methods to override those in ExtUtils::MM_Unix when ExtUtils::MakeMaker is used on a Windows system.
ExtUtils::MakeMaker	Used to create makefiles for an extension module.
ExtUtils::Manifest	Utilities to write and check a MANIFEST file
ExtUtils::Miniperl	Contains one function to write perlmain.c, a bootstrapper between Perl and C libraries.
ExtUtils::Mkbootstrap	Contains one function to write a bootstrap file for use by DynaLoader
ExtUtils::Mksymlists	Contains one function to write linker options files for dynamic extension
ExtUtils::Packlist	Contains a standard .packlist file manager.
ExtUtils::testlib	Adds blib/* directories to @INC

F

Name of Module	Function
Fatal	Provides a way to replace functions which return false with functions that raise an exception if not successful.
Fcntl	Loads the libc fcntl.h defines.
File::Basename	Provides functions that work on a file's full path name
File::CheckTree	Allows you to specify file tests to be made on directories and files within a directory all at once.
File::Compare	Compares the contents of two files.
File::Copy	Copies files or directories.
File::DosGlob	Implements DOS-like globbing but also accepts wildcards in directory components.
File::Find	Searches \ traverses a file tree for requested file.
File::Glob	Implements the FreeBSD glob routine.
File::Path	Creates or deletes a series of directories.
File::Spec	Group of functions to work on file properties and paths.

Name of Module	Function
File::Spec::Functions	Support module for File::Spec
File::Spec::Mac	Contains methods to override those in File::Spec::Unix when File::Spec is used on a MacOS system.
File::Spec::OS2	Contains methods to override those in File::Spec::Unix when File::Spec is used on a OS/2 system.
File::Spec::Unix	Methods used by File::Spec
File::Spec::VMS	Contains methods to override those in File::Spec::Unix when File::Spec is used on a VMS system.
File::Spec::Win32	Contains methods to override those in File::Spec::Unix when File::Spec is used on a Win32 system.
File::stat	A by-name interface to Perl's built-in stat() functions
FindBin	Locates the directory holding the currently running Perl program.
FileCache	Allows you to keep more files open than the system allows.
FileHandle	Provides an OO-style implementation of filehandles.

G

Name of Module	Function
GDBM_File	Provides an interface for access and make use of the GNU Gdbm library.
Getopt::Long	Enables the parsing of long switch names on the command line. See Chapter 9 for more on this.
Getopt::Std	Enables the parsing of single-character switches and clustered switches on the command line. See Chapter 9 for more on this.

I

Name of Module	Function
I18N::Collate	Compares 8-bit scalar data according to the current locale. **Deprecated in Perl 5.004.**
IO	Front-end to load the IO modules listed below.
IO::Dir	Provides an OO-style implementation for directory handles.
IO::File	Based on FileHandle, it provides an OO-style implementation of filehandles.

Table continued on following page

Name of Module	Function
IO::Handle	Provides an OO-style implementation for I/O handles.
IO::Pipe	Provides an OO-style implementation for pipes.
IO::Poll	Provides an OO-style implementation to system poll calls.
IO::Seekable	Provides seek(), sysseek() and tell() methods for I/O objects.
IO::Select	Provides an OO-style implementation for the select system call
IO::Socket	Provides an OO-style implementation for socket communications
IO::Socket::INET	Provides an OO-style implementation for AF_INET domain sockets
IO::Socket::UNIX	Provides an OO-style implementation for AF_UNIX domain sockets
IPC::Msg	Implements a System V Messaging IPC object class.
IPC::Open2	Opens a process for both reading and writing
IPC::Open3	Opens a process for reading, writing, and error handling
IPC::Semaphore	Implements a System V Semaphore IPC object class.
IPC::SysV	Exports all the constants needed by System V IPC calls as defined in your system's libraries.

M

Name of Module	Function
Math::BigFloat	Enables the storage of arbitrarily long floating-point numbers.
Math::BigInt	Enables the storage of arbitrarily long integers.
Math::Complex	Enables work with complex numbers and associated mathematical functions
Math::Trig	Provides all the trigonometric functions not defined in the core of Perl.

N

Name of Module	Function
NDBM_File	Provides access to 'new' DBM files via tied hashes.
Net::Ping	Provides the ability to ping a remote machine via TCP, UDP and ICMP protocols.
Net::hostent	Replaces the core gethost*() functions with those that return Net::hostent objects.
Net::netent	Replaces the core getnet*() functions with those that return Net::netent objects.

Name of Module	Function
Net::protoent	Replaces the core getproto*() functions with those that return Net::protoent objects.
Net::servent	Replaces the core getserv*() functions with those that return Net::servent objects.

O

Name of Module	Function
O	This is the generic frontend for the Perl compiler. The backends in the B module group are all addressed with this.
Opcode	Allows you to disable named opcodes when compiling Perl code

P

Name of Module	Function
Pod::Checker	Provides a syntax error checker for pod documents. Note that this was still in beta at the time of publication.
Pod::Html	Pod to HTML converter.
Pod::InputObjects	A set of objects that can be used to represent pod files.
Pod::Man	Pod to *roff converter.
Pod::Parser	Base class for creating POD filters and translators.
Pod::Select	Used to extract selected sections of POD from input
Pod::Text	Pod to formatted ASCII text converter.
Pod::Text::Color	Pod to formatted, colored ASCII text converter.
Pod::Usage	Print a usage message from embedded pod documentation
POSIX	Provides access to (nearly) all the functions and identifiers named in the POSIX international standard 1003.1.

S

Name of Module	Function
Safe	Creates a number of 'safe' compartments in memory in which perl code can be tested and the functions for this testing.
SDBM_File	Provides access to sdbm files via tied hashes.
Search::Dict	Provides function to look for a key in a dictionary file.

Table continued on following page

Name of Module	Function
SelectSaver	Selects a filehandle on creation, saves it and restores it on destruction.
SelfLoader	As Autoloader, works with Autosplit to delay the loading of subroutines into the program until they are called. These subroutines are defined following the __DATA__ token in a package file.
Shell	Allows shell commands to be run transparently within perl programs.
Socket	Imports the defines from libc's socket.h header file and makes available some network manipulation fucntions.
Symbol	Qualifies variable names and creates anonymous globs.
Sys::Hostname	Makes several attempts to get the system hostname and then cahces the result.
Sys::Syslog	Perl's interface to the libc syslog(3) calls

T

Name of Module	Function
Term::Cap	Provides the interface to a terminal capability database.
Term::Complete	Provides word completion on the word list in an array.
Term::ReadLine	Provides access to various 'readline' packages.
Test	Provides a simple framework for writing test scripts
Test::Harness	Implements a test harness to run a series of test scripts and return results.
Text::Abbrev	Takes a list and returns a hash containing the elements of the list as the values and unambiguous abbreviations of each element as their respective keys.
Text::ParseWords	Provides functions for parsing a text file into an array of tokens or an array of arrays.
Text::Soundex	Implementation of the Soundex Algorithm.
Text::Tabs	Works through lines of text replacing tabs with spaces, or if space-saving, replacing spaces with tabs if there are none in the text.
Text::Wrap	Simple paragraph formatter. Takes text, wraps lines around text boundaries and controls the indenting of the text.
Tie::Array	Base class for tied arrays.
Tie::Handle	Base class definitions for tied handles.
Tie::Hash	Base class definitions for tied hashes.

Name of Module	Function
`Tie::RefHash`	Allows you to use references as the keys in a hash if it is tied to this module.
`Tie::Scalar`	Base class definitions for tied scalars.
`Tie::SubstrHash`	Allows you to rigidly define key and value lengths within the hash for the entire time it is tied to this module.
`Time::gmtime`	Object-based interface to Perl's built-in `gmtime()` function.
`Time::Local`	Provides efficient conversion functions between GMT and local time.
`Time::localtime`	Object-based interface to Perl's built-in `localtime()` function.
`Time::tm`	Internal object used by `Time::gmtime` and `Time::localtime`

U

Name of Module	Function
`UNIVERSAL`	The base class for ALL classes (blessed references)
`User::grent`	Object-based interface to Perl's built-in `getgr*()` functions
`User::pwent`	Object-based interface to Perl's built-in `getpw*()` functions