1

Meet Perl

This chapter introduces Perl to the non-Perl programmer. Competent Perl programmers need only skim through this material.

The main objective of this chapter is to provide sufficient Perl to allow readers to comfortably work through the rest of *Programming the Network with Perl*. Another is to show that Perl is a rather special programming technology: interesting, powerful, useful and fun.

Newcomers to Perl are advised to work through a good introductory Perl text. See the *Print Resources* section at the end of this chapter for suggestions.

For the purposes of this chapter, it is assumed that the reader is already a programmer. Veterans of the 'C type' languages may find much of Perl familiar. However, although Perl may *look* a lot like C, its behaviour is oftentimes a little strange, and can consequently be quite unlike C. Fans of other programming languages will, initially, just find Perl to be strange. Perl does not set out to be strange, but it is sometimes so unlike the mainstream programming languages that the differences are seen as strangeness in the eyes of 'traditional' programming folk. Which leads into the first bit of Perl strangeness: *default behaviour*.

1.1 Perl's Default Behaviour

Unlike other programming languages, Perl assumes a lot. It does a lot of things by default, and unless told otherwise, a Perl program will inherit this default behaviour. Think of this as Perl's way of giving programmers something for nothing. This is unlike the vast majority of programming languages, which generally assume nothing, and where nothing is free.

1.1.1 Our first Perl program

A simple example[1] will help to illustrate this default behaviour:

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

while (<>)
{
    print;
}
```

The first line (all that `#!` stuff) looks a little strange, so let us conveniently ignore it, for now. The core of this program is the `while` loop, which is printing something with a `print` statement. Just what is getting printed is not clear, but as the `print` statement is in a loop, the assumption is that the printing is happening over and over again (as long as the condition of the `while` loop is true). But, what is getting printed?

The answer lies in the bit of code that has not been explained yet. No, not the strange first line – that is still too strange to discuss – it is the other bit, the `<>` bit. If you are reading this and saying 'that cannot do much', then welcome to the wonderful world of Perl!

The `<>` is a Perl operator which, unless Perl is told otherwise, hooks a program up to standard input and, when it appears in code, returns a line from standard input to the program. Magically, the line of input comes into the program at the top of the `while` loop, then makes its way (magically, again) to the `print` statement and gets printed!

Now, if a selection of lines is fed to this program, they will all get printed one after the other (remember: the `print` statement is within a loop, so the code keeps executing while some condition is true). In this case, the program keeps going while there are input lines to process. To get lines into the program, pass them in from the keyboard (the *hard* way), or from a file (the *easy* way).

Use the `vi` editor to create a file containing the Perl code as shown above. For want of a better name (and a distinct lack of imagination), let us call this program `first`. At the Linux command prompt, enter the following command to test the program's functionality:

```
perl first /etc/passwd
```

The program should print each line from the `/etc/passwd` system file to the screen, one line at a time, until it runs out of lines.

If something other than the content of the file appears, check to see if the Perl code has been typed in correctly. You can ask Perl to check code for syntax errors using the `-c` command-line switch. That is, typing `perl -c first` will check the `first` program for errors, but *not* run it.

---

[1] Borrowed rather shamelessly from Chapter 2 of Nigel Chapman's book: *Perl: The Programmer's Companion*. See the *Print Resources* section at the end of this chapter.

So, this program will, by default, use the filename from the command-line as standard input to the program. Perl finds the file, opens it, keeps reading from the file one line at a time until there are no more lines to read, then closes the file when the program ends. And all this *behaviour* happens by *default*.

As if that were not enough, the program actually does more. If it is provided with more than one filename on the command-line, as follows:

```
perl first /etc/passwd /etc/inittab .bash_profile
```

it not only prints all the lines from `/etc/passwd` but, when it is done, moves onto `/etc/inittab` and prints all the lines in that file, before moving onto `.bash_profile` and printing any lines contained therein. Again, all this occurs by default, free of charge, no questions asked!

A good question to ask at this point is: while the program is running, just where does Perl put the line that it reads in with <> and then prints out with `print`? We can answer this question after introducing another bit of Perl strangeness: *the default variable*.

1.1.2 Perl's default variable

Meet the default variable `$_`, the most used and abused variable in all of Perl.

The general rule-of-thumb is this: if a piece of Perl code expects a variable to be used and one is not provided, 9 out of 10 pieces of code will use `$_`.

This is exactly what happens with <> and `print` in the `first` program. The code could have been written as follows:

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

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

To re-emphasize, this code is *functionally identical* to the code in the `first` program (albeit, somewhat more explicit).

1.1.3 The strange first line explained

All that remains of the `first` program is an explanation of the strange `#!` line. This line is actually doing two things. As the # symbol in Perl indicates the start of a comment that extends to the end of the current line, the strange line is, first and foremost, a comment that is ignored by the Perl interpreter. Secondly, if the very first character of a file is # followed by a !, then the line takes on special meaning on Linux (and UNIX-like) systems. The `#!` combination tells the command processor to run the command identified by the rest of the line (in this

case, it is the `/usr/bin/perl -w` part), and then send the rest of the file to it as standard input. So, make the `first` program executable on Linux using the following command:

```
chmod +x first
```

and then invoke the program as follows:

```
./first /etc/passwd /etc/inittab .bash_profile
```

Linux will find the Perl interpreter (rather conveniently called `perl`) located in the `/usr/bin` directory and run the contents of the `first` file through it. This is a cool feature of Linux (and UNIX), but of lesser importance if running on Mac OS (prior to X) or one of the Windows varieties.

The `-w` part of the Perl command is a *switch* asking Perl to compile the code with extra warnings enabled. Until you know what you are doing, the advice is to always run Perl with the `-w` switch.

Oh, by the way (and in case you hadn't noticed), Perl statements end with the `;` character, and blocks of code are enclosed in curly braces, { and }. Note, too, that Perl is case sensitive (so be careful). And Perl is an *interpreter*, which means that each time a program is run through Perl, the interpreter scans the code for errors, converts the code to Perl's internal bytecode format, optimizes the bytecode, then runs it. If this all sounds slow, do not worry, in the big scheme of things, it really is not. Once a Perl program is actually running *inside* the interpreter, its performance compares favourably with the traditional compiled languages.

Whew! We are finally done with the `first` program. Which just goes to show that it is sometimes much easier (and shorter) to write a few lines of Perl code than it is to explain them!

1.2 Using Variables in Perl

So far, the only variable seen and used is `$_`, the default variable. Creating containers for variables in Perl is easy. Give the container a name (which is made up of a combination of the letters A-Z, a-z, the digits 0-9 and the underscore character), then precede the name with one of Perl's special variable naming characters, depending on what the variable will be used for:

$ – a *scalar* variable (one of something);

@ – an *array* variable (a collection of somethings, a list);

% – a *hash* variable (a collection of name/value pairs); and

**** – a *referenced* variable (a 'pointer' to something else, *usually* another variable).

1.2.1 One of something: scalars

When looking for a place to put one copy of something within a Perl program, use a scalar. Here are some examples:

```
$greeting = "Welcome to Perl!";
$score = 20;
$score = "Goal!";
$next = <>;
$_the_answer = 42;
$WhoWantsToBeA = 1_000_000;
```

Some readers may be looking at the `$greeting` variable and saying 'that's not one of something, that's three words', but they would only be half right. Yes, it is three words, but they are contained within *one single string*, and Perl refers to this single string with a scalar variable container.

Other readers may find it interesting (even strange) that the variable `$score` is being set to two different types of values – one a number (20) and the other a string ('Goal!'). Assuming that these six lines were in fact a little Perl program, surely the Perl interpreter would complain that the second usage of `$score` causes an error, as `$score` was initially used within a numeric context? On the contrary, Perl does not care what value is assigned to a scalar[2], because Perl has *no real notion of variable types*, at least not like that which readers might be used to in C, C++, Pascal, or Java. So, it is OK (with Perl) to assign seemingly different typed values to the same scalar variable.

Note that variable names can start with and include the underscore character, which can also be used to make literal numbers easier to read – simply place an underscore where it is usual to expect a comma. So:

```
$WhoWantsToBeA = 1_000_000;
```

is equivalent to:

```
$WhoWantsToBeA = 1000000;
```

whereas:

```
$WhoWantsToBeA = 1,000,000;
```

sets the variable `$WhoWantsToBeA` to 1, but readers will have to wait until lists are covered to find out why.

In the code, the `$next` variable is assigned the result of Perl processing the <> operator. What happens is that a line of text from standard input is read into the program and assigned to the `$next` variable. Note that <> on a line by itself within a program does not take a line from standard input and put it into the `$_` variable, as this behaviour only works within loops, as was the case with the

[2]Well, just so long as it is one of something.

`first` program. Be warned: if the <> operator appears on a line by itself within a Perl program, a line *is* read in from standard input, but, as Perl has nowhere to put the contents of the line, it is discarded, never to be seen again (unless read again from standard input). Hence, the use of the `$next` variable in the above example.

1.2.2 A collection of somethings: arrays and lists

Here is an example that shows a relatively standard usage of arrays and lists in Perl:

```
@networks = ('Ethernet', 'Token-Ring', 'Frame-Relay', 'ATM');
```

On the left of the assignment operator (the = symbol), there is an array called `@networks`. Note the prefixed @ character, which indicates that this is an array variable. On the right, there are four network names in the form of a Perl list. The four names are surrounded by single quotes, enclosed in parentheses, and separated by commas. The elements of the list are all 'one of something' scalar values (in this case, *strings*). This single line of code takes the four network names and assigns them to the first four elements of the `@networks` array.

In Perl, array indices start counting from zero, and the first element is actually referred to as element 0. So, `@networks` is a four-element array with elements numbered 0, 1, 2 and 3.

The current size of an array can be determined in one of two ways. Add one to the value of the largest index (using `$#`), or assign the array to a scalar variable:

```
print "The size of the array is: ", ( $#networks + 1 ), "\n";
$ssize = @networks;
print "The size of the array is: $size\n";
```

will print the following:

```
The size of the array is: 4
The size of the array is: 4
```

Looking at the code, the `$size` scalar value is substituted into the part of the double quoted string that gets printed. This process is called *variable interpolation*, and it will be returned to later on in this chapter. The first `print` statement is actually printing a list, made up of a string, an expression, and another string[3].

Arrays in Perl are automatically dynamic, so it is possible to add elements to the array without first having to reserve space for them. Here is how to add more network names into the array:

```
@networks = ( @networks, ( 'FDDI', 'Arcnet' ) );
```

[3]Did you notice this?

The `$#networks` variable now has the value 5, which means there are *six* elements in the array.

Accessing a single, specific array element is accomplished using the standard square bracket notation. The code which follows prints the word `Token-Ring` (followed by the newline character):

```
print "$networks[1]\n";
```

When accessing a single, specific element of an array, we are no longer referring to a collection of somethings, we are instead referring to the single, scalar element located within the array (i.e. one of something). Hence, the use of the `$` prefix in the previous example as opposed to the `@` prefix, which would refer to the entire array.

Surprisingly (or strangely), Perl does not complain when code refers to a single array element with the `@` prefix. The following code will also print the word `Token-Ring` (followed by the newline character):

```
print "@networks[1]\n";
```

Technically, this is a single element *array slice*, and should be avoided in situations like this. Strange? Most definitely. Something to worry about? Probably not. Just be sure to prefix single-element array accesses with `$` and everything will be OK.

When used in a *scalar context*, arrays and lists have a value. An array has a numeric value equal to the number of elements in the array, whereas a list has a value equal to the first element in the list. This helps explain why the following sets `$WhoWantsToBeA` to `1` and not `1,000,000` as might initially be expected:

```
$WhoWantsToBeA = 1,000,000;
```

Scalar context can be forced on an array by use of the inbuilt `scalar` subroutine. This code tells Perl to treat the array as a scalar, even though it really still is an array:

```
print "The size of the array is: ", scalar @networks, "\n";
```

An experienced Perl programmer may frown at the initial technique used when creating the first version of the `@networks` array, which is repeated here:

```
@networks = ('Ethernet', 'Token-Ring', 'Frame-Relay', 'ATM');
```

and could be rewritten as follows:

```
@networks = qw(Ethernet Token-Ring Frame-Relay ATM);
```

This has exactly the same meaning as the initial code, and many Perl programmers prefer it (mainly due to the fact that the latter requires less typing). The `qw` is called the *quoting operator* and is shorthand for 'quote words'.

1.2.3 Hashes

The third Perl variable container is the hash, more formally known as the *associative array*. Hashes are somewhat like arrays, in that they hold a collection of scalar somethings. However, whereas arrays are indexed using numeric values, hashes are indexed using string values (which are also called 'keys' or 'names'). Each string value index has associated with it a scalar value. These associations are often referred to as *name/value pairs*.

Hashes in Perl are prefixed with a % character. Here is a hash which will hold data on the maximum frame size for a collection of popular networking technologies:

```
%net_mtus;
```

To refer to the entire hash, use the % prefixed name. To refer to an individual element, prefix the hash name with the $ character (just as was done when referring to individual array elements). Here is how to add an element (which is referred to as a 'hash entry') into the newly created hash:

```
$net_mtus{'Ethernet'} = 1500;
```

The name (or key) is the word `Ethernet` and has the value `1500` associated with it. Lists can be used to initialize a hash:

```
%net_mtus = ( 'Token-Ring', 4464, 'PPP', 1500, 'ATM', 53 );
```

The list (which should have an even number of elements) is taken to contain a set of name/value pairs. Note that this line of code refers to the entire hash using the % prefix. This is shorthand for the following functionally identical code:

```
$net_mtus{'Token-Ring'} = 4464;
$net_mtus{'PPP'} = 1500;
$net_mtus{'ATM'} = 53;
```

Yet another version of this code takes advantage of the Perl feature which aliases the => symbol with the comma. Additional (entirely optional) white space adds to the readability of this code, and helps to highlight the name/value pairs:

```
%net_mtus = ( 'Token-Ring' => 4464,
              'PPP'        => 1500,
              'ATM'        => 53 );
```

It is often useful to think of => as meaning 'has the value' when working with hashes.

Of note here is that, just like arrays, hashes grow dynamically and automatically in Perl. Also important is the fact that hash keys are (and must be) unique[4].

[4]The same restriction applies to array indices, although there tends to be much less fuss made of this fact.

When a hash entry is created, a value part does not need to be initially specified. The special value undef can be used to set a hash entry (or any variable) to the undefined value:

```
$net_mtus{'SMDS'} = undef;
```

The above code is identical to:

```
$net_mtus{SMDS} = undef;
```

the only difference being that the single quotes are missing around the SMDS. In 'Perl-speak', this is referred to as a *bareword*. If a hash name contains no white-space, the single quotes are generally not required.

A convenient way to clear an entire hash is to set it equal to an empty list:

```
%net_mtus = ();
```

Hashes (and arrays, for that matter) are very useful 'right out-of-the-box'. For the vast majority of the programs in *Programming the Network with Perl*, these inbuilt variable containers are all that is needed. However, on occasion, more complicated data structures help to simplify a solution. The problem with arrays and hashes is that they can only store scalar values. This restriction, on first glance, seems limiting in that it appears no method exists to provide for, say, storing an array in a hash entry, or storing a hash in an array element. This restriction is overcome by the use of references.

1.2.4 References

A reference is a Perl scalar variable container that *refers to* something else. The 'something else' can be one of a number of things, including another scalar, array, hash, subroutine, or Perl object. In this subsection, references to scalars, arrays and hashes are described. References to subroutines and objects will be discussed in later sections.

If a scalar reference refers to an array, the scalar reference can then be, for example, added to an existing hash, creating a hash entry that refers to an array. The entry in the hash is still a scalar value (satisfying the restriction placed on hash values), but as a reference, it now refers to something more complicated than a scalar – in this case, an array.

Creating a reference is very easy. Simply place a \ before the thing to be referenced. Here is code which creates a reference to an existing array, then adds the reference into an existing hash:

```
%networks = ();
@ethernets = qw( Ethernet-II IEEE802.3 IEEE802.3-SNAP );

$e_ref = \@ethernets;

$networks{'Ethernet Standards'} = $e_ref;
```

The `'Ethernet Standards'` hash entry now refers to the `@ethernets` array. It is important to realize that the `@ethernets` array and the `$e_ref` scalar both refer to *the same data*. If the array is changed, then what the scalar refers to also changes. Think of the scalar reference as an *alias* to the array's memory location.

To access the array (referred to in the hash), *dereference* the hash entry:

```
print "The Ethernet standards are: ";
print "@{$networks{'Ethernet Standards'}}\n";
```

As it is known that the `$networks{'Ethernet Standards'}` entry is a reference to an array, prefix the use of the hash entry with the `@` symbol. (The hash entry is also enclosed in an extra pair of curly braces, although this is not strictly required.) Read this code as: 'access the array referred to by the hash entry'.

In the earlier code, the use of the `$e_ref` scalar is redundant. It is equally valid to write the code this way:

```
%networks = ();
@ethernets = qw( Ethernet-II IEEE802.3 IEEE802.3-SNAP );

$networks{'Ethernet Standards'} = \@ethernets;
```

Even this can be shortened, if the sole purpose of having the `@ethernets` array is to create a reference to it within a hash entry. Here is another, equally valid, way to code this:

```
%networks = ();

$networks{'Ethernet Standards'} =
        [ 'Ethernet-II', 'IEEE802.3', 'IEEE802.3-SNAP' ];
```

By enclosing the list of array elements in square brackets, this code creates an *anonymous array* (i.e. one that has no name). The array is then assigned to a hash entry, and Perl is smart enough to use a reference.

Here is some code which shows the creation and use of references to scalars and hashes:

```
$scalar = 42;
$refs = \$scalar;
print 'Both $scalar and $refs have the value: ', ${$refs}, ".\n";

%hash = ( 'Name' => "Paul Barry",
          'Book' => "Programming the Network with Perl",
          'Year' => 2002 );

$array_of_hashes[0] = \%hash;

print "There's a great book called ";
print "${$array_of_hashes[0]}{'Book'} by\n";
print "${$array_of_hashes[0]}{'Name'}, published in ";
print "${$array_of_hashes[0]}{'Year'}.\n";
```

Unfortunately, as shown in this piece of code, the syntax for accessing an individual hash entry within an array of hashes is complex. To yield the string 'Paul Barry' from the array of hashes, the code referred to it as:

```
${$array_of_hashes[0]}{'Name'}
```

which reads (from the inside out): 'take whatever is at element zero of the `@array_of_hashes` array, treat it as a hash, then access the value paired with the `Name` key'. Thankfully, Perl provides an alternative syntax, which can reduce this level of complexity. It is possible to yield 'Paul Barry' like this:

```
$array_of_hashes[0]->{'Name'}
```

or like this:

```
$array_of_hashes[0]{'Name'}
```

When the -> appears between a right and left bracket pair – either]{, }[, }{, or][– its use is not required.

It is also possible to create and use anonymous hashes by enclosing the hash entries in curly braces. Here is another (equally valid) way to populate the first element of the `$array_of_hashes` array:

```
$array_of_hashes[0] =
    { 'Name' => "Paul Barry",
      'Book' => "Programming the Network with Perl",
      'Year' => 2002 };
```

Or, this code would also do:

```
%hash = ( 'Name' => "Paul Barry",
          'Book' => "Programming the Network with Perl",
          'Year' => 2002 );

$array_of_hashes[0] = { %hash };
```

When working with the `%networks` hash of arrays (from earlier), either of the following work, yielding the string 'IEEE802.3':

```
${$networks{'Ethernet Standards'}}[1];
$networks{'Ethernet Standards'}->[1];
$networks{'Ethernet Standards'}[1];
```

Which technique to use is a programmer preference, but be aware that many Perl programmers freely mix their use of each, so it is important to be able to recognize all of them.

To check if some variable is a reference, use the inbuilt `ref` subroutine, which returns the type of reference as a string.

1.2.5 Built-in variables

The Perl interpreter defines (and uses) a large collection of built-in variables. The default variable (`$_`) has already been described. A full list of built-ins can be viewed in the `perlvar` manual page. Type this command at the Linux command-line to page through the material:

```
man perlvar
```

Here is a list of frequently used built-in variables (examples of their use – with explanations – appear throughout *Programming the Network with Perl*).

`$!` – contains the operating system error code after some operation has failed.

`$|` – the auto-flush variable, which when set to 1 switches off buffering when Perl writes to an output filehandle (such as standard output). With auto-flushing off, the output buffer will typically not get flushed until a newline character is written.

`$1, $2, etc.` – the pattern-match variables, created as the result of successful pattern matches and regular expressions.

`$a, $b` – used by the inbuilt `sort` subroutine when doing comparisons.

`$@` – when `eval` is used, `$@` is set upon the return from `eval`.

`@_` – the *default array*, used when processing parameters inside Perl subroutines.

`@ARGV` – contains the list of command-line parameters sent to a program.

`@INC` – lists the series of directories Perl searches when loading (and looking for) add-on modules.

`%ENV` – a hash containing the current contents of the operating systems environment.

`%SIG` – a hash of operating system signals and signal handlers.

1.2.6 Scoping with `local`, `my` and `our`

Variable containers in Perl need not be declared prior to their first use. Perl will automatically create them as needed. As a direct result of this behaviour, all variable containers in Perl are *global* in scope.

It is possible to localize a global variable container to a block of code using the inbuilt `local` subroutine. When `local` is used in a block, at the end of the block the variable will revert to the value it had prior to entering the block within which it appears. Within the block, the variable can be used in any which way. If the block of code invokes a subroutine, the `local` value is visible (not the global) within the invoked subroutine. Due to its strange behaviour, use of `local` tends to be frowned upon nowadays. Consequently, none of the code in *Programming the Network with Perl* uses `local`.

The inbuilt `my` subroutine can be used to give a variable container *lexical scope.* The variable is not globally visible, nor is it visible by any invoked subroutine. It is only visible within the block that declares it. When the `use strict` compiler directive is used at the top of a Perl program, global variables are forbidden, and `my` must be used. That is, of course, assuming the variable containers have not been declared with `our`.

The inbuilt `our` subroutine is new as of release 5.6.0 of Perl. When `our` is used, the variable container can be used as if it is global, even though it really is not. This allows a program to use the `use strict` compiler directive, and still use global variable containers when it is convenient (or necessary) to do so.

1.3 Controlling Flow

Perl has the usual collection of flow control statements: `if`, `while` and `for`. Perl also has some of its own: `unless`, `until` and `foreach`. In addition, two built-in subroutines (`do` and `eval`) can impact on a program's flow of control. In this section, we will look at each of these constructs in turn.

1.3.1 `if`

To decide on one or more courses of action, use the `if` statement. Assuming the `$net` scalar is set to an appropriate value, a simple `if` statement might be:

```
if ( $net eq 'Token-Ring' )
{
    print "The network is of the token passing variety.\n";
}
```

This code prints the message if the condition-part (i.e. the first line) of the `if` statement is true. Veterans of the C-type programming languages need to note that the use of curly braces *is required* by Perl. Two-way decisions are accommodated by the use of an `else` statement:

```
if ( $net eq 'Token-Ring' )
{
    print "The network is of the token passing variety.\n";
}
else
{
    print "The network is NOT a token passer.\n";
}
```

Multi-way decisions are also possible (note the strange spelling of `elsif`):

```
if ( $net eq 'Token-Ring' )
{
    print "Your network is of the token-passing variety.\n";
}
elsif ( $net eq 'Ethernet' )
{
    print "Your network is of the CSMA/CD variety.\n";
}
elsif ( $net eq 'Frame-Relay' )
{
    print "Your network is of the ISDN variety.\n";
}
else    #  Assuming a value of 'ATM' ...
{
    print "Your network is of the cell-switching variety.\n";
}
```

This is the only way to perform a multi-way decision in Perl. C and Java programmers may miss the convenience of the `switch` statement, just as Pascal programmers may miss their `case` statement. To such programmers, Perl offers nothing more than a shrug of apology (and a nod in the direction of code similar to that shown above).

1.3.2 The ternary conditional operator

Perl supports the *ternary conditional operator*. The following `if` statement sets the value of `$do` depending on the current value of `$today`:

```
if ( ($today eq 'Sat') or ($today eq 'Sun') )
{
    $do = "Play";
}
else
{
    $do = "Work";
}
```

and is functionally identical to this (somewhat more compact) code:

```
$do = ( ($today eq 'Sat') or ($today eq 'Sun') ) ? "Play" : "Work";
```

The condition-part of the `if` statement comes before the `?` symbol. If the condition-part is true, the value `Play` results, otherwise the value `Work` results (with the `:` symbol separating the two possible results). The variable `$do` is, therefore, assigned an appropriate value.

1.3.3 while

The `while` statement was part of the `first` program written at the start of this chapter. Like the `if`, curly braces are required with `while` statements.

Code inside the loop (i.e. within the curly braces) keeps executing while the condition-part of the `while` statement is true. A loop can optionally include a `continue` block which will execute at the end of each loop (or iteration).

Three statements provide programmers with the ability to fine tune the behaviour of loops:

next - jump immediately to the bottom of the loop and execute the `continue` block (if one exists), then start a new iteration if the condition-part evaluates to true;

last - exit the loop, bypassing any `continue` block, and resume execution at the first statement immediately following the loop;

redo - abandon the current iteration of the loop, jump to the first statement of the loop, and re-execute the loop statements.

Example uses (with explanations where required) of these statements appear throughout *Programming the Network with Perl*.

The `while` statement is often used to iterate through a hash, as follows:

```
while ( ($name, $value) = each %net_mtus )
{
    print "$name has the value: $value\n";
}
```

The inbuilt `each` subroutine returns a name/value pair from the specified hash, and with each iteration, the next name/value pair is returned until no more pairs are left. Of note is that the name/value pairs come out of the hash in no particular order (and most definitely not in the order that they were inserted).

1.3.4 for

The `for` statement iterates a fixed number of times over code, and is typically used to process arrays. Here is how to print each element of the `@networks` array on its own line:

```
for ($i = 0; $i <= $#networks; $i++)
{
    print "$networks[$i]\n";
}
```

This code initializes `$i` to zero *before* the loop starts to iterate, then checks to see if the value of `$i` is less than or equal to the largest index of the array (`$#networks`). If it is, the loop iterates, then the value of `$i` is incremented *after*

the iteration completes, and prior to the test against `$#networks` is performed again. The loop iterates until the entire array has been printed.

1.3.5 unless

In Perl, the `unless` statement is the opposite of `if`. This code prints the line for values of `$net` other than `Token-Ring`:

```
unless ( $net eq 'Token-Ring' )
{
    print "The network is NOT of the token passing variety.\n";
}
```

The `unless` statement *can* have an `else` part, but no `elsifs`.

1.3.6 until

The `while` statement also has an opposite, the `until` statement. Rather than iterating *while* a condition is true, this statement iterates *until* some condition is true.

1.3.7 foreach

The `for` statement described earlier is used so often to process arrays that a special shorthand version exists, the `foreach` statement. This code is identical to the example used with `for` above:

```
foreach $i (@networks)
{
    print "$i\n";
}
```

But be careful: while inside the loop, the `$i` variable is an *alias* to the actual element within the `@networks` array. If `$i` is changed, so is the corresponding array element.

The `foreach` statement can also be used to iterate over a hash:

```
foreach $name ( keys %net_mtus )
{
    print "$name has the value: $net_mtus{$name}\n";
}
```

Another inbuilt subroutine, `keys`, returns a list of all the name-parts of the named hash. This list (array) is then used normally by the `foreach`. The `keys` subroutine can also be used, in conjunction with `scalar` to calculate the number of elements in any hash:

```
$size = scalar keys %net_mtus;

print "The size of the hash is: $size\n";
```

A common extension to the above hash traversal code is to call the `sort` or `reverse` inbuilt subroutine to force an ordering on the hash name-parts:

```
foreach $name ( sort ( keys %net_mtus ) )
{
    print "$name has the value: $net_mtus{$name}\n";
}
```

1.3.8 do

The do subroutine takes a block of code as its sole parameter, and executes the block of code, returning the value of the last line of the code as the result of the do.

The following example sets the value of `$res` to 15, then prints the value to the screen, and illustrates the basic mechanism:

```
$res = do
       {
           $a = 5;
           $b = 3;
           $a * $b;
       };
print $res;
```

A second form of do takes the name of a file as its sole parameter and executes the contents of the file as a Perl program. Assuming a file called `es1` exists in the current working directory, this code opens the file from within the current program and runs any Perl code contained therein:

```
do 'es1';
```

1.3.9 eval

The `eval` subroutine takes a string as its sole parameter and executes the string as a Perl program (or fragment thereof). This code does the same thing as the do example:

```
$some_code = '$a = 5; $b = 3; $res = $a * $b; print $res;';
eval { $some_code };
```

What makes this `eval` code different[5] from the equivalent do code is that a certain amount of protection is provided by `eval`. Specifically, if the code within the `eval`

[5]Some would say 'better'.

block causes a fatal error (which would normally cause the Perl interpreter to exit immediately from a program), `eval` catches the fatal error and provides a way to recover from the error. The built-in variable `$@` will be set by `eval` on exit from the code block. If the value of `$@` is set to the empty string, no fatal error occurred. However, if the value of `$@` contains a non-empty string, a fatal error occurred and a description of the error is in `$@`. This, therefore, provides a simple, yet highly useful, exception-handling mechanism.

In the code which follows, a call to the inbuilt `die` subroutine causes the generation of a fatal error (and message). When called outside an `eval` block, the program would end immediately. When called inside an `eval` block, the evaluated code ends and the `$@` variable is set to the message from `die`.

Here is an `eval` example:

```
eval
{
    print "This is Apollo 13.  Mission Log.\n";
    print "We are half-way to the Moon.\n";

    die "Houston, we have a problem!\n";
};
if ($@)
{
    print "Message from Apollo 13 - $@";
    print "Let's bring them home safely ... ";
}
else
{
    print "Maybe 13 is not that unlucky after-all ... ";
}
```

Due to the fact that the call to `die` is within the `eval` block, the program can recover gracefully from what would otherwise be a fatal error (and go on to save the astronauts). The `else` part of the `if` statement will never execute in this code, but is included here for illustration only.

1.3.10 Statement modifiers

In addition to the control flow mechanisms which enclose blocks of code, Perl also provides support for *statement modifiers*. With these, individual statements can be qualified. A series of examples will help to illustrate.

This code prints the words 'Hello World' if the value of `$should_we` has some value (i.e. if it is *defined*):

```
print "Hello World" if defined( $should_we );
```

This code prints the words 'Hello World' unless the value of `$today` is `Saturday`:

```
print "Hello World" unless $today eq 'Saturday';
```

This single line of code is another way of writing the `first` program from the start of the chapter. The code takes lines of input from standard input (one at a time) and prints them. It keeps going until there is no more input lines to process:

```
print while (<>);
```

It is possible to use `until` anywhere that `while` is used, and this is also true of statement modifiers. When combined with `do`, statement modifiers provide for loops that will iterate at least once, as the condition-part of the loop comes at the end of the block. Here is the general form of the `do...while` construct:

```
do
{
    # Do something ...
}
while  # some condition is true;
```

and here's the general form of a `do...until`:

```
do
{
    # Do something ...
}
until  # some condition is true;
```

1.4 Boolean in Perl

Strangely, Perl has no inbuilt Boolean type[6]. Instead, any Perl expression can be used as a condition, and can be evaluated to be either true or false. A number of rules help determine whether something is true or false as follows.

Strings – a string is true, unless it contains "0" or the empty string. (Strangely, "00" and "0.00" are true.)

Numbers – a number is true, unless it evaluates to zero. (Strangely, this means −42 is true.)

References – all references are true. (This is initially quite strange until one considers what a reference contains.)

Undefs – anything with the *undefined* value is false. (This is *not* strange – how can anything that is undefined possibly be true?)

Lists – empty lists are false, lists with any number of elements are true. (Like `undef`, this is *not* at all strange.)

[6]Are you not getting used to all this Perl strangeness yet? After a while, it all becomes quite normal. Strange, but true.

1.5 Perl Operators

In a number of the code examples seen so far, various code snippets have relied on condition tests that have used comparison operators. They have been rather sneakily used, without providing any detailed description. Nobody likes a sneak, so here is the entire list of Perl operators (with brief explanations) in precedence order, starting with those with the highest precedence.

-> - the infix dereference arrow operator, used when working with references (and objects).

++ and -- - the increment and decrement operators.

****** - the exponential operator.

!, ~, \, + and - - the logical negation (!), bit-wise negation (~), reference (\), numeric affirmation (+) and arithmetic negation (-) operators.

=~ and !~ - the binding operators (used when working with regular expressions and pattern matches).

***, /, % and x** - the multiply, divide, modulus (%) and repetition (x) operators.

+, - and . - the addition, subtraction and concatenation (.) operators.

<< and >> - the left and right bit shifting operators.

<, >, <=, >=, lt, gt, le and ge - the relational operators. There are two of each, one for working with numbers and the other for working with strings. Be careful to use <, >, <= and >= when comparing numbers, and use `lt`, `gt`, `le` and `ge` when comparing strings.

==, !=, <=>, eq, ne and cmp - the equality operators. As with the relational operators, different versions exist for use with numbers and for strings. The <=> and `cmp` operators are used for comparison, and are typically used in conjunction with the inbuilt `sort` subroutine.

& - the bit-wise AND operator.

| and ^ - the bit-wise OR and eXclusive OR operators.

&& - the logical AND operator.

|| - the logical OR operator.

.. and ... - the range operators.

?: - the ternary conditional operator.

=, **=, +=, *=, &= and so on - the assignment operators.

, and => - the comma operator (typically used to separate list items).

not - a lower precedence alternative to !.

and - a lower precedence alternative to &&.

or and xor - lower precedence alternatives to || and a logical eXclusive OR operator.

1.6 Subroutines

Perl supports the creation of named, user-defined blocks of code, which go by the generic name of 'subroutine'[7]. Creating subroutines could not be easier: give the subroutine a name, pass the name to the inbuilt `sub` subroutine together with the block of code. Here is a simple example:

```
sub simple {
    print "Hello from the simple subroutine!\n";
}
```

The subroutine can now be invoked (from within the program that defined it) in one of four ways:

```
simple;
simple();
&simple;
&simple();
```

It rarely matters which of these techniques is used, as they all do the same thing, i.e. invoke `simple`. When Perl gets picky about a particular style of invocation, the interpreter will tell you.

1.6.1 Processing parameters

Every user-defined Perl subroutine can take any number of parameters. If `simple` was invoked as:

```
simple( "Hey!  Print this!\n" );
```

it would simply ignore the parameter, and Perl would not complain. Any parameters that are sent to a subroutine become available within the *default array*, which is called `@_`. Here is another version of `simple` that can process parameters:

```
sub simple {
    $print_what = $_[0];

    print $print_what;
}
```

where `$_[0]` refers to the first element of the `@_` array. To print the entire array contents, use this code:

[7] Also known as 'function', 'procedure', 'routine' and/or 'method', depending on your programming background.

```
sub simple {
    print "@_";
}
```

If more than one parameter is passed, each becomes available within the subroutine as a separately accessible array element. Let us assume that another version of `simple` supports calls like this:

```
simple( "On line one", "On line two" );
```

To access each 'line' within `simple`, the subroutine could be rewritten to access each element of the default array:

```
sub simple {
    print $_[0], "\n";
    print $_[1], "\n";
}
```

This technique is rarely seen, as most seasoned Perl programmers prefer to take advantage of the inbuilt `shift` subroutine, which removes *and returns* the first element from a named array, or the first element from the default array if no array is specified. Once again, a rewritten `simple` illustrates this technique:

```
sub simple {
    print shift, "\n";
    print shift, "\n";
}
```

It is important to realize that when parameters are passed to any user-defined subroutine, they are 'flattened' into a list. So, if a subroutine is written to expect as parameters a hash, followed by an array, and then a scalar, the three parameters will enter the subroutine as one long, flattened list (which is rarely what was wanted). Pass the hash, array, and scalar as references to ensure they arrive in the subroutine in the correct format. In this case, the `@_` default array will contain three elements: a hash reference, an array reference, and a scalar reference. Note, too, that when accessed inside the subroutine, the references refer to the original variable containers, so if they are changed within the subroutine, they will be changed elsewhere. By default, all parameters are passed *by value*.

1.6.2 Returning results

A subroutine can return a value, and unless explicitly stated, this will be the value of the last statement in the subroutine. The `simple` subroutine would therefore return a true value (assuming the call to `print` was successful). To control what value is returned, use the inbuilt `return` subroutine:

```
sub simple {
    my $count = 0;
    print shift, "\n";
    $count++;
    print shift, "\n";
    $count++;

    return( $count );
}
```

This code will (always) return the value 2.

1.6.3 I want an array

Perl subroutines can be written to return either a scalar or a list. Consider the inbuilt `keys` subroutine introduced during the discussion of hashes. When called in what is known as *list context*, `keys` returns a list of hash name-values. When called in *scalar context*, `keys` returns the number of elements in the hash.

By employing the services of the inbuilt `wantarray` subroutine, it is possible to write a user-defined subroutine that works in a similar fashion:

```
sub return_hash_keys {
    %a_hash = @_;

    return ( wantarray ? keys %a_hash : scalar keys %a_hash );
}

%net_mtus = ( 'Token-Ring' => 4464,
              'PPP'        => 1500,
              'ATM'        => 53 );

@keys = return_hash_keys( %net_mtus );
print "Hash keys: @keys\n";

$size = return_hash_keys( %net_mtus );
print "Hash size: $size\n";
```

The `wantarray` subroutine knows whether the user-defined subroutine has been called in list or scalar context. If called in list context, `wantarray` evaluates to true.

1.6.4 In-built subroutines

Perl has a large collection of inbuilt subroutines. The entire collection is documented in the `perlfunc` online documentation. Use this command to view the manpage:

```
man perlfunc
```

Alternatively, use the `perldoc` program (which comes with Perl) to search the `perlfunc` manpage for documentation on a specific subroutine. For instance, to view the documentation for the inbuilt `print` subroutine, use this command:

```
perldoc -f print
```

The inbuilt subroutines take a varying number of parameters. Check the documentation for specifics. Be aware that some inbuilt subroutines can do different things based on how they are invoked and used. The code in this chapter has already used some of the more popular inbuilt subroutines. Here is an abbreviated list[8]:

alarm - signal an alarm to occur a number of seconds in the future;

chomp - deletes the trailing newline character from a scalar;

chop - deletes the last character from a scalar;

close - close a previously opened filehandle;

defined - returns true if a variable has a value associated with it;

delete - delete elements/entries from an array/hash;

die - exit the current program after displaying a user-specified message;

do - execute a block of statements as one, or read in a collection of statements from another file and execute them;

each - used to iterate over a hash;

eof - test for the end-of-file condition;

eval - evaluate a block of code, and provide exception handling;

exists - returns true if a specific array element or hash entry exists;

exit - exit the current program;

fork - create a child process which is a clone of the current process;

gmtime - return the date and time relative to GMT;

goto - jump to a labelled location within a program[9];

join - join a list of strings together;

keys - returns a list of keys for a specified hash;

last - exit from the current loop;

length - return the length of a scalar variable;

local - localize a variable;

localtime - return the date and time relative to the local time zone;

my - mark a variable as being lexically scoped;

[8]Subroutines of specific interest to network programmers are not presented here, as they are the subject of Chapter 3.

[9]But *real* programmers never use `goto`, do they?

next - start the next iteration of the current loop;

open - open a file, and associate a filehandle with it;

our - declare a global variable;

pack - convert a collection of variables into a string of bytes;

package - declare a new namespace;

pop - treat an array like a stack, and pop the last element off the end of the array;

print - print something (to a named output handle);

printf - print to a particular format;

push - treat an array like a stack, then push an element onto the end of the array;

read - read a specified number of bytes from a filehandle;

redo - restart the current loop iteration;

ref - check to see whether a scalar is a reference, and if it is, return the type of reference as a string;

return - return a value from a subroutine;

scalar - force a list to be treated as if it were a scalar;

shift - treat an array like a stack, and pop the first element off the start of the array;

sleep - pause execution for a specified number of seconds;

sort - sort a list using string comparison order (by default), or by using some user-specified ordering;

splice - remove specified elements from an array;

split - split a delimited string into a list of individual elements;

sprintf - like `printf`, except the result is assigned to a scalar;

sub - declare a subroutine;

substr - extract a substring from a string;

system - call an operating system command, and return its exit status to the calling program;

time - returns the number of non-leap seconds since the operating system's 'epoch'[10];

undef - take a previously defined variable, and undefine it;

unpack - the reverse of **pack**: extract a list of values from a string of bytes;

unshift - treat an array like a stack, and push an element onto the start of the array;

wait - wait for a previously created child process to terminate;

[10] What the operating system thinks is the start of time. It varies from system to system.

wantarray - return true if a subroutine was called within a list context, false otherwise;

warn - sends output to standard error;

write - write a specified number of bytes to a filehandle.

Example uses of these inbuilt subroutines appear throughput *Programming the Network with Perl*. If their meaning is not clear from the context in which they are used, further description is provided.

1.6.5 References to subroutines

Perl allows references to subroutines.

Here is a preview of a code snippet from the end of Chapter 2 which assigns a subroutine reference to a scalar called `$packet_handler` based on the value of another scalar called `$opt_u`:

```
if ($opt_u)
{
    $packet_handler = \&udp_both_packet;
}
else
{
    $packet_handler = \&tcp_both_packet;
}
```

Note the use of the \ character which turns the call to the subroutine into a reference to the subroutine. Later in the code, the previously selected subroutine can be invoked as follows:

```
&$packet_handler;
```

1.7 Perl I/O

Performing input and output (I/O) in Perl is as simple as it gets. Disk files have associated *filehandles*, and each time some input and output is performed, the code need only reference the correct filehandle to work. Filehandles can be opened to read from, write to, or read/write to/from.

Four filehandles are automatically opened for every Perl program: STDIN, STDOUT, STDERR, and DATA. These correspond to standard input (usually the keyboard), standard output (usually the screen), standard error (usually the screen, but sometimes a system log file), and the thoroughly strange 'standard data'. The DATA filehandle is associated with anything that comes after the `__END__` symbol at the end of a source code file, and is generally useful when testing code prior to deployment.

To read from a filehandle, enclose the filehandle in the <> angle brackets. This code reads a line from standard input:

```
$line = <STDIN>;
```

which is longhand for:

```
$line = <>;
```

as Perl will assume standard input by default. Writing to output filehandles usually involves the use of the inbuilt `print` subroutine. Here is how to write to standard output:

```
print STDOUT "Writing to standard output, usually the screen.\n";
```

which is longhand for:

```
print "Writing to standard output, usually the screen.\n";
```

as Perl will assume standard output by default. To write to standard error, use code similar to this:

```
print STDERR "Writing to standard error.\n";
```

or use the inbuilt `warn` subroutine which will always write to STDERR:

```
warn "Writing to standard error.\n";
```

To create filehandles, give them a name and associate them with a file. By convention, filehandle names are specified in uppercase, although Perl does not enforce this as a rule. To open a file for reading, use code similar to the following:

```
open MYINFILE, "readme.txt" or die "Could not open: $!";
```

This code will open `readme.txt` and associate it with the `MYINFILE` filehandle, or if something goes wrong, exit with an appropriate error message.

Once a filehandle is opened, read from it using the <> operator:

```
$a_line = <MYINFILE>;
@entire_file = <MYINFILE>;
```

Here the <> operator is used in both scalar and list context. Be careful when using <> in list context, as reading a large file in one go can result in memory problems.

To close a filehandle, call `close` on the filehandle name:

```
close MYINFILE;
```

which Perl will do anyway when your program ends.

To write to a file, use code like this:

```
open MYOUTFILE, ">readme.txt" or die "Could not open: $!";

print MYOUTFILE "Writing to readme.txt\n";

close MYOUTFILE;
```

Note the > symbol before the name of the file to open for writing. To append to a file, replace the first line above with this line (note the use of the >> symbol):

```
open MYOUTFILE, ">>readme.txt" or die "Could not append: $!";
```

To open a file for reading and writing, use this line:

```
open MYOUTFILE, "+<readme.txt" or die "Could not read/write: $!";
```

To open a file for writing then reading, use this line:

```
open MYOUTFILE, "+>readme.txt" or die "Could not write/read: $!";
```

But be careful with this, as the +> form will first delete the file if it already exists[1], then allow the program to write to the *newly emptied* file and read from it.

1.7.1 Variable interpolation

Some readers may have noticed two differing uses of the inbuilt `print` subroutine. Here are some lines of code repeated from an earlier example:

```
print 'Both $scalar and $refs have the value: ', ${$refs}, ".\n";

        .
        .
        .

print "There's a great book called ";
print "${$array_of_hashes[0]}{'Book'} by\n";
```

On occasion, the string that gets printed by `print` is enclosed in single quotes, and at other times is enclosed in double quotes. The reason for the two styles has to do with what happens to variable containers when used within each quote type. The usage rule is very straightforward: when you want to print something literally, use single quotes, when you want to include actual variable container values in the output, use double quotes. The process of including variable containers in double quoted strings is called *interpolation*. A short example will illustrate the difference:

[1] Known, rather affectionately, as 'clobbering'.

```
$mynet = "Ethernet";
$yournet = "Token-Ring";

print '$mynet is incompatible with $yournet\n';
print "\n$mynet is incompatible with $yournet\n";
```

will print the following:

```
$mynet is incompatible with $yournet\n
Ethernet is incompatible with Token-Ring
```

The first `print` statement outputs the string in its literal form, including the newline *escape code* at the end of the line. The second `print` statement interpolates the `$mynet` and `$yournet` scalars, as well as the newlines at the start and end of the string.

1.8 Packages, Modules and Objects

Perl supports the creation and use of *namespaces*, which are nothing more than a place to put, group and organize a program's variable containers and subroutines. If no namespace is specified in a program, all the variables belong to the *default namespace*, which is called `main`. Use `package` to create namespaces:

```
$ns = 30;

package MyNameSpace;

$ns = 100;
print 'The value of $ns is: ', $ns, "\n";

package main;

print 'The value of $ns is: ', $ns, "\n";

package MyNameSpace;

print 'The value of $ns is: ', $ns, "\n";
```

which produces the following results:

```
The value of $ns is: 100
The value of $ns is: 30
The value of $ns is: 100
```

As each namespace has its own place to put things, this code actually has two different variable containers called `$ns`, one in each of the namespaces. The *fully qualified name* of each container is `$main::ns` and `$MyNameSpace::ns`.

1.8.1 Modules

When a `package` statement, together with whatever code goes with it, is placed in a separate file, it becomes a *module* that can be used by other programs. If the `MyNameSpace` package from above was in a separate file (called `MyNameSpace.pm`), the following code would import its variables and code into the current program:

```
use MyNameSpace;
```

1.8.2 Objects

Modules form the basis of Perl's object-oriented capabilities (as well as being Perl's main code-reuse technology). The code for a Perl *class* is, by convention, placed in a module file. Objects in Perl are, surprisingly, no more than a special type of reference (in effect, a reference that has been marked as containing an object by an inbuilt subroutine called `bless`). It is beyond the scope of this chapter to describe the mechanics employed in creating Perl classes[12]. However, every Perl programmer needs to know how to use objects created from the use of object-oriented modules.

Let us pretend that some friendly programmer has created a truly useful module for use with Perl, and that the module uses an object-oriented interface. Here is how code might `use` the module then access its variable containers and subroutines (which are known as 'methods' in OO speak):

```
use ReallyDead;

$my_cool = ReallyDead->new( First => 'Elvis', Last => 'Presley' );

print "$my_cool->{First} $my_cool->{Last} really is dead.\n";
```

The code uses the `ReallyDead` module, then calls the modules *constructor*, which in this case is called `new` (but could just as easily have been called anything, as using `new` is a convention). The constructor creates a new `ReallyDead` object (passing two named arguments), which will then be referred to by the `$my_cool` variable container. Remember, objects in Perl are a type of reference, so objects are stored in scalars.

To access the data encapsulated within the object and call any subroutines associated with the object, the infix dereference arrow operator is used to refer to the *instance data* and methods. The code above, assuming the `ReallyDead` module actually existed, would print the following:

```
Elvis Presley really is dead.
```

[12]However, we will learn a little about creating Perl objects in Chapter 6, *Mobile Agents*.

An alternative invocation technique exists when working with Perl objects, and is known as the *indirect object syntax*. The call to the new constructor above could have been written as:

```
$my_cool = new ReallyDead First => 'Elvis', Last => 'Presley';
```

and some Perl programmers prefer this technique.

1.8.3 The joy of CPAN

The module and object creation technology in Perl is useful. So useful, in fact, that a large collection of third-party add-on modules have been developed for use with Perl. Some of these modules come with Perl and are part of the Perl distribution. These are the *standard modules*. Others are not part of the standard distribution, but are instead made freely available on a central website to anyone with a use for them. This central website[13], is known as CPAN, the Comprehensive Perl Archive Network.

CPAN is a truly wonderful place (if you are a Perl programmer, that is). It is a large software repository of reusable Perl modules, of the object-oriented and functional kind. On CPAN, the modules are organized by category, and cover every conceivable programming activity, from working with database systems to processing digital images. Of particular interest are the modules which provide support to the network programmer.

In addition to the standard networking modules (which come with Perl), a growing collection of third-party networking modules can be found in the following CPAN categories.

- Networking, Device Control and Inter-process Communication.
- Authentication, Security and Encryption.
- World Wide Web, HTML, HTTP, CGI and MIME.
- Server and Daemon Utilities.
- Mail and Usenet News.

When creating modules for use with Perl, one simple rule needs to be adhered to: *do not reinvent the wheel, check CPAN!* If the exact module required does not already exist, a close match may. Take a copy of the close match, make it perform the required way, then resubmit the newly modified module back to CPAN. The entire Perl community benefits as a result of this practice.

Throughout *Programming the Network with Perl*, numerous third-party modules from CPAN are used. Installation instructions are supplied for each third-party module employed. Refer to the *Web Resources* section at the end of this chapter for more information on the CPAN website.

[13]Which is actually mirrored extensively on the Internet.

1.9 More Perl

There is an awful lot more to Perl than is presented in this chapter. The intent in this chapter has been to cover only the essentials necessary to support the program code in the remainder of *Programming the Network with Perl*. Of the pieces of Perl not covered, the integrated *regular expression* technology is by far the most important[14]. This technology is often referred to as *pattern matching*, and is thought of by many as Perl's programming language within a programming language.

All of the code in this chapter can be classified as being *functional* in nature. The code specifies *how* Perl is to solve the problem, stating exactly what Perl should do. With regular expressions, Perl can also be programmed in a *declarative* way. The code specifies *what* is required, then leaves it to Perl to work out how to do things. Perl's regular expression technology is very powerful, and is one of the main reasons for Perl's popularity. It is especially useful when working with text data.

A treatment of Perl's regular expression technology is not appropriate for an introductory chapter. However, at some stage, every Perl programmer needs to master its capabilities. Refer to the *Print Resources* section at the end of this chapter for suggested texts. Certain regular expressions are used in later chapters, and when they are, they are accompanied by appropriate explanation.

1.10 Where To From Here?

Trying to cover all of Perl in just one chapter was never going to be easy. The classic Perl text, *Programming Perl*, runs to well over one thousand pages! The great thing about Perl is that any level of proficiency is acceptable within its programming community. There is more than enough material in this chapter to allow the reader to understand the rest of the programs in *Programming the Network with Perl*. And, of course, there is plenty of additional material available elsewhere, should it be required. Refer to the *Print Resources* and *Web Resources* sections below for some guidelines on where to start your search.

As the authors of *Programming Perl* advise at the end of their first chapter: *have the appropriate amount of fun.*

1.11 Print Resources

As a rule, I recommend the following book to programmers wishing to make the move to Perl: *Perl: The Programmer's Companion* by Nigel Chapman (Wiley,

[14] And totally *strange*, if regular expressions are new to you.

1997)[15]. This book contains an excellent and highly readable treatment of Perl's regular expression technology.

Additionally, every Perl programmer should have the following books in their collection.

Programming Perl, 3rd edn, by Larry Wall, Tom Christiansen and Jon Orwant (O'Reilly, 2000). This book is known among all Perl programmers as *The Camel*, and is the ultimate reference text for Perl.

Perl Cookbook, by Tom Christiansen & Nathan Torkington (O'Reilly, 1998). The title speaks for itself. The signal-handling code used at the end of Chapter 2 is based on a technique described in *Perl Cookbook*, and many other code snippets draw on material from this book.

Finally, when developing large-scale applications in Perl, it is advisable to master the object-oriented capabilities of the language. The following book is another must have: *Object-Oriented Perl*, by Damian Conway (Manning, 1999).

1.12 Web Resources

`http://www.perl.com` – The home of the Perl community, the Perl website. Always start here when looking for something Perl related.

`http://use.perl.org` – The Perl gossip site.

`http://www.perl.org` – The Perl advocacy site. This is also the home of the *Perl Mongers*.

`http://www.cpan.org` – The official location of CPAN, although nearly every Perl website has a link to it. Also useful is the `http://search.cpan.org` website.

`http://www.perldoc.com` – The Perl 5.6 online documentation as a searchable website.

`http://www.tpj.com` – The quarterly magazine of the Perl community, *The Perl Journal*, maintains a Web presence at this address.

[15] The fact that Chapman's book is published by the same publisher as *Programming the Network with Perl* is purely coincidental. Honest.

Exercises

1. If you have not done so already, run all of the code examples from this chapter through Perl in order to convince yourself that they work the way you expect them to.

2. Run the `first` program through the `Deparse` module with the following command: `perl -MO=Deparse first`. Can you explain the output generated? If not, type `man B::Deparse` at the Linux command-line to learn about this module. Remember this module's existence when debugging your Perl programs.

3. Type `man perldebug` to learn about the inbuilt Perl debugger. Run the `first` program through the debugger.

4. Use any Web browser to surf to your nearest CPAN archive and download the `Devel::Coverage` module[16]. Read any documentation associated with the module, then install `Devel::Coverage` into Perl. Use the facilities of this module to perform a *coverage analysis* on a Perl program of your choosing, preferably one that you have written.

[16]You may have to search for the module first. Start at `http://search.cpan.org`.