Learning To Program With Perl

An introduction to the Perl programming language for those who haven't programmed before

Version 1.1

Licence

This manual is © 2007-14, Simon Andrews.

This manual is distributed under the creative commons Attribution-Non-Commercial-Share Alike 2.0 licence. This means that you are free:

- to copy, distribute, display, and perform the work

- to make derivative works

Under the following conditions:

- Attribution. You must give the original author credit.

- Non-Commercial. You may not use this work for commercial purposes.

- Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a licence identical to this one.

Please note that:

- For any reuse or distribution, you must make clear to others the licence terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.

Full details of this licence can be found at
http://creativecommons.org/licenses/by-nc-sa/2.0/uk/legalcode

Introduction

For a long time Perl has been a popular language among those programming for the first time. Although it is a powerful language many of its features mean make it especially suited to first time programmers as it reduces the complexity found in many other languages. Perl is also one of the world's most popular languages which means there are a huge number of resources available to anyone setting out to learn it.

This course aims to introduce the basic features of the Perl language. At the end you should have everything you need to write moderately complicated programs, and enough pointers to other resources to get you started on bigger projects. The course tries to provide a grounding in the basic theory you'll need to write programs in any language as well as an appreciation for the right way to do things in Perl.

Section 1: Getting Started with Perl

What is Perl / perl?

Perl is a high-level programming language. It is an interpreted language which means that your programs just consist of plain text code – there's no separate compiling step needed to run your programs. Perl is designed to be flexible and easy to use, it is a language whose main purpose is to get things done. The time it takes to write a solution to a problem in Perl is usually MUCH quicker than if you'd had to do the same thing in C / C++ / Java.

Perl is *not* PERL! It is not an acronym (despite what a lot of people will tell you), it is also not perl. Perl is the name of the language, whilst perl is the name of the interpreter you need to run a Perl program (you run your Perl program by passing it to perl :-).

Good things about Perl
- It's free
- It works on pretty much all computers
- It's easy to write
- There are a huge number of pre-written scripts available for most common tasks
- It allows you to develop programs in a short amount of time

Bad things about Perl
- Its flexibility means that in some situations it can be slower than other languages
- Its flexibility means that bad people can write shocking looking code!
- It's mostly command line rather than GUI focussed.

How to install perl

On Linux/Unix/MacOSX etc.
Perl (sorry, perl) comes installed on pretty much every unix-based operating system there is. Perl scripts are widely used by systems administrators, and most unix derivatives won't function without perl installed. If you want a newer version of perl then you can get this from www.perl.com and compile it yourself, but there's usually no need for that.

On Windows
Although you can download the source code for perl and compile it under windows this would require you to have a C compiler installed (Windows doesn't come with one by default), so the easiest way to get a perl installation is to get a pre-compiled version.

The most commonly used pre-packaged perl distribution for windows comes from a company called ActiveState and is known as ActivePerl. You can download ActivePerl (for free) from http://www.activestate.com/activeperl .

How to tell if you have perl installed, and which version
If you're not sure whether you have perl installed on the computer you're working on you can easily find out. First you need to get a command prompt. If you're using unix/Linux you probably know how to get a shell prompt already but if not, try right-clicking on your desktop and it's probably one of the options there. For Macs you use Applications → Utilities → Terminal.
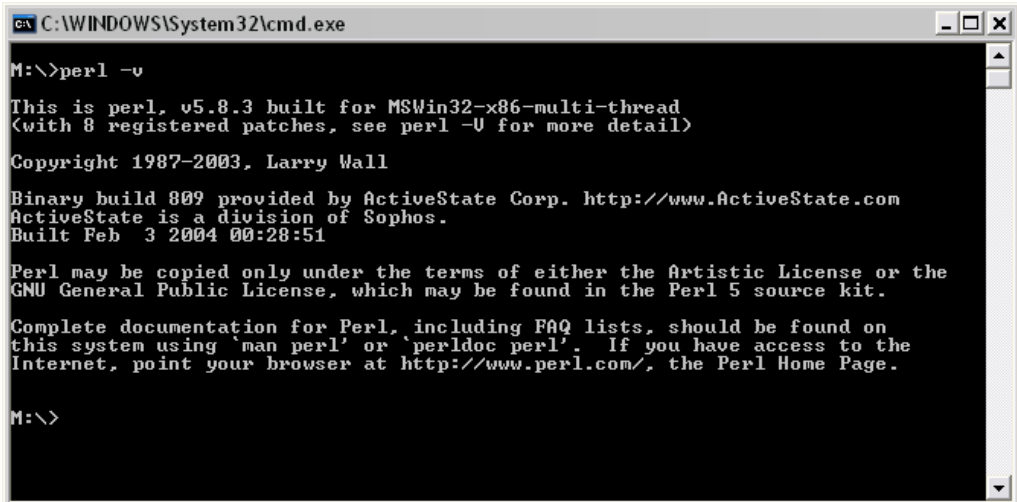
For windows you should try one of the following:

1) Look in Start > Programs for an entry called "MS-DOS Prompt"

2) Look in Start > Programs > Accessories for an entry called "Command Prompt"

3) Go to Start > Run. In the box type "cmd" and press return

Hopefully one of these will get you a command prompt.

At the command prompt type in the command

```
perl -v
```

If perl is installed you should see something like this:

Using Perldoc – the Perl help system

One of the first things you'll need to know is where to go to find the documentation for perl. This is actually distributed with perl itself, so if you have perl installed you already have all the documentation you could ever want.

To access the perl documentation you use the "perldoc" utility. You run this from a command prompt in the same way as if you were going to run a Perl program.

If you don't know where to start, you should try:

```
perldoc perl
```

This lists the other options for perldoc. There are several introductory documents listed which provide introductions to the main areas of functionality in Perl. If you're new to a topic then these guides are well worth reading.

For more specific help there are a couple more ways of launching the perldoc command which may provide more useful:
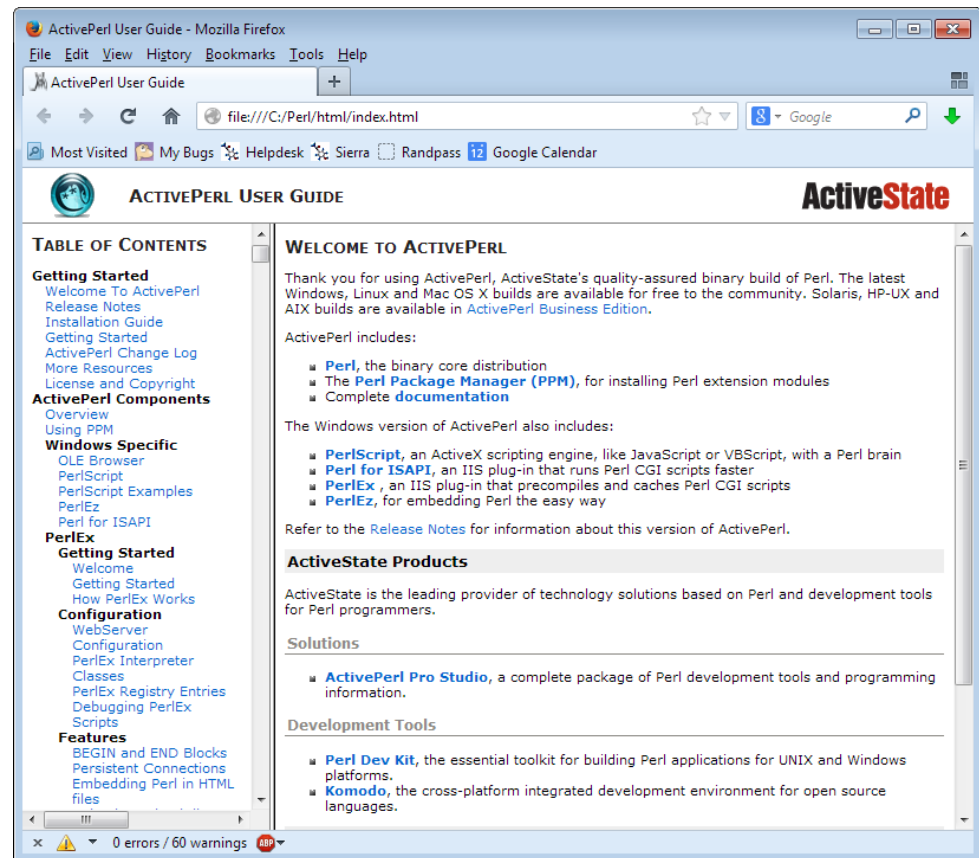
```
perldoc -f XXXXX   This gives the documentation for the function XXXX
```

```
perldoc -q XXXXX    This searches the Perl FAQ for the keyword XXXX
```

If you're using the ActiveState version of perl then the documentation also comes as HTML files. You can access these from Start > Programs > ActivePerl > Documentation.

Using JEdit to write Perl

At a basic level Perl programs are just text files, and you can use any kind of a text editor to write them. On a more practical note however it is extremely useful to use an editor which helps you out as you write your code rather than just playing dumb and letting you do whatever you want.

There are a number of text editors which are specifically designed to be used for writing code and each has their supporters. Choosing an editor can get to be a bit like choosing a religion and as long as you've found something you like then that's OK, just be aware that there are better alternatives than trying to write code in MS Word!

The suggested editor for this course is JEdit. This is a cross-platform (so both Mac and PC) code editor which support lots of languages including Perl. You can use it simply as a text editor which understands code syntax, but it contains lots of more advanced features which may be useful as you progress.
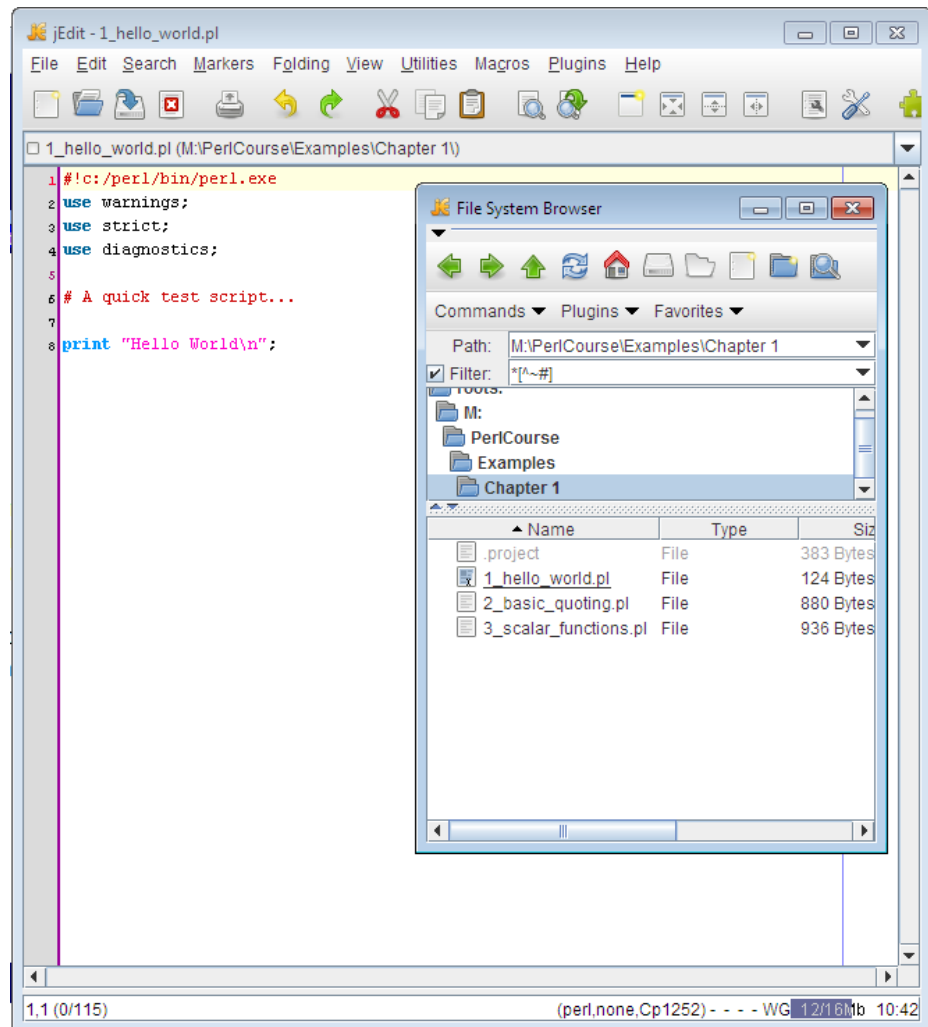
To start writing a new Perl script you simply select File > New in Mode from the main menu. From the list of available languages select 'perl' (you can press 'p' to jump to approximately the correct place. Once you've done that then you can start writing.

Most of the operation of the program is straight forward (saving files, selecting text, copying, pasting etc). One additional feature which is useful is the File System Browser (Utilities > File System Browser). This is an extra window you can open to allow you to quickly switch between different perl programs you're working on.

You can see that the editor actually understands the Perl language and will colour your text to show useful pieces of information in your program which should help you spot when things are going wrong.

Your first Perl script

By tradition, the first program you should write when you're learning a new language is one which prints the words "Hello World" on the screen, and then exits. It's surprising how much you can learn about a language just from being able to do this.

Our hello world script is called hello_world.pl and is shown below. Perl programs don't have to be named with a .pl extension but you will need to name them like this for windows to recognise that they're Perl scripts. It's also useful to keep this convention just so you can tell what your files are just by looking at the name.

In the script below I've added line numbers at the start of each line. These aren't part of the program, they're just there so I can refer back to them later on.

```
1 #!c:/perl/bin/perl.exe
2 use warnings;
3 use strict;
4 use diagnostics;
5
6 # A quick test script...
7
8 print "Hello World!\n";
```

To run this script use the "cd" command in you command shell to move to the directory where you created it, and then type:

```
perl hello_world.pl
```

You should see the words "Hello World!" printed to the screen before your command prompt returns to allow you to enter more commands.

So how does it work?

Now you've seen a working Perl script, let's go through it so we can see how it works.

The first line in a Perl script should always be a pointer to the location of the perl interpreter you'd like to use to run the script. This is mostly only used on unix-like systems, but it's good practice to include it even on windows-based scripts. The syntax for the line is #! (pronounced "hash – bang), followed by the path to perl.

From this point on your program is just a set of Perl statements. Each statement is usually put on a separate line, but is always terminated by a semi-colon. Perl doesn't care how your code is laid out – it could all be on one line as far as it's concerned, but it will make your code much more readable if it's organised sensibly.

Unless instructed otherwise the perl interpreter will start at the top of your file and keep executing statements until it gets to the bottom, at which point it will exit.

Lines 2-4 tell the program that you'd like to introduce some extra functionality into your program from an external file called a Perl Module. Modules are a way of easily being able to extend the base Perl language, and we'll talk a lot more about them later. For now, all you need to know is that:

Lines 2 and 3 are the Perl equivalent of fastening your safety belt. Perl, by default lets you get away with some really bad programming practices. This is because many people use Perl as a command line tool where they write a whole program in a single command, and to do this you need to save as much space as possible. The programs we'll be writing though aren't constrained by space and will therefore not cut corners and will be done properly!

Line 4 is useful when you're starting out with perl and can be omitted later on once you've been using it for a while. The effect of including this line is that if your program encounters an error you would usually just get a terse message pointing out what went wrong. By including the diagnostics module you also get a longer more friendly explanation of what this might mean. [On some macs we've found that 'use diagnostics' doesn't work unless you have the mac developer tools installed so if you get an error about this line just comment it out until you can install these]

Line 6 is a comment. If you put a hash (#) into a Perl script then everything from that point on up to the end of the line is ignored by the perl interpreter. Perl does not have separate syntax for multi-line comments. It's generally a good idea to include comments in your code to help explain the reasoning around a particular piece of code.

Line 8 is where the work actually happens. It uses the print function to print the text "Hello World!" to the screen. The "\n" at the end of the text indicates that perl should print a newline character (equivalent to pressing return).

Scalars and Scalar variables

The first thing we're going to look at in Perl is how it stores and manipulates data. In our hello world script we've actually already used some data – the string "Hello World!\n". If we'd changed that data then our program would have printed something different.

If we had some data we wanted to use in several places in our program, rather than typing it out each time we can store it in a variable. A variable is simply a way of associating some data with a short name which you can use to refer back to it later.

The Perl data structure which is used to hold a single item of data (such as a piece of text, or a number) is called a scalar. A variable which can store a piece of scalar data is called a scalar variable.

Scalar variables in Perl have names which start with a dollar sign, and then have a name which consists of letters, numbers and the underscore character. By convention they are usually put in all lowercase. Examples of typical variable names would be;

```
$x
$name
$first_name
```

Unlike a lot of other languages, Perl does not have a separate data type to hold characters, strings, integers and floating point numbers. The scalar variable type can hold all of these and perl will automatically convert them to the right kind of data depending on the context in which you use it (but as long as you have your safety belt fastened it will warn you if you try to do something stupid like "hello world"+3!).

Assigning values to scalars

To create a new scalar variable you use the syntax shown below;

```perl
my $first_name = "Simon";
```

When you want to create a new variable you need to use the keyword "my" in front of the variable name. This tells the parser that you know that you're creating a new variable, and allows it to catch problems which occur from spelling mistakes such as the one below;

```perl
my $first_name = 'Simon';
$frist_name = 'Bob';
```

If you tried to run this code you'd get the error shown below;

```
Global symbol "$frist_name" requires explicit package name at line 7.
Execution aborted due to compilation errors.
```

Declaring a new variable also sets up the variable's 'scope', that is it defines which parts of the program can see the variable you have created – we'll come back to this topic in a later chapter.

Quoting

When you're assigning a value to a variable you may need to "quote" the data you're assigning so that perl knows that it's data and not a function or variable name. In general, text needs to be quoted and numbers can be entered directly.   You quote data by adding quote marks around it (either "xxx" or 'xxx') and the types of quotes you use act in a slightly different way.

Data contained in single quotes is interpreted literally. Whatever characters you put between the quotes go into your variable.

```perl
my $var = 'This is some $text';
print $var;
```

This would produce - This is some $text – on the command line when run.

If you use double quotes instead however, then certain parts of what you quote will be substituted for something else. The data in double quotes are said to be "interpolated". There are two kinds of substitution which happen in double quotes, variable interpolation and the expansion of special characters.

Below you can see an example of variable interpolation.

```perl
my $name = 'Simon';
my $message = "Hello, my name is $name";
print $message;
```

In this case what you would see printed is - Hello, my name is Simon. By using double quotes the $name in the message will be substituted with the data contained in the $name variable. Of course the example above shows and unnecessarily long way of doing this, and we could just do the interpolation in the print statement.

```perl
my $name = 'Simon';
print "Hello, my name is $name";
```

Special characters are those which you might want to include in data, but which you can't type on a keyboard without doing other strange things to your program. The two most used special characters are the tab and the newline characters.

Special characters in Perl are single letters preceded by a backslash. The example below shows the use of both a tab and a newline.

```perl
print "1\t2\t3\nA\tB\tC\t\n";
```

This produces the output shown below, where the "\t" has been substituted for a tab, and the "\n" has become a newline.

```
1       2       3
A       B       C
```

The list of special characters are:

Character	Meaning
\a	Alarm (print a beep!)
\b	Backspace (lets you overwrite existing text)
\f	Form feed (move down but not back)
\n	New line (move down and back)
\r	Carriage Return (move back but not down)
\t	Tab

You can also use the same backslash to stop a character from being treated as something special. This would be needed if you wanted to insert a $ symbol into a double quoted string or a single quote into a single quoted string.

```
print "Simon says\"When writing Perl \$ has special meaning\"\n";
```

The set of characters which need to be escaped in a double quoted string are:

$ @ % " \

One last option for you. If you have a large chunk of text to quote you can use something called a "here document". This allows you to embed data (including newlines) in between two text delimiters you set yourself. The syntax for a here document is two greater-than symbols followed by a delimiter string in either single or double quotes, then the data you want to include, finished off by the delimiter at the start of a line on it's own. Using single or double quotes has the same effect in a here document as it does in a normal quoted sting.

```
my $long_text = <<'END_LONG_TEXT';
This is some long text
Which spans over multiple lines.
Because I used a single quote I can write things like $hello
and they won't be interpolated.  I can even use single  quotes
like in the last sentence and that's OK too!
END_LONG_TEXT

print $long_text;
```

Concatenation and Multiplication

If you want to build up a complex string you can do this using concatenation and multiplication. The dot operator ' . ' is used to concatenate two or more strings.

```
my $name = "Bob" . " " . "Smith";
```

You can also add to the end of an existing string using the same operator

```
my $name = "Bob ";
$name = $name . "Smith";
```

As an aside, any time you see a construct like the one above, with the same variable on both sides of the = sign you can use the more convenient form shown below:

```
my $name = "Bob ";
$name .= "Smith";
```

This works for any scalar operator (.= += -= *= etc);

Another useful thing to know about is the multiplication operator 'x'. This repeats a string a defined number of times. For example:

```
print "0"x10;
```

Prints 10 zeros. This will also work with multi-character strings:

```
print "deadly sin"x7;
```

Mathematical Operations on Scalars

Perl does not have a separate data type for numbers as opposed to text, not does it make any distinction between integers and floating point numbers. All of these data types are just scalars as far as Perl is concerned. In fact, behind the scenes, perl does treat these data types differently, it just hides it from you and automatically converts between them depending on how you're using a variable. The rule is that Perl will just "Do the right thing"™.

You don't need to quote a number when you assign it to a variable, but nothing bad will happen if you do.

Perl supports all of the standard mathematical operators you'd expect.

Operation	Example
Addition	`$x = $y + $z`
Subtraction	`$x = $y - $z`
Multiplication	`$x = $y * $z`
Division	`$x = $y / $z`
Exponentiation	`$x = $y ** $z`
SquareRoot	`$x = sqrt($y)`
Modulus	`$x = $y%$z`
Log (base *e*)	`$x = log($y)`

Because Perl does not distinguish between integers and floats it will automatically increase the precision with which it stores a number as necessary.

Given the information you've had so far it would also seem that perl should allow you to do nonsensical operations such as:

```
print "bob" / 2;
```

…and in some cases it will! However, when you wrote your hello world program you included the line "use warnings;" in your script. This forces perl to check mathematical operations to ensure that the data being passed to them is numerical. Code like that shown above produces the warning:

```
Argument "bob" isn't numeric in division (/) at line 6.
```

Increments and Decrement

Another useful shortcut are the mathematical increment and decrement operators. If you find yourself doing:

```
$x = $x + 1;
```

Then we've already seen that you can save a bit of space by doing:

```
$x += 1;
```

Which will work when adding any value. If you're only adding 1 though you can use the special increment or decrement operators ++ and –

```
++$x; # Adds 1 to $x
--$x; # Subtracts 1 from $x
```

Functions for use on scalars

Functions are the main way to perform an operation in perl. They are simply named blocks of code which perform a specific operation. Later in the course we will see that we can construct our own functions by writing sub-routines, but for now we're going to focus on built-in functions.

Using a function in perl is achieved by using the construct shown below

```perl
my $result = function_name($data)
```

You simply use the name of the function to run it. Many functions rely on being provided with some data to work with so the way to provide this is by including a set of round brackets after the function name and then including in there the data the function needs to work with. It the function needs more than one piece of data you separate the different data pieces using commas.

In many cases you can omit the brackets and just call the function as

```perl
my $result = function_name $data
```

You will often see this in scripts, but it's not generally a good idea, especially when you're first starting to write your own scripts. Keeping the brackets makes it very clear which data is going into the function, but to some extent it's a matter of style whether you include them or not.

All functions return some data after they have run. For some functions this data is empty or not useful, but in many cases you do want to keep the data passed back from the function. To keep data from a function you can assign it to a variable the same way you would with a raw piece of data.

There are a number of perl built-in functions which you can begin to use one you have some scalar data.

print

This is the one function you've seen so far. Print takes either a single scalar or a list (multiple scalars separated by commas) and by default prints them to STDOUT (the standard output – usually the console)

```perl
print "This is some text";

print "I can however","print more than one scalar", "in the
same statement";
```

length

The `length` function returns the length of the scalar

```perl
my $length = length("abcde");
print $length;    # prints 5
```

uc / lc

The functions `uc` and `lc` can be used to convert a string into upper or lower case. They do not alter the original string, but instead return the adjusted string, which can be assigned to a new variable, or back to the original one.

```perl
my $mixed = "cASe is ALL oVeR The PlaCE";
print lc($mixed); # All lower case, but $mixed unchanged

$mixed = uc($mixed);
print $mixed;  # All upper case
```

reverse

The `reverse` function reverses a scalar. As with uc/lc it doesn't change the scalar itself but returns a reversed version of it for you to play with.

```perl
my $string = "\n?siht daer uoy nac";
my $reversed = reverse $string;
print $reversed;
```

substr

The `substr` function allows you to extract a substring from a string. Its syntax is

```perl
substr ([string],[offset],[length])
```

String is the scalar you want to extract the substring from
Offset is the position in the string you want to start from (counting from 0). If you want to be clever you can use a negative offset to start counting back from the end of the string

Length is the length of substring you want to extract

Conditional Statements

In the simple scripts we have seen so far execution starts at the top and every line is executed until we reach the bottom when we stop. In most programs though things aren't that simple. It's very useful to have pieces of code which only get executed under certain conditions. In this section we're going to look at conditional statements.

A simple conditional statement is shown below.

```perl
my $salary = 100000;

if ($salary > 40000) {
  print "You must be in management...\n";
}
```

To construct a conditional statement you need a conditional keyword (`if` in this case) followed by a statement to be tested in round brackets, followed by a code block in curly brackets, to be executed if the test passes. In the above example the print statement only happens if `$salary > 20000` is evaluated as true.

What is truth?

A key concept when writing conditional statements in Perl is "truth". Conditional statements get executed based on whether they are "true" or "false". Unlike other languages though perl does not have a boolean variable type to specifically store true/false values, instead it simply groups all scalar values into being either true or false.

The simplest way to explain what is true/false in perl is to show all the possible false values. Everything else is true. Things which evaluate as false in perl are:

undef   (The value given to a variable you have declared but not yet assigned to)
""   (The empty string)
0   (The numerical value 0)

Putting this into practice:

```perl
if (0) {
  print "This won't print";
}
if ("") {
  print "And neither will this";
}
if (42) {
  print "But this will";
}
if ("hobgoblin") {
  print "And so will this";
}
```

All built-in functions in Perl return a value to indicate whether they have succeeded. Most of the time we don't bother reading this value and just ignore it (you don't often test that your print statement worked – but you could!). Some functions, such as the comparison operators, are only useful because of their return value.

Making Comparisons in Perl

To make conditional statements you therefore need a statement to evaluate, and the most common one you'll use will be comparisons.

Comparing Numbers: The simplest type of comparisons to make are numerical comparisons. These are:

$x > $y	X is greater than Y
$x < $y	X is less than Y
$x >= $y	X is greater than or equal to Y
$x <= $y	X is less than or equal to Y

For these comparisons to work of course you have to have a number in $x and $y. This is another reason for having warnings enabled in your program as perl will tell you if you try to do a silly comparison:

```perl
if ("one" < "two") {
  print "True";
}
```

produces…

```
Argument "two" isn't numeric in numeric gt (>) at example.pl line 5.
Argument "one" isn't numeric in numeric gt (>) at example.pl line 5.
```

Comparing Strings: You can also compare strings, which is a test based on their alphabetical order, so that 'a' is less than 'b'. Comparing between cases is a bit odd ('A' is less than 'a') and is normally not a good idea.

```
$x gt $y            X is greater than Y
$x lt $y            X is less than Y
```

Testing for equality

Probably the most common test you'll do is a test to see if two variables are the same. This however is the cause of many problems in perl.

Comparing Numbers: Numbers are compared using the `==` operator or inequality using `!=`. However this only works reliably for integers (whole numbers). Floating point numbers are only stored approximately in computers (this has nothing to do with perl) and two numbers which may be mathematically equivalent may end up testing to be different. If you want to compare for equality multiply them up so they become integers and remove any trailing decimal places (for instance, when working with currency you should always use the smallest denomination as your base, eg pence rather than pounds).

Comparing Strings: The other common mistake people make is to try to compare strings as if they were numbers. Instead you should use the `eq` operator in perl to compare strings for equality and the `ne` operator to test for inequality. Again note that the strings have to be exactly the same. Even a trailing space will cause the comparison to fail.

More Complex Conditions

In the previous examples we used a singe if statement, but often a more complex decision structure is required.

if – elsif – else

A full conditional statement in Perl can have up to 3 different types of entry, if, elsif (which can occur multiple times) and else. All conditions have to have the first type and the others are optional. A more complex if statement is shown below.

```perl
my $value = 100;
if ($value == 0) {
      print "Nothing there";
}
elsif ($value < 75){
      print "A little bit";
}
elsif ($value < 150) {
      print "Quite a lot";
}
else {
      print "Loads!";
}
```

In this case the if statement is evaluated first. If this fails the code moves through the elsif statements in order until one of them matches, at which point it stops. If none of the elsifs match then the else block is run. The fact that the elsif statements are evaluated in order means that in order to print "Quite a lot" you don't need to check that $value is greater than 75 as this elsif statement won't be checked unless the previous one failed.

unless

For some statements an if statement is less than optimal as you only want to do something if it fails (ie have an empty code block after the if, and something functional in the else block). In these cases you can use the unless keyword in Perl, which works in the same way as if, but passes if the expression given to it returns false rather than true.

The three bits of code below are therefore completely equivalent.

```perl
if ($value == 100) {
}
else {
      print "You don't have 100";
}

if ($value != 100) {
      print "You don't have 100";
}

unless ($value == 100) {
      print "You don't have 100";
}
```

Compound Statements (and / or)

If you want to check more than one condition in an if statement you can nest them together to produce more complex logic.

```perl
if ($day eq 'Friday') {
    if ($date == 13) {
        print "Ooh, unlucky!";
    }
}
```

However this can be overkill and sometimes you want to clean things up by putting everything into one statement. Perl therefore has the `and` and `or` keywords which can be used to chain conditions together.

```perl
if ($day eq 'Friday' and $date == 13) {
    print "Ooh, unlucky!";
}
```

For simple chains of ands you can just put them one after the other, but with more complex statements you need to put round brackets around each logical group so you can be explicit about what you mean. For example if you were to write:

```perl
if ($day eq 'Friday' and $date == 13 or $cat eq 'black') {
    print "Ooh, unlucky!";
}
```

You need to add some extra brackets because your logic is not clear. You could mean

```perl
if (($day eq 'Friday' and $date == 13) or ($cat eq 'black')) {
    print "Ooh, unlucky!";
}
```

or you could mean

```perl
if (($day eq 'Friday') and ($date == 13 or $cat eq 'black')) {
    print "Ooh, unlucky!";
}
```

These sort of errors won't produce a warning from perl as it has a set of rules it uses to decide on your meaning, but you are always better off using brackets to make things explicit and not relying on remembering the rules right!

Section 2: Arrays, Hashes and Loops

Arrays

Up to this point the only type of data structure we have seen is the scalar. Perl however has 3 built-in data types (scalars, arrays and hashes) so it's time to look at the other two.

The first of these is the array. The easiest way to think of an array is that it is a list of scalars. We need to be a bit careful using the word list though as perl also has a concept of a list, which is slightly different to an array.

Arrays therefore allow you to store scalars in an ordered manner and to retrieve them based on their position. Unlike many other languages arrays in Perl are dynamic, which means that you can add things to them or remove things from them at will. You don't need to say in advance how big they're going to be.

Arrays in Perl are denoted by a leading @ symbol (eg `@array`), array elements are scalars and therefore start with a $, but in order to identify which element of the array you mean they must also have their position in the array (known as their index) included in square brackets at the end of their name. The first element in an array has index 0 (this catches a lot of people out!!).

So for example the third element in `@array` would be `$array[2]`.

Time to look at some code:

```perl
my @array = ("dog","cat","fish","monkey");

print "The first element of \@array is not $array[1], but $array[0]";
```

Here you can see that when you create your array you can assign values to it by providing a list of scalars (a comma separated list in round brackets). As with scalars, when you want to create a new array you need to use the "my" keyword.

Another useful convenience is that you can use negative indices to count back from the end of an array.

```perl
print "The last element of \@array is $array[-1]";
```

Adding and taking away…

Arrays in Perl are dynamic, so you can add and remove elements after the array has been created. The function you use depends on whether you want to add or remove, and which end of the array you want to operate on.

To add elements to an array you use push (to add to the end) and unshift (to add to the start). To remove elements you use shift (to take from the front) and pop (to take from the end).

You can add more than one element at a time by passing a list to push/unshift, but you can only remove one element at a time using shift/pop.

```perl
my @array = ("dog","cat","fish","monkey");

my $first = shift @array; # dog

my $last = pop @array; # monkey

push @array, ("vole","weasel");

# Array is now cat,fish,vole,weasel
```

Array Slices

A slice is to an array what substring is to a scalar. It's a way to extract several values from array in one go without having to take the whole thing. The syntax of a slice is @array[list_of_indexes]. It returns a list of scalars which you can either assign to individual scalars or use it to initialise another array.

```perl
my @array =
("red","orange","yellow","green","blue","indigo","violet");

my ($two,$three,$five) = @array[2,3,5]; # two is yellow etc.

my @last_two = @array[5,6];
```

When assigning to a list of scalars (as in $two, $three, $five) the values are mapped from the list returned by the slice onto the scalars in the list. This same technique can also be used to extract values from an array without changing it (as would happen if you used shift/pop).

```perl
my @array = ("red","orange","yellow","green","blue","indigo");
my ($red,$orange,$yellow) = @array;
```

In this example the values are transferred to the scalars, but @array is left in tact. It doesn't matter that @array has more values than the list, the rest are just ignored.

Getting an array's length

One useful thing to be able to extract is the length of an array. There are two ways to get this. For every array there is a special variable associated with it which holds the highest index number contained in the array. As array indexes start at 0, this value is always one less than the length. The special variable is $#array_name. It's a good idea to get used to the notion of special variables in perl as they crop up a lot. They can produce some odd looking code, but as long as you realise you're looking at a special variable

```perl
my @array = (1,2,3,4,5);
print "The last index is ",$#array; # prints 4
```

Alternatively you can get the length of an array by using it in a situation where perl expects to see a scalar. If you use an array as a scalar you get the length of the array. For example:

```perl
my @array = (1,2,3,4,5);
my $length = @array;
```

```perl
print $length; # prints 5
```

Note that in this case the scalar you're assigning to does not have brackets around it and so isn't a list. Therefore the whole array is evaluated and you get the length. If you put brackets around `$length` when you assign to it, its value would be the first element of `@array`.

If you use this technique in a situation where an array could be interpreted either as a scalar or a list then you can ensure it is evaluated as a scalar by using the function `scalar`.

```perl
my @array = (1,2,3,4,5);
print "The length of the array is ",scalar @array;
```

Functions using arrays

As with scalars before there are a couple of functions which are only really useful in combination with arrays.

The join function turns an array into a single scalar string and allows you to provide a delimiter which it will put between each array element. It is commonly used when outputting data to write it to a file as either comma separated or tab delimited text.

```perl
my @array = ("tom","dick","harry");
print join("\t", @array);  # Produces tab delimited text
```

You can also go the other way and use the "split" function to split a single scalar into a series of values in an array. The split command actually uses a regular expression to decide where to split a string – don't worry about the details of this bit for the moment – we'll come to these later, just think of it as a string in between two "/" characters.

```perl
my $scalar = "HelloXthereXeveryone";

my @array = split(/X/,$scalar);

print "Second element is ",$array[1],"\n"; # there

print join(" ",@array); # prints "Hello there everyone"
```

Sorting Arrays

A common requirement having populated an array is to sort it before doing something with it. Sorting is actually quite a big topic in Perl, but most of the complexity is only relevant in situations you're not likely to come across until much later, so here we're going to do just the executive summary.

The function to sort an array is (you guessed it) `sort`. To work it uses a small block of code telling it how to do the comparison, and the array you want sorted. Sort doesn't alter the array you pass it but rather returns a sorted list of the elements contained in the original array.

When you sort an array what is actually happening behind the scenes is that perl is making lots of comparisons between pairs of values in the array. It uses this information to decide on the sorted order of the array. All you need to supply therefore is a bit of code which can take in two values and tells perl whether they came in the right order or not. To do this Perl uses the special variable names `$a` and `$b` – these variable names should be reserved for use in sort code blocks, they will work elsewhere, but it's bad practice to use them.

```perl
my @array = ("C","D","B","A");

@array = sort {$a cmp $b} @array;

print join(" ",@array); # prints A B C D
```

This code sorts the array alphabetically. The code block in the sort is the bit between the curly brackets {}. The block must contain a statement using $a and $b to say which one comes first. The two operators you can use for comparing scalars in Perl are cmp for comparing strings and <=> (called the spaceship operator) for comparing numbers. You can apply whatever transformations you like on $a and $b before you do the comparison if you need to.

The easiest way to illustrate the possibilities is with some examples:

Sort numerically in ascending order:
```perl
@array = sort {$a <=> $b} @array;
```

Sort numerically in descending order (same as before but with $a and $b swapped):
```perl
@array = sort {$b <=> $a} @array;
```

Sort alphabetically in a case-insensitive manner:
```perl
@array = sort {lc $a cmp lc $b} @array;
```

Hashes

The final variable type in Perl is the hash. A hash is a kind of lookup table, it consists of a collection of key-value pairs, where both the key and value are scalars. You can retrieve a value from the hash by providing the key used to enter it.

A couple of things to note about hashes:

- Although you can have duplicate values in a hash the keys must be unique. If you try to insert the same key into a hash twice the second value will overwrite the first.

- Hashes do not preserve the order in which data was added to them. They store your data in an efficient manner which does not guarantee ordering. If you need things to be ordered use an array. If you need efficient retrieval use a hash!

Hash names all start with the `%` symbol. Hash keys are simple scalars. Hash values can be accessed by putting the hash key in curly brackets `{}` after the hash name (which would now start with a `$` as we're talking about a single scalar value rather than the whole hash. For example to retrieve the value for "simon" from `%ages` we would use `$ages{simon}`.

When you create a hash you can populate it with data from a list. This list must contain an even number of elements which come as consecutive key value pairs.

```perl
my %hair_colour = ("Simon","brown","Iain","brown","Conor","blonde");

print $hair_colour{Simon}; # prints brown
```

In the above example note that you do not have to quote the key data (Simon) when retrieving data from a hash, even though it is a bare string.

Because the syntax used above for populating the hash doesn't make it clear which scalars are keys and which are values Perl allows an alternative syntax for writing lists. In this version you use the `=>` operator in place of a comma (it's also known as a fat comma). This has the same effect as a comma, and in addition it also automatically quotes the value to its left so you don't need to put quotes around the key names. The code below does exactly the same thing as the one above.

```perl
my %hair_colour = (Simon => "brown",
                   Iain =>"brown",
                   Conor =>"blonde",);

print $hair_colour{Simon}; # prints brown
```

This version makes it much clearer which are the keys and which are the values. One useful tip is that when you create a list this way you can leave a trailing comma at the end of the list. This means that if you later come back and add an extra key-value pair you are unlikely to forget to put in the extra comma which would be required.

Testing for keys in a hash

One very common operation is to query a hash to see if a particular key is already present. This looks like a straightforward operation, but it can catch you out if you're not careful.

One of the features of a hash is that although you need to declare the hash itself (using my) the first time you use it you don't need to declare each element each time you add one. This makes hashes very flexible, but also means you can put bugs like this in your code:

```perl
my %hair_colour = (Simon => "brown",
                   Iain=>"brown",
                   Conor =>"blonde",);

print $hair_colour{Smion}; # Compiles OK, but get warning at runtime
```

To allow hashes to be flexible, if a key is used which does not exist then that key is automatically created in the hash so you can proceed.

```perl
        my %hair_colour;
        $hair_colour{richard} = "grey"; # No error

        print $hair_colour{richard}; # prints grey
```

This functionality has implications when it comes to testing for the presence of a key in the hash. The most obvious (**and wrong**) way to do this would be something like:

```perl
        my %hair_colour;
        $hair_colour{richard} = "grey";

        if ($hair_colour{simon}) {
                print "We know about Simon"; # Doesn't print
        }

        if (defined $hair_colour{bob}) {
                print "We know about Bob"; # Doesn't print
        }
```

If you run this code it all seems to work the way it should, and in this isolated case it does. The problem is that because you have used the hash keys $hair_colour{simon} and $hair_colour{bob} in your tests, these have both been created in the hash. Because you didn't supply a value for them they have been created with undef as their value. This can come back to bite you should you later want to iterate through all the keys in your hash when you find you have more than you expect, many of which have undefined values.

The correct way to test for the presence of a key in a hash is to use the exists function. This is specifically designed for this purpose and will not alter the hash when used.

```perl
        my %hair_colour;

        if (exists $hair_colour{simon}) {
                print "We know about Simon"; # Doesn't print
        }

        if (exists $hair_colour{bob}) {
                print "We know about Bob"; # Doesn't print
        }
```

Loop Structures

Up to this point all of the code we've seen has run starting at the top of the program and finishes when it gets to the bottom (with some bits missed out on the way for if/else statements). We're now going to look at the functions perl has for creating loops where the same bit of code can run multiple times.

While loops

Probably the simplest kind of loop is the while loop. To form a loop you provide a block of code in curly brackets preceded by a statement which is evaluated as being either true or false. If it's true the looped block of code is run once. The condition is then tested again. The loop continues until the condition returns a false value.

To make a while loop work you must have something change in the block of code which affects the condition you supplied at the start. If you don't have this then the loop will either not run at all, or it will continue running forever (known as an infinite loop). A simple while loop is shown below which illustrates the normal syntax for a loop.

```perl
my $count = 0;

while ($count < 10) {
        print "Count is $count\n";
        ++$count;
}
```

In this loop the condition being evaluated is `$count < 10`, and the loop finishes because `$count` is increased by 1 every time the loop code runs.

Foreach loops

The other commonly used loop structure in perl is the foreach loop. This is used to iterate through a list of values where you can supply a block of code which will be run once for each value in the list supplied with that value being assigned to a scalar variable which is available within the looped block of code. If you don't supply a named variable to assign each value to it goes into the default `$_` variable.

A simple foreach loop is shown below:

```perl
foreach my $value (2,4,6,8,10) {
        print "Value is $value\n";
}
```

Here I've supplied a list of values which are assigned to the `$value` variable at each run through the loop. Because I'm creating a new variable to store the looped values I must use the `my` keyword before it.

If you don't supply a named variable for the looped values you can use `$_` instead.

```perl
foreach (2,4,6,8,10) {
        print;
        print "\n";
}
```

In this example the empty print statement operates on the default scalar variable $_ so the values in the loop do get printed.

Although you can manually create a list for a foreach loop it's much more common to use an existing data structure instead. This is usually either an array, or a list of the keys of a hash.

```perl
my @animals = ("doggies","bunnies","kittens");

foreach my $animal (@animals) {
        print "I just love cute little $animal\n";
}
```

If you want to iterate through a hash you can get a list of the keys of the hash using the keys function, or a list of the values using the values function (wasn't that easy!).

```perl
my %test_scores = (
        Adam => 50,
        Joe => 30,
        Sarah => 40);

foreach my $person (keys %test_scores) {
        print "$person got a score of ",$test_scores{$person},"\n";
}
```

Note that in this example the keys (on my machine at least) don't come out in the same order as they went in. If you want to have the results sorted then you must sort the list used in the loop.

```perl
foreach my $person (sort {$a cmp $b} keys %test_scores) {
        print "$person got a score of ",$test_scores{$person},"\n";
}
```

Finally, another useful bit of Perl syntax to use with foreach loops is the range operator. This consists of two scalars separated by a double dot and creates a list in which all values between the two scalars are filled in.

```perl
foreach my $number (100..200) {
        print "$number is a very big number\n";
}

foreach my $letter ("a".."z","A".."Z") {
        print "I know the letter $letter\n";
}
```

The behaviour of numbers in ranges is pretty intuitive (goes up by one each time). Letters are OK as long as you keep everything either upper or lower case and don't mix the two. You can do ranges with multiple letters, but watch out because they get pretty big pretty quickly!

a..z   = 26 combinations
aa..zz   = 676 combinations
aaa..zzz   = 17576 combinations
aaaa..zzzz   = 456976 combinations

For loops

For is just an alias for foreach. For loops, although common in other languages aren't often used in perl. For loops are available in perl but the things you'd normally use them for are usually better accomplished by using foreach instead (iterating through arrays for example). Since for is actually just an alias for foreach you can use the two interchangeably, it's just that your code is normally easier to understand if you use foreach.

The only common situation where a for loop is clearer than foreach is when you want to iterate through a range of values. In this case a loop like:

```perl
for (1..10) {
        print "$_ green bottles sitting on a wall\n";
}
```

Is probably a bit clearer than its foreach counterpart.

Extra controls within loops

In the loop examples shown so far the loops have always run to completion and all the code has been run each time, but often you want to loop through some data and only process some of it. It's also useful to be able to break out of a loop entirely.

There are two extra perl functions which work in all types of loop. The `next` function immediately sends a loop back to check its condition statement and start over. No code from the rest of the block is run. You can use this to skip over certain values when processing a list.

```perl
my @days = ("Mon","Tue","Wed","Thur","Fri","Sat","Sun");

foreach my $day (@days) {
        next if ($day eq "Sun" or $day eq "Sat");
        print "Ho hum it's $day - time to go to work\n";
}
```

To break out of a loop completely you can use the `last` function. This exits the loop immediately and moves on to the next statement below the loop. Using `last` you can break out of what would otherwise appear to be an infinite loop.

```perl
while (1) { # Infinite loop
        my $guess = int(rand(10));
        if ($guess==5) {
                print "You guessed!\n";
                last;
        }
        else {
                print "No, not $guess, try again!\n";
        }
}
```

In this example a random integer between 0 and 10 is created. If it equals 5 the loop exits. If not it goes round again. The random nature of the code means that the loop will not always be run for the same amount of times. You can use perldoc to see the details of what the `int` and `rand` functions do.

Nested Loops

Next and last affect whatever is the closest enclosing loop block, but sometimes you want to have a loop within a loop, and be able to use next or last on the outer loop. To do this you can optionally tag a loop with a name which you can use with next and last to move around in nested loops.

```perl
YEARS: foreach my $year (2000..2005) {
      if (($year % 4) != 0) {
            foreach my $month (1..12) {
                  # I only work until July
                  if ($month == 7) {
                        print "Yippee, 6 months off!\n";
                        next YEARS;
                  }
                  print "Suppose I'd better do some work in month
                        $month of $year\n";
            }
      }
      print "Ooh $year is a leapyear!\n";
}
```

In this case I've tagged the outer loop with the string YEARS (it's conventional, although not required that these sorts of tags are all in uppercase), so that I can jump to the next year from within the months loop.

Section 3: File Handling

Since perl is particularly well suited to working with textual data it often gets used to process files. This section covers the aspects of Perl necessary for reading and writing to files safely.

Creating a filehandle

In order to do any work with a file you first need to create a filehandle. A filehandle is a structure which perl uses to allow functions in perl to interact with a file and is usually represented as a normal scalar variable. You create a filehandle using the `open` command. This can take a number of arguments;

- The variable you're going to use for the filehandle
- The mode of the filehandle (read, write or append)
- The path to the file

When selecting a mode for your filehandle perl uses the symbols;

- < for a read-only filehandle
- > for a writable filehandle
- >> for an appendable filehandle

If you don't specify a mode then perl assumes a read-only filehandle, and for reading files it is usual to not specify a mode. The difference between a write and append filehandle is that if the file you specify already exists a write filehandle will wipe its contents and start again whereas an append filehandle will open the file, move to the end and then start adding content after what was already there.

The code below shows an example of opening a read-only and a writeable filehandle.

```
open (my $in,'C:/readme.txt');

open (my $out,'>','C:/writeme.txt');
```

In older code you may well see the mode combined with the filename, you may also see filehandles written as bare text in all capitals. Both of these practices will still work, but doing things this way is now discouraged.

```
open (OUT,'>C:/writeme.txt');
```

You should note that even when working on windows systems you should use the forward slash (/) as a delimiter in your file paths. The backslash notation you often see is actually only a function of the shell that windows uses, and behind the scenes forward slashes work just as well. Backslashes cause problems in Perl as they have special meaning (they make the following character a special character). For example:

```
open (my $out,">","C:\table.txt");
```

In the above code you've just tried to open a file called [tab]able.txt, which probably isn't what you meant!

Closing a filehandle

When you've finished with a filehandle it's a good practice to close it using the `close` function. This is more important for writable filehandles than read-only ones (as we'll see later), but it never hurts. If you don't explicitly close your filehandle it will automatically be closed when your program exits. If you perform another open operation on a filehandle which is already open then the first one will automatically be closed when the second one is opened.

Error Checking

All of the code shown so far in this section has a big problem. It doesn't cope with failure. What happens if readme.txt doesn't exist or if you don't have permission to write to C:/writeme.txt? The short answer is that perl will happily swallow the failure and move on and you'll get odd errors later in your code when you try to use the filehandle you created. Therefore:

When opening any filehandle, or closing a writeable filehandle you MUST ALWAYS check that the operation succeeded before continuing.

You can check that the operation succeeded by looking at the return value it supplies. If an open operation succeeds it returns true, if it fails it returns false. You can use a normal conditional statement to check whether it worked or not.

```perl
my $return = open (my $in,"C:/readme.txt");

if ($return) {
      print "It worked!";
}
else {
      print "It didn't work :-(";
}
```

The code above is more complex than you really need. Firstly you create a variable (`$return`) which you don't need – you can examine the return value directly. Secondly you usually don't need to do anything if the open succeeds; you're only interested in trapping any failure. The final problem is what happens when it fails. At the very least you want to be told that it failed and why. Normally you want to stop the program from going any further if things didn't work. If a file open fails then perl stores the reason for the failure in the special variable `$!`. If you include this in your error message you'll see what went wrong. If you want to stop your program you can use the die function to quit immediately with a message telling you what went wrong and where.

```perl
open (my $in,"C:/readme.txt") or die "Can't read C:/readme.txt: $!";
```

Gives me:

```
Can't read C:/readme.txt: No such file or directory at line 5.
```

Special Filehandles

In addition to creating your own filehandles perl actually comes with three of them already defined for you. These are special filehandles which you can use to interact with whatever process launched your perl script (usually a shell or IDE). The special filehandles are:

STDOUT – A writeable filehandle. Used as the default output location for print commands. Is usually redirected to the console from which you ran your perl script.

STDIN – A readable filehandle. Used to pass information from the console into perl. Allows you to ask questions on the console and get an answer.

STDERR – Another writable filehandle usually attached to the console but normally only used for unexpected data such as error messages.

These special filehandles are all implemented in the older style of bare all capital names so you don't need to put a $ symbol before them when using them.

Reading from a file

To read data from a file you use the <> operator. You put the identifier of the filehandle you want to read from in between the angle brackets. This reads one line of data from the file and returns it. To be more precise this operator will read data from the file until it hits a certain delimiter (which you can change, but we won't in the examples we show here). The default delimiter is your systems newline character ("\n"), hence you get one line of data at a time.

```perl
my $file = "M:/tale_of_two_cities.txt";
open (my $in,$file) or die "Can't read $file: $!";

my $first_line = <$in>;

print $first line;
print "The end";
```

This produces the following output:

```
It was the best of times, it was the worst of times,
The end
```

You'll notice that even though there is no "\n" at the end of the first print statement, the second one still appears on the next line. This is because the <> operator doesn't remove the delimiter it's looking for when it reads the input filehandle. Normally you want to get rid of this delimiter, and perl has a special function called chomp for doing just this. Chomp removes the same delimiter that the <> uses, but only if it is at the end of a string.

```perl
my $file = "M:/tale_of_two_cities.txt";
open (my $in,$file) or die "Can't read $file: $!";

my $first_line = <$in>;
chomp $first_line;

print $first_line;
print "The end";
```

This code produces:
```
It was the best of times, it was the worst of times,The end
```

Whilst we can use the <> operator to read just one line from a file this is not the way things are usually done. Apart from anything else we really should have checked that there really was a first line in this file and that $first_line had some data in it.

The more normal way to read a file is to put the <> operator into a while loop so that the reading continues until the end of the file is reached.

```perl
my $file = "M:/tale_of_two_cities.txt";
open (my $in,$file) or die "Can't read $file: $!";

my $line_count = 1;

while (<$in>) {
        chomp;
        print "$line_count: $_\n";
        last if $line_count == 5;
        ++$line_count;
}
```

Gives:

```
1: It was the best of times, it was the worst of times,
2: it was the age of wisdom, it was the age of foolishness,
3: it was the epoch of belief, it was the epoch of incredulity,
4: it was the season of Light, it was the season of Darkness,
5: it was the spring of hope, it was the winter of despair,
```

OK, so I cheated and bailed out after 5 lines, but this would in theory have worked to number every line of the whole file. It would also have done the right thing had there been less than 5 lines in the file as the while condition would have failed so the loop would have exited. Note that by not specifying a variable name to assign to in the loop condition statement the assignment goes to the default $_ variable. This in turn means that I don't need to pass an argument to chomp as it (like most scalar functions) operates on $_ by default.

Reading from STDIN

A really useful facility is the ability to read in information from the console. This allows you to produce interactive programs which can ask the user questions and receive answers. To achieve this you simply use the STDIN filehandle in the same way as you'd use any other read-only filehandle. You usually only read one line at a time from STDIN (so the input stops when the user presses return), but you can put the read into a while loop which will continue until the use enters the end-of-file symbol (Contol+D on most systems, but this can vary).

```perl
#!perl
use warnings;
use strict;

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

print "What is your quest? ";
my $quest = <STDIN>;
chomp $quest;

print "What is your favourite colour? ";
my $colour = <STDIN>;
chomp $colour;

if ($colour eq "Blue... no, yellow!") {
        die "Aaaarrrgggh!";
}
```

Writing to a file

Writing to a file is really simple. The only function you need to use is print. All of the previous print statements shown so far have actually been sending data to the STDOUT filehandle. If a specific filehandle is not specified then STDOUT is the default location for any print statements.

```perl
#!perl
use warnings;
use strict;

open (my $out,'>',"M:/write_test.txt") or die "Can't open file
for writing: $!";

print $out "Sending some data\n";

print $out <<"END_DATA";
Now I'm sending a lot of data
All in one go.
Just because I can...
END_DATA

close $out or die "Failed to close file: $!";
```

The main thing to remember when writing to a file is that in addition to checking that the open function succeeded, you must also check that you don't get an error when you close the filehandle. This is because errors can occur whilst you are writing data, for example if the device you're writing to becomes full whilst you're writing.

When using the print function to write data to a filehandle be careful that you **don't** put a comma between the filehandle and the data. If you do then the data will be written to STDOUT instead of your filehandle.

Buffering

One thing which can catch people out when writing to a file is buffering. If you write a program which does a lot of work and periodically writes some information out to a file, then you set it running and watch the contents of the file what you may find is that nothing gets written to the file for ages, and then loads of data appears at once. This is because perl optimises writes by holding them in a buffer until its got enough data together to bother writing it out to the specified filehandle. It therefore makes no guarantees that print statements appear at their destination straight away. The buffer is only definitely emptied when you close the filehandle (which is why that's where you have to check for failures on a writable filehandle).

If you want to turn off buffering you can do this, but it is a global preference which will affect all the writeable filehandles in your program. You do this using one of perl's special variables called $|. If you set this to a value of 1 it will turn buffering off. Setting it to 0 restores buffering.

```perl
#!perl
use warnings;
use strict;

$| = 1;
```

```perl
open (my $out,'>',"M:/write_test.txt") or die "Can't open file
for writing: $!";
print $out "I'll definitely appear straight away!";
close $out or die "Failed to close file: $!";
```

It's not a good idea to disable buffering unless you really need to. Disk access is quite slow and an unbuffered filehandle will really slow your program down if you send a lot of data to it.

Working with Binary Files

All of the examples we've used so far in this section have used text files, and this will undoubtedly account for the majority of files you're ever likely to process. Sometimes however you will need to work with binary files (for example images). In these cases you'll want to do things slightly differently.

A normal filehandle in perl expects to send and receive textual data and as such it does a clever thing to help you. For historical reasons different computer platforms actually use different characters to tell you when you are at the end of a line in a text file. Windows, Unix/Linux and Macintoshes all uses different characters to indicate that you're at the end of a line. Filehandles in perl know this and when you read in and write out data they will automatically translate the end of line characters to whatever is appropriate to the system you're using. This means that in some circumstances if you read in and write out a text file with perl the two files won't be exactly the same. For text files this is a good thing. For binary files it can be a disaster!

To ensure that perl won't change the data from a file as it's read in or written out you need to put the filehandles into binary mode. You do this using the binmode command. Once this has been applied to a filehandle it will not alter binary data.

```perl
#!perl
use warnings;
use strict;

open (my $in,'M:/test.jpg') or die "Can't read file: $!";
binmode $in;

open (my $out,'>','M:/test2.jpg') or die "Can't write to file:
$!";
binmode $out;

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

close OUT or die "Can't close file: $!";
```

File System Operations

As well as being able to read and write files Perl offers a number of other filesystem operations within the language.

Changing directory

Instead of having to include a full path every time you open or close a file it is often useful to move to the directory you want to work in and then just use filenames. You use the `chdir` function to change directory in perl. As with all file operation you must check the return value of this function to check that it succeeded.

```perl
chdir ("M:/Temp") or die "Couldn't move to temp directory: $!";
```

Deleting files

Perl provides the `unlink` function for deleting files. This accepts a list of files to delete and will return the number of files successfully deleted. Again you must check that this call succeeded.

```perl
# This works:
unlink ("M:/Temp/killme.txt","M:/Temp/metoo.txt") == 2 or die
"Couldn't delete file: $!";

# But this is better:
foreach my $file ("M:/Temp/killme.txt","M:/Temp/metoo.txt") {
      unlink $file or die "Couldn't delete $file: $!";
}
```

Listing files

Another common scenario is that you want to process a set of files. To do this you often want to get a file listing from the filesystem and work with that. Perl provides a function called file globbing to do this kind of task. There are two ways to perform a file glob, one is just a shortcut to the other.

```perl
chdir ("M:/winword") or die "Can't move to word directory: $!";

# Quick easy way
my @files = <*.doc>;
print "I have ",scalar @files," doc files in my word directory\n";

# Longer way to do the same thing
my @files2 = glob("*.rtf");
print "I have ",scalar @files2," rtf files in my word directory\n";
```

Although you can pull a glob into an array as shown above a nicer way to use it is actually to treat it a bit like a filehandle and read filenames from it in a while loop.

```perl
chdir ("M:/winword") or die "Can't move to word directory: $!";

while (my $file = <*.doc>) {
      print "Found file $file\n";
}
```

Testing files

If you want to open a file for reading it's much nicer to specifically test if you can read the file rather than just trying to open it and trapping the error. Perl provides a series of simple file test operators to allow you to find out basic information about a file. The file test operators are:

- -e   Tests if a file exists
- -r   Tests if a file is readable
- -w   Tests if a file is writable
- -d   Tests if a file is a directory (directories are just a special kind of file)
- -f   Tests if a file is a file (as opposed to a directory)
- -T   Tests if a file is a plain text file

All of these tests take a filename as an argument and return either true or false.

```perl
chdir ("M:/winword") or die "Can't move to word directory: $!";

while (my $file = <*>) {
      if (-f $file) {
            print "$file is a directory\n";
      }
      elsif (! -w $file) {
            print "$file is write protected\n";
      }
}
```

As an aside, in the above example I needed to test whether the writable test failed as part of an else statement. Because there is no elsunless function in perl I added an ! to the start of the statement. This operator reverses the sense of the test which follows. It's the language equivalent of putting NOT at the end of a sentence.

Section 4: Regular Expressions

Perl's initial popularity stemmed from its powerful tools for processing text. It initially included the functionality from two different text processing languages called sed and awk. The most powerful tool for text processing in perl is the regular expression engine which is a tool to find, extract and modify text patterns.

Regular expressions are almost a language in themselves. Their main function is to provide a flexible syntax for finding patterns in text, but the advanced features they now provide virtually make them a language in themselves (for instance someone wrote a regular expression which would find prime numbers!).

Fortunately the basics of regular expressions are very simple and you can be writing your own in no time.

Finding Patterns

The basis of all regular expressions is the pattern find. Everything else is a variation on this. A pattern find can be used in a condition statement to determine whether or not it succeeded. The syntax for a simple find is:

```perl
my $string ="aaaaaabobaaaaaa";

if ($string =~ /bob/) {
    print "I found bob!";
}
```

The regular expression is the part between the two forward slashes. In this cases it's the literal string "bob", but as we'll see this can be much more flexible. You tell the regular expression which string its going to look in using the =~ operator. If you don't specify a variable then the default $_ variable is used. If you want to test for a match failing you can use !~ to bind the string to the regex.

The big thing to learn in regular expressions is how to construct the expression string itself. You can use a literal string, but that's not taking any advantage of the power of regular expressions.

For simple characters you can just the literal character. Special characters we've already seen in strings use the same expansions in regular expressions (\t for a tab, \n for a newline etc). You can also use a dot (.) to indicate that you can match any character at that position.

```perl
my $string ="aaaaaabobaaaaaa";

if ($string =~ /b.b/) {
    print "I found b[something]b!";
}
```

Some characters have special meaning inside a regular expression and if you want to use these literally you must escape them with a backslash. Symbols which are special in a regex are:

$$\backslash \quad | \quad (\quad) \quad [\quad \{ \quad \char`^ \quad \$ \quad * \quad + \quad ? \quad .$$

Character Groups

If a position in a string could have a number of different characters then you can use a character group to indicate this. A character group is a set of characters in square brackets.

```perl
my @strings =("dog","cog","log","fog");

foreach my $string (@strings) {
    if ($string =~ /[dclf]og/) {
        print "$string matched\n";
    }
}
```

You can also make negative character groups which match everything except the characters in them. These have a ^ as the first symbol in the group.

```perl
my @strings =("dog","cog","log","fog");

foreach my $string (@strings) {
    if ($string =~ /[^cl]og/) {
        print "$string matched\n";
    }
}
```

This matches dog and fog, but not cog or log.

Because some character groups are used very commonly these have shortcuts in the regular expression syntax. These shortcuts are:

- \w Matches any word character (a-z A-Z and _)
- \W Matches any NON word character
- \d Matches any digit
- \D Matches any NON digit
- \s Matches any whitespace (spaces and tabs – a tab is just \t)
- \S Matches any NON whitespace

```perl
my @strings =("dog1","dog2","dogX");

foreach my $string (@strings) {
    if ($string !~ /dog\d/) {
        print "$string had no number\n";
    }
}
```

Repeated Characters

If you are going to have a run of the same character or character group you don't want to have to write it out multiple times. Instead you can tell the regex how many times you want that character to match.

You can match a character a specific number of times by including a number in curly brackets after that character.

```perl
my $string ="aaaaXXXXaaaa";

if ($string =~ /aX{4}a/) {
    print "Matched\n";
}
```

If you'd rather be able to match a range of numbers then you can include 2 numbers separated by commas. If you miss out the number before the comma then 0 is assumed (match up to X occurrences), if you miss out the number after the comma then the match can continue indefinitely (match more than X occurrences).

```perl
my $string ="aaaaXXXXaaaa";

if ($string =~ /aX{2,5}a/) {
    print "Matched\n";
}
```

Again there are a couple of shortcuts for the most common cases of matching repeated characters. The * character matches 0 or more of the previous character (equivalent to {0,}). The + character matches 1 or more of the previous character (equivalent to {1,}).

```perl
my $username ="abc123";

if ($username =~ /\s+/) {
    print "You can't have spaces in a username\n";
}
```

Alternatives

We've seen that by using character groups we can allow more than one option at a particular position. This approach won't work though if we want to allow alternatives of more than one character. In these cases we use multiple options in round brackets separated by bars (|).

```perl
if ($name =~ /(Bart|Lisa|Maggie)\s*Simpson/) {
    print "D'Oh!\n";
}
```

Anchors

The matches we've made so far could have happened anywhere in a string. We haven't yet been able to constrain where the match occurred. This can lead to some nasty bugs in code.

```perl
my $value = "Yah boo 123 sucks!";
if ($value =~ /\d+/) {
    print "It's a number\n"; # Oh no it isn't
}
```

To constrain our matches we can use anchors in our regular expression. The most common anchors to use are those which match the start and end of the string. If your regular expression starts with ^ then it is anchored to the beginning of the string. If it ends with $ then it is anchored to the end of the string. If you use both ^ and $ then your pattern will need to match all of the string.

```perl
if ($sentence !~ /^[A-Z]/) {
  print "You must start your sentence with a capital letter\n";
}

if ($sentence !~ /[\.!\?]$/) {
  print "You must end your sentence with appropriate punctuation\n";
}

if ($value =~/^\d+$/) {
      print "This really is a number\n";
}
```

The other anchor commonly used is the word boundary. This is denoted by \b and anchors the surrounding matches to a word boundary (anywhere where a \w+ match finishes).

```perl
my $name = "manfred";

if ($name =~/fred/) {
      print "You could be fred\n";
}

if ($name =~ /\bfred\b/) {
      print "You ARE fred\n";
}
```

Case Sensitivity

By default regular expression matches are case sensitive. You can modify the behaviour of a regular expression by putting modifiers after the last /. Some of these modifiers will be discussed later, but a useful one to know is that if you use the i modifier then your match becomes case insensitive.

```perl
if ("hELlo tHErE" =~ /hello there/i) {
      print "See, it works";
}
```

Making Replacements

It's all well and good finding out whether a string matches a particular pattern, but often you will want to modify the pattern once you find it. A variant of the regular expression matching syntax can be used to perform find and replace operations (called substitutions).

In a substitution you put an `s` before your first `/` and you include and extra section after the pattern to say what you want to replace the matched sequence with. This substitution will change the string it operates on.

```perl
$string = s/\t/    /; # Replace tabs with spaces

$string = s/\bBob\b/Robert/; # If we're being formal
```

The second part of a replacement is just a string (which will interpolate any variable names you use in it – it acts like a double quoted string).

Looping Finds

If you have a pattern which may be found more than once in a string and you want to perform some action each time it is found then you need to do a looped regex. These can be used either as part of a replacement to replace all occurrences of a pattern in a string, or with a while loop to allow you to run through a code block each time you get a match. To create a looped match you simply add the `g` modifier to the end of your regex.

```perl
$string = s/\bBob\b/Robert/g; # Do things properly


my $count = 0;
while ($string =~ /(gosh|darn|flip)/g) {
      ++$count;
}

print "You said $count naughty words";
```

Whilst looping through a series of matches it can also be useful to know exactly where in your string the last match happened. By default a looped match will start searching at the end of the last match position. This last position is stored for each string and is updated during a looped match. You can read (and set) this position using the `pos` function.

```perl
my $string = "012X456X89X";
while ($string =~ /X/g) {
      print "Found match at position ",pos($string),"\n";
}

Found match at position 4
Found match at position 8
Found match at position 11
```

You can see that the position returned by pos is the index of the next character after the end of the previous match. If you want the start of the match you need to subtract the length of the thing you were searching for.

Pos is also useful if you want to be able to find overlapping hits. In this case you need to use pos to move the start point for the search back to just after where the last search started to make sure you found all possible hits.

This example shows a normal, non-overlapping search:

```perl
my $string = "bobobobobob";
while ($string =~ /bob/g) {
        print "Found bob at position ",pos($string),"\n";
}

Found bob at position 3
Found bob at position 7
Found bob at position 11
```

Whilst this shows the same thing, but altering the search start position to allow an overlapping search:

```perl
my $string = "bobobobobob";
while ($string =~ /bob/g) {
        print "Found bob at position ",pos($string),"\n";
        pos($string) -= (length("bob") - 1);
}

Found bob at position 3
Found bob at position 5
Found bob at position 7
Found bob at position 9
Found bob at position 11
```

Capturing Data

When you use a regular expression which can match a range of strings it is often useful to be able to find out exactly what it did match. There is therefore facility for capturing all or part of the matched pattern in the regular expression engine. This functionality is extremely useful for stripping out important bits of data from a longer string.

To capture part of a regular expression you simply surround it with round brackets. When that expression is matched the parts in brackets are put into the special numbered variables $1, $2, $3 etc. The furthest left set of brackets gets $1 and the numbers increase as you move further right.

```perl
my $string = "File code=123 name=test.txt";

if ($string =~ /code=(\d+)\s+name=([\w\.]+)/) {
        print "Code is $1\nName is '$2'\n";
}
```

Because the special values are only populated when a match succeeds you must only use them in a code block which is conditional on the match having succeeded. Using these values anywhere else will cause nasty bugs which are difficult to trace. You also shouldn't rely on the number variables being maintained outside the block even if you know the match succeeded. It's better to test for success and then immediately transfer the data to another conventional data structure.

This example goes through a whole book and keeps count of the number of times a word beginning with a capital Q appears. It writes out the answers sorted by the number of occurrences (with the highest occurring one first).

```perl
open (my $in,"M:/tale_of_two_cities.txt") or die "Can't read file $!";

my %q_count;
while (<$in>) {
        while (/\b(Q\w+)\b/g) {
                ++$q_count{$1};
        }
}

foreach my $word (sort {$q_count{$b}<=>$q_count{$a}} keys %q_count) {
 print "The word $word appeared ",$q_count{$word}," times\n";
}
```

If you're interested – this produced:

```
The word Queen appeared 8 times
The word Quick appeared 7 times
The word Quarter appeared 7 times
The word Quite appeared 2 times
The word Quickly appeared 1 times
The word Quietly appeared 1 times
```

When not to use Regular Expressions

Having achieved the perl equivalent of a white belt in regular expressions you'll quickly find that you want to start using them for all sorts of tasks for which they are inappropriate. This is a quick summary of a few situations in which you could use regular expressions, but for which a better alternative exists.

Splitting delimited data

A bit of a cheat to start with, as this does partially use regular expressions. In section 2 we came across the split function which takes a string and by providing a delimiter you can split it into a list of elements. The delimiter you pass in this case is a regular expression, but you need to make sure you're never tempted to use a looped regex where a split would be more appropriate.

Swapping single characters

If you want to swap all occurrences of a single character in a string for a different character then it's tempting to use a regex, and this will undoubtedly work. Perl however has a separate function called `tr` (for transliterate) for this purpose which offers a couple of advantages over a regex.

A `tr` operation looks very much like a regex. The two following pieces of code have the same effect;

```perl
$string =~ s/a/b/g;

$string =~ tr/a/b/;
```

Tr in this case is much faster than a regex (but frankly in most situations they'll both be so quick you won't notice). However tr has another advantage.

Let's say you actually wanted to swap all occurrences of "a" for "b" and all occurrences of "b" for "a". If you try to do this in a regex it all goes horribly wrong:

```perl
my $string = "aaabbb";

$string =~ s/a/b/g;
$string =~ s/b/a/g;

print "String is $string\n";

String is aaaaaa
```

Tr however lets you perform more than one swap at the same time:

```perl
my $string = "aaabbb";

$string =~ tr/ab/ba/;

print "String is $string\n";

String is bbbaaa
```

Tr can also be used without the replacement string filled in. This function can be used in two ways. Using the same syntax as above you can use tr to count the number of occurrences of a character (or series of characters) in a string.

```perl
my $string = "aaabbb";

my $count = ($string =~ tr/a//);

print "The letter a occurred $count times";
```

Alternatively you can leave the replacement blank and use the d modifier to tr to have it delete the matched characters from the string.

```perl
my $string = "banana";

$string =~ tr/a//d;

print "What's left is $string";
```

Extracting fixed position data

If you have some data you know to be at a fixed position in a string then it is much more efficient to use substr to extract it than construct the corresponding regex.

```perl
my $string = "12345helloxxxxx";

if ($string =~ /^.{5}(.{5})/){
      print "$1\n"; # This might look clever
}

print substr($string,5,5),"\n"; # But this is better
```

Finding the position of an exact string

Although you can use a regex and the pos function to find the position of substring within a string it is much faster to use the dedicated index function for this purpose.

```perl
my $string = "xxxxxxhixxxxxxxxxhixxxxxxxxhi";

my $lastpos = 0;

while (1){
      my $pos = index($string,"hi",$lastpos);
      last if ($pos == -1); # Substring not found
      print "Found hi at index $pos\n";
      $lastpos = ++$pos;
}
```

Section 5: Subroutines, References and Complex Data Structures

It's time to get a bit dirty in Perl. Until now we've kept things fairly simple in our scripts by keeping a roughly linear execution path and using simple data structures. However, what you'd soon find if you stopped learning perl at this point is that two really bad things would start happening to your scripts.

1. You would find yourself using copy / paste an awful lot as you found yourself wanting to do a fairly simple task several times within the same script.

2. You would end up with a ridiculous number of variables when you tried to model complex data structures.

This section aims to address these problems and lead you to the path of perl enlightenment. If you really get to grips with the ideas in this section then you will be amazed at how much easier you suddenly find writing your Perl scripts.

Subroutines

One of the cardinal sins made by people starting out using any language is a tendency to overuse copy and paste. Given the choice I would ideally like to disable copy and paste on anyone's machine when they start learning a new language. Copying and pasting is bad because:

1. If you just copy / paste something you did before it's much less likely to stick in your brain than if you have to type it out again.

2. You rarely want an exact copy of what you've done before, and by the time you've made all the relevant changes for what you're doing now, and then fixed the bugs you created because you missed some changes it's usually faster to type the whole thing in again.

3. If you're copying and pasting a lot in a single application then it's a sign that you should be finding a more modular way to build your code so you can reuse it in a more robust manner.

The simplest form of modular code reuse in Perl is the subroutine. This is a block of code to which you can pass a list of scalars, and from which you can collect a list of scalars. You can then call this code from anywhere in your program and thereby provides a sensible way of reusing code. It's the closest you come in Perl to being able to add a new function to the language.

The simplest kind of subroutine is one which takes no arguments and returns no values.

```perl
sub print_it {
    print "Look! A subroutine!\n";
}
```

Subroutines start with the sub keyword, followed by their name (conventionally all in lowercase), followed by a block of code surrounded by curly brackets. That's all there is to it. To call this code from somewhere else in your code you simply do:

```
        print_it();
```

Just put the name of the subroutine followed by a pair of round brackets.   Again, conventionally, subroutines go at the bottom of your script, after the main part of the program – but they don't have to.

This subroutine is OK, but it's pretty boring. It does the same thing every time its run which is rarely of use. Let's make it a bit more interesting by passing in some arguments to the subroutine.

Arguments to a subroutine are passed in the special perl array @_. A subroutine can check this array to see what elements it has been passed. Elements are passed to the subroutine by putting a list in the round brackets following the subroutine name when calling it.

```perl
reverse_it("Bolton");

sub reverse_it {
  my $string = shift @_;
  unless (defined $string) {
      die "The reverse_it subroutine requires a string";
  }

  print "A palindrome of $string would be ",scalar reverse $string;
}
```

The conventional way of picking up values from @_ would either be to use shift (as in the example above), where the @_ is actually optional since this is the default for shift, or to map the values onto a list, for example:

```perl
        my ($string) = @_;
```

It's normally a bad idea to access @_ directly, since any changes you make to it will actually affect the variable which was passed to the subroutine. This is a really good way to confuse yourself!

```perl
        my $bank_account = "1000";

        innocent_sub($bank_account);
        print "I have £$bank_account in my account\n";

        nasty_sub($bank_account);
        print "I have £$bank_account in my account\n"; # What

        sub innocent_sub {
            my ($ammount) = @_;
            $ammount -= 500;
        }

        sub nasty_sub {
            $_[0] -= 500;
        }
```

Getting data back from a subroutine is achieved using the return function. This passes back a list which is given as the return value wherever the subroutine was called from.

```perl
my ($total,$count,$average) = number_stats(1,2,3,4,5,6);

print "Total=$total Count=$count Average=$average\n";

sub number_stats {
        my @numbers = @_;

        my $count = 0;
        my $total = 0;

        foreach my $number (@numbers) {
                ++$count;
                $total += $number;
        }

        my $average = $total/$count;
        return ($total,$count,$average);
}
```

This example shows a subroutine which can take a list of numbers of any size and will return
the total, count and average of those numbers. It shows the full range of syntax you can use
with a subroutine as it both takes arguments and returns results. It's reuse of some variable
names also illustrates the first of the big perl ideas in this section; variable scope.

Scoping

Whenever we're created a new variable in our scripts we've always prepended its first use with the keyword `my`. This has the effect of telling perl that we are aware that we're creating a new variable and not to bother warning us about it. However, this isn't the only effect that `my` has. In addition the position of the `my` statement determines where in your program that variable is visible. Most of the code you've seen so far has used variables which are visible throughout the program, but this is not always the case.

Imagine for a moment that you are writing a big perl program. Thousands of lines of code. You're writing a small subroutine and as part of this you'd like to keep a count as you go through a loop. You decide to make a variable called `$count` to hold the count value – but then you think "have I used `$count` before?". Do you really want to have to go on a big hunt through the rest of your program to see if you're going to write over some other variable in your program by using $count. Of course you don't – and indeed you don't have to!

When you create a new variable using `my` it is actually only a valid variable within the current block of code. A block of code is defined by being surrounded by curly brackets (such as a subroutine, loop or conditional statement). Any code blocks within the current block can see the variable, but to any code outside the current block it appears that that variable doesn't exist.

```perl
if (1) { # Always succeeds
      my $name = "Bond, James Bond";
}

print "Hello there $name";
```

Gets you:

```
Global symbol "$name" requires explicit package name at line 9.
Execution aborted due to compilation errors.
```

This is because `$name` was declared inside the block of code following the `if` statement. It therefore isn't visible outside that block.

This is a really good thing as it means that you don't need to worry about clashing variable names outside your current block of code. You can have 20 subroutines, all of which declare a `$count` variable, and they will all work happily as the variables are kept within the scope of the subroutine and so won't clash.

So what if you wanted to have the `$name` variable visible outside your if block in the above example? The simple answer is that you declare it outside the loop, but give it a value inside the if block.

```perl
my $name;

if (1) { # Always succeeds
      $name = "Bond, James Bond";
}

print "Hello there $name";
```

We're not out of the woods yet though. What happens if we declare a variable in a block which has previously been declared in a wider code block? The answer is that we are free to declare that variable again, and for the extent of that block our new variable masks the one with wider scope.

```perl
my $name = "Ernst Stavro Blowfeld";

if (1) {
        my $name = "Bond, James Bond";
        print "In here I'm $name\n";
}

print "But here I'm $name";
```

Gives me:

```
In here I'm Bond, James Bond
But here I'm Ernst Stavro Blowfeld
```

Perl won't throw any errors or warnings when you temporarily mask a variable which has a wider scope as it's a perfectly valid thing to do. The only time you'll get an error is if you try to declare the same variable more than once in the same scope.

The golden rule for scoping in Perl is that wherever possible variables should be declared where they are first used and that they should occupy the smallest possible scope. A sure sign of a really dodgy Perl script is a big list of variables all being declared right at the start! In anything more than a 50 line script you should have hardly any variables which are declared so they are available throughout your whole program. It seems a pain, but as your programs get larger (and they will) you will really start to see the benefits of keeping your variables tightly scoped.

References

The other big perl idea in this section is references. A quick way to illustrate what references are any why they are useful is shown below. The compare arrays subroutine should compare two arrays and say which one has more elements in it.

```perl
my @array1 = (1,2,3,4);
my @array2 = (5,6,7);

compare_arrays (@array1,@array2);

sub compare_arrays {
      print "Length of \@_ is: ",scalar @_,"\n";
}
```

The problem is that all you can pass to a subroutine is a list of scalars. When you try to pass two arrays therefore they become "flattened" into one list, and once that happens there's no way to split them back into the arrays you started with.

What references do is to provide you with a scalar value which acts like a pointer to another data structure. A reference can point at any perl data structure (scalar, array or hash). You can then use that reference to get to the underlying data structure again.

Making a reference from an existing data structure

To make a reference to an existing data structure you simply refer to that data structure with a backslash in front of it.

```perl
my $scalarref = \$scalar;
my $arrayref = \@array;
my $hashref = \%hash;
```

Making a reference directly

If you know you are going to convert your array or hash into a reference you can actually create it that way by creating what is known as an anonymous data structure. Both arrays and hashes provide some extra syntax which allows you to create a reference directly without using another named variable.

Both forms of the following snippets of code are equivalent in terms of the references they produce.

To get an anonymous array you simply put a list in square brackets.

```perl
my @array = (1,2,3,4);
my $slow_arrayref = \@array;

my $quick_arrayref = [1,2,3,4];
```

To get an anonymous hash you put a list in curly brackets.

```perl
my %hash = (
        dog => 'woof',
        cat => 'meow',
);
my $slow_hashref = \%hash;

my $quick_hashref = {
        dog => 'woof',
        cat => 'meow',
};
```

Dereferencing

Once you have your reference it's only really useful if you can use it to get back to the underlying data. One way to do this is to reverse the process of creating the reference to copy whatever underlying data structure was there originally.

You can dereference a reference by prepending it with the appropriate symbol for the type of underlying data structure ($ for scalars, @ for arrays, % for hashes).

```perl
my %hash = %$hashref;
my @array = @$arrayref;
my $scalar = $$scalarref;
```

You can now use the new copy of the original data. We can now see how we could have solved the initial problem with the compare_arrays subroutine.

```perl
my @array1 = (1,2,3,4);
my @array2 = (5,6,7);

compare_arrays (\@array1,\@array2);

sub compare_arrays {

        my ($arrayref1,$arrayref2) = @_;

        if (@$arrayref1 > @$arrayref2) {
                print "First array was longer\n";
        }
        else {
                print "Second array was longer\n";
        }
}
```

You can also use this method to access any of the values contained in the data structures. Just use the same syntax as would normally be used for that data structure but include the extra $ to indicate that you're working with a reference.

```perl
my $arrayref = [10,20,30];
print "First element is ",$$arrayref[0],"\n";

my $hashref = {
        duck => 'quack',
};
print "The duck says ",$$hashref{duck},"\n";
```

The problem with this method of dereferencing is that the syntax can become cluttered (especially when we start nesting references later) and it's not always immediately clear that you're working with a reference. Another form of syntax is therefore available which allows you to dereference to access data inside a reference. This syntax uses an arrow operator to dereference. The code below does exactly the same as the last example above but uses the alternative syntax:

```perl
my $arrayref = [10,20,30];
print "First element is ",$arrayref->[0],"\n";

my $hashref = {
        duck => 'quack',
};
print "The duck says ",$hashref->{duck},"\n";
```

In this version it's a lot clearer that `$arrayref` and `$hashref` are references and that the `[0]` and `{duck}` are dereference operations.

Copying and changing references

When you pass your data around using references you need to be aware that because a reference is just a pointer to an underlying data structure it can sometimes behave in unexpected ways, especially in regards to making copies. When you copy a normal Perl data structure you duplicate all the data in it, such that if you modify the copy the original remains unaffected. In the following example the copy is altered so the second element is 40, but the original still says 4.

```perl
my @original = (2,4,6,8);
my @copy = @original;

for (0..$#copy){
        $copy[$_] *= 10;
}

print "Copy says ",$copy[1]," but original was ",$original[1];
```

Now if you try the same thing with references what you're actually duplicating is just a pointer. Both the original reference and the copy point to the same data structure so once you've made the change it doesn't matter which reference you use to access the data the change will still be visible. In the following example both the original and copy references point to an array whose second element is 40.

```perl
my $original = [2,4,6,8];
my $copy = $original;

for (0..(@$copy-1)){
      $copy->[$_] *= 10;
}

print "Copy says ",$copy->[1]," but original was ",$original->[1];
```

This behaviour can actually be extremely useful. If you have created a large data structure which you want to pass around you don't want to have to make copies of it as this will be very wasteful in terms of system resources. It's much tidier to work with references and just pass around pointers to the same data structure.

Using References to build complex data structures

Now that we've seen what references are it's time to put them to good use. We've already seen that one use for references is the passing of non-scalars to subroutines (you can also use them to pass data back), but there is another really good use for them.

Let's imagine for a moment that we had the following data which we wanted to store and access in a program.

Country	Population	Language	Currency
UK	60,441,457	English	Pounds
France	60,656,178	French	Euros
Ireland	4,015,676	English/Irish	Euros

None of the basic perl data structures is really suitable. We could use a hash to relate country to population, but we've have to use another one to relate country to language and another to relate country to currency, all of these would have the same keys. What it would be nice to have is a hash with the country as the key and another hash as the value in which we could store information about population, language and currency.

You'll not be surprised to hear that by using references you can do exactly this kind of thing. Arrays and hashes can't contain other arrays and hashes because they can only store scalar values, but they can store references to other arrays or hashes. By storing references in a data structure therefore we can build up more complex data structures to more accurately model more complex data.

A long winded way of showing how this works is below.

```perl
my %countries;

my %uk_info = (
    population => 60441457,
    language => 'English',
    currency => 'Pounds',
);

my %france_info = (
    population => 60656178,
    language => 'French',
    currency => 'Euros',
);

my %ireland_info = (
    population => 4015676,
    language => 'English / Irish',
    currency => 'Euros',
);

$countries{uk} = \%uk_info;
$countries{france} = \%france_info;
$countries{ireland} = \%ireland_info;

print "Population of France is ",
```

```
        $countries{france}->{population},"\n";
```

The final line shows how you can retrieve the hash reference for France from the countries hash and then use the arrow notation to dereference it to obtain the population.

In this example all of the individual country data structures have been created as separate hashes first before being added to the `%countries` hash, but by using anonymous hashes we could have done all of this in one step.

```
    my %countries = (
        uk => {
            population => 60441457,
            language => 'English',
            currency => 'Pounds',
        },

        france => {
            population => 60656178,
            language => 'French',
            currency => 'Euros',
        },

        ireland => {
            population => 4015676,
            language => 'English / Irish',
            currency => 'Euros',
        },
    );

    print "Population of France is ",
        $countries{france}->{population},"\n";
```

In this version we still create the `%countries` hash all in one go, but by using anonymous hashes for the population / language / currency information we save ourselves the bother of making intermediate data structures for each country.

The more common use for such data structures though is to build up a complex data model a bit at a time. The nice thing about using anonymous hashes and arrays is that you don't need to declare to perl that you're going to put an anonymous hash into another hash value, you just start doing it and perl does the right thing.

```
    my %countries;

    $countries{uk} -> {population} = 60441457;
    $countries{france} -> {population} = 60656178;
    $countries{france} -> {currency} = 'Euros';

    print "Population of France is ",
        $countries{france}->{population},"\n";
```

Nowhere in this code are the anonymous hashes used to store the population and currency information ever declared, we just started adding data to them as if they did and it all just works.

By using these techniques you can model pretty much any kind of data structure you can think of. You can mix normal scalars in with your references too if that makes sense.

```perl
my @experiments = (
    {
        sample_id => 1,
        sample_name => 'kidney',
        sample_measures => [12,56,34,65,76],
    },
    {
        sample_id => 4,
        sample_name => 'liver',
        sample_measures => [24,66,12,17,26],
    },
        {
        sample_id => 26,
        sample_name => 'spleen',
        sample_measures => [85,93],
    },
);

foreach my $expt (@experiments) {
    print "The first measure for sample ",
        $expt->{sample_id},
        " (",$expt->{sample_name},") was ",
        $expt->{sample_measures}->[0],"\n";
}
```

This data model uses different levels of references for different part of the structure. It's an array of hash references, but one of the hash values (`sample_measures`) is an array reference. You can see the different levels of dereferencing in the print statement at the end.

Section 6: Perl Modules

Perl has many advantages as a language. Larry Wall, the guy who first developed perl, says he did so because he was "Lazy and impatient", and he wanted a language for people like him. Perl already helps by having a simple set of flexible data types, by having concise flexible syntax and by being interpreted so you can test things quickly. Those all cover impatient well, but we haven't yet dealt with lazy.

Lazy is where modules come in. Having a good grasp of what perl modules are and how to use them will easily halve the amount of code you have to write – and in many cases will reduce it to practically none at all.

What is a module?

A perl module is just a chunk of Perl code written in such a way to allow it to be easily reused. If you like, perl modules are ways of providing extensions to the core perl language, in fact in addition to the core perl language the perl distribution actually comes with its own set of core modules. These are extensions which you know will always be present when someone is using perl.

A module allows perl to do something it couldn't do before. It can provide pretty much any function from communicating with a web server, to allowing you to write your Perl programs in latin (I swear I'm not making that one up!).

How to find modules

Although you can write your own Perl modules most of the time you don't need to. The truly lazy way to use modules is to find one someone has already written to do the job you're trying to do, install it and be done with it. For all but the most specialised of tasks there's probably a module out there already to do it. If you can't find it you're probably not looking hard enough!

The best place to find perl modules is in the Comprehensive Perl Archive Network (CPAN). This is the official public repository for perl modules and when I last checked it contained over 8700 different modules. That's a lot of code you can get for free.

The easiest way to find what you want is to use the CPAN search tool at http://search.cpan.org.
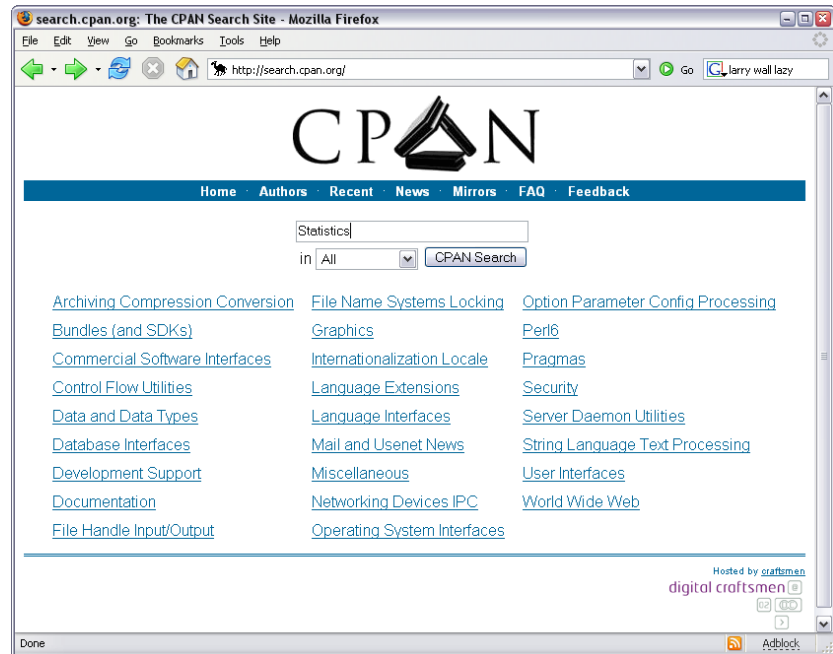
From here you can either browse through the categorised list of modules, or enter some keywords to go searching for what you want. If for instance you wanted to do some statistical calculations then by putting "Statistics" into the search box you would find that there are 677 different statistical modules for you to select from. One of them will probably do what you want!
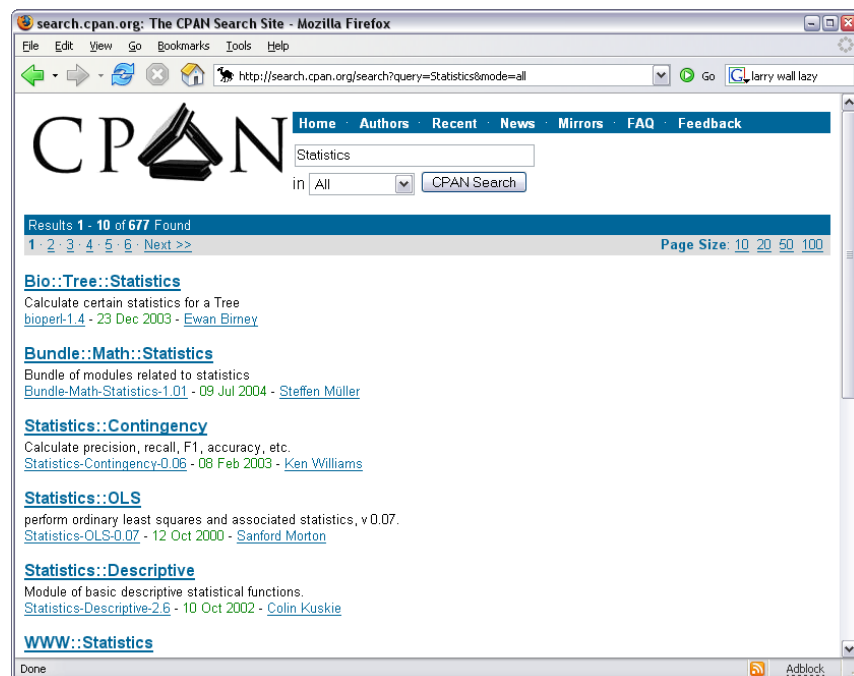
Having found a likely looking candidate you can then click on its name to get a more detailed description of what it does to decide if this is really what you want.
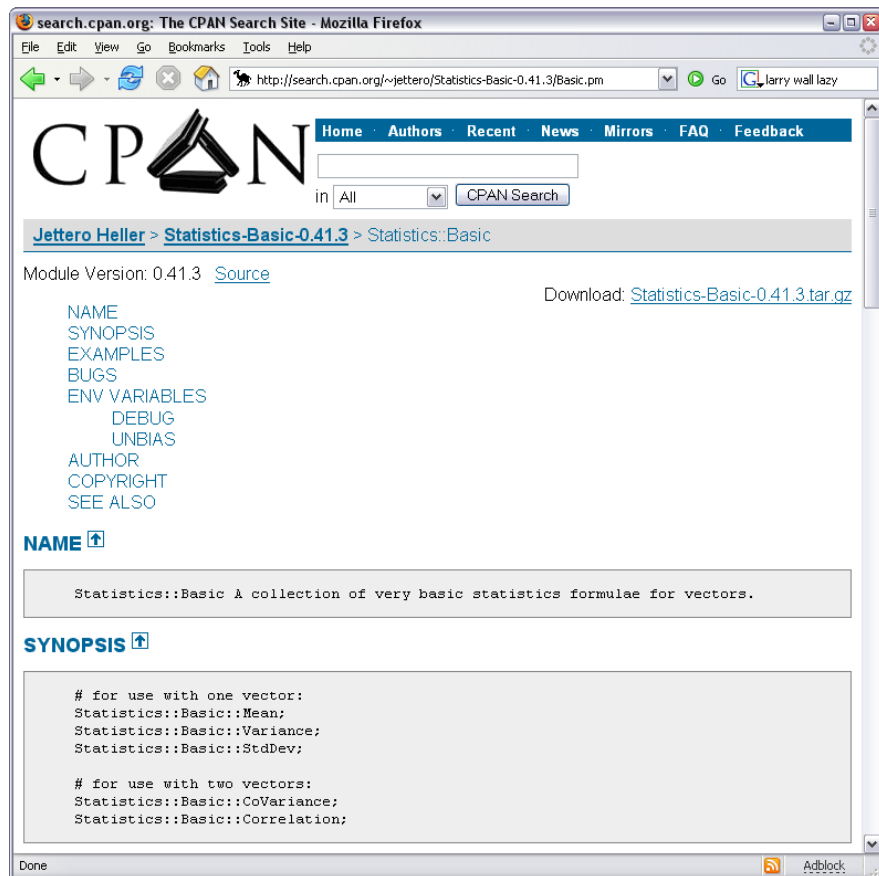
How to install modules

Once you've found the module you're after you need to install it. Well actually you should first check that it's not already installed. The easy way to do this is to look to see if its documentation is on your system. You can check this with perldoc, just do `perldoc Module::Name`, and if you see any documentation appear then the module is probably installed already.

Manual installation

If you want to manually install a module then you can do so by downloading the tar.gz file from CPAN (there is a link from the module details page). The process of installing a module is the same for all the modules in CPAN so once you've done one you can do them all. The process goes:

```
tar -xvf downloaded_file.tar.gz
perl Makefile.pl
make
make test
make install
```

You uncompress the file, run the setup script which checks that all the prerequisites needed by the module are in place. Compile the module (if needed, if not do it anyway, it won't hurt!), run the tests to make sure everything's OK, and then finally save the module into your perl installation.

This all assumes that you're on a system which provides tar, make and probably a C compiler (some perl modules have binary C files as part of them). This is probably true if you're on a Mac (if you've got the dev tools installed) or Linux, but not if you're on windows. On windows you can install most modules by downloading the nmake.exe program from Microsoft and using that instead of make. You can also use WinZip to extract the tar.gz file if you don't have the tar program on your system.

CPAN.pm

The problem when installing modules is that many of them depend on other modules (module writers are lazy too!), so when you do your perl Makefile.PL it will stop and tell you that in order to install module X you need module Y. Module Y will tell you that it needs modules A, B and C, and in no time you'll be banging your head against the screen and sobbing.

Fear not though, help is at hand via.. you guessed it… a module! The CPAN module takes care of all the nitty gritty of downloading, configuring, resolving dependencies and installing. All you do is tell it which module you want to use at the end of it and it takes care of everything in between. You'll also be glad to know that the CPAN module is a core module so you don't need to install it, it comes with every installation of perl.

To use CPAN.pl just type at a command line (use double quotes if on windows):

```
perl -MCPAN -e 'shell'
```

This will cause your shell prompt to read:

```
cpan>
```

…and from here you can install any module by simply typing "install Module::Name". Go off and have a cup of tea or watch the messages fly by, you're not needed again until it's all over and everything is installed for you.

PPM

If you're using perl on Windows then you're probably using ActivePerl, which is a binary distribution of perl which is very easy to install on windows. In addition to getting perl you also get a tool called ppm (the perl package manager), which is the easiest way to install modules if you're using ActivePerl. PPM works in a very similar way to CPAN.pm in that it automatically resolves any module dependencies for you and you just sit back and let it go. To use ppm simply open a shell and do:

```
ppm install Module::Name
```

..and that's it.

How to use Modules

Once you've got your module installed on your system you can start to use it in your programs. To tell perl that you want to use a module in one of your programs you need to include a use statement at the top of the script. This loads the module and makes it available to the rest of your script. All of the scripts you've written so far should have included the lines:

```perl
use warnings;
use strict;
```

At the top. This is because warnings and strict are modules (core modules so they're always there).

There are two different approaches a module can use to add functionality into your program, either it can export functions into the program, or it can provide an object oriented interface. The only way to know what a particular module is doing is to read its documentation (perldoc Module::Name).

Functional Modules

The simplest kind of modules simply export functions into your program. This has exactly the same effect as you would get if you had written an extra subroutine in the script yourself and they're used in the same way.

Some of these modules will export a default list of function names so you just need to put a bare use statement. Some will allow you to selectively export a list of function names which you put in a quoted list after the use statement. In these latter cases you can normally use the shortcut name :all to export all functions.

```perl
#!/usr/bin/perl
use warnings;
use strict;
use Date::Calc qw(:all);

print "In Feb 2020 there are ",Days_in_Month(2020,2)," days";
```

The Days_in_Month() function was exported from the Date::Calc module, but it acts in exactly the same way as if it had been included directly in your own script.

Object Oriented Modules

This is another big topic in perl which we're going to gloss over slightly. Perl was originally designed as a functional programming language, but as of perl 5 it was extended to cover object oriented programming as well.

Objects in perl are scalars, they are actually a special kind of reference. Objects are created using a function called new() which must be provided by the object oriented module. Once you have created an object it then provides a number of methods which you can call from it using the arrow notation we've already seen for dereferencing. The easiest way to show how an object works is to show an example of an object oriented module in action.

```perl
#!perl
use warnings;
use strict;
use LWP::UserAgent;

my $object = LWP::UserAgent -> new();
$object -> proxy("http://wwwcache.bbsrc.ac.uk:8080");

my $response = $object ->
get("http://www.bioinformatics.bbsrc.ac.uk");
if ($response -> is_success()){
        print $response -> content();
}
```

This example, which fetches a webpage and prints out the HTML, actually uses two different objects. The first is `$object` (which is a really bad name for your object!). This is the user agent object which is created by calling the `new()` method of the `LWP::UserAgent` module. Note that ONLY for the `new()` method you make the call using the name of the module. After you've created the object all method calls should be made on the object itself.

Once we've created the object we can start calling the methods it provides. First we call its proxy() method to set the value for the BBSRC web proxy. The object will retain that information for the rest of its life and will apply it to any transfers we subsequently ask it to make.

We then call the `get()` method to make an HTTP request. What is returned from this (`$response`) is actually another different type of object (`HTTP::Response`). If you want to see what sort of object something is you can just print it. If you print `$response` what you get is:

`HTTP::Response=HASH(0x1bb1688)`

The first part tells us that it's an `HTTP::Response` object. The second part is actually what a reference looks like if you try to print it directly. It's actually a memory location for the hash that the object is based around (remember objects are just special references). If you didn't know what methods an object provided, now that you know its an `HTTP::Response` object you can just do perldoc `HTTP::Respose` and look it up.

We can then use the `is_success()` method of `$response` to find out if everything worked. If it did then we can call the `content()` method to see what we got.

Although there is a lot of theory behind object orientation, and a lot of detail about what you can do with objects in perl you really don' t need to know much to take advantage of them. As long as you know that:

- You can find the documentation for `Spiffy::Module` by doing perldoc `Spiffy::Module`

- The documentation for all modules should provide a synopsis which shows you how to use it

- If you get returned an object you weren't expecting you can just print it to see which module it comes from

Then you know enough to make effective use of pre-existing modules in Perl.

Writing your own Modules

Although you will find a huge amount of functionality in existing modules at CPAN sometimes it is useful to create your own modules. You can quite easily create either functional or object oriented modules.

As with Perl programs, modules are just text files which contain Perl code. The main thing which distinguishes them from Perl programs is that modules have a file name which ends in .pm rather than .pl for programs.

Every module has a name, and the name of the module determines the filename under which it must be saved and the directory it has to be in.

A typical module name looks something like:

```
Example::Module::Name
```

To translate this module name into a file name you should treat the :: characters as if they were directory delimiters, and then add .pm to the last part of the name to get the filename it should be saved under. So from the above example you would have a directory structure and filename which looked like:

```
Example/Module/Name.pm
```

Perl will look for modules in a pre-defined set of directories which are stored in a perl special variable called @INC. You can view the contents of @INC on your system by running

```
perl -V
```

and looking at the end of the output.

```
@INC:
  C:/Perl/site/lib
  C:/Perl/lib
  .
```

This shows that our Example directory would have been found if it was placed at:

```
C:/Perl/site/lib/Example…
C:/Perl/lib/Example…
./Example…
```

The last entry, which is always present, says that you can always place module directories in the same directory as your script and they will be found. For modules which you are only going to use with a single script this is usually the best thing to do as it keeps all of your code together.

If you want to put your modules into another directory then you need to let your script know this by altering @INC. The best way to do this is to use a module (of course!). If you wanted to add the directory C:/Modules to @INC you would add the line;

```perl
use lib ('C:/Modules');
```

to the top of your script. This would append this directory to @INC for the duration of your script.

In addition to defining your module by its filename and where it is stored, you should also specify the module name inside the file. You do this by means of a 'package' statement. This defines the namespace of the module, which provides a private area in which your variable names will be localised. The package name should match your module name. Both should use the same names and capitalisation, even if you are using a case-insensitive file system.

The bare bones of a module should therefore look like this:

```perl
#!perl
use warnings;
use strict;

package Example::Module;

# Module code goes here

1;
```

The one last common addition to every module is the last line shown. When a module is loaded the way that perl checks that there were no errors is to check its return value. The return value of any perl module is the return value of the last evaluated expression. Since you don't want to take any chances with working out what the last evaluated expression is the easiest way to ensure this is true is to add 1; at the end of the module (because 1 is always true). If there are errors compiling your module then this expression will never be reached and the return value will be false.

From this point onwards the way you construct your module depends on the type of module you're going to write.

Functional Modules

A functional module is really just a collection of subroutines which have been split into their own file.

To create a functional module you use the simple template shown previously and add to it as many subroutines as you like.   You can have public subroutines which are intended to be run by users of your module and private subroutines which are only supposed to be run by code inside the module. As with most things in perl this public/private division is only enforced by convention – which is that if you want a subroutine to be considered private you should start its name with an underscore.

Once you have the subroutines present you can leave it at that if you like. In this case anyone using your module can call your subroutines but would have to do so using the fully

qualified name of the subroutine. For a module called Example::Module containing a subroutine called function you would need to call it as:

```
Example::Module::function()
```

To make life simpler functional modules are normally written so that they 'export' some subroutines into the namespace of the calling program. This means that as long as 'function' is in the list of subroutines to be exported you can treat it as if it was present in your program and call it using just:

```
function()
```

To export functions from your module you need to use another module called Exporter. Below is an example of using exporter to selectively export subroutines from a module.

```perl
#!perl
use warnings;
use strict;
use Exporter;

our @ISA = qw(Exporter);
our @EXPORT_OK = qw(will_be_exported);

sub public {
    # This is designed to be seen, but can only
    # be addressed by using its fully qualified
    # name eg Example::Functional::public()
}

sub _private {
    # This is only for internal use and shouldn't be
    # used from outside the module.
}

sub will_be_exported {
    # This can be exported into the namespace of the
    # calling program
}

1; # Must return a true value
```

Object-oriented Modules

The theory of object orientation is somewhat beyond the scope of this introductory course so I'm not going to try to cover them – merely outline the syntax for creating a simple object oriented module.

In Perl an object is simply a special kind of reference which maintains a link to its class, where the class is an object oriented module. The link to the original module is created using the Perl function 'bless'.

Objects can be any kind of reference, but the most commonly used by far is a hash reference.

In an object oriented Perl module methods are just subroutines. You must supply a subroutine called 'new' to create new objects. Methods are assumed to be public unless their name starts with an underscore – although the public/private contract is only enforced by convention.

Once you have created an object you can call the other methods in the class through that object. When calling a subroutine via an object, the object itself is always passed as the first argument to that subroutine – any other arguments are appended to @_ after the object itself.

Below is an example of an object oriented module which stores and retrieves a single piece of data.

```perl
#!perl
use warnings;
use strict;

package Example::Object_oriented;

sub new {

    # This creates the refernece which is going
    # to be our object
    my $hashref = {};

    # We then call bless to associate it with
    # this module.
    bless $hashref;

    # Finally we return it so the calling program
    # can start using it.
    return $hashref;
}

sub save_value {

    # This subroutine takes a single arguement
    # which it stores in the hash reference.
    #
    # The $object is provided automatically as
    # the first argument to every object oriented
    # subroutine (other than 'new').

    my ($object,$new_value) = @_;

    $object->{value} = $new_value;
}

sub get_value {

    # This subroutine retrieves a value which was previously
    # stored via the save_value subroutine.  If there isn't
    # a value to retrieve it returns the undefined value.

    my ($object) = @_;

    if (exists $object->{value}) {
        return $object->{value};
    }
    else {
        return undef;
    }
}

1;
```

This would be an example of a program which used this module.

```perl
#!perl
use warnings;
use strict;
use Example::Object_oriented;

my $object = Example::Object_oriented->new();

$object->save_value("Hello");

print "The object says '". $object->get_value() . "'\n";
```

```perl
#!perl
use warnings;
use strict;
```

Section 7: Interacting with external programs

Often when you're putting together a script a big chunk of the functionality you want is already present in another program and it's a lot easier to simply call that program from within perl than to re-implement what it does in perl. Using the functions described in this next section it is straightforward to either pass data through an external program as part of your Perl script, use Perl as a glue language to automate the execution of other programs, or simply use Perl as a convenient way to launch anther program.

System, backticks and exec

There are three different functions you can use within Perl to launch an external program. These are system, backticks (` `) and exec and they all have slightly different purposes.

System

System is used where you want to launch an external program and check that it worked, but you don't need to collect any data back from it.

```perl
my $web_address = "http://www.google.com";

chdir ("C:/Program Files/Internet Explorer/") or die "Couldn't move dir: $!";

my $return = system("iexplore.exe",$web_address);

print "Return was $return\n";
```

System takes a list as an argument. The first element of the list must be the program you want to call. Other elements are optional and are passed as command line arguments to the program.

A system call will cause your perl program to stop execution until the program you called has finished executing. It returns the return code that the program provides. If there is an error in a system call it is put into the special perl variable $?. Conventionally a program which has exited cleanly returns a code of 0. Any other return indicates that an error occurred. This means that when checking for errors from a system call you need to do the following.

```perl
system("iexplode.exe",$web_address)==0 or die "Error: $?";
```

Backticks

Although system is fine for launching an external program the only information you get back from it is whether it succeeded or not. Many programs will spit out useful information on STDOUT which it would be useful to collect and process further. If you want to get hold of this information then you need to use backticks rather than system.

Backticks are used in much the same way as any other quoting in Perl and you can use then to assign data to a variable. The main difference is that if you use backticks the contents are passed out to whatever shell your platform provides and the contents of STDOUT are written into the variable you're assigning to.

```perl
my $config = `ipconfig`; # Windows only!

if ($config =~ /IP Address.*:\s*(\d+\.\d+\.\d+\.\d+)/) {
      print "The IP address of this machine is $1";
}
```

Exec

The final built in function for interacting with external programs is exec. Exec is a somewhat unusual function in perl as it causes execution of your perl program to end immediately, and your running perl program is replaced by whatever program you specify. It's useful for example if you write a script to process some data and write it to a file and then finally use exec to launch another program to view it.

There's no point doing any error checking on an exec call, since as soon as you've made it your perl program is no longer around to look at the return value!

```perl
while (1) {
      print "Are you bored yet? [y/n]: ";
      my $answer = <STDIN>;

      if ($answer =~ /^y/i) {
          exec("C:/Windows/system32/sol.exe");

          # This never gets called, and you'll get
          # a warning about it!
          print "Here you go";
      }
}
```

Using pipes

With the backticks command we saw how it is possible to retrieve data from STDOUT from an external command. However the data we get back come all in one lump which can do nasty things to the memory consumption of our program if there's a lot of it. We also haven't yet seen a way to launch a program which needs a lot of data sent to it in a memory efficient way.

Both of these problems can be solved by interacting with external programs using filehandles and pipes. A pipe is a mechanism where you can write data to the STDIN of a program or read data from its STDOUT using the normal filehandle mechanism in perl. What you're actually doing is connecting STDIN or STDOUT on the external program to a named filehandle in your perl script.

Piping data out of an external program

To create a pipe out of an external program you use the open function specifying the program in the same way you'd normally specify a file, but putting a pipe character (|) as the last character in the open string. You can then read the data outputted by the program in the same way as you'd read a normal filehandle.

```perl
open (my $log,"tail -f /var/log/httpd/access_log |") or die
"Can't open pipe to web logs: $!";

while (<$log>) {
  if (/Safari/) {
    print "Oooh, a Mac user...\n";
  }
}
```

Piping data into an external program

Putting data into an external program works the same way as getting it out. You simply open a filehandle to the program putting pipe character as the first character in the open statement. You can then print data to that filehandle. Generally the data won't get processed until the input filehandle is closed.

```perl
open (my $zip,"| zip compress.zip -") or die "Can't open pipe
to zip: $!";

print $zip "I want to be smaller...";

close $zip or die "Can't close pipe to zip : $!";
```

IPC::Open2 / 3

You might think, having seen the previous examples of interacting with pipes, that if you wanted to pipe information both into and out of a program you should be able to do something like:

```perl
open (BROKEN,"| niftyprog |") or die "Can't open dual pipe: $!";
```

But you can't. Apart from anything else, when you closed the filehandle to indicate you'd sent all the input data you wouldn't have a filehandle left to read any output data from. Instead you need to use the `open2` or `open3` functions to allow you to perform this kind of operation. The difference between the two is that open2 allows you to interact with STDIN and STDOUT whereas open3 allows you to interact with STDIN, STDOUT and STDERR.

Open2 and open3 are not part of the core perl language (although they are part of the core distribution). They come as add-ins as perl modules. Modules will be discussed in much greater detail later, but for now all you need to know is that if you put the line:

```perl
use IPC::Open2;
```

.. near the top of your program then the open2 function will suddenly start working (just swap the 2 for 3 for open3).

Open2 and open3 work in a very similar fashion to a standard open except that they generate either 2 or 3 filehandles in a single operation.

```perl
#!/usr/bin/perl
use warnings;
use strict;
use IPC::Open2;

my $pid = open2 (\*OUT, \*IN,"seqret -filter -osf embl") or die
"Can't open pipes to seqret: $!";

print IN ">test\ncgatctgatgctgatgctgatgcta";

close IN or die "Can't close pipe input: $!";

print while (<OUT>);

waitpid($pid,0);
```

There are a couple of things in this code snippet which will be glossed over, but need to be noted. The filehandle names passed to open2 need to be preceded by *. This is because you're actually passing on an existing (although non-functional) filehandle rather than creating a new one as in an open command. Don't worry about this – just add the extra symbols and forget about it.

The open command returns the process identifier of the external program which is launched. After you've finished reading from the output filehandle it's a good idea to use the waitpid function to make sure that the external process has cleaned up properly after itself. For a simple, single run command this won't matter, but if you run your open2 commands in a loop then you will find that old dead processes start stacking up and this can cause problems. System commands run the waitpid for you, but for open2/3 you need to do it yourself.

If you want to read error messages from your program then you need to use open3 instead of open2. This works in exactly the same way as open2, but just creates 3 filehandles. Bizarrely, the order of filehandles changes between open2 and open3 – this is a real classic for catching you out if you're not careful.

- Open2 = STDOUT , STDIN

- Open3 = STDIN, STDOUT, STDERR

Whoever thought that was a good idea??

Section 8: Cross Platform issues

The final section in this course is concerned with deploying your perl programs. Perl is available on many different platforms and therefore any code you write should be highly portable. There are a few things to think about when writing portable code, as well as other issues which will allow you to run your Perl scripts ever where perl is not installed.

Resource Locations

Probably the biggest thing you need to think about when writing portable programs is that resources on different systems are in different locations. It's very easy to make assumptions about what should be available and where which will work in all your tests but may fail on other systems.

The Perl interpreter

The most fundamental problem you're likely to have is that in order to be directly executable on Unix/Linux systems (and sometimes on Windows) the first line of your Perl program should contain the path to the Perl interpreter you're going to use. For Unix type systems it's a fair bet that there will be an interpreter at /usr/bin/perl, but you can't absolutely rely on this. It's not uncommon for perl to be in /usr/local/bin/perl or /opt/perl or some other location. It's also common for /usr/bin/perl to be a very old version which is kept for the sake of the system scripts which need it, but which isn't normally used to run other Perl scripts.

Unfortunately there isn't a nice way around this problem. You can run your scripts using perl scriptname.pl, but even that assumes that the correct version of perl is the first one in the path. It's a good idea to use /usr/bin/perl as a default, but include in whatever documentation you provide with your script the fact that the perl location may need to be changed.

File locations

It's very easy to make assumptions about file locations in your programs. For example if you have a text file in the same directory as your script it's tempting to write something like:

```perl
open (my $in,'text.txt') or die "Can't read text file:$!";
```

However by doing this you're making an assumption that the current working directory is the one your script is in. This may have been true when you tried it on your system, but different systems will set different default directories.

If you want to open a file you should, wherever possible either do a `chdir` first to move to a defined directory, or use a full path to the file when opening it.

If you want to open a file in a location relative to the location of your script then you can use the `FindBin` module to get the filesystem location of your Perl program.

```perl
#!perl
use warnings;
use strict;
use FindBin qw($Bin);

print "Script is located in $Bin\n";
```

The same sort of problems also occur should you use any external programs in system, exec or open statements. In these cases you're making an assumption that the program in question is in the default path for the environment you're working in. Again, some systems may have an incomplete path, or no path at all. The only way to deal with these situations is to use full paths to external programs, or to set your own path using the special perl variable `$ENV{PATH}`.

Text Encoding

As discussed previously different operating systems use different characters to indicate end of line in a text file. For the most part you don't need to worry about this as perl will handle the transition between line endings automatically. If you want to change this value for writing then just use the appropriate characters rather than \n (which will change depending on your platform). If you want to access raw line feed or carriage return symbols then you can use `\cM` for carriage return and `\cJ` for line feed.

Dependencies

If your code depends on the presence of certain modules to function correctly you need to think about how to handle this. Many of the most common modules are actually core modules and can be assumed to be present with any recent installation of perl, so you don't need to worry about those. If you want to check whether a certain module was included with perl then you can use the `Module::CoreList` module (which ironically isn't a core module!) to check if a module is a core module, and from which version it was introduced.

If you want your code to use non-core modules then you can include them in alongside your application. Just create a folder for the modules and copy them from their normal location to your folder. You can then add this folder to the list of locations perl searches for installed modules by using the a `use lib` statement at the top of your script. If you combine this with the `FindBin` module you can add your new folder wherever your script gets installed.

```perl
#!perl
use warnings;
use strict;
use FindBin qw($Bin);
use lib "$Bin/ExtraModules";
```

This code will add the ExtraModules directory to the module search path.

Although this approach offers a partial solution it won't work in some cases. Most perl modules are written purely in Perl, but many have a compiled C component which must go alongside them. Pure perl modules are completely portable, but any module with a compiled component is only guaranteed to work on the same platform (and strictly only if compiled with the same compiler used to compile perl). In these cases you just need to provide a mechanism to inform people using your script which modules they need to install.