Here is the basic perl program that we'll use to get started.

```
#!/usr/local/bin/perl
#
# Program to do the obvious
#
print 'Hello world.';           # Print a message
```

Each of the parts will be discussed in turn.

The first line

Every perl program starts off with this as its very first line:

```
#!/usr/local/bin/perl
```

although this may vary from system to system. This line tells the machine what to do with the file when it is executed (ie it tells it to run the file through Perl).

Comments and statements

Comments can be inserted into a program with the # symbol, and anything from the # to the end of the line is ignored (with the exception of the first line). The only way to stretch comments over several lines is to use a # on each line.

Everything else is a Perl statement which must end with a semicolon, like the last line above.

Simple printing

The `print` function outputs some information. In the above case it prints out the the literal string *Hello world.* and of course the statement ends with a semicolon.

You may find the above program produces an slightly unexpected result. So the next thing to do is to run it.

Type in the example program using a text editor, and save it. Emacs is a good editor to use for this because it has its own Perl mode which formats lines nicely when you hit tab (use `M-x perl-mode'). But as ever, use whichever you're most comfortable with.

After you've entered and saved the program make sure the file is executable by using the command

`chmod u+x` *`progname`*

at the UNIX prompt, where *progname* is the filename of the program. Now to run the program just type any of the following at the prompt.

```
perl progname
./progname
progname
```

If something goes wrong then you may get error messages, or you may get nothing. You can always run the program with warnings using the command

```
perl -w progname
```

at the prompt. This will display warnings and other (hopefully) helpful messages before it tries to execute the program. To run the program with a debugger use the command

```
perl -d progname
```

When the file is executed Perl first compiles it and then executes that compiled version. So after a short pause for compilation the program should run quite quickly. This also explains why you can get compilation errors when you execute a Perl file which consists only of text.

Make sure your program works before proceeding. The program's output may be slightly unexpected - at least it isn't very pretty. We'll look next at variables and then tie this in with prettier printing.

The most basic kind of variable in Perl is the *scalar variable*. Scalar variables hold both strings and numbers, and are remarkable in that strings and numbers are completely interchangable. For example, the statement

```
$priority = 9;
```

sets the scalar variable $priority to 9, but you can also assign a string to exactly the same variable:

```
$priority = 'high';
```

Perl also accepts numbers as strings, like this:

```
$priority = '9';
$default = '0009';
```

and can still cope with arithmetic and other operations quite happily.

In general variable names consists of numbers, letters and underscores, but they should not start with a number and the variable **$_** is special, as we'll see later. Also, Perl is case sensitive, so $a and $A are different.

Operations and Assignment

Perl uses all the usual C arithmetic operators:

```
$a = 1 + 2;     # Add 1 and 2 and store in $a
$a = 3 - 4;     # Subtract 4 from 3 and store in $a
$a = 5 * 6;     # Multiply 5 and 6
$a = 7 / 8;     # Divide 7 by 8 to give 0.875
$a = 9 ** 10;   # Nine to the power of 10
$a = 5 % 2;     # Remainder of 5 divided by 2
++$a;           # Increment $a and then return it
$a++;           # Return $a and then increment it
--$a;           # Decrement $a and then return it
$a--;           # Return $a and then decrement it
```

and for strings Perl has the following among others:

```
$a = $b . $c;   # Concatenate $b and $c
$a = $b x $c;   # $b repeated $c times
```

To assign values Perl includes

```
$a = $b;        # Assign $b to $a
$a += $b;       # Add $b to $a
$a -= $b;       # Subtract $b from $a
$a .= $b;       # Append $b onto $a
```

Note that when Perl assigns a value with **$a = $b** it makes a copy of $b and then assigns that to $a. Therefore the next time you change $b it will not alter $a.

Other operators can be found on the **perlop** manual page. Type **man perlop** at the prompt.

Interpolation

The following code prints *apples and pears* using concatenation:

```
$a = 'apples';
$b = 'pears';
print $a.' and '.$b;
```

It would be nicer to include only one string in the final print statement, but the line

```
print '$a and $b';
```

prints literally *$a and $b* which isn't very helpful. Instead we can use the double quotes in place of the single quotes:

```
print "$a and $b";
```

The double quotes force *interpolation* of any codes, including interpreting variables. This is a much nicer than our original statement. Other codes that are interpolated include special characters such as newline and tab. The code **\n** is a newline and **\t** is a tab.

A slightly more interesting kind of variable is the *array variable* which is a list of scalars (ie numbers and strings). Array variables have the same format as scalar variables except that they are prefixed by an `@` symbol. The statement

```
@food  = ("apples", "pears", "eels");
@music = ("whistle", "flute");
```

assigns a three element list to the array variable @food and a two element list to the array variable @music.

The array is accessed by using indices starting from 0, and square brackets are used to specify the index. The expression

```
$food[2]
```

returns *eels*. Notice that the **@** has changed to a **$** because *eels* is a scalar.

Array assignments

As in all of Perl, the same expression in a different context can produce a different result. The first assignment below explodes the @music variable so that it is equivalent to the second assignment.

```
@moremusic = ("organ", @music, "harp");
@moremusic = ("organ", "whistle", "flute", "harp");
```

This should suggest a way of adding elements to an array. A neater way of adding elements is to use the statement

```
push(@food, "eggs");
```

which pushes *eggs* onto the end of the array @food. To push two or more items onto the array use one of the following forms:

```
push(@food, "eggs", "lard");
push(@food, ("eggs", "lard"));
push(@food, @morefood);
```

The **push** function returns the length of the new list.

To remove the last item from a list and return it use the **pop** function. From our original list the **pop** function returns *eels* and @food now has two elements:

```
$grub = pop(@food);      # Now $grub = "eels"
```

It is also possible to assign an array to a scalar variable. As usual context is important. The line

```
$f = @food;
```

assigns the length of @food, but

```
$f = "@food";
```

turns the list into a string with a space between each element. This space can be replaced by any other string by changing the value of the special **$"** variable. This variable is just one of Perl's many special variables, most of which have odd names.

Arrays can also be used to make multiple assignments to scalar variables:

```
($a, $b) = ($c, $d);            # Same as $a=$c; $b=$d;
($a, $b) = @food;               # $a and $b are the first two
                                # items of @food.
($a, @somefood) = @food;        # $a is the first item of @food
                                # @somefood is a list of the
                                # others.
(@somefood, $a) = @food;        # @somefood is @food and
                                # $a is undefined.
```

The last assignment occurs because arrays are greedy, and @somefood will swallow up as much of @food as it can. Therefore that form is best avoided.

Finally, you may want to find the index of the last element of a list. To do this for the @food array use the expression

```
$#food
```

Displaying arrays

Since context is important, it shouldn't be too surprising that the following all produce different results:

```
print @food;    # By itself
print "@food";  # Embedded in double quotes
print @food.""; # In a scalar context
```

Here is the basic perl program which does the same as the UNIX **cat** command on a certain file.

```
#!/usr/local/bin/perl
#
# Program to open the password file, read it in,
# print it, and close it again.
```

```
$file = '/etc/passwd';          # Name the file
open(INFO, $file);              # Open the file
@lines = <INFO>;                # Read it into an array
close(INFO);                    # Close the file
print @lines;                   # Print the array
```

The **open** function opens a file for input (i.e. for reading). The first parameter is the *filehandle* which allows Perl to refer to the file in future. The second parameter is an expression denoting the filename. If the filename was given in quotes then it is taken literally without shell expansion. So the expression '~/notes/todolist' will not be interpreted successfully. If you want to force shell expansion then use angled brackets: that is, use <~/notes/todolist> instead.

The **close** function tells Perl to finish with that file.

There are a few useful points to add to this discussion on filehandling. First, the **open** statement can also specify a file for output and for appending as well as for input. To do this, prefix the filename with a **>** for output and a **>>** for appending:

```
open(INFO, $file);      # Open for input
open(INFO, ">$file");   # Open for output
open(INFO, ">>$file");  # Open for appending
open(INFO, "<$file");   # Also open for input
```

Second, if you want to print something to a file you've already opened for output then you can use the print statement with an extra parameter. To print a string to the file with the INFO filehandle use

```
print INFO "This line goes to the file.\n";
```

Third, you can use the following to open the standard input (usually the keyboard) and standard output (usually the screen) respectively:

```
open(INFO, '-');        # Open standard input
open(INFO, '>-');       # Open standard output
```

In the above program the information is read from a file. The file is the INFO file and to read from it Perl uses angled brackets. So the statement

```
@lines = <INFO>;
```

reads the file denoted by the filehandle into the array @lines. Note that the <INFO> expression reads in the file entirely in one go. This because the reading takes place in the context of an array variable. If @lines is replaced by the scalar $lines then only the next one line would be read in. In either case each line is stored complete with its newline character at the end.

More interesting possiblities arise when we introduce control structures and looping. Perl supports lots of different kinds of control structures which tend to be like those in C, but are very similar to Pascal, too. Here we discuss a few of them.

---

foreach

To go through each line of an array or other list-like structure (such as lines in a file) Perl uses the foreach structure. This has the form

```
foreach $morsel (@food)         # Visit each item in turn
                                # and call it $morsel
{
        print "$morsel\n";      # Print the item
        print "Yum yum\n";      # That was nice
}
```

The actions to be performed each time are enclosed in a block of curly braces. The first time through the block $morsel is assigned the value of the first item in the array @food. Next time it is assigned the value of the second item, and so until the end. If @food is empty to start with then the block of statements is never executed.

Testing

The next few structures rely on a test being true or false. In Perl any non-zero number and non-empty string is counted as true. The number zero, zero by itself in a string, and the empty string are counted as false. Here are some tests on numbers and strings.

```
$a == $b                # Is $a numerically equal to $b?
                        # Beware: Don't use the = operator.
$a != $b                # Is $a numerically unequal to $b?
$a eq $b                # Is $a string-equal to $b?
$a ne $b                # Is $a string-unequal to $b?
```

You can also use logical and, or and not:

```
($a && $b)              # Is $a and $b true?
($a || $b)              # Is either $a or $b true?
!($a)                   # is $a false?
```

---

for

Perl has a **for** structure that mimics that of C. It has the form

```
for (initialise; test; inc)
{
         first_action;
```

```
        second_action;
        etc
}
```

First of all the statement *initialise* is executed. Then while *test* is true the block of actions is executed. After each time the block is executed *inc* takes place. Here is an example for loop to print out the numbers 0 to 9.

```
for ($i = 0; $i < 10; ++$i)      # Start with $i = 1
                                 # Do it while $i < 10
                                 # Increment $i before repeating
{
        print "$i\n";
}
```

---

while and until

Here is a program that reads some input from the keyboard and won't continue until it is the correct password

```
#!/usr/local/bin/perl
print "Password? ";              # Ask for input
$a = <STDIN>;                    # Get input
chop $a;                         # Remove the newline at end
while ($a ne "fred")             # While input is wrong...
{
    print "sorry. Again? ";      # Ask again
    $a = <STDIN>;                # Get input again
    chop $a;                     # Chop off newline again
}
```

The curly-braced block of code is executed while the input does not equal the password. The **while** structure should be fairly clear, but this is the opportunity to notice several things. First, we can we read from the standard input (the keyboard) without opening the file first. Second, when the password is entered $a is given that value including the newline character at the end. The **chop** function removes the last character of a string which in this case is the newline.

To test the opposite thing we can use the **until** statement in just the same way. This executes the block repeatedly until the expression is true, not while it is true.

Another useful technique is putting the **while** or **until** check at the end of the statement block rather than at the beginning. This will require the presence of the **do** operator to mark the beginning of the block and the test at the end. If we forgo the *sorry. Again* message in the above password program then it could be written like this.

```
#!/usr/local/bin/perl
do
```

```
{
        "Password? ";           # Ask for input
        $a = <STDIN>;           # Get input
        chop $a;                # Chop off newline
}
while ($a ne "fred")            # Redo while wrong input
```

Of course Perl also allows if/then/else statements. These are of the following form:

```
if ($a)
{
        print "The string is not empty\n";
}
else
{
        print "The string is empty\n";
}
```

For this, remember that an empty string is considered to be false. It will also give an "empty" result if $a is the string *0*.

It is also possible to include more alternatives in a conditional statement:

```
if (!$a)                        # The ! is the not operator
{
        print "The string is empty\n";
}
elsif (length($a) == 1)         # If above fails, try this
{
        print "The string has one character\n";
}
elsif (length($a) == 2)         # If that fails, try this
{
        print "The string has two characters\n";
}
else                            # Now, everything has failed
{
        print "The string has lots of characters\n";
}
```

In this, it is important to notice that the elsif statement really does have an "e" missing.

One of the most useful features of Perl (if not *the* most useful feature) is its powerful string manipulation facilities. At the heart of this is the *regular expression* (RE) which is shared by many other UNIX utilities.

Regular expressions

A regular expression is contained in slashes, and matching occurs with the =~ operator. The following expression is true if the string *the* appears in variable $sentence.

```
$sentence =~ /the/
```

The RE is case sensitive, so if

```
$sentence = "The quick brown fox";
```

then the above match will be false. The operator **!**~ is used for spotting a non-match. In the above example

```
$sentence !~ /the/
```

is true because the string *the* does not appear in $sentence.

The $_ special variable

We could use a conditional as

```
if ($sentence =~ /under/)
{
        print "We're talking about rugby\n";
}
```

which would print out a message if we had either of the following

```
$sentence = "Up and under";
$sentence = "Best winkles in Sunderland";
```

But it's often much easier if we assign the sentence to the special variable **$_** which is of course a scalar. If we do this then we can avoid using the match and non-match operators and the above can be written simply as

```
if (/under/)
{
        print "We're talking about rugby\n";
}
```

The **$_** variable is the default for many Perl operations and tends to be used very heavily.

More on REs

In an RE there are plenty of special characters, and it is these that both give them their power and make them appear very complicated. It's best to build up your use of REs slowly; their creation can be something of an art form.

Here are some special RE characters and their meaning

```
.       # Any single character except a newline
^       # The beginning of the line or string
$       # The end of the line or string
*       # Zero or more of the last character
+       # One or more of the last character
?       # Zero or one of the last character
```

and here are some example matches. Remember that should be enclosed in /.../ slashes to be used.

```
t.e     # t followed by anthing followed by e
        # This will match the
        #                 tre
        #                 tle
        # but not te
        #         tale
^f      # f at the beginning of a line
^ftp    # ftp at the beginning of a line
e$      # e at the end of a line
tle$    # tle at the end of a line
und*    # un followed by zero or more d characters
        # This will match un
        #                 und
        #                 undd
        #                 unddd (etc)
.*      # Any string without a newline. This is because
        # the . matches anything except a newline and
        # the * means zero or more of these.
^$      # A line with nothing in it.
```

There are even more options. Square brackets are used to match any one of the characters inside them. Inside square brackets a - indicates "between" and a ^ at the beginning means "not":

```
[qjk]           # Either q or j or k
[^qjk]          # Neither q nor j nor k
[a-z]           # Anything from a to z inclusive
[^a-z]          # No lower case letters
[a-zA-Z]        # Any letter
[a-z]+          # Any non-zero sequence of lower case letters
```

At this point you can probably skip to the end and do at least most of the exercise. The rest is mostly just for reference.

A vertical bar | represents an "or" and parentheses **(**...**)** can be used to group things together:

```
jelly|cream     # Either jelly or cream
(eg|le)gs       # Either eggs or legs
(da)+           # Either da or dada or dadada or...
```

Here are some more special characters:

```
\n              # A newline
\t              # A tab
\w              # Any alphanumeric (word) character.
                # The same as [a-zA-Z0-9_]
\W              # Any non-word character.
                # The same as [^a-zA-Z0-9_]
\d              # Any digit. The same as [0-9]
\D              # Any non-digit. The same as [^0-9]
\s              # Any whitespace character: space,
                # tab, newline, etc
\S              # Any non-whitespace character
\b              # A word boundary, outside [] only
\B              # No word boundary
```

Clearly characters like **$**, |, **[**, **)**, \, / and so on are peculiar cases in regular expressions. If you want to match for one of those then you have to preceed it by a backslash. So:

```
\|              # Vertical bar
\[              # An open square bracket
\)              # A closing parenthesis
\*              # An asterisk
\^              # A carat symbol
\/              # A slash
\\              # A backslash
```

and so on.

Some example REs

As was mentioned earlier, it's probably best to build up your use of regular expressions slowly. Here are a few examples. Remember that to use them for matching they should be put in /.../ slashes

```
[01]            # Either "0" or "1"
\/0             # A division by zero: "/0"
\/ 0            # A division by zero with a space: "/ 0"
\/\s0           # A division by zero with a whitespace:
                # "/ 0" where the space may be a tab etc.
\/ *0           # A division by zero with possibly some
                # spaces: "/0" or "/ 0" or "/  0" etc.
\/\s*0          # A division by zero with possibly some
```

```
                # whitespace.
\/\s*0\.0*      # As the previous one, but with decimal
                # point and maybe some 0s after it. Accepts
                # "/0." and "/0.0" and "/0.00" etc and
                # "/ 0." and "/  0.0" and "/   0.00" etc.
```

As well as identifying regular expressions Perl can make substitutions based on those matches. The way to do this is to use the **s** function which is designed to mimic the way substitution is done in the vi text editor. Once again the match operator is used, and once again if it is omitted then the substitution is assumed to take place with the **$_** variable.

To replace an occurrence of *london* by *London* in the string $sentence we use the expression

```
$sentence =~ s/london/London/
```

and to do the same thing with the **$_** variable just

```
s/london/London/
```

Notice that the two regular expressions (*london* and *London*) are surrounded by a total of three slashes. The result of this expression is the number of substitutions made, so it is either 0 (false) or 1 (true) in this case.

Options

This example only replaces the first occurrence of the string, and it may be that there will be more than one such string we want to replace. To make a global substitution the last slash is followed by a **g** as follows:

```
s/london/London/g
```

which of course works on the **$_** variable. Again the expression returns the number of substitutions made, which is 0 (false) or something greater than 0 (true).

If we want to also replace occurrences of *lOndon*, *lonDON*, *LoNDoN* and so on then we could use

```
s/[Ll][Oo][Nn][Dd][Oo][Nn]/London/g
```

but an easier way is to use the **i** option (for "ignore case"). The expression

```
s/london/London/gi
```

will make a global substitution ignoring case. The **i** option is also used in the basic /.../ regular expression match.

Remembering patterns

It's often useful to remember patterns that have been matched so that they can be used again. It just so happens that anything matched in parentheses gets remembered in the variables **$1**,...,**$9**. These strings can also be used in the same regular expression (or substitution) by using the special RE codes **\1**,...,**\9**. For example

```
$_ = "Lord Whopper of Fibbing";
s/([A-Z])/:\1:/g;
print "$_\n";
```

will replace each upper case letter by that letter surrounded by colons. It will print *:L:ord :W:hopper of :F:ibbing*. The variables **$1**,...,**$9** are read-only variables; you cannot alter them yourself.

As another example, the test

```
if (/(\b.+\b) \1/)
{
        print "Found $1 repeated\n";
}
```

will identify any words repeated. Each **\b** represents a word boundary and the **.+** matches any non-empty string, so **\b.+\b** matches anything between two word boundaries. This is then remembered by the parentheses and stored as **\1** for regular expressions and as **$1** for the rest of the program.

The following swaps the first and last characters of a line in the **$_** variable:

```
s/^(.)(.*)(.)$/\3\2\1/
```

The **^** and **$** match the beginning and end of the line. The **\1** code stores the first character; the **\2** code stores everything else up the last character which is stored in the **\3** code. Then that whole line is replaced with **\1** and **\3** swapped round.

After a match, you can use the special read-only variables **$`** and **$&** and **$'** to find what was matched before, during and after the seach. So after

```
$_ = "Lord Whopper of Fibbing";
/pp/;
```

all of the following are true. (Remember that eq is the string-equality test.)

```
$` eq "Lord Wo";
$& eq "pp";
$' eq "er of Fibbing";
```

Finally on the subject of remembering patterns it's worth knowing that inside of the slashes of a match or a substitution variables are interpolated. So

```
$search = "the";
s/$search/xxx/g;
```

will replace every occurrence of *the* with *xxx*. If you want to replace every occurence of *there* then you cannot do **s/$searchre/xxx/** because this will be interpolated as the variable $searchre. Instead you should put the variable name in curly braces so that the code becomes

```
$search = "the";
s/${search}re/xxx/;
```

Translation

The **tr** function allows character-by-character translation. The following expression replaces each *a* with *e*, each *b* with *d*, and each *c* with *f* in the variable $sentence. The expression returns the number of substitutions made.

```
$sentence =~ tr/abc/edf/
```

Most of the special RE codes do not apply in the tr function. For example, the statement here counts the number of asterisks in the $sentence variable and stores that in the $count variable.

```
$count = ($sentence =~ tr/*/*/);
```

However, the dash is still used to mean "between". This statement converts **$_** to upper case.

```
tr/a-z/A-Z/;
```

A very useful function in Perl is **split**, which splits up a string and places it into an array. The function uses a regular expression and as usual works on the **$_** variable unless otherwise specified.

The **split** function is used like this:

```
$info = "Caine:Michael:Actor:14, Leafy Drive";
@personal = split(/:/, $info);
```

which has the same overall effect as

```
@personal = ("Caine", "Michael", "Actor", "14, Leafy Drive");
```

If we have the information stored in the **$_** variable then we can just use this instead

```
@personal = split(/:/);
```

If the fields are divided by any number of colons then we can use the RE codes to get round this. The code

```
$_ = "Capes:Geoff::Shot putter:::Big Avenue";
@personal = split(/:+/);
```

is the same as

```
@personal = ("Capes", "Geoff",
             "Shot putter", "Big Avenue");
```

But this:

```
$_ = "Capes:Geoff::Shot putter:::Big Avenue";
@personal = split(/:/);
```

would be like

```
@personal = ("Capes", "Geoff", "",
             "Shot putter", "", "", "Big Avenue");
```

A word can be split into characters, a sentence split into words and a paragraph split into sentences:

```
@chars = split(//, $word);
@words = split(/ /, $sentence);
@sentences = split(/\./, $paragraph);
```

In the first case the null string is matched between each character, and that is why the @chars array is an array of characters - ie an array of strings of length 1.

Ordinary list arrays allow us to access their element by number. The first element of array @food is $food[0]. The second element is $food[1], and so on. But Perl also allows us to create arrays which are accessed by string. These are called *associative arrays*.

To define an associative array we use the usual parenthesis notation, but the array itself is prefixed by a % sign. Suppose we want to create an array of people and their ages. It would look like this:

```
%ages = ("Michael Caine", 39,
         "Dirty Den", 34,
         "Angie", 27,
         "Willy", "21 in dog years",
         "The Queen Mother", 108);
```

Now we can find the age of people with the following expressions

```
$ages{"Michael Caine"};         # Returns 39
$ages{"Dirty Den"};             # Returns 34
$ages{"Angie"};                 # Returns 27
$ages{"Willy"};                 # Returns "21 in dog years"
$ages{"The Queen Mother"};      # Returns 108
```

Notice that like list arrays each **%** sign has changed to a **$** to access an individual element because that element is a scalar. Unlike list arrays the index (in this case the person's name) is enclosed in curly braces, the idea being that associative arrays are fancier than list arrays.

An associative array can be converted back into a list array just by assigning it to a list array

variable. A list array can be converted into an associative array by assigning it to an associative array variable. Ideally the list array will have an even number of elements:

```
@info = %ages;          # @info is a list array. It
                        # now has 10 elements
$info[5];               # Returns the value 27 from
                        # the list array @info
%moreages = @info;      # %moreages is an associative
                        # array. It is the same as %ages
```

Operators

Associative arrays do not have any order to their elements (they are just like hash tables) but is it possible to access all the elements in turn using the **keys** function and the **values** function:

```
foreach $person (keys %ages)
{
        print "I know the age of $person\n";
}
foreach $age (values %ages)
{
        print "Somebody is $age\n";
}
```

When **keys** is called it returns a list of the keys (indices) of the associative array. When **values** is called it returns a list of the values of the array. These functions return their lists in the same order, but this order has nothing to do with the order in which the elements have been entered.

When **keys** and **values** are called in a scalar context they return the number of key/value pairs in the associative array.

There is also a function **each** which returns a two element list of a key and its value. Every time **each** is called it returns another key/value pair:

```
while (($person, $age) = each(%ages))
{
        print "$person is $age\n";
}
```

Environment variables

When you run a perl program, or any script in UNIX, there will be certain environment variables set. These will be things like USER which contains your username and DISPLAY which specifies which screen your graphics will go to. When you run a perl CGI script on the World Wide Web

there are environment variables which hold other useful information. All these variables and their values are stored in the associative **%ENV** array in which the keys are the variable names. Try the following in a perl program:

```
print "You are called $ENV{'USER'} and you are ";
print "using display $ENV{'DISPLAY'}\n";
```

Like any good programming langauge Perl allows the user to define their own functions, called *subroutines*. They may be placed anywhere in your program but it's probably best to put them all at the beginning or all at the end. A subroutine has the form

```
sub mysubroutine
{
        print "Not a very interesting routine\n";
        print "This does the same thing every time\n";
}
```

regardless of any parameters that we may want to pass to it. All of the following will work to call this subroutine. Notice that a subroutine is called with an **&** character in front of the name:

```
&mysubroutine;          # Call the subroutine
&mysubroutine($_);      # Call it with a parameter
&mysubroutine(1+2, $_); # Call it with two parameters
```

Parameters

In the above case the parameters are acceptable but ignored. When the subroutine is called any parameters are passed as a list in the special **@_** list array variable. This variable has absolutely nothing to do with the **$_** scalar variable. The following subroutine merely prints out the list that it was called with. It is followed by a couple of examples of its use.

```
sub printargs
{
        print "@_\n";
}

&printargs("perly", "king");    # Example prints "perly king"
&printargs("frog", "and", "toad"); # Prints "frog and toad"
```

Just like any other list array the individual elements of **@_** can be accessed with the square bracket notation:

```
sub printfirsttwo
{
```

```
        print "Your first argument was $_[0]\n";
        print "and $_[1] was your second\n";
}
```

Again it should be stressed that the indexed scalars **$_[0]** and **$_[1]** and so on have nothing to with the scalar **$_** which can also be used without fear of a clash.

Returning values

Result of a subroutine is always the last thing evaluated. This subroutine returns the maximum of two input parameters. An example of its use follows.

```
sub maximum
{
        if ($_[0] > $_[1])
        {
                $_[0];
        }
        else
        {
                $_[1];
        }
}

$biggest = &maximum(37, 24);    # Now $biggest is 37
```

The &printfirsttwo subroutine above also returns a value, in this case 1. This is because the last thing that subroutine did was a **print** statement and the result of a successful **print** statement is always 1.

Local variables

The **@_** variable is local to the current subroutine, and so of course are **$_[0]**, **$_[1]**, **$_[2]**, and so on. Other variables can be made local too, and this is useful if we want to start altering the input parameters. The following subroutine tests to see if one string is inside another, spaces not withstanding. An example follows.

```
sub inside
{
        local($a, $b);                  # Make local variables
        ($a, $b) = ($_[0], $_[1]);      # Assign values
        $a =~ s/ //g;                   # Strip spaces from
        $b =~ s/ //g;                   #   local variables
        ($a =~ /$b/ || $b =~ /$a/);     # Is $b inside $a
```

```
                                        #   or $a inside $b?
}

&inside("lemon", "dole money");         # true
```

In fact, it can even be tidied up by replacing the first two lines with

```
local($a, $b) = ($_[0], $_[1]);
```
