PERL Programming

Exercise Guide

Contents

Perl Programming

Exercises

These exercises are meant to consolidate the information learnt in each section. Always remember that your Instructor is there to help. Some of the questions are deliberately vague to encourage you to think (that's our story), if you cannot understand what an exercise wants you to do, then please ask the Instructor right away.

General Format

Exercises are given for each chapter, followed by suggested solutions.

Perl code lines, program output, and replies to prompts, are shown in `Courier New` font.

Many of these exercises have alternative instructions for running on UNIX and Microsoft Windows, but most run unaltered on both.

Optional Exercises

Some chapters have optional exercises – they really are optional! They are there partly because everyone works at a different pace, and partly to complete back home. If you start an optional exercise, expect your Instructor to interrupt with the next chapter (or, if you are lucky, lunch).

In practice *all* the exercises are optional really. If you want to try something else then feel free.

Solution code

Solutions are provided in this guide, and the sub-directory *`labsolutions`* in your home directory. This also includes data files which you may find useful. Example output is sometimes given, but be aware that the exact error message text can vary between Perl releases.

Please remember *tim-toady* (TIMTOWTDI – There Is More Than One Way To Do It). If your solution works then it is as good as ours. Our solution is only *one* way to do it!

Editors

Any text editor can be used for this course, it really depends on what you are familiar with.

On ***Linux*** the X-Windows GUI offers a number of text editors, and which to use is really a matter of personal preference. There are also a number of different GUI schemes, the default on Red Hat Linux is called GNOME, and their presentation differs between releases.

`Kate`

In a terminal session type:

```
kate 2>/dev/null &
```

To configure **`Kate`** for Perl, choose menu item **Document**, then **Highlight Mode**, **Scripts**, **Perl**.

The KDE Advanced Editor is also very good, and is a subset of Kate, it's Highlight Mode menu is under 'Settings'.

`DDD`

DDD is a debugger environment which uses **`vi(1)`** as its editor. It can usually be run from a 'Programming' menu.

To configure **`DDD`** for Perl, click on **Edit**, then **Preferences**. Click the **Startup** button, then under **Debugger Type**, select **Perl**. Click the **OK** box and a dialog box will be displayed asking if you wish to restart `DDD`, click **Yes** and `DDD` will restart in Perl mode. Be sure to **Save Options** under the **Edit** menu.

On ***UNIX*** character terminals the usual editors are available, and most seasoned UNIX developers are 'happy' with `vi(1)` or emacs(1).

On ***Microsoft Windows*** there are only two editors normally bundled with the operating system, **`Notepad`** and **`Wordpad`**.

`Wordpad` is recommended since it can handle text files where the record delimiter is a single new-line character ("\n"), whereas **`Notepad`** requires a carriage-return before the new-line ("\r\n"). The perl interpreter on Windows will handle both formats. Whichever you use, please remember to Save As … a Text Document.

More sophisticated products, including Microsoft Visual Studio, can improve productivity, although even a relatively simple editor like **`TextPad`** is a great improvement over **`Wordpad`**.

Chapter 1: Language Basics

These exercises start with the basics: getting your first Perl program running. Their purpose is to get you familiar with the environment used in the rest of this course. Please take this opportunity to investigate the online Perl documentation. This will be your main source of reference (documentation on all platforms is similar), so practice in using it is worthwhile.

On ***Linux***, if you wish you may alter your PATH in `.profile` to include the current directory:

```
PATH=${PATH}:.
```

On ***Microsoft Windows*** the current directory is searched before the %PATH% variable used.

Exercise 1

Type in a 'Hello World' script, using the examples from the slides.
Run it using:

```
perl -w hello.pl
```

On ***Microsoft Windows***, use a Command Line session (*start – Run* – type `cmd`).

Exercise 2

Find out which version of perl you just used. Get the version number using:

```
perl -v
```

Optional:
If the low-level internals of Perl interests you, try:

```
perl -V
```

that is an uppercase V.

Exercise 3

On *UNIX*:

Alter hello.pl to use the `#!` method. Make sure you set execute permissions:

```
chmod +x hello.pl
```

will do. Run it using:

```
./hello.pl
```

On *Microsoft Windows*:

Launch hello.pl from Windows Explorer, did you have any problems reading the message? You may wish to run your perl from a Command Line session for the rest of the course, but can you suggest an alternative for running from Explorer?

Hint, we have not covered I/O yet, but you can pause on keyboard input with the single statement:

```
<STDIN>
```

Exercise 4

If you have been using the –w (warnings) switch when running perl, that's good, but now run the script without it. Modify your `hello.pl` script to include the line:

```
print Hello;
```

Note: no quotes characters, and no \n.
Now run the script (without warnings), what happened? Not a lot!

Change your script to switch warnings on, there are several way to do this, and run it again.

Chapter 1: Language Basics - Solutions

Exercise 1 Solution

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

Exercise 2 Solution

You will most likely find you used perl version 5.8.8 or later. For example:

```
This is perl, v5.8.8 built for i386-linux-thread-multi

Copyright 1987-2006, Larry Wall

...
```

Exercise 3 Solution

On ***UNIX*****,** your `hello.pl` should look something like this:

```
#!/usr/bin/perl -w
print "Hello, world\n";
```

If you have any problems, make sure you:

a) Specified the `#!` on the first line of your script
b) There are no characters before the #!
c) You've specified the full search path to perl
d) You've marked to file executable using `chmod +x`

On ***Microsoft Windows***, when hello.pl is launched from Windows Explorer the console window flashes by! You can run perl from the Command Line, but there is no equivalent to the UNIX `#!`, so you may always have to explicitly call `perl`, with the script name as a parameter.

Alternatively, your `hello.pl` could look something like this:

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

print "Press <Return> to continue...";
$reply = <STDIN>;
```

The prompt will stop the console window from closing before you get chance to read it.

Exercise 4 Solution

Nothing should have been printed from the line `print Hello;`, not even an error message! Warnings can be switched on in a number of ways:

at the command line:	`perl -w hello.pl`
at the #! line:	`#!/usr/bin/perl -w`
as a pragma:	`use warnings;`

With warnings set on, you should have got something like:

```
$ perl -w hello.pl
Name "main::Hello" used only once: possible typo at hello.pl
line 3.
Filehandle main::Hello never opened at hello.pl line 3.
```

Perl is guessing that Hello is the name of a file handle, which we will look at later. It's not happy though, because it detects that Hello has only been used once. If we forget to enclose quote characters around text Perl regards this as a "bare-word", and has to guess what it is.

The `-w` option prints warnings for a large number of programming errors at compile time. Taking note of these (even when they are annoying) can avoid difficult to spot run-time errors. See also `perldiag`.

Chapter 2: Fundamental Variables

These exercises work with data types: defining variables of varying types, assigning literal values, and string interpolation.

Some exercises require you to ask the user for data. This is achieved with the `print` command to show the prompt, then by reading <STDIN>, the standard input channel, into a variable. Finally the line termination character/s are removed using `chomp`. We will be covering all this later, but for now the method is shown in this example:

```
print "Enter a first name: ";        # The prompt (no new-line)
$name = <STDIN>;                     # Read from standard input
chomp $name;                         # Remove new-line character
```

Exercise 1

Using the code fragment above as a guide, prompt the user for a first name then a last name, storing these in different scalar variables.

Now transfer the contents of both these scalars into an array, using a list assignment. Print the array with interpolation, not forgetting the new-line character.

Now put the first and last names into a hash, using the keys `'first'` and `'last'`. Is it possible to print the hash?

Dump the hash into the array, using a single assignment, and print the array again. Can you explain what is printed?

Exercise 2

Create an array `@numbers` with the five numbers: 1, 5, 9, 3.14159 and 2. Print it with and without interpolation (`"@numbers"` and `@numbers`), and note differences.

Using the predefined variable `$"`, print each element of the array on a different line using a single `print` statement.

Exercise 3

Write a Perl program that reads a string `$text` (see above), then print it using each of the interpolations, `\l \L \u \U \Q` (see Page 9 of this chapter). For example, using \U means: `print "\U$text\n";`.

Try this with various input values:

(a) A number, e.g. 123
(b) A name, e.g. John
(c) A sentence, e.g. How are you today?
(d) Non-alphanumeric character sequences, e.g. * & % $ \

Optional Exercises

Exercise 1

What is the difference between '\n' and "\n" (if you do not know, try it!).

Create a variable that contains just a new-line - without using interpolation, and use it in a `print` statement.

Exercise 2

Open the file **`labsolutions/Ex2.2.pl`**. The machines used in the classroom are each used by a specific user, and this relationship is already coded for you in the hash `%machines`. Print the machine name for the current user, using the environment variable LOGNAME (on ***UNIX***) or USERNAME (on ***Microsoft Windows***) to find the username.

Can you list the entire hash in one `print` statement?

Exercise 3

A ***here document*** is a text fragment spanning multiple lines that can be included in Perl. They start with the special characters <<, followed by a label name. The label gives the unique text that terminates the data. The example below shows a here document:

```
$num = 5;
$a = <<'_END_';
This is the first line
The number is $num
This is the last line
_END_

print "[$a]\n";
```

You can choose the terminating text, in this case `_END_`; which must be on the start of a line, all by itself. The semicolon to end the here document must follow the label name on the first line.

(a) Enter and run the program with (as always) warnings set. Can you explain the warning?

(b) Try the following variation on the program (note the quotes around `_END_`):

```
$num = 5;
$a = <<"_END_";
This is the first line
The number is $num
This is the last line
_END_

print "[$a]\n";
```

Chapter 2: Fundamental Variables - Solutions

Exercise 1 Solution

Here is our solution, with a few comments:

```
print "Enter a first name: ";
$first = <STDIN>;
chomp $first;

print "Enter a last name: ";
$last = <STDIN>;
chomp $last;
```

Notice the syntax used for the list assignment.

```
# Transfer the scalars into an array
@array = ($first, $last);
```

Interpolation on an array gives (by default) spaces between each element.

```
print "@array\n";
```

Here we dynamically create a new hash. We do not need to quote the key names unless we are using the `strict` pragma.

```
# Now a hash
$hash{first} = $first;
$hash{last}  = $last;
```

Printing a hash is not so good, after all hashes are not meant to be printed! If you tried using interpolation, that would not have worked.

```
# Not too good!
print %hash, "\n";
```

Dumping the hash back into the array allows us to print it using interpolation. Did you notice the order of the key/value pairs?

```
# Now back to an array
@array = %hash;
print "@array\n";
```

Exercise 2 Solution

The program below will show what's going on:

```
@numbers = (1, 5, 9, 3.14159, 2);

print "With interpolation: @numbers\n";
print "Without interpolation: ", @numbers, "\n";
```

If you run the program, it will generate the following output:

```
With interpolation: 1 5 9 3.14159 2
Without interpolation: 1593.141592
```

As you can see, when you print an array with interpolation, a space is inserted between each pair of elements. When you don't use interpolation, no space is inserted.

The predefined variable $" sets the value used between elements when interpolating arrays. If we set it to "\n" (don't forget the ") then each element gets displayed on a different line.

```
$" = "\n";
print "@numbers\n";
```

gives the following output:

```
1
5
9
3.14159
2
```

Exercise 3 Solution

Your program should look something like this:

```
print "Enter a text: ";
$text = <STDIN>;
chomp $text;

print "Using no commands: $text\n";
print "Using the \\l command: \l$text\n";
print "Using the \\u command: \u$text\n";
print "Using the \\L command: \L$text\n";
print "Using the \\U command: \U$text\n";
print "Using the \\Q command: \Q$text\E\n";
```

When runs, it will generate output like this:

```
$ Ch2Ex3.pl
Enter a text: Hello
Using no commands: Hello
Using the \l command: hello
Using the \u command: Hello
Using the \L command: hello
Using the \U command: HELLO
Using the \Q command: Hello

$ Ch2Ex3.pl
Enter a text: bye
Using no commands: bye
Using the \l command: bye
Using the \u command: Bye
Using the \L command: bye
Using the \U command: BYE
Using the \Q command: bye

$ Ch2Ex3.pl
Enter a text: Special *.+-()" characters
Using no commands: Special *.+-()" characters
Using the \l command: special *.+-()" characters
Using the \u command: Special *.+-()" characters
Using the \L command: special *.+-()" characters
Using the \U command: SPECIAL *.+-()" CHARACTERS
Using the \Q command: Special\ \*\.\+\-\(\)\"\ characters
```

The various interpolation commands have the following meaning:

`\l` Turn the next character to lower case
`\u` Turn the next character to upper case
`\L` Turn the rest of the string, or all text until the `\E` character into lowercase
`\U` Turn the rest of the string, or all text until the `\E` character into uppercase
`\Q` Protect all special characters in the rest of the string, or all text until the `\E` character, by escaping them with a backslash (needed for regular expressions.)

Optional Exercises

Optional Exercise 1 Solution

Single quotes around a text value indicate a *non-interpolated* string. The expression '\n' defines a string of two characters, a backslash and the character **n**. If you want \n to be interpreted as a new-line character, you must use interpolation, indicated with double quotes. So, "\n" indicates a string that contains one character, the new-line character. If you want to achieve this without using interpolation, just insert a <RETURN> into a non-interpolated string.

The difference between '\n' and "\n" can be illustrated:

```
$text = '\n';
print "\'\\n\': <newline> $text <newline>\n\n";
```

Produces the output:

```
'\n': <newline> \n <newline>
```

Whereas using double quotes:

```
$text = "\n";
print "\"\\n\": <newline> $text <newline>\n\n";
```

Produces the output:

```
"\n": <newline>
 <newline>
```

Create a variable that contains a single new-line character :

```
$text = '
';

print "Variable: <newline> $text <newline>\n";
```

Produces the output:

```
Variable: <newline>
 <newline>
```

Optional Exercise 2 Solution

There is more than way to do this (of course). We could assign the value of the environment variable to a scalar, and then use this as a key to our hash:

```
$user = $ENV{'LOGNAME'};
print "Machine is: $machines{$user}\n";
```

or we could cut out the middleman and use the ENV hash direct:

```
print "Machine is: $machines{$ENV{'LOGNAME'}}\n";
```

If we print a hash in a single statement, we cannot use interpolation:

```
print %machines, "\n"
```

Optional Exercise 3 Solution

Perl 'here documents' start on the line following the << and end on the line with the end text. Interpolation of here documents is optional, and can be determined by the type of quotes used:

By using single quotes, no interpolation is performed. This way, you can include all characters in a string without having to do any escapes. The warning was generated because single-quoting the document label preserves the characters '$num' as text.

By using double quotes, interpolation is performed. Strange characters must be escaped, but variables can be included easily. Not quoting at all is equal to using double quotes: interpolation is performed.

You can use `here` documents when you want to embed long literal texts in your program, e.g. to embed bits of perl code for `eval` (see a later chapter) or bits of HTML for a CGI program. The example below shows how to generate a simple personal html page:

```
# Generate a simple HTML page
print "What is your first name: ";
$first = <STDIN>; chomp $first;
print "What is your last name: ";
$last = <STDIN>; chomp $last;
print "What is your job?: ";
$job = <STDIN>; chomp $job;

print <<_HTML_END_;
<HTML>
<HEAD>
<TITLE>Personal page for $first $last</TITLE>
</HEAD>
<-- Automatically generated -->
<BODY>
<H1>Personal page for $first $last</H1>
      My first name is <I>$first</I>;
      my last name is <I>$last</I>;
      my job is <b>$job</b>.
</BODY>
</HTML>
_HTML_END_
```

Chapter 3: Arrays and Hashes

Exercise 1

The command line arguments for a script are available within the program from a pre-defined system variable `@ARGV`.

Write a program, echo.pl, that prints all its command line arguments to the screen. Call the program with a shell wildcard (glob construct), for example:

On ***UNIX***:

```
echo.pl /etc/*
```

On ***Microsoft Windows NT/2000***:

```
perl -w echo.pl c:\winnt\*
```

On ***Microsoft Windows 95/98/Me/XP***:

```
perl -w echo.pl c:\windows\*
```

What happens? Which program is responsible for the handling of wildcards?

Exercise 2

Create a hash that contains the following *`first-name => last-name`* pairs:

```
John Norman
Robert Anson
Christopher Fowler
Robert Harris
Dan Simmons
```

Now add code to prompt for a first name. Use the reply to obtain the last name from the hash, and print the full name.

What happened to the common first names (Robert)? Hint: try swapping them around.

Exercise 3

This is a "pencil and paper" exercise.

(a) Explain what the following mean and what difference, if any, there will be in the contents of the array after each assignment, and the number of elements.

```
@a = qw(a b c);

@a = qq(a b c);

@a = ('a', 'b', 'c');

@a = qw('a', 'b', 'c');
```

(b) Explain what the following mean and what difference, if any, there will be in the contents of the array after each assignment, and the number of elements?

```
@a = qw(11 12 13);

@a = (11, 12, 13);

@a = (11..13);

@a = qw(11..13);

@a = qq(11..13);

@a = qw(011 012 013);

@a = (011, 012, 013);
```

Exercise 4

Create an array from a list of eight computer names:

```
yogi, booboo, grizzly, rupert, baloo, teddy, bungle, care
```

now write code to alter the array:

i. Change `'baloo'` to `'greppy'`
ii. Add `'fozzie'` to the end of the array
iii. Print the resulting array, and its length

Exercise 5

This question is concerned with array slices.

Using the array created in the previous exercise:

- In one statement, copy the last four elements of the array into a list of scalar variables, one for each element (you can call the scalars anything you like). Print the four scalars.
- In one statement, copy the first element of the array into a scalar and the next three elements into a new array called `@new`. Print the scalar and the new array.
- In one statement, add `@new` onto the end of the array (this does not involve a slice), then print the result.

Optional Exercises

Exercise 1

These exercises work with the $# array prefix (`$#array`):

(a) Create an array with 10 elements
(b) Print the highest index of the array
(c) Assign beyond the end of the array, to an element at index 20
(d) Save the current highest index in a scalar, and print it.
(e) Set the array size to 5 elements (index 4)
(f) Print the array
(g) Set the array size back to the previous size (using the scalar created in (d)).
(h) Print the array

Exercise 2

The function `localtime` returns a list consisting of:

seconds, minutes, hour, day, month, year, weekday, year-day, daylight-saving-flag

The weekday element is of the form 0..6, where 0 is Sunday. Can you write a single line of code to call `localtime` and print the day of the week in English (i.e. Monday, Tuesday…)?

Chapter 3: Arrays and Hashes – Solutions

Exercise 1 Solution

To display the command line arguments, print `@ARGV` using interpolation:

```
print "@ARGV\n";
```

The results will depend on the operating system:

On ***UNIX***:

A long list of files is returned.

On ***Microsoft Windows NT/2000***:

c:\winnt*

On ***Microsoft Windows 95/98/Me/XP***:

c:\windows*

When a command is invoked with a wildcard on UNIX the shell expands this to a list of files. The Microsoft Windows shells (COMMAND.COM or CMD.EXE) do not support this, so the program would have to do the expansion itself.

Exercise 2 Solution

If you use a key multiple times within the same hash, the last instance wins. This is similar to assigning to the same array element multiple times: each time you use the same key (array: index)
again, the previous value is overwritten. The example program below shows how you create a hash, and ask for a key to find the value.

```
%names = (John        => 'Norman',
          Robert      => 'Anson',
          Christopher => 'Fowler',
          Robert      => 'Harris',
          Dan         => 'Simmons');

print "Enter a first name: ";
$first = <STDIN>;
chomp $first;
print "Full name: $first $names{$first}\n";
```

To do the same thing for last names, you would need a reverse function that finds the first name for each last name. This requires an additional hash, or the use of a `for` or `while` loop (which we haven't done yet!).

Exercise 3 Solution

(a) Here are the results from running this code::

`@a = qw(a b c)` gives `a b c`
This line assigns a three element list to an array, which will have the same number of elements.

`@a = qq(a b c)` gives `a b c`
Although this appears to produce the same result as the previous line, in this example all the elements are contained in the first element of the array, `$a[0]`! The function `qq` places double-quotes (interpolation) around the whole string thus: `"a b c"`, which is a single scalar, whereas `qw` places a single-quote and comma, between each element of the list.

`@a = ('a', 'b', 'c')` gives `a b c`
This example produces the same result as the first.

`@a = qw('a', 'b', 'c')` gives `'a', 'b', 'c'`
This is a classic mistake! We get a three element array, but end up with the quotes and commas imbedded in each element. Use either `qw`, or quote and comma separators, not both! If you ran this code with warnings switched on (as you should) you will get something like:

```
Possible attempt to separate words with commas at …
```

(b) Here are the results from running this code::

`@a = qw(11 12 13)` gives `11 12 13`
This assigns a list with three *text* values to the array.

`@a = (11, 12, 13)` gives `11 12 13`
`@a = (11..13)` gives `11 12 13`
These two have the same meaning, both assign a list value with three *numbers* to the array. Notice the subtle difference with the previous line, qw, or single quotes, applied to numbers produce text, bare numbers are numeric. In practice this doesn't make a difference with decimal numbers, since the values will be automatically converted from text to number and vice versa. When special characters are used, like 0x prefix (hex.), engineering notation, or underscores, there is a difference.

`@a = qw(11..13)` gives `11..13`
`@a = qq(11..13)` gives `11..13`
Another classic mistake, but this time you will not get a warning! The functions `qq` and `qw` place quotes around the whole string, which gives a single scalar, not a range. Only one element will be present in the array.

`@a = qw(011 012 013)` gives `011 012 013`
Since `qw` (and single quotes) produces text, the leading zero is included in the array values

`@a = (011, 012, 013)` gives `9 10 11`
Oops! A leading zero indicates an octal number. Without quotes the elements are numeric, octal 11 is decimal 8 + 1 = 9.

Exercise 4 Solution

This is fairly simple, if you remember that arrays start at index 0:

```
# Create an array from a list of eight computer names:

@names = qw(yogi booboo grizzly rupert baloo teddy bungle
care);

#i.   Change 'baloo' to 'greppy'
$names[4] = 'greppy';

#ii.  Add 'fozzie' to the end of the array
$names[@names] = 'fozzie';
```

You may have used:

```
$names[scalar(@names)] = 'fozzie';
```

that is fine, although this context (an array index) is already scalar! You didn't use `$#names+1` did you?

```
#iii. Print the resulting array, and its length
print "@names\nlength: ", scalar(@names), "\n"
```

Exercise 5 Solution

```
# Using the array created in the previous exercise:
@names = qw(yogi booboo grizzly rupert baloo teddy bungle
care);
$names[4] = 'greppy';
$names[@names] = 'fozzie';

# In one statement, copy the last four elements of the array
into a list
# of scalar variables, one for each element.
($one, $two, $three, $four) = @names[-4..-1];
# Print the four scalars.
print "$one $two $three $four\n";

# In one statement, copy the first element of the array into a
scalar
# and the next three elements into a new array called @new.
($ascalar, @new) = @names[0..3];

# Print the scalar and the new array.
print "$ascalar @new\n";

# In one statement, add @new onto the end of the array, then
print the result
@names = (@names, @new);
print "@names\n"
```

The final array looks like this:

```
yogi booboo grizzly rupert greppy teddy bungle care fozzie
booboo grizzly rupert
```

Optional Exercises

Optional Exercise 1 Solution

The program below is one possible solution:

```
# (a) Create an array with over 10 elements
@words = qw(zero one two three four five six seven eight
nine);

# (b) Print the highest index of the array
print "Highest index 1: $#words\n";

# (c) Assign beyond the end of the array, to an element
at index 20
$words[20] = 'last';

# (d) Save the current highest index in a scalar, and
print it.
$old_size = $#words;
print "Highest index 2: $old_size\n";

# (e) Set the array size to 5 elements (index 4)
$#words = 4;

# (f) Print the array
print "Array contents 1: @words\n";

# (g) Set the array size back to the previous size
$#words = $old_size;

# (h) Print the array
print "Array contents 2: @words\n";
```

It generates the following output:

```
Highest index 1: 9
Highest index 2: 20
Array contents 1: zero one two three four
Array contents 2: zero one two three four
```

As you can see, reducing the array size and then extending it again will set all `tail' elements to `undef`. Printing the array in its final form will produce a warning for each undef value, 16 warnings like this one: `Use of uninitialized value in join or string at …`

Exercise 2 Solution

Here is one solution:

```
print "Today is ",
(Sun,Mon,Tues,Wednes,Thurs,Fri,Sat)[(localtime)[6]], "day\n";
```

We have a list of literals (`Sun,Mon,Tues,...`) which we index using the day number. We get the day number by picking the seventh element returned by `localtime`. Notice that the call to `localtime` is in parentheses.

A more readable (but long winded) way would be:

```
@week = qw(Sun Mon Tues Wednes Thurs Fri Sat);
@time = localtime;
$weekday = $time[6];

print "Today is $week[$weekday]day\n";
```

Chapter 4: Expressions and Operators

These questions exercise numerical and string expressions and operators, and glob constructs, with optional exercises on data type conversion and array context.

Exercise 1

Create a program that prints a text value a variable number of times. Prompt the user to enter both the text and the number of times it is to be printed. Hint: use the 'x' operator, we have not covered loops yet!

Is there a limit to the number of times we can print the text (try entering a very large number)?

Exercise 2

This is a 'pencil and paper' exercise on operators. What will be displayed? When you are done, run the program from the `labsolutions` directory `04ExpressionsOperators/Ch4Ex2.pl`. to check your answers.

N.B. **`\t`** is a tab character.

```
#  initialise a variable

      $i = 7;

#  increment operators

      print "\$i started off as $i\t";
      print "++\$i is ", ++$i, "\t";
      print "and (think about this!) \$i++ is ", $i++, "\n\n";

#  assignment operators/updaters

      $j = $i;
      print "Assignment operators ... so\t\t";
      print "\t\$i = \$i * 10 is ", $i = $i * 10, "\n";
      print "but can also be written as:\t\t";
      print "\t\$i *= 10  which is now ", $i *= 10, "\n", ;

      $i = $j;
      print "\nalso available are -, / and % ... so\t";
      print "\t\$i = \$i % 7 is ", $i = $i % 7, "\n";
      $i = $j;
      print "but can also be written as:\t\t";
      print "\t\$i %= 10  which is ", $i %= 10, "\n\n";

#  Finally

      print "We also have Perl specific .= and x, so:\t";
      print "\$i .= \$j  gives ", $i .= $j, "\n\n";

      @array = (42) x $i;
      print "\@array = (42) x \$i gives \$#array: ", $#array;

      print ", with scalar(\@array): ", scalar(@array), ".\n";
```

Exercise 3

Write a Perl script to display the number of files in a directory, the name of which is specified by a command line argument, for example:

```
filecount.pl labsolutions
```

Hints: Use **@ARGV** to get the command line argument, and the **glob** function to get an array of files.
Do not try and distinguish between regular files and directories at this stage (that's for later).

Optional Exercises

Exercise 1

Predict what the following program will do, without running it. Then run it and explain any differences.

```
$a = (1, 2, 5, 'b');
$a[1] += 2;
$a[3] .= 'a';
print "Result: $a[0]; $a[1]; $a[2]; $a[3]\n";
```

Now run the program again, but make sure you have warnings on (perl -w *script-name,* or, on UNIX, `#!/usr/bin/perl -w` as the first line of your script). This should clarify what happens.

Exercise 2

Here is another 'pencil and paper' exercise, try and predict what the following will do:

```
$max_val = 17;

# Do we have integers in Perl?
$bignumber = 0xFFFFFFFFFFFFFFFF;

print "\n\$bignumber starts off as $bignumber\n";
print "\$bignumber *= 100 is ", $bignumber *= 100, "\n";

$i = $j = $k = 0;

$i = -8 + $max_val - 2;
$j = $i++;
$k = 14 + 1 * 2;

print "\n\$i is $i\t\t\$j is $j\t\t\$k is $k\n";
```

Chapter 4: Expressions and Operators - Solutions

Exercise 1 Solution

The program below will do the trick:

```
print "Please enter a text: ";
$text = <STDIN>; chomp $text;

print "Please enter a number: ";
$num = <STDIN>; chomp $num;

print "The text [$text] multiplied [$num] times is: ",
       $text x $num, "\n";
```

If you try and create a large amount of text, then (eventually) you will get the message '`Out of memory!`'. The exact point at which this occurs varies with the implementation, operating system, and set-up.

Exercise 2 Solution

The output generated is as follows:

```
$i started off as 7  ++$i is 8    and (think about this!) $i++
is 8

Assignment operators ... so              $i = $i * 10 is 90
but can also be written as:              $i *= 10  which is
now 900

also available are -, / and % ... so     $i = $i % 7 is 2
but can also be written as:              $i %= 10  which is 9

We also have Perl specific .= and x, so:  $i .= $j  gives 99

@array = (42) x $i gives $#array: 98, with scalar(@array): 99.
```

If you did not get the right answers, don't worry! So long as you understand why we get these (ask your instructor if you are not sure).

Exercise 3 Solution

Here is our solution:

```
@files = glob ("$ARGV[0]/*");
print "Files in labsolutions: ", scalar(@files), "\n";
```

Notice that :

```
print "Files in labsolutions: ", scalar(glob
("$ARGV[0]/*")), "\n";
```

will NOT WORK! This is because, like many built-in functions, `glob` detects that it is being run in scalar context, and acts differently. It will only return the first file name in the directory, not a list.

Optional Exercises

Exercise 1 Solution

The program may have mislead you in line 1 because it assigns to a scalar variable, $a. It does not assign to an array, or assign an array value to a scalar. Instead it uses the comma operator for a sequence expression, and uses parentheses for priority. The net effect of line 1 is `$a = 'b';`. In the subsequent lines, addition and string concatenation is used on elements of the array `@a`, which is completely independent from the scalar $a. Since you've not initialised `@a`, all elements are undef when first used. If you use the perl option -w you will get a number of compile time and run time warnings:

```
Useless use of a constant in void context at ch4exopt3.pl
line 4.
Useless use of a constant in void context at ch4exopt3.pl
line 4.
Use of uninitialized value at ch4exopt3.pl line 7.
Use of uninitialized value at ch4exopt3.pl line 7.
Result: ; 2; ; a
```

The warning option is very useful and should be the first thing to try when testing or debugging a perl script.

Exercise 2 Solution

This is another slightly sneeky question. Perl implementations convert hex. numbers into an integer, the size of which varies between platforms. For 32-bit machines this gives:

```
Integer overflow in hexadecimal number at ./Ch4ExOpt2.pl line
7.
Hexadecimal number > 0xffffffff non-portable at ./Ch4ExOpt2.pl
line 7.

$bignumber starts off as 1.84467440737096e+19
$bignumber *= 100 is 1.84467440737096e+21

$i is 8         $j is 7         $k is 16
```

Chapter 5: Scalar Functions

Exercise 1

Create an array of 5 numbers. Print the numbers, each separated (join'ed?) by a semicolon and blank.

Exercise 2

Write a program to extract the directories from the PATH environment variable. Print the number of directories, and their names. The values of each environment variable are available in a hash, %ENV, with the key as the variable name, for example, `$ENV{'PATH'}`.

If you print the names of the directories in one long line, they can be difficult to read. Can you think of a way of printing each directory on a different line? TMTOWTDI

On ***UNIX***:

The PATH environment variable uses a colon (:) as a directory separator, and can be displayed on the command line using:

```
echo $PATH
```

On ***Microsoft Windows***:

The PATH environment variable uses a semicolon (;) as a directory separator, and can be displayed on the command line using:

```
path
```

Exercise 3

(a) The **`substr`** function is used to extract a sub string from the middle of a text. Use it to extract the word 'feed' from the text 'Did you feed the dog today?'.

Of course it would be unrealistic to expect the word 'feed' and the text line to be hard-coded, they would probably come from some outside source. Therefore do not hard-code the length of the word or its offset, use **`length`** and **`index`** to find them instead.

(b) You can also assign to a sub-string (i.e. use the function as an lvalue). Use this feature to replace the word `dog' by `cat' in the example text.

(c) You can grow and shrink strings by assigning to the `substr` function. Use this feature to replace 'dog' with 'Labrador' in the example text.

Exercise 4

Take a filename (see below), and split it into the individual elements of the path, storing the result into an array. Print the array as a whole, then the first and last elements separately.
What is the first element, and why?

On ***UNIX*** use:

```
/usr/local/lib/locale/US_C.C/messages.dat
```

On ***Microsoft Windows*** use:

```
        \Program Files\Microsoft Visual
Studio\VC98\Bin\Rebase.exe
```

It does not matter if these files do not exist!

If you did this exercise on UNIX, and feel up to a challenge, try the Windows file name!

Optional Exercises

Exercise 1

What be printed from the following code (and why)?

```
@list = qw (The quick brown fox jumps over the lazy Perl
programmer);
print length(@list), "\n"
```

Chapter 5: Scalar Functions - Solutions

Exercise 1 Solution

Use join:

```
@numbers = (5, 4, 3, 2, 1);
print "Numbers: ", join('; ', @numbers), "\n";
```

Exercise 2 Solution

This is a fairly straightforward application of the `split` function. The 'if' statement at the start caters for different separators on Microsoft Windows and UNIX. You may find this useful if you are writing portable code.

```
if ($^O eq "MSWin32")
{
   $Sep = ';';
}
else
{
   $Sep = ':';
}

$path = $ENV{PATH};
@dirs = split $Sep, $path;

print "Number of directories: ", scalar(@dirs), "\n";

# print the directories, one per line TMTOWTDI
# Method one:
$" = "\n";
print "Directories:\n@dirs\n";

# Method 2:
print "Directories:\n", join ("\n", @dirs), "\n";
```

Exercise 3 Solution

The sample program below shows how to use `substr`, `index`, and `length`:

```
$text = 'Did you feed the dog today?';
$replace = 'feed';

# Find the offset
$offset = index ($text, $replace);
# Get it's length
$len = length $replace;

# Extract characters at offset
$action = substr($text, $offset, $len);
print "At $offset,$len: [$action]\n";

# Make a copy of the text, then change 'dog' to 'cat'
$copy    = $text;
$replace = 'dog';
$offset  = index ($copy, $replace);
$len     = length $replace;
substr($copy, $offset, $len) = 'cat';
print "New string: [$copy]\n";
```

```
# Make a copy of the text, then change 'dog' to
'Labrador'
$copy = $text;
substr($copy, $offset, $len) = 'Labrador';
print "New string: [$copy]\n";
```

Assigning to `substr` makes it easy to change the middle of strings, for example strings that were just read from the keyboard. Note that the `index` function gives text positions which may be used in `substr`.

Exercise 4 Solution

This is a job for the `split` function, but we have to be careful of the directory separator character. Here is a portable solution :

```
if ( $^O eq "MSWin32" )
{
   $filename = '\\Program Files\\Microsoft Visual
Studio\\VC98\\Bin\\Rebase.exe';
   $delimiter = '\\\\';
}
else
{
   $filename = '/usr/local/lib/locale/US_C.C/messages.dat';
   $delimiter = '\/';
}

@parts = split /$delimiter/, $filename;

print "@parts\n";
print "First element: <$parts[0]>, Last element: $parts[-1]\n";
```

On ***UNIX***, we have to "escape" the forward slash, otherwise it would appear to be the regular expression delimiter in the `split` function.

On ***Microsoft Windows***, we could have used a forward slash as well, since (32-bit) Windows supports both. Using the back slash we double our problem. We need to escape the back slash, but this in turn has to be escaped! Hence the four back slashes.

On both platforms, the first element is not the first directory name, it is blank. This is so the original string may be constructed from the array using `join`, which would not otherwise insert the leading delimiter.

Optional Exercises

Exercise 1 Solution

The answer is **2**. Why? Because the parameter to `length` is a scalar, so the array is converted to give the number of its elements. This happens to be 10 for this array which, converted to s string, is two characters. Using `length` on an array gives us the logarithm of the number of elements!

Chapter 6: Flow control

Exercise 1

Write a program that uses a `foreach` (or `for`) loop and writes 50 lines of output: the first containing a single `X', the second containing two `XX', etc. Hint: for the list of values, use a range 1..50.

Exercise 2

Write a program that prompts the user for a word, then prints it 5 times on one line. The program should then ask the user for another word, which it prints 5 times again. Keep looping until the user enters the word `quit` (don't worry if 'quit' is also printed). Hint: after reading from the keyboard, don't forget to `chomp`.

Exercise 3

Write a Perl program that emulates the high-street bank mechanism for checking a PIN. Keep taking input from the keyboard (<STDIN>) until it is identical to a password number (hard-coded by you in the program).

Restrict the number of attempts to 3 (be sure to use a variable for that, we may wish to change it later), and output a suitable message for success and failure. Be sure to include the number of attempts in the message.

Just to keep things interesting, give the PIN a leading zero.

Optional Exercises

Exercise 1

Find out if a hash contains duplicate values, without using for, while or any other loop constructs. You may assume the hash has no `undef` values.

Exercise 2

One of the first programs many people write is a lottery number generator – to generate six random integers between 1 and 50. A Perl lottery program can use the function `rand` to get a pseudo-random number. This function generates a number between zero and its first argument, for example `rand 1234` will give a number between zero and 1234. Unfortunately, the numbers it returns are *double precision*.

Easy so far? Remember that you need six numbers, and that you could call `rand` six times but get the same number more than once (there are many possible solutions to this).

Print out your six numbers in sorted order, with a space between each, for example:

```
1 8 12 27 30 34
```

We have only shown how to do a textual sort so far, to sort numerically use:

```
sort {$a<=>$b} (some-list);
```

Chapter 6: Flow control - Solutions

Exercise 1 Solution

The program below shows how it works:

```
for $amount (1..50)
{
  print 'X' x $amount, "\n";
}
```

Exercise 2 Solution

The program below will work:

```
$word = '';              # Stops a warning on next line

while ($word ne 'quit')
{
   print "Enter a word, 'quit' to stop: ";
   chomp ($word = <STDIN>);
   print $word x 5, "\n";
}
```

Exercise 3 Solution

There are many possible solutions to this, if yours meets **all** the criteria of the question, then it is correct! Here is one version:

```
$pin = '0138';
$limit = 3;
$counter = 0;

while ( $counter < $limit )
{
    print "Please enter your PIN\t";
    $input = <STDIN>;
    chomp $input;
    $counter++;
    if ($input eq $pin)
    {
       last;
    }
}

if ( $input ne $pin )
{
    print "You had $limit tries and failed!\n";
}
else
{
    print "Well done, you remembered it!\n";
    print "... and only after $counter attempts\n";
}
```

Exercise 4 Solution

If you are using a C mindset, you might come up with a solution like the one below, which uses nested loops:

```
%programs = ('Word' => 'Microsoft',
             'Excel' => 'Microsoft',
             'WordPerfect' => 'Corel',
             'Notes' => 'Lotus',
             'WordPro' => 'Lotus',
             'Navigator' => 'Netscape');

# Find duplicates, the wrong way

for $key (keys %programs)
{
      for $try (keys %programs)
      {
            if ($key eq $try) { next };
            if ($programs{$key} eq $programs{$try})
            {
                  print "Duplicate value: ",
                        "$programs{$key}\n";
            }
      }
}
```

The perl way is to use an extra hash, indexed by the values found, which indicates whether or not a value is unique:

```
# Find duplicates, the right way

for $key (keys %programs)
{
      $val = $programs{$key};
      if (! exists $suppliers{$val})
      {
            $suppliers{$val} = undef;
      }
      else
      {
            print "Duplicate value: $val\n";
      }
}
```

Notice we have set the value to undef, since we are only interested in the existence of the key. This is marginally more efficient than setting the value to an arbitrary value like 1.

Optional Exercises

Exercise 1 Solution

This one is not so easy. It works like this: if you reverse a hash, the keys become the values and the values become the keys. The number of new keys is equal to the number of old values, unless here were duplicate values. If there were duplicate values, the number of new keys is less than the original number of values. You don't know which values were duplicates.

At first you may try this code:

```
if (scalar(values %hash) == scalar(keys(reverse %hash)))
```

however the Perl function *keys* does not accept *reverse* as an argument (it must be a hash), so we have to use a temporary hash. The `scalar()` calls are not strictly necessary since the comparison operator (==) is in scalar context anyhow. We are left with this code:

```
my %temp = reverse %hash;
if (values %hash == keys %temp)
```

A more elegant solution uses a hash reference, which we have not covered yet. Here is a taster:

```
if (values %hash == keys %{{reverse %hash}})
```

Exercise 2 Solution

This problem appears simple, but has some hidden complexities.

The use of the `int` function solves the problem of `rand` returning a double. The number range required is 1..50, but `rand` acts from zero. Therefore we specify a maximum number of 49 and add one to the result.

We could use an array of six numbers, but how would we check to see if a number has already occurred? That would probably involve multiple loops. An elegant solution is to use a hash, with the key and the value both holding the random number. If the number reoccurs it will just overwrite the existing entry. We keep going until we have six keys in the hash.

Now we must extract those keys, sort them numerically, and print them. Obviously there are many solutions to this, and, equally obviously, the randomness of the result depends on the seed (see `srand`) and the underlying algorithm. We can safely say that no one has yet won the lottery using this solution (and is not likely to)!

```
while ( scalar(keys %iHash) < 6 )
{
   $rand = int(rand 49) + 1;
   $iHash{$rand} = $rand;
}

@iArray = sort {$a<=>$b} keys %iHash;
print "@iArray\n";
```

Chapter 7: Array and Hash Functions

Exercise 1

- Create a hash with several key/value pairs ('cut and paste' one from a previous exercise if you like).
- Now create an array from the keys, and another from the values.
- Print these two arrays with a colon (':') between each value or key (TMTOWTDI).

Exercise 2

We need to do some maintenance on our list of machines:

```
%machines = (user1  => 'yogi',
             user2  => 'booboo',
             user3  => 'rupert',
             user4  => 'teddy',
             user5  => 'care',
             user6  => 'winnie',
             user7  => 'sooty',
             user8  => 'padders',
             user9  => 'polar',
             user10 => 'grizzly',
             user11 => 'baloo',
             user12 => 'bungle',
             user13 => 'fozzie',
             user14 => 'huggy',
             user15 => 'barnaby',
             user16 => 'hair',
             user17 => 'greppy');
```

Don't type this in! It should be available for you to edit in Ex6.2.pl in the `labsolutions` directory.
Without altering the initial definition of the hash, write code that will implement the following changes:

(a) user15 no longer has a machine assigned

(b) The name of user16's machine is changed to 'Ursa'

(c) user17 is leaving the company, and a new user, user18, will be assigned his machine

(d) user5, user6, and user7 are all leaving at exactly the same time, but their machine names are to be stored in an array called `@unallocated`. Hint: delete with a slice.

(e) Print a list of each (hint) user, with their machine, in any order. Do not print users that have no machine defined for them (like user15, for example).

(f) Print a list of unallocated machines, sorted alphabetically.

Optional Exercises

Exercise 1

If you have a hash, how can you create a new hash with the values of the original hash as its keys, and the keys of the original hash as its values?

Will the number of keys remain the same?

Hint: esrever .

Exercise 2

Getting the reverse of the string 0123456 is easy with an assignment:

```
$var = '0123456';
$rav = reverse $var;
```

But the following does not work:

```
print "The reverse of $var is ", reverse($var), "\n";
```

Why? Can you fix it?

Chapter 7: Array and Hash Functions - Solutions

Exercise 1 Solution

Create a hash with several key/value pairs:

```
%count2 = ( 'English'    => 'two',
            'Borrowdale' => 'tyan',
            'Eskdale'    => 'taena',
            'Old Welsh'  => 'dou',
            'Cornish'    => 'deu',
            'Breton'     => 'daou');
```

Now create an array from the keys:

```
@dialect = keys %count2;
```

and another from the values:

```
@twos = values %count2;
```

Print these two arrays with a colon (':') between each value or key:

```
print "\@dialect: ", (join ':', @dialect), "\n";
print "\@twos: ", (join ':', @twos), "\n";
```

Notice the parentheses around the call to `join`, without them the `"\n"` would be included in the list, and an extra `':'` would be printed.

TMTOWTDI. Using the special variable `$"` makes for much tidier code:

```
$" = ':';
print "\@dialect: @dialect\n";
print "\@twos: @twos\n";
```

Exercise 2 Solution

(a) user15 no longer has a machine assigned

```
$machines{user15} = undef;
```

(b) The name of user16's machine is changed to 'Ursa'

```
$machines{user16} = 'Ursa';
```

(c) user17 is leaving the company, and a new user, user18, will be assigned his machine

```
$machines{user18} = $machines{user17};
delete $machines{user17};
```

(d) user5, user6, and user7 are all leaving at exactly the same time,
but their machine names are to be stored in an array called @unallocated.

```
@unallocated = delete @machines{user5, user6, user7};
```

(e) Print a list of each (hint) user, with their machine, in any order.

```
while (($key, $val) = each %machines)
{
   if (defined $val)
   {
      print "$key: $val\n"
   }
}
```

(f) Print a list of unallocated machines, sorted alphabetically.

```
@sorted = sort @unallocated;
print "Unallocated machines: @sorted\n";
```

The following output is produced:

```
user1: yogi
user2: booboo
user10: grizzly
user3: rupert
user11: baloo
user4: teddy
user12: bungle
user13: fozzie
user14: huggy
user8: padders
user9: polar
user16: Ursa
user18: greppy

Unallocated machines: care sooty winnie
```

The exact order of the key/value pairs may vary between releases of Perl

Optional Exercises

Exercise 1 Solution

The trick to this lies in the hash-to-array and array-to-hash conversions. If you create an array from a hash, you have an array with (key, value) pairs, starting with a key, and ending on a value.

If you reverse the array, you still have an array of (value, key) pairs. Even though the order has been reversed, each key and value are still together. If you create a hash from this reversed array, you have the reverse hash. You can even do this in one line: `%hash2 = reverse %hash;`

The number of keys in %hash2 will be equal to, or less then, the number of keys in %hash1, for two reasons: the first hash might have `undef` values, which will be dropped from the second hash; and the first hash may have duplicate values, of which only one will be retained as a key.

Exercise 2 Solution

The reason the `print` statement does not work is that it takes a list as its argument, so everything is in list context. The string is seen as a single element array to `reverse`. Of course reversing a single element array has no visible effect. We therefore have to force `reverse` into scalar context:

```
print "The reverse of $var is ", scalar(reverse($var)),
      "\n";
```

Naturally TMTOWTDI:

```
print "The reverse of $var is ", reverse($var) . "\n";
```

This uses the 'dot' operator to concatenate the result of `reverse` and the new-line character, which has the side effect of forcing both into scalar context. Neat!

Chapter 8: Advanced Flow Control

Exercise 1

This exercise tests your understanding of the loop variable in a `foreach` loop, so try this without running the code. Explain the difference between the following code fragments. What will be printed in each case?

(a)

```
print "\n(a):\n";
my $counter = 0;

@array = (1..10);
for $counter (@array)
{
   print ++$counter, " "
}

print "\nThe whole array:\n@array\n";
print "\$counter: $counter\n";
```

(b)

```
print "\n(b):\n";
my $counter = 0;

@array = (1..10);
for $counter (1..10)
{
   print ++$counter, " "
}

print "\nThe whole array:\n@array\n";
print "\$counter: $counter\n";
```

Now run the code in `labsolutions/08AdvancedFlowControl/Ch8Ex1.pl` to see if you are correct (or look at the solution).

Exercise 2

Write a Perl program that checks the PATH environment variable (`$ENV{'PATH'}`). Check that each entry is actually a directory which exists, and that we have (on ***UNIX*** only) execute access. Print the result of the check for each directory.

Hint: You can save time by using your answer from Chapter 5, Exercise 2, as a template.

Exercise 3

Write a program that continually prompts the user for a number until the user enters 'quit'. Collect these numbers in an array, and print:

- All the numbers in the array
- How many numbers in the array
- The sum of all the numbers
- The average of all the numbers
- The largest and smallest number

You should not perform any computations while the program is still prompting for numbers.

Exercise 4

Write a simple command line Perl shell. The "shell" is to output the prompt "Perl > ", and Perl commands are read from the keyboard. Errors are displayed on the screen.

Hints:

Keep it simple!
We suggest you use an infinite loop.
Read the line from the keyboard using something like:

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

Use eval to execute the line.
Check the result of the `eval` (TMTOWTDI) and print any error messages to the standard error output stream.

Do you need to chomp the command line before executing it?
How does the user exit the shell?

Optional extension

The Perl command to delete a file is: `unlink` *`filename`*. Modify your script to prevent the user from deleting a file using this command, and output a suitable message when an attempt is detected.

Test your code. **First make a backup copy of your script!** Now pretend you are a malicious user who types:

```
print unlink $0
```

We hope you remembered to backup your script!

Note: there are other ways of deleting a file by using a shell command from Perl, ignore these for the purposes of this exercise.

Optional Exercises

Exercise 1

Word prefixes are also called *stems*. Write a program that reads standard input with one word per input line, finishing on end-of-file (<CTRL>D on UNIX, <CTRL>Z on Microsoft Windows). Find (and print) the most popular stems of 2 to 6 characters long.

When given the following input:

test
tester
jest
compute
computer
literate
literal
literacy
continue
collaborate

It should give the following output:

```
Most popular stem of size 2 is: co (occurs 4 times)
Most popular stem of size 3 is: lit (occurs 3 times)
Most popular stem of size 4 is: lite (occurs 3 times)
Most popular stem of size 5 is: liter (occurs 3 times)
Most popular stem of size 6 is: litera (occurs 3 times)
```

When you have this small test working, test on the supplied file called '`words`', in the `labsolutions` directory, using redirection for standard input (`script.pl < words`) – it takes a few seconds! You should get these results:

```
Most popular stem of size 3 is: con (occurs 737 times)
Most popular stem of size 4 is: inte (occurs 254 times)
Most popular stem of size 5 is: inter (occurs 197 times)
Most popular stem of size 6 is: counte (occurs 37 times)
```

Hints:

You can read STDIN into $_ until end-of-file with:

```
while (<STDIN>) {
      # loop body
}
```

Think about the design! You will probably need some nested loops, and to collect the stems into a hash.

On Windows, to get redirection to STDIN, you must run your script by prefixing 'perl', i.e. you cannot rely on association. For example:

```
perl Ch8ExOpt1.pl < ..\words
```

Chapter 8: Advanced Flow control – Solutions

Exercise 1 Solution

The first example will change `@array`, and the second example does not change anything. In both cases `$counter` is unchanged, because the loop variable is not a 'real' lexical variable, but only an alias for each list element. Here is the output produced:

```
(a):
2 3 4 5 6 7 8 9 10 11
The whole array:
2 3 4 5 6 7 8 9 10 11
$counter: 0

(b):
2 3 4 5 6 7 8 9 10 11
The whole array:
1 2 3 4 5 6 7 8 9 10
$counter: 0
```

Exercise 2 Solution

There are several solutions, here is one which is portable:

```
use strict;

# This allows for different separators on UNIX & Windows
my $Sep = $^O eq 'MSWin32' ? ';' : ':';

foreach my $dir (split /$Sep/, $ENV{PATH})
{
   if ( -d $dir ) {
      print "$dir exists ";
      if ( -x $dir ) {
         print 'and we';
      }
      else {
         print 'but we do not';
      }
      print " have execute access\n"
   }
   else {
      print "$dir does not exist!\n"
   }
}
```

Exercise 3 Solution

The program below shows how to do this, using a `while` loop to read all numbers in an array, and a `foreach` loop to find out minimum, maximum and sum.

```
use strict;

my @numbers = ();
while (1) {
   print "Enter a number, or 'quit' to end: ";
   my $num = <STDIN>;
   chomp $num;
   last if ($num eq 'quit');
   push @numbers, $num;
}

print "All numbers: @numbers\n";
print "Amount: ", scalar(@numbers), "\n";

my ($sum, $min, $max) = (0, $numbers[0], 0);
foreach my $num (@numbers) {
   $sum += $num;
   $min = $num if ($min > $num);
   $max = $num if ($max < $num);
}

print "Sum of all numbers: $sum\n";
print "Average: ", $sum / @numbers, "\n";
print "Smallest: $min; Largest: $max\n";
```

Exercise 4 Solution

Here is our simple solution:

```
while (1) {
   print "Perl > ";
   my $line = <STDIN>;

   eval $line
}
```

You may have tested the return value from `eval` instead of `$@`, which is fine. There are several ways of printing to standard error, we could also have used `print STDERR`, or even `die`.

This is one occasion when `chomp` is not necessary, since a trailing "\n" has no effect on the command.
The user simply has to type the `exit` command to exit the script.

Optional extension

Since the unlink command could be imbedded inside another, it is important to search the entire string, rather than just the first few characters. Our solution therefore uses index rather than substr.

```
while (1)
{
   print "Perl > ";
   my $line = <STDIN>;

   if (index($line, 'unlink') == -1)
   {
      eval $line;
   }
   else
   {
      $@ = "This command is not allowed!\n"
   }
}
```

This has a few undesirable side-effects, for example we could not enter the command:

```
print "unlink deletes files\n"
```

A regular expression (see later) could solve that.

Optional Exercise Solutions

Exercise 1 Solution

Of course, there's more than one way to do it, and this is one of them:

```
# Read the words, save the stems in a hash

use strict;

my %stems;

while (<STDIN>) {
   chomp;
   foreach my $count (1..length($_))    {
      $stems{substr($_, 0, $count)} += 1;
   }
}

# Process the stems

foreach my $stem_size (2..6) {
   my $best_stem;
   my $best_count = 0;

   while ( my ($stem, $count) = each %stems)
   {
      if ($stem_size == length($stem) &&
          $count > $best_count)
      {
         $best_stem  = $stem;
         $best_count = $count;
      }
   }
   if ( defined $best_stem ) {
      print "Most popular stem of size $stem_size is: ".
            "$best_stem (occurs $best_count times)\n";
   }
}
```

Chapter 9: Input and Output

Exercise 1

Develop a program to find the number of lines and characters in a file. You may wish to create a small text file using your editor in order to test your code, or use the supplied 'words' file. The filename is to be passed on the command line, not from <STDIN>, so read from ARGV with:

```
while (<>)
```

or

```
while (<ARGV>)
```

Make sure you print an error message and exit if the user forgets to supply a filename.

On ***Microsoft Windows*** the character count will be less than that reported by Windows Explorer. The reason is that Windows records are terminated by "\r\n", but Perl removes the "\r" before you see it! Therefore, add one to your character count each time you read a line.

Exercise 2

Using an editor, construct a text file which has a blank line ***after*** each line of text, for example:

```
1

22

333

4444
```

Note that there is a blank line at the end. Now, using the Input Line Separator variable `$/` and `chomp`, read this file, then print it without the blank lines, and with ">>>" at the start of each line and "<<<" at the end, for example:

```
>>>1<<<
>>>22<<<
>>>333<<<
>>>4444<<<
```

Exercise 3 *

Warnings and error messages from perl include the line number. Some editors do not show line numbers, so we need a small utility that will print specific lines from a script.

The utility is to take a filename as its first command line argument, then a list of line numbers as the second and subsequent arguments. If an invalid filename is supplied, or no line number is given, output an error message to STDERR.

The line numbers supplied on the command line could be in any order. If a line number is out of range, output an error message to STDERR, but be sure to process each line number supplied.

Output the required lines, prefixed with its line number.

Important notes (and hint):

- Our perl scripts are relatively small, and can easily fit into memory.
- The line numbers reported by perl start from 1.

Test using one of your own scripts. Since you never have any errors in your scripts ;-) you may have to introduce some, for example `$x = $y` (where `$y` has not been defined) will produce a warning.

Exercise 4

Create a program that asks the user four questions, then saves the replies in a file. The questions are:

(a) Name of the file
(b) First name
(c) Last name
(d) e-mail address

If you know HTML, you may wish to create it in that format.

Optional extension:

An email address can often be constructed using the first and last names, followed by @domain, for example: Fred.Bloggs@qa.com. Enhance your code to prompt for the domain instead, and construct the email address based on this assumption.

Optional Exercises

Exercise 1

Write a Perl program to take a list of files from the command line, and print out the number of words in each.

Hint: By default the `split` command splits $_ on whitespace.

Exercise 2

Write a simple spell checker. Read all words from the 'words' file into a hash. Then read lines from standard input and check each word is spelt correctly. Make sure it is not case sensitive (and don't forget to chomp)!

Hint: This time use a Regular Expression for the `split`:

```
split /[^a-zA-Z]+/;
```

Hints: The 'words' file is not perfect, it omits single character words like '`I`' and 'a' – don't worry about these. Also, on UNIX, ***check its record delimiter***, it may be in Microsoft Windows format (`"\r\n"`).

Exercise 3

We want to store a *single*, simple, count in a file (count.txt) which is incremented each time someone calls our script.

Reading the value then rewriting it is problematic, since the file pointer will move after the read. We do not really want to open the file twice, so we need some new functionality, the `seek` function:

```
      seek HANDLE, 0, 0 or die "Unable to rewind $filename:
$!";
```

will reset the current file pointer to the beginning of file after the read. You may then overwrite the previous count.

The exercise so far is OK until two sessions call our script at the same time. It is quite possible that both sessions will read the count before either has updated it. We therefore need a locking mechanism to make one session wait while the other does the update.

On ***UNIX*** we can get exclusive access to a file using the `flock` function. It takes two arguments: the handle of the file we wish to lock, and the type of lock operation. You will need the following:

```
      $LOCK_EX = 2;           # Value to get an exclusive lock
      $LOCK_UN = 8;           # Value to unlock

      flock HANDLE, $LOCK_EX; # get an exclusive lock of the
file

      # Now do the file IO

      flock HANDLE, $LOCK_UN; # release the lock
```

To test this, put a prompt in the script before we release the lock, run a second session and hopefully it will wait until we acknowledge the prompt.

On ***Microsoft Windows*** the `flock` function is not implemented, but we can use a mutex instead. First we need to load the Win32 Mutex module:

```
      require Win32::Mutex;
```

We need a named mutex, since we will be sharing between processes. The following will create a new mutex if it does not exist, and return a mutex object.

```
      $mutex = Win32::Mutex->new(0,'mutex_count');

      $mutex->wait;     # to get ownership of the mutex (lock)

      # Now do the file IO

      $mutex->release;  # release the mutex (unlock)
```

So what's the `->` operator? These calls actually create a reference to a mutex object and call methods on it. We will see more of this later…

Chapter 9: Input and Output - Solutions

Exercise 1 Solution

To get the number of records in a file, just increment a counter for each line read. The number of bytes in each line can easily be counted using the length function. If no argument is supplied for this function, it uses the scalar variable $_, which is also set automatically by the short form of `while`.

```
use strict;

@ARGV || die "Please supply a filename";

my $lines;
my $bytes;

while (<>) {
   $lines++;
   $bytes += length;    # Implicit: length($_)
}

print "Lines: $lines, bytes: $bytes\n";
```

Exercise 2 Solution

If the Input Record Separator (the special variable $/) is changed to "\n\n" so that perl reads an entire paragraph at a time, `chomp` changes its behaviour to follow suit.

```
# Using the command line arguments
@ARGV || die "Please supply a filename";

my $filename = shift;

open (FILE, $filename) || die "Cannot open $filename:
$!";

$/ = "\n\n";

while (<FILE>)
{
   chomp;
   print ">>>$_<<<\n";
}

close (FILE);
```

Exercise 3 Solution

There are several ways to do this. One way would be to sort the list of line numbers, then use a couple of nested loops to march sequentially through the file, printing the line numbers we want. That's a lot of work! We know that the scripts are small, so it makes sense to read into an array. Lines can then be accessed directly, remembering that line 1 will be at index zero in the array.

```
use strict;
die "Usage: $0 filename line-number\n" if @ARGV < 2;

my $file = shift @ARGV;

open (FILE, $file) || die "Unable to open $file: $!";
my @lines = <FILE>;
close FILE;

for my $lineno (@ARGV) {
   if ( $lineno < 1 || $lineno > @lines)    {
      print STDERR "Line number $lineno is invalid\n"
   }
   else {
      print "$lineno: $lines[$lineno-1]"
   }
}
```

Exercise 4 Solution

A sample solution with trivial HTML is show below:

```
use strict;

print "Enter the name of the result file: ";
my $file = <STDIN>; chomp $file;

print "Enter your first name: ";
my $first = <STDIN>; chomp $first;

print "Enter your last name: ";
my $last = <STDIN>; chomp $last;

print "Enter your email: ";
my $email = <STDIN>; chomp $email;

open (HTML, '>', $file) || die "Error opening $file: $!";
print HTML <<__END__;
<HTML><HEAD>
<TITLE>Chapter 7 Exercise 5 HTML page for $first
$last</TITLE>
</HEAD>
<BODY><H1>Dummy HTML page</H1>
This is a simple HTML page for <strong>$first
$last</strong>
<P>
He has the email <ADDRESS><A HREF="mailto::$email">$first
$last</ADDRESS>
</P>
</BODY></HTML>
__END__

close HTML;
```

Optional extension:

The trick here is with the @ symbol, because of course Perl has a special meaning for that. Here is the relevant line of code:

```
He has the email <ADDRESS>
<A HREF="mailto::$first.$last\@$domain">
$first $last</ADDRESS>
```

Optional Exercises

Exercise 1 Solution

You may have a different solution, but if it works then it's right!

```
use strict;

my %files;
while (<>) {
   my @words = split;
   $files{$ARGV} += @words;
}

while ( my ($file, $words) = each %files ) {
   print "Number of words in $file is $words\n";
}
```

Exercise 2 Solution

Yes, the words file is in Windows format. On UNIX, here is one way to do it:

```
use strict;

open (WORDS, '../words') or die "Can't open words file:
$!";

my $old = $/;
$/ = "\r\n" if $^O ne "MSWin32";

my %dictionary;
while (<WORDS>) {
   chomp;
   $dictionary{lc ($_)} = undef;
}

close (WORDS);

# Don't forget to reset $/ !
$/ = $old;

while (<STDIN>) {
   chomp;
   my @words = split /[^a-zA-Z]+/;

   foreach my $word (@words) {
      print "Unknown word <$word>\n"
          if (!exists $dictionary{lc ($word)})
   }
}
```

Exercise 3 Solution

The following code runs on UNIX and Microsoft Windows. You will notice that we have a prompt after we obtain the lock for testing purposes – a second process should halt until we hit <RETURN>.

```
use strict;

# Alter this filename to suit
my $filename = 'count.txt';

my ($mutexname, $mutex);   # Used on Windows
my ($LOCK_EX, $LOCK_UN);   # Used on UNIX

if ( $^O eq "MSWin32") {
   require Win32::Mutex;

   $mutexname = 'count_mutex';

   # This returns a handle to the mutex, and optionally creates
it
   $mutex = Win32::Mutex->new(0,$mutexname);
}
else {
   # flock() operations
   $LOCK_EX = 2;
   $LOCK_UN = 8;
}

# If the file does not exist, create it
if ( ! -f $filename ) {
   # open the file for writing
   open (HANDLE, '>', $filename) or die "Unable to create
$filename: $!";

   # get an exclusive lock of the file
   if ( $^O eq "MSWin32") {
      $mutex->wait;
   }
   else {
      flock HANDLE, $LOCK_EX;
   }

   # Write the first value
   print HANDLE 1;
}
else {
   # open the file for reading and writing
   open (HANDLE, "+<$filename") or die "Unable to open
$filename: $!";

   # get an exclusive lock of the file
   if ( $^O eq "MSWin32") {
      $mutex->wait;
   }
   else {
      flock HANDLE, $LOCK_EX;
   }

   # read the file
   my $value = <HANDLE>;
```

```
   # Test in numeric context
   die "Invalid count value <$value>" if ($value == 0);

   # Rewind the file
   seek HANDLE, 0, 0 or die "Unable to rewind $filename: $!";

   # bump the count and save it
   print HANDLE ++$value;

}

#  This is for testing purposes only, comment or delete them!
print "Hit <RETURN> to continue";
<STDIN>;

# release the lock
if ( $^O eq "MSWin32") {
   $mutex->release;
}
else {
   flock HANDLE, $LOCK_UN;
}
# close the file
close (HANDLE);
```

Chapter 10: Running Processes

***Microsoft Windows*, important note**
Depending on the way perl has been installed, the process creation methods on Microsoft Windows (back-ticks, `system()`, `fork/exec`, and pipes) might only take the name of an executable as a program name, not the name of a script. Therefore when running a Perl script you might have to prefix the script name with 'perl ', for example:

```
$result = `perl client.pl argument`;
```

Exercise 1

In the labsolutions directory you will find a simple Perl program, **client.pl**, which lists files to STDOUT. The name of the file is specified at the command line, and if it cannot be read then an error is returned, using `die`.

a) Now call the Perl program client.pl from another using back-ticks, passing a filename and capturing its output in an array. If you can't think of a file to list, use the current program, or use the 'words' file from the labsolutions directory. Print out the number of lines returned by the client.pl program.

   Output an error message if, for some reason, the client.pl fails.
   Test this by:
   altering client.pl to **`exit`** with a non-zero value
   passing a non-existent file name
   calling a non-existent program

b) Modify the calling program to use a pipe instead of back-ticks. Test as before.

Can you think of any advantages or disadvantages with these methods?

Exercise 2 *

Execute the program client.pl (from the previous exercise) without capturing the output, using `system()`. This time do not hard-code the name of the program to be executed, but use the first argument specified on the command line. Any further arguments on the command line are to be passed to the client program. As before, test the return code from the client program, and report if non-zero.

Redirect STDOUT to a file named *`program-name`*`.out`, and STDERR to *`program-name`*`.err`. If you are not sure how to do this, both UNIX and Microsoft Windows NT/2000/XP shells support the same syntax:

```
program-name >program-name.out 2>program-name.err
```

Introduce some errors in the client program to test your redirection, for example read an unopened file handle.

Exercise 3 *UNIX only*

We need to find the number of lines and bytes in a compressed file, without uncompressing it to disk (we may not have the space available).

Use the standard UNIX utility `compress` to compress a text file, for example:

```
$ compress filename
```

A file called *filename*.Z will be produced. Note that `compress` can have no effect on a small file (the name does not have the `.Z` suffix), so we suggest that the '**words**' file is used from the `labsolutions` directory.

Now write a perl script to count the number of lines and bytes, using the UNIX utility `uncompress`. HINT: you can get `uncompress` to print to stdout using:

```
uncompress -c filename
```

the file name argument need not include the .Z.

Exercise 3 ***Linux only***

We need to find the number of lines and bytes in a compressed file, without uncompressing it to disk (we may not have the space available).

Use the standard utility `zip` to compress a text file, for example:

```
$ zip filename
```

A file called *filename*.Z will be produced. Note that `compress` can have no effect on a small file (the name does not have the `.Z` suffix), so we suggest that the '**words**' file is used from the `labsolutions` directory.

Now write a perl script to count the number of lines and bytes, using the utility `unzip`. HINT: you can get `unzip` to print to stdout using:

```
unzip -c filename
```

Optional Exercise

We are going to see how output redirection really works. The problem with `system()` is that two child processes are created, the 'shell' and the program we wish to run.

Correct this inefficiency by writing a new version of the program from Exercise 2, using `fork` and `exec` (see the slides after the summary).

On Microsoft Windows
The full path name of the client program must be specified to `exec()`, and STDOUT goes to the parent's console. You may wish to investigate Win32::Process as an alternative, but there are some hidden traps. A solution using this is provided as an example.

Chapter 10: Running Processes - Solutions

Note: these solutions require slight differences for Microsoft Windows so they have an operating system check in them (**$^O**) to make them portable. Your solutions are not expected to have this check.

Exercise 1 Solution

The supplied client.pl looks like this:

```
@ARGV or die "No filename supplied\n";

for my $file (@ARGV) {
   open (FILE, '<', $file) or
      die "Cannot open $file: $!\n";

   while (<FILE>) { print }

   close FILE
}
```

a) The easiest way to run a program and capture the output is to use back-ticks:

```
use strict;

my $filename = '../words';
my @lines;

if ( $^O eq 'MSWin32') {
   @lines = `perl client.pl "$filename"`
}
else {
   @lines = `client.pl $filename`
}

if ($?) {
   print STDERR "client.pl error ", $? >> 8, "\n"
}
else {
   print "Number of lines: ", scalar @lines, "\n"
}
```

b) However a pipe is not that difficult either:

```
use strict;

my $filename = '../words';

if ( $^O eq 'MSWin32') {
   open (PIPE, "perl client.pl \"$filename\" |") ||
die "Unable to open client.pl: $!"
}
else {
   open (PIPE, "client.pl $filename |") || die
"Unable to open client.pl: $!"
}

my @lines = <PIPE>;
```

```
close PIPE;

if ($?) {
   print STDERR "client.pl error ", $? >> 8, "\n"
}
else {
   print "Number of lines: ", scalar @lines, "\n";
}
```

There is not a lot to choose between these methods. Using a pipe gives us a little more information on failure, particularly when the program cannot be found. The main advantage of a pipe is when we access it in scalar context. The data does not all have to reside in memory, instead we can process one record at a time.

Exercise 2 Solution

The `system()` command is the obvious candidate for this job, although we could use `fork/exec` instead. Passing arguments to the client program is straightforward, since the command line argument array, `@ARGV`, is exactly the parameters needed by `system()`. Notice how, on Microsoft Windows, we have to emulate the "file extension" association ourselves.

```
my $result;
if ( $^O eq "MSWin32") {
   my $script = $ARGV[0];
   unshift (@ARGV, 'perl') if (substr($ARGV[0], -3) eq '.pl');
   $result = system ("@ARGV >$script.out 2>$script.err");
}
else{
   $result = system ("@ARGV >$ARGV[0].out 2>$ARGV[0].err");
}

if ($result) {
   print STDERR "@ARGV gave error ", $? >> 8, "\n"
}
```

Exercise 3 Solution

The program below reads the compressed file and counts the number of lines and bytes. It reads from `uncompress`'s standard output using a pipe.

```
@ARGV || die "Please supply a filename";

# Use uncompress in a pipe

open (IN, "uncompress -c $ARGV[0] |") ||
      die ("Could not open $ARGV[0]: $!");

while (<IN>) {
   $lines++;
   $bytes += length; # Implicit: length($_)
}

close IN;

print "Lines: $lines, bytes: $bytes\n";
```

The Linux version will be similar, substitute 'unzip' for 'compress'.

Optional Exercise

This simple program illustrates the use of `fork` and `exec`:

```
$pid = fork;

die "Unable to fork: $!" if ( !defined $pid );

if ( !$pid ) {
   # Child

   select STDOUT;$| = 1;      # Ensure STDOUT is flushed

   open (STDOUT, ">$ARGV[0].out") ||
      die "Unable to open $ARGV[0].out: $!";
   open (STDERR, ">$ARGV[0].err") ||
      die "Unable to open $ARGV[0].err: $!";

   exec (@ARGV);
   die "Failed to exec: $!"
}
else {
   waitpid ($pid, 0);
   if ( $? ) {
      print STDERR "$ARGV[0] gave error ", $? >> 8, "\n"
   }
}
```

Yes, `system()` is simpler! Depending on the application, you may decide that the extra overhead of the shell process is worth it.

Microsoft Win32 solution

The Win32 API `CreateProcess()` includes an option to redirect standard handles (in the STARTUPINFO structure), but the Perl interface does not include it. Therefore we have to set the standard handle, create the child process, then set the handle back.

To run Perl scripts, we have to prefix the name of the Perl executable. Like UNIX, file association is not supported, but unlike UNIX neither is the #! line.

Notice that we do not use `$!` or `$?`. The Win32 API uses a different error numbering system to the C/C++ runtime library, which Perl normally uses. The error number is obtained using `GetLastError()`, and this is translated into text using `FormatMesage()`. Return codes also require an API call, this time a method on the Process ID object. Notice that there is no Close method on this (`$PHandle` in the example), so ideally it should be a `my` variable if the parent does not close at once (otherwise the Win32 equivalent of a zombie results).

```
use strict;
use Win32::Process;

# Save the STDOUT handle
open (SAVEDOUT, ">&STDOUT");
open (SAVEDERR, ">&STDERR");

# Open the output file
```

```
open (STDOUT, ">$ARGV[0].out") || die "Unable to open
$ARGV[0].out: $!";
open (STDERR, ">$ARGV[0].err") || die "Unable to open
$ARGV[0].err: $!";

$| = 1;

# Have to do our own file association
if (substr($ARGV[0],-3) eq '.pl') {
   unshift @ARGV, "C:\\Perl\\bin\\Perl.exe"
}

# Cmd line has to be a scalar
my $Cmd_line = join ' ', @ARGV;
my $PHandle;

Win32::Process::Create ($PHandle,
        "$ARGV[0]",
        $Cmd_line,                  # cmd line arguments
        1,                          # inherit handles
        NORMAL_PRIORITY_CLASS,      # Creation flags
        ".") || die Win32::FormatMessage(
Win32::GetLastError() );

$PHandle->Wait (INFINITE);

# Restore the saved STDOUT & STDERR
open (STDOUT, ">&SAVEDOUT");
close SAVEDOUT;
open (STDOUT, ">&SAVEDERR");
close SAVEDERR;

my $ExitCode;
$PHandle->GetExitCode ($ExitCode);

print "Child returned $ExitCode\n" if ( $ExitCode );
```

Chapter 11: Subroutines

Exercise 1

When a number of programmers are working on the same application, we often need a standard way of reporting an error. A subroutine is ideal for this: it a single point to maintain and modify as the need for further diagnostics arise.

Write and test a basic error handing subroutine called `report_error`. It is to take a single parameter only - a scalar containing an error message string. Print the error message, together with a date/time stamp (`scalar(localtime)`), and the current error message `$!`, to STDERR.

It would be useful if our subroutine also printed the Perl script line number which called it. This may be obtained using the `caller` function, for example:

```
($package, $filename, $line) = caller;
```

Test your subroutine by calling it after trying to open a file which does not exist, for example:

```
$filename = "some_non_existent_file";
open (OOPS, $filename) ||
      report_error ("Failed to open $filename");
```

and on trying to close an invalid file handle.

Exercise 2

(a) Create a subroutine without a prototype. Make it print the number of arguments, and the value of its first argument. Call it with:

- a scalar
- two scalars
- an array with 4 elements
- a hash with 3 key/value pairs

(b) Now add a prototype to accept a single scalar. Call it again with the same set of arguments and explain what happens.

(c) Remove any lines that generate errors, then run the program again.

Exercise 3 *

Recall Chapter 9 Input and Output, Exercise 3, in which we printed certain lines from a file. Take your solution for this exercise (or use the supplied solution) and modify it so that the functionality is supplied by a subroutine called `file_details`.

The subroutine is to take as its arguments a filename, followed by a list of line numbers.

When called in array context, it is to return a list of the required lines. If a line number is invalid, its entry in the list should be `undef`.

When called in scalar context, it is to return the total number of lines in the file.

Optional Exercises

Exercise 1

Write a program with a subroutine which takes a variable number of parameters (no prototype). The first is a scalar, and the others are 'named' parameters which all have the following defaults:

Parameter	Default value
owner	undef
name	Untitled
colour	Black
texture	Weave
width	0
background	White

Call the function, supplying the first scalar (any text), and values for the name, colour, and width parameters only. Print out the final values of all parameters in the subroutine.

Exercise 2

Create a subroutine to convert a decimal number in the range 1 to 4000 to Roman numerals, and another to convert Roman back to decimal (use that number range because Roman numbers do not include zero or negative numbers, and the symbol for 5000 is not in our character set!).

Warning: This question is rather more difficult than it looks. Don't spend too much time on it.

The Roman numerals are:

Numeral	Decimal value
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

Roman numerals generally go from large values to small values, with one notable exception: a single small numeral can be used before a larger numeral, and is then subtracted from that larger numeral. So XC means 90. However, only the following combinations are valid:

Numeral	Decimal value
IV	4
IX	9
XL	40
XC	90
CD	400
CM	900

Chapter 11: Subroutines - Solutions

Exercise 1 Solution

This is where a prototype can be useful. In this solution note how we have passed the filename, it gets interpolated on the call. Also notice the use of **my**, if we did not use it then the global variable **$filename** would be overwritten by the subroutine.

```
sub report_error ($)
{
   print STDERR scalar(localtime), " Error: @_: $!\n";
   my ($package, $filename, $line) = caller;
   print STDERR "Called from $package $filename ".
                "line: $line\n\n";
}

# Generate errors to test our subroutine
$filename = "some_non_existant_file";
open (OOPS, $filename) ||
             report_error ("Failed to open $filename");
close (OOPS) || report_error ("Whoops! close failed!");
```

Exercise 2 Solution

(a) The program below calls the subroutine with varying parameter lists.

```
use strict;

sub print_args {
   print "No. of args: ", scalar(@_),
         "; first arg: $_[0]\n";
}

my $avar  = 1;
my $bvar  = 'text';
my @array = (5, 1, 'a', 2);
my %hash  = (global => 'simple',
             my     => 'like C',
             local  => 'like LISP');

print_args($bvar);
print_args($avar, $bvar);
print_args(@array);
print_args(%hash);
```

As the output below shows, all parameter lists are passed as lists:

```
No. of args: 1; first arg: text
No. of args: 2; first arg: 1
No. of args: 4; first arg: 5
No. of args: 6; first arg: global
```

(b) If you add a prototype to the subroutine, it will look like this:

```
sub print_args ($) {
   print "No. of args: ", scalar(@_),
         "; first arg: $_[0]\n";
}
```

It will generate a fatal compile time error, shown below:

```
Too many arguments for main::print_args at ...
```

(c) If you remove the offending line, the program will convert all parameter lists to scalars before calling the subroutine, i.e. as if a `scalar()` function had been applied to each parameter list. For arrays, this returns the number of arguments; for hashes, this returns a Perl internal string value of the format num1/num2.

The alteration to the program and its output are shown below:

```
# print_args($a, $b);   commented out

No. of args: 1; first arg: text
No. of args: 1; first arg: 4
No. of args: 1; first arg: 3/8
```

Exercise 3 Solution

The structure of the original has to change a little, and we can use `wantarray` to see in which context we have been called.

```
use strict;

sub file_details {
   my @req_lines;
   my $file = shift;

   open (FILE, $file) || die "Unable to open $file: $!";
   my @lines = <FILE>;
   close FILE;

   if (wantarray)    {
      my @req_lines;

      for my $lineno (@_)       {
         if ( $lineno < 1 || $lineno > @lines)          {
            push @req_lines, undef
         }
         else          {
            push @req_lines, $lines[$lineno-1]
         }
      }
      return @req_lines
   }
   else    {
      return scalar (@lines)
   }

}  # End of file_details

die "Usage: $0 filename line-number\n" if @ARGV < 2;

my $lines = file_details (@ARGV);
print "Number of lines: $lines\n";

my @thelines = file_details (@ARGV);
print "Required lines: @thelines\n"
```

Optional Exercises

Exercise 1 Solution

The parameters to the subroutine are not actually passed as a hash: they're passed as array elements, in the exact order specified. Perl contains a language exception in that items to the left of a *fat comma* (=>) do not need to be quoted..

A way to process these parameters is shown below. This method allows the user to specify any combination of parameters out of a larger set, say 3 out of 50.

```
use strict;

sub varfunc {
   my $object = shift;
   print "Scalar: $object\n";

   my %defaults = (name       => 'Untitled',
                   colour     => 'Black',
                   width      => 0,
                   texture    => 'Weave',
                   background => 'White',
                   owner      => undef);
   my %param = (%defaults, @_);

   for my $par (keys %param) {
      if (defined $param{"$par"}) {
         print "$par param: ", $param{"$par"}, "\n";
      }
      else {
         print "No $par param specified\n";
      }
   }
}

varfunc('ZZZ',
        name   => 'Label',
        colour => 'Red',
        width  => 2);
```

And here is the output generated:

```
Scalar: ZZZ
width param: 2
No owner param specified
colour param: Red
name param: Label
background param: White
texture param: Weave
```

Exercise 2 Solution

A sample solution is shown below. It cheats, because it doesn't actually understand how Roman numerals work, but just converts numbers and strings. If you have an interest in this particular problem, have a look for the `Roman` module in CPAN.

```
use strict;

# Global variables used in both conversion routines

my @values =
       (1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1);

my %decimals =
       (1000 => 'M',  900 => 'CM', 500 => 'D',
        400 => 'CD', 100 =>  'C',  90 => 'XC',
         50 => 'L',   40 => 'XL',  10 => 'X',
          9 => 'IX',   5 => 'V',    4 => 'IV', 1 => 'I');

my %numerals =
        ('M' => 1000, 'CM' => 900,
         'D' =>  500, 'CD' => 400,
         'C' =>  100, 'XC' =>  90,
         'L' =>   50, 'XL' =>  40,
         'X' =>   10, 'IX' =>   9,
         'V' =>    5, 'IV' =>   4, 'I' => 1);

# Convert a decimal number to roman numerals
sub dec_to_roman ($) {
   my ($dec) = @_;
   my $result = '';
   my @workval = @values;

   while ($dec > 0) {
      my $current = shift @workval;
      while ($dec >= $current) {
         $result .= $decimals{$current};
         $dec -= $current;
      }
   }

   return $result;
}

# Convert a string in roman numerals to a decimal number
# Performs no error-checking !
sub roman_to_dec ($) {
   my ($roman) = @_;
   my $result = 0;

   while ($roman) {
      my $current = substr($roman, 0, 2);
      if (!defined $numerals{$current}) {
         $current = substr($roman, 0, 1);
      }

      $result += $numerals{$current};
      $roman = substr($roman, length($current));
   }

   return $result;
```

```
}

# Main program
my @testvals = qw(1 3 4 5 6 9 10 13 399 400 401 499 500 501 850
                  900 999 1000 1500 1776 1863 1996 2000 2001);

for my $num (@testvals) {
   my $rom = dec_to_roman($num);
   my $dec = roman_to_dec($rom);
   print "Value [$num] converts to roman [$rom], ".
         "converts back to [$dec]\n";
}
```

Chapter 12: Modules

Exercise 1

(a) How would you ensure the CGI.pm module is used?

(b) How would you use the Send.pm module in the Mail directory (assuming Mail is in the library directory)?

(c) How would you use the FTP.pm and Domain.pm modules, both from the Net directory (assuming Net is in the library directory)? Can you do this in one command?

Exercise 2

This exercise illustrates the use of the %INC hash in identifying the actual module files in use. Write a program that includes a number of modules (for example, Carp, CGI, sigtrap) and print the %INC hash. Examine the results, noting that there may be more modules listed than you included in your program.

Exercise 3

This exercise is to show how using a module can affect your entire program.

The task here is to write a module that will cause all output from every `print` function to be automatically piped to the external program '`more`'.

The caller should not need to explicitly call one of the module subroutines to use this feature.

Hints: Create a module called 'more.pm'

In the module's `BEGIN` block:

Open a pipe to the external program 'more' (remember pipes from the 'Running Processes' Chapter)

Override the default file handle with the pipe handle using `select` (from the 'Input and Output' chapter), saving the returned handle in a global scalar. This provides the functionality of the module.

In the module's `END` block:

Using `select`, return the default handle to its original setting.

Close the pipe handle.

Test this by writing a simple Perl program that lists several files to the screen, using your module.

Note: This will not work on Microsoft Windows 95/98/Me because '`more`' is a `command.com` built-in. There are similar problems on Windows NT, but Windows 2000 and XP should be OK.

Exercise 4

(a) Create a module called `PrintSupport' that contain dummy functions that do nothing but print their name. These functions should be: print_HP, print_Xerox, print_Apple, print_fax. Then create a program that uses this module and call all functions from the main program.

(b) Now change the start of the module to read:

```
package PrintSupport;

use Exporter;
BEGIN
{
      @ISA = qw(Exporter);
      @EXPORT = qw(print_HP print_fax);
}
```

Now try and remove the use of :: in the main program wherever possible.

(c) Change the module to use on-demand exports (`@EXPORT_OK`), and change the main program to import only the routines required.

Optional Exercises

Exercise 1

Let's write a Perl module which incorporates some C code. This is useful if we want to call API functions for which there are no Perl interfaces. If you do not know any C then don't worry, we will supply some basic code.

We will run the program `h2xs` (don't do it yet!) to create some skeleton code in its own subdirectory. We will edit that, then generate and compile the module itself in our own module directory. Finally we will call our program from a Perl script.

The steps below use the module name 'ctest', but you can choose a different name if you like.

1. Create the skeleton files from the command line with:

   ```
   h2xs -cn ctest
   ```

2. That should have created a sub-directory called `ctest`, with several files. `cd ctest` and have a look at them. Stay in the `ctest` directory.

3. Create a Makefile to generate the module in our own subdirectory called `mylib`, using:

   ```
   perl Makefile.PL LIB=~/mylib
   ```

4. Our C function will be called 'hello', so edit `ctest.pm`, and replace the `@EXPORT, @EXPORT_OK, and %EXPORT_TAGS` definitions with:

   ```
   our @EXPORT = qw (hello);
   ```

   You may wish to edit other parts of the module as well.

   Edit `ctest.xs`, which holds our C code, to look like this (don't miss the first include statement):

   ```
   #include <stdio.h>
   #include "EXTERN.h"
   #include "perl.h"
   #include "XSUB.h"

   MODULE = ctest          PACKAGE = ctest

   int
   hello ()
         CODE:
            int iRetn;
            iRetn = printf ("Hello Perl from K&R world!\n");
            RETVAL = iRetn;
         OUTPUT:
            RETVAL
   ```

   If you are a C programmer you may wish to embellish the CODE section, but remember to assign an int to RETVAL.

5. Now built the module with:

```
make install
```

This will create a number of directories, including `~/mylib`, and call the C compiler. Have a look at the files generated.

On ***Microsoft Windows***, use `nmake` instead of `make`.

6. Finally, write a Perl script to call the hello function from the ctest module. Remember that it resides in `~/mylib/i386-linux-thread-multi`.

Chapter 12: Modules - Solutions

Exercise 1 Solution

(a) `use CGI;`

(b) `use Mail::Send;`

(c) Using two separate use commands: `use Net::FTP;` and `use Net::Domain;`. You cannot do this in one command. (`use Net::FTP Net::Domain` will include the `Net::FTP` module with the parameter `Net::Domain`).

Exercise 2 Solution

The example program below includes a number of modules, which in turn include modules of their own; then the %INC hash is printed.

```
use Carp;
use CGI;
use sigtrap;


for $key (sort keys %INC)
{
   print "$key: $INC{$key}\n"
}
```

The output of this program shows that the %INC hash maps modules names, including the `.pm` suffix, to full path names (note the directory separator in the Microsoft Windows example). If you are using a different version of Perl, or a different operating system, then the modules loaded may be different.

This was generated on Microsoft Windows XP with Perl 5.8.0:

```
CGI.pm: C:/Perl/lib/CGI.pm
CGI/Util.pm: C:/Perl/lib/CGI/Util.pm
Carp.pm: C:/Perl/lib/Carp.pm
Exporter.pm: C:/Perl/lib/Exporter.pm
constant.pm: C:/Perl/lib/constant.pm
overload.pm: C:/Perl/lib/overload.pm
sigtrap.pm: C:/Perl/lib/sigtrap.pm
strict.pm: C:/Perl/lib/strict.pm
vars.pm: C:/Perl/lib/vars.pm
warnings.pm: C:/Perl/lib/warnings.pm
warnings/register.pm: C:/Perl/lib/warnings/register.pm
```

This was generated on GNU/Linux with Perl 5.8.0:

```
CGI.pm: /usr/lib/perl5/5.8.0/CGI.pm
CGI/Util.pm: /usr/lib/perl5/5.8.0/CGI/Util.pm
Carp.pm: /usr/lib/perl5/5.8.0/Carp.pm
Exporter.pm: /usr/lib/perl5/5.8.0/Exporter.pm
constant.pm: /usr/lib/perl5/5.8.0/constant.pm
overload.pm: /usr/lib/perl5/5.8.0/overload.pm
sigtrap.pm: /usr/lib/perl5/5.8.0/sigtrap.pm
strict.pm: /usr/lib/perl5/5.8.0/strict.pm
unicore/Exact.pl: /usr/lib/perl5/5.8.0/unicore/Exact.pl
unicore/lib/Word.pl:
/usr/lib/perl5/5.8.0/unicore/lib/Word.pl
utf8.pm: /usr/lib/perl5/5.8.0/utf8.pm
utf8_heavy.pl: /usr/lib/perl5/5.8.0/utf8_heavy.pl
vars.pm: /usr/lib/perl5/5.8.0/vars.pm
warnings.pm: /usr/lib/perl5/5.8.0/warnings.pm
warnings/register.pm:
/usr/lib/perl5/5.8.0/warnings/register.pm
```

Exercise 3 Solution

Here is a simple implementation of the package:

```
package more;

my $g_OldHandle;

BEGIN
{
   open (MORE, "|more") or die "No more: $!\n";
   $g_OldHandle = select MORE
}
END
{
   # Just in case other END blocks use a bare print
   select $g_OldHandle;
   close (MORE)
}

1;
```

and this is a simple test program:

```
use more;
$current_file = "";
$files = 0;

while (<>)
{
  # print a separator between each file
  if ( $ARGV ne $current_file )
  {
    print "\n","-" x 20," ",$ARGV,
           " ","-" x 20,"\n";
    $current_file = $ARGV;
    $files++
  }
  print
}
```

Exercise 4 Solution

a) A sample PrintSupport.pm module is included below:

```
package PrintSupport;

sub print_HP
{
   print "PrintSupport: HP\n"
}

sub print_Xerox
{
   print "PrintSupport: Xerox\n"
}

sub print_Apple
{
   print "PrintSupport: Apple\n"
}

sub print_fax
{
   print "PrintSupport: fax\n"
}

1;
```

The main program uses explicit module calls only:

```
use PrintSupport;

PrintSupport::print_HP();
PrintSupport::print_Xerox();
PrintSupport::print_Apple();
PrintSupport::print_fax();
```

(b) When we use the Exporter, all subroutines exported are known globally, so you don't have to specify the module name when calling them:

The main program uses explicit module calls only when required:

```
use PrintSupport;

print_HP();
PrintSupport::print_Xerox();
PrintSupport::print_Apple();
print_fax();
```

(c) When you use selective exports, all subroutines are exported, and the `@EXPORT_OK` array is used instead of the `@EXPORT` array:

```
package PrintSupport;

use Exporter;
BEGIN
{
      @ISA = qw(Exporter);
      @EXPORT_OK = qw(print_HP print_Xerox
                      print_Apple print_fax);
}
```

The main program only knows subroutines if it explicitly specifies their name on the `use` statement:

```
use PrintSupport 'print_HP', 'print_fax';

print_HP();
PrintSupport::print_Xerox();
PrintSupport::print_Apple();
print_fax();
```

Optional Exercises

Exercise 1 Solution

This is a hard copy of the build from GNU/Linux using Perl 5.8.0 username user1:

```
$ h2xs -cn ctest

Defaulting to backwards compatibility with perl 5.8.0
If you intend this module to be compatible with earlier
perl versions, please
specify a minimum perl version with the -b option.

Writing ctest/ppport.h
Writing ctest/ctest.pm
Writing ctest/ctest.xs
Writing ctest/Makefile.PL
Writing ctest/README
Writing ctest/t/1.t
Writing ctest/Changes
Writing ctest/MANIFEST

$ cd ctest
$ perl Makefile.PL LIB=~/mylib
Checking if your kit is complete...
Looks good
Writing Makefile for ctest

$ ls
Changes   ctest.xs  Makefile.PL  ppport.h  t
ctest.pm  Makefile  MANIFEST   README

$ make install
cp ctest.pm blib/lib/ctest.pm
AutoSplitting blib/lib/ctest.pm (blib/lib/auto/ctest)
Manifying blib/man3/ctest.3pm
Warning: You do not have permissions to install into
/usr/share/man/man3 at
/usr/lib/perl5/5.8.0/ExtUtils/Install.pm line 84.
Files found in blib/arch: installing files in blib/lib
into architecture dependent library tree
Installing /home/user1/mylib/i386-linux-thread-
multi/ctest.pm
Installing /usr/share/man/man3/ctest.3pm
Writing /home/user1/mylib/i386-linux-thread-
multi/auto/ctest/.packlist
Appending installation info to /home/user1/mylib/i386-
linux-thread-multi/perllocal.pod
```

and so on ... The script to run `hello()` is trivial:

```
use lib q(/home/user1/mylib/i386-linux-thread-multi);
use ctest;

hello ();
```

Chapter 13: Advanced Array Functions

Exercise 1

Sort the list of names below in case insensitive mode, and print the result:

```
John mark carol Neil bruce Dave anna Julia robert Ted
```

Exercise 2

Given the hash:

```
%oshash = ('MVS'     => '5.2.1',
           'Windows' => 'NT',
           'OS/2'    => 'Merlin',
           'Linux'   => '2.2',
           'HP-UX'   => '10',
           'Solaris' => '2.5',
           'Mac'     => 'Copland');
```

Print the following:

(a) sorted keys
(b) sorted keys, with their values
(c) sorted values
(d) keys, in order of sorted values
(e) values, in order of sorted keys

Exercise 3

Write a Perl program which will list all the Perl scripts (*.pl) in the current directory sorted by size.

There are several ways to tackle this, here is one approach:

Construct a hash of file names with their sizes.

Hint: Don't use the file size as a key
(although that would make the problem simpler)!
Hint: The simplest way to get a list of files is to use a GLOB construct.
Hint: The simplest way to get the size of a file is to use `-s filename`.

Sort the hash, using a custom sort, and print it.

Exercise 4

a. Using the Perl `grep` function, print a list of sub-directories (not files). **Hint:** Remember Globbing?

   Normally the script is to list sub-directories in the **`/etc`** directory, but if run on Microsoft Windows it is to list sub-directories in the System Root directory (C:\WINNT on Windows NT 4.0 and Windows 2000, C:\WINDOWS on Windows XP). **Hints**: The built-in variable **`$^O`** is set to "**`MSWin32`**" on Microsoft Windows, and the System Root directory name can be obtained from the environment variable `%SystemRoot%` (**`$ENV{SystemRoot}`**).

   There are several solutions to this exercise, not all involving the use of **`grep`** (**`map`** is an alternative). Can you achieve this in a single line of code?

b. Now take your directory name from a command line argument. If the user does not supply an argument, or it is not a valid directory name, prompt for it. Note: for the purposes of this exercise, do not worry about sub-directory names with imbedded spaces.

c. The Glob output retains the specified directory name as a prefix. Further refine your script by removing it for printing. For example, `/etc/sysconfig` becomes `sysconfig`. Test this with the directory `/etc/X11` (`C:\WINNT\System32` on Microsoft Windows NT and 2000, or `C:\Windows\System32` on Microsoft Windows XP).

Optional Extension

d. You will have noticed that the output scrolls off the screen, and only one column of output is given. Alter your script to output the sub-directory names in a variable number of columns, depending on the length of the largest name (like the UNIX `ls` command). Assume a screen width of 80 characters. This is *not* a quick hack!

Optional Exercises

The sort function does not remove duplicate array elements. In the next three exercises, we'll work our way around this, using the "words" dictionary file.

Exercise 1

Read the "words" file and store the length of each word in an array element. Then sort the array from large to small, removing duplicates.

The sort operation may well be very slow on your system. If so, reduce the size of the array to 1000 elements before sorting.

Exercise 2

Repeat the previous exercise without the array.

Exercise 3

Alter the above results to print a histogram of the occurrence of each size (in percent), giving output similar to the one below:

```
Distribution of 45402 words in dictionary:
2 49 (0%)
3 536 (1%) X
4 2236 (4%) XXXX
5 4176 (9%) XXXXXXXXX
6 6176 (13%) XXXXXXXXXXXXX
7 7371 (16%) XXXXXXXXXXXXXXXX
8 7074 (15%) XXXXXXXXXXXXXXX
9 6089 (13%) XXXXXXXXXXXXX
10 4593 (10%) XXXXXXXXXX
11 3069 (6%) XXXXXX
12 1880 (4%) XXXX
13 1137 (2%) XX
14 545 (1%) X
15 278 (0%)
16 103 (0%)
17 57 (0%)
18 23 (0%)
```

```
19 3 (0%)
20 3 (0%)
21 2 (0%)
22 1 (0%)
28 1 (0%)
```

Hint: Calculate the frequency as a percent, then use the `x` operator.

Exercise 4

We often need to sort dates. In many cases, such a problem can be solved by converting the dates to the internal time format (seconds since *Epoch*) using the `localtime` and `timelocal` functions, then sorting the converted dates. However, this doesn't work if the dates cannot be represented in that format, for example, dates before 1970 (Epoch on many operating systems) or past 2038 for 32-bit machines.

(a) Sort an array of dates in the format yyyy-mm-dd, e.g. 1998-08-31.

(b) Sort an array of dates in the U.S. format mm-dd-yyyy, where months and days can be either 1 or 2 digits, e.g. 8-31-1998.

(c) Sort an array of dates in the European format dd-mm-yy, where years before 70 are taken to be after the year 2000, e.g. 31-12-99 is followed by 1-1-00.

Chapter 13: Advanced Array Functions - Solutions

Exercise 1 Solution

To compare the lower-case versions of the names we can use an in-line subroutine.

```
@names = qw(John mark carol Neil bruce Dave anna
            Julia robert Ted);
@snames = sort {lc $a cmp lc $b} @names;

# Simple output
print "Simple: @snames\n";

# Neat output
print "Neat: ";
print map {uc} ("@snames"), "\n";
```

You might have used a **`foreach`** loop instead of **`map`**:

```
for (@snames)
{
   print "\u$_ ";
}

print "\n";
```

Exercise 2 Solution

Most of these questions can be done using a simple sort of keys and values. However, question (d) requires a custom sort function. This solution uses in-line code, but you could have used a separate subroutine, and **`map`** is used instead of **`foreach`** loops.

```
%oshash = ('MVS'     => '5.2.1',
           'Windows' => 'XP',
           'OS/2'    => 'Merlin',
           'Linux'   => '2.0',
           'HP-UX'   => '10',
           'Solaris' => '2.5',
           'Mac'     => 'Copland');

# (a) Sorted keys
print "(a):\n";
print map {"$_\n"} (sort keys %oshash);

# (b) Sorted keys plus values
print "\n(b):\n";
print map {"$_\t$oshash{$_}\n"} (sort keys %oshash);

# (c) Sorted values
print "\n(c):\n";
print map {"$_\n"} (sort values %oshash);
```

```
# (d) Keys, in order of sorted values
print "\n(e):\n";
print map {"$_\n"}
      (sort { $oshash{$a} cmp $oshash{$b} } keys %oshash);

# (e) values, in order of sorted keys
print "\n(f):\n";
print map {"$oshash{$_}\n"} (sort keys %oshash);
```

Exercise 3 Solution

There are several possibilities for this:

```
sub bysize
{
   $files{$a} <=> $files{$b}
}

# Construct a hash of filenames and sizes
@names = glob('*.pl');
@files{@names} = map {-s} @names;

print map {"$_: $files{$_} bytes\n"} (sort bysize
@names);
```

The **`map`** statements could have been replaced by **`foreach`** loops.

```
foreach $name (@names)
{
   $files{$_} = -s $name;
}

foreach $name (sort bysize @names)
{
   print "$name: $files{$name} bytes\n";
}
```

An alternative (bad) solution, without using a hash might have a sort routine like this:

```
sub bysize
{
   (-s $a) <=> (-s $b)
}
```

the problem with this is that we have to get the size of the file many times, which can be very inefficient.

Exercise 4 Solution

a. Of course there's more than one way to do it. A solution using **`grep`** would be this:

```
$pattern = $^O eq 'MSWin32'? "$ENV{SystemRoot}\\*" :
'/etc/*';

$, = "\n";
print (grep {-d} glob ($pattern));
```

If you prefer a conventional if/else statement, you could replace the ternary with:

```
if ( $^O eq 'MSWin32' )
{
   $pattern = "$ENV{SystemRoot}\\*";
}
else
{
   $pattern = '/etc/*'
}
```

The **`$pattern`** variable is not strictly needed, we could have placed the ternary inside the argument list for glob, but that would not be very readable!

b. We now have to generate our pattern from a user supplied directory name:

```
$pattern = shift;

while (! defined $pattern || ! -d $pattern) {
   print "Please give me a directory name: ";
   $pattern = <STDIN>;
   chomp $pattern;
}

# This is portable between Win32 and UNIX
$pattern .= '/*';

grep {print "$_\n" if (-d $_)} glob($pattern);
```

c. Again, there are a number of solutions, here we have used the length of the original directory name, plus one (for the directory separator) as input to the `substr` function.

```
$pattern = shift;

while (! defined $pattern || ! -d $pattern)
{
   print "Please give me a directory name: ";
   $pattern = <STDIN>;
   chomp $pattern;
}

$pos = (length $pattern) + 1;
$pattern .= '/*';

grep {print substr($_,$pos),"\n" if (-d $_)} glob($pattern);
```

You may see some similarity with the UNIX command `basename`, and there is a standard module which supports functions such as these: look at `perldoc File::Basename`.

Optional Extension

d. The tricky part comes in deciding how we will actually extract the names from the array in the right order, and space things correctly. There are a many ways of doing this, this is only one of them.

```
use strict;

my @subdirs;
my ($maxlen, $pos, $pattern, $lines);
my ($columns, $i, $j);

# This subroutine is called by the grep
sub storedata ($$)
{
   my $sub = shift;
   my $pos = shift;

   if (-d $sub)
   {
      my $name = substr ($sub, $pos);
      my $len  = length ($name);
      $maxlen = $len if ( $len > $maxlen);
      push (@subdirs, $name);
   }
}

$pattern = shift;

while (! defined $pattern || ! -d $pattern)
{
   print "Please give me a directory name: ";
   $pattern = <STDIN>;
   chomp $pattern;
}

$pos = (length $pattern) + 1;
$pattern .= '/*';
$maxlen = 0;

# Get the sub-directory names
grep { storedata ($_,$pos) } glob($pattern);

# Allow for a space after the longest dir name
$maxlen++;

# Find the number of columns (screen width of
# 80 chars - 1 to prevent wrap of \n)
$columns = int(79/$maxlen) || 1;

# Find the number of lines
$lines = int ( ($#subdirs + $columns) / $columns);
```

```
# Construct our lines and print them
# $i = current line number
# $j = current column

for ( $i = 0; $i < $lines; $i++)
{
   my $line;
   for ( $j = 0; $j < $columns; $j++ )
   {
      my $idx = $i + ($lines*$j);

      if ( defined $subdirs[$idx] )
      {
         # Pad name out to $maxlen characters
         my $spaces = $maxlen - length($subdirs[$idx]);
         $line .= $subdirs[$idx] . " " x $spaces;
      }
   }
   print "$line\n";
}
```

Optional Exercises - Solutions

Exercise 1

The most efficient way of removing duplicates is to use a hash:

```
my @word_lengths;

# Read data
open (WORDS, "words") || die "Cannot open file: $!";

while (<WORDS>)
{
   chomp;
   push @word_lengths, length;
}

close WORDS;

print "Array has ", scalar(@word_lengths), " elements\n";

# Store into hash
my %length_hash;
@length_hash{@word_lengths} = (undef) x @word_lengths;

# Store hash into array
my @sorted_array = sort { $b <=> $a } keys %length_hash;

print "Sorted array: @sorted_array\n";
```

Exercise 2

Without the need for an array we can read straight into a hash:

```
my @word_lengths;
my %length_hash;

# Read data & Store into hash

open (WORDS, "../words") || die "Cannot open file: $!";

while (<WORDS>)
{
   chomp;
   $length_hash{length($_)} = undef;
}

close WORDS;

# Store hash into array
my @sorted_array = sort { $b <=> $a } keys %length_hash;
print "Sorted array: @sorted_array\n";
```

Exercise 3

This will create the diagram:

```
my %length_hash;
my $num_words = 0;

# Read data
open (WORDS, "words") || die "Cannot open file: $!";

while (<WORDS>)
{
   chomp;
   $length_hash{length($_)} += 1;
   $num_words += 1;
}

close WORDS;

# Now print diagram
print "Distribution of $num_words words in
dictionary:\n";

foreach my $size (sort { $a <=> $b } keys %length_hash)
{
   my $amount = $length_hash{$size};
   my $freq = int(($amount / $num_words) * 100);
   print "$size\t$amount ($freq%)\t", 'X' x $freq, "\n";
}
```

Exercise 4

(a) It turns out that this date notation can be sorted alphabetically to give proper results.

```
my @dates = qw(1969-08-07 1973-12-23 1918-11-11
               1999-12-31 1994-12-03 2010-02-05);

print "Sorted dates: ", join(', ', sort @dates), "\n";
```

(b) In this case, we need to take a bit more care. In order to create a neat solution, we create a sort compare subroutine. In the subroutine, we compare first using years, then using months, then using days:

```
@dates = qw(8-7-1969 7-8-1969 12-23-1973 11-11-1918
            12-31-1999 12-3-1994 2-5-2010);

# Date compare subroutine
sub date_cmp_b
{
   my ($ma, $da, $ya) = split /-/, $a;
   my ($mb, $db, $yb) = split /-/, $b;
   return ($ya <=> $yb) unless ($ya == $yb);
   return ($ma <=> $mb) unless ($ma == $mb);
   return ($da <=> $db);
}

print "Sorted dates: ", join(', ', sort date_cmp_b
@dates),"\n";
```

(c) The answer is a minor variation of the previous solution:

```
@dates = qw(7-8-69 8-7-69 23-12-73 11-11-18
            31-12-99 3-12-94);

# Date compare subroutine
sub date_cmp_c
{
   my ($da, $ma, $ya) = split /-/, $a;
   my ($db, $mb, $yb) = split /-/, $b;

   # Year 2000 corrections
   $ya = ($ya < 70 ? $ya + 2000 : $ya + 1900);
   $yb = ($yb < 70 ? $yb + 2000 : $yb + 1900);

   return ($ya <=> $yb) unless ($ya == $yb);
   return ($ma <=> $mb) unless ($ma == $mb);
   return ($da <=> $db);
}

print "Sorted dates: ", join(', ', sort date_cmp_c
@dates),"\n";
```

Chapter 14: Regular expressions

Exercise 1

Write a Perl script that will perform *basic* validation and formatting of UK postcodes. The postcodes are to be read from standard input, or from a file specified at the command line (use **`while (<>){…}`**). The input lines are only to contain a postcode, with no other text, and blank lines should be ignored.

The format of a UK postcode is as follows:

	One or two uppercase alphabetic characters
followed by:	between one and two digits
followed by:	an optional single uppercase alphabetic character
followed by:	a single space
followed by:	a single digit
followed by:	two uppercase alphabetic characters

Alphabetic characters are those in the range A-Z.

The following formatting is to be done:

Remove all tabs and spaces
Convert to uppercase
Insert a space before the final digit and 2 letters

Print out all the reformatted postcodes, indicating which are in error, and a count of valid and invalid codes at the end.

Test your program using the file postcodes.txt in the labsolutions directory.

Hints:

Keep it simple; don't try to do all the formatting on one line of code!
Do the formatting first, and then test the resulting pattern
The test file has 25 valid and 5 invalid postcodes.

Exercise 2

In the `labsolutions` directory for each chapter we have Perl scripts giving suggested solutions. In each one we have a comment giving the Chapter number, followed by the Exercise.

For a single directory, list each script file with the Exercise it is intended for, based on the comment. For example:

```
Ch14Ex1.pl Exercise 1
Ch14Ex2.pl Exercise 2
Ch14Ex3.pl Exercise 3
```

Exercise 3

(a) Write a program to verify that an input line contains only alpha-numeric characters.
(b) Modify it to also show the first non alpha-numeric character found.
(c) Further refine your program to show **all** invalid characters, without using an extra loop.

Optional Exercises

Exercise 1

Part 1

Do a search and replace of 'dog' to 'cat', 'cat' to `mouse', and 'mouse' to 'dog' in:

Do dogs eat cats, or does your cat eat a mouse?

Part 2

Part 1 results in the word 'mouses', which is not very good English! Modify your solution to also replace 'cats' with 'mice' .

Hint: We need to distinguish 'cat' from 'cats' .

Exercise 2

Show how nested parenthesis groups work with the $1 to $999 variables. For instance, if you match the text `Hello Dudes Is There Any Beer?' using the pattern `/((\w+)\s+(\w+))\s+(\w+)/`, what side effect variables are set?

Exercise 3

You may recall that in Chapter 9 (Input and Output) Exercise 3 we printed specific lines from a Perl program. Then in Chapter 10 (Running Processes) Exercise 2 we ran another script, redirecting its STDERR to a separate file. Now we are going to tie these together, so that when we get an error from running a script, we also display the line causing the error.

Using code from previous exercises, write a program that will execute a Perl script, redirecting STDERR output to a file. When the script has finished, your program will inspect the error output file for Perl errors, and, using a regular expression, extract the line number from the file. Display the line from script, and the error message.

Hint: Perl error messages end in "… at *scriptname* line *n*.", where *scriptname* is the name of the script, and *n* is the line number.

You will also have to generate a Perl script for testing, that has a few errors in it, or use an earlier example.

Extension

You may have noticed that the file name is also included in the error line. Errors might not come from the main script, but from a module, in which case that should be searched instead. Modify your program to cope with this extra complication.

Chapter 14: Regular expressions - Solutions

Exercise 1 Solution

Here is one simple solution:

```
while (<>)
{
   # Formatting

   s/\s//g;              # Remove all tabs and spaces
   next if (!$_);        # Ignore blank lines
   $_ = uc;              # Convert to uppercase

   # Insert a space before the final digit and 2 letters
   s/\d[A-Z]{2}$/ $&/;

   # Validation
   # One or two uppercase alphabetic characters
   # followed by: between one and two digits
   # followed by: an optional single alphabetic character
   # followed by: a single space
   # followed by: a single digit
   # followed by: two uppercase alphabetic characters

   print "Postcode $_ is ";

   if (/^[A-Z]{1,2}\d{1,2}[A-Z]? \d[A-Z]{2}$/)
   {
      $valid++;
      print "valid\n"
   }
   else
   {
      $invalid++;
      print "invalid\n"
   }

}

print "\n$valid valid codes, $invalid invalid codes\n"
```

Exercise 2 Solution

Reading files and extracting text is quite a common requirement. One of the problems we have to overcome is that we must stop processing the file as soon as we find the Exercise number we want, otherwise we will display our regular expression! There are many possible solutions to this question, we supply two here:

This first solution uses conventional file IO methods. The angled brackets `<*.pl>` are used here as the *globbing* operator (remember globbing at the end of Chapter 4?), *not* the IO operator. It may have been more readable to use the `glob` function instead.

```
FILE:
foreach my $file (<*.pl>)
{
   open THEFILE, $file or
         die "Unable to open $file: $!";

   while (<THEFILE>)
   {
      if ( /#.*Exercise /i )
      {
         close THEFILE;

         $list .= "$file Exercise $'";
         next FILE;
      }
   }

   close THEFILE;

}

print "$list\n";
```

The following solution is somewhat simpler, and uses ARGV. The trick is to close the handle when we find what we are looking for:

```
@ARGV = <*.pl>;

while (<>)
{
   if ( /#.*Exercise /i )
   {
      $list .= "$ARGV Exercise.$'";
      close ARGV;
  }
}

print "$list\n";
```

Exercise 3 Solution

(a) The ^ and $ anchors, with the \w shortcut, may be used to verify if the line contains alpha-numeric characters.

```
while (<>)
{
   if (/^\w+$/)
   {
      print "Line contains only letters\n";
   }
   else
   {
      print "Line contains non-letters\n";
   }
}
```

Note that the $ anchor will automatically swallow the newline at the end of the line, without doing a chomp.

(b) Another way of checking for letters is to check whether there is a non letter in the line. Checking for violations is often easier than checking for compliance, as is shown below:

```
while (<>)
{
   chomp;
   if (/(\W)/)
   {
      print "Line has non-letters; first found is $1\n";
   }
   else
   {
      print "Line contains nothing but letters\n";
   }
}
```

(c) This is a tricky one. Remember that split takes a regular expression? Using split and join, you can avoid using a loop:

```
while (<>)
{
   chomp;
   my @non_letters = split (/\w+/, $_);

   if (scalar(@non_letters))
   {
      print "Line has non-letters: <@non_letters>\n";
   }
   else
   {
      print "Line contains nothing but letters\n";
   }
}
```

Optional Exercises - Solutions

Exercise 1 Solution

Part 1

In this case, you cannot do three search-and-replace actions in sequence. However, a single search-and-replace that replaces from a hash will do:

```
$work = 'Do dogs eat cats, or does your cat eat a
mouse?';
$text = $work;
%reps = (dog   => 'cat',
         mouse => 'dog',
         cat   => 'mouse');
$pattern = join ('|', keys %reps);
$work =~ s/($pattern)/$reps{$1}/g;

print "Orig: $text\nResult: $work\n";
```

The output is shown below:

```
Orig: Do dogs eat cats, or does your cat eat a mouse?
Result: Do cats eat mouses, or does your mouse eat a dog?
```

Part 2

We need to add the new translation pair, `cats => mice`, to the hash, but the problem is that we might find 'cat' in 'cats'. We need to sort the words in the pattern by length, in descending order, which ensures 'cats' is searched for before 'cat'.

```
%reps = (dog => 'cat',
         mouse => 'dog',
         cat => 'mouse',
         cats => 'mice');
$pattern =
  join ('|', sort{length($b) <=> length($a)} keys %reps);
$work =~ s/($pattern)/$reps{$1}/g;

print "Orig: $text\nResult: $work\n";
```

The output is now:

```
Orig: Do dogs eat cats, or does your cat eat a mouse?
Result: Do cats eat mice, or does your mouse eat a dog?
```

It might be easier to indicate the word boundaries, using `\b`, but that would not work with 'dog' and 'dogs'.

Exercise 2 Solution

Nested parentheses count; each open parenthesis creates a new variable. The program below shows the answer to the question:

```
$text= 'Hello Dudes Is There Any Beer?';

$text =~ /((\w+)\s+(\w+))\s+(\w+)/;

print "1: $1\n2: $2\n3: $3\n4: $4\n5: $5\n";
```

It generates the following output:

```
Use of uninitialized value at ch12optex1.pl line 8.
1: Hello Dudes
2: Hello
3: Dudes
4: Is
5:
```

Note that four variables are set, the first being the outer brackets. The warning message is set because `$5` is not set.

Exercise 3 Solution

This uses code from previous exercises:

```
@ARGV || die "You must supply a program name";

# Chapter 10 Exercise 2
$result = system ("@ARGV >$ARGV[0].out 2>$ARGV[0].err");

if ($result)
{
   print STDERR "$ARGV[0] gave error ", $? >> 8, "\n"
}

exit 0 if ( -z "$ARGV[0].err" );

# Chapter 9 Exercise 3
# Open the Perl script
open (FILE, "$ARGV[0]") ||
      die "Unable to open $ARGV[0]: $!";
@lines = <FILE>;
close FILE;

# Read the error file, and extract the line numbers

open (ERR, "$ARGV[0].err") ||
      die "Unable to open $ARGV[0].err: $!";
while (<ERR>)
{
   if ( / line (\d+)\.$/ )
   {
      print STDERR;
      print STDERR "$lines[$1 - 1]"
   }
}
```

Extension

This time a subroutine is useful, and the method of reading is rather different. The down side is that we have to read the source file/s every time we want a line. It would be dangerous to cache all the lines of all the modules that could error, we might run out of memory.

```
sub get_line
{
   my $file = shift;
   my $line_no = shift;

   open (FILE, $file) || die "Unable to open $file: $!";
   while (<FILE>)
   {
      if ($. == ($line_no - 1))
      {
         close FILE;
         return $_
      }
   }

   close FILE;
   return "Invalid line: line_no\n"

}  # End of get_line

@ARGV || die "You must supply a program name";

# Chapter 10 Exercise 2
$result = system ("@ARGV >$ARGV[0].out 2>$ARGV[0].err");

if ($result)
{
   print STDERR "$ARGV[0] gave error ", $? >> 8, "\n"
}

exit 0 if ( -z "$ARGV[0].err" );

# Read the error file, and extract the file names and
line numbers

open (ERR, "$ARGV[0].err") ||
      die "Unable to open $ARGV[0].err: $!";

while (<ERR>)
{
   if ( / at (.*) line (\d)+\.$/ )
   {
      print STDERR;
      print STDERR get_line ($1, $2)
   }
}
```

Chapter 15: References

Exercise 1

The object of this exercise is to construct an array of arrays which stores details of files. The array is to be constructed as follows:

index	0	1	2	3
0	Filename	Mode	Size	Date last modified
1	Filename	Mode	Size	Date last modified
2	Filename	Mode	Size	Date last modified
n	etc…			

The Perl `stat` command takes a filename (or handle) as an argument, and returns a list of file attributes. The elements of interest are:

element index	**value**
2	mode, which includes type and permissions as a bit mask
7	size in bytes
9	time last modified (in seconds since epoc)

Obtain a list of filenames, we suggest all the Perl scripts in the current directory using `glob`. Now call `stat` on each one (`stat` *`filename`*), storing the required data in a multi-dimensional array.

Finally print the resulting array of arrays. Some formatting is required for the raw data, as follows:

mode convert the permissions to octal using:

```
printf "%lo", mode & 07777
```

time last modified convert to a readable date/time using:

```
scalar(localtime(mtime))
```

You might find that `printf` is the simplest way of printing each record. Your output should look something like this:

```
Ch16Ex1.pl          755    526 Tue Feb 11 13:22:54 2003
Ch16Ex2.pl          755    308 Thu Sep  4 10:58:11 2003
Ch16Ex3.pl          755    366 Tue Feb 11 13:22:54 2003
Ch16Ex4.pl          755    695 Tue Feb 11 13:22:54 2003
Ch16OptEx1.pl       755    346 Tue Feb 11 13:22:54 2003
Ch16OptEx1a.pl      755    351 Tue Feb 11 13:22:54 2003
Ch16OptEx2.pl       755    656 Tue Feb 11 13:22:54 2003
Template.pl         755    627 Tue Feb 11 13:22:54 2003
```

Note that on ***Microsoft Windows*** the permissions will probably be 666 (rw-rw-rw-).

Exercise 2

Given the following subroutine and program code, predict the output. Now run the program (Ch16Ex3.pl) and check your answer.

```
sub new_scalar_ref
{
   my $a = 1;
   return \$a;
}

$one = new_scalar_ref();
$two = new_scalar_ref();
$three = $one;
$four = $two;
$five = $$one;
$six = $$two;

$$one++;
$$one++;
$$two++;
$$three++;
$$four++;
$five++;
$six++;

print "$$one; $$two; $$three; $$four; $five; $six\n";
```

Exercise 3

Consider a text file, which consists of characters arranged in words. `%words` is a complex data structure in which a hash is indexed by (text) words, and contains array references. Each array reference points to an array that contains :

- the word itself
- the number of times it is used
- a reference to an array, which contains each of the lines containing the word

Now create `%words` from an input file, then print `%words` in a neat layout:

Word: *number-of-occurrences*
Text-line

repeat …

So, for the input file (word_input.txt) :

```
There once was
a dog that was
fond of kids
```

Output something like:

```
There: 1
There once was
that: 1
a dog that was
a: 1
a dog that was
dog: 1
a dog that was
kids: 1
fond of kids
was: 2
There once was
a dog that was
of: 1
fond of kids
once: 1
There once was
fond: 1
fond of kids
```

On UNIX: There is a further complication if you are using the supplied file `word_input.txt`, it is in "dos" format. Coming from a PC the line delimiter is "\r\n", how would you cope with this?

On Microsoft Windows: How would you read this file if it came from UNIX, i.e. with the line delimiter of "\n"?

Optional Exercises

Exercise 1

A 'pencil and paper' exercise, what will be printed from the following code?

```
$c = [\"Two", \"Enter", \"Hell", "Harold"];

$cp = [ 3, 0, 2, 1 ];

$cpp = \$cp;

${$c->[0]} =~ /.*(\w)$/;

$i = $$cpp->[2]*2;

# What is printed ?

print "${$c->[2]}$1 ";

print substr(${$c->[-$i]} ,$cp->[3]);

print substr(${$c->[1]}, $i);

print substr($c->[--$i], -$cp->[2]);

print "\n";
```

Exercise 2

Take a look at the code on the slide titled "Reference Counting: Example". It is provided for you in Template.pl, in the `labsolutions` sub-directory for this chapter.

Modify this code to:

> Process all the user records of the /etc/passwd file, not just the first three (for ***Microsoft Windows***, a `passwd` file is supplied in the `labsolutions` directory).
>
> Store the user information in a hash, with the key being the username and the value a reference to a hash containing the remaining user information.
>
> Display all the usernames and uids as before.

Hint: You may need to modify the `make_user` subroutine to return a two element list, the username and a reference.

Chapter 15: References - Solutions

Exercise 1 Solution

There are a number of alternatives for this problem, for example `map` could have been used instead of the `for` loops, and `sprintf` could have been used with `print` or `write/format` instead of `printf`. Any way that works is correct!

```
my @filearray;

for my $file (<*.pl>)
{
   my @a = ($file, (stat $file)[2, 7, 9]);
   push @filearray, \@a
}

for my $ref (@filearray)
{
   printf "%-20s %lo %6d %s\n",
      $ref->[0], $ref->[1] & 07777, $ref->[2],
      scalar(localtime($ref->[3]));
}
```

Exercise 2 Solution

The output generated by the program is:

```
4; 3; 4; 3; 2; 2
```

What happens is the following:

- The subroutine new scalar ref return a new, independent scalar reference each time you run it. (Because of the my).
- $one and $three point to the same scalar.
- $two and $four point to the same scalar.
- $five is a new scalar variable (not a reference), that has as value whatever $one points to when $five is assigned.

Exercise 3 Solution

The program below is a sample implementation:

```
use strict;

open (WORDS, "words_input.txt") || die "Cannot open: $!";
my %words;
my $line;

if ( $^O ne "MSWin32" )
{
   # The input file is in "dos" format
   $/ = "\r\n";
}

while ($line = <WORDS>)
{
   chomp $line;
   my @elems = split / /, $line;
   for (@elems)
   {
      if (!defined $words{$_})
      {
         $words{$_} = [ $_, 1, [ $line ] ];
      }
      else
      {
         my $aref = $words{$_};
         $aref->[1] += 1;     # One more line
         push @{$aref->[2]}, $line;
      }
   }
}

close WORDS;

# Neat output:
for (values %words)
{
   print "\t$$_[0]: $$_[1]\n";
   for (@{$_->[2]})
   {
      print "\t$_\n";
   }
}
```

This program could be improved by using a hash that has keys equal to values, and index that hash by the lines to be stored. Then don't store the full line in the array, but keep references to the hash values.

This kind of data-structure allows you to build full text indices, which can be very useful when you want to know the frequency and use of words, or if you want to build a search engine.

The end-of-line delimiter (`$/`) needs to be set to "`\r\n`" for UNIX Perl to read an MS-DOS format file. Perl on Microsoft Windows can read UNIX or MS-DOS files without having to alter the delimiter.

Optional Exercises - Solutions

Exercise 1 Solution

```
$c = [\"Two", \"Enter", \"Hell", "Harold"];

$cp = [ 3, 0, 2, 1 ];

$cpp = \$cp;

${$c->[0]} =~ /.*(\w)$/;
```

This regular expression searches for the last alpha-numeric character .
$c is a reference to an anonymous array, the first element of which is a reference to a scalar.

```
$i = $$cpp->[2]*2;
```

$cpp is a reference to a reference (cp). Third element is 2,
2 * 2 = 4, `$i = 4`

What is printed ?

```
print "${$c->[2]}$1 ";
```

"Hell" (third element of $c),
"o", first bracketed match from the last RE, space.

```
print substr(${$c->[-$i]} ,$cp->[3]);
```

The –4th element of $c is a reference to "Two". Take the substring starting at offset 1 (`$cp->[3]`), we get "wo".

```
print substr(${$c->[1]}, $i);
```

Here we get the fourth character from "Enter", 'r'.

```
print substr($c->[--$i], -$cp->[2]);
```

Finally we get the last two characters from "Harold" (which is not a reference).

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

Exercise 2 Solution

```
use strict;
my %Hash;

# Return a new hash at every call
sub make_user ($)
{
   my $line = shift;
   my ($userid, $groupid, $username, $shell) =
                              (split /:/, $line)[2,3,0,6];

   # Return a list, username and a reference to a hash
   return ($username,
     {uid => $userid, gid => $groupid, shell => $shell});
}

open (PWD, "/etc/passwd");
while (<PWD>)
{
   my ($name, $ref)  = make_user($_);
   $Hash {$name} = $ref;
}
close PWD;

for my $uname (keys %Hash)
{
    print "Name: $uname; Uid: $Hash{$uname}{uid}\n";
}
```

Chapter 16: Object Oriented Programming

Exercise 1

Create a PrintSupport class that has a constructor and the following methods, which just print their class and method name:

- setup
- layout
- display

Create a PrintSupport object and call setup(), layout(), and display() on the object.

Exercise 2

Create the classes Fax and Report that each derive from PrintSupport.

These classes have their own setup() and display() methods, but inherit the layout() method from PrintSupport.

Create Fax and Report objects, and call each method.

Exercise 3

Alter the Fax class so that its constructor can be inherited.

Exercise 4

Alter the Report class and introduce the extra method title page(). Then create a FaxedReport class that inherits first from Fax, then from Report.

Show from which class FaxedReport inherits setup(), display(), layout() and title page().

Exercise 5

How can you make sure that a FaxedReport's display() function is implemented by the Report class, while not changing the meaning of setup()?

Chapter 16: Object Oriented Programming - Solutions

Exercise 1 Solution

The PrintSupport.pm module below and the following main program are a sample implementation.

```
package PrintSupport;

sub new
{
   return bless {};
}

sub setup
{
   print "PrintSupport: setup() called\n";
}

sub display
{
   print "PrintSupport: display() called\n";
}

sub layout
{
   print "PrintSupport: layout() called\n";
}

1;
```

Main program:

```
use PrintSupport;

my $doc = new PrintSupport;
$doc->setup();
$doc->layout();
$doc->display();
```

The program will generate the following output:

```
PrintSupport: setup() called
PrintSupport: layout() called
PrintSupport: display() called
```

Exercise 2 Solution

A sample Fax.pm module is shown below:

```
package Fax;

use PrintSupport;

BEGIN
{
   @ISA = ('PrintSupport');
}

sub new
{
   return bless {};
}

sub setup
{
   print "Fax: setup() called\n";
}

sub display
{
   print "Fax: display() called\n";
}

1;
```

A sample Report.pm module is shown below:

```
package Report;

use PrintSupport;

BEGIN
{
   @ISA = ('PrintSupport');
}

sub new
{
   return bless {};
}

sub setup
{
   print "Report: setup() called\n";
}

sub display
{
   print "Report: display() called\n";
}

1;
```

A sample main program is shown below:

```
use Fax;
use Report;

print "Doing Fax...\n";
my $fax = new Fax;
$fax->setup();
$fax->layout();
$fax->display();

print "Doing Report...\n";
my $report = new Report;
$report->setup();
$report->layout();
$report->display();

print "Done\n";
```

It generates the following output, which clearly shows how method inheritance works in perl:

```
Doing Fax...
Fax: setup() called
PrintSupport: layout() called
Fax: display() called
Doing Report...
Report: setup() called
PrintSupport: layout() called
Report: display() called
Done
```

Exercise 4 Solution

A constructor can be inherited if it uses the two-argument form of the bless-command. The second argument to bless must be the actual class name specified. If the new method is called like new FaxChild, the actual class name passed to the new method is FaxChild. By merely using the first parameter to new with the bless command, the constructor is safe to inherit. The code below shows how:

```
# Can be inherited by any child class
sub new
{
      my $class = shift;
      return bless {}, $class;
}
```

Exercise 5 Solution

A new simple method added to Report.pm is show below:

```
sub title_page
{
   print "Report: title_page() called\n";
}
```

A sample (and simple) FaxedReport.pm module is shown below:

```
package FaxedReport;

use Fax;
use Report;

BEGIN
{
   @ISA = ('Fax', 'Report');
}

# All methods inherited
1;
```

An example main program:

```
use FaxedReport;

my $doc = new FaxedReport;
$doc->setup();
$doc->title_page();
$doc->layout();
$doc->display();
```

This generates the following output:

```
Fax: setup() called
Report: title_page() called
PrintSupport: layout() called
Fax: display() called
```

Exercise 6 Solution

The easy solution would be: change order in the @ISA array, to make sure Report::display() is used instead of Fax::display(). However, this will also change the setup() method called. A real solution is to override display() in the FaxedReport class, and to actually implement it using Report::display() (not copying any code). A sample implementation, in the form of an updated FaxedReport.pm module, is shown below:

```
package FaxedReport;

use Fax;
use Report;

BEGIN
{
   @ISA = ('Fax', 'Report');
}

# display() overrides inheritance order
sub display
{
   # All arguments passed unchanged
   return Report::display(@_);
}
1;
```

It generates the following output:

```
Fax: setup() called
Report: title_page() called
PrintSupport: layout() called
Report: display() called
```

.