UNIX/Linux

LAB MANUAL

ET4725 – Spring 2016

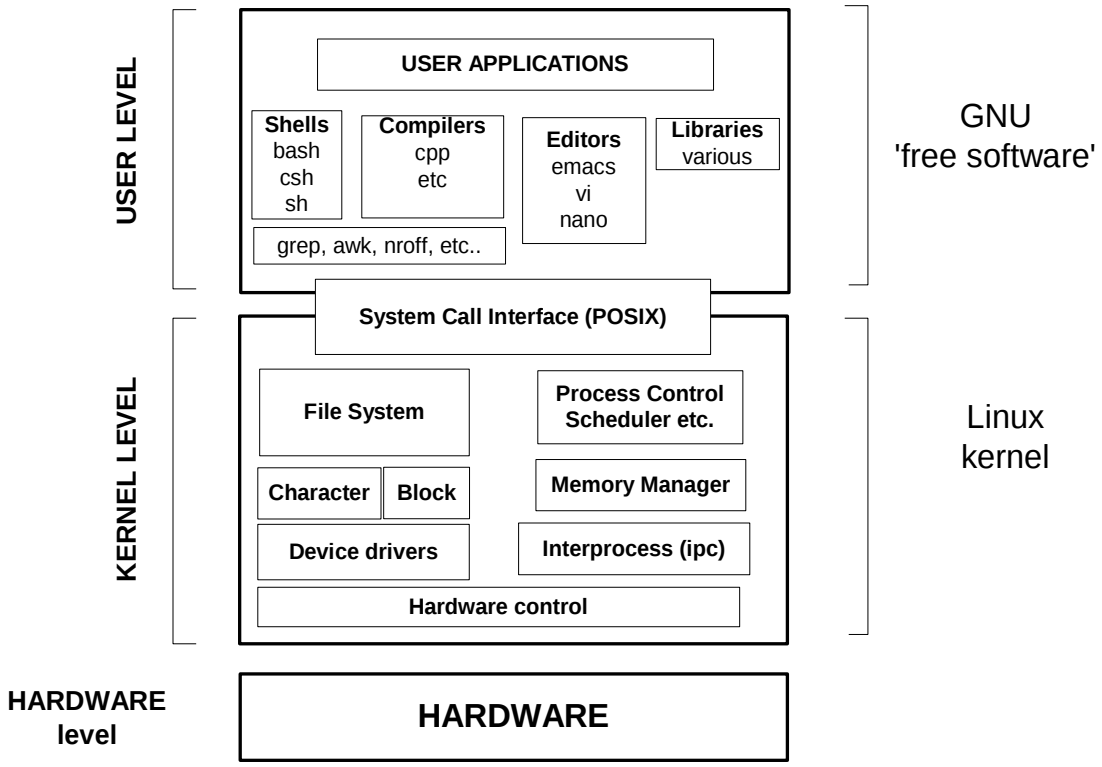
TUTORIAL STYLE INTRODUCTION

Donal Heffernan
University of Limerick

Ver: 2/Feb2016

LAB BOOK
This is an exact from the full lab book

SIMPLIFIED UNIX (LINUX) BLOCK DIAGRAM



1 Getting started

The shell

Before you start to write any programs under UNIX/Linux it is most important that you are confident with the basic shell commands. The word “shell” is a term for the user command level interface, where a user can directly write commands to instruct the computer to perform certain actions. The UNIX/Linux shell, sometimes referred to as a command interpreter, provides a standardised approach for interacting with the system. The shell can be used interactively, or in a programmable sense using shell scripts. In the interactive mode the user types specific commands and reads the outputs on the screen. In using shell scripts, the user develops a program by typing a set of commands into a text editor. The program is executed directly as a shell script program.

UNIX/Linux systems can support different shells. This document will introduce the Bash shell in particular. The word Bash is an acronym for ‘Bourne-Again SHell’ and it is a pun on Stephen Bourne's classic UNIX Bourne shell. Bash was written by Brian Fox for the GNU Project, intended as a free replacement for the Bourne shell, and was first released in 1989. Other UNIX shells, such as the Korn shell (ksh), from which Bash is derived, are similar to Bash and once the reader learns Bash, she will find it easy to use the other shells also. Some other UNIX shells include the Tenex shell (tcsh), the Bourne shell (sh), the C Shell (csh), the Falstad shell (zsh), and the rc shell. Once a student becomes familiar with one UNIX shell, then it is easy to learn another shell, as many of the commands are quite similar. Note the C shell is not recommended to learn as a first UNIX shell as it is quite different to the other shells and it has other inherent problems. The Tenex shell is a superset of the common C shell.

In summary, the Bash shell has been chosen for this document as it is a good general-purpose modern shell, and it is the default shell on many systems including Linux and Apple OS X. Bash has been ported to Microsoft Windows and is available with Cygwin, MinGW and DOS with the DJGPP project. Bash is available on Android using terminal emulation applications.

The Bash shell contains many useful standard short programs, referred to as utilities, for doing jobs, such as copying files, recording time and dates etc. These utilities have predefined names, which are used to run the utilities. The utility names are referred to as commands. Thus the words commands and utilities are often used interchangeably. Once you understand how to use the individual shell commands it will then be easy to learn how to write simple shell script programs.

The man and info pages

The shell includes *command manual* pages, which are the so-called **man** pages. This is effectively an on-line user's manual for the shell commands. For each shell command you can call up a text description for the command by simply typing **man** *command name*. The description for each command is very complete but not always easy to understand by the novice. In addition to the **man** pages, there are *information* pages about each command, referred to as the **info** pages. These pages are accessed using the **info** command.

The objective of this tutorial is to lead the student to a point where he/she feels comfortable with the common UNIX/Linux shell commands and becomes confident enough to explore each command on his/her own through the **man** pages and the **info** pages.

File names and path specifications

In UNIX/Linux filenames are strictly case sensitive, i.e. they do not confuse upper case and lower case characters. In file path name specifications use the forward slash (/) character, unlike the equivalent use of the backslash (\ ) character in Microsoft Windows.

In using shell commands and programming, the proper use of spaces between words and characters is very important, so be sure to use spaces exactly as shown in the examples.

Some useful control keys

CONTROL KEYS

CTRL-C	This key combination will kill (break) a process that is running on the terminal.
CTRL-D	This is the 'end of file' (EOF) key combination. Can be used as a fast logout from a terminal or to finish inputting text from the keyboard.
CTRL-Z	This key combination is used to suspend a process and can put a process temporarily in the background.
CTRL-S /CTRL-Q	Control-S can be used to pause a fast scrolling screen – the scrolling can be resumed again by typing Control-Q.
CTRL-L	Clears the terminal window's display (same as the **clear** command)

Text editors

There are a number of text editors available for UNIX/Linux. The **vi** editor (visual editor) and the **emacs** editor are probably the best-known editors. The **vim** editor (**vi** i**m**proved) is a popular modern *vi* editor for Linux users. A simple command line editor, called **nano,** is available on most UNIX/Linux systems. Use any text editor to create the files listed below. If you are new to UNIX/Linux you should select a simple editor that is intuitive to learn. For example, if you are using a **Gnome** desktop, you could try using the **gedit** editor.

Creating some sample text files

Use an editor to create some text files which will be used is the early part of this tutorial.

- Create a file called **title** with the following content:

```
THE SONG OF THE WANDERING AENGUS

by

W.B.Yeats

------------------------------------------------------------
```

- Create a file called **verse_1** with the following content:

```
I went out to the hazel wood,
Because a fire was in my head,
And cut and peeled a hazel wand,
And hooked a berry to a thread;
And when white moths were on the wing,
And moth-like stars were flickering out,
I dropped the berry in a stream
And caught a little silver trout.
```

- Create a file called **verse_2** with the following content:

```
When I had laid it on the floor
I went to blow the fire aflame,
But something rustled on the floor,
And some one called me by my name:
It had become a glimmering girl
With apple blossom in her hair
Who called me by my name and ran
And faded through the brightening air.
```

- Create a file called **verse_3** with the following content:

Though I am old with wandering
Through hollow lands and hilly lands,
I will find out where she has gone,
And kiss her lips and take her hands;
And walk among long dappled grass,
And pluck till time and times are done
The silver apples of the moon,
The golden apples of the sun.

2 Shell Commands

2.1 Introducing Some Common Commands

Some of the more common UNIX shell commands will be introduced. The first command that we will look at is the **cat** (concatenate) command. Before we attempt to try this command, let's look at some ways on getting help with using a command. We will take the **cat** command as a first example command to see what help is available for this and other commands.

2.1.1 The man pages

Type **man cat**

The screen will show an overwhelming amount of information about the **cat** command. You do not need to study all of this information now, rather use the **man** pages as reference information. A **pager** utility is used to show all of the **man** page content information on the terminal screen. Try the following keys to view the screen:

Hit the **space bar** to move down a screen
Hit the **b** key to move back up a screen
Hit the **up arrow** ↑ key to move up a line
Hit the **down arrow** ↓ key to move down a line
Hit the **q** key to exit the man pages

The **apropos** command is useful to search all the **man** pages and display a list of topics that are related to a query. The apropos command is effectively a wrapper for the "`man -k`" command.

2.1.2 The info pages

The **info** (information) pages are similar to the **man** pages.

Type **info cat**

Use the same pager keys as the **man** pages to navigate the **info** pages. Again, please note that these pages are available as reference information and do not need to be studied right now.

2.1.3 The help option

There is also a simple help option for most commands.

Type **cat - - help**

A summary display of the command options is displayed. Note the use of the double hyphen in calling the help option.

2.1.4 The command prompt

Traditionally, a shell prompt ends with one of the symbols $, % or #. The $ indicates a shell that is compatible with the Bourne shell, i.e. POSIX shell, Korn shell, or Bash shell. The % indicates a C shell, e.g. csh shell or tcsh shell. The # indicates the shell is running under a superuser account (root account).

So, the **$** symbol is the generic shell prompt for the Bash shell. For example the following command could be typed at the shell prompt:

$ whoami

The computer will reply as follows (assuming that your name is **john_smith**)

$ whoami
john_smith

As stated, the **#** symbol is used as the shell prompt when you are logged on as a **root** user. However, you should avoid using root user sessions, until you are an experienced user, so as to avoid any unnecessary accidental harm to the system.

2.1.5 The cat command

The **cat** (con**cat**enate) command is useful for displaying the contents of a text file and for concatenating files.

Example: Type **cat verse_1**
You will see the contents of the **verse_1** file listed to the screen

Example: Type **cat verse_1 verse_2 > big_file**
This command concatenates the two files and directs the output to a file called **big_file**.

Example: Type **cat verse_3 >> big_file**

Here the >> operator will cause the output of file **verse_3** to be appended to file **big_file**, i.e. big_file is not overwritten but rather the file verse_3 is added to big_file.

Now, create a file called **full_poem** using the **cat** command as follows:

Type **cat title verse_1 verse_2 verse_3 > full_poem**

Now type **cat full_poem** to view the contents of the file **full_poem**.

2.1.6 The echo command

The **echo** command is used, typically, to output text strings to the screen.

Try the following examples:

echo Hello outputs ‘Hello’ to the screen
echo “Hello” also outputs ‘Hello’ to the screen

The **-e** option is used to interpret backslashed escape characters in the string. Try the following examples:

echo -e “Hello \n” outputs ‘Hello’ to the screen and adds a new line

echo -e "Hello \n \a" outputs 'Hello' to the screen, adds a new line and sounds the bell

The **-n** option can be used to specify that the default new line is not to be generated.

There are many options for the **echo** command. Look these up in the **man** pages.

2.1.7 The printf command

It is interesting to note that the Bash shell also supports the **printf** command. The **printf** command does a formatted print based on the well-known C language **printf()** statement. It is intended to be a successor for the **echo** command and has a richer feature set. It is the preferred POSIX solution towards standardized code.

The following **printf** command example behaves like its equivalent **echo** command, so theses two commands are equivalent:

printf "Hello \n\a" outputs 'Hello' to the screen, adds a new line and sounds the bell.

echo -e "Hello \n \a" outputs 'Hello' to the screen, adds a new line and sounds the bell.

We will visit the **printf** command again in a little while.

2.1.8 The more command

For outputting large files to the screen the **cat** command is not very useful because we will see the entire file data rolling too quickly up the screen. The **more** command outputs just one screen full (e.g. 24 lines) of information at a time, under control of a pager. We already introduced the pager with the **man** command above.

Note, the **more** command is similar to the **less** command. Some systems support the **less** command but not **more**. However, l**ess** is not **more** – but **less** is similar to **more**, more or less! In fact **less** has more features than **more**. For example **less** supports backwards paging.

Hitting the **space bar** will output another screen page of data and so forth. Hit the **b** key to go back a page. A message at the bottom of the screen will tell you how much of the file has been displayed. Hit the **enter** key to display the next page. Hit the **q** key to quit.

Create a larger text file using the following commands:

Type **cat full_poem > large_file**

cat full_poem >> large_file

cat full_poem >> large_file

Display **large_file** to the screen, first using **cat** and then using **more.** You will appreciate the difference between these two commands.

2.1.9 The pwd command

The **pwd** (**p**rint the current **w**orking **d**irectory) command will tell you where you are within the directory structure.

Simply type **pwd**

The response will tell you where you are within the directory structure e.g.: **/usr/customers/donal**

2.1.10 The ls command

The **less** command (**list**) is a very useful shell command, which can list files. The command has a wide range of options. Look at the **man** pages for the **ls** command (type **man ls**), or use the help facility by typing the **ls --help | less** command.

Simply type the command **ls** at the command prompt. You will see the list of file names sorted down columns in alphabetical order, something like the following:

```
$ ls
dec             don.tar         proc_1          proc_5          process2        ps
dec1            dump            proc_1.c         proc_5.c       proc_p          psfile
delay           game            proc_2          proc_6          proc_p.c        ps_temp
delay_prog      perfromance     proc_2.c        proc_6.c        procs_per_user  pstest1
delay_prog.c    pid             proc_3          proc_9          proc_X          pstest2
dem             pid.c           proc_3.c        proc_9.c        proc_X.c        vtemp
dem.c           play            proc_4          process1        prog_5n
don             prc_4.c         proc_4.c        process_1       prox
```

Now try the command: **ls –l** *(NB this is bash elle, not dash one)*

You will see a response something like this:

```
total 15256
-rw-r--r--   1   don2006  myGroup      3523    Jan 16 09:03   dec
-rw-r--r--   1   don2006  myGroup       448    Jan 16 09:03   dec1
-rwxr-xr-x   1   don2006  myGroup      5257    Jan 16 09:03   delay
-rwxr-xr-x   1   don2006  myGroup      5266    Jan 16 09:03   delay_prog
-rw-r--r--   1   don2006  myGroup       431    Jan 16 09:03   delay_prog.c
-rwxr-xr-x   1   don2006  myGroup      4737    Jan 16 09:03   dem
-rw-r--r--   1   don2006  myGroup        70    Jan 16 09:03   dem.c
drwxr-xr-x   2   don2006  myGroup      4096    Jan 16 09:02   don
-rw-r--r--   1   don2006  myGroup    112640    Jan 16 09:03   don.tar
-rw-r--r--   1   don2006  myGroup  14792036    Jan 16 09:03   dump
drwxr-xr-x   2   don2006  myGroup      4096    Jan 16 09:02   game
etc……..
```

The purpose of each field will be explained a little later on.

Try **ls -l /** to list the root directory

Try **ls –d */** to list directory entries only (a useful command!)

2.2 The Glob Constructs (wild cards)

The so-called wild cards or file globs can be used in specifying a file name, using filename patterns that include special characters such as * and ? etc. Under UNIX/Linux these are referred to as **glob constructs**. This type of pattern matching is part of a subject called *regular expressions*, which can support powerful pattern languages, as will be introduced later on.

The ? Glob Construct

Any single character in a file name is made wild.

e.g.: **ls ?est_demo** might list files such as the following (fictitious example) :

pest_demo best_demo test_demo

The * Glob Construct

Any string of zero or more characters is wild

e.g.: **ls *t_demo** might list the following:

pest_demo best_demo test_demo bat_demo bert_demo t_demo

The [ ] Glob Construct

Any character inside the square brackets can be used to match **one** (and only one) character in the file name.

e.g.: **ls num[xyz]test** might list the following:

numxtest numytest numztest

Since only one character within the bracket can be selected, a file called **numxytest** would not be selected.

ls [a-e]test will list any file name whose first character in the name is in the range **a** to **e** and the rest of the file name is 'test'. For example, a valid list might be:

atest btest ctest

Try making up some of your own examples using these glob constructs and see the results.

2.3 Commands to Copy (cp), Move (mv) And Remove (rm) Files

2.3.1 The cp command

The **cp** command is the UNIX/Linux **c**o**p**y command.

For example, type: **cp verse_1 verse_temp**

Type **ls** to see that the new file **verse_temp** does now exist, and that it is a copy of the **verse_1** file.

2.3.2 The mv command

The **mv** command moves (**m**o**v**es) a file. It effectively renames the file.

For example, type: **mv verse_temp verse_demo**

Use **ls** to see what happened to the file. You will see that the file **verse_temp** no longer exists as it has been effectively renamed to **verse_demo**.

2.3.3 The rm command

The **rm** command removes (**r**e**m**oves) a file, i.e. deletes a file.

For example, type: **rm verse_demo**

Use **ls** to satisfy yourself that the file **verse_demo** no longer exists.

2.4 Using Directories

Earlier we used the **pwd** command to see where we were in the directory structure. Now we will move about within the directory structure and create a new directory. The **cd**, **mkdir** and **rmdir** commands will be shown to respectively: change, make and remove directories.

2.4.1 The ls command

For practice try these commands:

ls . lists the files in your current directory

ls .. lists the files in your parent directory

ls / lists the files in the root directory

ls *pathname* lists the directory specified in the pathname

Some more **ls** options:

ls -R lists files and sub directories (recursively)

ls -a lists **a**ll files including files which begin with '**.**' i.e. the UNIX utilisation files

ls -l list file details, see later for more detail (this option is **dash elle**).

ls –1 lists files in a single column (this option is **dash one (-1)**, not **dash elle (-l)**)

Examples to list information on directories:

ls */ lists the contents of all the subdirectories
ls –d */ lists only the directories in the current directory

Use the **man** pages to learn more about the **ls** command and its options.

2.4.2 The cd command

The **cd** command (**c**hange **d**irectory) changes the current directory. Note, when you log onto a UNIX/Linux system, your current directory is, by default, your **home** directory e.g. **/home/claire**.

For practice try the following examples to change to other directories, each time you change to a new directory you can verify by using **pwd** to show where you are and use **ls** to see the contents of the new directory:

cd .

cd ..

cd /

cd .. /../

cd *pathname* (try some specific path name)

Make sure you understand what each one of the above commands do.

If you get lost somewhere within the maze of the directory structure and want to go back to your home directory, simply type:

cd ~ changes to your home directory (tilde (~) represents your home directory).

or, even more simply, type:

cd

2.4.3 The mkdir command

The *make directory* command, **mkdir**, is used to create a new directory.

For example, make a new directory called **my_test** and copy all the files from your home directory to the **my_test** directory. Try the following steps:

cd ~ go to your home directory

mkdir my_test make a new directory called **my_test**

cd my_test	change into **my_test** directory
ls	this will show that no files exist here yet as this is a new directory
cp ../* .	copies all files from parent directory to **my_test** directory

Obviously, you could have copied these files without actually going into the new directory, but there are many ways to navigate the file system.

If you have directory files in the parent directory you will receive an error message something like:

cp: ../my_test: omitting directory

To copy directory files you must specify the **-r** option (see manual pages for **cp**)

Type **cp -r ../* .**	Now this command should copy all files including directory files. Try this to ensure that it works.

2.4.4 The rmdir command

The **rmdir** (**rem**ove **dir**ectory) command will remove the specified directory or directories.

Switch back to your home directory (**cd ~**) and try to remove the **my_test** directory by typing:

rmdir my_test

Just as you would have expected, you get a message telling you that the directory is not empty, and thus removal of the directory is denied.

How would you remove a directory and its contents without first deleting the contents?

You would use: **rm -r my_test** Try this, it works!

But this is not the **rmdir** command, why does it use the **rm** command? Well, read the **rm** man pages and you will get some explanation.

2.4.5 The directory stack

The **pushd** and **popd** commands work with the command line directory stack. The **pushd** command saves the current working directory in memory. An example usage:

pushd ~/examples/my_vault

The **popd** command returns to the path at the top of the directory stack. This directory stack is accessed by the **dirs** command, which will display the list of directories in the directory stack, as pushed to the stack using the **pushd** command. The command **dirs +n** displays the nth. entry in the list, in left-to-right order.

2.5 File Details

2.5.1 Exploring files using the ls command

Type **ls -l** and you will get a screen something like the following:

```
total 15256
-rw-r--r--    1    donal    myGroup      3523    Jan 16 09:03    dec
-rw-r--r--    1    donal    myGroup       448    Jan 16 09:03    dec1
-rwxr-xr-x    1    donal    myGroup      5257    Jan 16 09:03    delay
-rwxr-xr-x    1    donal    myGroup      5266    Jan 16 09:03    delay_prog
-rw-r--r--    1    donal    myGroup       431    Jan 16 09:03    delay_prog.c
-rwxr-xr-x    1    donal    myGroup      4737    Jan 16 09:03    dem
-rw-r--r--    1    donal    myGroup        70    Jan 16 09:03    dem.c
drwxr-xr-x    2    donal    myGroup      4096    Jan 16 09:02    don
-rw-r--r--    1    donal    myGroup    112640    Jan 16 09:03    don.tar
-rw-r--r--    1    donal    myGroup  14792036    Jan 16 09:03    dump
drwxr-xr-x    2    donal    myGroup      4096    Jan 16 09:02    game
-rw-r--r--    1    donal    myGroup    270337    Jan 16 09:03    performance
-rwxr-xr-x    1    donal    myGroup      5220    Jan 16 09:03    pid
etc……..
```

What does each field in the above represent? First of all, each UNIX/Linux user is assigned a unique user ID number. When you create a file, you normally become the file **owner**. The owner of the file can specify access permissions to the file, e.g. **read** access permission, **write** access permission, **and execute** access permission.

rwx access permissions for regular files:

r can **r**ead a file
w can **w**rite to a file
x can e**x**ecute a file

rwx access permissions for directory files are as follows:

r can **r**ead names of entries in the directory
w can create new files and sub-directories
x can check to see if a file or sub-directory e**x**ists by specifying the file or sub-directory name. Not as strong as the **r** permission.

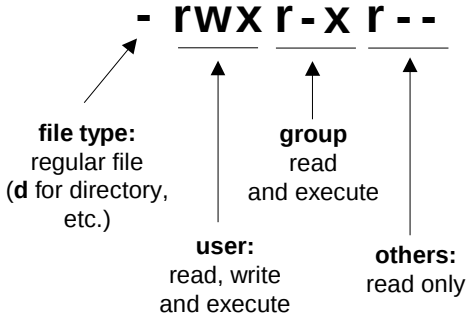
UNIX/Linux also has the **group** concept. A number of users can belong to a group. In fact one user can belong to several groups but one group will be the user's primary group. Just like the owner's access permissions, the group will also have access permissions, i.e. another user in the group can read or write or execute your file. The user access permissions are noted using the **rwx** access convention also.

So, for a given file the **user** and the **group** access permissions are defined; but what file access permissions should apply to everyone else? Well, there are another three **rwx** bits to define the access permissions for everyone else. Everyone else is often referred to as '**others**'.

Let's look at the following line which results from a **ls -l** command:

-rwxr-wr-- 2 donal userx 512 oct17 15:11 test_file

The first field is used to define the file type and the access permissions, as follows:



The second field (showing the number **2** in this example) shows that there are two **links** to the file (the concept of a link will be explained later on.)

The next field tells us who is the file's owner (**donal** is the owner in this example**)**

Next field (**userx** in this example) specifies the file's group (ignore for now)

The next field shows the file size in bytes (**512** bytes in this example)

The next field shows the **date** and **time** stamps for the file (**oct 17 15:11 in this example**)

The final field shows the file name (**test_file** in this example)

Note the access permissions are sometimes referred as the '9-bit' permission field. In the above example the **rwx, r-x, r- -** code is represented internally in the UNIX file system by the binary code **111,101,100** (or 754 on octal) where each bit position represents a permission status.

2.5.2 Touching files

The touch command

The touch command is useful. It updates the last *modification time* and *access time* of named files to the current system time. If a specified file does not exist, then **touch** will create that file with a zero size.

Try the following:

touch BrandNew

Use the **ls –l** command to verify that a new file called **BrandNew** has been created and that this file has a zero size.

2.6 Changing File Owner And Access Permissions

2.6.1 The chmod command

The **chmod** command will allow you to change a file's access permissions. The command uses the following parameters:

u means **user** who owns the file
g means **group** that owns the file
o means **others**
a means **all**

\+ means permission **on**
- means permission **off**
= means add permission (on) but remove all other permissions

r means **read** permission
w means **write** permission
x means **execute** permission

For example:

chmod a-w test_file	says you want to **turn off** the **write** permission for **all** i.e. user, group and others.
chmod u+rwx test_file	says you want to **turn on** the **read**, **write** and **execute** permissions for the **user**.
chmod go+w test_file	says you want to **turn on** the **write** permission for the **group** and **others**.
chmod 644 test_file	this sets the file's permissions to (rw - r- - r- -); saying to give **read/write** permission to the **user** and **read-only** permission to everyone else.

For a given file, practice revoking (turn-off) and granting (turn-on) access permissions (rights) for various files and directory files. To be on the safe side make some temporary files to practice on.

2.6.2 The chown command

The **chown** (**ch**ange **own**er) command changes the file ownership. The user level access here is restricted. Look up this command in the **man** pages to find out more.

The **chown** command allows a root user (**chown** is not available to ordinary users) to change the ownership for a file.

For example:

chown claire test_file	changes the owner of the file **test_file** to **claire**.
chown daniel test_file	changes the owner of the file **test_file** to **daniel**.

2.7 Finding Files

2.7.1 The locate command

The **locate** command will look for a file which matches a search file name. The **locate** command will quickly return any matching results, as **locate** searches a procreated database file. However, this database file needs to be updated frequently so that it can contain an up-to-date list of current files. The **updatedb** command will perform the update. Note that root user privileges are required to run **updatedb**.

Note, the **slocate** command, secure locate, is a better command as it does not search directories where the user does not have access privileges.

Try running the **updatedb** command on your system, if you have the rights to do so. Beware that the command may take a long time to complete.

Try using the **locate** command as follows:

locate locatedb

The results will advise if a **locatedb** file or an **slocatedb** file exists, and will show the path to the files.

2.7.2 The find command

Whereas the **locate** command searches a database file for matching files, the **find** command will actually parse through the real file system, searching for files which match user defined criteria. Hence the **find** command will be much slower than the **locate** command, but it acts on current file information and provides great flexibility in defining the search criteria.

WARNING – it might seem that a find command will take forever to complete!

The **find** command will perform a recursive search through the directory structure.

The general format for the **find** command is:

find *pathname* *expression* *-print*

Note the **–print** option is enabled by default so there is no need to specify it when you use **find**. The **find** command is recursive in its search through directories.

Read the **find** command's man pages.

Examples: Try the following **find** commands and satisfy yourself that you understand what each command is doing:

find . -name "verse*"
Finds all files and directories, starting at the current directory level (.) which have the string 'verse' as the start of the file or directory name, and lists the output to the screen.

find / –user Ann2014
Finds all files and directories starting at the root directory level (/) which are owned by the user Ann2014.

find . -type f
Finds all normal files (f), starting at the current directory level (.) and lists the output to the screen.

find . -ctime 7
Finds all files starting at the current directory level (.) which have been changed seven days ago (-ctime 7) and lists the output to the screen.

find . -ctime -8
Finds all files starting at the current directory level (.) which have been changed within the past seven days (-ctime -8) and lists the output to the screen.

find . -atime 3
Finds all files starting at the current directory level (.) which have been accessed three days ago (-atime 3) and lists the output to the screen

find . ! -atime -3
Finds all files starting at the current directory level (.) which have **not** been accessed within the past two days (! -atime -3) and lists the output to the screen.

find / -type d
Finds all directory files (i.e. files of type d), starting at the root (/) directory, and lists the output to the screen.

There are many other interesting search options for the find command. Look them up!

2.8 Redirection

Redirection (>, <, >>)

By default the output from a command is written to the screen, and a command's input is assumed to be from the keyboard. There is also the concept of error messages being displayed to the screen.

The redirection symbols can be used to specify files as inputs or outputs for a command, as shown in some of the following examples.

Redirect a command's output to a file using the > character

The command **cat testfile** will print the content of t**estfile** to the screen by default.

The command **cat testfile > fileOne** will print the content of **testfile** to the file named **fileOne**. Note, if **fileOne** had already existed, then its contents would have been overwritten.

Note both of the following syntax notations are correct:

command >file
command > file

Append a command's output to a file using the >> characters

The command **cat testfile >> fileTwo** will print the content of **testfile** to the file named **fileTwo**. However, the file **fileTwo** will not be overwritten, but rather the **testfile** content will be appended to the file **fileTwo**.

You can specify redirection in command lines.

What will the following command do?

ls -l > logfile

And in what way is this command different?

ls -l >> logfile

A file becomes an input for a command by using the < character

For example, the **sort** command is useful to do a line by line sort on an input file (the sort command will be explained shortly).

Try this example:

sort < somefile the **somefile** file is sorted by line and outputted to the screen

An interesting use of redirection using the cat command

Often we use the **cat** command as follows: **cat file_name**, which will list the contents of **file_name** to the screen. The **cat** command is taking **file_name** as its input and it is outputting to the standard output device, i.e. the console screen. The command **cat > scrapfile** specifies the file **scrapfile** as its output and the input, which is not specified in this command, defaults to the console i.e. the keyboard.

Type **cat > scrapfile**. Now type some characters on the keyboard and they will be written to the file **scrapfile**. To terminate the input type the end-of-file control: CRTL D.

This is a very quick way to generate a small text file without using an editor.

Try this example:

Type: **cat > hamlet**

Now the **cat** command is expecting characters from the keyboard.

Type the following text:

To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune,
Or to take arms against a sea of troubles,
And by opposing end them?

Hit the CRTL D keys to terminate the input text.

Now type:

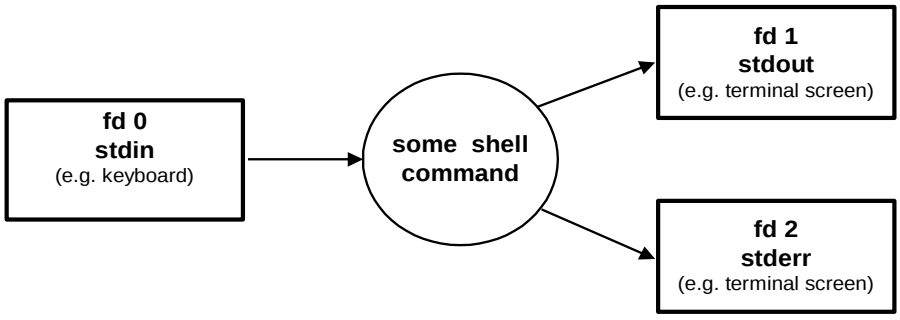
cat hamlet Now you know how to create a very simple text file without using an editor.

2.9 More I/O Redirection

The shell commands use file descriptors (referred to as 'fd') to identify the standard input (**stdin**), standard output (**stdout**) and standard error (**stderr**), as follows:

Standard I/O	File descriptor	Default device
stdin (standard input)	0	keyboard
stdout (standard output)	1	screen
stderr (standard error)	2	screen

A file descriptor is simply a number that is assigned to a device, or a file (or other such as pipes, sockets, or terminals), for convenient internal reference by the operating system kernel. By default fd 0 is assigned to the standard input (stdin), fd 1 is assigned to the standard output (stdout) and fd 2 is assigned to the standard error (stderr).



Some examples of the special redirection operators are given below. We have already looked briefly at the shell's basic redirection operators for output redirection (>), input redirection (<), and appending (>>). Now, we will look at some commonly used redirection operators that use the file descriptors. We can direct a file descriptor to a specified file using the following operator, where **n** is the file descriptor number:

n> filename

A common example is to redirect the standard error (the number for the stderr fd is 2) to a file using the code:

2> filename

Assume that your current working directory contains four files which have file names starting with the letter ‘d’ and there are no file names that start with the letter ‘z’. Assume that you will type the following command, to list all files that begin with the letter ‘d’ and all files that begin with the letter ‘z’.

ls d* z*

Assume the command returns the following information:

ls: z*: No such file or directory
dan.tar document1 document2 duplicatefile

The first line in the output is an error message which is intended for the **stderr** device and the second line is a normal output for the **stdout** device. Since the screen is the default output for both **stdout** and **stderr**, both messages appear on the screen, as seen above.

Assume that you type the following command:

ls d* z* 2> errorfile

This command causes the **ls** command to send its **stderr** output (remember **stderr** has the file descriptor number 2, hence **2>)** to a file called **errorfile**. Now only the **stdout** output is sent to the screen. Thus the following output appears on the screen:

dan.tar document1 document2 duplicatefile

The **2>** character pair is saying that the file descriptor number 2 (i.e. **stderr**) is redirected to the **errorfile** file. If the character set **2>>** was used, then the **stderr** output would have been **appended** to the **errorfile** file.

The &n operator

The **&n** represents the file descriptor **n**, so **>&n** is saying that the standard output of a command is to be redirected to the file descriptor **n**. So we use the **>&** to duplicate fd 1 and put this duplicate in fd **n**.

Here is an example:

ls d* z* 2> errorfile1 >&2

Now, both the **stderr** and **stdout** are directed to the file **errorfile1**. The command is evaluated from left to right (as usual). The **2> errorfile1** code causes the **stderr** (file descriptor 2) to be redirected to the file **errorfile1**. The **>&2** code causes the **stdout** output to be redirected to the **stderr** (file descriptor 2) device, and since the **stderr** has been already redirected to the file **errorfile1**, in this command, then **stdout** is directed to **errorfile1**.

Some additional comments

1>&n has the same meaning as **>&n** above, since the **stdout** (file descriptor **1**) was assumed in **>&n**. Thus the command line above could have been written as:

ls d* z* 2> errorfile1 1 >&2

The **m>&n** character set shows a general representation, where the file descriptor **m** is redirected to the file descriptor **n**.

The **<&n** character set shows input redirection where the file descriptor **n** is redirected to the standard input.

2.10 Sorting Text Files

2.10.1 The sort command

The sort command will sort the contents of a text file based on some well-defined criteria.

For example, if you use the **cat** command to display the content of a file called McGee:

cat McGee
There are strange things done 'neath the midnight sun
By the men who moil for gold.
The arctic trails have their secret tales
That would make your blood run cold

Now, you can sort the **McGee** file's output as follows:

sort McGee
By the men who moil for gold.
That would make your blood run cold
The arctic trails have their secret tales
There are strange things done 'neath the midnight sun

Now try the **sort** command on some text file of your own, e.g.: **sort full_poem**

You will see that the **sort** command sorts the file to the screen display, by line, in alphabetical order. The content of the input file is unchanged. To sort the input file to an output file try the following:

sort full_poem > poem_sorted

Use the command **cat poem_sorted** to convince yourself that the sort did happen.

Multiple files can be sorted. For example try the following command and then inspect the output file:

sort verse_1 verse_2 verse_3 > sort_file

The **sort** command includes some powerful sort options. Look up the **sort** command in the **man** pages. There are interesting options. For example, look at the following command that uses sort's **-g** and **–r** and **-k** options:

As an example, type **ls –l > myList** to create a file called **myList** that contains a listing of your files, in the following format:

```
-rw-r--r--    1    donal    myGroup      448    Jan 16 09:03    dec1
-rwxr-xr-x    1    donal    myGroup     5257    Jan 16 09:03    delay
-rwxr-xr-x    1    donal    myGroup     5266    Jan 16 09:03    delay_prog
etc…
```

This creates a file called **myList** which contains the list of files in your current directory.

Now, assume you want to sort these files in order of the size of the actual files. Note, the 5th column contains the file size information, as seen in the listing above.

Type **sort -g -r -k5 myList**

This sorts the file, where **–g** specifies a general numeric sort, **-r** specifies a reverse order sort, i.e. the high numbers to the top, and **–k5** specifies that field 5 (column 5) is to be the key column for the sort.

Type **cat myList** (or **less myList** if you have a long file) to see that the file is actually sorted as required, i.e. in numerical order based on file size.

Of course we did not really have to use the **sort** command at all in the above example. We could simply have used the sort option in the **ls** command (ls –l –S). However, we did want to provide a simple example of how to use the **sort** command. In UNIX/Linux there is usually many different ways of achieving the same result.

2.11 Using Pipes

The use of pipes in commands allows the output from one command to become the input of another command. The | symbol denotes a pipe.

Try the following examples:

Create a file called **demo_file1** as a text file with a number of lines.

cat demo_file1 | more	The **demo_file1** text is sent to the **more** utility for screen display.
cat demo_file1 | sort | more	The **demo_file1** text is sent to the **sort** utility, where the lines are sorted and the output from **sort** is then sent to **more** for screen display.

2.11.1 The tee command with pipes

The **tee** command allows the splitting of a pipe so that a command's output can be simultaneously directed to the standard output, which is usually the next stage of the pipe, and to a file (or a named pipe).

An example usage of the **tee** command:

cat demo_file1 | sort | tee demo_file1_sorted | more

Here the **demo_file1** file is listed to the screen, using the **more** command. However, the sorted output is also sent to the **demo_file1_sorted** file using the **tee** command.

2.12 Search For Patterns With grep

In this section we will look at using regular expressions, the regular expression metacharacters, and searching for patterns. The **grep** utility will be introduced for these purposes.

Note, the GNU grep, as used in the examples, supports extended regular expressions.

grep [Global Regular Expression Print]

The **grep** utility scans text files, looking for a string match.

The general format for the command is: **grep** ***string*** ***file***

Example:

Type **cd ~** takes you to your home directory:

Type **grep the verse_1** displays the lines in **verse_1** which contain the string 'the'

The **grep** command displays all the lines which contain the matching string 'the'. Optionally we could have used quotation marks, which is better in practice, as follows:

grep 'the' verse_1

The **grep** command is **case sensitive**, it knows the differences between uppercase and lowercase characters. Try the following two commands and note the different results:

grep 'and' verse_2

grep 'And' verse_2

To ignore the **case** of letters you should use the **-i** option. Try the following command and note the result:

grep -i 'and' verse_2

Note, **grep** searches for the expression one line at a time, it does not look for the expression across line boundaries.

The **–v** option can be used to find the lines where the specified string does **not** appear in the file. Try the following example, to display all the lines that **do not** contain the string ‘the’:

grep –v ‘the’ verse_2

2.12.1 Using egrep

The **grep** command has an Extended Regular Expression (ERE) mode that can be called using grep –E, i.e. by using the –E switch the extended features are used so that the expression is evaluated as an ERE as opposed to grep’s normal pattern matching. The **egrep** command was developed to provide a command to use grep in the extended mode. Here the following two commands are equivalent:

grep –E
egrep

In many cases we would simple use egrep to be inclusive of the extended cases.

2.12.2 Line and word anchors

The line anchors

The **grep** examples above search for a simple string. A regular expression describes a pattern of characters. For example, the metacharacter caret (^) represents the beginning of a line.

e.g.: **grep -i ‘^and’ verse_2**

Here, within the quotation marks is a regular expression which says that we are searching for the string "and" (case insensitive because of the **–i** option) and this string must exist at the beginning of a line, as defined by the use of the ^ character.

As another example, type the following two commands, which will list all of the **directory files** in your current directory to the file **temp_file**:

First, make a few directory files, using the **mkdir** command, if such directory files do not already exist.

ls -l > temp_file **temp_file** now contains the directory listing

grep ‘^d’ temp_file now, all lines which start with the letter 'd' (i.e. directory files) are listed to **temp_file**.

Remember a directory file will have a ‘d’ character at the beginning of the line, such as the following example where **donDir** is a directory file.

drwxr-xr-x 2 donal myGroup 4096 Jan 16 09:02 donDir

The metacharacter (**$**) represents the **end of a line**.

For example: Type **grep ‘floor$’ verse_2** and see the result, where only lines that finish with the string ‘floor’ are outputted.

The word anchors

The \< and \> are the word anchors, where \< matches the start of a word and \> matches the end of a word. Consider the following two-line file called play.

$ cat play
In the beginning we all sat down,
then Joe said we should begin to read aloud.

Now we can check lines that can match the pattern ‘begin’. We will use a pipe in the example but we would expect the same result from both these two commands below:

$ cat play | grep 'begin'
In the beginning we all sat down,
then Joe said we should begin to read aloud.

We used a pipe in the example as this is a common way to use **grep**, but we would expect the same result from this command:

$ grep 'begin' play
In the beginning we all sat down,
then Joe said we should begin to read aloud.

Now we check lines that can match the pattern ‘begin’ that must be at the start of a word.

$ cat play | grep '\<begin'
In the beginning we all sat down,
then Joe said we should begin to read aloud.

Now we check lines that can match the pattern ‘begin’ that must be at the end of a word.

$ cat play | grep 'begin\>'
then Joe said we should begin to read aloud.

We can use the following to ensure that the pattern ‘begin’ is indeed as separate word, i.e. it is separated by spaces:

$ cat play | grep '\<begin\>'
then Joe said we should begin to read aloud.

However, if we want to find a string that is a separate word, it is easier to use the **-w**, as follows:

$ cat play | grep -w 'begin'
then Joe said we should begin to read aloud.

So, the **–w** is used to force a pattern to match only whole words. If we wanted to force a pattern to match only whole lines we would use **–x**.

2.12.3 Wildcards

In **grep** the metacharacter **‘.** ‘ represents a single wild character that matches any one character.

For example: Type **grep -i ‘.he’ verse_3**

$ grep -i '.he' verse_3
I will find out where she has gone,
And kiss her lips and take her hands;
The silver apples of the moon,
The golden apples of the sun.

Note the matches such as 'The', 'she' and ' he' will be found. Notice how the 'he' as part of 'her' is matched (as ‘space he’).

Note, to display lines containing the literal dot character, use grep’s ‘-F’ option.

To find all the lines that contain a four letter word at the beginning of a line we could use:

$ grep '^.... ' verse_2
When I had laid it on the floor
With apple blossom in her hair

Note the space in the pattern '^.... 'in the above.

To find all the lines that contain a three letter word that begind with I ot T at the beginning of a line we could use:

$ grep '^[AT].. ' verse_3
And kiss her lips and take her hands;
And walk among long dappled grass,
And pluck till time and times are done
The silver apples of the moon,
The golden apples of the sun.

Note the pattern '^[^AT].. ' would modify the above example to mean find the lines that do not start with A or T.

Both of the following statements are equivalent:

$ grep '^[AT].. ' verse_3’
$ egrep '^[AT].{2} ' verse_3 # note we used egrep here to support the quantifier.

Here the {} is a quantifier that determines how many times the previous character should occur.

The asterisk ‘*’ is used for matching multiple characters, the match is made on repetitions of a character. Say we want to match all words that start with ‘c’ and end with ‘t’, and we try the following:

$ cat verse_1 | grep 'c.*t'
And cut and peeled a hazel wand,
And moth-like stars were flickering out,
And caught a little silver trout.

Did we really want to accept the string: ‘fli**c**kering ou**t**’? Ok, we got what we asked for, but maybe not what we hoped for; we should have been more careful to specify word boundaries, so now we try the following:

$ cat verse_1 | grep -w 'c.*t'
And cut and peeled a hazel wand,
And caught a little silver trout.

In the above example note the use of ‘.*’ in **grep**. where the ‘.’ operator is followed by the ‘*’ operator. The ‘*’ operator specifies that the preceding item will be matched zero or more times, and in the example the ‘.’ is the preceding operator which specifies a match on any single character. So, any expression consisting of a character followed by a ‘*’ matches any number of repetitions of that character (and this could possibly be zero repetitions).

A summary, not a complete list in **grep**, of the regular expression operators is provided in the table below.

Operator	**Interpretation**
.	Matches any one characters
*	Matches zero or more of the previous characters
.*	Matches any number or type of characters
[ ]	Matches characters listed in the brackets
[^]	Does not match any characters that are listed in the brackets
-	Represents the range if it's not first or last in a list or the ending point of a range in a list.
^	Denotes the beginning of a line
$	Denotes the end of a line
\<	The empty string at the beginning of word is matched.
\>	The empty string at the end of word is matched.

As an example, consider the following code which will search for any word that begins with an uppercase letter in the range **B** to **D**, or a lowercase letter in the range **k** to **n**.

$ grep '\<[B-Dk-n]' verse_1
Because a fire was in my head,
And when white moths were on the wing,
And moth-like stars were flickering out,
And caught a little silver trout.

2.13 Finding Differences

2.13.1 The diff command

The **diff** command will compare two text files and display the differences.

Try the following steps:

a) Type **cp verse_1 verse_temp** makes an exact copy of **verse_1** called **verse_temp**

b) Using the editor modify the **verse_temp** file: change the word ‘berry’ to 'worm' and the word 'trout' to 'salmon'.

Now type the command **diff verse_temp verse_1**

The command output shows the lines which contain the differences, the (>) character signifies that the line is from the first file and the (<) character signifies that the line is from the second file. You will see cryptic notes on the output such as (**4c4**) and (**8c8**). These are to advise what must be done to eliminate the differences. For example (**4c4**) means that line 4 of **verse_temp** must be changed to line 4 of **verse_1**, and (**8c8**) gives similar information about line 8. There are other messages which can be outputted by **diff** (e.g. **a** for append, **d** for delete etc.) but these can be discovered later.

2.14 Count Words, Lines And Characters

2.14.1 The wc command

The **wc** command will tell you the number of lines, the number of words and the number of characters in a text file.

For example: Type **wc full_poem**

The command prints a message like:

35 172 917 full_poem

This is saying that there are 35 lines, 172 words and 917 characters in the file **full_poem**.

The **wc** command if often used to simply count the number of lines in a file, as follows:

wc -l myFile *(NB the dash elle here means count lines)*

Try this on a few sample text files.

2.15 Calendar

2.15.1 The cal command

The **cal** command displays a simple calendar. Try the following commands:

cal 7 2017

This outputs a calendar for July 2017; you might get the following result:

```
$ cal 7 2017
     July 2017
Su Mo Tu We Th Fr Sa
              1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

2.16 Time and Date

2.16.1 The date command

The **date** command can set or display the system's current time and date. Try using the **date** command as follows:

```
$ date
Thu, Jul 16, 2015  7:57:32 PM
```

In Bash, many of the commands are very flexible and contain more features and options than what might be first perceived. This is why it is always worth examining the range of options for each command. Consider the following example where we want to be very specific regarding what format we want to display for the date. The **%b** parameter represents the short name of the current month, **%d** represents the day of the month, and **%Y** is a four-digit representation of the year:

```
$  date "+%b %d, %Y"
Jul 02, 2015
```

Using the date command to measure UNIX's lapsed time

The following command will give us the current time.

```
$ date +%s
1435845758
```

This is time in the UNIX's style, where time is the number of seconds since the 'UNIX epoch', sometimes referred to as the POSIX epoch, which is the time **00:00:00 UTC** on **January 1, 1970**.

Here is an interesting example, if non-intuitive, it is a niche use for the **date** command:

Type the following:

```
$ date  +%N
538537300
```

Repeat the above a few times and look at the results, which appear to be random numbers. The **+%N** is an output option which shows the **nanosecond** portion of the current time. Since nanoseconds represent such a high time resolution, by the random use of this command we can generate informal random numbers.

2.17 Disk Storage Utilities

2.17.1 The df command

The **df** (**d**isk **f**ree) command displays information about space usage on the file systems. Use the **man** pages to help you to interpret the results and to see what options are available.

Type **df** to see information on the various mounted file systems.

Type **df .** to show information on the file system for the current directory. A response similar to the following will be seen.

```
Filesystem      1K-blocks      Used      Available    Use%   Mounted on
/dev/sd3a        58502432     12749248   45753184     22%        /
```

The report shows the file systems (i.e. disk partitions) sizes in **1kByte** blocks, with column 2 showing the full size, column 3 showing the **Used** space and column 4 showing the **Available** space. Column 5 shows the percentage space that is used.

Type **df –h .** to show the space sizes in 'human readable' form, e.g. file sizes in megabytes, gigabytes etc. The response will be similar to the following:

```
Filesystem      Size   Used   Avail   Use%   Mounted on
/dev/sd3a       56G    13G    44G     22%       /
```

2.17.2 The du command

The **du** (**d**isk **u**sage) command/utility shows the estimated amount of disk space used by specified files. The current directory is the default directory. Thus the **du** command without options will list the file sizes for files in the current directory. Disk space is normally reported in fixed disk block sizes of 1024 bytes (1k bytes) per block.

Type the **du** command and you will see a display something like the following:

```
104     ./labs/LAB1
112     ./labs
248     ./prox
16      ./.AbiSuite
16      ./.eggcups
16      ./.xemacs
8       ./testt
68      ./URLsol
```

172 ./.mozilla/firefox/4jqxey2v.default/Cache
16 ./.mozilla/firefox/4jqxey2v.default/extensions

2.18 Head And Tail

The **head** and **tail** commands are useful for picking lines off the top and bottom of files.

2.18.1 The head command

The **head** command will display a specified number of lines from the beginning of a text file.
For example type:

head -2 verse_1

This command will display the first two lines of the text file **verse_1**

2.18.2 The tail command

The **tail** command is like the **head** command, but, as you have already guessed, it works from the end of the file, instead of the top of the file.

2.19 Command History

2.19.1 The history command

The **history** command displays the commands which were previously executed by the shell.

For example, the command **history 3** will display the last three commands.

Also, experiment with the 'up arrow' key (↑) as an easy way to step back to previous commands.

2.20 The uniq Command

2.20.1 The uniq command

The **uniq** command is useful to ensure that all lines in a list of lines in a file are unique. The command will, by default, read an input file and then output that file to exclude any adjacent lines that are repeated. There are various options, as described in the **man** pages.

For example:

uniq -u text.txt > textu.txt

This command takes the file **text.txt** as input and outputs all the unique lines to the file **textu.txt**. The **–u** option specifies that only the unique lines are to be outputted.

2.21 The cut Command

The **cut** command is useful to cut out columns of data from a file. The command option arguments specify which fields or characters are to be extracted.

For example, at the command line prompt type: **ls –l > logfile**

Assume the content of the **logfile** is now as follows, as might be seen using the command **cat logFile**:

```
-rw-r--r--   1 donal     myGroup   198  Apr  3  2007    april
drwxr-xr-x   2 donal     myGroup  4096  Jan 12  11:11   archives
-rwxr-xr-x   1 donal     myGroup    65  Jul  4  2007    busy_wait.txt
drwxr-xr-x   2 donal     myGroup  4096   Jan 12  11:11  limerick
-rwxr-xr-x   1 donal     myGroup   923  Oct 19  2007    load_reduce
```

Now you want to cut out, or extract, the final column from **logfile**, by counting characters.

Type: **cut -c 51-65 logfile**

The following is now displayed:
april
archives
busy_wait.txt
limerick
load_reduce

Here, the **–c** option defined that the characters in the range between the 51th. character and the 65th. character were to be extracted.

The **–f** option can be used to extract fields (columns), and the **–d** option can be used to specify the field delimiter. The *tab* is the default field delimiter. However, in the above example the **ls** command output does not use simple delimiters (it uses multiple spaces), so using **awk** (see later) would be more useful to extract columns.

2.22 The seq Command

The **seq** command can be used to generate a sequence of numbers. Try this example:

```
$ seq 1 5
1
2
3
4
```

5

This example will print the numbers 1 to 5, line by line, to the screen, as the default separator is a new line.

2.23 Alias Substitution

Alias substitution allows you to define your own name for a command. For example, suppose you wanted to make some UNIX/Linux shell commands emulate some of the MS-DOS commands, you could define the following aliases:

alias DIR="ls -l"

alias TREE="ls -R"

alias MD="mkdir"

etc..

Now try using these MS-DOS-like commands and see how they perform.

The general format for alias substitution is:

alias name="string"

To remove an alias assignment use the **unalias** command, e.g.:

unalias DIR
etc.

2.24 Variables

2.24.1 General

We have already used some shell variables. The shell variables are normally in uppercase characters by convention. Bash lists two types of variables: global variables and local variables. The various variables can be of the following categories:

· string variables
· integer variables
· constant variables
· array variables

2.24.2 Global variables

The global variables are the environmental variables. One of the following commands can be used to display the environment variables along with their values.

- the **printenv** command
- the **env** command
- the **set** command

We will now use the **printenv** command in an example.

Type **printenv** to show the variables for the shell program which you are using. If the command displays too much information you can use the following to display one screen of information at a time:

printenv | more

The list below shows just some of the key environmental variables that might be listed:

TERM=xterm
SHELL=/bin/bash
USER=user
USERNAME=user
DESKTOP_SESSION=gnome
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
PWD=/home/user
LANG=en_US.utf8
MANDATORY_PATH=/usr/share/gconf/gnome.mandatory.path
GDM_LANG=en_US.utf8
GDMSESSION=gnome
SPEECHD_PORT=7560
HOME=/home/user

Try the following command and see what output it gives:

set | grep "USER"

You can display some individual variables as follows:

echo $PWD

echo $SHELL

2.24.3 Local variables

The local variables are available only in the current shell. The **set** command will display a list of all variables, which includes environment variables and also functions.

Variable names are case sensitive. It is common for the global variables to be capitalised by default, and local variables are given lowercase names. This is not a rule but rather it is a style convention.

2.24.4 Creating new variables

You can create new variables using the general format: **name='string'**. This is known as the variable assignment statement.

Consider the following examples:

name=Patrick
fullname='Patrick Kelly'
x=38

Note, there are no spaces on either side of the equals sign. If the value of the variable is more than one word, then use quotes.

To use the value of a variable in a command, add the $ character, e.g.:

echo $x

For example type the command: **my_name='Aengus'**

Now type the **set** command and you will see the variable **my_name** in the list.

To remove (unset) a variable use the **unset** command. For example type: **unset my_name**
and then type **set** to see that the variable **my_name** has if fact been removed.

2.24.5 Exporting a variable

A local variable is known only in the current shell. A subshell (child shell) of the current shell will not see this variable. To pass a variable to a subshell, the **export** command is used. A variable can be set (assigned a value) on exporting, e.g.:

export my_name="Wally"

A subshell has local scope on the variables that it inherits from the parent, i.e. a subshell can make changes to the variable but the parent's variable is not affected.

The command **set -a** will automatically mark all variables being created, from now on, so that they are exported.

2.24.6 The prompt variable

The prompt variable name is **PS1** and is referred to as the command prompt string.

You can define a new prompt e.g. **PS1="Limerick "**. Try this, and leave the spaces after the string 'Limerick', exactly as shown. You now have a new prompt.

Now try using the prompt to display the current working directory. The variable **PWD** defines the current working directory. Try this command:

PS1='$PWD ' (don't forget to include the space following $PWD)

2.24.7 Using quotes with variables

Single quotes are used to preserve the literal value of each character enclosed within the quotes. Double quotes are used to preserve the literal value of all characters enclosed, with the exception of the dollar sign, the backslash and the backticks.

The double quotes are used to expand a variable's name. For example, type the following commands:

```
x=5
echo $x  '$x' "$x"
```

The output is:

```
5  $x  5
```

The result shows that the variable x was expanded for the double quotes. Single quotes do not allow expansion of the variable. It is good practice to use single quotes for variables unless you need the double quotes, as the double quotes are less predictable; but they are required for variable expansion.

Note, the above **echo** command could also have been written using the escape character (\), as follows:

```
x=5
echo "$x  \$x  $x"
```

The escape character specifies that the following **$** character is not to be interpreted.

2.24.8 Further note on using quotes

The following command line will output two words to the screen:

```
$ echo  Hello world
Hello world
```

However, if you wanted to have a number of spaces between the two words, you might be tempted to use the following command:

```
$ echo  Hello          world
```

Hello world

If you try this you will notice that the spaces are not included in your output. This is because the **echo** command takes each word as a parameter, and interprets a space (or multiple spaces) to separate the parameters. Thus, in the above example, the **echo** command assumes it is simply being asked to output two separate words, so the extra spaces are ignored. If you use quotation marks (double quotes) you can specify a string as a single argument, as follows, to achieve the desired result:

$ echo "Hello world"
Hello world

Note on **IFS**: an internal field separator (abbreviated to IFS) refers to a variable which defines characters used to separate a pattern into tokens. IFS typically includes the space, tab and newline.

Some other special characters

Suppose you wanted the **echo** command to output the string, exactly as shown below, with spaces and the quotes shown in the output:

John said "go home"

You could try this command:

$ echo John said "go home"
John said "go home"

You will see that you do not get the desired output. The space will be interpreted as a parameter separator. You could try this solution:

echo "John said "go home""
John said go home

You will see that "go home" is still is not shown in quotes, as the shell still thinks that the quotes are being used to define a string. Now, here is a workable solution:

echo "John said \"go home\""
John said "go home"

However, we could have worked out a simpler solution by using single quotes, because in using single quotes preserve the literal value of each character enclosed within the quotes. Now, try this solution:

$ echo 'John said "go home"'
John said "go home"

We learned some lessons from the above. One lesson is that it is generally safer to use single quotes where we can. Another lesson here is to show the significance of the escape character (\ ). The \ character placed before a special character says that the special character is not to be treated (interpreted) as a special character, i.e. it is not to be interpreted and is to be treated literally. So **\"** simply means a double quote symbol and nothing else!

2.25 Shell Expansion

By expansion we mean that in processing a shell command the individual tokens (words) are resolved. Thus the word expansion means to resolve the tokens in this context. In Bash there are eight kinds of expansion performed. We will introduce some of the more commonly used one here.

2.25.1 Brace expansion

We can use brace expansion to allow arbitrary strings to be generated. An optional preamble is followed by a series of comma-separated strings between a pair of braces, and this can be followed by an optional postscript. Some examples are as follows:

bash$ echo b{a,o,u,e,i}d
bad bod bud bed bid

bash$ echo re{test,condition,start}ing
retesting reconditioning restarting

This is not to be confused with parameter expansion, which is of the form the string ${}.

2.25.2 The tilde prefix

Assume the tilde-prefix is replaced with the home directory associated with the specified login name, and thus the tilde has the value /home/donal. In this case we might see the following command responses:

$ echo ~
/home/donal

We expect the same result here:

$ echo $HOME
/home/donal

Command	Interpretation
~	This is the path /home/donal
~/temp	This is the path /home/donal/temp
~joe/temp	Subdirectory temp of the home directory for user joe
~+/temp	Subdirectory temp in the path $PWD/temp
~-/temp	Subdirectory temp in ${OLDPWD-'~-'}/temp
~N	The string as displayed by ‘dirs +N’ for the dir stack
~+N	The string as displayed by ‘dirs +N’ (same as above row)
~-N	The string as displayed by ‘dirs -N’ for the dir stack

2.25.3 Command substitution

Command substitution is a very useful feature of the shell. When the shell sees a command, in a command line, of the format **$(command)**, the shell will execute the command within the brackets and the command output is substituted for **$(command)**.

An alternative syntax, which is no longer encouraged, uses backticks, so the following are equivalent:

`command`
$(command)

As an example on command substitution consider the following two commands:

more $(ls)
ls | more

With the **ls | more** command the **ls** command is executed first and the output from the **ls** command is used by **more**. So we simply display a list of files.

However, the command **more $(ls)** will list to the screen the contents of each file which is listed by the **ls** command. The command **more $(ls)** is equivalent to the command **more (file1 file2 file3 ..etc...)** where all of the files in a directory are specified.

Here is another example, that uses command substitution, which counts the number of files in the home directory and puts the result in a variable called **h_count**:

h_count=$(ls –l ~ | wc –l)

The **h_count** here is a simple variable.

Try **echo $h_count** to see the result of how many files are in the home directory.

Here is one more simple example that uses command substitution:

$ echo "Do not forget this date: $(date)"
Do not forget this date: Sat, May 23, 2015 14:45:36 PM

2.26 More On Accessing A Simple Variable

The following table shows some, but not all, ways for getting the value of a variable:

Syntax	What value is accessed?
var	The actual variable i.e. var
$var	Value of var
${var}	Value of var. The braces are used to exclude from the variable name any immediately following alphanumeric characters, e.g. **${var}abc**
${var[n]}	Value of an element in an array
${#var}	The length of the value of var
${#var[*]}	The number of elements in the array var[]

Note, array variables will be introduced later in this document.

2.26.1 Special variables

The shell has some predefined special parameters (variables) which are useful in script programming. These parameters can be referenced but assignment to them is not possible. Here are some of the key special parameters:

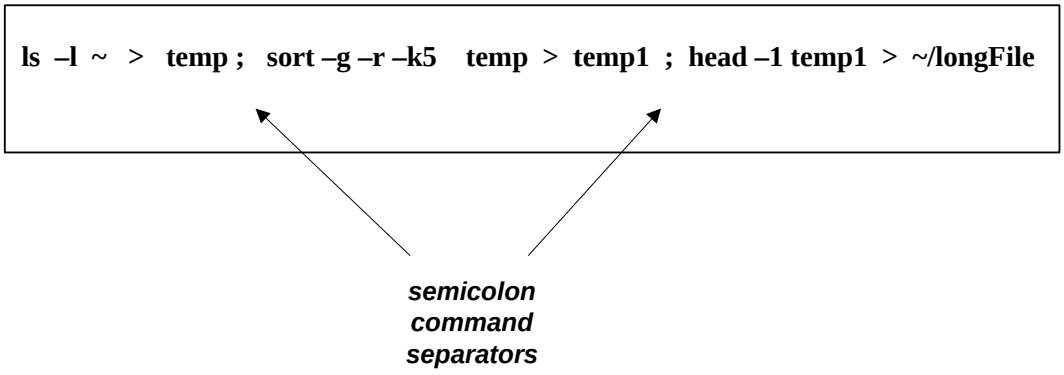
Variable	Description
$?	Exit status of the previous command
$$	Process ID for the shell process
$!	Process ID for the last background command
$0	Name of the shell or shell script
$*	Expands positional parameters starting from one
$@	Expands positional parameters starting from one
$#	The number of positional parameters in decimal
$-	The current option flags
$_	Absolute file name of the shell or script

Note – there are more than subtle differences with $* and $@. It is better to use the $@ variable in many cases.

2.27 Command Sequences

Later we will look at script files that will allow simple programs to be written using the various commands/utilities. However, it is possible to enter a series of simple commands on a single line using the semicolon to separate the commands. The commands are executed in sequence. Such a command sequence can even run beyond a single line by using the \ character (backslash) to show that the line continues to the next line.

Here is an example command sequence which will find the biggest file in your home directory, and lists the file information in a file called **longFile**, in you home directory.



To see the result of the above commands, simply type **cat ~/longFile** to display the file.

Of course, the above command sequence could be shortened by using pipes. Also, the use of pipes would eliminate the temporary files. Consider the following solution, which uses pipes:

ls -l ~ | sort –g –r –k5 | head –1 > ~/longFile

And, of course we could have used the command **ls –S** to do the sorting, but we are learning some basic steps in these small examples.

2.28 Using AWK

AWK is an interpreted programming language designed for text processing. It is common to use AWK in shell script programs, typically for data extraction and reporting. AWK is a standard feature of most Unix-like operating systems.

The name 'awk' sounds awkward, but it is derived from the names of its inventors: **A**ho, **W**einberger and **K**ernighan.

A fundamental function of AWK is to search files for text that contains one or more patterns, and on finding a match on patterns to take defined actions. Gawk is the GNU version of the UNIX awk program. Programs in AWK are 'data-driven', where the data is first described and then the action follows.

In Bash, we generally use AWK as a utility, named **awk**, where the **awk** utility scans a file, or files, and performs an action on each line that matches a specified condition. The **awk** utility is very comprehensive; in fact it is a full C-like programming language; more specifically it is a programmable text processing language. However, **awk** it is often used in simplistic ways, as a shell utility. Here we will consider only simple uses for **awk**, where **awk** statements are included in the command line, within single quotes.

However, it is also possible to run an actual **awk** program. To run **awk**, an **awk** program is specified. This can be done by putting the awk commands in a script and executing that script in the following fashion:

awk -f AWK-PROGRAM-FILE inputfile

However, as stated, in our examples we will consider short command line executions of **awk**. For such short **awk** programs, it is common to enter the program on the **awk** command line, which is best achieved by enclosing the entire program in single quotes. Double quotes will also work.

The **awk** command line has the following format where the **awk** program is contained within the single quotes and the program actions are contained within the {} braces:

awk 'condition { action }' filename

The **awk** utility reads a file one line at a time, where a line is considered as a set of fields, which are separated, by tabs (or other specified separator). Built-in variables are used where the variables $1, $2, $3, ..., $N hold the values of the fields: **$1** stands for the first field, **$2** stands for the second field and so forth. The variable **$0** (zero) holds the value of the entire line. **NF** stands for the number of fields in a line, and **NR** stands for the line number of the current line.

For example, the following list is from a student record file, which has four fields; see how the variables **$1** to **$4** relate to the fields:

ID_Number	Discipline	QCA	Home_County
141112	Computing	3.31	Limerick
142214	Engineering	3.62	Cork
145238	Science	3.03	Kerry

$1 **$2** **$3** **$4**

Assume the file is named ‘students’ and contains the following data and uses tabs for field separation:

ID_Number	Discipline	QCA	Home_County
141112	Computing	3.31	Limerick
142214	Engineering	3.62	Cork
145238	Science	3.03	Kerry
139987	Computing	3.42	Clare
132241	Engineering	3.97	Limerick
142318	Science	3.55	Tipperary
159927	Computing	3.23	Clare
138579	Science	3.40	Kerry

The awk print command

We can now try out some examples, using a**wk’s** print command to output some specific results.

The following command displays all the lines for students who are from the county of Clare:

```
$ awk   '/Clare/  { print $0 }'    students
139987        Computing       3.42    Clare
159927        Computing       3.23    Clare
```

The following command displays the ID number and QCA scores for all students who are studying Science:

```
$ awk   ' /Science/  { print $1, $3 } '   students
145238  3.03
142318  3.55
138579  3.40
```

Note, the following command would put a tab separation in the printed output to separate the columns.

```
$ awk   ' /Science/  { print $1, “\t”, $3 } '   students
145238      3.03
142318      3.55
138579      3.40
```

The following command prints the 4th column only to a file called ‘counties’, the **cat** command is added to display that output:

$ awk ' { print $4 } ' students > counties; cat counties
Home_County

Limerick
Cork
Kerry
Clare
Limerick
Tipperary
Clare
Kerry

Using regular expressions with print

A regular expression can be tested against the text of each record line, using the following syntax:

awk ‘expression { action }’ filename

Suppose we want to list the home counties for all students who have an ID number of that begins with 13 or 15 (they registered at the university in year 2013 or 2015), we could use the following code:

$ cat students | awk ' /^(13|15)/ { print $4 } '
Clare
Limerick
Clare
Kerry

A program can be preceded with a comment using the BEGIN statement as follows:

$ cat students | awk 'BEGIN { print "Selected counties:\n" } /^(13|15)/ { print $4 }'

Selected counties:

Clare
Limerick
Clare
Kerry

A program can have an ending comment using the END statement as follows:

$ cat students | awk '/^(13|15)/ { print $4 } END { print "That was my final list!\n" }'
Clare
Limerick
Clare
Kerry

That was my final list!

It is common to do some simple text processing on printing, such as changing case, as in the following example:

$ echo "Hello World!" | awk ' {print tolower ($0)} '
hello world!

The **awk** language is very powerful and we are using only selected features that enhance the functionality of Bash scripts.

The printf and awk

Note, **awk** supports the well-known **printf** command, as illustrated in this example:

$ awk ' BEGIN {printf "Pi is poorly approximated to: %f \n" , 22/7} '
Pi is poorly appoximated to: 3.142857

The implementation of the **printf** in **awk** has a rich set of features.

2.29 The sed Stream Editor

2.29.1 The sed stream editor

The **sed** (**S**tream **ED**itor) is a simple but powerful line editor utility which performs textual transformations to a sequential stream of input text data. It reads input files, line by line, and performs defined operations on specific lines, one at a time, and outputs the result. The **sed** editor is often used with piped commands. Although **sed** can be used to solve complex problems, here we will consider only the more commonly used editing features in **sed** e.g.: **p**rint, **d**elete and **s**ubstitute.

The **sed** editor uses the **vi** editor's style of commands and accepts regular expressions, e.g. it supports the metacharacters: ^, $, etc. The **sed** editor can be used to read commands from the command line or from a script. A common use for **sed** is to perform find-and-replace actions on lines based on a pattern.

Some of the more commonly used **sed** commands are listed in the table below, and users with experience of the **vi** editor will recognise the style similarity.

Command	**Interpretation**
a\	Append text below the current line.
c\	Change text in the current line with a new text.
d	Delete some text.
i\	Insert text above current line.
p	Print some text.
r	Read a file into sed from filename
s	Search and replace some text (i.e. substitution).
w	Write to a file from sed.

Some of the more commonly used options used in the **sed** editor's commands are listed in the table below.

Option	**Interpretation**
-e script	Add this script to the commands to be executed
-f script-file	Add the contents of script-file to the commands to be executed.
-n	Silent mode, suppress automatic printing.

INTERACIVE EDITING EXAMPLE

The general usage for the **sed** editor is as follows:

sed [options] commands [file-to-edit]

Let's take some simple examples for interactive editing using the **sed** editor.

We will take the verse_1 text file which has the following content, when listed using the **cat** command. Note the use of **cat –n** option to list line numbers.

$ cat -n verse_1

```
1  I went out to the hazel wood,
2  Because a fire was in my head,
3  And cut and peeled a hazel wand,
4  And hooked a berry to a thread;
5  And when white moths were on the wing,
6  And moth-like stars were flickering out,
7  I dropped the berry in a stream
8  And caught a little silver trout.
```

Now we will attempt use **sed** because we want only to see the lines that begin with the string ‘And’. Note the need for single quotes to enclose the **sed** commands. The following is the ouput – but it it is not what we really wanted:

```
$ sed  '/^And/p'  verse_1
I went out to the hazel wood,
Because a fire was in my head,
And cut and peeled a hazel wand,
And cut and peeled a hazel wand,
And hooked a berry to a thread;
And hooked a berry to a thread;
And when white moths were on the wing,
And when white moths were on the wing,
And moth-like stars were flickering out,
And moth-like stars were flickering out,
I dropped the berry in a stream
And caught a little silver trout.
And caught a little silver trout.
```

The above example does not seem to have done what we expected. All of the file’s lines are displayed, and the lines beginning with ’And’ are displayed twice. By using the **–n** option we can suppress some printing so that we get just the lines that we require, as in the following example:

```
$ sed  -n  '/^And/p'  verse_1
And cut and peeled a hazel wand,
And hooked a berry to a thread;
And when white moths were on the wing,
And moth-like stars were flickering out,
And caught a little silver trout.
```

In the following example we will display an output that deletes all of the lines that begin with ‘And’. Note, we do not actually delete lines from the verse_1 file; rather we simply display the file without the specified lines as follows:

```
$ sed  '/^And/d'  verse_1
I went out to the hazel wood,
Because a fire was in my head,
I dropped the berry in a stream
```

In the following example the lines from 5 to 8 are deleted:

```
$ sed  '5, 8d'  verse_1
I went out to the hazel wood,
```

Because a fire was in my head,
And cut and peeled a hazel wand,
And hooked a berry to a thread;

In the following example the lines from 5 to 8 are printed.

$ sed -n '5, 8p' verse_1
And when white moths were on the wing,
And moth-like stars were flickering out,
I dropped the berry in a stream
And caught a little silver trout.

The above example could have been equivalently written as follows, specifying a starting line and three following lines:

$ sed -n '5, +3p' verse_1

In the following example we replace the string ‘hazel’ with ‘cedar’.

$ sed 's/hazel/cedar/' verse_1
I went out to the cedar wood,
Because a fire was in my head,
And cut and peeled a cedar wand,
And hooked a berry to a thread;
And when white moths were on the wing,
And moth-like stars were flickering out,
I dropped the berry in a stream
And caught a little silver trout.

Next, we hope to replace all the occurrences of the indefinite article ‘a’ with the definite article ‘the’.

$ sed 's/a/the/' verse_1
I went out to the hthezel wood,
Bectheuse a fire was in my head,
And cut thend peeled a hazel wand,
And hooked the berry to a thread;
And when white moths were on the wing,
And moth-like stthers were flickering out,
I dropped the berry in the stream
And ctheught a little silver trout.

Now there are two problems with the above result against what we had hoped for. The first problem is that the ‘a’ and the ‘the’ are simple strings, but they should have been specified as being separate actual words. The next time we will use the \<\> meta-characters to anchor the words. The second issue is that we want to check for multiple occurrences of ‘a’ and ‘the’ on a line so will use the ‘**g**’ flag on the substitute command, placing it after the substitution set, to advise **sed** to check the whole line. See the following example.

$ sed 's/\<a\>/the/g' verse_1
I went out to the hazel wood,
Because the fire was in my head,
And cut and peeled the hazel wand,

And hooked the berry to the thread;
And when white moths were on the wing,
And moth-like stars were flickering out,
I dropped the berry in the stream
And caught the little silver trout.

If for the above we had only wanted to change the second instance of ‘a’ that **sed** finds on each line, then we could have use the number ‘2’ instead of the ‘g’.

Supposing we wanted to substitute ‘a’ with ‘the’, but also substitute ‘hazel’ with ‘cedar’; then consider the following example where multiple commands can be separated using the **–e** option.

$ sed -e 's/\<a\>/the/g' -e 's/hazel/cedar/' verse_1
I went out to the cedar wood,
Because the fire was in my head,
And cut and peeled the cedar wand,
And hooked the berry to the thread;
And when white moths were on the wing,
And moth-like stars were flickering out,
I dropped the berry in the stream
And caught the little silver trout.

Here are some simple examples showing the use of **sed**:

sed ‘/Rocky/d ‘ testFile	Deletes all lines in file **testFile** that contain the string “Rocky”
sed ‘/^$/d ‘ testFile	Deletes any **blank lines** for text file **testFile**
sed –n ‘ /Simba/p’ testFile	Outputs only lines in **testFile** that contain the string “Simba”
sed ‘s/Simba/Rocky/‘ testFile	Substitutes the string “Rocky” for “Simba” for **testFile**
sed ‘s/%/ /’ testFile	Substitute the % character with a blank space for **testFile**

Example question:
Assume that some variable **x** has the value **25%**. Using **sed** write a command to eliminate the **%** character so that the variable can be used as an integer for arithmetic operations.

x=$(echo $x | sed ‘ s/%/ / ‘)

The variable **x** now represents an integer value (i.e. **25**).

Another example question:
Assume that some variable **x** has a value of **99.9**. Since the value is reported as a single decimal place number, assume that you want to round this down to the integer, **99**, so that you can use integer arithmetic on the variable.

Assume the variable **$x** represents a floating point decimal number with one decimal place.

x=$(echo $x | sed ‘ s/\..$/ / ‘)

Here with the **sed** command, note how the metacharacters are used. The **\..$** code says that any character at the end of a line (*dot for wild character, and $ for end of line*), that follows a stop character (**\.** *means the stop character is escaped, i.e. it is interpreted literally*) is to be substituted. The **/ /** code says that substitution value is to be a space, i.e. the / / represents a space. We could have used // to present no character at all.

2.29.2 Simple edits using cut and tr

The **sed** examples above are simplistic so as to introduce **sed**. However, such simple edits to remove characters etc. can be easily achieved using the **cut** command as introduced earlier. The **tr** (translate) utility is also useful. The **tr** utility takes the input and produces an output with substitution or deletion of selected characters. The **tr** command can take as parameters two sets of characters, and replaces occurrences of the characters in the first set with the corresponding characters from the second set, thus translating characters.

Here’s a simple example to use **tr** to remove the **%** character as in the earlier example:
Assume **x** has the value **25%**. The **–d** option means delete, so this command removes the **%** character:

x=$(echo $x | tr –d “%”)

2.29.3 The bash string replacement feature

Later on in this document we will give a comprehensive introduction to the topic of parameter expansion (PE) but for now we will look at a single feature of PE that concerns the replacing of a matched pattern with a defined string. In this way Bash can replace a string with another string as shown in the example below, where we will declare a variable called **name** and give it a string value:

$ name="ABCfoods Inc"
$ echo $name
ABCfoods Inc

Now we will declare a new variable **name1** that has a value that is modified from the **name** variable where the string ‘food’ is substituted with ‘drink’:

$ name1=${name/food/drink}
$ echo $name1
ABCdrinks Inc

Now let’s show an example that converts the string **25%** to **25** by substituting the % character with a space. Note we escape the % as follows \%, this is necessary because the % symbol has a special meaning in parameter expansion, signifying the end of a line.

$ var=25%
$ echo $var
25%
$ var=${var/\%/ }

$ echo $var
25

Here is an example that converts the string **99.9** to **99** by substituting the dot and any further trailing characters with a space:

$ var=99.9
$ echo $var
99.9

$ var=${var/.*/ }
$ echo $var
99

The above examples used the following feature of the topic of parameter expansion (PE):

${parameter/pattern/string}	The value of 'parameter' (expanded) that first matches 'pattern' is replaced by 'string'.

2.29.4 Process Substitution

As we saw earlier command substitution can be used to set a variable to the result of a command, as in this example where the variable ‘count’ is set to the result of the command code that is contained within the $(…) enclosure:

count=$(grep "And" verse_1 | wc -l)

Now we will look at another concept, which is that of process substitution. With process substitution the output of a process is fed into another process; that is to say the results of a command is fed into another command. We can think about a simple pipe as being a basic form of process substitution as in this example, where the output of the **grep** command is fed into the **sort** command:

grep "And" verse_1 | sort

However, Bash supports a process substitution feature that is much more powerful than using simple pipes. Here is the above example again, this time using process substitution:

sort <(grep "And" verse_1)

Note the use of the syntax for process substitution:

<(command)

The results of the process within parentheses are sent to another process. There are no spaces between the parentheses and the "<" symbol, as a space would imply redirection from a subshell, and not process substitution. Process substitution also supports the **>(command)** form to feed process output in the other direction in the command sequence.

Process substitution is more powerful than using simple pipes. For example consider the following example where we want to sort some lines from the outputs of multiple commands:

sort <(grep "And" verse_1) <(awk ' /the/ {print} ' verse_2) <(cat verse_3)

Here the output from three processes is fed into the sort command.

3 Shell Arithmetic

The Bash shell's built-in arithmetic features are based on integer mathematics only. Floating point arithmetic can be achieved using external programs as will be briefly introduced.

3.1 Programming Simple Arithmetic

1.1.1 The let command: how to assign an arithmetic value to a variable

We will look at the well-known UNIX shell's **let** command for doing simple arithmetic operators. However, be aware that the **let** command is now quite antiquitated and later we will see how the shell provides a more convenient way of assigning arithmetic values.

Consider the following examples:

Example 1:
```
x=10
y=$x+5
echo $y
```

Here, the **echo** command outputs the value of y to the screen as follows:

```
$ echo $y
10+5
```

Clearly, no arithmetic operation has been performed.

However, the **let** command allows us to perform simple arithmetic operations. The **let** command will assign an actual arithmetic value to a variable, as follows:

Example 2:
```
x=10
let y=$x+5
echo $y
```

This time the **echo $y** command outputs the expected answer, **15**, as follows:

```
$ echo $y
15
```

The **let** command works on simple integer arithmetic only and includes the following operators:

-x	negate x
x*y	multiply
x/y	divide
x%y	remainder
x+y	addition
a-y	subtraction

1.1.2 Arithmetic ((..)) syntax

An more intuitive syntax, which should be used in stead of the **let** command, is to wrap the arithmetic statement in double parenthesis. This form is a more familiar style for programmers and is more forgiving about the use of spaces. Variables inside (( )) parenthesis do not require $ symbol (as string literals are not supported).

The evaluation of the arithmetic expressions is based on fixed-width integers without overflow checking, however, the divide-by-zero is error is recognised.

The operators (not a complete list) are similar to that of the C programming language as listed below in order of precedence:

VAR++ and VAR--	post-increment and post-decrement
++VAR and --VAR	pre-increment and pre-decrement
- and +	unary minus and plus
! and ~	logical and bitwise negation
**	Exponentiation
*, / and %	multiplication, division, remainder
+ and -	addition, subtraction
<< and >>	left and right bitwise shifts
<=, >=, < and >	comparison operators
== and !=	equality and inequality
&	bitwise AND
^	bitwise exclusive OR
|	bitwise OR
&&	logical AND
||	logical OR
expr ? expr : expr	conditional evaluation
=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^= and |=	Assignments
,	separator between expressions

Here is an example using the arithmetic ((..)) syntax

```
x = 77
(( y = 3 * x ))  #  $x can be represented as simply x
echo $y
```

Of course, y will have a value of 231following the above.

Note, spaces can be omitted if preferred as follows:

```
((y=3*$x))  # spaces can be left out if preferred
```

The following three operations are equivalent:

((x=$x+9))

((x = x + 9))

((x += 9))

Here is an example command to make a random number:

((x = RANDOM % 10 + 1)) # a random number from 1 to 10

3.1.1 Arithmetic expansion

In arithmetic expansion the arithmetic expression is evaluated and the result is substituted. The format for arithmetic expansion is:

$((arithmetic expression))

The arithmetic expression is evaluated to expand to the result. The result will be a one word digit in Bash.

There is an alternative syntax for arithmetic expansion in Bash as follows:

$[arithmetic expression]

Do not use (**DO NOT USE!**) this style as it is now now deprecated, giving preference to the $((...)) form.

At the command prompt, try the following simple examples:

x=2; y=3; echo $((x + y))

var=$((x * 3)); echo $var

var=$((++x)); echo $var

A constant with a leading 0 (zero) is interpreted as an octal number, a leading "0x" or "0X" indicates hexadecimal. Otherwise, numbers take the form [BASE#]N, where BASE represents the arithmetic base and N is the number in that base, as in the following examples:

$ echo $((10#34))
34

$ echo $((8#34))
28

$ echo $((16#39))
57

$ echo $((16#3A))
58

$ echo $((13#34))
43

$ echo $((2#10101110))
174

By default the base 10 is assumed.

As seen in some of the examples, variables in arithmetic expansion can be used with or without variable expansion, for example:

```
x=7
echo $((x))       #  Normal use. Good.
echo $(($x))      # Works fine, but not recommended, better to use variables directly.
```

3.1.2 Arithmetic operation using 'expr'

The Bash shell also supports the traditional UNIX shell **expr** command. It is not recommended (NOT RECOMMENDED!) **to use expr**. The original Bourne shell does not support arithmetic expansions, but arithmetic expansion is required by POSIX, so use the $(…) style and not **expr** and the backticks (not single quotes), as follows:

sum=` expr $x + $y `

The **expr** command means evaluate **expressions**

The **expr** arguments are treated as expressions. Integer arithmetic is used. The result is directed to the standard output. The format of the **expr** command is very exacting, requiring spaces between expressions and the operator. The asterisk is used to indicate multiplication, but it needs to be escaped. Imagine the multiplication operator as * rather than *. The multiplication operator could also be written as "*".

The operators (not a complete list) are listed below in order of precedence:

expr +	Addition
expr -	subtraction
expr *	multiplication
expr /	division
expr %	remainder

Examples:

At the command prompt try the commands below and check that you get the expected answers.

a) **expr 1 + 2**

b) **expr 8 * 7**

c)
```
num1=10
num2=20
sum=$( expr  $num1 + $num2 )

echo $sum
```

Note the third line, line (c), could have been written as follows based on the deprecated style for command substitution:

sum=` expr  $num1 + $num2 `

All of the following styles give logically equivalent results:

```
sum=$( expr $num1 +  $num2 )
sum=$((  $num1 + $num2 ))
sum=$(( num1 +  num2 ))
sum=`expr  $num1 + $num2`
```

3.1.3 Floating point arithmetic

Bash does not natively support floating point arithmetic. There are command line tools that can be used to perform floating point calculations. Probably the best-known tool (utility) is called **bc**, which is defined as an arbitrary precision calculator language. The **bc** has four special variables: scale, ibase, obase, and last. The default base for input and output is 10. The variable ‘scale’ is use to set the precision for results. We will look at some easy examples to show some features of **bc** with simple arithmetic.

Type **bc** on the command line and you will enter the **bc**. Type ‘quit’ to exit.

```
$ bc
bc 1.06.95
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
```

You are now in the bc utility. Type the following example where we set the scale value to 3 and enter a small equation. We see the answer is 1.618 to three decimal places as defined by the scale value:

```
scale = 3
( 1 + sqrt(5) ) / 2
1.618
```

Next we see how to pipe our expression directly into the **bc** utility:

$ echo "scale=3; (1 + sqrt(5))/2" | bc
1.618

For the above we could have used a heredoc (heredocs will be described later on) as follows:

$ bc <<< "scale=3; (1 + sqrt(5))/2"
1.618

Using awk for arithmetic

Floating point arithmetic can also be achieved using **awk**. Here are small examples:

$ awk 'BEGIN { print 1000/9 }'
111.111

$ awk 'BEGIN { print 2 ^ 8 }'
256

The **printf** statement can be used to format the output including the precision of results:

$ awk "BEGIN { printf \"%.2f\n\", (1 + sqrt(5))/2 }"
1.62

Here we calculate the value of phi and assign it to a variable:

$ phi= $(awk "BEGIN { printf \"%.2f\n\", (1 + sqrt(5))/2 }")
$ echo phi
1.62

As we can now imagine, the full power of **awk's** arithmetic operations is available to the shell programmer.

4 Shell Programming: A Short Primer!

4.1 General

By now we have looked at enough shell commands to be able to write some small programs at the shell level. A program written in the shell is known as a shell script program.

4.2 A Simple Shell Script Program

We will now create a very simple shell script program. The example program is kept deliberately simple so that we can concentrate on the steps involved in editing and executing the program.

Using the editor (any plain text editor), create a file called **easy_text** with this content:

easy_text

```
#! /bin/bash

echo    "Hello world!"
echo    "The University of Limerick is situated"
echo    "down by the banks of the Shannon river,"
echo    "about three miles from Limerick city."
```

The **easy_text** file is a simple shell script file. It contains four **echo** commands. Now we need to make the file executable. The first line of all the script examples contains **#! /bin/bash** to specify to the system that the **/bin/bash** file is to be used to execute the script file. In other words, we need to tell the system to use the Bash shell for this script program.

To run this shell script program, type:

bash easy_text

This command will execute the **easy_text** program. However, the **bash** command in the command line will start a new copy of the shell program. The new copy of the shell program executes the **easy_text** file and then the shell quits passing control back to the calling shell. A more common way of running the script program is to make the shell script file directly executable.

4.2.1 Make the shell script program executable

Now that you have created the file **easy_text** you can make this file executable. List the file using the following command:

ls -l easy_text

You will get a report something like the following:

-rw-rw-r-- 1 donal myGroup 6 Jan 12 11:52 easy_text

You will see from this line that the file is not executable (i.e. no **x** in the permission field). To make the file executable you could enter the following commands:

chmod u+x easy_text

The **ls -l easy_text** command will now list the file, something like as follows, showing that the file is now executable:

-rwxrw-r-- 1 donal donal 6 Jan 12 11:52 easy_text

The file is now executable by the user as we can see by inspecting the above line. To execute the script file **easy_text** you can simply type the file name. Usually you will need to specify its path (e.g. **./**) here also, as shown:

./easy_text

4.2.2 Using Comments

Just like any programming language you can include comments in your script files to make the program more readable. The hash (#) symbol is used to indicate a comment. Add some comments to the **easy_text** script program and run the program again. You will see that the comments do not affect the operation of the program. Please note that the **#! /bin/bash** line is not a comment, since the #! character pair forms a special combination to define which shell is to be used to execute the script.

Example comments in **easy_text**:

easy_text

```
#! /bin/bash
# This is a simple script file
echo "Hello world!"
echo "The University of Limerick is situated"        # the premier university!
echo "down by the banks of the Shannon river,"       # longest river in Ireland
echo "about three miles from Limerick city."
```

4.2.3 Getting Keyboard Input (read command)

Script file programs often require input from the user. The **read** command has the format:

read [options] NAME1 NAME2 ... NAMEN

When the shell executes a **read** command it waits for the user input and assigns the input to the variable, e.g.:

read my_name *(user must hit 'enter', input something and hit 'enter')*

echo “Your name is: $my_name”

There are many options for the read command but the **–r** option is important and it is recommended by some experts to always include this option. The **–r** option allows for raw input i.e. it disables interpretation of backslash escapes and line-continuation in the read data.

The read command can accept multiple words as input. Try the following example.

```
#!/bin/bash
echo "What are your favourite colours?"
read  -r  col1  col2  col3
echo  "You like: $col1"
echo  "You also like: $col2"
echo  "And you say you like $col3 also!"
```

You can expect a dialogue something like as follows:

```
$ bash example
What are your favourite colours?
green white orange
You like: green
You also like: white
And you say you like orange also!
```

In the above example we could have used the **–p** option for the **read** command to issue a prompt to the terminal as follows and thus there is no need to **echo** the question:

```
$ read -p  "What are you favourite colours? " col1 col2 col3
What are you favourite colours?
```

4.3 Parameter Passing

Arguments can be passed to a shell script program using positional parameters. Positional parameters are the arguments given to a script when it is invoked. These arguments are represented by the variables from $1 to $N. Note, when N is a number of more than a single digit, it must be enclosed in braces, such as ${N}. The variable $0 is the basename of the script program itself.

For example, suppose you want to write a utility called **difference** which will take two specified input files, find the differences between the two files, and output the difference information to a specified output file.

Here is a shell script file program which implements our **difference** utility.

```
#! /bin/bash
# This script is named ‘difference’ compares $1 file to $2 file and outputs to  $3 file

diff  $1   $2  >  $3
echo  "DONE - the $3 file contains the difference information"
```

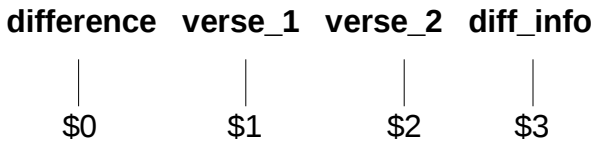
Use the editor to create this file, named **difference**, and use the **chmod** utility to make the file executable.

Now use the **difference** utility, which you have just written. In the command line you must specify the two input files and the output file. Here is an example that calls the **difference** utility, try it out:

./difference verse_1 verse_2 diff_info

The input file names, **verse_1** and **verse_2,** are substituted in the **difference** utility for the formal parameters **$1** and **$2** respectively. The file name **diff_info** is substituted for the **$3** parameter.

The formal parameters of the program ($1, $2 etc.) are substituted by the actual parameters based on the position of the actual parameters in the argument list. The special positional parameter **$0** stands for the file name of the shell script file itself. The positional parameters for the example **difference** utility can be shown as follows:



The shift command

The **shift** command will 'shift' the positional parameters by 1 place by default, or by **n** places if specified. Thus the number and the position of the positional parameters are changed. Consider the following example script program:

```
#!/bin/bash

# Demonstration of shell parameters and 'shift'
# Run this program with 6 arguments.

echo $1, $2, $3, $4, $5, $6    # display the six arguments
shift 1
echo $1, $2, $3, $4, $5, $6
shift 2
echo $1, $2, $3, $4, $5, $6

exit
```

The above program will run as follows:

```
$  .\example  a b c d e f
a, b, c, d, e, f
b, c, d, e, f,
d, e, f, , ,
```

It can be seen from the output that the parameters can be shifted in their position.

4.4 Conditional Expressions

4.4.1 Comparisons

Variables and values can be compared and program branch decisions can be made based on the results of the comparisons. Some arithmetic comparisons can include the following relational operators:

<=	less than or equal to
>=	greater than or equal to
<	less than
>	greater than
= =	equal
! =	not equal
&&	logical AND
||	logical OR

4.4.1 The if-statement and conditional blocks

A test will determine whether something is true or false. The **test** command is very useful in program control constructs, such as the **if** statements.

A Bash command on termination results in an exit code, which represents a return value that is an integer between 0 and 255. A 0 (zero) exit code denotes success, and a positive number denotes some failure. The actual exit code number is application-specific.

There are commands that are intended to *test* some condition or conditions and to return an appropriate exit status. The **test** command is designed to do just this and can be used as follows to test the truth of some expression and will exit with the status determined by EXPRESSION:

test EXPRESSION

The command can also be called using the following equivalent syntax, which is more commonly used:

[EXPRESSION]

We will look at simple examples of the **test** command to check if a file called big_file exists in the home directory. The **–e** means to test if the file exists.

test –e ~/big_file

or

[-e ~/big_file]

Both of the above commands have the same meaning, Note the **necessary requirement** to always use spaces following the opening [bracket and preceding the closing] bracket.

NOTE ON STYLE

The **[-e ~/big_file]** format is preferred to the **test –e ~/big_file**
However, it far better to use the **[[-e ~/big_file]]** format which will be presented soon in the text.

We will look at some practical examples using such test operations by studying the **if** statement, which will execute a command (or commands) and check the exit code for success or otherwise. Here is the general outline form of the **if** statement:

```
if TEST-COMMANDS
then CONSEQUENT-COMMANDS
fi
```

Many people prefer a style where the ‘then’ keyword is written on the same line as the ‘if’ and the semicolon is required as follows:

```
if TEST_COMMANDS; then
   CONSEQUENT-COMMANDS
fi
```

Note that the **then** and **fi** are considered to be separate statements and that is why they are separated by a semicolon when wriiten on the same line.

Here is a simple example using an **if** statement that includes the **if-then-else** construct:

```
if true;  then
  echo "true!"
else
  echo "false!"
fi
```

Note the use of ‘true’ in the first line of the example which is a built-in command that always evaluates to true.

The following code checks if the file **big_file** exists in the user’s home directory. Here is a sample program using this test and illustrates the **if-then-else** construction:

```
if  [ -e ~/big_file ];  then
   echo “Yes, big_file is there”
else
  echo “No, big_file is not there”
fi
```

Here is a simple example that includes the **elif** (else if) construct:

```
my_status=Clown
```

```
if [ $my_status = "student" ]; then
  echo "You have so much to learn!"
elif [ $my_status = "teacher" ]; then
  echo "You must be kind to students!"
else
  echo "Well Mr. $my_status - you probably think you know it all by now!"
fi
```

4.5 Expressions With Logical Operators

It is important to note that we do not always need the **if** statement to test conditions. Actions can be performed based on the success of a previous command by using control operators, which are && and ||, that respectively represent a logical AND and a logical OR. This gives us the concept that is known as conditional execution. Here is an example command:

```
$  [ $name = "Ann" ]  &&  echo "Her name is $name"
```

This will check the result of the test, and if the result is true (exit code 0), then Bash will execute the echo command and return a true (exit code 0) on the successful echo. If the test had failed Bash would skip the echo command and return a non-zero exit code. Try this out on the command as follows and see there is an echo output because the test is true:

```
$ name=”Ann”
$  [ $name = "Ann" ]  &&  echo "Her name is $name"
Her name is Ann
```

Now try this and the test is false so there is no echo:

```
$ name=”Claire”
$  [ $name = "Ann" ]  &&  echo "Her name is $name"
```

So, we now know that above conditional execution example is equivalent to the **if** statement as follows:

```
if  [  $name = "Ann"  ]; then echo "Her name is $name"; fi
```

Now we will look at a simple example of conditional execution that uses the OR logic:

```
mkdir ~/accounts  ||  echo "Could not make that directory"
```

If the directory **~/accounts** does not already exist, this command will make a directory called accounts in the user’s home directory and the mkdir command will return true (exit code 0). No further command will be executed, i.e. the echo command will not be executed. However, if the directory **~/accounts** does already exist then the mkdir returns false and the echo command is executed.

The [[command

We have seen that the **test** command can have an alternative syntax called ‘[‘ command, which is based on the square brackets []. This syntax is generally preferred over the ‘test’ syntax. However, there is a

more versatile version that is called the ‘[[‘ command, which used doubly squared bracket syntax. Thus, the following three commands, which each compares two strings, are functionally equivalent:

if [[$string_x = $string_y]]; then echo "They are equal!"; fi

if test $string_x = $string_y; then echo "They are equal!"; fi

if [$string_x = $string_y]; then echo "They are equal!"; fi

However, the third line uses the ’ [[‘ syntax and this form supports more advanced features which are not evident from this simple example. It is recommended to always use the [[..]] style.

The ‘[[‘ command was introduced in the Korn shell but because of its added features it was adopted by the Bash shell developers. Today, the use of the [[syntax is preferred as the default command for testing.

An example feature of the ‘[[‘ command that does not exist in the ‘[‘ command is pattern matching. Here is a test that uses pattern matching:

if [[$filename = *.html]]; then echo "File name suggests a HTML file!"; fi

A list of some of specific tests that are supported by Bash are listed in the table below.

Specific test	Condition for TRUE
-e FILE	file exists
-f FILE	file is a regular file
-d FILE	file is a directory
-h FILE	file is a symbolic link
-p PIPE	pipe exists
-r FILE	file is readable by you
-s FILE	file exists and is not empty
-t FD	FD is opened on a terminal
-w FILE	file is writable by you
-x FILE	file is executable by you
-O FILE	file is effectively owned by you
-G FILE	file is effectively owned by your group
FILE -nt FILE	first file is newer than the second
FILE -ot FILE	first file is older than the second
-z STRING	string is empty (length is zero)
-n STRING	string is not empty (length is not zero)
STRING = STRING	the first string is identical to the second
STRING != STRING	first string is not identical to the second
STRING < STRING	first string sorts before the second
STRING > STRING	first string sorts after the second
EXP -a EXP	both expressions are true (AND)
EXP -o EXP	either expression is true (OR)
! EXP	Inverts the result of the expression (NOT)
INT -eq INT	both integers are identical
INT -ne INT	integers are not identical
INT -lt INT	first integer is less than the second
INT -gt INT	first integer is greater than the second
INT -le INT	first integer is less than or equal to the second
INT -ge INT	first integer is greater than or equal to the second

In addition to the list of specific tests listed in the table above, some additional tests are supported by the [[command as follows; this is not an exclusive list:

EXP && EXP:
This is similar to the '-a' operator (AND) in the **test** command but the second expression is not evaluated if the first one results as false.

EXP || EXP:
This is similar to the '-o' operator (OR) in the **test** command but the second expression is not evaluated if the first one results as true.

STRING = PATTERN:
Here *pattern matching* is supported so this is not a simple string comparison as in the **test** command. The test returns true there is a glob pattern match. Note STRING == PATTERN is valid syntax also.

STRING =~ REGEX:
This results as true if there is a string match on the regex pattern.

Testing for command exit status using 'if'

Note, **$?** is the built-in shell variable for the exit status of whatever last command was used.

The following example will echo a 1 if the file ~/hobbies does not exist, it will echo a 0 if the file does exist:

$ [[-e ~/hobbies]] ; echo $?

Sometimes it is useful to check the exit status using an **if** statement as follows:

$ if [[$? -ne 0]] ; then echo "output of previous command is false" ; fi

Numeric comparisons

Numeric comparisons are also supported in the testing using the follow operators to act on integer operands as listed in the table above: -eq, -ne, -lt, -gt, -le and –ge. Here is a small code example:

```
echo "Input your age"
read  your_age
if [[ $your_age -ge 18 ]] ; then
  echo "You can vote!"
fi
```

A little later on, we will learn that arithmetic tests can be made using the double parentheses construct ((…)) to encapsulate the condition, allowing us to use this more intuitive syntax for tests, and this will be the recommended style for arithmetic conditional testing:

if (($your_age >= 18)); then echo "You can vote!"; fi

4.5.1 The case statement

The **case** statement construct is of the general form that is common with many popular computer programming languages.

Let’s assume we want to write a program where the logic branch will depend on the content of a variable. We could use the **if** statement to do this. Here is an example that looks for a result on testing against a glob. The user is asked to type in three or more characters to identify a particular academic subject, and the intended subject is then selected by the program.

```
#! /bin/bash
echo "Select subject: type at least the three first characters for subject in lower case"
read subj

if [[ $subj = mat* ]]; then
```

```
   echo 'You selected: Mathematics!'
elif [[ $subj = sci* ]]; then
   echo 'You selected: Science!'
elif [[ $subj = eng* ]]; then
   echo 'You selected: Engineering!'
elif [[ $subj = lan* ]]; then
   echo 'You selected: Languages!'
elif [[ $subj = his* ]]; then
   echo 'You selected: History!'
elif [[ $subj = geo* ]]; then
   echo 'You selected: Geography!'
else
   echo 'Sorry - your subject is not in our curriculum.'
fi

exit 0
```

The above code makes some redundant comparisons. It would be neater to use the **case** statement. In a **case** statement each choice is a pattern followed by a right ‘)’ bracket, then a code block is executed if there is a match. Note, two semicolons are used to end the code block (so as not to confuse this with a statement separator). Each case plus its associated commands is called a clause. The matching is stopped when one case is successfully compared. The * pattern can be used at the end to match any case that has not been matched earlier. Each **case** statement is ended with the **esac** statement. Here is the above example again but this time it is written using a **case** statement.

```
#! /bin/bash
echo "Select subject: type at least three first characters for subject in lower case"
read subj

case $subj in
  mat*) echo 'You selected: Mathematics!' ;;
  sci*) echo 'You selected: Science!' ;;
  eng*) echo 'You selected: Engineering!' ;;
  lan*) echo 'You selected: Languages!' ;;
  his*) echo 'You selected: History!' ;;
  geo*) echo 'You selected: Geography!' ;;
  *) echo 'Sorry - your subject is not in our curriculum.' ;;
esac

exit 0
```

I a **case** statement we can use the ‘|’ symbol to separate multiple patterns. We could have used this clause in our example above:

```
  mat* | sci*) echo 'You selected: Mathematics with Science.' ;;
```

4.5.2 The select construct

The **select** construct is useful for generating menus. The equivalent of **select** is not found in many conventional programming languages. The syntax for **select** is similar to the **for** construct, as follows:

```
select name in LIST
do
        commands that use $name
done
```

The **select** construct will create a menu containing each item in a list, where each entry is automatically numbered. The s**elect** statement will prompt for an input number. The commands in the body of **select** are executed continuously in a loop, until some break condition exists. Try the following example:

```
#! /bin/bash

PS3="Which of the main Japanese islands do you want to visit? "
select choice in Honshu Hokkaido Kyushu Shikoku ; do

  if [[ $choice =  Honshu ]] ; then
    echo "Honshu is the largest island and the most populated." ; break; fi
  if [[ $choice = Hokkaido ]] ; then
    echo "Hokkaido is the second largest island and it is the coldest." ; break; fi
  if [[ $choice = Kyushu ]] ; then
     echo "Kyushu is the third largest island and it is to the south."; break; fi
  if [[ $choice = Shikoku ]] ; then
    echo "Shikoku is the smallest of the islands; Matsuyama is its capital."; break; fi

 echo "Not a valid option - choose again!"

done

exit 0
```

On running the above script program a menu will be displayed as follows; assume the name of the script is **menu_japan**:

```
$  ./menu_japan
1) Honshu
2) Hokkaido
3) Kyushu
4) Shikoku
Which of the main Japanese islands do you want to visit?
```

4.5.3 Arithmetic conditionals

Arithmetic tests can be made using the double parentheses construct ((…)) to encapsulate the condition. The common relational operators, such as that found in the C language, are supported. For example the following expression returns an exit status of 0, i.e. the expression is true:

```
((  (5 > 2)  &&  (6 <= 9)  ))
```

So, in an **if** statement we can use this example:

$ if (((x == y) && (x == z))); then echo "true"; fi

However for a string comparison we could not use the ((…)) construct, we would use:

if [["$x" == "$y" && "$a" == "$z"]]

4.6 Conditional Loops

Now we will learn how to write scripts that use control loops for repetitive tasks using **for**, **while** and **until** loops, where the **until** loop is a variant of the **while** loop. Previously we learned about making basic decisions in scripts, but very often we need to repeat operations and we need to use a loop for that. The **for** loop is used to go through a list of items sequentially. The **while** loop is used where it is not known in advance how many times a loop needs to be repeated.
.

4.6.1 The for loop

The general format of the **for** construct is:

```
for  var  in  list; do
  commands
done
```

The loop is repeated for each item in **list**, setting the loop index variable **var** to each item in turn. Here **list** can be any list of items: words, numbers, or strings. The list can be literal or generated by some command. Note the following syntax is also valid, where the semicolon is not used as the **do** is started on a new line.

```
for  var  in  list
do
  commands
done
```

The **for** loop begins by assigning the first item in the **list** to the variable **var** and it then executes the **command** or list of **commands**. Then the next item in the **list** is assigned to the variable **var** and the command, or list of commands, is executed again. The process continues until each item in the **list** has been processed. Here is an example to display a line three times:

```
for i in {1..3};  do
echo "This is $i iteration!"
done
```

In the above example brace expansion was used to form the list but there are other syntax options; the following syntax examples will all give the same output.

for i in {1..3}; do echo "This is $i iteration"; done

for i in 1 2 3; do echo "This is $i iteration"; done

for i in $(seq 1 3); do echo "This is $i iteration"; done

An alternative and very useful syntax for the for loop is as follows, which is a style more often used in common programming languages (e.g. in the C language):

for ((expression; expression; expression))

The first arithmetic expression is evaluated, the loop is repeated so long as the second arithmetic expression is successful, and the third arithmetic expression is evaluated at the end of each loop. Here is an example to list a line three times as in the above examples:

```
for (( i=1; i < 4 ; i++ )); do
echo "This is $i iteration!"
done
```

In the following example a script program will accept a command line argument to specify a directory and will list the number of lines in each file within that directory. The example illustrates a useful application that uses a **for** loop to process a list of files, using the * wildcard.

```
#!/bin/bash
# List the number of lines in each file of a specified directory

for  file  in  $1/* ; do
  echo  "There are $( wc -l  < "$file" ) lines in $file"
done

exit 0
```

Here is an example output of the script, assume the script is names ‘example’:

```
 $ ./example  ~/hobbies
There are 101 lines in /home/Donal/hobbies/model_cars
There are 207 lines in /home/Donal/hobbies/model_planes
There are 83 lines in /home/Donal/hobbies/hiking
```

4.6.2 The while loop

The general format of the **while** loop construct is:

```
while  [ test something ] ; do
  commands
done
```

The following script program uses a while loop and will echo five lines to the terminal.

```
#! /bin/bash
num=5
```

```
count=1

while   [ $count  -le  $num ] ; do
do
      echo “This is line number $count”
      ((count++))
done
exit 0
```

The following script example will keep looping until a valid string match is found for the requested name.

```
#! /bin/bash

while   [ "$my_name"  !=  "John Smith" ]  ; do
  echo  "Guess my name"
  read  my_name
  echo  "You guessed $my_name"
done
echo "You guessed it correctly!"

exit 0
```

4.6.3 The until loop

The **until** loop is similar to the while loop. It will execute commands until the defined test becomes true. The **until** is not used very often, the **while** loop is much more widely used in practice. So you may be asking, 'Why bother having the two different kinds of loops? In some cases the logic of a program is easier to read if it is phrased with until rather than while. Here is a simple example script that uses an **until** loop and will echo 5 lines to the terminal, similar to the earlier script that was based on a **while** loop.

```
#! /bin/bash
num=5
count=1

until   [ $count  -gt  $num ] ;   do
      echo "This is line number $count"
      ((count++))
done
exit 0
```

4.6.4 Break and continue

The **continue** statement causes a loop iteration to be terminated and the next iteration then begins. In the example below a script program will list the number of lines in each file of a specified directory, however, if any file of type ‘directory’ is encountered, that file will be not be processed for line counting, and loop continues to next iteration.

Do not forget to provide a directory name as an argument is calling this program.

```
#!/bin/bash
# List the number of lines in each file of a specified directory,
# if any directory file is encountered - do not line count it.

if [ $# != 1 ]; then echo "You must provide a single target dir name. Try again!"; exit; fi

for  file  in  $1/* ; do
  if [  -d "$file" ]; then
     echo "The $file is a directory so will not do line count on it!"
     continue
  fi
echo  "There are $( wc -l < "$file" ) lines in $file"
done

exit 0
```

The **break** statement causes a program to exit the script before its normal ending. Consider a feature extension to the above program where if a file of type ‘pipe’ is found then the script is exited.

```
#!/bin/bash
# List the number of lines in each file of a specified directory.
# If any directory file is encountered - do not line count it.
# If a file of type pipe is found then exit the script.

if [ $# != 1 ]; then echo "You must provide a single target dir name. Try again!"; exit; fi

for  file  in  $1/* ; do
  if [  -p "$file" ]; then
     echo "Sorry $file is a pipe so script will exit this script!"
     break
  elif [  -d "$file" ]; then
     echo "The $file is a directory so will not do line count on it!"
     continue
  else
    echo  "There are $( wc -l < "$file" ) lines in $file"
  fi
done

exit 0
```

From the above we see the **continue** statement is used to exit a loop iteration but the **break** statement is used to exit the script completely.

4.6.5 Loops and I/O redirection

We saw how to use control loops by testing command results or by reading user input. It is also possible to specify a file from which to read input and in that way to control the loop.

For example the following style can be used to read a file line by line and the loop will terminate when there are no more lines to be read.

while read i; do echo $i; done < /dir/filename

As an example, suppose we have a file called ‘list_file_owners’ and it contains a list of entries where each line contains a file name and the respective file owner, as follows:

File: list_file_owners
joe file_A
bill file_B
root file_C
joe file_D
bill file_E
joe file_F
don file_G

Now assume we are asked to write a script that will display how many files belong to each individual owner; for example joe has three files. Here is an example solution that uses a control loop that reads its input from a file.

```
#! /bin/bash
#  Display number of files for each owner

# Make an unique list of owners in uniq_list
sort list_file_owners | awk '{print $1}' | uniq > uniq_list

# Make simple column titles
echo -e "\nOwner  \t Num_files \n"

# Loop to read each owner and count number of entries
while  read  owner; do
  x=$( grep "$owner" list_file_owners | wc -l )
  echo  -e   "$owner  \t  $x"
done  <  uniq_list

rm   uniq_list
exit 0
```

The result from the above script will be as follows:

```
$ ./example

Owner  Num_files

bill        2
don         1
```

joe 3
root 1

4.7 More On Variables

We will will look at some more shell features for variables and associated elements, including declaration of variable types, arrays, associative arrays, parameter expansion for transforming variables, and the use of heredocs.

4.7.1 Variable declarations

In Bash some different types of variables can be declared although it is not a typed language. The variable type is used to define the content type that is permitted, and includes the following type information as summarised in the table.

Declaration option	**The type of information for variable**
declare -a *variable*	An array of strings. The –a is seldom used and array=(...) is more common.
declare -A *variable*	An associative array of strings.
declare -i *variable*	An integer. Assigning values automatically triggers arithmetic evaluation; but –i is seldom used.
declare -r *variable*	Cannot be modified or unset.
declare -x *variable*	Marked for export. It will be inherited by any child process.

There is a useful **-p** option for the **declare** command that can be used to display attributes and values for each variable. In using the **-p** option additional options are ignored. Consider this small example to illustrate the use of the **–p** option and note that arithmetic evaluation works as the variable **num** is delared with the **–i** option:

```
$ declare  -i  num=99+2
$ declare -p num
declare -i num="101"
```

4.7.2 Arrays

In Bash an array is an indexed list of strings to conveniently store multiple strings without needing a delimiter which is often error prone since word splitting breaks a string wherever there is whitespace. The array feature provides a safer way to represent multiple string elements where an array simply maps

integers to strings and such a string element can contain any character, including a whitespace. The use of arrays is strongly encouraged in Bash. Associative arrays are also supported in Bash.

Array element numbeing is zero-based, that means the first element is indexed with the number 0. Here is an example to show how an explicit declaration of an array is done using the **declare** command:

declare -a my_array

However, array variables are more often created using compound assignments:

my_aray=(element1 element2 …. elementN)

We can see how to dereference the elements in an array with the following example:

$ my_array=(black brown red)

$ echo ${my_array[@]}
black brown red

Notice the syntax used to expand the array is using **${my_arrays[@]}**, telling Bash to replace this syntax with each element in the array.

It is usually safer to use quotes when assigning string variables. Here is another example:

$ my_array=(“black” “ brown” “ red” “sea blue” “ sky blue”)

We can specify explicit indexes as follows:

$ my_array=([0]=“black” [1]=“brown” [2]=“red” [6]=“sea blue”)

echo ${my_array[@]}
black brown red sea blue

In the above example there is a gap in the indices sequence between 2 and 6 so this is a sparse array.

The following example illustrates the use of a **for** loop to iteate over array elements.

```
#! /bin/bash
my_array=("black" "brown" "red" "sea blue")
for colour in "${my_array[@]}"; do
   echo  "$colour"
done
exit 0
```

The output of the above example will be:

$ ./array_example
black
brown
red
sea blue

Note in the example the use of quotes in **“${my_array[@]}”** to achieve the intended expansion. Without the quotes Bash will wordsplit the array elements where there are white spaces.

The example above expanded the array using a **for** loop statement. The array can be expanded easily to access its elements as arguments. Here is an example using the **printf** command.

Assume our sample array has been created as follows:

$ my_array=("black" "brown" "red" "sea blue")

The **printf** command will display an output as follows:

$ printf "%s\n" "${my_array[@]}"
black
brown
red
sea blue

Using * to expand array elements
Another form of expanding array elements, in place of **${my_array[@]},** is to use **${my_array[*]}**. However this form converts an array into a simple single string, which is useful for display purposes but loses the proper separation of elements. To illustrate this, the **printf** command is now run again but this time using the **${my_array[*]}** syntax, as follows:

$ printf "%s\n" "${my_array[*]}"
black brown red sea blue

You will see in this output that a simple single string (one line) is displayed.

Number of elements in an array

It is often useful to know the number of element in an array variable. The **${#my_array[@]}** syntax can be uses as follows:

Assume our sample array has been created as follows:

$ my_array=("black" "brown" "red" "sea blue")

We can count the number of array elements as follows:

$ echo The number of elements is: ${#my_array[@]}
The number of elements is: 4

The length of a single element in an array can be found as follows:

$ echo The length of element 3 is ${#my_array[3]} characters
The length of element 3 is 8 characters.

Copying an array

To make a copy of an array, you can expand the array elements and store them in the new array as follows:

$ your_array=("${my_array[@]}")

4.7.3 Associative arrays

Bash supports associative arrays, which are arrays that go beyond the classical array that maps a number to a string. The associative array can map one string to another, so as to realise key/value pairs. An associative array is declared using the 'declare –A' command. We will introduce the concept with the aid of a small example which will use an array to list teachers that are assigned to specific subjects.

First we will declare an associative array called teachers:

$ declare -A teachers

Next we will assign some element to the associative array:

$ teachers=(["Science"]="I. Newton" ["Maths"]="A. Einstein" ["English"]="W. Shakespeare")

Next we will look at a single element:

$ echo ${teachers[Maths]}
A. Einstein

Now we will look at all the elements:

$ echo ${teachers[@]}
A. Einstein W. Shakespeare I. Newton

We will now write a small script to use a **for** loop to iterate over the 'teachers' associative array, as follows:

```
#! /bin/bash
declare -A teachers
teachers=( ["Science"]="I. Newton" ["Maths"]="A. Einstein" ["English"]="W. Shakespeare" )

for subj in "${!teachers[@]}"; do
  echo "For the subject $subj, the teacher is: ${teachers[$subj]}."
done

exit 0
```

When we run the above the output will be as follows:

```
$ ./assoc_array_example
For the subject Maths, the teacher is: A. Einstein.
For the subject English, the teacher is: W. Shakespeare.
For the subject Science, the teacher is: I. Newton.
```

There are some points to note from the above example.

First to clarify what is a key and what is a value. In the example array element below “English” is the key and “W.Shakespeare” is the value:

["English"]="W. Shakespeare"

The keys are accessed using an exclamation mark: **${!array[@]}**, the values are accessed using **$ {array[@]}**.

We can iterate over the key/value pairs like this:

```
for i in "${!array[@]}"
do
  echo "key  : $i"
  echo "value: ${array[$i]}"
done
```

So in the **for** ststement the **${!teachers[@]}** the keys are accessed using an exclamation mark, and the values are accessed using **${array[@]}**.

Note, the order of the keys returned from the associative array using the **${!teachers[@]}** syntax is not predictable.

4.7.4 Transforming variables using parameter expansion

Parameter expansion (PE) is the recommended approach for string modification; it is a term that refers to an operation which causes a parameter to be expanded, i.e. to be replaced by some content. Curly braces are placed around a parameter’s name and then PE allows some editing to be performed within the curly braces. Consider the following example, which assumes that two variables are set as follows:

```
$ city1="Paris-capital"
$ city2="London-capital"
```

Now we want to make a simple statement using the **echo** command, such as:

```
$ echo $city1 and $city2 are big cities!
Paris-capital and London-capital are big cities!
```

However, we want to improve on our statement for clearer reading, so we can edit each variable within the **echo** command using parameter expansion, as follows:

```
$ echo ${city1%-capital} and ${city2%-capital} are big cities!
Paris and London are big cities!
```

The above example showed substring removal where the **%-capital** code is intrepted as saying that from the end of the string the “-capital” substring is to be removed.

We will now start to look at the syntax used in the above example, and other examples, so as to introduce the concepts for PE.

Earlier we saw some examples that use external utilities to modify strings, using utilities such as **sed**, **awk**, **cut** etc. However, such solutions necessitate the running of an extra process; and this can impact on performance. Parameter expansion offers a simpler and faster solution for string modification
Some of the key parameter expansion features are listed in the table below.

The general syntax	**Brief description**
${parameter:-word}	**Use Default Value**. If 'parameter' is unset or null, the 'word' (can be expanded) is substituted. If not, the value of 'parameter' is substituted.
${parameter:=word}	**Assign Default Value**. If 'parameter' is unset or null, 'word' (can be expanded) is assigned to 'parameter' and the value of 'parameter' is substituted.
${parameter:+word}	**Use Alternate Value**. If 'parameter' is unset or null, there is no substitution, otherwise 'word' (can be expanded) is substituted.
${parameter:offset:length}	This is **Substring Expansion**. Substitute by starting at the character specified by 'offset' (0 indexed), and expand up to 'length' characters of ‘parameter’. If 'length' is not specified then expand to the end of 'parameter'. The 'offset' can be negative (in this case better to use parentheses) and if so it counts backwards from the end of 'parameter'.
${#parameter}	The **length** of the value of 'parameter' in characters is substituted.
${parameter#pattern}	**Substring removal**. On a match for 'pattern' from the **beginning** of 'parameter' the result is the value (expanded) of 'parameter' but the shortest possible match is **deleted**.
${parameter##pattern}	**Substring removal**. This is the same as immediately above, however this time it is the *longest* match that is **deleted**.
${parameter%pattern}	**Substring removal**. On a match for 'pattern' from the **end** of ‘parameter’, the result is the value (expanded) of 'parameter' but the shortest possible match is **deleted**.
${parameter%%pattern}	**Substring removal**. This is the same as immediately above, however this time it is the *longest* match that is **deleted**.
${parameter/pattern/string}	**Substring replacement**. The value of 'parameter' (expanded) that **first** matches 'pattern' is replaced by 'string'. If ‘string’ is omitted then first match of ‘pattern’ is replaced by nothing, i.e. deleted.
${parameter//pattern/string}	Same as directly above, but **every** match of 'pattern' is replaced by ‘string’

The following examples illustrate the use of some aspects of parameter expansion (PE).

Assume we set the following variable, which is a simple string:

$ my_file="$HOME/bands/beatles"

We can use PE to report the length of the string as follows:

$ echo "The length of my_file pathspec is: ${#my_file} characters."
The length of my_file pathspec is: 24 characters.

We can, of course, echo the full string as follows:

$ echo "This is my favourite file: $my_file"
This is my favourite file: /home/user/bands/beatles

But we may want to display the file name only, as follows:

$ echo "This is my favourite band: ${my_file##*/}"
This is my favourite band: beatles

In the above example this is what has been actually deleted: '/home/Donal/bands/'. In the example we could have extracted the file name using '**basename** $my_file' but we want to focus the discussion examples to parameter expansion so we will not consider **dirname** and **basename** commands here.

In the above example we used the **${my_file##*/}** syntax. If we had used just a single '#' symbol then we would have got an undesired result as only the first '/' would have been deleted, as follows:

$ echo "This is my favourite band: ${my_file#*/} "
This is my favourite band: home/Donal/bands/beatles

If we want to convert the full file pathname to just the path directory (without using the **dirname** command) then we could use the following PE based code:

$ echo "Directory name for file: ${my_file%/*}"
Directory name for file: /home/user/bands

In the above example this is what has been actually deleted: '/beatles'. If in the above example we had used a double '%' then we would have the following result, which is not what was intended, as now all of '/home/Donal/bands/beatles' is deleted, right back to the first '/':

$ echo "Directory name for file: ${my_file%%/*}"
Directory name for file:

Next we will show an example to illustrate the **${parameter:-word}** syntax. The full pathname for my_file is reported as follows:

$ echo "File pathname is: ${my_file:- There is no such file}"
File pathname is: /home/user/bands/beatles

However, if next we assume that there is no file at all that is called 'paul_file' (Paul's file), we will show that PE will determine that the relevant parameter is not set and therefore it will substitute the defined string word as follows:

$ echo "File pathname is: ${paul_file:-There is no such file}"
File pathname is: There is no such file

But, in the above example maybe we wanted to say that if there is no 'paul_file' then we should assign the full file pathname for 'my_file' to Paul's file as follows:

$ echo "If there is no Paul's file, assign this one: ${paul_file:= $my_file}"
If there is no Paul's file, assign this one: /home/user/bands/beatles

The following **echo** command illustrates that the assignment to the ‘paul_file’ parameter has indeed been achieved:

$ echo $paul_file
/home/Donal/bands/beatles

We see that the assignment operation did in fact occur.

Examples on substring expansion and substitution using parameter expansion

To create some examples, let’s first set the following example variable:

$college="University of Limerick, Ireland! "

Using **substring expansion**, select the 8 characters beyond the first 14 characters, as follows:

$ echo "The city is: ${college: 14: 8}"
The city is: Limerick

Now, using substring expansion, locate to 9 characters from the end of the string and select seven characters from there, as follows:

$ echo "The country is: ${college: (-9): 7}"
The country is: Ireland

Note – it was safer to use parentheses for the ‘(-9)’ above as the following command would have a very different meaning, as the ‘:-‘ is the ‘use default value’ operator:

$ echo "The country is: ${college:-9: 7}"
The country is: University of Limerick, Ireland!

Here is another example:

$ echo "The city is still: ${college: (-19): 8}"
The city is still: Limerick

Let’s take another simple string variable example to illustrate further use of parameter expansion, as follows:

$ car="the car drove fast along the road."

We can replace the first ‘the’ pattern with the string ‘a’, as follows:

$ echo "She said ${car/the/a}"
She said a car drove fast along the road.

If the pattern begins with '/' then all matches of pattern are replaced with the new string, as follows:

$ echo "She said ${car//the/a}"
She said a car drove fast along a road.

Note, if a pattern begins with '#' symbol, the match must occur exactly at the beginning of the expanded value of parameter. If a pattern begins with '%', the match must occur exactly at the end of the expanded value of parameter. See how the following examples are different:

There are no matches in the following example so our string remains unchanged:

$ echo "She said ${car/%the/a}"
She said the car drove fast along the road.

However, there is a match at the beginning in the following example so 'the' is substituted:

$ echo "She said ${car/#the/a}"
She said a car drove fast along the road.

There is a match at the end in the following example so substitution does occur:

$ echo "She said ${car/%road./highway.}"
She said the car drove fast along the highway.

The various examples above are intended to help illustrate the significance of parameter expansion (PE). The various examples, however simplistic they may be, will hopefully help to get the reader started in her understanding and appreciation of the PE features.

4.7.5 Using heredocs

In some scripts we may want to use a short block of multi-line data without having to store that data in a file. The heredoc (here document) can represent such a block of text. The heredoc can also represent a block of code. We may want to redirect some variable's block data content directly into a command. A heredoc can be useful for such applications.

We will see that for heredocs the << code is used simply for string literals and does not indicate the usual redirection operator; rather it is simply a starting delimiter for the heredoc. We need to define a word to act as a sentinel to signify the end of the heredoc. In the following script program example we use a heredoc and we choose the word 'END' as the sentinel; we say here that **<<END** is the heredoc operator:

```
#!/bin/bash
grep "the" << END
And when white moths were on the wing,
And moth-like stars were flickering out,
I dropped the berry in a stream
And caught a little silver trout.
END

exit 0
```

This script program example executes as follows, where the heredoc block of text is inputted to **grep**. Assume 'example' is the name of the script:

```
$ ./ example
And when white moths were on the wing,
I dropped the berry in a stream
```

Sometimes there are tabs used in the heredoc that we want to remove. We can use **<<-END** to tell Bash to remove tab characters from the beginning of each line in the heredoc before sending it into the command.

Herestring

A herestring is a compact and simple use of the heredoc concept. The <<< is the herestring operator. Here is a simple example of using a herestring in a command line to count the number of words in a text string:

```
$ wc -w <<< 'And when white moths were on the wing'
8
```

4.7.6 Dangers – executing code from variables

It is possible to have the content of a variable represent executable code and this can have serious security implications as injected variables can become dynamically executable. The **eval** command will be introduced as an example security threat.

The eval command has the following simple form:

eval [arguments ...]

The **eval** command evaluates each argument and concatenates the resulting strings, separated by spaces, and executes this string in the current shell environment. Here is a simple example:

$ COMMAND="echo $USER"
$ eval $COMMAND echo 'is the user'
ringo is the user

The line with eval above is equivalent to:

echo $USER
echo 'is the user'

Note – rogue use of the eval command has security implications. For example consider this example:

$ FRIEND="rm -f $USER*"
$ eval FRIEND

Here the variable FRIEND is an enemy that will silently remove any files whose name begins with the user's name. Thus, if an intruder can somehow access some variable then a lot of damage can be caused to a system.

4.8 Shell Script Functions

4.8.1 Functions

A function is a short script that can be called any number of times within the full script. The Bash shell supports the use of functions, which can be called by the function name, as if calling a shell command. Functions in Bash, like many programming languages, are a useful way to reuse code. The function name must be unique within the script. The commands that make up a function are executed as regular commands. A function is executed within the shell in which it has been declared, i.e. a new process is not created to interpret the commands. A function can be defined in one of two ways as follows:

```
fun_name () {
        command list
      }
```

Alternatively, the **function** keyword can be used as follows:

```
function  fun_name {
```

```
    command list
}
```

The first style above, that does not use the function keyword, is much preferred and should always be used. The other style is presented here so that the reader can read older code. The function definition must exist in the script before any calls are made to the function. The function name should be descriptive of whatever task is required of the function.

In general programming languages arguments are passed to the function as listed inside the parentheses (), but in Bash the parentheses are used only to state that the declared identifier is a function and thus no arguments will ever exist inside the parentheses.

The following is a simple script program that uses a function called **greet**:

```
#! /bin/bash

# Define the 'greet' function
greet () {
  my_name=$USER
  echo "Hello $my_name"
}

# The main program – just calls 'greet' a couple of times.

greet     # call the 'greet' function
greet     # call the 'greet' function

exit
```

The script will give the following outcome; assume the name of the script is 'example':

```
$ ./example
Hello donal
Hello donal
```

Next we will learn how to pass arguments to a function.

Passing arguments to functions

Arguments are passed into the function in a similar manner to passing command line arguments to a script. The arguments are stated directly after the function name on calling a function. In the actual function the arguments are accessible as positional parameters $1, $2 etc. The following example passes a single argument to a function:

```
#! /bin/bash

# Define the 'name_greet' function
name_greet () {
   echo "Hello $1"
}
```

```
# The main program – just call 'name_greet' a couple of times.

name_greet Ringo     # call the 'name_greet' function
name_greet George    # call the 'name_greet' function

exit
```

The script will give the following outcome; assume the name of the script is 'example':

```
$ ./example
Hello Ringo
Hello George
```

The script example was very simple, passing just a single argument.

Note, the positional parameters passed to a function are not the same ones passed to the actual full script program. On executing a function, the arguments passed to the function become the positional parameters, as seen by the function. The parameter $#, which indicates the number of positional parameters, is updated to reflect this change. The positional parameter $0 is not changed and is the name of the actual script program. When a function is executing, the shell variable $FUNCNAME is set to the name of the function. On completion of a function, the values of the script program's positional parameters and the parameter $# are restored.

Returning a value from a function

Bash does not return a value from a function in the same sense as functions can return values in most other general programming languages. Rather, in Bash a return status can be set, which is similar to a command exiting with an exit status to indicate success or otherwise. A function uses the optional **return** statement to return the exit status. The return value defaults to the exit status of the last command that was executed in the function, where typically a return status of 0 indicates success; a non-zero value indicates an error. If an **exit** statement is used within a function, then the entire script program is exited, regardless of how deeply nested a function may be. The function's exit status is accessible using the special **$?** shell variable.

When the **return** statement is executed within a function, the function then completes and then execution resumes with the next command following the function call.

Now, here is a simple example script program where two arguments are passed to a function, and the function calculates the product of the two arguments, and returns the answer using the **return** ststement. However, note that this is NOT THE INTENDED use of the **return** statement.

```
#! /bin/bash

# Example script – but NOT the intended use of 'return' in Bash

# Create the 'product' function
product () {
  (( product_var = $1 * $2 ))
   return  $product_var
```

```
}


# The main program
product  22  3     #  product function is called with two arguments
echo  "The answer is: $product_var”

exit
```

The script will give the following outcome; assume the name of the script name is ‘example’:

```
$ ./example
```
The answer is: 66

Here is the example again but this time the product value is accessed using a global variable.
NOTE – generally using such global variables should be avoided.

```
#! /bin/bash

# Example showing function to return a product value using a global variable

# Create the 'product' function
product () {
  product_var=$(( $1 * $2 ))  # global variable since keyword ‘local’ is not used
}

# The main program
product 22 3     # product function called with two arguments
echo  "The answer is: $product_var"

exit
```

Note, using global variables is often undesirable, which leads us to a short discussion on the scope of a variable in Bash.

Scope of function variables

By default a variable that is declared in Bash is global, meaning that it is visible everywhere in the script program. A variable can be declared as a local variable by using the keyword **loca**l when first declaring the variable. If we declare a local variable inside a function, then that variable is seen only within that function. The keyword **local** is used as follows:

```
local  name_of_var=<var_value>
```

Usually in computer programming it is good practice to use local variables inside functions, as it is safer so as to avoid the possibility of inadvertently changing the value of a variable by some other code in the overall program. Consider the following example script:

```
#! /bin/bash
```

```
# demonstrate local and global variables

some_fun () {
  local  name1='Imposter'
  name2='Pretender'
  echo "Within the function my two friends are $name1 and $name2! "
}

name1='Punch'
name2='Judy'

echo "Before calling the function my two friends are $name1 and $name2! "
some_fun
echo "After calling the function my two friends are $name1 and $name2! "

exit
```

Here is the expected output for the script; assume the name of the script name is ‘example’:

```
$./example
Before calling the function my two friends are Punch and Judy!
Within the function my two friends are Imposter and Pretender!
After calling the function my two friends are Punch and Pretender!
```

Examine the logic of the script to see why the function was able to globally change the **$name2** variable but the value of the **$name1** variable was not changed outside of the function, as this is a local variable.

Commonly used functions

The commonly used functions can be listed in a file and used by various script programs without needing to repeat writing out the functions for each script program As a simple example here is a file called **my_functions** that contains two functions. The file is located in the **home** directory:

```
# This file is a list of functions saved in a file called ‘my_functions’

# Define the ‘name_greet’ function
name_greet ()  {
   echo "Nice to meet you $1!"
}

# Define the ‘name_bye’ function
name_bye ()  {
   echo "Goodbye my new friend $1"
}
```

Now we will write a simple script program that will use these two functions as follows:

```
#! /bin/bash
```

```
# Include the list of functions
. ~/my_functions

# The main program – simply calls the ‘name_greet’ and 'name_bye' functions.
name_greet $1     # call the ‘name_greet’ function
name_bye $1     # call the ‘name_bye’ function

exit
```

The output from this script will be as follows; assume the name of the script name is ‘example’:

```
$ ./example George
Nice to meet you George!
Goodbye my new friend George!
```

Note, the script file used the following line of code rather than listing the two functions:

```
. ~/my_functions
```

This line is telling the script to include this **my_functions** file in the script. Note, the **dot** operator (at the start of the line) is the similar to the include directive in C programs.

4.9 Some Miscelleanous Commands

Consider the following shell built-in commands which were inherited from the Bourne Shell, and are supported in the POSIX standard.

4.9.1 The null command

The null command is represented by a simple colon as follows:

: [*arguments*]

This command does nothing beyond expanding its arguments and performing redirections. The command’s return status is zero. The null command can be useful is some circumstances as in the following examples.

Sometimes we might want to allow parameter expansions side-effects to occur. Consider the following code which uses the parameter expansion operator := to achieve the desired operation:

```
read -p "Please enter your name: "  name
: ${name:=NotaName}  # if an empty string is entered assign a default name
echo "$name"
```

We can put this a bit in a more general sense. Assume we want to assign the string “abcd” to the variable var, if var is not already set or if it is set to null:

```
# Assign "bar" to the variable "foo" if it is not set,
: ${var:="abcd"} # quotes not necessary here but often preferred.
```

In the example we just needed the side-effect of the parameter expansion and the null command allowed us to achieve that.

Another example using the null command is its use in an endless loop:

```
while : ; do
        echo "Hit Ctrl=C to stop.."
        sleep 1
done
```

The while loop holds since the null command always returns true. We could have used 'while true' in the above example.

4.9.2 The 'dot' or source command

The command is simply represented as a period and it has a synonym which is the source command, as follows:

. (a period)

The command is called as follows:

. filename [arguments]

This command will read and execute in the current shell the commands from the *filename* context. If arguments exist they become positional parameters for filename on execution. The return status is the exit status of the last command executed or zero if no commands are executed. If *filename* is not found, or cannot be read, the return status is non-zero.

4.10 Debug Mode

For most UNIX shells a common way to debug shell scripts is to write **set -x** at the beginning of the script. On running the script this will cause each command of execution to be listed to the standard output; along with its arguments after the commands have been expanded but before they are executed.

To set the debug mode in a bash script simple insert the **set -x** command to the start of the script as follows:

```
#! /bin/bash
set –x
:
rest of script program as normal
:
:
```

Sometimes you may want to set this debug mode for just a section of code in the script. You can use **set –x** where you want to start the debug and **set +x** where you want to end the debug, as follows:

```
#! /bin/bash
:
various script commands
:
:
```

```
set –x     # start debug mode here
:
code that you want to debug
:
set +x    # stop debug mode here
:
rest of script program as normal
:
:
```

Another option is not to insert the **set –x** in the script code but simply run a bash script in this debug mode by calling the script with the **–x** option as in this example:

```
bash  –x  myScript
```

The Bash Debugger Project

A source-code debugger for bash has been developed under the The Bash Debugger Project. This debugger follows the **gdb** command syntax.

4.11 Bash Programming Style Guide

Donal Heffernan 27/November/2015

A style guide is intended to recommend how to write Bash scripts with a style that makes them safe and predictable. There is no definitive style guide for Bash programmers. The following list presents some documents with recommended style features for Bash programming.

Google style guide
Google have a recommended style guide for Bash programming:
https://google-styleguide.googlecode.com/svn/trunk/shell.xml

The Bash-Hackers Wiki
The Bash-Hackers Wiki provides coding guidelines that is intended to help programmers to read and understand Bash shell code, and to help produce robust code:
http://wiki.bash-hackers.org/scripting/style

Greg Wooledge's guide to Bash styles
This link by Greg Wooledge is very good and makes strict recommendations for good UNIX shell programming: http://mywiki.wooledge.org/BashGuide/Practices

A summary style guide
Here is a summary style guide which is largely based on Greg Wooledge's pages above:
https://github.com/bahamas10/bash-style-guide

How about the C shell?
Bruce Barnett is author of a well-known article on why you should not use the C shell; and strongly suggesting not to learn the C shell as a first shell in learning UNIX scripting. Here is the link:
http://www.grymoire.com/unix/CshTop10.txt

The article is: "Top Ten Reasons not to use the C shell"; written by Bruce Barnett. Last known update: June 28, 2009

4.11.1 Bash Style Summary Guide

Since there is no single style guide for Bash, presented here is a summary of some specific recommendations based on reviewing the content of the documents that are referenced above. This is not intended to be a definitive work; rather it is just a summary of the various inputs from the above mentioned documents.

Executable file extensions
An executable file should not have an extension, unless it is the .sh extension

Comments
Every file should have a top-level comment to briefly describe the content, for example:

```
#!/bin/bash
#
# Monitor disk usage on the system
```

Formatting
Indentation – indent 2 spaces but do not use tabs at all.
Put blank lines between blocks.
Maximum line length should be 80 characters, if a longer string is needed use a 'here document' or an embedded newline.
Line spacing - do not use more than one blank line in a row

Variable expansion
Quote the variables, "${var}" (i.e. brace quoting) is preferred over "$var" as a general guideline.
However, avoid using the ${var} form when possible in arithmetic equations.
Do not brace quote single characters, shell special variables and positional parameters.

Using quotes
Always use quotes for strings that contain variables, command substitutions, spaces or meta characters. Although sometimes unquoted expansion is necessary.

Do not quote literal integers.
Pattern matching may require special quoting rules

As a general guide use double quotes for strings that will require variable expansion or command substitution interpretation, otherwise use single quotes.
In general, if a variable will undergo word-splitting it must be quoted, otherwise the variable may remain unquoted. The predefined variables such as like $$, $?, $#, etc. do not require quotes as they will never contain spaces, tabs, or newlines.

Command substitution
For command substitution, you should use $(…) instead of back ticks.

```
Do not use:    var=`date`   # not used any more
Do use:        var=$(date)   # good!
```

Conditional tests
For conditional test always use [[...]] instead of [..].
Do not use **test** as in: test –e ~/big_file

The [[...]] style is safer, it reduces errors as no pathname expansion or word splitting takes place. It supports regular expression matching, which is not supported with [...] .

String tests
It is preferred to use -z (string length is zero) and -n (string length is not zero) over testing for an empty string, such as [["${my_name}" = ""]]

Pipe and the while loop
Piping to a while loop – there is a special problem when piping directly to a while loop; as an implicit subshell in the pipe to while can make debugging difficult.
Variables modified in a while loop will not propagate to the parent as the loop's commands have been run in the subshell. Here is an example that pipes to a while loop:

```
some_command | while read var; do   # piping to a while
  echo ${var}
done
```

Use process substitution in preference to piping to a while loop or use a for loop in preference if this is possible in the program logic.

Do not use the Bash eval command
Use of the eval command can be a security concern as the content of a variable can be dynamically executed as script commands.

Suggestion for naming conventions
There are no hard and fast rules on naming conventions. The information below suggests some good practices and recommendations.

Source filenames: - a good suggestion is to use lowercase, and use underscores to separate words if this enhances the meaning of the program name.

Function names: - use the same recommendation as for source filenames above.

Variable names - use the same recommendation as for source filenames above.

Constants and environment variable names: - use all uppercase, separated by underscores if desired. Declare constants at the top of the file. Constants and variables exported to the environment will be capitalised.

Use of echo and printf
Use echo for the simplest outputs, better to use printf if any options are needed.

Use of Semicolons
Do not use semicolons in scripts.

Use this: name='donal';
Do not use this: name='donal'

Function style
Do not use the function keyword.
It is best to ensure all variables created in a function are made local.

Use this style:

```
funct() {
   local x=funct   # local variable
}
```

Function placement
Put all the functions together in the file, positioned immediately below the constants.

Use a main function in bigger programs
For larger script programs that included functions, it is often useful to uses a function called main. This easily identifies the start of the program. The main function will be positioned as the bottom most function. The program can start as the last non-comment line in the file can be a call to the main function as **main "$@"** . Google encourages this style.

Comments for functions
All function comments should include:

- Brief description of the function
- List of the global variables (if any) that are used and modified
- The arguments for the function
- The function's returned values, other than the default exit status the last command

Syntax in Block Statements
For **if**, **do** and **while** statements, the **then** keyword should be on the top line, as follows:

Use this style:

```
if true; then
   ..
fi
```

Do not use this style, as the **then** is not on the top line:

```
if true
then
   ..
fi
```

Sequences
Use the built-ins that are provided in in bash when making sequences.

Do not use:

```
n=10
for  i  in  $(seq 1 9);  do
   ...
done
```

Do use:

```
for   i  in {1..5};  do
   ...
done
```

And this is good to use:

```
for  ((i = 0; i < 9; i++));  do
   ...
done
```

Arithmetic
Use the following ‘double parenthesis’ style for arithmetic operations: ((...))
Do not use the **let** command, nor the **expr**; they are long past the ‘sell by date’.
Use the following style for arithmetic expansion: $((...))
Do not use this style for arithmetic expansion, it is deprecated: $[…]
For arithmetic conditional tests use: if ((x > y)); then ..
Do not use: if [[$x -gt $y]]; then ..

Use Parameter Expansion
Bash has powerful features for parameter expansion (PE) and it is strongly encouraged to use PE in preference to using external commands such as sed, grep and awk etc. Consider the following simple example:

```
pw=’4ringo1234'   # this is a password
```

We want to remove all numbers from the above password.

You could use this code, based on sed:

```
$ pw_alpha=$(echo  $pw  |  sed  's/[0-9]// g')   # the g is for global
```

However, the above code takes too much time to execute as it needs to call echo and sed. It is much better, faster, to use PE as follows:

```
$ pw_alpha=${pw//[0-9]/}
```

Trust in ls?
Since a filename can legally contain characters that include whitespace, newlines, commas, pipe symbols etc. we must examine how the **ls** command behaves for some rare but legal file names that contain such characters. There are some problem issues, for example **ls** will separate filenames with newlines, when stdout is not a terminal.

Based on the above discussion, the following style in not encouraged, and some documents suggest that the use of this style is very wrong:

```
for x in $(ls);  do
 ...
done
```

It is much more preferable to use:

```
for x in *;  do
  ...
done
```

Use arrays where possible for string lists
In Bash an array is an indexed list of strings to conveniently store multiple strings without needing a delimiter. The array feature provides a safer way to represent multiple string elements where a string element can contain any character, including a whitespace. The use of string arrays is strongly encouraged in Bash.

Consider the following example that uses an array with a for loop:

```
colours=("black" "brown" "red" "sea blue")
for col in "${colours[@]}";  do
   echo  "$col"
done
```

Notice the syntax used to expand the array is using **${my_arrays[@]}**. Another form of expanding array elements is to use **${my_array[*]}**. However this form converts an array into a simple single string, which is useful for display purposes but loses the proper separation of elements.

Positional parameters
In many cases it is better to use “$@ “ in preference to using $* .

The "$@" represents an array-like construct of all positional parameters {$1, $2, $3 ...},
while "$*" represents the IFS expansion of all positional parameters, $1 $2 $3 ...

Declaration of variables
Uppercase variable names are generally discouraged unless there is a good reason to use them.
Only use **declare** for declaring associative arrays.
Almost always use **loca**l in functions when declaring arrays.

Read-only variables
Read-only variables should be explicitly declared.

Careful error checking
Be careful on error checking as even simple commands can unpredictably fail as in the following example that uses the **cd** command:

This might fail as the path might not exist:

```
cd ../cpulist    # what if the path fails…
```

Better to be careful like this, we now define the behaviour on failure:

```
cd ../cpulist  || exit
```

It is best to always test on using cd.

Global variable testing
Test for errors on using global variables to avoid unforeseen errors.

Some don’t like the cat!
For performance reasons it is better for a command to read a file by its name where possible. If a command supports reading from stdin, then we can use Bash redirection. We should use a separate command, such as cat for example, only if we need to. Here are some examples:

Preferred not to use **cat** if it is avoidable, so do not use this for example:

cat myfile | grep something

It is better to use this: grep something < myfile

Or, this is good also: grep something myfile

5 Processes

5.1 General

The UNIX/Linux operating system has multitasking features. Multiple processes can be run at the same time and processes can communicate with one another, using **pipes** and **signals** to provide IPC (**I**nter**p**rocess **c**ommunication).

5.2 Monitoring Process Execution

To see what processes are running, type the **ps** command (see **man** and **info** pages for a description of the **ps** command). Note, the **ps** command's options may be a little different for other flavours of UNIX/Linux shells.

For example type:

ps au

A list of process activity is shown, something like as follows:

$ ps au

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	3321	0.0	0.0	1876	408	tty1	Ss+	2007	0:00	/sbin/mingetty tt
root	3340	0.0	0.0	2484	408	tty2	Ss+	2007	0:03	/sbin/mingetty tt
donal	17205	0.0	0.0	4420	1468	pts/2	Ss	08:31	0:25	-bash
joe	19168	0.0	0.0	2928	776	pts/2	R+	09:30	0:00	ps au

The various fields can be described as follows:

USER	Name of the process user
PID	Process ID number
%CPU	What percentage of the CPU the process is using
%MEM	What percentage of memory the process is using
VSZ	Virtual memory usage
RSS	Real memory usage
TTY	Terminal associated with USER
STAT	The current state of the process
START	Time when the process started
TIME	Total CPU usage time
COMMAND	Name of process

The **STAT** field indicates the status of each process. Some of the codes are as follows:

R	Runnable
S	Sleeping (for a short time)
I	Idle (sleeping for a longer time)

L Waiting to acquire a lock
Z Zombie (dead)
D Uninterruptible sleep (woken from outside)
T Trace – temporarily halted for debug tracing

Some additional characters have the following meaning:
+ Foreground process group with control of the terminal
s Process is a session leader

Now we will demonstrate a really busy program which will take up a lot of the processor's time. We will write a simple program, called **busy_wait**, which is spinning in a loop, doing nothing except using up valuable processor time, since the condition that it is waiting on will never happen.

Create the **busy_wait** shell script program below, make it executable (using chmod).

The **busy_wait** program

```
#!  /bin/bash
#  Meaningless program – but useful to demonstrate the busy-wait concept

while true
do
  ((  x++  ))
done
```

Run this program as a **background** process by typing:

./busy_wait &

Note, the **&** symbol following a command tells the system that the program is to be run in the background. The shell can continue to run in the foreground.

Now type **ps au** and see what percentage CPU utilisation (%CPU) that this **busy-wait** program is using (probably close to 100% utilisation).

Run the **busy_wait** program a few more times by typing **busy_wait &** again.

Now type **ps au** and this will give a display something like the following:

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	3321	0.0	0.0	1876	408	tty1	Ss+	2007	0:00	/sbin/mingetty tt
donal	17205	0.0	0.0	4420	1472	pts/2	Ss	08:31	0:00	-bash
donal	19291	36.7	0.0	5572	972	pts/2	R	09:34	0:17	/bin/bash ./busy_
donal	19294	33.2	0.0	4660	972	pts/2	R	09:34	0:14	/bin/bash ./busy_
donal	19295	32.7	0.0	4436	968	pts/2	R	09:34	0:14	/bin/bash ./busy_
donal	19321	1.0	0.0	2560	776	pts/2	R+	09:34	0:00	ps au

This is a simple example of multitasking; a number of programs (or a number of copies of the same program) are running at the same time, sharing the processor's time.

You can **kill** any one of these processes by using the **kill** command, e.g. **kill 19291** where 19291 is the process PID number. Now kill all of these **busy_wait** processes.

The top utility

There is a useful utility called **top** that will periodically display the process activity. Try this by typing **top**. Run the **busy_wait** script and monitor it in the **top** display. The **top** utility's data is updated on a periodic basis, so you can watch this activity on the screen. Your output should be of the following format:

Tasks: 174 total, 4 running, 170 sleeping, 0 stopped, 0 zombie
Cpu(s): 85.1% us, 14.9% sy, 0.0% ni, 0.0% id, 0.0% wa, 0.0% hi, 0.0% si
Mem: 2072932k total, 1684976k used, 387956k free, 628876k buffers
Swap: 2031608k total, 176k used, 2031432k free, 313104k cached

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
19291	donal	25	0	5572	972	3972	R	33.2	0.0	2:07.11	busy_wait
19294	donal	25	0	4660	972	3972	R	33.2	0.0	2:04.42	busy_wait
19295	donal	25	0	4436	968	3972	R	33.2	0.0	2:03.98	busy_wait
19494	donal	17	0	3324	1004	1664	R	0.3	0.0	0:00.03	top
1	root	16	0	2532	564	1408	S	0.0	0.0	0:01.41	init
2	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	migration/0
3	root	34	19	0	0	0	S	0.0	0.0	0:07.49	ksoftirqd/0
4	root	5	-10	0	0	0	S	0.0	0.0	0:00.21	events/0
5	root	5	-10	0	0	0	S	0.0	0.0	0:00.00	khelper
6	root	15	-10	0	0	0	S	0.0	0.0	0:00.00	kacpid
30	root	5	-10	0	0	0	S	0.0	0.0	0:00.00	kblockd/0

5.3 Signals

A program will need to be able to deal with unpredicted external events. Such events are often referred to as interrupts, as they interrupt the normal flow of a program. At a higher level, UNIX/Linux sends **signals** to processes to indicate that some type of event has occurred.

A signal is a short notification sent to a process to notify that process of a particular event. The signal interrupts the process so that it can act on this received signal. This provides a mechanism for handling asynchronous events in a system. There are a number of different kinds of signal, and each kind signal is associated with an integer identifier number and an associated symbolic name.

A signal is simply an event notification that is sent by some running process. Any process can send a signal to another process, as long as it has permission to do so. Signals do not carry any data. A process does not know what process sent the signal.

Signals are often used by the operating system kernel to notify a process that some event has occurred, without the process needing to poll for the status of the event. We say that the signal happens asynchronously.

When a process receives a signal, the corresponding **trap** code gets called. The Bash **trap** command allows the user to specify a command, or a function, that is to be executed when the shell receives a particular signal. The **CRTL-Z** and **CRTL-C** key inputs make direct use of signals. The **trap** command causes the shell to execute a specified command when a numbered signal, or signals, arrive. If numerous signals arrive they are handled in numerical order.

Signal Identifiers

POSIX has standardised the signal handling scheme for UNIX. Each signal has a symbolic name starting with the prefix SIG. For example SIGKILL is the signal sent when a process is forcefully terminated. Signals are all defined in the header file <signal.h>. The signals are pre-processor definitions that represent positive integers, i.e. each signal is associated with an integer identifier number as well as its symbolic name. The signal numbers start at 1 (SIGHUP). Signal number 0 has a special interpretation that we will not cover here.

There are many different signals defined. Use the **kill –l** command to see what signals are supported on your system. Here is an example (incomplete) output:

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN
```

For now we will consider some of the more common signals as listed in the table below.

Signal	Number	Explanation
SIGHUP	1	Hangup – controlling terminal no longer connected.
SIGINT	2	Interrupt – usually from keyboard (CTRL-C)
SIGQUIT	3	Quit – user has pressed a quit key (e.g. CTRL-\)
SIGKILL	9	Kill. Cannot be trapped or ignored. Fatal with no clean-up.
SIGTERM	15	Process termination signal. Default signal sent by kill

SIGHUP, the UNIX ‘hangup’ signal, is used to signal a process to say that the terminal is no longer connected. However, the SIGHUP signal is sometimes used as a general signal for other purposes. By default, the signal will terminate the process. By using the **trap** command, alternative action can be programmed to specify the response to the SIGHUP signal, or the response to other signals...

The **kill** command is used to send a signal to a running process, using the following syntax:

kill –(SignalNumber) ProcessID

For example, if you wanted to send a SIGHUP signal (which is signal number 1) to the process with the PID, 4365, you could use the command:

kill –HUP 4365

or, the equivalent command is:

kill –1 4365

Note, it is generally much preferred to use signal names, rather than signal numbers, so as to be more portable for different versions of UNIX/Linux shells. Use the signal name without the leading 'SIG' for more general compatibility.

It is important to note that not all signals can be trapped. The SIGKILL signal, signal number 9, cannot be trapped. Consider the following command which uses signal 9:

kill -KILL 4365

This command will be received by process 4365 and a signal trap cannot be programmed to intercept the SIGKILL signal.

The SIGINT signal, which is signal number 2, is the **interrupt** signal which is generated from the user keyboard by using the CTRL-C keys, and sometimes by other defined keys as well.

The SIGQUIT signal, which is signal number 3, is also generated from the keyboard (usually) and has a special use.

The SIGTERM signal, which is signal number 15, is usually generated by another process. The kill command uses this signal by default, so if we issue the command **kill 19291**, the SIGTERM signal is sent to the process with PID number 19291.

Using a trap

The general syntax of the trap command is:

trap 'command' <signal list>

Note the use of single quotation marks in the trap command. The 'signal list' is often a single signal, but can be a list of signals. When any one of the signals is received then the specified command is executed. The setting of a trap overwrites a previous trap on a given signal.

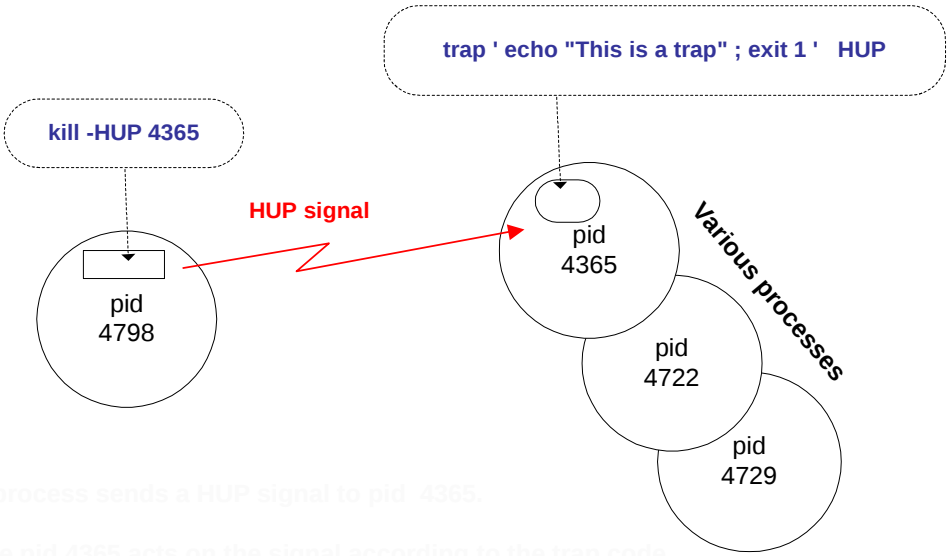
Here is a simple example program that uses the **trap** command. The program will continuously loop, but will exit with the message "This is a trap" displayed if a SIGINT signal is sent to this program.

```
#! /bin/bash
trap 'echo "This is a trap" ; exit'  INT

while true;  do
  echo  "I'm looping"
  sleep 1
done
```

Try out the above program; when it is running hit CTRL-C on the keyboard and you will see the altered behaviour of the CTRL-C operation.

The diagram below illustrates a HUP signal being sent (using **kill**) and a **trap** acting upon the receipt of that signal.



Other examples for the trap command are listed below:

1) The trap command allows the use of the signal number or signal name:

trap ‘ echo “This is a trap” ‘ 1 # uses signal number rather than signal name

2) The trap command using sequential commands:

trap ‘ echo “This is a trap” ; exit 1 ‘ 1 # includes a second command

3) The trap command allows the use of more than one signal:

trap ‘ echo “This is a trap” ‘ SIGINT SIGHUP # either signal will trap

Some special cases of interest are as follows:

If the command is an empty string, then the signals are ignored, as with the following command:

trap ‘ ‘ SIGINT

This is a useful way to ignore signal interrupts. Note, the signal **9**, i.e. **SIGKILL**, cannot be ignored.

The following command will restore the default behavior for each signal in the list:

trap - <signal list>

Trap using a function

The **trap** commands are often written as a single function. Consider the following example where on receipt of a SIGINT signal a program will remove all file in the home directory with the extension ‘.tmp’.

```
#! /bin/bash

# The function is written here before the main code
trap_function () {
  rm -f ~/*tmp              # remove files with extension .tmp
  echo "This program is closing now!"
  echo "I have removed all of your .tmp files!"
}

# Define response to a SIGINIT signal
trap 'trap_function ; exit'   SIGINT

# The main code is here - it is just a simple loop to simulate real activity
while true; do
  echo "I'm looping"
  sleep 1
done
```

EXAMPLE QUESTION

Consider the Bash script exhibit program as below.

```
# Exhibit program  D.H. 15/June/2015  ver. 1.0.0
#! /bin/bash

# The main code is here
./progB &        # start program progB in the background

# a simple loop to simulate real activity
while true ; do
   echo "I'm looping"
   sleep 1
done

wait     # wait for child to exit properly
exit
```

Modify this program so that it will include a **signal trap**. The **trap** will do the following:

i) Acts on receipt of a **SIGINT** signal (i.e. Ctrl C from keyboard)
ii) Contains a function called **trap_function()**
iii) The **function** does the following:
- displays (echoes) a simple message to say what is the **PID** for **progB**
- sends a **TERM** signal to the running **progB** program
- properly exits the script program without **orphaning** progB

NB: the shell variable **$!** *is always the process ID for the last background command*

SAMPLE ANSWER TO ABOVE

```
#! /bin/bash

trap_function() {
  echo
  kill  -TERM   "$!"
  echo "I have just terminated progB who's PID was $! and I am now exciting!"
  wait              # wait for child to exit properly
  exit
}

trap  'trap_function'  SIGINT

# The main code is here
./progB &        # start program progB in the background

# a simple loop to simulate real activity
while true ; do
   echo "I'm looping"
   sleep 1
done

wait    # wait for child to exit properly
exit
```

5.4 Multiple Processes

As discussed earlier, you can run a script program by simply typing its name. When a shell script executes a command, or a script program, it runs another copy of the shell as a subprocess, referred to as a subshell. This subshell executes the command and terminates, handing control back to the calling shell, referred to as the parent shell. For example, consider a script called **mainprog** that calls a script called **program1**. We will study some examples to learn the behaviour of a process and a subprocess that run concurrently.

5.4.1 Example for a program and a subprogram executing serially

The **mainprog** script, shown below, echoes a message to the screen and then calls the **program1** script, which echoes a message to the screen four times at one second intervals. The **mainprog** script then echoes two messages to the screen at one second intervals, and then exits. The **program1** script is also shown below.

Type in this **mainprog** script, as follows, and give it execute permission:

```
#! /bin/bash
# Demo program to call subprograms
echo 'Main program starting!'
./program1       # Here we start program1
for (( x=0 ; x < 2 ; x++ ))
  do
  sleep 1
   echo 'I am the main program'
done
exit
```

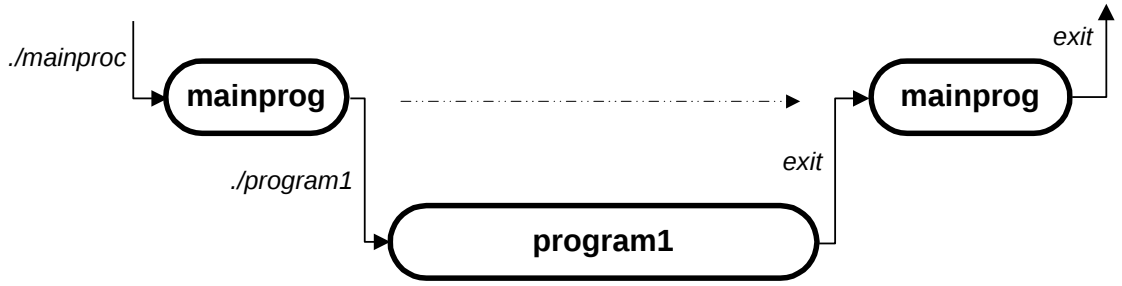
Type in the **program1** script, as follows, and give it execute permission:

```
#! /bin/bash
# Announce name 4 times at one second intervals
for (( x=0 ; x < 4 ; x++ ))
do
  sleep 1
  echo 'I am program 1'
done
exit
```

Run the **mainprog** script program on your system to make sure that it works! The output of your program will be as follows:

```
Main program starting!
I am program 1
I am program 1
I am program 1
I am program 1
I am the main program
I am the main program
```

The **mainprog** program execution behaviour can be plotted in a diagram as follows:



Since a UNIX/Linux operating system is multitasking, we should be able to run the subprocesses concurrently with the main process, as if we had a parallel processing system.

5.4.2 First attempt to execute the main program and a subprogram concurrently

We will modify the **mainprog** script, and call it **mainprog1**, so that it calls **program1** as a background process. This is achieved simply be adding the ampersand symbol '&' as shown in the bolded line below:

mainprog1

```
#! /bin/bash
# Demo program to call subprograms
echo 'Main program starting!'
./program1 &
for (( x=0 ; x < 2 ; x++ ))
  do
  sleep 1
   echo 'I am the main program'
done
exit
```

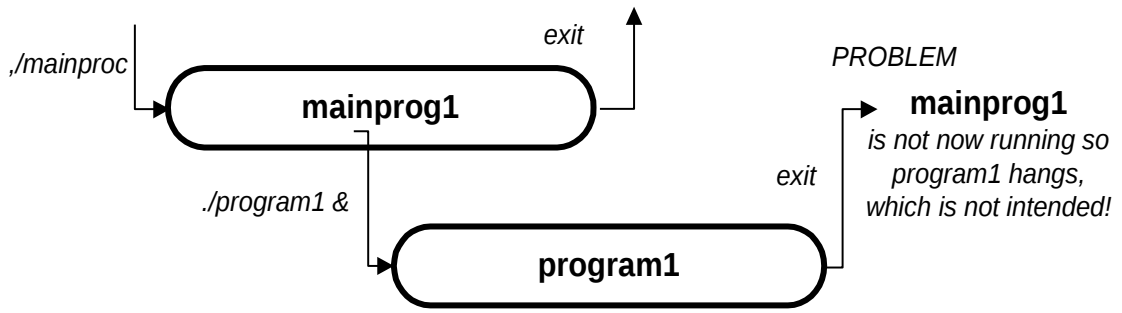
Now run this modified mainprog1 script by typing:

./mainprog1

A serious problem is now seen. The **mainprog1** seems to run and then exits back to the shell allowing **program1** to continue and then to hang, because its calling program, **mainprog1**, has already exited. The output is shown below, where on the 5th. line the main program exits and leaves **program1** running and then **program1** hangs when it is finished because its calling program is no longer active.

```
Main program starting!
I am the main program
I am program 1
I am the main program
donal:~/Desktop$ I am program 1
I am program 1
I am program 1
```

The execution sequence for **mainprog1** is shown in the following diagram:



5.4.3 Fix the concurrency problem by use of the wait command

Now, we will use the **wait** command to cause the main program to wait until **program1** exits, before the main program itself exits. Modify **mainprog1** to include a **wait** command as shown below in bold font. Name the modified program **mainprog2**, as follows:

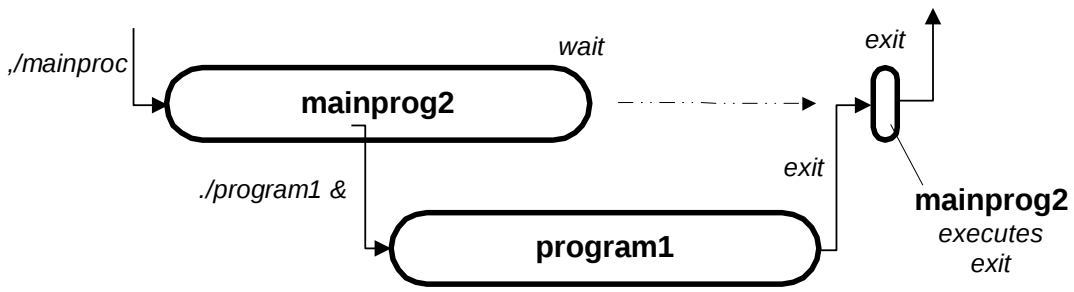
mainprog2

```
#! /bin/bash
# Demo program to call subprograms
echo 'Main program starting!'
./program1 &
for (( x=0 ; x < 2 ; x++ ))
  do
   sleep 1
   echo 'I am the main program'
done
wait
exit
```

The **wait** command will wait for all subprocesses to complete before **mainprog2** completes. Run this program, by typing **./mainprog2**, and you will see a result like as follows:

```
Main program starting!
I am the main program
I am program 1
I am the main program
I am program 1
I am program 1
I am program 1
```

The execution sequence for **mainprog2** is shown in the following diagram:



5.4.4 Demonstrate a main program and two subprograms executing concurrently

We will now modify **mainprog2**, let's call it **mainprog3**, to run two subprocesses in the background, as follows:

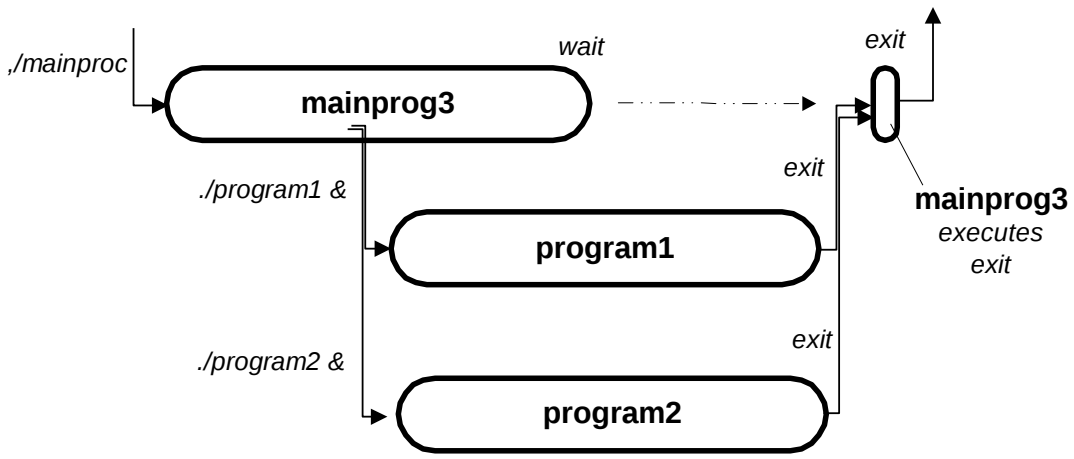
mainprog3

```
#! /bin/bash
echo 'Main program starting!'
./program1 &
./program2 &
for (( x=0 ; x < 2 ; x++ ))
do
sleep 1
echo 'I am the main program'
done
wait
exit
```

You must create the script file called **program2** which will be identical to **program1**, except it echos 'I am program 2' to the screen. Now run **mainprog3** and you will get a result as follows:

```
Main program starting!
I am the main program
I am program 1
I am program 2
I am the main program
I am program 1
I am program 2
I am program 1
I am program 2
I am program 1
I am program 2
```

The execution sequence for **mainprog3** is shown in the following diagram:



The above program leads to the following questions:

1) If all three processes are writing to the same terminal (screen) then is this not confusing? The answer is that it is not normally desirable for three processes to be writing to the same terminal. Often the background processes redirect their standard outputs to a file.

2) How can you predict the sequence in which the processes will write to the terminal? The answer is that it is not easy to predict the sequence or timing of access to the terminal. Therefore we say that a 'race condition' problem exists here. Such race condition problems are classical problems in multitasking (concurrent) systems.

3) How do three processes run in parallel (concurrently) on a system with just a single processor (a CPU)? The answer is that we have the illusion of parallelism. In reality the same processor is being shared by switching very quickly among all of the processes.

A note on wait and exit

UNIX/Linux processes will be discussed in more detail later in these notes. In the previous script examples you saw the **wait** and **exit** commands being used. The correct use of these commands is important for the multiple programs to work correctly.

When a script program calls a subprogram then there is a **parent/child** relationship between the program and its subprogram.

It is common for the parent to suspend, using a **wait** command, until the child process, or child processes, are terminated. If the parent did not wait then the child process would become an **orphan process**.

The **exit** command terminates a process. When a process calls **exit**, the kernel ensures that all file descriptors are closed, deallocates code, data and stack and the process terminates by sending a SIGCHILD signal to the parent. If the parent does not react properly, the child process can be left in a **zombie** state, i.e. it has terminated but the termination has not been recognised by the system.

5.5 More On Pipes

We have already seen the use of a pipe to connect the standard output of one command to the standard input of another, as in the following example command:

ls . a* | wc –l

In this example, the **ls** command lists all the files in the current directory, which start with the letter ‘a’, and the list output is passed to the **wc** utility, where the number of lines is counted.

The **ls** utility is running concurrently with the **wc** utility. We say that there are two processes running and that the pipe is the intercommunication mechanism between the processes. This simple pipe, as frequently used in shell commands, is referred to as an **unnamed** pipe, or an **anonymous** pipe.

The **unnamed pipe** is a unidirectional communication link between processes. The pipe is a simple buffer, where the writer process will block if the pipe is full and the reader process will block while waiting for an output from an empty pipe.

Named pipes

UNIX/Linux also supports a **named pipe** mechanism, which has advantages over the simple unnamed pipe. The named pipe name exists in the file system and thus it is accessible to all processes that have the appropriate access permissions. In contrast, the unnamed pipe is accessible only by parent/children related processes. The named pipe will continue to exist, just like a file, until it is explicitly deleted. The named pipe is designed as a unidirectional communications link, just like the unnamed pipe.

The named pipe has a FIFO (first in first out) buffer structure and is created using the **mkfifo** command, as in the following example:

mkfifo mypipe

If you now type the **ls – l** command you will see that the pipe exists as an entry in the file system. The ‘p’ character at the start of the line signifies that **mypipe** is a named pipe as follows:

prw-r--r-- 1 donal usergroup 0 Feb 14 11:46 mypipe

Since a named pipe is constructed to look like a file, we can use the normal UNIX read and write commands and system calls to communicate on the pipe.

The following is a simple example where two processes are created and one communicates with the other using a named pipe. Try this example to see that it works. Note, this is a simple example of a multitasking program with a communications mechanism. It is easy to run into problems, even with simple examples, where a process might block indefinitely, waiting for an input from another process.

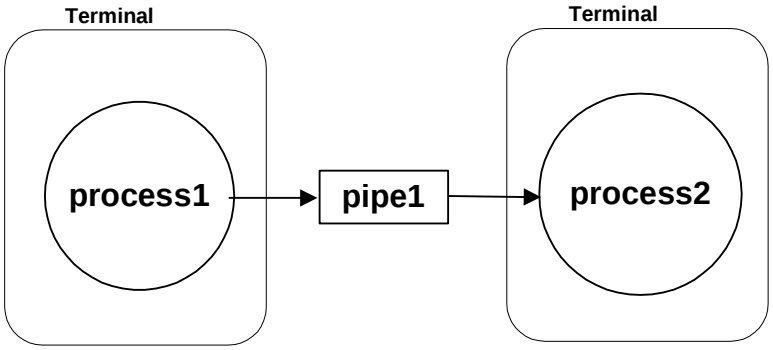
Instructions:

NB: this example asks you to use two terminals, running one program in each terminal.

1) Using an editor write the programs **process1** and **process2** as below.
2) Make these programs executable using **chmod +x**

3) Run these two programs (e.g. ./process1), running one program in each terminal.
4) The result should be as follows, as an output from **process2**:

```
Hello from ./process1
Hello from ./process1
Hello from ./process1
Last message!
I am finished!
```



Below is the **process1** program, it creates **pipe1**, if that pipe does not already exist, and sends a message 3 times on the pipe and then a final message. The **$0** shell variable always represents the name of the shell or shell script. Note that **process1** writes to the pipe as if the pipe were a simple file as follows: **echo "Hello from $0" > pipe1**. There is nothing new to learn in terms of writing, but the program might block on writing to the pipe if the pipe has not been read. This can be tricky to debug.

```
#! /bin/bash
# Create the pipe, if it does not already exist.
if [ ! -p pipe1 ]
then
mkfifo pipe1
else echo “pipe already exists”
fi

# Send a message 3 times
for (( x=1; x<4; x++ ))
do
echo "Hello from $0" > pipe1
sleep 1
done

# Send the last message and remove the pipe.
echo "Last message!" > pipe1
sleep 1
rm pipe1

exit 0
```

Here is the **process2** program, it creates **pipe1**, if the pipe does not already exist, and receives messages on the pipe, checking for the final message. Note that **process2** reads the pipe as if the pipe were a simple file as follows: **read input < pipe1**. The program will block on reading the pipe if the pipe is empty.

```
#! /bin/bash
# Create the pipe, if it does not already exist.
if [ ! -p pipe1 ]
 then
 mkfifo pipe1
 else echo “pipe already exists”
fi

# Wait for last message
while [ "$input" != "Last message!" ]

# Read the pipe
do
 read input < pipe1
 echo $input
 sleep 1
done

echo I am finished!
exit 0
```

5.6 Job Control

Each process is assigned a unique number, referred to as a PID (**P**rocess **Id**entifier). The PID is assigned when the process is created. The shell supports the concept of foreground jobs and background jobs. The term job simply means a command that has been started from the shell command line. Normally a foreground job has control of the terminal. A program can be run as a background job by appending the **&** symbol. For example, a program called **testrun** could be run as a background job as follows:

```
testrun &
```

The shell will respond with a message as in the following example, where 2 is the job number and 4365 is the PID:

```
$ testrun &[2] 4365
```

If multiple jobs were started in the background, the shell will number these jobs in the order in which they were started, with the numbers 2, 3, 4 etc., for example:

```
$ testrun &[2] 4365
$ backup &[3] 4374
$ account &[4] 4377
```

When a background job completes a message such as the following is outputted:

[1]+ Done testrun

The **jobs** command will list all of the background jobs. The **jobs –l** option will list the PIDs also.

As an example, the **jobs** command might give the following result:

[2] Running testrun &
[3]- Running backup &
[4]+ Running account &

The **fg** (foreground) command brings a background job to the foreground. By default **fg** will bring the most recent job to the foreground. The following command will put job number 2 into the foreground:

fg %2

Alternatively the **fg %testrun** command could have been used. The most recent background job can be referred to by **%+** , while **%-** refers to the second-most-recent background job.

A foreground job can be put in the background by suspending the foreground job. This is achieved by typing **CRTL-Z** for the running job. The job will now be suspended and put in the background. You can use **CRTL-Z** followed by the **bg** (background) command to put a foreground job running in the background, instead of being suspended in the background.

Note - The **kill** command can also use job numbers, as follows:

kill -QUIT %3

APPENDIX B: Commands

Quick Command Reference Chart

The bash shell commands and utilities – a brief summary card (8/Dec/15)

Command/Util	Brief description
awk	Scans a file(s) and performs an action on lines that match a condition. General format: *awk ' condition { action } ' filename* Example: *awk ' /University/ {print $3,"\t", $11}' myFile*
bc	Arbitary precision calculator Example: *echo "scale=3; (1 + sqrt(5))/2" | bc* …. calculates phi to 3 places
cal	Display a calendar output
cat	Concatenate file to the standard output
cd	Change directory
chmod	Change file access permissions
chown	Change file owner/group
cp	Copy files and subdirectories
cut	Cut columns from a data file Example: *cut –c 49-59 logfile* … extract column defined between characters 49 to 59
dd	Copy a file, converting and formatting Example: *dd if=/dev/zero of=myFile bs=1k count=10* … makes myFile of 10 kiloBytes
date	Display current time, set date etc. Example: *date +%s%N* …time with nanosecond resolution
df	Display disk space information
diff	Compare files line by line to find differences
du	Display disk usage information
echo	Display a line of text
exit	Exit the process e.g.: exit 0 … exits with the code 0
find	Search for files Examples: *find / -type d –print* …find directory files starting at root and display *find . –name "verse"* …find all files, starting at the current directory, with "verse" string at start of name
grep	Scans text files looking for a string match. Examples: *grep "and" myFile* … search for lines containing "and" *grep "^The" myFile* … search for lines that begin with "The" *grep "floor$" myFile* … search for lines that end with "floor"
head	Display a number of lines at the head of a file
history	Display previous commands
kill	Sends a signal Example: *kill –HUP 43165* … send HUO signal to process 43165
less	Outputs a file to the console, a page at a time
ls	List directory(s) content ls –l long listing to show file details ls –R list subdirectories recursively ls –a list all files, including ones that start with *a* .
mkdir	Make directories
mkfifo	Make a named pipe Example: *mkfifo mypipe*

more	Outputs a file to the console, a page at a time
mv	Move files (effectively means to rename files)
ps	Show process status ps au show all processes, for all users
pwd	Print the name of the current working directory
read	Read user input
rm rm -R	Remove files and/or directories rm –r (or rm –R) will remove files recursively
rmdir	Remove directories (assuming directory is empty).
sed	A stream editor Example: *sed 's/Jack/Jill' filebook* … substitute the string 'Jill' for 'Jack' in file filebook
seq	Generates a sequence of numbers. Examples: seq 1 9 … generates numbers 1 to 9, line by line seq –s "-" 1 9 … default separator can be changed, using the –s option
set	If no options are used, set displays the names and values of all shell variables Examples: set …. shows all shell variables set | grep "USER" … shows shell variables with a specified string
sort	Sort lines in a text file sort –g general numeric sort sort –r reverse result of sort sort -k sort for a key position sort –n sort to string numerical value
tail	Display a number of lines at the end of a file
tee	Diverts a piped input to a second separate output Example: *cat demo_file1 | sort | tee demo_file1_sorted | more*
trap	Defines actions to take upon receipt of a signal or signals Example: *trap ' echo "This is my trap" ' SIGHUP* …. echo some text on receipt of HUP
uniq	Output a file's lines, discarding all but one successive identical lines
wc	Count number of lines, words, bytes etc. in a file wc –l count number of lines wc -c count number of bytes wc -m count number of characters
wait	Wait for child process to exit before finishing. e.g.: wait

Some common built-in shell variables

Variable	Description
$?	Exit status of the previous command
$$	Process ID for the shell process
$!	Process ID for the last background command
$0	Name of the shell or shell script
$PPID	Process ID for the parent process
$UID	User ID of the current process
$HOME	The home directory
$SHELL	The shell

Bash function example

```
# Example script program that uses two function parameters.
# The function calculates the product of the # two arguments:
# Not intended use of return
#! /bin/bash

# product is declared as a function and defined
product () {
(( product_var = $1 * $2 ))
return $product_var    # The product_var is returned
}

# The main program

product  22  3   # The product function is called, with two arguments
echo  "The answer is: $product_var"
exit
```

Bash array example

```
#! /bin/bash
my_array=("black" "brown" "red" "sea blue")
for colour in "${my_array[@]}"; do
   echo  "$colour"
done
exit 0
```

APPENDIX D: Exercises

Various exercises and short questions

Test Questions and Exercises
Please state what each one of the following bash shell commands does:

1) ls –l ./ ../ ../ (is it different to ls –l ./../../ ?)

2) cat ~/games/playfile | more

3) cat titlefile verse_1 verse_2 verse_3 >> full_poem

4) cp ./* ../

5) rm ../?est

6) chmod o+rwx test_file

7) find . -name "verse*" -print

8) grep "And" verse_2

9) grep "^d" temp_file

10) uniq ../testfile > ./ testfile1 ; wc –l testfile1

11) ls num[xyz]test

12) mv verse_temp verse_demo

13) awk ' /Science/ { print $1, $3 } ' students

14) $ x=$(ls -l | wc -l); echo $x

15) x=5 ; echo $((x * x + 5))

16) x= 7; y=$((x ++)) ; echo “$x $y”

17) x= 7; y=$((++x)) ; echo “$x $y”

18) x=$(echo $x | sed ' s/%/ / ')

Some sample questions:

The list below shows one-line command exercises, with solutions. It will be important that you know how to do such exercises, as you will use similar style commands and utilities for later exercises and assignments. So, if you are not confident with these type of exercises, please study and practice them.

Example single line set of commands as solutions to the following problems:

Q1 Find the **largest file** in the **home** directory and put the result in a file called **longFile** in your **home** directory

A solution:
ls -l ~ > temp ; sort –g –r –k5 temp > temp1 ; head –1 temp1 > ~/longFile

A solution using pipes:
ls -l ~ | sort –g –r –k5 | head –1 > ~/ longFile

Q2 Find how many files are in the **parent** directory of your **home** directory and put the answer in a file called **countFile**, in your **home** directory.

A solution:
ls -l ~/.. > temp ; wc –l temp > ~/countFile

A solution using pipes:
ls -l ~ /.. | wc –l > ~/countFile

Q3 **Find** how many **directory files** are in your **home** directory, and put the answer in a file called **countDir**, in your **home** directory.

A solution:
ls -l ~ > temp ; grep “^d” temp > temp1 ; wc –l temp1 > ~/countDir

A solution using pipes:
ls -l ~ | grep “^d” | wc –l > ~/countDir

Q4 Find out how many times the **mkdir** command been used in the past 100 history lines and list the result in a file called **histCount** in your **home** directory.

A solution:
history 100 > temp ; grep “mkdir” temp > temp1 ; wc –l temp1 > ~/ histCount

A solution using pipes:

history 100 | grep “mkdir” | wc –l > ~/histCount

*NB: In Q3 above, the **ls -l** command output includes an extra line to indicate total size of the directory in blocks. Thus one extra line needs to be subtracted from the line count to get a more accurate number of files. This correction can be ignored for now.*

DISCUSSION

In what way does the use of pipes give an improved solution in the examples above?

The use of pipes results in a shorter command sequence, and more importantly pipes allow us to avoid the use of temporary files; as the use of such files slows down the command execution time and clutters up disk space.

What if you wanted to use a variable to represent a result for some of the above commands?

The **'command substitution'** feature of the bash shell will allow us to do this, using the following syntax: **$(command)**. As an example, for **Q3** above, we could use a variable called **dirNum** to contain the number of directory files, as follows:

```
dirNum=$( ls –l  ~  |  grep "^d"  | wc –l  )
```

Q5 Find the **largest file** in the **home** directory and put the result in a variable called **bigFile**.

The following line will display the size of the largest file as a number on the terminal:

```
ls  -l  ~   |  sort –g –r –k5   |  head –1  |  cut  –d  '  '  –f5
```

Now, we were asked to get this value as a variable, so the following line will take the result of the commands and put that value in the variable **bigFile**, as was requested in the question:

```
bigFile=$(  ls  -l  ~   |  sort –g –r –k5   |  head –1  |  cut  –d  '  '  –f5 )
```

Simply, type **echo $bigFile** to see the value of the variable on the screen**.**

Q6 Assume that some variable **x** has the value **25%.** Write a command to eliminate the **%** character so that the variable can be used as an integer.

A solution:

```
x=$( echo $x  |  sed  ' s/%/ / ' )
```

Example questions: file system statistics using the df command

The **df** command displays information about space usage on the file systems.

NOTE: THERE MAY BE A DELAY AS YOUR FILE SYSTEMS ARE BEING EXAMINED!

Type **df** to see information on the various mounted file systems on your computer.

Type **df .** to show information on the file system for the current directory. A response similar to the following will be seen.

Filesystem	1K-blocks	Used	Available	Use%	Mounted on
/dev/sd3a	58502432	12749248	45753184	22%	/

The report shows the file system (disk partition) sizes in **1kByte** blocks, with column 2 showing the full size, column 3 showing the Used space and column 4 showing the Available space. Column 5 shows the percentage space that is used.

Type **df –h .** to show the space sizes in human readable form, e.g. megabytes, gigabytes etc. The response will be similar to the following:

Filesystem	Size	Used	Avail	Use%	Mounted on
/dev/sd3a	56G	13G	44G	22%	/

Questions:

Provide a single line command answer to the following:

Q. Find and display the size of the file system, in **disk blocks**, where your current directory resides. Your answer will be returned to a variable called **disk_sizeB**.

A solution:
```
disk_sizeB=$( df . | awk ' { print $2 } ' | tail –1 ) ; echo $disk_sizeB
```

Q. Find and display the size of the file system, in disk **human readable** form, where your current directory resides. Your answer will be returned to a variable called **disk_sizeH**.

A solution:
```
disk_sizeH=$( df -h . | awk ' { print $2 } ' | tail –1 ) ; echo $disk_sizeH
```

Q. Find and display the percentage of space used for the file system, where your current directory resides. Your answer will be returned to a variable called **usage**.

A solution:
```
usage=$( df -h . | awk ' { print $5 } ' | tail –1 ) ; echo $usage
```

Example problem

Write a utility that will check the amount of disk space that is available on your disk, and if there is more than 90% of the disk space in use, then send a warning message to the user to advise that the disk is more than 90% full.

A solution

In the example above, we saw how to get a variable, e.g. **usage**, to represent the percentage of space used on the disk, as follows:

usage=$(df -h . | awk ' { print $5 } ' | tail –1)

The problem is that the variable is the form **num%**, but we need a simple integer variable. The following line uses the **sed** utilty (see lab notes) to substitute the % character with a blank space character:

usageNum=$(echo $usage | sed ' s/%/ / ')

The variable **usageNum** now represents an integer value (e.g. 22) to denote the percentage of used disk space.

Now we can write our script program, as follows:

```
#! /bin/bash
# Program to check disk space usage against a specific % limit
#
# DH  29/March/2007

#Check percentage usage
usage=$( df -h . | awk ' { print $5 } ' | tail -1 )

# Use sed to delete the % character
usageNum=$( echo $usage | sed ' s/%/ /' )

# Check usage against the limit (90%)
if (( usageNum >= 90 ))
then
echo "WARNING: your disk is more than 90% full!!!"
else echo "OK - Your disk is not more than 90% full!"
fi

exit
```

NOTE: In the above example, the following two commands could have been piped together, but there were used separately for clarity of teaching and reading of the program:

THESE TWO COMMANDS:
usage=$(df -h . | awk ' { print $5 } ' | tail -1)

usageNum=$(echo $usage | sed ' s/%/ /')

CAN BE COMBINED AS FOLLOWS:

usageNum=$(df -h . | awk ' { print $5 } ' | tail -1 | sed ' s/%/ / ')

Example questions: process statistics using the ps command

Q For all the processes belonging to the user **donal**, list the names of these processes along with their respective **%CPU** utilisations, in a file called **ps_temp1**, in your home directory.

A solution:

Type:

ps au > ps_temp

Now you have **ps_temp** file that contains the following:

```
USER       PID %CPU %MEM   VSZ  RSS TTY      STAT START   TIME COMMAND
root      3302  0.0  0.0  1700  408 tty1     Ss+  Oct10   0:00 /sbin/mingetty tty1
root      3307  0.0  0.0  2996  408 tty2     Ss+  Oct10   0:00 /sbin/mingetty tty2
root      3308  0.0  0.0  2924  408 tty3     Ss+  Oct10   0:00 /sbin/mingetty tty3
root      3309  0.0  0.0  2640  408 tty4     Ss+  Oct10   0:00 /sbin/mingetty tty4
root      3358  0.0  0.0  1492  408 tty5     Ss+  Oct10   0:00 /sbin/mingetty tty5
root      3407  0.0  0.0  1636  408 tty6     Ss+  Oct10   0:00 /sbin/mingetty tty6
donal        23727  0.0  0.0  6136 1424 pts/1    Ss   09:34   0:00 -bash
donal        23818 49.2  0.0  4220  968 pts/1    R    09:49   0:17 /bin/bash ./busy_loop
donal        23824 48.3  0.0  5516  968 pts/1    R    09:50   0:10 /bin/bash ./busy_loop
donal        23826  0.0  0.0  3824  772 pts/1    R+   09:50   0:00 ps –au
```

To list the processes belonging to **donal** , listing only the **command names** and the **%CPU** utilization, you could use the following command:

awk ' /donal / {print $3,"\t", $11}' ps_temp > ~/ps_temp1
(Note , the "/t" is to insert a TAB)

Where the file **ps_temp1** contains:

0.0 -bash
49.2 /bin/bash
48.3 /bin/bash
0.0 ps

You could do all the above in a single command line as follows:

ps au > ps_temp ; awk ' /donal / {print $3,"\t", $11}' ps_temp > ~/ps_temp1

or, shorter by using pipes, as follows:

ps au | awk ' /donal / {print $3,"\t", $11}' > ~/ps_temp1

Example problem
Write a bash script file, called **procs_per_user**, that will display the number of processes for each individual user on a system. Note the **ps aux** (or you can use ps –aux) command will provide a full list of processes.

A Solution:

```
#! /bin/bash
#  Display number of process for each user
#  Script name: procs_per_user       DH 3/April/07

# Make list of all users to file: names
ps aux |  awk ' $1 != "USER" {print $1} ' > names

# Make an unique of users in file: uniq_list
sort names | uniq > uniq_list

# Make simple column titles
echo -e "\nProcs  \t Users \n"

# Loop to read each user name and count number of entries

while  read  xuser
do
x=$( grep "$xuser" names |  wc –l )
echo  -e   "$x  \t  $xuser"
done <  uniq_list

# echo new line and exit
echo  -e  "\n"

# remove any temporary files
rm   names   uniq_list


exit 0
```

//

Example result

```
Procs       Users

1           canna
1           daemon
1           dbus
8           donal
1           gdm
2           higginsm
2           htt
1           jafere
1           nobody
117         root
2           rpc
```

1 rpcuser
1 smmsp

Some observations on the above script program:

1) It is good practice (an essential practice) to always remove temporary files before you finish your script program as seen in the above example.

   1) In the above example the **uniq_list** file was directed into the body of the **while** loop. Another solution, which might be easier to read, is to use a **for** loop as follows:

```
#! /bin/bash
#  Display number of process for each user .. using a for loop
#  Script name: procs_per_user       DH 26/February/2013

# Make list of all users to file: names
ps aux |  awk ' $1 != "USER" {print $1} ' > names

# Make an unique of users in file: uniq_list sort names | uniq > uniq_list

# Make simple column titles
echo -e "\nProcs  \t Users \n"

# Loop to read each user name and count number of entries

for xuser in $(cat uniq_list)

do
x=$( grep "$xuser" names | wc -l )
echo  -e   "$x  \t  $xuser"
done

# echo new line and exit
echo  -e  "\n"

# remove any temporary files
rm   names   uniq_list

exit 0
```

Example problem using a signal

Consider the bash script exhibit program as below.

```
# Exhibit program  D.H. 15/March/2012  ver. 1.0.0
#! /bin/bash

# The main code is here
./progB   &         # start program progB in the background

while true          # a simple loop to simulate real activity
do
  echo "I'm looping"
  sleep 1
done

wait
exit
```

Modify this program so that it will include a **signal trap**. The **trap** will do the following:

iii) Acts on receipt of a **SIGINT** signal

iv) Contains a function called **trap_function()**

iii) The **function** does the following:
- displays (echoes) a simple message to say what is the **PID** for **progB**
- sends a **HUP** signal to the running **progB** program
- properly exits the script program without **orphaning** progB

A solution

SAMPLE ANSWER TO ABOVE

```
#! /bin/bash

trap  'trap_function'  SIGINT

trap_function()
          {
           echo PID for ProgB is "$!"
           kill   -HUP    "$!"
           wait
           exit
          }

# The main code is here
./progB   & # start program progB in the background
while true  # a simple loop to simulate real activity
do
  echo "I'm looping"
```

```
   sleep 1
done
wait
exit
```