GNU/Linux & UNIX Shell Scripts

A shell script is an executable file of commands which the shell program invokes line-by-line.

As the shell works with the standard command line format of: -

($command arg1 arg2 arg3 etc.)

the shell script syntax must be compatible with this too. This tends to make shell scripts very pernickety regarding syntax and less easy to follow for novices.

However the main advantage of shell scripting is that it is very simple to employ and build upon the existing Linux utility commands and input/output redirection.
A very simple script could just be a batch of commands that the user would normally type of the command line.

```
#!/bin/sh
grep fred userlist.log > freduser.txt
cat freduser.txt | updateMyusers
echo "You've been updated" >> freduser.txt
mail fred@example.com  <  freduser.txt
```

The script above does the following:-

1. Searches for a line containing "fred" in the file userlist.log, if found the line is saved in freduser.txt.
2. "cat" sends freduser.txt as input for upDateMyusers to process.
3. The text "You've been updated" is appended to the file.
4. Finally the file is emailed to user fred@example.com

Note that the script above is not robust in that it could not cope with any exceptions to the expected scenario.
For example, what if "fred" did not exist in the log file or it appeared several times? What if the updateMyusers utility failed?
To handle these situations, a script should be able to test conditions and make appropriate choices. This means that, at the very least, control structures must be supported by the shell (e.g. if- then-else).
The precise syntax for these control structures varies from shell to shell.

We will use the Bourne Again Shell (bash) shell as it is widely supported, almost identical to the original Bourne shell and is the "default" shell for Linux distributions.

Note that the first line in the script above is a safety feature.
(#!/bin/sh) ensures that the default /bin/sh shell will always be used to run this script. Execution would fail if an incompatible shell attempted to run a script.

The Bourne Again Shell (bash)

Like other shells, the bash shell provides programmatical features.

- Creation and use of variables
- Testing Conditions
  - File/directory existence, permissions
  - Command success/failure
- Control Structures
  - Decisions (if-then-else, case)
  - Loops (for, while, until)

Shell Variables

Variables defined in scripts can take string or numeric values.

Assignment is done using the "=" (equals) character, note there cannot be spaces around the "=" as the command line would then interpret as a separate argument.

myvar1=wibble	string assignment
myvar2=45	number assignment

In order to use the content of a variable we just prepend a "$" to the variable name, this $ubstitutes the name of the variable with its content.

A simple way to view a variable is by using the "echo" command.

```
# echo $myvar
# wibble
# echo $myvar2
# 45
```

If we try

```
# myvar2=$myvar2+1
# myvar2=$myvar2+1
# echo $myvar2
```

We get the answer "45+1+1" as, unfortunately, this shell cannot do arithmetic.

There are other options for storing values in variables.

A variable value can be read from standard input (i.e. the keyboard).
You should probably prompt the user here.

```
echo "Do you want to continue? (y/n)"
read   userchoice
```

Alternatively "read" could take its input from a file.

```
# read myvar7 < savedvar.txt
```

The variable assignment can be combined with another command using grave accents (back quotes).
For example save the current working directory pathname in a variable for use later. Note that ``` quotes must be used here, not apostrophes.

```
# savepwd=`pwd`
```

Extract a line from a file and save in a variable:-

```
# saveuser=`grep fred userdetail.log`
```

Predefined variables

Some variables contain information about the current configuration and are available to be used by the users' scripts. These variables generally have uppercase names.
The command "set" can be used to view the names and content of all the current shell variables.

Here are some of the most useful pre-defined shell variables:-

```
# echo $USER
fred                The current user's username

# echo $HOME
/home/fred          The current user's home directory pathname

# echo $HOSTNAME
socweb7.napier.ac.uk        The machine hostname

# echo $PWD
/home/fred/csn08101/scripts     The current directory

# echo $SHELL
/bin/bash               The path to the default shell program
```

Editing Scripts

Shell scripts are just text files and can therefore be created and edited using any available text editor. The graphical editor which is included with Puppy Linux is Geany which has enough features and is intuitive to use.

Other fairly common editors which are not graphical but are simple to use are “pico”, “joe” or “nano” one or other of these can usually be found on a Linux distribution.

The one editor which the user can assume will always be present is “vi” (pronounced vee-eye), there is also an improved version called “vim”.

vi the Visual Editor

This is the generic editor; it is not particularly intuitive for new users but is ubiquitous and very powerful.
Because vi is so widely deployed it is worth becoming familiar with the basic functions as an administrator may find that, at some point, it is the only editor available to them.
Unlike many editors vi does not start in text entry (insert) mode.
To enter “insert mode” the user must remember to press “i”.
Once in insert mode any text typed will be entered into the file.
The keyboard cursor keys can be used to move around and the backspace key used to delete characters.

To leave insert-mode the Esc key is pressed and the editor is back in “command mode” where key presses perform particular control actions.
There are a large number of powerful edit commands but the most useful are:-

Inserting Text

i enter insert mode at the cursor location
I insert text at the start of the current line.
A append text to the end of the current line.
o open a new line of text after the current line.
O open a new line of text before the current line.

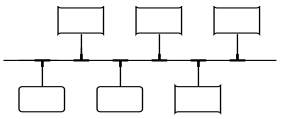
Copy Text

yy copy the current line into the paste buffer

Deleting Text

x delete a character
dw delete a word
dd delete a whole line
D deletes the rest of a line

Each of the above commands may be preceded by a number:
For example “6dd” deletes 6 lines.



Deleted text is “cut”, i.e. it is put into a “paste” buffer.

Paste Text

P insert deleted/copied text before cursor position.
p insert deleted/copied text after cursor.

File commands

:w write (save) to file
:q quit vi
:wq save and quit
:q! quit and abort all changes

Beware of text file differences

Although practically all computer systems use the same ASCII codes to represent characters, there is a subtle difference in the way that UNIX/Linux represents new lines compared to the standard adopted by Windows/DOS.

MS-DOS/Windows

Every line of text ends with a Line-feed (**LF**) and a Carriage-Return (**CR**) character.

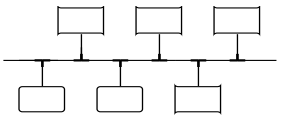
UNIX

Text lines end with a (**CR**) character only.

Therefore if a file is created in Windows Notepad and then transferred to a Linux machine, a Linux utility would be confused by the extra (**LF**) characters at the end of every line. This difference would not be apparent when the file is displayed using “cat” but would cause syntax errors if the file were run as a script.
Similarly a Linux text file, when transferred to a Windows environment, will appear to be lacking the necessary (**LF**) characters and so the line formatting would be lost.

Luckily it is easy to convert a file from UNIX to DOS styles (and vice versa) using the commands: -

dos2UNIX *filename* **UNIX2dos** *filename*



Running a Script

To run a script using a shell you would just supply the scriptname following the reference shell program location.

```
$ /bin/sh scriptname.sh
```

Note that you do not need to use ".sh" when you name a script file but it can serve as a useful reminder of the content of the file.

A more concise alternative for running a script is just to enter the pathname on the command line, like it was any other Linux command.

$ ./scriptname.sh

for the above to function there are a few conditions.

- The script should have #!/bin/sh as its first line so the correct shell will be used to execute.
- The user running the script must have execute permissions.
- The pathname to the script must be used.
  i.e. It is not enough just to enter the script file name, even if it is in the working directory. The example here starts with "./" (dot slash) which says the file is in the current directory.

Script Debugging

It is fairly likely that your first implementation of a script does not work correctly, often due to a syntax error like a missing space or incorrect bracket. These errors cause the shell to misinterpret other parts of the script and therefore the error messages rarely explicitly refer to the actual error location. To avoid this it is best to try and test each command in principle before including it in a script.
To get more detail from a script as it runs the "-v" or "-x" shell options can be used.

```
$ /bin/sh -v mynewscript.sh
```

displays the lines of the script as they are read

```
$ /bin/sh -x mynewscript.sh
```

shows parameter substitutions and displays command lines as they are executed.

Script Arguments

Values can be passed to a script on the command line to modify or detail what it is meant to do.

For example an “archive.sh” script may expect a filename and a pathname to follow the script command.

```
$ /bin/sh archive.sh  myfile.txt /home/demo/work/
```

or just

```
$ ./archive.sh myfile.txt /home/demo/work/
```

These values are referred to as “parameters” or” arguments” and the internal script code will use these in a similar way to variables.

Arguments are referenced as $1, $2 ... $n inside the script.
The name of the script is available as $0 and the number of arguments as $#

If our example script archive.sh contains the lines

```
#!/bins/sh
zip $2/myarchive.zip $1
rm $1
```

then the command

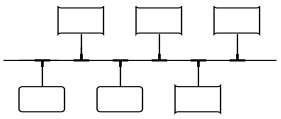
```
$ ./archive.sh myfile.txt /home/demo/backup/
```

Executes the following two commands:-

- **zip /home/demo/**backup**/myarchive.zip myfile.txt**
- **rm myfile.txt**

In this case the argument list when the script runs is:-

$0	**=**	**./archive.sh**	(the scriptname used)
	$1	**=** myfile.txt	(the first argument)
		$2 **=** /home/demo/work/	(the second argument)
$#	**=** 2		(the number of arguments)
$@	**=** myfile.txt **/home/demo/work/**		(the whole argument list)



Flow Control – Loops

In order to control the sequence of commands, bash shell scripts support familiar looping and branching control structures.

The "for – do - done" loop

This allows a script to repeat a set of commands once for each value supplied in an index list.

```
for name in list
do    command
      command
      command
done
```

name is the loop variable and can be any identifier ("i" is often used) and ***list*** is the sequential index list of values that *name* takes on.

Example:

```
for i in davie mike peter mickey
do    mkdir /home/$i
      chown $i /home/$i
      makeuserX $i
done
```

The first time through the loop **$i** is substituted with the text "davie", on the second iteration "mike" and so on until the whole list has been used.
The list of index values that the loop variable uses can be implicitly specified (as above) but more practical options are given below.

The loop index list could be retrieved from:-

a command	`for i in `cat userlist`; do ........`
a variable	`for i in $mylist ; do ........`
a script argument list	`for i in $@ ; do ........`
a directory listing	`for i in work/progs/*.txt ; do ........`

Note the ";" separator is necessary if "do" appears on the same line.

The "if – then – else – fi" Decision Branch

The allows two alternative sets of commands to be run depending on the result of a true/false test, or alternatively one set of commands can be run conditional on the result of a test.

```
if condition
then command
     command
     command

else command
     command
     command
fi
```

```
if condition
then command
     command
     command
fi
```

The types of condition which can be tested are string or numerical equivalence, numerical greater than, less than etc.
The existence and properties of files and directories can also be tested.

Unfortunately just as the shell cannot inherently perform arithmetic operations it also cannot perform comparison operations. Therefore a separate utility is employed to do so.

"**test**" is the utility which evaluates a true/false expression. It supports string, integer and file system operations. It is usually used in conjunction with a control structure.

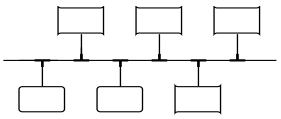
```
$ test $n –eq 3
```

This will result in "success" or "true" if the "n" variable is equal to 3, "error" or "false" otherwise.

In a decision control structure this could look like:-

```
if test $n –eq 3
then command
     command
     command
fi
```

the three conditional commands are only executed if the "n" variable is 3.



Testing "test" conditions

If the "**`test $n –eq 3`**" command above is entered on its own, the user will see no result.

In order to experiment with and assess the function of a test, it would need to be used in conjunction with a conditional instruction.

&& any command following "&&" will only execute if the previous instruction succeeded or gave a "true" result.

|| any command following a "||" will only execute if the previous instruction failed or gave a "false" result.

So the user can experiment with "test" operations by using a command line like: -

```
$ test $n –eq 3 && echo True || echo False
```

The displayed result will either be "True" or "False" depending on the result of the test. This is very useful for verifying your understanding of the syntax used by "test".

As a concession to code readability the keyword "test" can be omitted if the comparison operation is put inside [ ] square brackets. Note that the brackets must be separated by spaces.

```
if [ $n –eq 3 ]
then command
     command
     Command
fi
```

The "test" utility is still used, it is just not explicitly mentioned in the code.

test - Integer Comparisons

-eq	Equal to..
-ne	Not equal to..
-lt	Less than

-le	Less than or equal to..
-gt	Greater than..
-ge	Greater than or equal to

test - Other Common Comparisons

=	Strings are equal `[ $aStr = $bStr ]`
!=	Strings are not equal `[ $aStr != $bStr ]`
-z	String is empty (zero) `[ -z $bStr ]`
-e	Exists `[ -e dustbin ]`
-f	File exists `[ -f myfile.txt ]`
-d	Directory exists `[ -d mydir ]`
-r **-w** **-x**	Read, write or execute permissions set. `[ -r readme.txt ]`
!	Reverse the sense of the comparison (i.e. NOT)

Example: Check if “dustbin” exists, if it doesn’t test that you have write permission and if so create it.

```
if [ ! -e dustbin ]
then
     if [ -w . ]
     then mkdir dustbin
     else echo You do not have write permission.
     fi
else
     echo dustbin already exists
fi
```

Other Loop Control Structures

```
while condition
do
      command
      command
      command
done
```

The commands will only execute if the test condition is true and will continue to execute so long as the condition remains true.

```
until condition
do
      command
      command
      command
done
```

The commands will only execute if the test condition is false and will continue to execute until the condition is true.

Scripts and loops can also be controlled using the special commands below

exit Ends the script immediately

break Terminates a control loop structure early, but carries on running the script.

continue Skips the rest of the loop commands and goes straight to the next iteration of the loop.

In the example below the three commands will be skipped for any file which doesn't have write permission set.

```
for $fname in *.html
do
      if [ ! -w $fname ]
      then continue
      fi
      command
      command
      command
done
```

Bash Shell Arithmetic

Although the shell itself does not handle arithmetic operations, another utility can be used to achieve this.

"expr" takes arithmetic operators and numeric values as input arguments and calculates the result.

Note that since these are command arguments, every operator and operand most be separated by spaces.

```
$ expr 25 + 17
42
```

To calculate a value and assign this to a variable requires the use of "back quote" command substitution.

For example to increment the value of the "x" variable:-

```
x=`expr $x + 1`
```

Explanation

- **$x** is replaced with the current value of "x", say it is 25.
- **expr** calculates the result of 25 + 1 and `expr $x + 1` is replaced with that result, **26**.
- Finally **x=26** is actioned and the new value of x is 26.