Unix Shell scripting

Dr Alun Moon

20th October 2014

Contents

Introduction

Shell scripts in Unix are a very powerfull tool, they form much of the standard system as installed.

What are these good for? So many file and system administration tasks are “easier" using nautilus or the windows explorer or other tools accessible from the graphical desktops.

Notation

In the notes here and in the `man` pages help, a particular notation is used. Examples here are shown without the shell prompt (`$`).

OPTIONAL PARAMETERS that may be used or left out are shown in square brackets [optional]

```
CP(1)                          User Commands                          CP(1)

NAME
       cp - copy files and directories

SYNOPSIS
       cp [OPTION]... [-T] SOURCE DEST
       cp [OPTION]... SOURCE... DIRECTORY
       cp [OPTION]... -t DIRECTORY SOURCE...
```

Figure 1: output from `man cp`

MANDATORY PARAMETERS are parameters that *must* be given to the command. In the `man` pages these are often shown in `CAPITALS`, in these notes I'll put them in angle brackets ⟨`required`⟩,

Spaces

In some places spaces in the bash script are important. Where necessary these will be shown as a ␣ symbol, for example

```
if␣[␣-f␣name
```

To try

Fragments of scripts to illustrate a point at marked with a blue line ones to try yourself are marked with a green line

Editing

A SCRIPT IS JUST A TEXT FILE CONTAINING A SEQUENCE OF COMMANDS. These are performed as if typed from the command line. Royal In Unix systems scripts start with the *shebang* line

```
#!/bin/bash
```

The script is just a text file and can be edited using any text editor[1] [2]. For example: the following script is put in a file `ascript`

[1] Good GUI based ones are `gedit` and `Scite`
[2] A good terminal based one is `nano`

```
#!/bin/bash
date
whoami
pwd
```

Figure 2: ascript

Permissions

THE FILE HAS TO HAVE ITS EXECUTE BIT SET in order for it to be used as a command

```
chmod +x ascript
```

It can now be run like any other unix command

```
ascript
```

Directories and the Working Directory

The concept of the working directory is important when working with files and scripts. The shell can be thought of as being in the *current working directory* commands that operate on files use this as the default directory.

For example `ls` with no parameters lists the files in the working directory.

TO CHANGE THE WORKING DIRECTORY. use the `cd` ⟨*directory name*⟩ command

TO FIND THE CURRENT WORKING DIRECTORY use the `pwd` command. Usually the modern convention is to have the prompt show the working directory[3]. The `pwd` command is useful to use in scripts.

WHEN REFERRING TO FILES it is in relation to the *working directory*[4]. A plain file-name refers to a file in the working directory. A name prefixed with a directory name (`directory/file`) it refers to a file in that sub-directory.

[3] or sometimes the last folder in the working directory

[4] Some editors default to the `Documents` directory. If your script is saved there you will need to `cd Documents` to make this the working directory

Aside on missing files

Try the following in the command shell

```
mkdir folder
cp *.h folder
```

You should get an error as follows

```
shell$ mkdir folder
shell$ cp *.h folder
cp: cannot stat '*.h': No such file or directory
```

The problem is that there are no files ending in a .h, the exact reason for the error is dependent on the way the shell handles the *.h

Command-line parameters

We want to be able to make scripts that can be given parameters, such as a directory name. We can call these like any other command

```
scriptname foldername filename
```

These are accessed using special variables $0...$9

Exercise: 1 Try the following script, take care with the quotes

```
#!/bin/bash
echo "the script name is '$0'"
echo "the first parameter is '$1'"
echo "the second parameter is '$2'"
```

Then use the following parameters running the script from the shell

```
names
names fred
names fred flintstone
```

Note how variables that have not been set are expanded

Parameters can be used in any way

Exercise: 2 Create a script initf

```
#!/bin/bash
mkdir $1/src
mkdir $1/bin
touch $1/makefile
```

Then try it with

```
intif .
mkdir folder
initf folder
initf
```

Question: 1 What is the meaning of the dot . as a directory name?

Question: 2 Without parameters, why does the script fail?

Variables

Shell variables are assigned by simply using their name with an equals sign and a value

```
bash $ foo=Something
bash $ echo $foo
Something
bash $ bar="Some more text (note quotes)"
bash $ echo $bar
Some more text (note quotes)
bash $
```

Note the lack of spaces round the equals sign.

Try

```
foo=This
bar=that
echo $foo and $bar
```

User input

The `read` command can be used to set variables from used input

```
#!/bin/bash
read -p "what is your name?: " first last
echo "Hello $first"
echo "in the phone book you would be \"$last, $first\""
```

Variable substitution

To use a variable prefix it by a dollar sign `$`. The variable name and its dollar are substituted for by the text stored in the variable. If the variable is unset, a blank is substituted.

```
echo :$fred:
fred=some-value
echo :$fred:
unset fred
echo :$fred:
```

Arithmetic

Variable substitution is by the text the variable contains. If we want to perform arithmetic we need a different syntax

```
a=1
b=2
echo $((a+b))
```

By putting the expression inside the `$((...))` several things happen

1. the value of variables is treated as a number
2. arithmetic operations can take place
3. numeric variables do not use the dollar prefix inside the brackets

Conditional Execution

The Unix shell gives us the full range of conditional operations.

LOGIC VALUES in unix are based on the exit status of commands[5]

[5] technically an exit status of 0 is true, non-0 is false

tests

In order to do logic we need to be able to do tests. For this we need a function `test` (more commonly seen as `[ ]`) some common tests are[6]

[6] see `man test`

- `[␣-e␣name␣]` the file called name exists
- `[␣-f␣name␣]` the file name exists and is an ordinary name
- `[␣-d␣name␣]` the file name exists and is a directory
- `[␣"string1"␣=␣"string2"␣]` a string comparison for equality
- `[␣N␣-eq␣M␣]` an integer numeric comparison for equality

Note the need for spaces in the expressions

if...fi

Try the following

```
#!/bin/bash
read -p "Type your name, please: " nm
echo Hello $nm.
echo The ultimate question of life, the universe, and everything...
read -p "Please Type the answer: " ch
if [ "$ch" == "42" ]
then
  echo That\'s right
else
  echo No, the answer is 42
fi
```

Note the spaces in the test and the `if..then..else..fi` pattern. The test is done as a string comparison, note how the variable expansion is in a string quote. This avoids the problem where nothing is entered and the variable expansion is a blank.

We can use these to check that a directory exists before creating it.

```
#!/bin/bash
if [ ! -d $1 ]
then
        mkdir $1
fi
```

Multiple choises

For a multi-way decision the `if... then... elif` exists

```
#!/bin/bash
if [ ]
then

elif [ ]
then

else

fi
```

Repetition – loops

With loops we can perform operations repetitively.

while

We can keep asking a question until a condition is met.

```
#!/bin/bash
read -p "Type your name, please: " nm
echo Hello $nm.
echo The ultimate question of life, the universe, and everything...
read -p "Please Type the answer: " ch
while [ "$ch" != "42" ]
do
read -p "No, try again! " ch
done
echo That\'s right
```

Note that the `do... done` marks the ends of the section of code repeated for each iteration of the loop

shift

The `shift` operation modifies the parameters supplied to the script in `$1`, `$2`, etc.

Try the following

```
#!/bin/bash
n=1
while [ "$1" != "" ]
do
        echo "parameter $((n++)) is $1"
        shift
done
```

Shift is useful for iterating over a list of parameters

```
#!/bin/bash
total=0
while [ "$1" != "" ]
do
        total=$((total+$1))
        shift
done
echo $total
```

for

The other operation useful for loops is the `for`. This takes a list of values and sets a variable to each one in turn.

```
#!/bin/bash
for n in a b c d e f
do
        echo $n
done
```

Backquotes and commands

The list of values may not be know ahead of time. What we want is a way of generating a list of value to put in the code.

We can do this by using file globbing

```
#!/bin/bash
for n in *
do
        echo file is $n
        echo file size is
```

```
        ls -hs $n
done
```

We can use the backquote notation to include the output of a command

Put the following in a text file, call the file `sample.text`

```
this is some standard text
split across two lines
```

Then try

```
#!/bin/bash
for n in `cat sample.text`
do
        echo word is $n
done
```

There is an alternative notation (that I prefer) that uses the characters `$( )`

```
#!/bin/bash
for n in $(cat sample.text)
do
        echo word is $n
done
```

Exercises

Exercise: 3 Write a script to take a list of filenames and write out some information about each[7]

[7] **hint:** take a look at the `file` command to identify the file

Exercise: 4 Write a script to count-down from 10 then write a message "blast-off"

Exercise: 5 Write a script to accept two numbers and an arithmetic operator and do the calculation, displaying the result. The operators could be + - * / %

Exercise: 6 Extend these scripts so that they run repeatedly, until the user opts to exit.

Exercise: 7 Write a script supporting a menu system of your own design