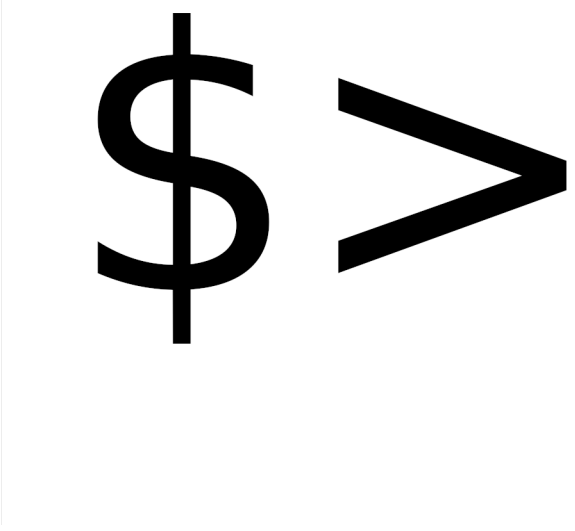
Bash Notes

Basic Bash How-To (A Real language?)

Jonathan Bell



1 Variables

Variables in bash are not strongly typed. You can define variables as such:-

```
$>myvar=123  <-- do not leave spaces
$>myvar="Test" <-- do not leave spaces
```

And then reference them as such:-

```
$>echo "$myvar'
```

Bash also uses special variables. for instance if you run a bash script the arguments can be gained from within a script:

```
$>myscript.sh arg1 arg2
```

The following is a summary of some of the more useful special arguments

```
$> echo $0 <-- returns myscriipt.sh
$> echo $1 <-- returns arg1
$> echo $2 < --returns arg2
$> echo $# <--returns the number of arguments past to the script (2)
$> echo $! <-- pid of the application last put into background mode
$> echo $$ <-- This returns the pid of the shell
$> echo $? <-- This shows the last result from calling an application
```

A common example is to confirm the number of arguments past to a script. Showing an usage example if the number is incorrect.

Example script:-

```
#!/bin/bash

usage()
{
   echo "Wrong number of Arguments given"
   echo "Usage: test.sh arg1 arg2 arg3"
}

if [[ $# != 3 ]]
then
usage;
exit -1;
fi

echo "All is fine";
```

The preceding script shows some basic operations of a bash script. The first line often called the hash bang, tells the shell where to find the interpreter. Note this may be different on your system. It is useful to run the which command to find it.

```
$>which bash
$>/bin/bash
```

The usage() is a function definition in bash. It is often useful to split your code up into sensible blocks.

The if...fi is a decision block. Bash often starts and ends a functional areas with an inverted word such as if,fi or case, esac

The '[[ ]]' or '[]' are decision blocks, which will give different interpretations. Note a common mistake is not to ensure you have spaces within the block.

2 If Comparisons

The 'if' command is often used within bash, it has the following format

```
if [ $value = "y" ]
then
echo "do something"
elif [ $value = "n" ]
then
echo "Do something else"
else
echo "Default action"
fi
```

You can also nest if blocks within other 'if blocks'.

2.1 if Numeric Comparisons

```
echo "Guess a Number 1-3 "
read val

if [ $val -eq 3 ];
 then
echo " you entered 3"
elif [ $val -eq 4 ]; then
...
```

2.2 Using OR in a if comparison

```
if [$age -gt 18 -o $age -lt 30 ]
then
echo "You can come in"
$>fi
```

2.3 Other Numeric comparison

```
-a <- and
-eq <- equal
-ne <- not equal
-lt <- less than
-gt <- greater than
-ge <- greater or equal
 -le less than or equal
```

2.4 String if Comparison

```
  z="Test"
  y="Test"
  if  [ test $y=$z ]
  then
  echo "The Same"
  fi
```

2.5 Using the '[[]]' Method

```
if [[ $z == $y ]]
  then
  echo "Same"
  fi
```

Note with the '[[' use '== ' comparison, and for '[' use the single '=' comparison

2.6 Checking for an empty string

```
  if [ -z $str ]
  then
  echo "String is empty"
  fi
```

2.7 Other if String comparisons

```
-n <- not empty
!= <- not eqaul to
<,> <- alphabetical comparison
 =~  <- Regex check
 if [ $z ~= '^[Te]' ];then
 echo "Starts with Te"
 fi
```

3 Loops

3.1 for

```
for I in {1,2,3,4}
do
echo "loop $i"
done
```

3.2 while

```
x=0
while [ $x -le 4 ];
do
echo "number = $x";
x=$((x + 1))
done
```

3.3 Across a list of files

```
for i in ./*
do
echo "file = $i"
done
```

4 Command Expansion

The () brackets expand the results of an eternal command

```
files=$(ls) <-- expands the list command into the variable files
```

Note you can also use ‘ls‘ back ticks to expand commands, however this is depreciated.

5 Arithmetic Expressions

The double (()) brackets can be used to perform numeric calculations

```
mycal=$((2+3) ) <- aritmetic expansion
```

6 Input and Output

```
read myvar
read -p "Enter a value" myvar
read -sp "Enter a password" mypass <-- hide what the user types
```

7 The case statement

The case statement is often useful when you have multiple choices to make

```
echo "Type a Number"
read val
case $val in
1)
echo "You selected 1"
2)
echo "You selected 2"
*)
echo "Default"
esac
```

8 Useful command from a bash terminal

8.1 history

Bash will like a good shell keep a log of what you type, allowing you to scroll back to a past command Search using Ctrl+s or Ctrl+r

```
$> history <- show a list of commands
$>!59 <- run command 59 again
$>history -w <-- write the history to the .bash_histroy file
$> history -c <-- clear your history
$> history -d 58 < -delete line 58 from history, ueful if you type a
    password in by mistake
```

8.2 aliases

You can create an alias for a command, this is useful if you want to create your own command from a list of commands or command options, the most famous of these being 'll' which is an alias to 'ls -al'

```
$>alias GREPC="find . -type f -name '*.[ch]" -exec grep -iHn $1 \{} \;"
$> GREPC main <-- searches all c files for the word main
```

It should be noted that you can alias an existing command. So for instance you can have an alias of the ls command.

```
$>alias ls="ls -l" <- ALias ls
$>ls <- now uses the alias
$>\ls <- uses the original version
$>alias -p <- see what aliases are available.
```

8.3 Pipes and redirection

Pipe a output from one command to the input of another command

```
  cat file.txt | sed 's/word/Word/g' <- prints the content of file.txt
      passes this to sed to search for word and replace it by Word,
      note if you put -i in the sed command it will change the input
      file.
```

Redirection to a file

```
$>echo "Hello World" > file.txt <- write to file flushing content
$>echo "Hello World Again >> file.txt <- append to existing content
$>gmake 2>&1 result.txt <-Print both standard output and standard error
    to a file
```

8.4 Configuration files

Dependent on the platform, git bash, bash or QNX terminal configuration files are available to store aliases,functions etc for continuous use.

Common files include .bashrc ¡- basic settings

.profile ¡- often for aliases

8.5 Debugging Bash

```
  set -x <- enablle debuf on your terminal
  set +x <-switch off debug
```