Bash shell programming
Part II – Control statements

Deniz Savas and Michael Griffiths

2005-2011

Corporate Information and Computing Services
The University of Sheffield

Email *M.Griffiths@sheffield.ac.uk*
D.Savas@sheffield.ac.uk

CiCS

Topics covered

- Control Structures
  - If constructs
  - For loops
  - While … do & Until … do loops
  - Break & Continue commands
  - Case constructs
  - Select constructs
- Functions

Conditional Statements (if constructs)

The most general form of the if construct is;

```
if command executes successfully
then
        execute command
elif this command executes successfully
then
        execute this command
        and execute this command
else
        execute default command
fi
```

Note that elif and/or else clauses can be omitted.

examples

BASIC EXAMPLE:

```
if date | grep “Fri”
then
echo “It’s Friday!”
fi
```

FULL EXAMPLE:

```
if  [ “$1” == “Monday” ]
then
echo “The typed argument is Monday.”
elif [ “$1” == “Tuesday” ]
 then
echo “Typed argument is Tuesday”
 else
echo “Typed argument is neither Monday nor Tuesday”
fi
```

Note1 : = or == will both work in the test but == is better for readability.
Note2: There must be spaces surrounding = or ==

string comparisons used with test or [[ ]] which is an alias for test and also [ ] which is another acceptable syntax

- string1 = string2 True if strings are identical
- String1 == string2 ...ditto....
- string1 !=string2 True if strings are not identical
- string Return 0 exit status (=true) if string is not null
- -n string Return 0 exit status (=true) if string is not null
- -z string Return 0 exit status (=true) if string is null

IMPORTANT NOTE: [and] must be written separated by spaces on each side. [[and]] must not have a space in the middle.

CiCS

Arithmetic comparison operations used with test and [ ] constructs

- int1 –eq int2 Test identity
- int1 –ne int2 Test inequality
- int1 –lt int2 Less than
- int1 –gt int2 Greater than
- int1 –le int2 Less than or equal
- int1 –ge int2 Greater than or equal

Combining tests using logical operators || (or) and && (and)

Syntax: if cond1 && cond2 || cond3 ...
An alternative form is to use a compound statement using the –a and –o keywords, i.e.
if cond1 –a cond22 –o cond3 ...

Where cond1,2,3 .. Are either commands returning a a value or test conditions of the form [ ] or test ...

Examples:

```
if  date | grep “Fri”  &&  `date +’%H’` -gt 17
then
    echo “It’s Friday, it’s hometime!!!”
fi
if [ “$a” –lt 0 –o “$a” –gt 100 ]   # note the spaces around ] and [
then
    echo “ limits exceeded”
fi
```

Important note: If you like to have a NULL if clause or else clause put a :
in one line.
: implies a statement that does nothing.

A cunning way of using compound logical statements

- || and && conditions can be used to control the execution or otherwise of a command according to the outcome of an earlier command.
- In a compound test such as

  command1 && command2

command2 will only be executed if command1's return-code was SUCCESS (i.e. 0) .

- On the other-hand in the compound statement command1||command2

  command2 will only execute if command1 FAILED

  (i.e have a non-zero return code)

Even more sophisticated control structures can be formed by multiple uses of these conditionals.

Example : command1&&command2&&command3||command4

(who | grep 'fred') && (echo "Hello Freddie" | write fred) || echo 'No fred!'

This will send a message to user fred only if he is logged in.

File enquiry operations

-d file	Test if file is a directory
-f file	Test if file is not a directory
-s file	Test if the file has non zero length
-r file	Test if the file is readable
-w file	Test if the file is writable
-x file	Test if the file is executable
-o file	Test if the file is owned by the user
-e file	Test if the file exists
-z file	Test if the file has zero length

All these conditions return true if satisfied and false otherwise.

Loops

Loop is a block of code that is repeated a number of times.

The repeating is performed either a pre-determined number of times determined by

- a list of items in the loop count (for loops)

or

- until a particular condition is satisfied (while and until loops)

To provide flexibility to the loop constructs there are also two statements namely break and continue are provided.

for loops

Syntax:

for ***arg*** **in** ***list***
do
command(s)
...
done

Where the value of the variable ***arg*** is set to the values provided in the list one at a time and the block of statements executed. This is repeated until the list is exhausted.

Example:

```
for i in 3 2 5 7
do
  echo " $i times 5 is  $((  $i  * 5 ))  "
done
```

CiCS

for loops c-like syntax

Syntax: **for ((index=start ; test ; increment))**
do
:
done

```
EXAMPLE :     for (( i=1 ; i <= maxfiles ; i++ ))
              do
                    cat file${i}
   done

   for (( jj=20 ; jj >= 0 ; jj = jj -2 ))
   do
        echo $jj
   done
```

CiCS

more on for loops

A common trick is to assign to a local variable a list of Items and use that variable as a list driving the for loop.
Example:

```
files=`ls`
for fil in $files
do
  backfil="${fil}.back"
  if [ -f $fil ] &&  ! [ -e $backfil ]
  then
      echo " taking a backup copy of $fil“
        cp -rp $fil $backfil
  fi
done
```

CiCS

while loops

Syntax:

```
while this_command_execute_successfully
do
    :
    :
done
```

EXAMPLE:

```
while test "$i" -gt 0      # can also be coded as while  [ $i > 0 ]
do
i=`expr $i - 1`
done
```

while loops

Example:

```
while
  who > $tmpfile
  grep “$friend”  $tmpfile > /dev/null
do
   ... commands ..
done
```

This loop will repeat until there are no jobs running for the user specified by the variable $friend.

until loops

The syntax and usage is almost identical to the while-loops.

Except that the block is executed until the test condition is satisfied, which is the opposite of the effect of test condition in while loops.

Note: You can think of *until* as equivalent to *not_while*

Syntax:

```
until test
do
   :
      :
done
```

break and **continue** statements

These two statements are associated with the for, while and until loops and provide mechanisms for the abrupt exiting or incrementing of a loop.

The break statement will exit a loop and the execution will continue following the done statement signifying the lexical extent of the loop.

On the other hand the continue statement will terminate the current particular iteration and send the control back to the loop control test statement at the lexical beginning of the loop to tackle the next item on the list.

While, break and **continue** exercises

Exercises.

Study the script named limits9 in the course examples directory.

Re-write it using if-else construct so as to remove the continue statement.

Improve it further to eliminate the break statement.

(selection from a number of possibilities)
Case statements

The case structure compares a string ‘usually contained in a variable’ to one or more patterns and executes a block of code associated with the matching pattern. Matching-tests start with the first pattern and the subsequent patterns are tested only if no match is not found so far.

```
case argument in
    pattern1 ) execute this command
               and this
               and this;;
    pattern2 ) execute this command
               and this
               and this;;
    esac
```

Note : Pattern can be pattern1 | pattern2 | pattern3 … so on to imply the same block for different patterns.

Case construct example

```
case "$1" in
*".txt" , *."doc" ) ls "$1"
     mv "$1"   txt
     echo "$1 moved to txt directory";;
*".tmp") ls "$1"
      mv "$1"  tmp
      echo "$1 moved to tmp directory";;
* )  echo " not a text file or temporary file " ;;
esac
```

Case constructs

The following patterns have special meanings

?) Matches a string with exactly one character.

[[:lower:]]) or [a-z]) matches any lowercase letter

[[:upper:]]) or [A-Z]) matches any uppercase letter

[0-9]) matches any digit

*) matches everything !!!

The last pattern can be used to mean ‘anything-else’

And must be the last pattern on the list.

select constructs

This construct is devised for menu generation.

A list of words representing the allowed choices is presented to the user with each word preceeded by a number. The user is then invited to enter a number to identify the choice. Following user entry a block of code is then executed where actions can be taken depending on the choice.

Select construct

select *word* in *list*
do
:
:
:
done

An example of using select

```
prefer='vegetarian vegan no_nuts halal kosher no_gluton quit'
select type in $prefer ;
do
 case $type in
    "veg"* ) food=1 ; break ;;
    "halal"|"kosher" ) food=2 ; break ;;
    "no_nuts" ) food=3 ; break ;;
    "no_gluton" ) food=3 ; break ;;
    "quit" ) food=0 ; break;;
 esac
done
```

Functions

- Functions are a way of grouping together commands so that they can later be executed via a single reference to their name. If the same set of instructions have to be repeated in more than one part of the code, this will save a lot of coding and also reduce possibility of typing errors.

```
functionname()
 {
     block of commands
 }
```

Advise: Put both curly brackets on a line on their own as shown here to avoid possible errors.

Functions

- Functions are like mini-scripts. But they run in the same shell environment as the containing script.
- Therefore all the locally declared variables in a shell script are also accessible and available in the functions it contains.
- However, the positional ‘command-line’ parameters passed to a script are not available to its containing functions.
- This is because the functions can take on their own positional parameters as they are invoked within the script.
- So, if you want to use the positional parameters of a script in a function within the script just pass them as parameters:

  Example: myinfunction $* or myinfunction $3 $1 ...
- Use functions for use re-usability and readability

References:

Follow the links from the research computing pages at Sheffield University to various documentation on shell script programming.

http://www.shef.ac.uk/wrgrid/documents/links.html

http://uspace.shef.ac.uk/clearspace/groups/iceberg