CMU MSP 36602: Intro to UNIX and Scripting: Part 2

H. Seltman Feb 13, 2019

I. Shell Scripting with bash

a. A **shell script** can carry out a simple set of commands, or it can be a complex program with complex inputs and outputs.

b. Here is a ***simple shell script*** called bscript:

```
#!/bin/bash
# Simple info script (no arguments)
# Shows home, wd, text files, and files starting with 'upper'
echo Your home is $HOME
echo You are now in $PWD
echo Here are the text files:
ls *.txt
echo Here are the last 2 files starting with \'upper\'
ls -l upper* | tail -n2
```

Sample use:

```
bash-3.1$ bash bscript
bash-3.1$ chmod u+x bscript
bash-3.1$ bscript     [or ./bscript if wording dir. not on path]
Your home is /TOIL-U3/hseltman
You are now in /TOIL-U3/hseltman/=A602/Hadoop
Here are the text files:
abc.txt  CarlEmail.txt  counts.txt  two.txt
Here are the last 2 files starting with 'upper'
-rw-r--r-- 1 hseltman users  626 Mar 15 17:24 upper.c
-rw-r--r-- 1 hseltman users 1128 Mar 16 17:09 upper.o
```

c. Enhancement #1: inputs as ***numbered parameters*** (bscript1):

```
#!/bin/bash
# Usage: bscript1 startingFileNameLetters
# Show the script name and the last 2 files
# starting with 'startingFileNameLetters'.

echo bscript1 was called with $# parameter\(s\)

echo The current program is $0.
echo Here are the last 2 files starting with \'$1\'
ls -l "$1"* | tail -n2
```

Sample use:

```
bash-3.1$ bscript1 bscr
bscript was called with 1 parameter(s)
The current program is ./bscript1.
Here are the last 2 files starting with 'bscr'
-rwxr--r-- 1 hseltman users 179 Mar 17 08:20 bscript
-rwxr--r-- 1 hseltman users 117 Mar 17 08:45 bscript1
```

Details:

The zeroth parameter (`$0`) when calling a script is the name or the script. The other parameters are $1, $2, ….

The syntax "$#" counts the parameters past the script name.

d. Enhancement #2: ***if statements*** (bscript2):

```
#!/bin/bash
# Usage: bscript2 startingFileNameLetters maxFiles
# Shows last 'maxFiles' starting with 'startingFileNameLetters'

# Assure valid parameter count
if [[ $# -lt 1 || $# -gt 2 ]]
then
  echo Usage: $0 startingFileNameLetters [maxFiles=3]
  exit 1
fi

# Default 'maxFiles' to 3 and assure it is positive
if [[ $# -eq 1 ]]; then
  maxFiles=3
elif [[ ! $2 =~ ^[+-]?[0-9]+$ ]]; then
  echo Usage: bscript2 startingFileNameLetters [maxFile=3]
  echo where \'maxFiles\' is a number greater than 0
  exit 1
elif [[ $2 -lt 1 ]]; then
  echo $2 is an invalid file count
  exit 1
else
  maxFiles=$2
fi

# Assure 'startingFileNameLetters' is provided
if [[ -z "$1" ]]; then
  echo Usage: bscript2 startingFileNameLetters maxFiles
  exit 1
fi

echo Here are the last $maxFiles files starting with \'$1\'
ls -l "$1"* | tail -n$maxFiles
```

Sample use:

```
bash-3.1$ bscript2
Usage: bscript2 startingFileNameLetters maxFiles
bash-3.1$ bscript2 up per
Usage: bscript2 startingFileNameLetters [maxFile=3]
where 'maxFiles' is a number greater than 0
bash-3.1$ bscript2 up 0
0 is an invalid file count
bash-3.1$ bscript2 up 1
Here are the last 1 files starting with 'up'
-rw-r--r-- 1 hseltman users  626 Mar 15 17:24 upper.o
bash-3.1$ bscript2 up
Here are the last 3 files starting with 'up'
-rwxr-xr-x 1 hseltman users 5172 Mar 16 17:14 upper
-rw-r--r-- 1 hseltman users  626 Mar 15 17:24 upper.c
-rw-r--r-- 1 hseltman users 1128 Mar 16 17:09 upper.o
```

Details

i. Reference: http://www.gnu.org/software/bash/manual/html_node/Bash-Conditional-Expressions.html

ii. "`[[ ]]`" is a "test". (For this class ***do not use the the old style*** with single brackets which is less safe and less powerful.) A space after "`[[`" and a space before "`]]`" are *required*.

iii. Here are a list of some of the unary operators for inside the test. ***A space between the operator and its argument is required***.

   1. `-n`: argument is not of zero length (i.e., not blank)
   2. `-z`: argument is of zero length
   3. `-d`: argument is an existing directory
   4. `-f`: argument is an existing file
   5. `-r`, `-w`, and `-x` test if the argument is an existing file and is ***readable, writeable, or executable***.

iv. Here are a list of some of the binary operators for inside the test. ***Spaces on both sides of the operator are required***.

   1. = (or ==), `!=`, <, and > test ***strings*** (equal, unequal, sorts below, sorts above)
   2. `-eq`, `-ne`, `-le`, `-ge`, `-lt`, `-gt` test ***integers***
   3. =~ does ***regular expression testing*** (re on the right)
   4. -nt and -ot require file arguments and ***test if the file*** on the left ***is newer*** (-nt) ***or older*** (-ot) than the one on the right

v. ***Multiple tests can be separated by || (or), or && (and), or preceded by ! (not)***, and can have parentheses. These operators are greedy, so the second part might never be evaluated.

vi. "then" must be on a separate line (or use `if [[ myTest ]]; then`).

vii. Note that `if` is ended by `fi`, which is "if" spelled backwards.

viii. Linux programs and scripts have an **exit code**, which may be examined by the calling program. By convention, "0" is OK. It is good practice to provide a non-zero exit code when an error occurs.

ix. The syntax `myVar=${otherVar:-defaultValue}` will set "myVar" to the value $otherVar (either a name or the number of a positional parameter) if it exists. But, if it does not exist, "myVar" is set to "***defaultValue***".

e. ***Shell script debugging***: change "#!/bin/bash" to "#!/bin/bash -x" to see more detail of what is happening when you run your script.

f. Enhancement #3: ***Looping*** (bscript3)

```
#!/bin/bash
# Check which .c files are missing .o files
# If .o is present, but .c is newer, recompile
# and warn if failed.

# basic numeric looping
for i in {2..10}; do
  echo $i squared is $(( i*i ))
done
echo

# check all .c files
for f in *.c; do
  basename=${f%.*}
  if [[ -f ${basename}.o ]]; then
    echo $basename.c has been compiled to ${basename}.o
    if [[ ${basename}.c -nt ${basename}.o ]]; then
      echo ... but recompiling because .c is newer than .o
      gcc -c "${basename}".c
      if [[ $? -ne 0 ]]; then
        echo Note: compile failed
      fi
    fi
  else
    echo ${basename}.c is being compiled to ${basename}.o
    gcc -c "${basename}".c
    if [[ $? -ne 0 ]]; then
      echo Note: compile failed
    fi
  fi
done
```

Sample use:

```
bash-3.1$ touch nothing
bash-3.1$ bscript3
2 squared is 4
...
10 squared is 100

nothing.c has no corresponding nothing.o
upper.c has been compiled to upper.o

bash-3.1$ touch upper.c
bash-3.1$ bscript3
2 squared is 4
...
10 squared is 100
nothing.c has no corresponding nothing.o
upper.c has been compiled to upper.o
```

Details:

i. Note that the syntax is:

```
for var in list
do
  code lines
done
```

therefore `for var in list; do` may be convenient

ii. {#..#} makes a sequence.

iii. $((expression)) evaluates an integer arithmetic expression (adding the $ signs)

iv. `${f%.*}`, where "f" is a shell variable name, drops an extension, while `${f##*.}` gets the file extension. [% does match at the end and remove, while # does match at the beginning and remove. Single is shortest, double longest.]

These are examples of "parameter expansion" (see http://mywiki.wooledge.org/BashGuide/Parameters#Parameter_Expansion)

v. A file **glob** is a doubly-anchored "wildcard" pattern with special characters ?, [...], and * (first 2 match 1 char). A glob can be the "list" needed in a "for" loop.

vi. $? is useful for checking the most recent exit code.

vii. Loops can be nested (based on syntax, not indenting)

viii. Also available: `while [[ expression ]]; do; codelines; done`

g. **`more`** and **`less`**: The utility program more is typically used to aid in viewing long output, e. g., more `myLongFile`, or `cat myLongFile | more`, show only the first screen of the file. Then press spacebar to see more or 'q' to quit.

Even nicer is less: `man bash | less` will show the first screen of the bas manual. Then use commands like "f" to go forward one screen, "b" to go back one screen, up-arrow to go back one line, down-arrow or Enter to go down one line, "/myREPattern" to go to the next match of regular expression "myREPattern" in the text, "n" to go to the next match of the previous pattern, "p" to go to the previous match, "q" to quit, and much more.

h. **Control characters**: Here are a few useful bash control characters you should know:

i. ctrl-c cancels whatever is going on (e. g., an infinite loop)

ii. ctrl-a moves the cursor to the beginning of the line

iii. ctrl-e moves the cursor to the end of the line

iv. ctrl-s suspends output scrolling by on your screen, e.g., a very long listing such as "ls -R /".

v. ctrl-q restarts after suspending with ctrl-s. ***If your terminal ever appears to be unresponsive***, often that is due to accidentally typing ctrl-s, so try ctrl-q to get back to normal. If you want to quit after ctrl-s, use ctrl-c then ctrl-q.

i. The **directory stack** keeps track of your working directories:

```
pwd
pushd .  # save where I am working
cd /usr/tmp
ls
dirs  # show the directory stack
popd   # return to the last directory on the stack
```

j. **Get user input** with `read`.

```
read –p "Say something> " something
echo "You said $something"
```

k. **Command substitution** is the way to get the output of a command into a variable or to use it as the argument of another command

   i. The old form is "back ticks". The new, highly preferred form is $().

   ii. Examples:

```
echo one two three > 123.txt
counts=`wc -l 123.txt`  # deprecated
echo $counts

cat upper.c | grep len | wc -l
rslt=$(cat upper.c | grep len | wc -l)
echo $rslt
```

l. **List constructs** are commonly used to ***replace complex if statements***. There are two types: and (&&) and or (||). In an "and list", several commands are carried out but the rest of the list is not carried out if one fails. In an "or list", several command are carried out, stopping at the first successful command. Think of this as following "greedy evaluation" rules.

Here is an example that sets the variable "opts" from either a file called "opts.txt" in the working directory or a file called "opt.txt" in the home directory, or it is set to a default.

```
opts=
[[ -f opts.txt ]] && opts=$(cat opts.txt)
[[ -z $opts ]] && [[ -f ~/opts.txt ]] && opts=$(cat ~/opts.txt)
[[ -z $opts ]] && echo "default options set" && opts="a=0;b=0"
echo "opts = " $opts
```

You can also group statements with {} and ";" so that, e.g., the final clause prints a message and exits with a failure code.

m. More on redirection

i. Each shell open three special files: stdin (input), stdout (standard output) and stderr (error output) and initially directs these to your terminal (keyboard and screen). Many programs read and write exclusively or optionally (when no file(s) are specified) to these three files. Such programs are often called "filters". By convention, programs should send normal output to stdout, and error messages to stderr.

ii. The numbers 0 to 2 are assigned to stdin, stdout, and stderr respectively. The redirection operators "<" and ">" are actually shortcuts for "0<" and "1>" respectively. You can use "2>myFile" to direct any error output to a file. To direct both error and standard output to a file, you need to use the special syntax ">myStdOutput 2>&1" which can be read as "direct stdout to 'myStdOutput' (as file 1) and direct stderr to the 'new' (as specified by &) file 1."

iii. A common example is **hiding errors**, e.g., `rm nonExistantFile 2> /dev/null`, which uses the "bit bucket" to discard whatever is sent to it.

iv. Another example is `ls -R /* 2>/dev/null | grep 36602`. Without the redirection, we would see many "permission denied" messages. (Note that a better way to do this task is: `find / -type f –iname *36602* 2>/dev/null`.)

n. Functions:

i. Creation syntax: Start with `functionName() {`. Then enter the bash code lines, using $1, etc. for function arguments. End with `}`.

ii. Use syntax: `functionName( [arg1 [arg2 […]]])`

iii. This works in a script file or outside.

iv. Example:

```
k() {
( rm $1 2>/dev/null && echo OK ) || echo 'not OK'
}
k existingFile  # deletes it and says 'OK'
k missingFile # says 'not OK'
```

o. Learning more:

See http://www.tldp.org/LDP/abs/abs-guide.pdf for advance scripting information.

There is a great guide at http://mywiki.wooledge.org/BashGuide. At least check out http://mywiki.wooledge.org/BashGuide/Practices.