COMP 2718: Shell Scripts: Part 2

By: Dr. Andrew Vardy

Outline

- Control Operators
- Shell Functions
- Local Variables
- `read` - Read Values form Standard Input
- IFS
- Loops
- `break`
- `continue`
- Reading Files within Loops

Control Operators

The following **control operators** can achieve branching without an `if` statement:

```
command1 && command2
```

and

```
command1 || command2
```

With `&&` `command1` is executed and only if it succeeds (status 0) is `command2` executed.

With `||` `command1` is executed and only if it fails (status 1-255) is `command2` executed.

For example, the following makes a `temp` directory and if that is successfull, changes into it:

```
$ mkdir temp && cd temp
```

The following tests for the existence of a `temp` directory and if that test fails (i.e. `temp`) doesn't exist then it creates it.

```
$ [ -d temp ] || mkdir temp
```

Something like this is very useful in scripts that require some precondition. For example, if we want to quit if the file `input.dat` doesn't exist, we could do the following:

```
[ -e input.dat ] || exit 1
```

Shell Functions

Structured programming involves breaking your problem down to smaller pieces. One of the main mechanisms to support structured programming are functions.

Shell functions are embedded into scripts and act like standalone programs. Theey have the following form:

```
[function] name {
    commands
    return
}
```

The word ‘function’ can be ommitted.

Here is a minimal example:

```
function funct {
    echo "Step 2"
    return
}

echo "Step 1"
funct
echo "Step 3"
```

Note that a function has to be defined prior to its being used.

Local Variables

In the scripts we have seen so far, all variables have been **global**. It is often useful to use variables that are purely local to a function. Otherwise, there is the possibility of name conflicts with global variables and with unintentional side effects.

We define a **local variable** as follows:

```
local name[=value]
```

The variable's scope is limited to the current function and any functions it may call. Lets see an example...

```
foo=0 # global variable foo

funct_1 () {
    local foo # variable foo local to funct_1
    foo=1
    echo "funct_1: foo = $foo"
}
funct_2 () {
    local foo # variable foo local to funct_2
    foo=2
    echo "funct_2: foo = $foo"
}

echo "global:  foo = $foo"
funct_1
echo "global:  foo = $foo"
funct_2
echo "global:  foo = $foo"
```

Example: check_maxlines

Lets see an example function that illustrates many of the scripting concepts discussed so far:

- Functions
- Local variables
- Branching and test statement usage

This script will also introduce the use of arguments within a function. Once again, the arguments are denoted $1, $2, ...

This script's job is to check whether the number of lines within a file ($1) is greater than a certain number ($2)...

```
function check_maxlines {
    local file=$1
    local n=$2
    if [ ! -e $file ]; then
        echo "$file missing."
        return
    fi

    count=$(wc -l < $file)
    if [ $count -ge $n ]; then
        echo "$file has too many lines: $count"
    else
        echo "$file has less than $n lines"
    fi
}

# Example usage
# check_maxlines /tmp/foo.txt 100
```

In order to get a count of the number of lines in a file the `check_maxlines` function does the following:

```
count=$(wc -l < $file)
```

Why redirect the file to standard input? Try running the following:

```
$ wc -l ~/.profile
22 /home/av/.profile
```

The behaviour of `wc` is to print the number of lines and the name of the file. By redirecting from standard input there is no filename to print and the number stands on its own!

```
$ wc -l < ~/.profile
22
```

We can execute this script as usual: `./check_maxlines.sh`.

We can also make the function available by putting it in `~/.bashrc`. Once a change is made to `~/.bashrc` you either have to start up a new shell, or use `source` to re-run `~/.bashrc`:

```
source ~/.bashrc
```

We can also remove the final line from `check_maxlines.sh` and make it a pure function. Then we can `source` `check_maxlines.sh` and run the function on the command-line as follows:

```
check_maxlines /tmp/foo.txt 100
```

Notice there is no `.sh` at the end. We are running the function, not the script.

`read` - Read Values form Standard Input

Many scripts need the ability to read from standard input. The `read` builtin achieves this. By default, `read` reads line-by-line from stdin:

```
read [-options] [variable...]
```

It reads from standard input into one or more variables. If no variables are specified it reads into a shell variable called `REPLY`. One basic workflow is to use `read` to get values from a user:

```
$ echo "Please enter an integer: " ; read int

Please enter an integer:
56

$ echo $int
56
```

The following script reads multiple values:

```
-----------------------------------------------
echo -n "Enter one or more values > "
read var1 var2 var3
echo "var1 = '$var1'"
echo "var2 = '$var2'"
echo "var3 = '$var3'"
-----------------------------------------------
```

Run this to see different cases such as when less than 3 or more than 3 values are provided.

The following are some options to `read`:

Option	Description
`-a` *`array`*	Assign the input to *array*, starting with index zero. We will cover arrays in Chapter 35.
`-d` *`delimiter`*	The first character in the string *delimiter* is used to indicate end of input, rather than a newline character.
`-e`	Use Readline to handle input. This permits input editing in the same manner as the command line.
`-i` *`string`*	Use *string* as a default reply if the user simply presses Enter. Requires the `-e` option.
`-n` *`num`*	Read *num* characters of input, rather than an entire line.
`-p` *`prompt`*	Display a prompt for input using the string *prompt*.

`-r`	Raw mode. Do not interpret backslash characters as escapes.
`-s`	Silent mode. Do not echo characters to the display as they are typed. This is useful when inputting passwords and other confidential information.
`-t` *seconds*	Timeout. Terminate input after *seconds*. `read` returns a non-zero exit status if an input times out.
`-u` *fd*	Use input from file descriptor *fd*, rather than standard input.

The following example combines these options:

`-p prompt` Displays prompt string
`-s` Silent mode
`-t seconds` Terminate after timeout (with non-zero status)

```
-----------------------------------------------
if read -t 10 -sp "Enter passphrase > " secret_pass; then
    echo -e "\nSecret passphrase = '$secret_pass'"
else
    echo -e "\nInput timed out" >&2
    exit 1
fi
-----------------------------------------------
```

IFS

How does `read` partition a line of text? It uses a variable called `IFS` (Internal Field Separator). By default, `IFS` consists of a space, a tab, and a newline character. So any of these can be used as separators.

But maybe you want to use ‘,’ as a separator. If so, do the following:

```
$ IFS=','  read a b c
```

This looks a bit weird, we have a variable assignment followed by a command. The purpose is to assign `IFS` to this custom value for only the command that follows. Effectively, it is a temporary setting of `IFS` as illustrated below...

```
$ read a b <<< "Alpha Beta"

$ echo "a: $a, b: $b"
a: Alpha, b: Beta
```

Notice the operator <<<. This is called a **here string**. It simply takes the given string ("Alpha Beta") and feeds it to the command (`read`) as standard input. Lets continue by changing `IFS` and reading a comma-separated string:

```
$ IFS=','
$ read a b <<< "Alpha,Beta"
$ echo "a: $a, b: $b"
a: Alpha, b: Beta
```

That worked nicely, but the effect of setting `IFS` is lasting. If we go back to reading whitespace-separated strings we will have a problem:

```
$ read a b <<< "Alpha Beta"
$ echo "a: $a, b: $b"
a: Alpha Beta, b:
```

Loops

We will look at `while` and `until` loops, as well as the `continue` and `break` statements. Lets start with the syntax of a `while` loop:

```
while condition_commands; do commands; done
```

Similar to `if` the `condition_commands` are executed and if successfull (exit status 0) then the loop is entered and `commands` are executed. This cyle will repeat until `condition_commands` yield a non-zero exit status.

Here's a simple example that just counts to 5:

```
count=1
while [ $count -le 5 ]; do
    echo $count
    count=$(($count + 1))
done
echo "Finished."
```

break

It is often necessary to break out of a loop. The `break` command immediately terminates the loop, passing control to the next statement after the loop. The following is a refactoring of our count-to-5 script:

```
count=1
while true; do
    if [ $count -gt 5 ]; then
        break
    fi
    echo $count
    count=$(($count + 1))
done
echo "Finished."
```

Notice that we had in addition to moving the test, we also had to change it from `-le` to `-gt`.

continue

The `continue` command skips the remainder of the loop's body and proceeds with the next iteration. The following example script prints the even numbers less than or equal to 10:

```
i=1
while [ $i -le 10 ]; do
    if [ $(($i % 2)) -ne 0 ]; then
        # Increment count, then
        i=$(($i + 1))
        continue
    fi
    echo $i
    i=$(($i + 1))
done
```

Note: This script could easily be shortened.

Reading Files within Loops

A `while` loop can process standard input. This provides a mechanism to read from a file if we redirect standard input to come from a particular file. The following example illustrates this:

```
-----------------------------------------------
while read line ; do
    if [ -z "$line" ] ; then
        continue    # Skip blank lines
    fi
    if [ "$line" == "stop" ] ; then
        break       # Exit the loop
    fi
    echo $line
done < input.txt
-----------------------------------------------
```

If content of `input.txt` is as follows:

```
asdf
qwerty

blah
blah
stop
still going?
nope
```

The script's output is the following:

```
asdf
qwerty
blah
blah
```