Chapter 1

Shell Scripting

1.1 Understanding Shell Scripting Core Elements

To help understand the components of the shell scripts, we're going to look at a couple of them. The first one is:

```
1 #!/bin/bash
2 #
3 # This is a script that greets the world
4 # Usage : ./hello
5
6 clear
7 echo Hello, World!
8
9 exit 0
```

The first line of the script `#!/bin/bash` is called the **shebang**. The shebang defines which program is going to execute the script that we're writing, and since we're writing bash scripts here, we put the location of the bash binary executable here on the first line. This is *extremely* important in Linux since many shells such as ksh, tcsh, zsh, etc., share a somewhat common syntax with bash, and as such unexpected errors may occur if a wrong shell is used!

The next few lines (*lines 2-4*) is a comment. It's a good idea to include comments in shell scripts, since bash has a tendency to seem cryptic at first sight, and the comments make the goal of the program easier to understand, and thus the code more readable.

The next few lines are self-explanatory, where the script clears the shell and then echoes (prints) *Hello, World!* to the terminal. Finally, the program exits with an **exit code of 0**. While this line is *not* required here, it's important in certain places. Every program in Linux tells it's parent shell if the operation it tried to perform was successful while exiting via the means of an exit code. An exit code of **0 = success** while anything else means failure. This gives us the opportunity to debug our programs via custom exit codes that indicate what a problem is in the script.

The exit code of the last command can be viewed both in the script or in the shell by:

```
1 $ chmod u+x hello
2 $ ./hello
3 ...
4 Hello, world!
```

```
$ echo $?
0
```

On line 1 we have to give the script executable permissions since otherwise bash won't be able to execute it! Then, we execute our script, which executes with an exit code of 0, which we can verify using `echo $?`.

1.2 Using Variables

1.2.1 Setting and getting the value of a variable

To set a variable to a certain value, we use the syntax: `var=<valueToSet>`, while anywhere that the value of variable is required to be output, we use the variable name with a `$` in front of it, `<something>=$var`. Thus, to assign the value of `var1` to `var2`, the code is:

```
var2=$var1
```

1.2.2 if-else flow control

In bash each `if` code block eventually ends with a corresponding `fi`. In between there can be multiple `elif` checks and finally an `else`, although the last two aren't compulsory.

Conditions are checked using the `test` command. Quite often, we use a shorthand notation for the test command: put the test between square brackets, i.e., `[ <testCriteria ]`. The other way would be to write `test <testCriteria>`. This command has several options that check for different criteria. For example, `test -z` checks to see if a variable is empty, `test -d` checks to see if a folder exists in the current directory with a name same as that of the content of the variable, etc. A list of all possible tests are documented in the manpage for test, accessible with: `man 1 test`.

1.2.3 Example program

Let us consider the following program that prints the first command-line argument if present, or prompts the user for one:

```
#!/bin/bash

if [ -z $1 ]; then
    echo "Enter a name:"
    read NAME
else
    NAME=$1
fi

echo "The name you entered is: $NAME"
```

The command line arguments are represnted by a variable corresponding to their position. So, `$1` is the first argument, `$2` the second and so on. `$0` is the name of the program/script itself!

On line 4 we check if a name has been provided in the form of the first command line argument by checking if the `$1` variable is empty or not. The `test -z` command succeeds if the variable is empty (i.e., if a name hasn't been provided). In that case, it asks for a value. Otherwise it prints the value of the command line argument. The read command stores input from `stdin` and stores it in the variable provided (here, *NAME*).

The output of the script is:

```
$ ./ex2
Enter a name:
Sam
The name you entered is: Sam
$ ./ex2 Dean
The name you entered is: Dean
```

1.3 Using Positional Parameters

Anything that's provided to a script as an argument becomes a positional parameter. For example, in the command `ls -l /etc`, the first positional parameter, i.e., `$1` is the value `-l` while the second parameter `$2` is `/etc`. One **wrong** way to use positional parameters in a script would be:

```
#!/bin/bash

echo parameter 1: $1
echo parameter 2: $2
echo parameter 3: $3
```

This script works as expected when 3 positional arguments are provided:

```
$ ./ex3 a b c
parameter 1: a
parameter 2: b
parameter 3: c
$ ./ex3 a b c d e f
parameter 1: a
parameter 2: b
parameter 3: c
```

In the second case, when more than 3 arguments are given, the ones after the 3^{rd} argument are simply ignored. But when the number of arguments is just 1, the second and the third line are executed anyway:

```
$ ./ex3 a
parameter 1: a
parameter 2:
parameter 3:
```

We should intelligently find out the number of arguments provided to a script and then treat them accordingly. For this, we need a mechanism like a for loop, which iterates over the contents of an item. So, the script becomes:

```
#!/bin/bash
count=1
```

```
for i in "$@";
do
    echo "parameter $count : $i"
    ((count++))
done
```

The count variable is used to keep track of the position of the argument being displayed, while the `$@` variable is actually an array (i.e., a variable that stores multiple values of the same type) which contains all of the positional arguments. The final line, `((count++))` increments the value of count and is called an *arithmetic expression* in bash. Such expressions must always occur inside the double brackets to be evaluated. The output of this script is:

```
$ ./ex4 a b c d e f
parameter 1 : a
parameter 2 : b
parameter 3 : c
parameter 4 : d
parameter 5 : e
parameter 6 : f
$ ./ex4 a
parameter 1 : a
$ ./ex4
$
```

1.4 Understanding if then else

Let us consider another script that checks if the passed argument is a file or a directory:

```
#!/bin/bash
# Run this script with one argument.
# Tells whether the passed argument is a file or a directory.

if [ -f $1 ]
then
    echo "$1 is a file"
elif [ -d $1 ]
then
    echo "$1 is a directory"
else
    echo "I don't know what \$1 is!"
fi
```

The output of the above script for different arguments is:

```
$ ./ex5 none
I don't know what $1 is
$ ./ex5 Desktop
Desktop is a directory
$ ./ex5 ex3
ex3 is a file
```

1.5 Understanding for

A for loop just keeps executing a bunch of statements or commands as long as some condition is true. Let us consider the following script:

```
#!/bin/bash

for ((counter=10; counter>=1; counter--)); do
    echo $counter;
done
```

It has the output:

```
$ ./ex6
10
9
8
7
6
5
4
3
2
1
```

This is another type of for loop that executes as long as a certain condition is true, unlike the last time we used the for loop where it executed as long as there were elements in the array `$@`.

1.5.1 For loop on the command line

Let us consider we have a bunch of hosts with the Network ID 10.0.99./24 and host IDs: `.11`, `.12` and `.99`. If we want to ping them all one after the other and check if they're up, we could do:

```
$ for hid in 11 12 99; do ping -c 1 10.0.99.$hid; done
PING 10.0.99.11 (10.0.99.11) 56(84) bytes of data.
64 bytes from 10.0.99.11: icmp_seq=1 ttl=64 time=0.314 ms

--- 10.0.99.11 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.314/0.314/0.314/0.000 ms
PING 10.0.99.12 (10.0.99.12) 56(84) bytes of data.
64 bytes from 10.0.99.12: icmp_seq=1 ttl=64 time=0.399 ms

--- 10.0.99.12 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.399/0.399/0.399/0.000 ms
PING 10.0.99.99 (10.0.99.99) 56(84) bytes of data.
From 10.0.99.11 icmp_seq=1 Destination Host Unreachable

--- 10.0.99.99 ping statistics ---
1 packets transmitted, 0 received, +1 errors, 100% packet loss, time 0ms
```

Let us now modify this script to only show an error when a host is down:

```
$ for hid in 11 12 99; do ping -c 1 10.0.99.$hid > /dev/null || echo "10.0.99.$hid is
↪  down..."; done
10.0.99.99 is down...
```

The normal output is sent to /dev/null to discard the results of the ping. If however the host is down, the `ping -c 10.0.99.$hid` command exits with an exit code *not equal* to zero, i.e., error. This indicates that the host is down and couldn't reply. This is when our script prints that the host is down.

The double pipe represents a or statement where the stuff on the right of the double pipe is executed if the entirety of the command on the left, (previous command) fails, i.e., exits with a non-zero value.

1.6 Understanding while and until

Both the `while` loop and the `until` loop work in a similar fashion – *while* keeps the loop running as long as a condition is true, and *until* keeps the loop running as long as a condition is false, i.e., until the condition is met.

1.6.1 While example

The following script shows us an example of the utility of the while loop:

```
#!/bin/bash
#
# Usage : monitor <processName>

while ps aux | grep top | grep -vE "(bash|grep)" > /dev/tty11
do
    sleep 1
done

clear
echo "Your process has stopped"
logger $1 is no longer present
```

Here, the *condition* is a command, i.e., as long as the exit status of the command is 0, the loop persists. As soon as the exit code is non-zero, for example when the output is empty, the loop stops.

The first part of the command shows all processes that are related to top (or contain the word top), and `grep -vE "(bash|grep)"` removes the irrelevant results. The `-E` flag is used for the regular Expression `"(bash|grep)"` to be passed and understood by *grep*. The `-v` option shows the lines that don't meet the criteria, i.e., inverts the match. Thus we end up with the lines that are related to the top process, but do not contain irrelevant results.

As long as the above condition is true, the script *sleeps*, i.e., waits 1 second and then checks again. When the process terminates, a statement is printed and also logged in *syslog* via `logger`.

1.6.2 Until example

Let us consider the following script:

```
#!/bin/bash

until users | grep $1 > /dev/null
do
    echo "$1 is not logged in yet"
    sleep 5
done

echo "$1 has just logged in"
mail -s "$1 has just logged in" root <.
```

The `users` command show us the names of the users that are currently logged in:

```
# users
somu somu somu
```

Now, the expression `users | grep $1` is always going to be false unless the username mentioned in $1 is presently logged in, i.e., till the user logs in. The `> /dev/null` is used to discard any output. The moment the user logs in, the root user is notified via email and a message is printed on the console!

1.7 Understanding case

`case` is used to check if an argument has a certain value within a list of values, and act accordingly. Below is an example of the `case` statement as used in the `service` commands in old RHEL versions. For example to start the `httpd` service, the command was `/etc/init.d/httpd start`. To the service script, the `$0` parameter would be `httpd`, the process name and `$1` would be the activity, `start`. The concerned script is:

```
case "$1" in
    start)
        start;;
    stop)
        rm -f $lockfile
        stop
        ;;
    status)
        status;;
    restart)
        restart;;
    reload)
        reload;;
    *)
        echo $"Usage: $0 {start|stop|restart|reload|status}"
        exit 1
esac
```

In the script, there'd be functions called *start, stop, status, restart and reload* to perform specific actions, and based on the argument passed, those actions would be performed.

The `;;` statements are to prevent *fall-through* - a condition where all the statements below the matched case are executed until a `;;` is encountered. We wouldn't want the code for `stop` to be executed after just starting the service!

Finally, the `*)` is the *default* case, which matches everything else for which we didn't define a case. Here, it shows an error message detailing how to properly use the command.

In the error message, `echo $"Usage: $0 {start|stop|restart|reload|status}"`, we see the use of `$0` which represents the name of the script itself!