Understanding the shell - Part 2

September 30, 2011

1 Introduction

We build upon the ideas presented in the previous lesson and learn how to use the shell as a programming language.

2 A simple shell "script"

The GNU/Linux operating system has dozens of powerful commands which can be used for doing lots of interesting things. Let's take one example. The *find* command is used for locating files in your directory hierarchy. Let's say you wish to find out all C program files under the folder /home/examples whose size is greater than 10Kb. You will invoke find like this:

```
find /home/examples -size +10k -name '*.c'
```

What if you wish to limit your search to only those files created between June 10 and July 20? You will have to add more options to find!

The point I wish to convey is this: once you get close to the commandline, you will definitely start using complex commands with many options. Sometimes, you might have to type not just one command but a sequence of commands to get your job done.

What if you need to type these command sequences say a few dozen times every day? Typing these lengthy commands is not really fun; there should be an easier way to get things done.

The easy way is to store these commands in a file and run them by simply typing the name of the file!

Let's see how this is done.

Create a file (called say *t1*) and store the following lines in it:

```
echo hello
date
ls
who
```

Now, type the following at the shell prompt:

```
sh t1
```

You will see that the commands stored in the file t1 are executed one after the other. Congratulations - you have created your first shell program (or shell *script*)[1]!

A shell *script* can be as simple as a text file containing a few Linux commands. But shell scripts can do much more than that - as the following example demonstrates, you can do things like run commands in a loop.

3 A more complex script

Create a text file called *t2* and type the following code:

```
while true
do
   echo hello
done
```

You can run the script by typing:

```
sh t2
```

You will see that the command *echo hello* is getting executed in an infinite loop!

How does this really work? Your experience with some other language (like C) will prompt you to think of this while loop as something similar to a while loop in C. But that is not correct - there is a very big difference in the way the shell handles looping/decision making with the way it is done in languages like C. Let's explore this in detail.

3.1 A simple experiment

Type the following at the shell prompt:

```
true
```

You will not get any output. But the shell does not respond with a message like: "true: command not found" which implies the fact that true is simply a command!

But what does this command really do?

Try the following command at the prompt[2]:

```
echo $?
```

You will see the number 0 getting printed as the output. Now, execute the command:

[1]Another way to run the shell script is to give it execute permission using the chmod command. You can then type "./t1" to run the script.

[2]Make sure that you do not run any other command after running "true"

```
false
```

Again, you don't see any output; you can now try running:

```
echo $?
```

you will see 1 getting printed as the output.

What is this $? and what do the commands true and false really do? We will do another experiment to answer this question.

3.1.1 Let's write some C code

Write a very simple C program (in a file called *a.c*):

```
int main()
{
    return 11;
}
```

Compile and run the code:

```
cc a.c
./a.out
```

Now try running:

```
echo $?
```

You will see the result as 11. Try changing the C code (make it return some other value from main) and repeat the experiment.

On Linux systems, when the main function returns a value, it is some way or other captured by the shell and made available through the notation $?

So what does "true" really do? It is a very simple program which returns 0 (and performs no other action); similarly, "false" is a program which returns 1.

3.2 The logic of the shell "while" loop

Let's now come back to the while loop:

```
while true
do
   echo hello
done
```

How does this really work? The "operand" to while is not some kind of conditional expression - it is a program! In the above case, it is a very simple program called *true*[3]. The shell will run the program and examine its return value[4]. If

[3]It can be any program with a proper return value. All standard GNU/Linux commands have proper return values

[4]A better term is "exit status"

it is 0, the program is supposed to have executed succesfully[5] and the shell will run the body of the loop; if the return value is non-zero, the program is assumed to have failed and the loop will terminate.

Another experiment; modify the C code written in section 3.1.1 to make *main* return 0 and compile the code. Now, run the following script:

```
while ./a.out
do
  echo hello
done
```

you will see that our C program now works as a perfect replacement for the command *true.*

Why does the shell use programs instead of conditional expressions? The reason is that shell scripts are intended not for performing numerical computation but for manipulating processes and files. In such a file/process manipulation context, the use of programs (instead of conditional expressions) in while loops and if statements is natural and elegant. The example in the next section illustrates this.

4 The "if" statement

Here is a simple problem: check whether a word is present in a text file and if so, print a message. Let's first create a text file (called *a.txt*) and store some words in it:

```
hello
integer
world
maximum
```

Here is a small script which checks whether the word *integer* is present in the file or not:

```
if grep integer a.txt
then
    echo word found
else
    echo word not found
fi
```

The *if* statement does not have any *condition* part - instead, it runs a command (in this case, the *grep* command) and looks at the command's return value. If the return value is 0, the *then* part is executed; otherwise, the *else* part will be executed. The word *fi* should be used to end the *if* statement.

[5]Why does the shell use 0 for success. That is an exercise for you to find out!

Run the script and you will see it it prints the message *word found*[6].

We can explicitly check the return value of grep by executing the following commands at the shell prompt:

```
grep integer a.txt
echo $?
grep minimum a.txt
echo $?
```

The *echo* statement after the first invocation of grep will display 0 (because *grep* has succeeded in finding the word *integer* in the file *a.txt*). After the next invocation of *grep*, the *echo* command will display 1 (*grep* has failed because the word *minimum* is not present in the file *a.txt*).

5 The use of "backquotes"

Try this command at the shell prompt:

```
a=date
echo $a
```

This will simply assign the string *date* to the shell variable *a* - the *echo* command simply prints *date* on the screen.

Now try this[7]:

```
a=`date`
echo $a
```

Here is the output I got on my machine:

```
Thu Aug 25 13:34:27 IST 2011
```

What does the backquote do? It interpets the enclosed string as a command, executes the command and assigns to the variable the output generated by the command.

6 Doing arithmetic

The Shell is not designed for arithmetic[8] - but there will be situations where you have to have to do simple computations - mostly as part of scripts which performs file/process manipulation.

[6]You will also note that there is another line in the output - this line is produced by grep itself. How do you make this line disappear? There are two ways to do this: one, using an option to grep and the other, using output redirection. Try to implement both. (you might also like to explore the null device: http://en.wikipedia.org/wiki//dev/null)

[7]We are enclosing *date* in two back-quotes. The back-quote key is usually present below Esc on the keyboard

[8]I have seen university question papers where they ask questions like "compute square root using shell" ... the people who set these papers miss the whole point behind shell scripting. The shell is NOT a general purpose programming language - it is suited only for a very specific task - and that task is manipulating files and processes.

Try this at the shell prompt[9]:

```
expr 1 '+' 2
```

The output will be the number 3.

The *expr* command accepts operators (symbols like +, * etc) and operands (numbers like 1, 2 etc) as command line arguments[10]. It applies the operator on the supplied operands and displays the result on the standard output[11]. It is always better to place the operator in single quotes - otherwise, the shell might perform special processing on symbols like * yielding incorrect results. The peculiar way by which shell performs arithmetic makes even simple things like incrementing the value of a variable look tricky.

Try the following:

```
i=0
i=`expr $i '+' 1`
```

The second line may be confusing to beginners. The right hand side is the *expr* command enclosed in backquotes. Remember that if you need to assign the output of a command to a variable, you have to enclose the command in backquotes!

7 Testing conditions

The shell builtin command *test* can be used to compare values and check for different types of files.

Try the following experiment:

```
test 1 -lt 2
```

Our objective behind invoking *test* was to check whether 1 is less than 2 (that is what the -lt does). But *test* has displayed no output! So how does it really work?

Try the following command:

```
echo $?
```

You will get 0 as the output. Now do:

```
test 2 -lt 1
echo $?
```

[9]There are two other ways in which arithmetic can be done - using double parenthesis and using the *let* statement. Check out http://tldp.org/LDP/abs/html/arithexp.html for more information.

[10]Take care to always separate the operators and operands with space

[11]If you have some experience writing C code, you can think of a creating a simple program which mimics expr!

The *echo* command displays 1.

The *test* command performs the requested comparison and sets its return value as 0 or 1 depending on the success/failure of the comparison.

Note the key difference between *expr* and *test* - *expr* generates its result as a value displayed on the standard output (the terminal) while *test* produces no output - it simply sets its return value to 0 or 1.

8 A simple script which prints 10 numbers

Here is a simple script which brings together many of the ideas which we have discussed:

```
i=0
while test $i -lt 10
do
     echo $i
     i=‘expr $i ’+’ 1‘
done
```

Because test sets its return value to 0 or 1 depending on the success or failure of the comparison, it can be used directly in a while loop or if statement.

9 Conclusion

The objective of this two-part lesson was to introduce you to the sophisticated tool which the Linux shell is. We have focused on the basic ideas while skipping over most of the details; you will learn more as you keep experimenting with different aspects of the GNU/Linux operating system!

10 Challenge

If you are a C programmer, you can study the working of the “fork” and “exec” system calls and implement a “toy” shell of your own in C!

11 Test Exercises

1. What is the output of the following command:

   echo *

   (a) *
   (b) *

(c) None of the above

2. How do you use "test" to check whether a file (named say "abc") is a symbolic link?

   (a) test -b abc
   (b) test -d abc
   (c) test -h abc
   (d) None of the above

3. What is the notation $# used for in shell scripts?

   (a) To get the list of command line arguments[12] to a shell script
   (b) To get the number of command line arguments to a shell script
   (c) None of the above

4. What is the notation $1 used for in shell scripts?

   (a) To get the second command line argument
   (b) To get the first command line argument
   (c) None of the above

5. The environment variable PS10 controls:

   (a) The way the shell prints the "command prompt"
   (b) It is not used by the shell - there is no such environment variable

6. The environment variable HOME refers to:

   (a) The user's login directory
   (b) Ther is no such environment variable

[12]When you type a command say "rm a.c b.c", you say that the names "a.c" and "b.c" are command line arguments