Understanding the shell - Part 1

September 30, 2011

1 Introduction

The *shell* is the medium through which the programmer interacts with the operating system. Whether you are a programmer or a system admin, whether you work on low-level embedded systems/kernels/networking protocols or high level databases/graphical applications, there is one skill without which you can never hope to work effectively on Unix like systems - and that is a proper understanding of the shell.

Screencast file: shellbasics/shellbasics1.mp4

2 What is the "shell"?

The shell is simply a C program which reads in the commands that you type and executes them. When you open a terminal on your GNU/Linux system, a shell starts running and waits for your input; the *prompt* that you see on the screen is displayed by the shell[1]:

```
recursive@sherlock:~$
```

The prompt contains some useful information. The $ sign indicates the fact that you are logged in as an ordinary user. On Linux systems, ordinary users have only limited privileges - for example, you can't partition your hard disk or configure your network interfaces if you are logged in as an ordinary user. There is a special *superuser* (whose login name is *root*) who can do all these things - if you are logged in as the superuser, your prompt will have the symbol # instead of the $.

The ~ sign in the prompt indicates the fact that your current directory is the login directory (if your login name is *rahul*, your login directory will usually be */home/rahul*). You will see it changing when you move into other folders (using the *cd* command).

The other two pieces of information available from the prompt are your login name and the machine name. The prompt shown above indicates that I am logged in as user *recursive* and my machine name is *sherlock*.

[1] On GNU/Linux, we normally use a program called *bash* (abbreviation for "Bourne Again Shell")

3 The shell is much more than a command interpreter

The shell can do much more than simply run the commands that you type. Here are some of the cool things that the shell will do for you:

- The shell performs *metacharacter* expansion
- The shell handles redirection and piping
- The shell can act as a programming language!

Let us now examine each of these things in detail.

3.1 Metacharacter expansion

Create a new folder (called say *test*) and store some empty files in it (the *touch* command can be used to create empty files):

```
mkdir test; cd test
touch a b c ad abcd
ls
```

(Note: You can type multiple commands in a line, just separate the commands with a ;)

You note that the touch command has created some empty files.

We know that the *echo* command can be used for displaying messages; the command *echo hello* will display the string *hello* on the terminal. Now, type the following command at the shell prompt:

```
echo *
```

What do you see in the output? You will see the names of the files you have just now created using *touch*.

The symbol * is a *metacharacter* (meaning it is interpreted in a special way be the shell; there are many other special characters besides the *). The shell expands the symbol * to the names of all the files in the current directory.

Try another experiment. Run the command:

```
echo a*
```

You will now see only those file names starting with the alphabet *a*. What if you type:

```
echo a*d
```

You will see just those names starting with *a* and ending with *d*[2].

[2]Note: The way the shell handles the * symbol is different from the way a command like grep handles it

There is a very important point to note: the interpretation of special characters like * is done by the shell; when you type echo a*, the *echo* command never sees the *; it only gets to see the expanded list.

Why is this important? Because it is the shell which performs metacharacter expansion (and not commands like *echo*), these symbols can be used with any program. Suppose you wish to remove all files in your current directory, you can do:

```
rm *
```

The rm command does not see the symbol *; it sees only the list of names of all files in the current directory.

3.1.1 Quoting

The special processing of symbols like * done by the shell is very useful; but there are situations where this can create trouble. Let's look at one such example.

Remove all the files in the folder *test* and create a new file called *hello.txt*. This file should contain the following line:

```
The April heat is killing me
```

Now, we are going to use the *grep* command to identify patterns like the following in the above file:

```
t
ht
hht
hhht
hhhht
.....
```

The pattern is: zero or more number of occurrences of the symbol *h* followed by the symbol *t*. The regular expression for this pattern is:

```
h*t
```

(Note: the * symbol has a different meaning in the context of regular expressions used by commands like grep. The sequence *h** implies zero or more occurrences of the symbol h. So *h*t* is: zero or more occurrences of *h* followed by *t).*

Do we have such a pattern in our file *hello.txt*? Yes. The word *heat* has a single occurrence of the alphabet *t* which matches the pattern!

Let's now try running grep:

```
grep h*t hello.txt
```

We expect to see the line "The April heat is killing me" getting printed on the screen because *grep* will match *h*t* with the single occurrence of the character *t* present in the word *heat*. But you will not see any output!

Why is this so? It is because the shell will execute *grep* only after performing its own metacharacter expansion on the word *h*t*! The shell will interpret *h*t* to stand for the list of names of all files starting with *h* and ending with *t*. There is one such file in the current directory and it is called hello.txt. So, the shell will run grep after replacing the word *h*t* with the word *hello.txt*. What really gets executed is the command:

```
grep hello.txt hello.txt
```

Grep is now trying to look for the pattern: hello.txt (Which means: the symbol *h*, followed by *e*, followed by two occurrences of *l*, followed by the symbol *o*, followed by one occurrence of *any symbol*, follwed by *t*, *x* and *t*. Note that grep interprets the symbol . as matching one occurrence of any character). There is no such pattern in the file and so, you get no output on the screen.

How can this problem be rectified? There should be some way to ask the shell not to do any kind of processing of special symbols. This is done by *quoting*.

Run the following command:

```
grep  'h*t'  hello.txt
```

Now you see that things are working perfectly! When you enclose a word in quotes (both single and double quotes can be used; but there are subtle differences between the two. For the moment, we will stick with single quotes), the shell will not look inside the quotes and process the * symbol - it will simply leave the word as it is. So, *grep* will be able to see the pattern *h*t* and process it properly.

You can try a simpler experiment using the *echo* command; just type:

```
echo  '*'
```

You will simply see the symbol * getting displayed on the terminal. The shell now leaves the * symbol untouched - so what echo sees is the single character * which it displays onto the screen.

Here is the moral of the story: be careful when you are using symbols like * in the command line - you may need to quote them in certain situations.

3.2 Redirection and piping

3.2.1 Redirection

We had seen how the shell handles redirection in an earlier lesson. The basic idea is that any command which displays its output on the terminal can be executed in such a way that its output gets stored in a file.

For example, if you run:

```
date > abc
```

you will not see any output on the terminal - instead, the output gets stored in a file called *abc*. This is called *standard output redirection*.

We shall now look at one or two ideas concerning redirection which were not covered in the previous lesson.

Try this experiment - run the rm command without any file name argument:

```
rm
```

You will get the following error message:

```
rm: missing operand
Try 'rm --help' for more information.
```

Let's now try to get this error message into a file; so we execute:

```
rm > abc
```

But this doesn't work! The error message is again getting printed out onto the terminal. If you check the file *abc*, you will see that it is empty!

Why does this happen? The idea is that Linux commands write error/usage messages not to the standard output but to something called the standard error[3]. The > symbol performs redirection only on the standard output - it does not redirect the standard error. You have to do some magic if you wish to redirect standard error! Try this:

```
rm 2>abc
```

You will now see that the output is getting stored in the file *abc*. The notation *2>abc* instructs the shell to redirect the standard error to the file *abc*; the number 2 represents standard error (0 represents standard input and 1 represents standard output).

It is possible to redirect both standard output and standard error to one file; here is how it is done:

```
rm > abc 2>&1
```

The notation 2>&1 redirects standard error to whatever standard output is pointing to; in the above case, the use of "> abc" ensures that standard output is redirected to the file abc.

3.2.2 Piping

Let's now look at how *pipelines* work.

The command *who* tells us who all are logged on to the system; here is the output of running *who* on my machine:

```
recursive tty7         2011-08-22 10:18 (:0)
recursive pts/0        2011-08-22 10:18 (:0.0)
recursive pts/1        2011-08-22 10:21 (:0.0)
recursive pts/2        2011-08-22 11:02 (:0.0)
recursive pts/3        2011-08-22 15:06 (:0.0)
```

[3] In order to understand how things like standard output/error etc really work, you need to learn about the Linux system call interface in detail. It is beyond our scope at this point.

Now, I wish to find out how many users are logged on; simply counting the number of lines in the output of *who* is sufficient for this. Try this command:

```
who | wc -l
```

The *wc* command is used for counting number of lines in a file (the -l option makes *wc* print only the number of lines - without this option, *wc* will print number of characters, words as well as lines). The symbol | is the pipe symbol. We use pipes to connect the output of one command to the input of another command - in the above case, the output generated by *who* is fed to the input of *wc*.

Pipes are incrediby powerful - you can't live without them on Linux!

In general, you can connect any number of commands in a pipeline to form a *multistage* pipeline. Here is an example:

```
who | grep '^recursive' | wc -l
```

The output of *who* is fed to the input of *grep* - *grep* will display only those lines starting[4] with the word *recursive*. The output generated by *grep* is fed to the input of *wc*. We are basically counting how many times one particular user (in this case, a user called *recursive*) is logged in.

3.3 The shell as a programming language

The real power of the shell comes from the fact that it is a programming language - but the trouble is that it is a very peculiar language. There are some basic concepts without getting a clear idea of which you will find it hard to program the shell. In this section, we shall learn more about these fundamental ideas.

3.3.1 Shell variables

Execute the following command at the shell prompt:

```
a=1
```

Make sure that you do not type any space in between the symbols *a*, = and 1.

You have just now created what is called a *shell variable*. Think of the symbol = as an assignment operator; we have assigned the value 1 to symbol *a*.

How do you retrieve the value stored in a?

Try the following command:

```
echo  $a
```

you will see the number 1 getting printed. If you wish to examine the value stored in a shell variable, you have to prefix the variable name with the special symbol $.

Try this command at the prompt:

[4] The ^ symbol is used for making grep match only at the beginning of a line

```
echo '$a'
```

You will simply see the string $a getting printed; the shell will always display as such any sequence of characters you specifiy in single quotation.

Now, change the single quotes to double quotes:

```
echo "$a"
```

You will see that now the shell is printing 1 (the value of the symbol *a*). The difference between double quotes and single quotes in the context of the shell is that the shell expands certain metacharacters (like the $) inside double quotes wheras metacharacter expansion is completely disabled inside single quotes.

Let's try one more experiment. Try to execute:

```
a = 1
```

(Note: the difference is that we have now added spaces in between the symbols).

The shell gives you an error message: "command not found"!

When you add a space after *a*, the shell interprets *a* as a command (like cp, rm etc) and the symbols = and 1 as its arguments. There is no command called *a* - so you get an error.

There is one very important thing you have to learn from this - and that is the fact that the shell tries to be both a programming language and a command interpreter at the same time. The use of whitespace is key to differentiating whether you wish something to be interpeted as a statement in a programming language (like assignment) or as the execution of a command; so treat whitespace with respect!

3.3.2 Running a sub shell

We will first create a shell variable and check its value:

```
b=20
echo $b
```

Now, execute the following command at the shell prompt:

```
bash
```

You will not notice anything; but there is a definite effect to running this command; "bash" is the name of our shell and by typing it explicitly at the shell prompt, we have launched another instance of the shell (let's call this a sub-shell). Any command that you type is now going to be processed by the sub shell.

Let's try running:

```
echo $b
```

We don't see the value 20! Why?

The idea is that variables created in the parent shell are not by default copied to the child shell.

We will now exit the sub shell by running:

```
exit
```

We are now back to the parent shell. Let us do:

```
export  b
```

The result of running this command is that the shell variable will now be copied to sub-shells. We can verify this easily by once again running:

```
bash
```

and checking out the value of *b*:

```
echo  $b
```

We are now getting 20! Let's try to modify the value of b in the sub-shell:

```
b=100
echo  $b
```

Now, let us go back to the parent shell and check the value of *b* once again:

```
exit
echo  $b
```

We see that the value is unchanged; it is still 20. This demonstrates the fact that the sub-shell gets a copy of the variable present in the parent shell.

3.3.3 Environment variables

Shell variables whose values are available in sub-shells are called *environment variables*. The *printenv* command can be used to get the list of all environment variables. You will see lots of lines in the output if you run this command on your GNU/Linux system. This is because the shell maintains some pre-defined environment variables - one example is PATH.

Try running the following command:

```
echo  $PATH
```

Here is the output I got on my machine:

```
/home/recursive/bin:/usr/local/bin:/usr/bin:/bin:
/usr/local/games:/usr/games:/home/recursive/jdk6/bin
```

The value of this environment variable is a colon separated list of directory names. The significance of this list comes from the fact that whenever the shell tries to run an external command (commands like *echo, cd, pwd* are "internal" to the shell - the shell need not go looking for external programs to run these commands), it will look for the executable file corresponding to that command under the folders given in PATH. For example, if you type *vi* at the command prompt, shell first looks for an executable file called *vi* under the folder /home/recursive/bin, then it looks under /usr/local/bin, then under /usr/bin and so on.

Let's now try an experiment with PATH. Modify the value of PATH by running the following command:

```
PATH=
```

We have set value of PATH to the empty string; you can verify this by running:

```
echo  $PATH
```

Now, try running the *ls* command; the shell responds with the message "no such file or directory". Try commands like *date*, *who*, *vi* etc and the result is the same!

How do you explain this behaviour? All the commands mentioned above (*ls*, *date*, *who*, *vi*) are external commands; the shell will examine only the directories specified in PATH in order to get the executable files for these commands. Because our PATH is empty, shell will simply not look anywhere!

Let's now start building up the PATH. Run the command:

```
PATH=/bin
```

We now have just one directory in our PATH. Try running the above commands once again - *ls* and *date* will work because the executable files of these commands are located under */bin*. But *vi* does not work because its executable file is present in /usr/bin which is not present in PATH.

Let's modify PATH once again by running:

```
PATH=$PATH:/usr/bin
echo $PATH
```

We are appending a colon and the string */usr/bin* to the current value of PATH. As a result of this change, you will note that you are now able to run *vi*. The executable files for a lot of commands will be present in either */bin* or */usr/bin* - so having just these two folders in our PATH is sufficient to get many commands running.

When you install new programs on your systems, sometimes the executable files of these programs might be installed under folders which are not usually found in PATH - you will have to modify PATH to get these commands running.

4 Conclusion

We have seen some of the ideas which are key to using the shell effectively. We will continue the discussion in the next lesson.