

Chapter 4: Shell Programming

Objectives

- Understand what the shell does to a command
- Learn the various ways of executing a shell script
- Take input from the keyboard and a file with **read**.
- Understand how command line arguments are passed to and interpreted as *positional parameters* inside a script.
- Learn the significance of the *exit status* of a command and the parameter \$?.
- Use simple one-line conditionals with the && and || operators.
- Use the **if** statement for multiway branching using a UNIX command as its *control command*

- Use the pattern matching features of **case** for matching strings with wild cards
- Perform integer computation and manipulate strings with **expr**.
- Learn how to design a script that behaves differently depending on the name by which it is invoked.
- Run a set of commands repeatedly using the **while** and **until** loops.
- Execute a set of statements for each member of a list using a **for** loop.
- Change filename extensions with **basename**.

•4.1 Function of the Shell

- When you log on to a UNIX machine, you see a prompt. The prompt could be a \$, a % or anything.
- Even though it may appear that nothing is happening out there, a UNIX command is in fact running. This command is the **shell**.
- It starts functioning the moment you log in and withers away when you log out.
- When you issue a command, the shell is the first agency to acquire the information.
- It accepts and interprets user requests; these are generally the UNIX commands we key in.
- The shell examines and rebuilds the command line and leaves the execution work to the kernel.

- The kernel handles the hardware on behalf of these commands and all processes in the system.
- Users can thus afford to remain ignorant of the happenings behind the scenes.
- The shell is generally sleeping. It wakes up when input is keyed in at the prompt.
- This input is actually input to the program that represents the shell (sh for the Bourne shell).
- The following activities are typically performed by the shell.
- It issues the prompt (\$ or otherwise) and sleeps until you enter a command

- After a command has been entered, the shell scans the command line for some special characters (metacharacters) that have a special meaning for it.
- Because it permits abbreviated command lines (like the use of `*` to indicate all files, as in `rm *`), the shell has to make sure the abbreviations are expanded before the command can act upon them.
- It then creates a simplified command line and passes it on to the kernel for execution.
- The shell can't do any work while the command is being executed, and has to wait for its completion.
- After the job is complete, the prompt reappears and the shell returns to its sleeping role to start the next "cycle." You are now free to enter some more commands.

4.2-Shell Programming

- The activities of the shell are not restricted to command interpretation alone.
- The shell has a whole set of internal commands that can be strung together as a language-with its own variables, conditionals and loops.
- Most of its constructs are borrowed from C, but they are compact and simpler to use than those in C.
- What makes shell programs powerful is that the external UNIX commands blend easily with the shell's internal constructs.
- Shell programs run in interpretive mode-one statement at a time.
- They consequently run slower than those written in high-level languages.

- Speed is not a factor in many jobs we do, and in many cases using the shell is an advantage-especially in system administrative tasks.
- The UNIX system administrator must be an accomplished shell programmer.

•4.2.1 Shell Variables

- A shell variable must start with a letter but can contain numerals and the underscore character.
- A variable is assigned with the = symbol without using the \$, but is evaluated by prefixing it with a \$.
- \$fname=profile**
- \$echo \$fname**
- profile**
- Variables are concatenated by placing them side by side;

```
$x=foo; y=.doc    //can make multiple assignments in one line  
$z=$x$y          //variable assigned by other variables  
$echo $z  
foo.doc
```

The shell uses an alternative notation when evaluating variables-a pair of curly braces that enclose a variable name.

Here is another way of evaluating the variable fname:

```
$echo ${fname}  
profile
```

A variable and a string can't be concatenated in the same way as two variable can.

To affix an x to a variable's value, you have to use either of these two forms:

```
$echo ${fname}x  
profilex  
$echo $fname "x"  
Profilex
```

These forms have at least one advantage; you can generate a second set of filenames by modifying the original name. You'll find these concatenation features useful when changing a file's extension.

4.2.2 Shell Scripts

```
$vi scripts.sh  
#Sample Shell Script  
echo "The date today is `date`" #Using command substitution  
echo Your shell is $SHELL  
echo Your home directory is $HOME  
echo The processes running on your system are shown below:  
ps
```

You can use **vi** and **emacs** to create this script. Note the comment character **#** that can be placed anywhere in a line.

The shell ignores characters placed on its right.

You can execute this file in the following manner.

First is to make the shell script executable with the **chmod** command before running it.

```
$chmod +x script.sh; ls -l script.sh
```

```
-rwxr-xr-x  1  Daniel dialout 154    Feb 25 22:33 script.sh
```

Now that there's executable permission for the script, You can run as follows:

```
./scripts.sh
```

The date today is Thur November 2 06:02:27 PDT 2016

You shell is /bin/bash

Your home directory is /home/daniel

The processes running on your system are shown below:

PID	TTY	TIME	COMMAND
-----	-----	------	---------

4552	pts/1	00:00:00	bash
------	-------	----------	------

4574	pts/1	00:00:00	bash
------	-------	----------	------

4576	pts/1	00:00:00	ps
------	-------	----------	----

All the statements have been executed in sequence.

Shell scripts are always run in a separate shell-a sub-shell.

When a script is running, the `ps` command always shows at least two **sh** (**ksh** and **bash** for the other shells) processes running. The login shell is the parent of the sub-shell that actually runs the script.

When running loops like **while** inside a script, more sub-shells are created, but keep in mind that there are at least two shell processes running when a script is in execution.

Now, which shell will you use to execute a shell script?

You will naturally want to use your login shell, but what happens if you have a script written in Bourne shell, and your login shell is Korn?

You might try running it by calling it like this:

```
script.sh or ./script.sh
```

But then this will call a Korn sub-shell to execute it—and possibly produce errors.

Since the Bourne shell is generally available in every system, you can call it as your sub shell:

`sh script.sh`

Here, you are executing the **sh** command which uses the script name as its argument.

Sh opens the file and executes the statements in the script file in sequence.

No executable permission for the script file is then required. Since you called up the sub-shell explicitly, you are sure of your shell too.

Specifying the Interpreter: It is also possible to specify the shell inside the script itself. So instead of executing a script with **sh** filename, it is better to place the following statement at the beginning of the script.

`#!/bin/sh`

The `#` is treated as the comment character by the shell, but when you follow it with `!`, the shell interprets the remainder of the string as the interpreter specification.

You can now execute the script without calling the **sh** explicitly.

If your script will run properly only under a specific shell, put the interpreter at the top of the script (and before you enter anything else).

4.2.3 **read**: Making Scripts Interactive

The **read** statement is the shell's internal tool for taking input from the user; therefore, making scripts interactive.

It is used with one or more variables.

Input supplied through the standard input is read into these variables.

When you use a statement like
`read name`

the script pauses at that point to take input from the keyboard.

Whatever you enter is stored in the variable `name`.

Since this is a form of assignment, no `$` was used before `name`.

Our first script **`emp1.sh`** uses **`read`** to take a search string and filename from the terminal

`$vi emp1.sh`

(hit `i` first to input data, then type the following)

`#!/bin/bash`

`#Script: emp1.sh-Interactive version`

`#The pattern and filename to be supplied by the user`

```
echo "Enter the pattern to be searched: "  
read pname  
echo "Enter the file to be used: "  
read fname  
echo "Searching for $pname from file $fname"  
grep "$pname" $fname  
echo "Selected lines shown above"
```

Now create a file called emp2 with the following data

ID	Name	postion	Movie
1	Abebe	director	Beautiful Mind
2	Daniel	director	Lost
3	Getachew	actor	Amen
4	Almaz	camera	Ghost

Now after assigning execute permission to emp1.sh, execute the file.

Enter the pattern to be searched for: **director**

Enter the file to be used: **emp2**

Searching for director from file emp2

```
1  Abebe director    Beautiful Mind
2  Daniel director   Lost
```

Selected lines shown above

First, the script asks for the pattern to be entered.

Input the string director, which the shell assigns to the variable pname

Next, it asks for the filename: enter the string emp2, which goes to the variable fname.

After accepting these inputs from the keyboard, the `grep` and `echo` statements are executed.

You can enter both arguments in one line before hitting *[Enter]*, but then you would have to use the **`read`** statement with two variables.

```
echo "Enter the pattern and filename: "  
Read pname fname
```

If the number of arguments supplied is less than the number of variables accepting them, any leftover variables will simply remain unassigned.

However, when the number of arguments exceeds the number of variables, the remaining words are assigned to the last variable.

4.2.3.1 Redirecting the read Input

The previous scripts arguments were read from the standard input by **read**.

The file can accept arguments from a file instead of a key board.

First create a file called list with the two arguments “director” and “emp2” separated by a newline.

Then

\$/emp1.sh < list

Enter the pattern to be searched: Enter the file to be used:

Searching for director from file emp2

1 Abebe director Beautiful Mind

2 Daniel director Lost

Selected line shown above

4.2.4 Positional Parameters

Shell scripts accept arguments in another manner-from the command line itself.

When arguments are specified with a shell script, they are assigned to certain “variables”-rather **positional parameters**.

The first argument is read by the shell into the parameter \$1, the second argument into \$2, and so on.

You can't technically call them shell variables because all variables are evaluated with a \$ before the variable name.

In case of \$1, you really don't have a variable 1 that is evaluated with \$.

- \$1 – The first argument
- \$2 – The second argument
- \$0 – The name of the script
- \$# -- The number of arguments
- \$* -- The complete set of positional parameters as a single string

In addition to the parameters signifying the arguments, the shell also uses three special parameters -- \$0, \$# and \$*.

Using vi text editor, create a file called emp2.sh with the following contents:

```
#!/bin/sh
```

```
echo "Program: $0"      # $0 contains the program name
```

```
echo "The number of arguments specified is $#"
```

```
echo "The arguments are $*"
```

```
grep "$1" $2
```

```
echo "Job Over"
```

Invoke the script with the pattern director and the filename emp2 as two arguments:

```
$ ./emp2.sh director emp2
```

```
Program: emp2.sh
```

```
The number of arguments specified is 2
```

```
The arguments are director emp2
```

```
1    Abebe  director    Beautiful Mind
```

```
2    Daniel director    Lost
```

```
Job Over
```

When arguments are specified this way, the first word (the command itself) is assigned to \$0, the second word (the first argument) to \$1, and the third word (the second argument) to \$2.

You can use more positional parameters in this way up to \$9.

4.2.5 Exit Status of a Command

Once **grep** couldn't locate a pattern, we said that the command failed.

Grep simply returns the prompt when it fails to find the pattern.

Every command returns a value after execution.

This value is called the **exit status** or **return value** of the command.

This value is said to be true if the command executes successfully and false if it fails.

The return value is extremely important for programmers.

They use it to devise program logic that branches into different paths depending on the success or failure of a command.

For example, there's not point in continuing with script execution if an important file doesn't exist or can't be read.

4.2.5.1 The Parameter \$?

There's another special parameter used by the shell: it is \$?.

It stores the exit status of the last command.

It has the value 0 if the command succeeds, and a non-zero value if it fails.

For example, if **grep** fails to find a pattern, the return value is 1, and if the file scanned is unreadable in the first place, the return value is 2.

In any case, the return value exceeding 0 are to be interpreted as failure of the command.

Try using **grep** in different ways:

```
$grep director emp2 > /dev/null; echo $?
```

0

```
$grep manager emp2 2> /dev/null; echo $?
```

1

```
$grep manager emp3 2> /dev/null; echo $?
```

2

You should note that all assignments to the positional and special parameters are automatically made by the shell.

You can't really tamper with their values.

But you can use them to great advantage in several ways.

They will be used in shell scripts over and over again.

Here is a Table that summarizes them.

Shell Parameter	Significance
\$1, \$2, etc	Positional parameters
\$*	Complete set of positional parameters as a single string
\$#	Number of arguments specified in a command line
\$0	Name of executed command
\$@	Same as \$* except when enclosed in double quotes
\$?	Exit status of last command
\$\$	PID of the current shell
#!	PID of the last background job

4.2.6 The Logical Operators && and ||--Conditional Execution

The script `emp1.sh` has no logic to prevent display of the “Selected lines shown above” even when the pattern search fails.

That is because we didn’t use `grep`’s exit status to control the flow of the program.

The shell provides two operators that allow conditional execution – the `&&` and `||`, which typically have this syntax:

```
cmd1 && cmd2
```

```
cmd1 || cmd2
```

The `&&` delimits two commands; the command `cmd2` is executed only when `cmd1` succeeds.

You can use it with `grep` in the following fashion:

```
$ grep 'director' emp2 && echo "pattern found in file"
```

```
1 Abebe director Beautiful Mind
```

```
2 Daniel director Lost
```

```
pattern found in file
```

The `||` operator play an inverse role; the second command is executed only when the first fails.

If you “grep” a pattern from a file without success, you can notify the failure:

```
$grep 'manager' emp2 || echo "Pattern not found"
```

```
pattern not found
```

4.2.4 **exit**: Script Termination

Sometimes, you test for a condition and find that it fails.

You'd then like the program to terminate since there's no point in continuing further if an essential resource is missing-say the file that you want to search is missing.

The **exit** statement is used to prematurely terminate a program.

When this statement is encountered in a script, execution is halted and control is returned to the calling program-in most cases the shell.

You don't need to place **exit** at the end of every shell script because the shell knows when script execution is complete.

Instead, it is quite often used with a command when there are chances that it may fail.

Now let us modify the previous program emp2.sh to show the results of a search if a pattern is found, but terminate if the search fails for any reason

Use a vi text editor to fill the contents of emp2a.sh as follows:

```
#!/bin/bash
echo "Program: $0"          # $0 contains the program name
echo "The number of arguments specified is $#"
```

echo "The arguments are \$*" #All the arguments stored in \$*

```
grep "$1" $2 >patlist 2>/dev/null || exit 2#exit with an argument
cat patlist
```

The program is aborted when the grep command fails, but the contents are shown if it succeeds.

```
$/emp1a.sh manager emp2
```

Program: ./emp1a.sh

The number of arguments specified is 2

The arguments are manager emp2

The last echo statement has not been executed not the contents of patlist displayed. But what was the return value? Check the value of \$?:

```
$echo $?
```

2

This was the value specified as **exit**'s arguments in the script.

The argument is optional, but when you specify one, the script returns that value only if it terminates at that point.

4.2.8 The if conditional

The if statement takes two-way decisions depending on the fulfillment of a certain condition.

In the shell, the statement uses the following forms, much like the one used in other languages:

```
If command is successful
then                                     //Form 1
    execute commands
else
    execute commands
fi
```

```
If command is successful
then
    execute commands                 //Form 2
fi
```

```
If command is successful
then
    execute commands
elif command is successful           // Form 3
then...
else...
fi
```

As in C++, if also requires then.

It evaluates the success of failure of the command that is specified in its “command line.”

If the command succeeds, the sequence of commands following it is executed.

If a command fails, then the **else** statement (if present) is executed.

This statement is not always required, as shown in Form 2.

Every **if** is closed with the corresponding **fi**, and you'll encounter an error if one is not present.

Now create a file called `emp3.sh` and fill its contents with the following data:

```
if grep "$1" /etc/passwd 2 > /dev/null
then
    echo "Pattern found – Job Over"
else
    echo "Pattern not found"
fi
```

This is a simple if-else construct where **if** tests the return value of **grep**.

Now let us search `/etc/passwd` for mail

\$/emp3.sh mail

mail:x:8:12:mail:/var/spool/mail:/sbin/nologin

mailnull:x:47:47::/var/spool/mqueue:/sbin/nologin

Pattern found-Job over

\$/emp3.sh firewall

Pattern not found

At this point it is important to realize that in UNIX, all commands return a value. This return value of the grep command is used in the if conditional.