Shell Scripts-I

Session Objectives

- **Shells**
- **Flavors of Shells**
- **Shell's special characters**
- **Shell variables**
- **Shell Scripts**
- **Parameter handling in shell scripts**

The Shell

The shell is a sophisticated utility program. It is unix's command interpreter. It prompts the user for commands, and makes the unix system obey them. Shell is the interface between the user and the system. When the user logs on to the system, through the session, till the user logs out, the shells stays with the user. When the user logs out, the shell controls the termination of the session.

Flavors of shell

Linux supports many different shells, with various different features. The most well known shells are

➢ ash - A Shell

'ash' is the minimum shell under Linux. During Slackware installation(another kind of linux distribution) this shell is used. But later when your system is running you don't need this shell anymore. But nevertheless: 'ash' is completely Bourne shell compatible and therefore works with most shell scripts that are picked up from other platforms.

➢ bash – Bourne Again Shell

'bash' is the standard shell on Linux. 'bash' must be installed on your system to boot properly. Like 'ash' the 'bash' shell is completely compatible to the Bourne shell found on nearly every Unix system. On Linux systems 'sh' is a link to 'bash'.

For interactive working on the command line 'bash' is very useful. It provides various features like automatic command line completion, history, aliases, editor capability, search utility, and more.

➢ pdksh - Public Domain Korn Shell

The Korn shell 'ksh' is a widely available extension of the Bourne shell on commercial Unix systems. But 'ksh' doesn't reach the quality of 'bash' and 'tcsh'. 'pdksh' is a free version of 'ksh' for Linux. On Linux systems 'ksh' is a link to pdksh.

➢ sash - Standalone Shell

This is a kind of rescue shell. When everything fails, (e.g. because the system cannot find libc.so) this shell will be very helpful:

➢ zsh – Z Shell

The Z shell is a well-known command interpreter. It is often used on real time operating systems like OS/9 because 'zsh' is more comfortable than the standard shells for these systems (history, editor capability). On Linux systems 'zsh' is obsolete, because there are more comfortable shells available ('bash' and 'tcsh'). zsh is wide configurable.

➢ tcsh – Extended C Shell

The TC shell 'tcsh' is a C shell with more functions and less bugs than the original ('csh'). On big-endian systems 'tcsh' is mostly around as an alternative to 'sh' and 'ksh'. For interactive work on the command line 'tcsh' is really great. 'tcsh' offers functions and comfort like 'bash': command completion, history, aliases, and more.

The original 'csh' had a lot of serious bugs and is therefore not suitable for shell programming. But 'tcsh' is more or less bugfree. So it is possible to use 'tcsh' for shell programming. The syntax is similar to the C programming language. Under Linux 'csh' is a link to 'tcsh'.

NOTE
Each shell has its benefits. The Bourne shell family (*i.e.*, **sh**, **ksh**, and **bash**) is generally regarded as being superior for writing shell-scripts. On the other hand, the C shell family (*i.e.*, **csh**, and **tcsh**) is generally regarded as being preferable for interactive use. The shells supported by Linux are in a file called /etc/shells

The Shell's special characters

The shell uses some special characters to change the interpretation of a command's elements. Special characters used by the shell are:

Character	**Interpretation**
$	Substitutesthe value of a variable.
;	A separator to break the command line into separate commands
\	Removes the special meaning of the character that immediately follows
" "	Removes the special delimiting meaning of spaces and certain other special characters contained between " "
' '	Removes the special meaning of any special characters contained between ' '
` `	Used for command substitution
&	Execute this command as a background process

$

When a string of characters begin with a $ sign the shell interprets it as the value of a variable. It does not read the literal characters of the string. For eg.,

```
$ a=hai
$ echo a
a
$ echo $a
hai
```

When a value contains a space then the value must be enclosed in quotation marks (" ").

```
$ b=Transed Software
Software: not found

$ b="Transed Software"

$ echo $b
Transed Software
```

The shell replaces $b with its value before running echo, so echo does not see variables. This is called variable substitution.

;

The shell recognises the semi colon (;) as a command delimiter or separator.A number of commands can be typed on a single line. Linux executes them from left to right.

```
$ clear;banner “hello”;sleep 10; clear; banner “hello TransEd”
```

****

The shell interprets some characters as special characters. To escape the special meaning of a character , it can be preceded by a backslash (\) . The backslash character is the shell’s escape character. The following characters can be preceded with a backslash to avoid its special meaning to be interpreted.

```
*  ?  [  ]  ;  &  <  >  |
```

```
$ echo  in this directory \* stands for *
```

The output will be

```
         in this directory * stands for ____
```

the filenames will be listed.

“ “

When a string on the command line is enclosed in a set of double quotes, the shell does not interpret the spaces as delimiters. The shell treats all characters enclosed in quotation marks as part of a single command.

```
$ a=*
$ echo $a
dira dirb myfile testfile

$ a=*
$ echo “$a”
*
```

If we want to use quotes within quotes, then also the backslash can be used.

```
$ echo “she said \”how are you\” “
she said “how are you”
$_
```

‘ ‘

Single quotation marks (‘ ‘) also turns off the special meaning of characters. The spaces are not treated as delimiters when they are enclosed in single quotes.

```
$ a=’*’
$ echo $a
dira dirb myfile testfile
$ echo ‘$a’
$a
```

``

The grave accent (`) is used for command substitution. When the shell encounters a command enclosed in a pair of grave accent , it executes the command and substitutes the command's output for the positional argument.

```
$ dir=`pwd`
$ dir
The output of pwd command  will be displayed.
```

It is possible to use more than one command in a single command line using command substitution. When a command is enclosed within backquotes, the command is replaced by the output it produces. This is called command substitution. The grave accent (`) is used to execute the command.

For example,

```
$ echo "today is `date`"
today is Mon May 14 14:39:33 IST 2001
$_
```

&

To run a command in the background, the & (ampersand) symbol can be placed at the end of the command. Linux assigns a process number which is displayed on the screen on the command being issued.

```
$ ls > filelist &
456
```

The & character enables the important feature of Linux operating system MultiTasking.

Shell Variables

There are various categories of shell variables like local variables, global variables, user-defined variables, referencing variables and environment variables.

The value of one variable can be assigned to another variable as shown below:

```
variable1 =${variable2}
```

Consider the following example,

```
$ a=21
```

If we want another variable x to have the value 21st, then

```
$ x=${a}st
```

Now x will contain the value 21st

In case, no concatenation is involved, we can write

```
$ x=$a
```

Local and global variables

The variables created in one shell are private or local to the shell. These variables are called local variables. When a new shell is created, it is unaware of the variables of the parent shell. Now the same variable can be given a different value without the parent shell knowing about it.

Consider the following example which demonstrates local variables:

```
$ course=java
$ echo $course
java

$ sh                    creates a new shell
$ echo $ course         No response is seen

$ course=linux          Assign a different value to course
$ echo $ course
linux
ctrl+d
$ exit                  return to parent shell

$ echo $course
java                    parent shell is unaware of linux
$ sh                    creates a new shell
$ echo $course                course has no value
ctrl+d
$ exit
$_
```

Sometimes it may be necessary for the child shell to know about the parent shell variables. The `export` command can be used to achieve this.

```
$ course=java
$ export course
$ echo $course
java
$ sh                    creates a new shell
$ echo $ course
java                    child shell has the variable course
$ course=linux          Assign a different value to course
$ echo $ course
linux
ctrl+d
$ exit                  return to parent shell

$ echo $course
java                    parent shell has the value of course
$_
```

Array Variables

The Bourne shell, sh, supports only scalar variables, which are the type of variables you have seen so far. The Korn shell, ksh, extends this to include array variables. Version 2.0 and later of the Bourne

Again shell, bash, also support array variables. The examples in the following section assume that you are using either ksh or bash 2.x or later.

Arrays provide a method of grouping a set of variables. Instead of creating a new name for each variable that is required, you can use a single array variable that stores all the other variables.
The difference between an array variable and a scalar variable can be explained as follows. Say that you are trying to represent the chapters in this book as a set of variables. Each of the individual variables is a scalar variable.

Some of these variables might be
CH01
CH02
CH15
CH07
Here is a format for each of the variable names: the letters CH followed by the chapter number. This format serves as a way of grouping these variables together. An array variable formalizes this grouping by using an array name in conjunction with a number that is called an index.

The simplest method of creating an array variable is to assign a value to one of its indices. This is expressed as follows:

```
name[index]=value
```

Here name is the name of the array, index is the index of the item in the array that you want to set, and value is the value you want to set for that item.

As an example, the following commands

```
$ FRUIT[0]=apple
$ FRUIT[1]=banana
$ FRUIT[2]=orange
```

set the values of the first three items in the array named FRUIT. You could do the same thing with scalar variables as follows:

```
$ FRUIT_0=apple
$ FRUIT_1=banana
$ FRUIT_2=orange
```

Although this works fine for small numbers of items, the array notation is much more efficient for large numbers of items. If you have to write a script using only sh, you can use this method for simulating arrays.

In the previous example, you set the array indices in sequence, but this is not necessary. For example, if you issue the command

```
$ FRUIT[10]=plum
```

the value of the item at index 10 in the array FRUIT is set to plum. One thing to note here is that the shell does not create a bunch of blank array items to fill the space between index 2 and index 10. Instead, it keeps track of only those array indices that contain values.

The second form of array initialization is used to set multiple elements at once.

In bash, the multiple elements are set as follows:

name=(value1 ... valuen)

Here, name is the name of the array and the values, 1 to n, are the values of the items to be set. When setting multiple elements at once, both ksh and bash use consecutive array indices beginning at 0. For example the ksh command

```
$ set –A band derri terry mike gene
```

or the bash command

```
$ band=(derri terry mike gene)
```

is equivalent to the following commands:

```
$ band[0]=derri
$ band[1]=terry
$ band[2]=mike
$ band[3]=gene
```

NOTE

When setting multiple array elements in bash, you can place an array index before the value:

myarray=([0]=derri [3]=gene [2]=mike [1]=terry)

The array indices don't have to be in order, as shown previously, and the indices don't have to be integers.

Accessing Array Values

After you have set any array variable, you access it as follows:

```
${name[index]}
```

Here name is the name of the array, and index is the index that interests us. For example, if the array FRUIT was initialized as given previously, the command

```
$ echo ${FRUIT[2]}
```

produces the following output:

```
orange
```

You can access all the items in an array in one of the following ways:

```
${name[*]}
${name[@]}
```

Here name is the name of the array you are interested in. If the FRUIT array is initialized as given previously, the command

```
$ echo ${FRUIT[*]}
```

produces the following output:

```
apple banana orange
```

If any of the array items hold values with spaces, this form of array access will not work and will need to use the second form. The second form quotes all the array entries so that embedded spaces are preserved.

For example, define the following array item:

```
FRUIT[3]="passion fruit"
```

Assuming that FRUIT is defined as given previously, accessing the entire array using the following command

```
$ echo ${FRUIT[*]}
```

results in five items, not four:

```
apple banana orange passion fruit
```

Commands accessing FRUIT using this form of array access get five values, with passion and fruit treated as separate items.

To get only four items, you have to use the following form:

```
$ echo ${FRUIT[@]}
```

The output from this command looks similar to the previous commands:

```
apple banana orange passion fruit
```

but the commands see only four items because the shell quotes the last item as passion fruit.

Read-only Variables

The shell provides a way to mark variables as read-only by using the read-only command. After a variable is marked read-only, its value cannot be changed.

Consider the following commands:

```
$ FRUIT=kiwi
$ readonly FRUIT
$ echo $FRUIT
kiwi
$ FRUIT= cantaloupe
```

The last command results in an error message:

```
/bin/sh: FRUIT: This variable is read only.
```

As you can see, the echo command can read the value of the variable FRUIT, but the shell did not enable us to overwrite the value stored in the variable FRUIT.

This feature is often used in scripts to make sure that critical variables are not overwritten accidentally.

In ksh and bash, the readonly command can be used to mark array and scalar variables as read-only.

Unsetting Variables

Unsetting a variable tells the shell to remove the variable from the list of variables that it tracks. This is like asking the shell to forget a piece of information because it is no longer required.

Both scalar and array variables are unset using the unset command:

```
unset name
```

Here name is the name of the variable to unset. For example,

```
unset FRUIT
```

unsets the variable FRUIT. You cannot use the unset command to unset variables that are marked readonly.

Environment variables

Linux has some variables that are already set such as PATH, PSI, HOME etc. By changing the values of these variables, a user can customize the environment. In Linux, each user is given a copy of the shell to work with. Each shell has its own environment that the user can uniquely configure.

The PATH variable

The PATH variable contains a list of colon-delimited path names of directories that are to be searched for an executable program. In Linux, the current directory is not searched automatically. Only the directories specified in the path are searched. For example, the command

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

specifies that the directories to be searched for any executable or command are /usr/bin and /bin in that order.

If the current directory is to be searched, it must be specified in the path by putting a dot somewhere in the path or by giving a null path.

Example

```
PATH=/bin::/usr/bin     searches in the order /bin, current
                              directory, /usr/bin
PATH=:/bin:/usr/bin     searches in the order current directory,
                                  /bin, /usr/bin
```

```
PATH=/bin:/usr/bin:     searches in the order /bin, /usr/bin, current
                                directory
```

The HOME variable

Every user in Linux has an associated directory called the HOME directory. When the user logs in, he is taken to the HOME directory. The location of the HOME directory is stored in the HOME variable.

The following command gives the home directory of the user

```
$ echo $HOME
```

The PS1 variable

The PS1 (Prompt String) variable contains the shell prompt $. The user can change the shell prompt by setting the value of this variable to the desired prompt.

For example to change the prompt to **&**, the following command can be used

```
$ PS1=″&>″
&>_
```

The new prompt should be enclosed within double quotes.

The PS2 variable

The PS2 variable sets a value for the secondary prompt, which is by default >. The secondary prompt appears when an incomplete command is given on the command line.

For example,

```
$ echo “hello how are
>″    ——————————————▶  secondary prompt appears because quotes
                       are not closed in the previous line
```

The LOGNAME variable

The LOGNAME variable contains the login name of the user.

```
$ echo $LOGNAME
```

The SHELL variable

The SHELL variable stores the user’s default shell.

```
$ echo $SHELL
/bin/sh
```

Shell Scripts

The commands we saw in the previous chapters were single line commands with pipes and filters. A group of commands can be grouped together under a single file name and the shell can be made to execute the commands. This file is called a shell script file.

Creating and executing a shell script

When the user first logs into the Linux system, a copy of the shell is given to him to work with. This shell is the login shell. Since the shell is itself a utility, the shell command `sh` can be used to create another shell. This new shell is known as the sub shell of the current shell. A child shell is created to execute a shell script so that the current shell is not affected by the script. The new shell is terminated as soon as the script running on it completes execution.

Consider the following shell script that asks the user for his name and welcomes him to Linux.

```
$ cat>welcome
echo “Enter your name:”
read name
echo “hi !! $ name welcome to Linux”
$_
```

The echo command is used to display messages on the screen. By default, the echo command displays the text and puts a newline character at the end of the line. The –n option can be used with the echo command to keep the cursor on the same line.

```
$ echo –n “Enter your name”
```

The read command is used to read value into a variable. The echo command prompts the user to enter the value and the read command waits for the user to enter a value for the variable.

The above script can be executed in two ways.

- The first method of executing the shell script is as given below:

```
$ sh welcome
Enter your name:
Harry
hi Harry welcome to Linux
$_
```

- The second method of executing the script is to change the permission of the file. By default, any file created in Linux does not have the execute permission. So, until the permissions are changed, the file cannot be directly executed by typing the file name at the shell prompt.

```
$ chmod u+x welcome
$ welcome
```

The # symbol is used to comment lines in shell scripts. On execution, the commented line will be ignored.

The expr command

Most shells do not support numeric values. All variables are treated as character strings. Consider the following declaration

```
a=35
```

it implies that the variable a contains the characters 3 and 5 and not the number 35. But we must be able to mathematically manipulate variables. Rather, we must simulate numeric values. This is possible by the use of the expr command.

```
$ expr 5 + 8
13
```

There must be a space before and after the operator.

Consider another example,

```
$ a=24
b=`expr $a + 4`
echo $b
$_
```

Finding the length of a string

The length of a string can be found using `expr` as shown below:

```
$ expr length "Hello"
5
```

Finding a substring of a string

```
$ expr substring "This is the string" 8 11
the string
```

Parameter handling in shell scripts

Shell scripts can be made to accept arguments from the command line. Thus, shell need not wait for the user to input data thereby making the execution faster. The arguments are named $1, $2, $3 etc. and are referred to within the script using these names. Since they represent their position, they are called positional parameters.

The shell interprets a command in Linux. When a command is entered and the <Enter> key is pressed, the shell puts each word on the command line into special variables as given below:

- The command name is put into a variable called $0
- The first argument of the command (second word on command line) is put into a variable $1
- The second argument of the command (third word on command line) is put into a variable $2 and so on.

We have some more shell variables that are set automatically. These are described in the table below:

Shell variable	Representation
$#	Number of positional parameters
$-	Shell options
$?	The exit status of the last executed command
$$	Process number of the current shell
$!	Process number of the last background command
$0	Name of the command being executed
$*	List of positional parameters

$@	Same as $*, except when enclosed in double quotes

Linux allows a maximum of nine variables other than $0.

HANDS-ON

```
case $# in
0)num=10;;
*)num=$1;;
esac
echo “Displaying numbers from 1 to $num”

while true
 do
  echo “$num”
  num=`expr $num – 1`
 if [$num –le 0]
  then
  echo “Good bye”
  exit
 fi
 done
```

The default argument will be executed if no input is given in the command line.

- **Shift**

The shift command is used to shift positional parameters. The positional parameters from $2.. are renamed.That is, using shift, we can access positional parameters above $9.

HANDS-ON

```
echo $ # arguments keyed in
echo they are -$*-
echo before shifting 1st argument is $1
echo before shifting 9th argument is $9
echo argument det after shifting
echo
shift
echo $# argument are now available
echo they are -$*-
echo after shifting 1st argument is $1
echo after shifting 9th argument is $9
```

Save the above file as argtest. Execute the file as given below:

```
$ argtest 10 20 30 40 50 60 70 80 90 100
```

Try replacing shift 2 in place of shift in the above code and execute the program to watch the output.

Exit status of a command

We have used linux commands; some commands executed in the way we wanted and some didn't. Say the command

$ cat xyz
cat: can't open xyz

generates an error message to the standard error, probably on account of the file xyz not present or being unreadable.Every command returns a value after execution. This value is called the exit status or return value of the command.A command is said to be true if it executes successfully, and false if it fails. The cat command above returns a false exit status.

The return values are of importance to the programmer because he can devise the logic of the program which branches into different paths, depending on the success/failure of a command.

The parameter $? stores the exit status of the last command.It contains the value 0 if the command succeeds and non-zero value if the command fails.

Special parameters used by the shell

Shell parameter	Significance
$0	Name of the executed command
$*	Complete set of positional parameters - $1,$2 etc
$#	Number of arguments in command line
$?	Exit status of a command

Conditional execution

The logical operators && and || can be used for conditional execution. The && command is used by the shell to delimit two commands; the second command is executed only when the first succeeds. It can be used with grep command in the following way:

$ grep 'manager' emp.txt && echo "pattern found"

The above command will search for the word manager and display the lines containing the pattern and displays pattern found at the end of the display.

The || operator is used to execute the command following it only when the previous command fails. With the help of the following command, the failure of the grep command can be notified.

$ grep 'manager' emp.txt || echo "pattern not found"

The above command will search for the word manager and display the lines containing the pattern. If the pattern is not found, pattern not found is displayed.The above two operators can be used in developing shell scripts.

Script Termination

The exit statement is used to prematurely terminate a program. When this statement is encountered in a shell script , execution is halted and the control is returned to the calling program, in most cases the shell. The exit statement is used with a command when it fails.

```
echo “enter the string to search for”
read str
echo “enter the file name”
read fname
grep $str $fname || exit 2
echo “Pattern found-job over”
```

The exit statement can take an argument. The argument is optional, but when one is specified, the script will terminate with a return value as specified with the statement.If no return value is specified, then the value returned will be zero(true). This value is assigned to the parameter $?.

REVIEW QUESTIONS

1. The _____ symbol is used to refer to the contents of a variable
2. The _____ variables are known to the child shell also.
3. Assume the variable name to contain the word apple, echo ‘$name’ will output _____, echo “$name” will output _____.
4. $* represents _____.
5. The _____ symbol is used to insert comment entries in shell scripts.
6. The grave accent is used for _____.