# Lab02:Programming Using Shell Scripting

## Objectives

1. **Getting familiar with shells**
2. **Write shell script**
3. **Run shell script**

## 1. What is Shell Program

When a user enters commands from the command line, he is entering them one at a time and getting a response from the system. From time to time it is required to execute more than one command, one after the other, and get the final result. This can be done with a ***shell program or a shell script***. A shell program is a series of Linux commands and utilities that have been put into a file by using a text editor. When a shell program is executed the commands are interpreted and executed by Linux one after the other.

## 2. Common Shells.

**C-Shell - csh** : The default on teaching systems Good for interactive systems Inferior programmable features

**Bourne Shell - bsh or sh - also restricted shell - bsh** : Sophisticated pattern matching and file name substitution

**Korn Shell** : Backwards compatible with Bourne Shell Regular expression substitution emacs editing mode

**Thomas C-Shell - tcsh** : Based on C-Shell Additional ability to use emacs to edit the command line Word completion & spelling correction Identifying your shell.

## 3. General Things

**The shbang line**
The "shbang" line is the very first line of the script and lets the kernel know what shell will be interpreting the lines in the script. The shbang line consists of a #! followed by the full pathname to the shell, and can be followed by options to control the behavior of the shell.
***EXAMPLE***

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

**Comments**
Comments are descriptive material preceded by a # sign. They are in effect until the end of a line and can be started anywhere on the line.
***EXAMPLE***

```
# this text is not
# interpreted by the shell
```

## 4. Assigning values to variables

A value is assigned to a variable simply by typing the variable name followed by an equal sign and the value that is to be assigned to the variable. For example, if you wanted to assign a value of 5 to the variable count, you would enter the following command:

```
count=5
```

## 5. Accessing variable values

To access the value stored in a variable precede the variable name with a dollar sign ($). If you wanted to print the value stored in the count variable to the screen, you would do so by entering the following command:

```
echo $count
```

If you omitted the $ from the preceding command, the echo command would display the word count on-screen.

## 6. Positional parameters

The shell has knowledge of a special kind of variable called a positional parameter. Positional parameters are used to refer to the parameters that were passed to a shell program on the command line or a shell function by the shell script that invoked the function. When you run a shell program that requires or supports a number of command-line options, each of these options is stored into a positional parameter. The first parameter is stored into a variable named 1, the second parameter is stored into a variable named 2, and so forth. These variable names are reserved by the shell so that you can't use them as variables you define. To access the values stored in these variables, you must precede the variable name with a dollar sign ($) just as you do with variables you define.

The following shell program expects to be invoked with two parameters. The program takes the two parameters and prints the second parameter that was typed on the command line first and the first parameter that was typed on the command line second.

```
#program reverse, prints the command line parameters out in reverse #order
    echo "$2" "$1"
```

If you invoked this program by entering

```
reverse hello there
```

The program would return the following output:

```
there hello
```

## 7. Built-in Variables

These are special variables that Linux provides that can be used to make decisions in a program. Their values cannot be modified.
Several other built-in shell variables are important to know about when you are doing a lot of shell programming. The following table lists these variables and gives a brief description of what each is used for.

| *Variable* | **Use** |
|---|---|
| $# | Stores the number of command-line arguments that were passed to the shell program. |
| $? | Stores the exit value of the last command that was executed. |
| $0 | Stores the first word of the entered command (the name of the shell program). |
| $* | Stores all the arguments that were entered on the command line ($1 $2 ...). |
| "$@" | Stores all the arguments that were entered on the command line, individually quoted ("$1" "$2" ...). |

## 8. The importance of quotation marks

The use of the different types of quotation marks is very important in shell programming. Both kinds of quotation marks and the backslash character are used by the shell to perform different functions. The double quotation marks (""), the single quotation marks (''), and the backslash (\) are all used to hide special characters from the shell. Each of these methods hides varying degrees of special characters from the shell.

The double quotation marks are the least powerful of the three methods. When you surround characters with double quotes, all the whitespace characters are hidden from the shell, but all other special characters are still interpreted by the shell. This type of quoting is most useful when you are assigning strings that contain more than one word to a variable. For example, if you wanted to assign the string hello there to the variable greeting, you would type the following command:

```
greeting="hello there"
```

This command would store the hello there string into the greeting variable as one word. If you typed this command without using the quotes, you would not get the results you wanted.

Single quotes are the most powerful form of quoting. They hide all special characters from the shell.
If the string being assigned to the greeting variable contained another variable, you would have to use the double quotes. For example, if you wanted to include the name of the user in your greeting, you would type the following command:

```
greeting="hello there $LOGNAME"
```

Remember that the LOGNAME variable is a shell variable that contains the Linux username of the person who is logged in to the system.

This would store the value hello there root into the greeting variable if you were logged in to Linux as root. If you tried to write this command using single quotes it wouldn't work, because the single quotes would hide the dollar sign from the shell and the shell wouldn't know that it was supposed to perform a variable substitution. The greeting variable would be assigned the value hello there $LOGNAME if you wrote the command using single quotes.

Using the backslash is the third way of hiding special characters from the shell. Like the single quotation mark method, the backslash hides all special characters from the shell, but it can hide only one character at a time, as opposed to groups of characters. You could rewrite the greeting example using the backslash instead of double quotation marks by using the following command:

```
greeting=hello\ there
```

In this command, the backslash hides the space character from the shell, and the string hello there is assigned to the greeting variable.
Backslash quoting is used most often when you want to hide only a single character from the shell. This is usually done when you want to include a special character in a string. For example, if you wanted to store the price of a box of computer disks into a variable named disk_price, you would use the following command:

```
disk_price=\$5.00
```

The backslash in this example would hide the dollar sign from the shell. If the backslash were not there, the shell would try to find a variable named 5 and perform a variable substitution on that variable. Assuming that no variable named 5 were defined, the shell would assign a value of .00 to the disk_price variable. This is because the shell would substitute a value of null for the $5 variable.
The disk_price example could also have used single quotes to hide the dollar sign from the shell.

The back quote marks (`) perform a different function. They are used when you want to use the results of a command in another command. For example, if you wanted to set the value of the variable contents equal to the list of files in the current directory, you would type the following command:

```
contents='ls'
```

This command would execute the ls command and store the results of the command into the contents variable. As you will see in the section "Iteration Statements," this feature can be very useful when you want to write a shell program that performs some action on the results of another command.

9. **READ Statement :**
   To get the input from the user.
   `read x y` (no need of commas between variables)

## 10.Test Command

Test command is used to evaluate conditional expressions. You would typically use the test command to evaluate a condition that is used in a conditional statement or to evaluate the entrance or exit criteria for an iteration statement. The test command has the following syntax:

```
test expression
or
[ expression ]
```

Several built-in operators can be used with the test command.

The shell integer operators perform similar functions to the string operators except that they act on integer arguments. The following table lists the test command's integer operators.
**The Test Command's Integer Operators**

| **Operator** | **Meaning** |
|---|---|
| Int1 -eq int2 | Returns True if int1 is equal to int2. |
| Int1 -ge int2 | Returns True if int1 is greater than or equal to int2. |
| Int1 -gt int2 | Returns True if int1 is greater than int2. |
| Int1 -le int2 | Returns True if int1 is less than or equal to int2. |
| Int1 -lt int2 | Returns True if int1 is less than int2. |
| Int1 -ne int2 | Returns True if int1 is not equal to int2. |

The string operators are used to evaluate string expressions. The table below lists the string operators that are supported by the three shell programming languages.

**The Test Command's String Operators**

| **Operator** | **Meaning** |
|---|---|
| Str1 = str2 | Returns True if str1 is identical to str2. |
| Str1 != str2 | Returns True if str1 is not identical to str2. |
| str | Returns True if str is not null. |
| -n str | Returns True if the length of str is greater than zero. |
| -z str | Returns True if the length of str is equal to zero. |

The test command's file operators are used to perform functions such as checking to see if a file exists and checking to see what kind of file is passed as an argument to the test command. The following is the list of the test command's file operators.

The test command's logical operators are used to combine two or more of the integer, string, or file operators or to negate a single integer, string, or file operator. The table below lists the test command's logical operators.

**The Test Command's Logical Operators**

| **Command** | **Meaning** |
|---|---|
| ! expr | Returns True if expr is not true. |
| expr1 -a expr2 | Returns True if expr1 and expr2 are true. |
| expr1 -o expr2 | Returns True if expr1 or expr2 is true. |

## 11. Conditional Statements

There are  two forms of conditional statements. These are the if statement and the case statement. These statements are used to execute different parts of your shell program depending on whether certain conditions are true. As with most statements, the syntax for these statements is slightly different between the different shells.

## 12. The if Statement

All three shells support nested if...then...else statements. These statements provide you with a way of performing complicated conditional tests in your shell programs. The syntax of the if statement is the same for bash and pdksh and is shown here:

```
if [ expression ]
then
commands
elif [ expression2 ]
then
commands
else
commands
fi
```

## 13. The case Statement

The case statement enables you to compare a pattern with several other patterns and execute a block of code if a match is found.

```
case string1 in
str1)
commands;;
str2)
commands;;
*)
commands;;
esac
```

String1 is compared to str1 and str2. If one of these strings matches string1, the commands up until the double semicolon (;;) are executed. If neither str1 nor str2 matches string1, the commands associated with the asterisk are executed. This is the default case condition because the asterisk matches all strings.

The shell languages also provide several iteration or looping statements. The most commonly used of these is the for statement.

## 14. The for Statement

The ‘for’ statement executes the commands that are contained within it a specified number of times. The ‘bash’ and ‘pdksh’ have two variations of the for statement.

```
for var1 in list
do
commands
done


for i in {2..10}
do
    echo "output: $i"
done
```

## 15. The while Statement

Another iteration statement offered by the shell programming language is the while statement. This statement causes a block of code to be executed while a provided conditional expression is true.

```
while expression
do
statements
done
```

Care must be taken with the while statements because the loop will never terminate if the specifies condition never evaluates to false.

## 16. The break Statement

The break statement can be used to terminate an iteration loop.

## 17. The exit Statement

The exit statement can be used to exit a shell program. A number can be optionally used after exit. If the current shell program has been called by another shell program, the calling program can check for the code and make a decision accordingly.

## 18. Functions

The shell languages enable you to define your own functions. These functions behave in much the same way as functions you define in C or other programming languages. The main advantage of using functions as opposed to writing all of your shell code in line is for organizational purposes. Code written using functions tends to be much easier to read and maintain and also tends to be smaller, because you can group common code into functions instead of putting it everywhere it is needed.

```
fname () {
shell commands
}
```

**Tasks:**

1. write a shell program to read two strings and concatenate them.
2. write a shell program to read two string and compare them that its equal or not.
3. write a shell program to find greatest of three numbers.
4. write a shell program to perform the arithmetic operations using case