UNIT 3

SHELL PROGRAMMING

Shell programs or shell scripts are files containing Linux commands, comments, and control structures. Any command that can be executed at the shell prompt can be included in a shell program. The shell provides very powerful features for writing shell programs. Many of the shell programs are used at start up and shutdown time.

A shell script is a group of commands, functions, variables, or just about anything else you can use from a shell. These items are typed into a plaintext file. That file can then be run as a command. You can create your own shell scripts to automate the tasks you need to do regularly. While dozens of different shells are available in Linux, the default shell is called *bash*, the Bourne Again Shell.

Various types of Shells

There are different types of shells are available.

1. The Bourne Shell
2. The C Shell
3. The Korn Shell
4. The GNU Bourne-Again Shell
5. TC Shell
6. Restricted Shell
7. A Shell
8. Z Shell

The Bourne Shell

This is the original Unix shell written by *Steve Bourne* of AT & T Bell Labs. It is available on all UNIX systems. The Bourne shell does provide an easy to use language with which you can write shell scripts. It is the preferred shell for shell programming because of its compactness and speed. It is distributed with all UNIX systems and is stored in the */bin* directory. The executable file name of Bourne shell is '**sh**'. A Bourne shell drawback is that it lacks features for interactive use, such as the ability to recall previous commands (history). The Bourne shell also lacks built-in arithmetic and logical expression handling.

For the Bourne shell the:

- Command full-path name is: */bin/sh*
- Non-root user default prompt is: $
- Root user default prompt is: #

The C Shell

It was developed by *William Joy* at the University of California at Berkeley to provide a programming interface similar to the C programming language. The C shell derives its name from C programming language, which resembles the C programming language in syntax. The executable file name for this shell is '**csh**'. It incorporates the features for interactive use, such as aliasing of commands and command history. It also includes convenient programming features, such as built-in arithmetic and a C-like expression syntax.

For the C shell the:

- Command full-path name is: */bin/csh*
- Non-root user default prompt is: %
- Root user default prompt is: #

The Korn Shell

It was developed by *David Korn* at AT&T Bell Labs. The Korn shell combines the features of both the Bourne and C shells. Its executable file

name is '**ksh**'. The Korn shell's command editor interface enables the quick, effortless correction of typing errors, plus easy recall and reuse of command history. It includes convenient programming features like built-in arithmetic and C-like arrays, functions, and string-manipulation facilities and is faster than the C shell.

For the Korn shell the:

- Command full-path name is: */bin/ksh*
- Non-root user default prompt is: $
- Root user default prompt is: #

Bourne Again Shell (bash)

This is a public domain shell written by the Free Software Foundation under their GNU project. Bash provides all the interactive features of the C shell (csh) and the Korn shell (ksh). Its programming language is compatible with the Bourne shell (sh). It is an enhancement of Bourne shell. Bash is an acronym for Bourne Again Shell.

Bash shell is a default shell for most Linux systems. It is stored in the */bin* directory. It stores the commands that we store in a session. It also stores the commands that we used in the previous session. The executable file name is '**bash**'. In Red Hat Linux, the *sh* command is a symbolic link to bash.

For the GNU Bourne-Again shell the:

- Command full-path name is: */bin/bash*
- Default prompt for a non-root user is: bash-x.xx$. (Where x.xx indicates the shell version number. For example, bash-3.50$)
- Root user default prompt is: bash-x.xx#. (Where x.xx indicates the shell version number. For example, bash-3.50$#)

TC Shell or Tcsh Shell

Tcsh stands for Tom's C shell and is an enhancement of the C shell. It is also known as the TC shell. In Linux the *csh* command is a symbolic link to the Tcsh shell. We can execute the Tcsh shell by typing either *csh* or *tcsh*

at the command prompt. This shell is available in the public domain. It provides all the features of the C shell together with *emacs* style editing of the command line. (*emacs* is a text editor which offers longer list of commands).

Restricted shell

Restricted shell is used to provide limited access on the operating system by the user. The restricted shell is typically used for guest users who only need limited rights and permissions.

A shell

A shell was developed by *Kenneth Almquist*. It emulates Bourne shell. A shell is suitable for the computers having limited memory. The executable file name for the A shell is **'ash'**.

Z shell

Z shell offers the features of *Tcsh* and *Korn* shell. It provides a large number of utilities and extensive documentation. The executable file name for the Z shell is **'zsh'**. It sports a number of useful features, including spelling correction, theming, nameable directory shortcuts, sharing your command history across multiple terminals etc.

Comparison between various Shells

Feature	Bourne	Korn	C	Tcsh	Bash
Background processing	Yes	Yes	Yes	Yes	Yes
Command history	No	Yes	Yes	Yes	Yes
I/O redirection	Yes	Yes	Yes	Yes	Yes
Shell scripts	Yes	Yes	Yes	Yes	Yes
Command alias	No	Yes	Yes	Yes	Yes
File name completion	No	Yes	No	Yes	Yes
Command completion	No	No	No	Yes	yes
Command line editing	No	Yes	No	Yes	Yes
Job control	No	Yes	Yes	Yes	Yes

Steps to create a Shell Script

1. Create a file using a vi editor (or any other editor). Name script file with extension **.sh**.
2. Start the script with **#! /bin/bash** (name of the interpreter).
3. Write some code.
4. Save the script file as **filename.sh**
5. To execute the script use any of these two steps:
   a) Give execute permission for the file using chmod command **chmodu+x filename.sh** or chmod 755 filename.sh and then run using **./filename.sh**
   b) Use **bash filename.sh**

Example of a simple shell script

```
#!/bin/bash
echo "Our first script"
echo "_____"
echo "Welcome to BASH Programming"
```

Save this script with name 'first.sh'. To execute, use the following commands:

```
chmod u+x first.sh
./first.sh
```

Output:

```
Our first script
_____
Welcome to BASH Programming
```

Shell variables

Shell variables are classified into two types as follows:

(1) **Bulit-in shell variables** - Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS.

(2) **User defined shell variables (UDV)** - Created and maintained by user. This type of variables defined in lower letters.

1. Built-in shell variables

These variables are created and maintained by the Linux system. These variables are used to define an environment. So, it is also called as "*environmental variables*". An environment is an area in memory where we can place definitions of Shell variables so that they are accessible from the programs. We can list these system variables corresponding to a user, using the set command. In Linux, the system variables are specified by uppercases for convenience. To see all the environment variables use ***env*** or ***printenv*** command. Some of the built-in variables are given below:

System Variables	Meanings
BASH=usr/bin/bash	Shell name
BASH_VERSION=1.14.7(1)	Shell version name
COLUMNS=80	No. of columns for the screen
HOME	Contains the path name of home directory
LINES=25	No. of columns for the screen
LOGNAME	Contains user's login name
OSTYPE=Linux	OS type
PATH=/usr/bin:/sbin:/bin:/usr/sbin	Contains the directories in which the Shell will search for files to execute the commands that are given by the users
PS1	Prompt string 1, normally PS1 is $
PS2	Prompt string 1, normally PS1 is >. It is used when Linux thinks that you have started a new line without hitting *Enter* key.
PWD	Current working directory
SHELL=/bin/bash	shell name
USERNAME	User name who is currently login to this PC
MAIL	Contains the name of the directory in which mails addressed to the user are placed

2. User-defined Shell variables

These variables are defined by users. A shell script allows us to set and use our own variables within the script. Setting variables allows you to temporarily store data and use it throughout the script. Unlike formal programming languages, a shell script doesn't require you to declare a type for your variables. The shell script automatically determines the data type based on the data that the user stores into it during runtime. Variables defined within the shell script maintain their values throughout the life of the shell script but are deleted when the shell script completes.

The syntax for creating a variable is as follows:

<variablenmae>=<value>

Note that there must not be a space on either of the equal sign. If the <value> contains blank spaces, then it should be enclosed within double quotes.

Rules for naming shell variables

1. The variables must begin with a letter or underscore character.
2. The variable name contains only letters, numbers or underscores characters.
3. Variable names are case sensitive.

Examples:

```
Num=10
netpay=14000
name="student"
```

To access the value stored in a variable, prefix its name with the **dollar sign ($).**

To display the value of a particular variable, use:

echo $variablename

Examples:

```
echo $Num
echo $name
```

echo

This command is used to display values on the screen. General format is:

echo <string> [<$variablename>]

The symbol $ prefixed to a variable gives the value of that variable. The echo without any argument produces an empty line.

Whenever the Shell finds continuous spaces and tabs in the command line, it compresses them to a single space. That's why, when you issue the following echo command, you find the output to be compressed.

```
echo GOD        IS      LOVE
GOD IS LOVE
```

To preserve the spaces, you have to place the string within quotes.

```
echo "GOD       IS      LOVE"
GOD             IS      LOVE
```

The echo command will generate carriage return by default. You can use the echo command with **–n** option to avoid such automatic generation of the carriage return.

```
echo "Welcome"
echo " To Linux"
```

Output:

```
Welcome
To Linux
```

```
echo -n "Welcome"
echo " To Linux"
```

Output: Welcome to Linux

The **–e** option of the echo command enables the interpretation of the following character sequences in the argument string.

Escape Sequences	Meanings
\b	Back space (removes all the spaces in between the text
\n	New line
\t	A horizontal tab
\v	A vertical tab.
\a	Alert (beep sound)
\c	Produce no further output after this.
\\	Back slash
\'	Single Quote
\"	Double Quote

read

This command is used to read a line of text from the standard input device and it assigns the read value on the specified variable. General format is:

read <variablename>

Example: read num

Read name

Back quotes (‘ ‘)

The back quotes (nearest to the *Esc* key) turn off the special meanings of all enclosed characters. And when a command is enclosed within back quotes, the command is replaced by the output that it produces.

Example:

echo Today’s date is date

Ouput : Today’s date is date

echo Today’s date is ‘date‘

Output: Today’s date is Mon July 04 10:21:26

It displays the output of command *date*.

Command Terminating Characters

Character	Meaning
;	Separates the commands when more than one command is given in a line. Command1; Command2 Executes the command1 and then executes command2
&	Like ; character, but does not wait the previous command to complete. Command 1 & Command 2 Unlike the ; character, the command2 will not wait for the completion of command1

Comment Character

#	If an # Symbol appears in a line, then the rest of the line will be treated as comment. For example: # This is a comment

Shell Keywords

Keywords are the words whose meaning has already been explained to the shell. The keywords cannot be used as a variable name because of it is a reserved word with containing reserved meaning.

echo	read	set	unset
readonly	Shift	export	if
fi	Else	while	do
done	For	until	case
esac	Break	continue	exit
return	Trap	wait	eval
exec	ulimit	umask	

Arithmetic Expansion

Arithmetic expansion allows the evaluation of an arithmetic expression and the substitution of the result. There are several ways to do arithmetic in Bash scripting.

1. Double parentheses
2. *expr* command
3. *let* command

1. Double parentheses

The double parentheses permit arithmetic expansion and evaluation. The syntax is as follows

$((expression))

Example: sum=$((3 + 5))

Sum=$((a+b))

2. expr command

The *expr* command evaluates a given expression and displays its corresponding output. To compute the result we enclose the expression in back ticks ‘ ‘. The syntax is as follows:

expr expression

Example: sum=‘expr 3 + 5‘

sum=‘expr $a + $b‘

Note that there must be spaces between the items.

The *expr* command can only work with integer values. For floating point numbers we use the ***bc*** command. You must enclose the variables in double quotes while using *bc* to handle floating point numbers.

Eg: a=10.5

b=5

Sum=‘expr “$a + $b” | bc‘

3. let command

let is a builtin function of Bash that allows us to do simple arithmetic. The format is:

let <arithmetic expression>

Example: let sum=3+5

let sum=$a+$b

Note that, you don't put quotes and space in the expression.

Operators

There are various operators supported by each shell.

- Arithmetic Operators
- Relational Operators
- Boolean Operators
- String Operators
- File Test Operators

1. Arithmetic Operators

These operators are used to perform normal arithmetic/mathematical operations. There are 8 arithmetic operators:

Operator	Description	Example	Evaluates To
+	Addition	echo $((20 + 5))	25
-	Subtraction	echo $((20 - 5))	15
/	Division	echo $((20 / 5))	4
*	Multiplication	echo $((20 * 5))	100
%	Modulus	echo $((20 % 3))	2
++	post-increment (add variable value by 1)	x=5 echo $((x++)) echo $((x++))	5 6
--	post-decrement (subtract variable value by 1)	x=5 echo $((--x))	4
**	Exponentiation	x=2 y=3 echo $ ((x ** y))	8

Note: In shell, * represents all files in the current directory. So, in order to use * as a multiplication operator, we should escape it with a backslash (*). If we directly use * in *expr*, we will get error message.

2. Relational Operators

Relational operators are those operators which defines the relation between two operands. They give either true or false depending upon the relation. Bash Shell supports the following relational operators that are specific to numeric values. These operators do not work for string values unless their value is numeric.

Operator	Description	Example (a=10 b=20)
-eq	Checks if the values of two operands are equal or not; if yes, then the condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the values of two operands are equal or not; if values are not equal, then the condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.	[$a -lt $b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand; is true. if yes, then the condition becomes true.	[$a -le $b]

Note that there must be spaces between the operators and the expressions.

3. Boolean operators (Logical Operators)

These are used to perform logical operations. The following Boolean operators are supported by the Bash Shell.

Operator	Description	Example (a=10, b=20)
!	This is a unary operator which returns true if the operand is false and returns false if the operand is true.	[! false] is true.
-o	This is logical OR. If one of the operands is true, then the condition becomes true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical AND. If both the operands are true, then the condition becomes true otherwise false.	[$a -lt 20 -a $b -gt 100] is false.

4. String Operators

The following string operators are supported by Bash Shell. Assume variable *a* holds "Linux" and variable *b* holds "Unix" then–

Operator	Description	Example
=	Checks if the values of two operands are equal or not; if yes, then the condition becomes true.	[$a = $b] is not true.
!=	Checks if the values of two operands are equal or not; if values are not equal then the condition becomes true.	[$a != $b] is true.
-z	Checks if the given string operand size is zero; if it is zero length, then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true.	[-n $a] is not false.

5. File Test Operators

We have a few operators that can be used to test various properties associated with a file.

Assume a variable *file* holds an existing file name "*test*", the size of which is 100 bytes and has read, write and execute permission:

Operator	Description	Example
-b	Checks if file is a block speeial file; if yes, then the condition becomes true.	[-b $file] is false.
-c	Checks if file is a character special file; if yes, then the condition becomes true.	[-c $file] is false.
-d	Checks if file is a directory; if yes, then the condition becomes true.	[-d $file] is not true.
-e	Checks if file exists; if exists, the return true.	[-e $file] is true.
-r	Checks if file is readable; if yes, then the condition becomes true.	[-r $file] is true.
-w	Checks if file is writable; if yes, then the condition becomes true.	[-w $file] is true.
-x	Checks if file is executable; if yes, then the condition becomes true.	[-x $file] is true.
-s	Checks if file has size greater than 0; if yes, then condition becomes true.	[-s $file] is true.
-f	Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true.	[-f $file] is true.
-g	Checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true.	[-g $file] is false.

-k	Checks if file has its sticky bit set; if yes, then the condition becomes true.	[-k $file] is false.
-p	Checks if file is a named pipe; if yes, then the condition becomes true.	[-p $file] is false.
-t	Checks if file descriptor is open and associated with a terminal; if yes, then the condition becomes true.	[-t $file] is false.
-u	Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true.	[-u $file] is false.

Conditional Statements or Branching Statements

In general, instructions in shell scripts are executed sequentially, i.e. in the same order in which they appear in the shell scripting. A branching or decision control statements, can execute a set of instruction in one situation, and an entirely different set of instructions to be executed in another situation. This kind of situation is dealt in shell scripting using a decision control instruction. Linux has following types of branching statements:

- The if statement
- The case statement

If statements

If statements are useful decision-making statements which can be used to select an option from a given set of options. Linux Shell supports following forms of if statement

- if...fi statement
- if...else...fi statement
- if...elif...else...fi statement

test command

The test command is used to evaluate expressions and generate a return code. It takes arguments that form logical expressions and evaluates the

expressions. The test command writes nothing to standard output. The return code will be set to 0 if the expression evaluates to true, and the return code will be set to 1 if the expression evaluates to false. The test command is initially presented alone so that you can display the return codes. But it is most commonly used with the *if* and *while* constructs to provide conditional flow control. General format is:

test <expression >

The test command can evaluate the condition of:

- Integers
- Strings
- Files

Example: test $a –gt $b

1) The if…then…fi Structure

The *if..then..fi* structure is used to check a condition. If the condition is true, then *Statement-block* is executed. If the condition is false (not true), the *Statement-block* is not executed. General format is:

if [conditional-command]
then
Statement- block
fi

Note that there must be spaces within square brackets.

Example:
```
#!/bin/bash
a=10
b=10
if [ $a -eq $b ]          # if test $a –eq $b
then
        echo "a and b are equal"
fi
```

Output:

```
a and b are equal
```

2) The if-then-else-fi Structure

This structure is used when you want to perform one of two actions depending on the result of a condition. The if-else construct allows you to execute one set of commands (*true block statements*) if the return code of the conditional command is 0 (true) or another set of commands (*false block statements)* if the return code of the conditional command is non-zero (false). General format is:

```
if [ conditional command ]
then
        True-block Statements
else
        False-block statements
fi
```

Example:

```
#!/bin/bash
a=10
b=10
if [ $a -eq $b ]
then
        echo "a and b are equal"
else
        echo "a and b are not equal"
fi
```

Output:

```
a and b are equal
```

3) The if...elif...else...fi statement

elif (which stands for "else if") is used to test for an additional condition. To use multiple conditions in one if-else block, then *elif* keyword is used in shelf. If condition1 is true then it executes statement-block1, and this process continues. If none of the condition is true then it executes default-statements in else part. General format is:

```
if [ condition1 ]
then
        statement-block1
elif [ condition2 ]
then
    statement-block 2
.
.
.
elif [ conditionN ]
 then
    statement-block N
else
        default statements
fi
```

Example

```
#!/bin/bash
a=10
b=20
if [ $a -gt $b ]
then
        echo "$a is greater than $b"
elif [ $a -lt $b ]
then
        echo "$a is less than $b"
else
        echo "Numbers are equal"
fi
```

Output:

```
10 is less than 20
```

4) Nested if statement

Nested if-else block can be used when, one condition is satisfies then it again checks another condition. General format is

```
if [ condition1 ]
then
        if [ condition2 ]
        then
                statement-block1
        else
                statement- block2
        fi
else
        statement-block3
fi
```

Example

```
#!/bin/bash
a=20
b=10
if [ $a -gt $b ]
then
        if [ $a –gt 50 ]
        then
                echo “$a is greater than $b and is greater than
        50”
        else
                echo “$a is greater than $b and is less than
        50”
        fi
else
        echo “$a is less than $b”
fi
```

Output:

```
20 is greater than 10 and is less than 50
```

The case Statement

The case construct provides a convenient syntax for multi-way branching. The branch selected is based on the sequential comparison of a word and supplied patterns. These comparisons are strictly ***string-based***. When a match is found, the corresponding list of commands will be executed. Each list of commands is terminated by a double semicolon (;;). After finishing the related list of commands, program control will continue at the *esac*. Genera format is:

```
case var in
pattern1)
        statement block1
        ; ;
pattern2)
        statement block2
        ; ;
...
...
pattern n)
        statement block n
        ; ;
*)
        default block
        ; ;
        esac
```

The value of *var* is checked. If this is equal to *pattern1*, the commands in the statement-block1 are executed. The first block ends at the ;; pattern. If the value of *var* matches *pattern2*, the commands in the statement-block2 are executed and soon. If *var* matches none of the pattern values, the commands in the last block after "*)" are executed.

Example:

```
#!/bin/bash
echo "Enter Two Numbers"
read a b
echo "Enter the Operator"
read op
case $op in
+)
        c=`expr $a + $b`
        echo "$a + $b = $c"
        ;;
-)
        c=`expr $a - $b`
        echo "$a - $b = $c"
        ;;
\*)
        c=`expr $a \* $b`
        echo "$a * $b = $c"
        ;;
/)
        c=`expr $a / $b`
        echo "$a / $b = $c"
        ;;
*) echo "Invalid Operator"
esac
echo "The case structure finished"
```

Output:

```
Enter Two Numbers
20 10
Enter the Operator
-
20 – 10 = 10
```

The looping Statements or Iterative statements

The looping statements are used to execute a sequence of commands repeatedly based on some conditions. The test command is frequently used to control the continuance of a loop. Unlike branches, which start with a keyword and end with the keyword in reverse (if/fi and case/esac), loops will start with a keyword and some condition, and the body of the loop will be surrounded by *do/done*.

There are three basic types of loops:

- while-do-done loop
- until-do-done loop
- for-do-done loop

1. while-do-done loop

The while-do-done loop checks for a condition and goes on executing a block of statements **until the condition becomes false** (in other words, while the condition is true). i.e, it tests a condition before the execution of the block of statements contained inside the loop. The loop continues to execute as long as the condition remains true. General format is:

while [condition]
do
Statements
done

The execution is as follows: Commands in condition are executed first. If the condition is 0 (true), execute body of the loop and evaluate the condition again. If the condition is not 0 (false), skip to the first command following the *done* keyword.

Example:

```
#!/bin/bash
i=1
while [ $i -le 10 ]                    #while test $i -le 10
do
```

```
        echo -n "$i"
        i=`expr $i + 1`
    Done
```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

2. Until-do-done loop

The until-do-done loop is like the while-do-done loop. The only difference is that it tests the condition and goes on executing as long as the condition remains false. ie, the until-do-done loop repeats the execution of a block of statements **until the condition becomes true** (in other words, while the condition is false). General format is:

until [condition]
do
Statements
done

The execution is as follows: Command condition is executed first. If the condition is not 0 (false), execute the body of the loop and evaluate the condition again. If the condition is 0 (true), skip to the first command following the *done* keyword.

Example:

```
#!/bin/bash
i=1
until [ $i -eq 10 ]                    #until test $i -eq 10
do
        echo -n "$i"
        i=`expr $i + 1`
done
```

Output:

```
1 2 3 4 5 6 7 8 9
```

3. for-do-done loop

The for-do-done loop, which is used to execute a block of commands for a fixed number of times. It is **executed on a list of elements**. The list of elements is assigned to a variable one-by-one. The value of this variable is processed inside the loop. The loop continues to execute until all of the list elements are processed and there are no more elements in the list. The general syntax of for-do-done loop is:

for var in list
do
Statements
done

Each element in the list is separated from the next by whitespace. The construct works as follows: The shell variable *var* is set equal to the first string in list and body of the loop is executed. Then the variable *var* is set equal to the next string in list and body of the loop is executed. This process continues until all items from list have been processed.

Example:

```
#!/bin/bash
for NUM in 1 2 3 4 5 6 7 8 9 10
do
        echo "The number is $NUM"
done
```

Output:

```
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5
```

```
The number is 6
The number is 7
The number is 8
The number is 9
The number is 10
```

We can use C programming like for statement, and the difference is that you should provide two brackets instead of one. Format is:

for((initialization;condition;increment/decrement))
do
Statements
Done

Example:

```
#!/bin/bash
for ((i=1;i<=10;i++))
do
        echo -n "$i  "
done
```

We can also process a series of numbers (range of values) and put it in curly braces {}. It's important when specifying a range like this that there are no spaces present between the curly brackets { }. When specifying a range you may specify any number you like for both the starting value and ending value and separated by two dots (..). The first value may also be larger than the second in which case it will count down.

for var in {start-value..end-value}

Example:

```
#!/bin/bash
for value in {1..10}
do
    echo -n "$value"
done
```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

It is also possible to specify a value to increase or decrease by each time. You do this by adding another two dots (..) and the value to step by.

for var in {start-value..end-value..step}

Example:

```
#!/bin/bash
for value in {10..0..2}
do
        echo -n "$value  "
done
```

Output:

10 8 6 4 2 0

Loop Control Statements

There are two statements that are used to control shell loops"

- break
- continue

1. The break Statement

The break statement is used to terminate the execution of the entire loop, after completing the execution of all of the lines of code up to the break statement. It then steps down to the code following the end of the loop. General format is:

break

If you are using nested loops,(i.e. one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Example:

```
#!/bin/bash
num=1
while [ $num -le 10 ]
do
```

```
        echo "Number is $num"
        if [ $num -eq 5 ]
        then
                echo "Exit from loop using break"
                break
        fi
        num=`expr $num + 1`
done
echo "Loop is complete"
```

Output:

```
Number is 1
Number is 2
Number is 3
Number is 4
Number is 5
Exit from loop using break
"Loop is complete"
```

We can also use *break* command in scripts with multiple loops. If we want to exit out of current working loop whether inner or outer loop, we simply use break but if we are in inner loop & want to exit out of outer loop, we use break with following format.

break n

Here **n** specifies the n^{th} enclosing loop to exit from.

Example:

```
#!/bin/bash
for (( i = 1; i <= 5; i++ ))
do
        echo "Outer loop: $i"
        for (( j = 1; j < 10; j++ ))
        do
```

```
            if [ $j -eq 5 ]
             then
            echo "Exit form nested loops"
            break 2
            fi
            echo "Inner loop: $j"
        done
    done
```

Output:

```
Outer loop: 1
Inner loop: 1
Inner loop: 2
Inner loop: 3
Inner loop: 4
Exit form nested loops
```

2. The continue statement

The *continue* statement is similar to the *break* command, except that it causes the current iteration of the loop to exit, rather than the entire loop. This statement is useful when an error has occurred but you want to try to execute the next iteration of the loop. General format is:

continue

Like with the break statement, an integer argument can be given to the continue command to skip commands from nested loops.

continue n

Here n specifies thenth enclosing loop to continue from.

Example:

```
#!/bin/bash
for i in {1..5}
do
```

```
        if [ $i -eq 3 ]
        then
                echo "Skipping number 3"
                continue
        fi
        echo "Number is $i"
done
```

Output:

```
Number is 1
Number is 2
Skipping number 3
Number is 4
Number is 5
```

Infinite Loops

All the loops have a limited life and they come out once the condition is false or true depending on the loop. A loop may continue forever due to required condition is not met. A loop that executes forever without terminating executes an infinite number of times. For this reason, such loops are called infinite loops.

Normally the loops are executed based on a condition. You can use special command with loops to set an infinite loop or an endless loop. An infinite loop occurs when the condition will never be met, due to some inherent characteristic of the loop. Only *break* or *exit* commands used within the body of the loop will exit the infinitely.

To set an infinite while loop use:

- *true* command
- *false* command
- : command

Syntax:

while true	***until false***	*'while :*	
do	***do***		
Statements	***Statements***	***Statements***	
done	***done***	***done***	

Examples:

Using: command:

```
#!/bin/bash
while :
do
        echo "Do something; hit [Ctrl+C] to stop!"
done
```

Using the *true* command:

```
#!/bin/bash
while true
do
        echo "Do something; hit [Ctrl+C] to stop!"
done
```

Using the *false* command:

```
#!/bin/bash
until false
do
        echo "Do something; hit [Ctrl+C] to stop!"
done
```

Passing Data through Environment Variables

Shell programs can access environment variables. We can pass data to a program by using environment variables. We can change the value of an environment variable in a program, but that change gets lost as soon as the program terminates. For example, we have an environment variable COLOR having value red and then change it to green. After execution of the program, when we check the value of the COLOR variable, it is still red.

Example:

```
#!/bin/bash
echo "The current COLOR variable is $COLOR"
COLOR = green
echo "The new COLOR variable is $COLOR"
```

Before executing this program, you need to set and export the COLOR variable using *export* command. You can verify the exported value of the variable by using the echo command. After you execute the program, it changes the value and prints the new value green. When the program finishes, you use the echo command again to check the variable value, and you find out that it is the same as it was before executing the program.

```
$ COLOR = red
$ export COLOR
$ echo $COLOR
red
```

Parameter Passing and Arguments (Command Line Arguments)

We can pass information to a shell program using command line arguments, the same as any other Linux command. These command line arguments can be used within the shell program. They are stored in variables, accessible within the program.

Arguments are passed from the command line into a shell program. The command line arguments are stored in variables that show the position of the argument in the command line. That is why these are also called positional parameters. The variables that store command line arguments have names from $1 to $9. Beginning with the tenth command line argument, the argument number is enclosed in braces.

$0 represents the name of the command (ie, shell script name). The first argument is read by the shell into the parameter $1, the second argument into $2, and so on. After $9, the arguments must be enclosed in brackets, for example, ${10}, ${11}, ${12}.

Example:

```
#!/bin/bash
echo "first parameter is $1"
echo "Second parameter is $2"
echo "Third parameter is $3"
```

Call this script with at least 3 parameters, for example: *./filename 4 5 6*

Exit Status

By default in Linux if a particular command or script is executed it returns two types of values which is used to show whether the command or script is successful or failure. This value is called exit status. If the value returned is zero, command or script is successful. If the value returned is nonzero, command or script is failure. To print that value use **$?** with the echo command. 100

Special Shell Variables

There are special shell variables which are set internally by the shell and which are available to the user.

Table below shows the list of special shell variables.

Variable	Description
$0	The filename of the current script.
$1-9	Parameters by position entered.
$$	The process ID of the current shell. For shell scripts, this is the process ID under which they are executing.
$#	The total number of arguments supplied to a script.
$@	All the arguments are individually double quoted. List all parameters used. For example '1 2' 3 becomes "1 2" "3". If a script receives three arguments, $@ is equivalent to $1 $2 $3.
$*	All the arguments are double quoted. List all parameters used. For example '1 2'3 becomes "1 2 3". If a script receives three arguments, $* is equivalent to $1 $2 $3.
$?	The exit status of the last command executed.
$!	The process ID of the last background command.
$_	The last argument of the previous command.

Example 1:

```
#!/bin/bash
echo "Process ID of shell = $$"
echo "Program name = $0"
echo "Number of args = $#"
echo "Argument 1 = $1"
echo "Argument 2 = $2"
echo "Complete list of arguments = $*"
echo "Complete list of arguments = $@"
```

Output:

```
$./example1.sh arg1 arg2 arg3
Process ID of shell = 48701
Program name = example1.sh
Number of args = 3
Argument 1 = arg1
Argument 2 = arg2
Complete list of arguments = arg1 arg2 arg3
Complete list of arguments = arg1 arg2 arg3

$ echo $?
0
```

Example 2:

```
#!/bin/bash
echo "Arguments are:"
for val in $*
do
    echo –n "$val  "
done
```

Output:

```
$./example2.sh 10 20 30 40 50
Arguments are:
10 20 30 40 50
```

Arrays

An array is a systematic arrangement of the same type of data. But in shell script array is a variable which contains multiple values may be of same type or different type. Arrays are zero-based: the first element is indexed with the number 0. Bash does not support multidimensional arrays.

Declaration of an array

We can declare an array in a shell script in three different ways.

1. **Indirect Declaration**

In indirect declaration, assign a value in a particular index of array variable. No need to first declare it.

Arrayname[index]=value

2. **Explicit Declaration**

In explicit declaration, declare array using declare statement and then assign the values.

declare –a Arrayname

3. **Compound Assignment**

In compound Assignment, declare array with set of values.

Arrayname=(value1 value2 … valueN)

Accessing array elements

To access array elements use curly brackets {}. To access an element at particular index, specify the array name with index in the following form.

${arrayname[index]}

To print all elements of an array use @ or * instead of the specific index number.

${arrayname[@]}

or

${arrayname[*]}

We can get the length of an array using the special parameter called $#.

${#arrayname[@]}

Example 1:

```
#!/bin/bash
echo "Creating an array to store numbers"
declare –a ar
ar[0]=10
ar[1]=20
ar[2]=30
ar[3]=40
ar[4]=50
echo "Total number elements in the array is ${#ar[@]}"
echo "Array elements are:"
echo ${ar[@]}
```

Output:

```
Total number of elements in the array is 5
Array elements are:
10 20 30 40 50
```

To traverse all array elements one by one, use any looping statement as follows.

```
for i in ${ar[@]}
do
        echo $i
done
```

Example 2:

```
#!/bin/bash
echo "Creating an array:"
name=(Linux Unix Windows)
echo "Size of the array is ${#name[@]}"
echo "Array elements are:"
echo ${name[*]}
```

Output:

Size of the array is 3
Array elements are:
Linux Unix Windows

Shell programs for automating system tasks

Sample Shell script to back up a file or folders is given below:

```
#!/bin/bash
BACKUPTIME=‘date +%b-%d-%y‘                    #get the current date
DESTINATION=”/home/usr/backup -$BACKUPTIME”
                                               #create a backup file using
                                               the current date in its name
SOURCE=”/home/usr/sourcedata”                  #the folder that contains the
                                               files that we want to backup
echo “Backing up files from $SOURCE to $DESTINATION
tar -cpzf $DESTINATION $SOURCE                 #create the backup
ls –lh $DESTINATION                            #Long listing of files in
                                               $DESTINATION to check
                                               file sizes
echo “Back up Finished”
```

The Linux ‘**tar**’ stands for *tape archive*, which is used by large number of Linux system administrators to deal with tape drives backup. It is used to create Archive and extract the Archive files. The tar is most widely used command to create compressed archive files and that can be moved easily from one disk to another disk or machine to machine.

REVIEW QUESTIONS

Part A

2. What is shell script?
3. What is bah shell?
4. What are the variables in shell script?
5. What is the use of echo?
6. What are environment variables?
7. How to create a variable in shell script?
8. What are the keywords used in shell script?
9. Give the syntax of case statement.
10. Write the commands to display first command line arguments and total number of command line arguments.
11. What is break statement?
12. What is the use of continue statement?
13. What do you mean by system shell variables?

Part B

14. Explain different types of variables in shell script.
15. Write a note on operators in shell script.
16. Explain different conditional statements used in shell script.
17. Explain looping statements in shell script.
18. Write note on parameter passing.
19. WAP to check whether the given number is prime or not?
20. Explain the file relational operators used in shell script.
21. Explain the file test operators used in shell script.
22. What are the different string operators?
23. Write note on arrays.

Part C

24. What is a shell in Linux? What are the different shells available in Linux?
25. Explain the operators available in shell script?
26. Explain different decision control and looping statements in shell script with example.